Qualitätssicherungs-konzept

Projektnummer: swp10-7
Projekttitel: jQuery Plugin – Erweiterte Autovervollständigung auf der Basis von SPARQL Endpunkten

Abgabe: 28.06.2010

Verantwortlicher für Dokumentation und Qualitätssicherung: Marina Mitjagin
Projektleiter: Marcus Nitzschke

1 Dokumentationskonzept

1.1 Styleguide/Konventionen

Nachfolgend werden die wichtigsten Punkte für einen sauberen und gut lesbaren Quellcode aufgeführt:

- Die Standardeinrücktiefe beträgt 4 Leerzeichen
- Nach einem Komma, Semikolon, Operator oder Schlüsselwort (z.Bsp. „if“) folgt ein Leerzeichen
- Sowohl Kommentare, als auch Bezeichner werden auf Englisch verfasst
- Bezeichner tragen „sprechende“ Namen; Ausnahmen bilden kurze, überschaubare Laufvariablen
  - „linesOfCode“ statt „loc“
  - aber: „for (int i = 0 ; i < 10, i++){ … }“ zulässig
- Öffnende und schließende Block-Klammern stehen jeweils auf einer extra Zeile

```
function foobar()
{
        …
}
```

- Zur Übersicht werden auch einzeilige Anweisungen in einem Block verfasst

```
- if (a == b)
{
        //do something
}
```

- weitere fortlaufende Schlüsselwörter (z.Bsp.: „else“) beginnen auf neuen Zeilen
- globale Variablen werden komplett groß geschrieben
- Klassen beginnen mit einem Großbuchstaben
- lokale Variablen und Funktionen beginnen mit einem kleinen Buchstaben
- Bei zusammengesetzten Bezeichnern (sowohl Klassen, als auch Variablen und Funktionen) beginnen die folgenden Wörter immer mit einem Großbuchstaben
  - z.Bsp.: FooBarKlasse
  - aber: function fooBar()
- Zur Unterteilung des Quellcodes und damit zur Verbesserung der Lesbarkeit ist an den richtigen Stellen von Leerzeilen Gebrauch zu machen

1.2 Quellcode-interne Dokumentation

Damit es für außen stehende Entwickler einfacher ist sich in das Projekt zu finden und eventuell etwas dazu beizutragen, ist es wichtig, dass der Quellcode des Projektes sorgfältig dokumentiert wird. Dies sollte direkt beim Schreiben der jeweiligen Funktionalität passieren, da im Nachhinein oft wichtige Details verloren gehen und nicht den Weg in die Kommentare finden.
Es gilt also jede Klasse, jede Funktion und jede wichtige Variable (Attribute) näher zu beschreiben. Dabei steht das „Was“ (…macht die Funktionalität) im Vordergrund und nicht das „Wie“ (…funktioniert sie). Bei Funktionen ist darauf zu achten die Übergabeparameter und die Rückgabewerte zu dokumentieren.

Um die intern formulierten Kommentare und Erklärungen extrahieren zu können, kommt das Dokumentationstool JSDoc zum Einsatz. Es gleicht in seiner Benutzung dem Java-bekannten javadoc. Die Kommentare, die durch das Tool erkannt werden beginnen mit „/**“ und enden mit „*/“. Für eine Referenz der möglichen Schlüsselwörter und für weitergehende Informationen steht die Webseite zur Verfügung.

1.2.1 SVN

Neben der Quellcode-internen Dokumentation ist auch jeder Programmierer angehalten seine Commits kurz aber präzise zu Beschreiben. So lassen sich schneller entsprechende Stände einer Datei wiederfinden und Änderungen verfolgen.

1.2.2 Team-interne Dokumentation

Damit die Erfahrungen und Kenntnisse der einzelnen Teammitglieder schnell den Rest der Gruppe erreichen, wird Team-intern ein Wiki gepflegt. Hier fließen regelmäßig neue Hinweise oder Bemerkungen ein, die für den Rest der Gruppe von Relevanz sein könnten. Jedes Teammitglied ist dazu angehalten diesem Prozess des Veröffentlichens nachzugehen.

1. 3 Externe Dokumentation

Die externe Dokumentation soll es dem Endbenutzer ermöglichen
a) einen Einblick in die Software und deren Funktionalitäten zu bekommen und
b) bei Problemen oder Fragen eine erste Anlaufstelle zu haben.
Um dies zu ermöglichen wird es im Laufe des Softwareprojektes zur Entstehung eines Benutzerhandbuchs kommen.

2 Organisatorische Festlegungen

Für die Dokumentation des Quellcodes ist jeder Programmierer selbst verantwortlich. Dies wird vom Verantwortlichen für Dokumentation und Qualitätssicherung , sowie vom Implementierer kontrolliert und somit gegebenenfalls der Programmierer kontaktiert, um Defizite zu beseitigen.
Die Erstellung des Benutzerhandbuches wird durch den Verantwortliche für Dokumentation und Qualitätssicherung zu gewährleisten.

Die Einhaltung der gestellten Termine sind stets durch den Projektleiter und den Verantwortlichen der jeweiligen Phase sicher zu stellen. Hierzu werden vor dem eigentlichen Abgabetermin interne Abgabetermine vereinbart.

Da auch bei einem solch umfassenden Softwareprojekt Fehler oder fehlerhafte Codesegmente auftreten können und werden, obliegt es dem Verantwortlichen für Test, diese durch geeignete Möglichkeiten festzustellen und deren Korrektur zu veranlassen.

3. Testkonzept

Da bei der Entwicklung von Software Projekten mit zunehmenden Umfang das Finden und Beheben von Fehlern komplizierter wird, ist frühzeitiges ausfindig machen solcher durch rechtzeitige Tests angebracht. Dafür ist im Rahmen des Extreme Programmings ein Testzyklus vorgesehen, der aus 3 Teilen besteht:

1. Komponententests
   In dieser Testphase werden die einzelnen Methoden und Klassen auf ihre Funktionalität und auf Fehler geprüft. Dazu ist von den jeweiligen (soweit vorhandenen) Programmierpaaren passende Testreihen zu entwickeln und anschließend eine Dokumentation der Tests zu erstellen. Für die Tests ist die Nutzung von QUnit vorgesehen – eine einfach benutzbare Testsuite die für jQuery im speziellen aber auch für JavaScript an sich geeignet ist. Der Aspekt „Tests first" des Extreme Programmings sieht vor, dass noch vor der Implementierung der Klassen und Methoden geeignete Tests zu schreiben sind. Diese Testklassen sollten demnach Test+Klassenname benannt werden.
   So wie die Klassen zu Paketen zusammen gesetzt werden, werden die entsprechenden Testklassen zu Testsuites zusammengeführt und Test+Paketname benannt.

2. Integrationstest
   In dieser Testphase wird das Zusammenspiel der einzeln entwickelten Klassen getestet. Somit ist es sinnvoll, vorher Zusammenhänge zwischen Klassen zu erkennen und das Zusammenwirken dieser dann zu testen. Dazu werden Klassen nach Geschäftsprozessfunktionalität integriert, getestet und erweitert.

3. Systemtest
   Nach erfolgreichem Integrationstest sind die Stories zu einer Zwischenversion des Produktes zusammenzuführen und einem Systemtest zu unterziehen. Dabei wird aus der Sicht des Anwenders geprüft, ob die Anforderungen an das Produkt erfolgreich umgesetzt wurden. Des weiteren sollten die einzelnen Gruppenmitglieder / Programmiergruppen die Zwischenversion eigenen Tests unterziehen, um so ein möglichst breites Spektrum an Tests durchzuführen.
   Bei Fehlern ist das Finden des Problems zu organisieren und die Behebung durch die Programmierer der beteiligten Klassen durchzuführen.

3.1 Dokumentation der Tests

Grundsätzlich sollte die Art der Tests festgehalten werden. Beim Auftreten von Fehlern sind diese, die verwendete Produktversion, die Quelle und ihre Lösung – wenn dann gefunden - festzuhalten. Auf diese Weise kann wiederholtes Auftreten des Fehlers einfacher behoben werden. Diese Dokumentation wird dabei von den jeweiligen Programmieren durchzuführen.

3.2 QUnit

Um QUnit verwenden zu können, muss man jQuery, qunit.js und qunit.css einbinden und eine einfache HTML-Struktur zur Anzeige der Testergebnisse bereit stellen.

Einbinden von Qunit.css:
```
<link rel="stylesheet"
href="http://github.com/jquery/qunit/raw/master/qunit/qunit.css"
type="text/css" media="screen" />
```

Einbinden von qunit.js:
```
<script type="text/javascript"
src="http://github.com/jquery/qunit/raw/master/qunit/qunit.js"></script>
```

Um die Testergebnisse anzuzeigen, muss nun im Body-Tag der Seite folgende Zeile eingefügt werden:
```
<ol id="qunit-tests"></ol>
```

Die Ausgabe wird dann wiefolgt aussehen:
Name des Tests (Anzahl fehlgeschlagenener Tests, Anzahl erfolgreicher, Anzahl gesamt)

Ein Test sieht so aus:
```
test(name, expected, test)
Bsp:
test("Das ist ein Test", function () {
      expect(1);
      ok(true, "Dieser Test war erfolgreich");
});
```

Die entsprechende Ausgabe: `Das ist ein Test (0,1,1)`

Da in einem test()-Aufruf mehrere Tests durchgeführt werden können, wird nachfolgend auf der HTML Seite aufgeführt, welche Tests erfolgreich waren und welche nicht.

`expected()` ist optional und gibt an, wie viele Tests, sogenannte Assertions, zu erwarten sind.
Es gibt 3 Assertions:

- `ok(Status, Nachricht)` überprüft, ob das erste Argument wahr ist.
- `equals(zu überprüfender Wert, erwarteter Wert, Nachricht)` funktioniert wie `ok()`, gibt jedoch beide Werte aus und ist für nicht-boolean Werte gedacht.
- `same(zu überprüfendes Objekt, erwartetes Objekt, Nachricht)` funktioniert wie `equals()`, ist aber strikter, benutzt === und ist deswegen zum Vergleich von Objekten geeignet.

Des weiteren besteht noch die Möglichkeit, die Tests in Module zu unterteilen. Dazu muss lediglich

```
module("Name", [lifecycle]);
```

vor den Tests, die diesem Modul zugeteilt werden sollen, aufgerufen werden. Allen Tests von diesem Modul wird nun der Name des Moduls vorangestellt.

Es können in lifecycle setup- und teardown-Callback-Methoden initialisiert werden, die vor bzw. nach jedem Test in diesem Modul ausgeführt werden. So können zum Beispiel Testdaten für jeden Test erstellt werden.

Bsp:

```
module("Modul abc", {
	setup: function () {
		this.user = new User("TestUser");
	},
	teardown: function () {
		this.user = null;
	}
});
```