Webanwendung zur Extraktion von Teildatensätzen aus DBpedia

Christian Ernst, Dominik Strohscheer, Hans Angermann
Till Nestler, Marvin Hofer, Robert Bielinski, Jonas Rebmann

Inhaltsverzeichnis

Qualitätssicherungskonzept

Dokumentationskonzept

Coding Standard

Um den Quelltext möglichst zugänglich für andere Entwickler zu halten und um sich somit auch mit etablierten Konventionen vertraut zu machen, werden wir uns in diesem Projekt an den Coding Standards von Drupal[1] orientieren[2] und, für Sprachspezifisches, am Ruby Styleguide[3] orientieren. Wichtige Richtlinien und Unterschiede sind:

- Sämtliche Dokumentation sowie Variablennamen und Kommentare sind in englischer Sprache
- Leerzeichen statt Tabs für die Einrückung (1 Tab = 2 Leerzeichen)
- Leerzeichen um Operatoren aber nicht neben dem logischen nicht `!`
- Leerzeichen um geschweifte Klammern `{ code }`, aber nicht um runde und eckige Klammern `["element1", "element2"]` `funktion(paramter)`
- Verzicht auf Kurzformen beziehungsweise Formen, bei denen die Sprache das Weglassen von sonst erzwungenen Zeichen erlaubt
  - Ausnahme: Wegfall von `return` und einzeilige Blöcke (z.B. `var.each { |temp| action }` statt `var.each do |temp| action end`)
  - Keine Ausnahme: `?:` statt `if then else end`
- Empfohlene maximale Zeilenlänge von 80 Zeichen
- `funktions_und_variablennamen`, `KlassenUndModule`, `KONSTANTEN_SNAKE`, `_globale_variablen_`
- Kein whitespace am Ende von Zeilen
- Newline am Ende von Dateien
- `when` wird auf die gleiche Ebene wie das zugehörige `case` eingerückt
- Benötigte Abhängigkeiten werden immer am Anfang jeder Datei mit `requires` eingebunden
- Gegenüber einer `for`-schleife ist ein Iterator (`.each`) zu bevorzugen
- In mehrzeiligen `if`-Bedingungen sollte kein `then` verwendet werden
- Anstatt der Schlüsselworte `and` und `or` sollten die entsprechenden Operatoren `&&` und `||` verwendet werden
- `else` sollte niemals in einer `unless`-Bedingung verwendet werden

- Die äußeren Klammern um `if`, `unless` und `while` Bedingungen sollten weggelassen werden.
- Zur Darstellung von Zeichenketten sollten doppelte Anführungszeichen verwendet werden. `"Literal"`

Für Kommentare verwenden wir die im folgenden Abschnitt festgelegten Konventionen.

Quelltextdokumentation (Kommentare)

Kommentare im und um den Quelltext sind ein weiteres Mittel, um den Quelltext anderen Entwicklern zugänglich zu machen und auch, um sich selbst später im eigenen Code wiederzufinden. Man unterscheidet hierbei meist zwischen quelltextinternen und quelltextnahen Kommentaren.

Wie oben erwähnt sollen alle Kommentare in englischer Sprache verfasst werden.

Quelltextinterne Kommentare

Dies sind die Kommentare, die meist innerhalb von Funktionen und Prozeduren stehen und dafür gedacht sind, einzelne Codeschnipsel (beziehungsweise deren Funktion) zu erläutern. Es ist nicht nötig, jede Zeile Code zu erklären, aber große Abschnitte und/oder komplexe Strukturen sollten erläutert werden. Im Allgemeinen sollte das Verständnis des Programmierers für die Komplexität und Verständlichkeit seines Codes ausreichen, um zu bestimmen, welche Codeabschnitte kommentiert werden sollten.

Interne Variablen, deren Funktion durch den Namen allein nicht ganz klar wird oder die anderweitig von näherer Beschreibung profitieren würden, sollten auf der Zeile der Definition kommentiert werden, bei Platzmangel in der Zeile darüber. Generell sollten Kommentare wie Überschriften über den Codeabschnitten stehen, die sie beschreiben, mit einer Leerzeile darüber und eventuellen Ergänzungen auf der jeweiligen Zeile (oder wie bei den Variablen bei Platzmangel darüber, aber ohne Leerzeile). Für größere Abschnitte oder anderweitige logische Abtrennungen, speziell in Klassen, kann eine auskommentierte Zeile aus Bindestrichen eingefügt weren, unter der an der rechten Seite ein Titel für den Abschnitt steht.

Quelltextnahe Kommentare

Dies sind die Kommentare, die zur Dokumentation der Methoden, globalen Variablen, und anderen Top-level Codeabschnitten dienen. Diese Kommentare sollen später mit einem Dokumentationstool (für uns sehr wahrscheinlich YARD[4]) extrahiert werden können, um eine externe Dokumentation zu erstellen. Sie sollen kurz, aber präzise, beschreiben, was der beschriebene Code tut. Bei Prozeduren und Funktionen sollen auch eventuelle Eingabeparameter und Ausgaben erläutert werden, was wie bei javadoc mit @tags getan werden kann.

Diese Kommentare sollen in einem kontinuierlichen (auch Leerzeilen auskommentiert), von einer Leerzeile umgebenen, Kommentarblock über dem zugehörigen Codeteil stehen. Zuerst soll die Beschreibung kommen, dann nach einer Leerzeile die eventuellen Parameter und/oder Rückgabewerte hinter @tags, wobei Parameter (@param) sowohl einen Namen und in eckigen Klammern den Typ annehmen, während Rückgabewerte (@return) nur einen `[Typ]` haben.

Beispielcode

```ruby
# test class
# @author Till Nestler

requires "stuff" # Some required procedures

# A class that does string operations and prints them to the output

class String_Ops

  # Prints a string horizontally with spaces and vertically
  #
  # @param input [String] string to operate on

  def vertical_horizontal(input)
    input.each_char { |c| print c, " " } # prints each character and a space
    print "\n"
```

```ruby
    # prints every character except the first below each other
    input[1..input.length].each_char { |c| puts c }
  end

#-------------------------------------------------------------------------------
                                                          # Advanced methods

  # An advanced string operation that does a lot of stuff with several
  # parameters whose description is so long it needs to be divided into several
  # lines due to the 80 character per line limit.
  #
  # @param input1 [String] The first string.
  # @param input2 [String] The second string, used for something.
  # @param some_integer [Integer] Useful for another thing.
  # @return [String] This method actually returns something.

  def advanced_op(input1, input2, some_integer)
    temp_var = 0 # some temporary variable
    stuff_to_return # What gets returned in the end

    # This happens in the code below
    if test && this || that # extra note
      "fancy operations"
    else
      "more stuff"
    end

    # More here
    [insert code here]

    stuff_to_return
  end
end
```

Allgemeine Dokumentation

Die allgemeine Dokumentation ist dazu gedacht, Nutzern des Endprodukts die Fähigkeiten, Funktionen und Benutzung des Produkts zu erklären und als Nachschlagewerk zu dienen. Sie sollte ausführlich, aber nicht zu technisch sein, da sie nicht dazu gedacht ist, die Funktionsweise des Codes zu erklären, sondern eher ein Benutzerhandbuch ist. In der Dokumentation finden sich daher auch exemplarische Durchführungen um die Funktionsweise zu veranschaulichen. Das Dokument sollte leicht zugänglich und gut durchsuchbar sein.

Auch diese Dokumentation soll auf Englisch geschrieben werden, im Gegensatz zu Quelltextkommentaren wäre aber eine Übersetzung in andere Sprachen möglich.

Testkonzept

Funktions- und Programmtests sind ein wichtiger Bestandteil in der Softwareentwicklung. Hierbei sollten bereits während der Entwicklungsphase Tests in einzelnen Abschnitten durchgeführt werden, um frühzeitige Fehler zu erkennen und zu beheben.

Komponententests (Unit-Tests)

Bei Klassen muss darauf geachtet werden, dass:

- der geforderte Wertebereich komplett verarbeitet werden kann
- der gewünschte Wertebereich der Ausgabe nicht verlassen wird
- die Bearbeitung von Anfragen nicht zu lange dauert

Hierfür liefert Ruby bereits von sich aus, mit Test::Unit[5] ein Framework, durch das Tests durgeführt werden können, die einzelne Abfragen an die Klasse auf Richtigkeit und Laufzeit testen.

Codeausschnitt

```
assert_match("Inception 2h 28m", MethodOne.new("Inception").add_length(), "Laenge anfuegen
    funktioniert nicht")
```

könnte z.B. eine Methode, die an den Filmtitel dessen Länge anhängen soll, darauf testen, ob dies auch wirklich geschieht. Für den Fall, dass der Ausgabestring dann die Länge nicht enthält, wird dann beim Bericht der spezifizierte String "Laenge anfuegen funktioniert nicht" ausgegeben, wodurch dann direkt bekannt ist, in welcher Funktion der Fehler liegt.

Diese Tests kann man auch gruppieren und somit z.B. nur Methoden einer bestimmten Klasse testen, wenn diese bearbeitet wird, anstatt das gesamte Programm bei jeder kleinen Änderung zu testen.

Manuelle Tests

Besonders in finalen Phasen des Projektes werden auch manuelle Tests häufiger notwendig. Hierzu sollten im besten Fall mehrere Tester genutzt werden, die Testen, ob die GUI ihren Zweck sinngemäß erfüllt und nicht Fehlerhaft ist. Insbesondere sollten dazu auch nach Möglichkeit verschiedene Betriebssysteme und Browser verwendet werden.

Organisatorische Festlegungen

Dokumentation

Alle Programmierer sollten ihren Code beim Schreiben oder möglichst zeitnah dazu kommentieren, um allen anderen zu ermöglichen, sich schnell einzufinden. Der Beauftragte für Qualitätssicherung und Dokumentation wird allen Code mindestens stichprobenartig lesen und auf Einhaltung des Coding- und der Dokumentationsstandards überprüfen. Falls Mängel gefunden werden, wird er je nach Schwere diese selbst beseitigen oder den Programmierer dies tun lassen. Die allgemeine Dokumentation sollte im Verlauf des Projekts langsam aufgebaut werden.

Tests

Die Unit-Tests werden vom Beauftragten für Tests geschrieben, sobald angefangen wird, an einer Methode zu arbeiten, damit der Testbauftragte den Überblick über die möglichen Folgefehler hat und direkt auf die entsprechenden Programmierer zugehen kann. Bei manuellen Tests sollten mindestens sowohl der Testbeauftragte, als auch der entsprechende Programmierer mitmachen, wobei hier mehr Tester besser sind, um möglichst jeden Fehler zu finden.

Gefundene Fehler werden sinnvoll und verständlich notiert und entsprechend an den für den fehlerbehafteten Teil zuständigen Programmierer weitergeleitet. In der Notiz zum Fehler sollte nach Möglichkeit aufgelistet sein:

- Klassen und Methodenname bzw. GUI-Element
- genutztes Setup (wenn es sich um manuelle Tests handelt)
- Fehlerbeschreibung
- Mögliche Folgefehler
- Verbesserungsvorschläge

Weekly-Scrum

Das Wetd16-Projektteam trifft sich immer Donnerstags von 15:15 bis 16:15 zur Besprechung des Projektstatus. Dabei sind nach Möglichkeit auch die Betreuer anwesend. Zu jedem Termin wird ein Protokoll mit den an- und abwesenden Personen und den besprochenen Themen erstellt. Am Ende jeder Sitzung werden Vorschläge für die Themen des nächsten Meetings gesammelt, welche im Git ebenfalls gespeichert sind und ergänzt werden können. Die Protokolle sind ebenfalls im TeamWiki immer auf dem aktuellsten Stand.

Erster Meilenstein

Der erste Meilenstein des Wetd-Projekts wird am 28.01.2016 um 15:15 bis 16:15 präsentiert.

Die Vorbereitung der Präsentation wird am 21.01.2016 zum Weekly Scrum Termin stattfinden.

Fußnoten

1 https://www.drupal.org/coding-standards

2 Die Drupal-Standards basieren ihrerseits auf den PEAR Coding Standards

3 https://github.com/styleguide/ruby

4 http://yardoc.org

5 http://ruby-doc.org/stdlib-1.9.3/libdoc/test/unit/rdoc/Test/Unit.html