Universität Leipzig
Softwaretechnik-Praktikum

Qualitätssicherungskonzept

DTP-16: LD Hypstrr - Datadriven Template Publication

31. Januar 2016

Inhaltsverzeichnis

1 Branching-Modell

Wir verwenden das Branching-Modell *git-flow*:
`http://nvie.com/posts/a-successful-git-branching-model/` . Dieses Branching-Modell ist ein üblich verwendetes Modell zur Balancierung zwischen Code-Fortschritt und Code-Stabilität.

2 Code-Reviews

Code-Reviews nützen einer hohen Qualität des Codes und technisches Wissen kann weitergegeben werden. Sie erweitern die Kritikfähigkeiten auf Seiten der Kritiker und der Kritisierten. Jeder Pull-Request wird durch eine andere Person begutachtet, gegebenenfalls kritisiert und durch den Autor verbessert und durch den Gutachter freigegeben. Dann und nur dann darf der Code in den develop-Branch gemergt werden.

3 Dokumentationskonzept

Das Dokumentationskonzept ist eine wichtige Grundlage für die erfolgreiche und strukturierte Arbeit an einem Software-Projekt. Das Konzept ermöglicht ein Arbeiten nach klaren Vorgaben. Eine verständliche Software-Dokumentation erlaubt die langfristige Weiterentwicklung und Nutzung der Software. Entwickler, die neu in das Projekt einsteigen, können sich schnell einarbeiten und den Code mittels der Dokumentation gut nachvollziehen.

3.1 Coding-Standard

3.1.1 Ruby

Wir verwenden den hier `https://github.com/DTP16/ruby-style-guide` formulierten Coding-Standard. Dies ist ein Fork von diesem Coding-Standard: `https://github.com/bbatsov/ruby-style-guide`, der einige gute und einige fragwürdige Empfehlungen abgibt. Ergeben sich im Laufe des agilen Programmierens Änderungswünsche an unserem Coding-Standard, so werden wir diesen entsprechend anpassen.

3.1.2 Rubocop

Wir verwenden für die statische Code-Analyse *Rubocop* `https://github.com/bbatsov/rubocop`, der auf dem hier verwendeten Coding-Standard aufbaut. Rubocop stellt einen experimentellen Formatter zur Verfügung.

3.2 Sprache

Die Dokumentationssprache ist Englisch.

3.3 Quelltextdokumentation

Die Quelltextdokumentation ist hilfreich für alle Programmierer in einem Projekt, denn sie erlaubt das schnellere Einarbeiten in von anderen Entwicklern geschriebenem Code. Zur Quelltextdokumentation wird *RDoc* `http://rdoc.sourceforge.net/` verwendet.

3.4 Quelltextnahe Dokumentation

Für die quelltextnahe Dokumentation wird *AsciiDoc* `http://www.methods.co.nz/asciidoc/` verwendet. *AsciiDoc* ist vollständig spezifiziert und wird von *GitHub* unterstützt (`https://github.com/github/markup#markups`). (*MarkDown* hat zahlreiche untereinander inkompatible Dialekte und ist nicht vollständig spezifiziert.)
Die folgenden beiden Beispiele stammen von: `http://asciidoctor.org/docs/asciidoc-writers-guide/` .

3.4.1 Beispiel 1

Quellcode:

```
[source,ruby]
----
require 'asciidoctor' # <1>
puts Asciidoctor.convert_file('sample.adoc', :header_footer => true) # <2>
----
<1> Imports the library
<2> Reads, parses and renders the file
```

Rendering:

```ruby
require 'asciidoctor' (1)

puts Asciidoctor.convert_file('sample.adoc', :header_footer => tru
```

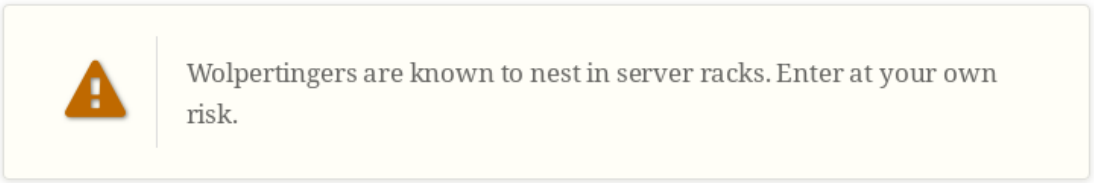
(1) Imports the library
(2) Reads, parses and renders the file

3.4.2 Beispiel 2

Quellcode:

```
WARNING: Wolpertingers are known to nest in server racks.
Enter at your own risk.
```

Rendering:



4 Testkonzept

Um eine zuverlässige Software zu entwickeln, welche in allen Hinsichten funktionieren soll, ist es notwendig, Tests durchzuführen. Dazu werden diese in Komponententests und Integrations bzw. Systemtests unterteilt. Diese kann man durch kontinuierliche Integration durchführen, bei der automatisierte Tests nach dem Kompilieren durchgeführt werden.
Ein bekanntes Tool hierfür stellt *Travis CI* dar. Jekyll geht in der eigenen Dokumentation auf diese Software ein und erklärt, wie man die Umgebung einrichtet: `http://jekyllrb.com/docs/continuous-integration/`. Aufgrund dessen und der hohen Flexibilität und guten Dokumentation (`https://docs.travis-ci.com/user/languages/ruby`) werden wir Travis CI nutzen. Die Komponententests – oder bei unserem Fall mit Ruby die sogannten Unittests – dienen zur Verifikation der Korrektheit von Modulen einer Software, zum Beispiel von einzelnen Klassen.
Nachdem der Komponententest erfolgreich war, wird der Integrationstest ausgeführt. In diesem wird das Zusammenspiel verschiedener voneinander abhängiger Komponenten in einem komplexen System getestet. Die Systemtests sollen die finale Erfüllung der Software entsprechend der Anforderungen überprüfen. Es gibt einen funktionalen und einen nicht-funktionalen Systemtest. Im funktionalen Systemtest wird die Qualität im Sinne von Korrektheit und Vollständigkeit geprüft. Bei dem nicht-funktionalen Systemtest geht es eher um die Sicherheit, die Zuverlässigkeit oder die Benutzbarkeit.
Für Ruby wäre Minitest eine geeignete Testumgebung. Wie der Name schon sagt, ist es ein kleines und schnelles testing framework. Die für uns wichtigsten Komponenten davon sind minitest/unit (kleine sehr schnelle Testumgebung) und minitest/spec (Integrationstest, lässt sich mit minitest/unit verbinden). Anmerkung: Es kann während der Implementierungs und Testphase zu Problemen kommen, die jetzt noch nicht vorhersehbar sind und demzufolge momentan noch nicht im Testkonzept berücksichtigt werden können.

5 Organisatorische Angelegenheiten

Die Gruppe trifft sich einmal wöchentlich (aktuell: mittwochs, 16 Uhr) zur Besprechung. Dort werden gemeinsam alle Aufgaben besprochen und verteilt. Des Weiteren können Fragen mit dem Betreuer und / oder dem Auftraggeber besprochen werden. Für die Kommunikation außerhalb der Treffen werden E-Mails und das Collaboration Tool Slack verwendet. Die Aufgabenverteilung und -überwachung erfolgt über das Issue-Tracking-System *JIRA/Atlassian*. Über *GitHub* wird die Software verwaltet.