Qualitätssicherungskonzept

Simon Bordewisch & Rene Brückner

28. April 2015

Inhaltsverzeichnis

1 Allgemeines

2 Anforderungen an das Projekt

	Sehr gut	Gut	Normal	Nicht relevant
Produktqualität	X			
Funktion	X			
Zuverlässigkeit		X		
Benutzbarkeit	X			
Effizienz			X	
Änderbarkeit			X	
Übertragbarkeit		X		

Tabelle 1: Übersicht Quallitätsanforderungen

Um den Quallitätsanforderungen zu entsprechen hat sich das Team auf ein einheitliches Dokumentations-/Testkonzept geeinigt und im folgendem beschrieben.

2.1 Organisatorische Festlegungen

Um sich im Team untereinander abstimmen zu können wird sich einmal wöchentlich zu einem Scrummeeting in der Universität zusammengefunden. Außerhalb der Vorlesungszeit werden nach Absprache Meetings über VOIP diensten wie Mumble Tox oder TeamSpeak gehalten.
Die erfahrungen des Teams zeigen das auch weiterhin diese Festlegungen beibehalten werden sollten.

3 Dokumentationskonzept

3.1 Ansprüche an den Code

Einzurückende Quelltextfragmente werden mit Tabs eingerückt. In einer Zeile stehen maximal 79 Zeichen Code, zu lange Zeilen werben mit einem Backslash umgebrochen. Methodendefinitionen in einer Klasse haben stets eine Zeile Abstand zur Klassendefinition. Toplevelfunktionen und Klassendefinitionen werden mit zwei freien Zeilen getrennt. Zur Quelltexterstellung wird ASCII und UTF-8 verwendet.

Module und Packages werden am Anfang des Quellcodes in getrennten Zeilen importiert. In Berechnungen werden Priotitäten und Gruppen mit Leerzeichen gekennzeichnet. Bei der Deklaration von Startwerten oder Standardparametern werden keine Leerzeichen um das Gleichheitszeichen gesetzt.

Desweiteren ist es wichtig sich auf Namenskonventionen zu einigen. Als Sprache wurde sich in der Gruppe auf Englisch geeinigt. Es wird kein kleingeschriebenes L, großes o oder großes i einzeln als Variablenname verwendet. Modulnamen werden klein geschrieben und optional werden Wörter mit Unterstrichen getrennt. Klassennamen werden am Anfang der Wörter die sie enthalten groß geschrieben. Exceptions werden wie Klassennamen behandelt. Funktionsnamen werden kleingeschrieben und die enthaltenen Wörter mit Unterstrichen getrennt. Konstanten sind in Großbuchstaben mit Unterstrichen zu benennen. Kommentare beginnen mit einer Raute, Blockkommentare in jeder Zeile mit einer Raute. Paragraphen in Blockkommentaren werden auch mit Rauten gekennzeichnet. Kommentare in Zeilen werden mit mindestens zwei Leerzeichen separiert.

3.2 Quelltextnahe Dokumentation

Nach Absprache werden die Kommentare, genau wie der Quellcode in Englisch verfasst. Kommentare in Python beginnen mit einem Doppelkreuz-Zeichen: # ,und gelten bis zum Zeilenende. Ein Kommentar kann dabei am Zeilenanfang, nach Code oder nach einem Leerzeichen auftreten, aber nicht in einem String-Literal.
Mehrzeilige Kommentare werden durch drei Anfuehrungszeichen: """,begonnen sowie be-

endet und heißen Docstrings (Documentation Strings).

Beispiele:
# Das ist ein Kommentar.
CODE = 1 # Das ist auch ein Kommentar.
# ...das hier ebenfalls.
STRING = "# Das ist kein Kommentar."
"""Das ist ein Kommentar
ueber mehrere Zeilen"""

3.3 Externe Dokumentation

Bei der Entwicklung der Webanwendungen wollen wir eine Oberflaeche gestalten, die intuitiv bedient werden soll, damit jeder Benutzer die Seite ohne Handbuch nutzen kann. Dennoch wird von dem Team eine Seite erstellt die Funktionen und Nutzungsweise erklaert, sowie haeufige Fragen (FAQ) enthaelt.

3.4 Dokumentation der Aenderungen

Wenn im Repository Dateien geaendert werden, muss auch das dokumentiert werden, um spaeter den Überblick zu behalten. Im Detail bedeutet das, dass jeder Commit einen Kommentar aufweist, der beschreibt was geaendert wurde und von wem.

4 Testkonzept

Für die Bearbeitung des Hauptprojekt wird das bereits vorhandene Testkonzept bestehen bleiben. Zusätzlich wird für jede Woche ein Zweierteam für die Entwicklung der Tests abgestellt. Diese Entwickeln die Test für die Woche entsprechend abzugebenen Komponenten. Durch die parallele Entwicklung der Test mit den Komponenten wird sichergestellt, dass die definierten Anforderungen an die Komponenten wirklich erfüllt sind, da beide unabhängig voneinander entwickelt werden.

4.1 Komponententest

In der Entwicklung der Komponententests, welche hauptsächlich das Testen der einzelnen Module beinhaltet, wird auf drei unterschiedliche Werkzeuge zurückgegriffen: `Doctest`, `Unittest` und ergänzend hierzu `Coverage`. Diese werden im folgenden genauer erläutert.

4.1.1 Doctest

Doctest stellen in Python eine einfache Funktion da, um Funktionen zu verifizieren. Sie werden innerhalb der Docstring einer Funktion implementiert:

```
"""
                                        ...
>>>FunktionsName(Variable)
erwarteterWert
                                        ...
"""
```

Die Syntax von Doctest orientiert sich dabei an einen Funktionsaufruf in einer Kommandozeile. In der ersten Zeile steht der Funktionsaufruf, während die nächste Zeile den Rückgabewert enthält. Dies erlaubt das theoretische Kopieren der ausgeführten Funktion aus der Kommandozeile in die Docstrings.
Hiermit wird vorallem die Zeitumwandlungen für die Semester getestet. Diese Funktionen werden mit Doctests getestet, da Unittest zu komplex ist, um für diese Tests effektiv zu sein.

4.1.2 Unittest

Auch das Werkzeug `Unittest` dient zur Entwicklung von Unittests. Es bietet mehr Funktionalitäten als `doctest` und wird in eigenen Modulen entwickelt. Dadurch ist komplexer als doctest und für viele unserer Funktionen nicht geeignet. Dennon wird Unittest zum Testen komplexerer Funktionen, beispielsweise die Umwandlung der Daten in den .ical-Format, benötigt, da hierfür die Funktionalitäten von doctest nicht ausreichen. Wichtig für Unittest ist `assert`. `assert` ist ein Keyword für viele verscheidene Funktionen, welche

für Tests gebraucht werden. Weitere Erklärungen der Funktionsweise sind sehr umfassend, deshalb an dieser Stelle der Verweis zur Dokumentation von Unittest[1] .

4.1.3 Coverage

`Coverage` ist eine Python-Erweiterung, mit welcher man die Abdeckung des Codes mit Doctests und Unittests überprüfen kann. Um es aufzurufen, muss lediglich `coverage run my_program.py arg1 arg2` in die Konsole eingeben werden. `Coverage` wird daraufhin das Programm starten und die Test evaluieren. Mit `coverage report` bekommt man einen Überblick über die getesteten/erreichten Programmteile, und welche Teile des Programmes nicht erreicht/getestet wurden.

4.2 Webtests

Neben den Systemtests, die primär die einzelnen Module testen sollen, wird auch die Interaktion der Module miteinander getestet. Da dies hauptsächlich im Rahmen der `app.py` geschieht, wird dies Mithilfe der Webtests getestet. Hierzu verwenden wir `Selenium`.

Diese Testumgebung bietet die Möglichkeit Funktionalitäten einer Webseite mit verschiedenen Browsern zu testen. Zudem bietet es die Möglichkeit Mithilfe eines Plug-Ins[2] die IDE direkt in Firefox zu nutzen und so Tests in Form von Makros zu schreiben. Selenium führt nach dem Schreiben des Programms den gewählten Browser automatisch aus und bedient die Webseite so, wie es im Programm hinterlegt wurde. Dadurch können verschiedene Navigationsiterationen innerhalb der Webseite schnell und automatisch getestet werden.

Für das Testen ist es lediglich nötig, die Webseite erreichbar zu machen, entweder über den Projekt-Webserver selber oder aber auch lokal. Je nach Konfiguration muss die entsprechende URL in den Tests hinterlegt werden. Die Testumgebung selber ist in Java geschrieben, die Bibliotheken für Selenium stehen aber auch für Python zur Verfügung.

[1] `https://docs.python.org/3/library/unittest.html`
[2] siehe SeleniumIDE unter `http://docs.seleniumhq.org/download/` (24.04.2015)

4.2.1 Verwendung

Hier soll ein wenig auf die Funktionen von Selenium eingegangen werden. Da dieses Werkzeug jedoch relativ komplex ist, ist die Einarbeitung in die dazugehörige Dokumentation[3] empfohlen.

Der Quellcode einer HTML-Webseite ist in verschiede Bereiche aufgeteilt. Selenium navigiert auf diesen, wie folgender Quellcode-Ausschnitt aus unser Webseite verdeutlichen soll.

Im obigen Bild sieht man die Studiengangauswahl der Webseite. Im Quellcode der Webseite ist die Liste ungefähr so aufgebaut:

```
<select id="course" name="course">
        <option value="val1">Bachelor Informatik</option>
        <option value="val2">Bachelor Wirtschaftsinformatik</option>
</select>
```

In Python können all diese Option nun wie folgt getestet werden:

```
for option in \
driver.find_element_by_name("course").find_elements_by_tag_name('option'):
```

Die Schleife geht jedes Element in dem Menü durch und führt den im Programm definierten Code aus.

4.2.2 XPath

Eine weitere wichtige Information auf Webseiten ist der sogenannte `XPath`. Es ist eine weitere Art um Elemente auf Webseiten zu adressieren. Dies wird hauptsächlich zur Betätigung der Buttons auf der Webseite benötigt. Das folgende Beispiel zeigt eine solche Betätigung. Der in Anführungszeichen geschriebenen Parameter ist hierbei der XPath des Buttons.

[3] siehe `http://docs.seleniumhq.org/docs/` (24.04.2015)

```
driver.find_element_by_xpath("//*[@id='main']/form/input").click()
```

Diese Funktionen ermöglichen es, die Webtests relativ komfortabel zu generieren und autmatische Tests durchzuführen.

4.3 Nicht automatische Tests

In unserem Projekt sind wir stark von der ODB der Fakultät für Mathematik und Informatik abhängig, da wir von hier die Daten für die Stundenplangenerierung beziehen.

Aufgrund der Inkonsistenz der Datensätze der Datenbank, da sich zum Beispiel Vorlesungzeiten ändern oder Übungen wegfallen, müsste für einen Test mit konsistenten Testergebnissen die Datenbank nachgebildet und für die Tests eingebunden werden. Da dies den Rahmen des Projektes sprengen würde, wird für solche Funktionen auf automatische Tests verzichtet.

Innerhalb der Programmierarbeiten dieser Funktionalitäten ist es somit umso wichtiger, dass auf die Richtigkeit der Funktionen geachtet wird. Daher wird jeder Programmier angehalten, die Funktionen für die SPARQL-Anfragen mit verschiedenen Datensätzen zu testen. Zudem muss die Funktionalität innerhalb der Webtests ebenfalls geprüft werden, z.B. in dem geprüft wird, ob die (mit Hilfe der Datenbank) generierten Listen nicht leer sind.

Zudem müssen diese Daten von Menschenhand validiert werden.

4.4 Abnahmetest

Der Abnahmetest ist der finale Test. Bei diesem wird gemeinsam mit dem Kunden die Anwendung von außen getestet. Interner sind hier vollkommen egal, lediglich die Funktionalität zählt. Ziel ist die Bestätigung der Anforderungserfüllung vom Kunden, beipspielsweise von der Webseite. Nach erfolgreichem Abnahmetest endet die Entwicklung(siteration) und das Produkt kann vom Kunden verwendet werden.