PROJEKTPHASE 2

Qualitätssicherungskonzept

abs16
25. April 2016

1 DOKUMENTATIONSKONZEPT

1.1 CODE DOKUMENTATION

Code wird auf zwei Arten dokumentiert: referentiell und inline. Referentielle Dokumentation ist die Dokumentation von Methoden, Klassen o. Ä. Inline Dokumentation ist die Dokumentation von der Funktionsweise von Methoden und Klassen.

1.1.1 REFERENTIELLE DOKUMENTATION

Da wir Java als Programmiersprache nutzen, werden wir zur Erstellung einer externen Dokumentation des Codes das Dokumentationswerkzeug JavaDoc benutzen und uns dabei an die Standards halten, welche in Abschnitt 4 festgehalten werden. So erhalten wir während der Entwicklung eine Dokumentation, die wir intern nutzen können, und es steht Personen, die unser Produkt weiterentwickeln wollen, nach Abschluss des Projektes eine umfassende Wissenbasis über den Code zur Verfügung.

1.1.2 INLINE DOKUMENTATION

Die Form der inline Dokumentation steht dem jeweiligen Entwickler frei. Grundsätzlich gilt, dass Klassen und Methoden so modelliert sein sollten, dass sie von einem Entwickler, der die dazugehörige Javadoc kennt, weiterentwickelt werden können.

1.2 DOKUMENTATION DER ENTWURFSBESCHREIBUNG

Die Entwurfsbeschreibung wird in folgende Punkte gegliedert:

1. Allgemeines
2. Produktübersicht
3. Grundsätzliche Struktur- und Entwurfsprinzipien
4. Struktur- und Entwurfsprinzipien einzelner Pakete
5. Datenmodell
6. Glossar

Diese Gliederung entstammt dem Dokument “Softwaredokumentation im Praktikumseinsatz der Abteilung BIS”. An geeigneten Stellen in der Entwurfsbeschreibung werden wir auf Basiskonzepte zurückgreifen. Im Mittelpunkt stehen dabei Basiskonzepte der algorithmischen und objektorientierten Sicht, wie UML.

1.3 Dokumentation von Entwicklungsprozessen

1.3.1 Dokumentation von Tests

Fehler werden in unserem Scrumtool Taiga als Issue vom Typ “Bug” festgehalten und dort mit einer Einschätzung der Schwere (Severity) sowie der Wichtigkeit (Priority) versehen. In diesen Issues werden zuständige Entwickler und betroffene Pakete erwähnt, sodass der Fehler festgehalten werden kann, um später behoben zu werden.

1.3.2 Dokumentation von Entwicklungsfortschritt

Da wir zur Verwaltung der Projektdaten vollständig auf ein git-System setzen, wird jede Änderung an diesem mit einer Commit-Nachricht versehen. Diese muss aussagekräftig und in deutscher Sprache formuliert sein. Commits erfolgen kurzfristig und in sinnvollen Einheiten.

1.4 Dokumentation von Organisation

1.4.1 Interne Dokumentationsrichtlinien

Um einen reibungslosen Arbeitsablauf zu gewährleisten, wurden einige Richtlinien im Umgang mit Dokumenten, die nicht zum Code gehören, festgelegt. Wir verwenden Markdown und/oder LaTeX auf Basis von Templates. Der Verantwortliche für Dokumentation trägt dafür Sorge, dass Dokumente im git-Verzeichnis geordnet und einheitlich dargestellt werden.

1.4.2 Treffen und wichtigen Entscheidungen

Bei jedem Weekly Scrum wird ein Protokoll erstellt und auf unserer Website sowie im git-Verzeichnis verfügbar gemacht. So können Absprachen und dort getroffene Entscheidungen nachvollzogen werden. Wichtige Festlegungen aus internen Meetings werden bei den Weekly Scrums präsentiert und in unserem Kommunikationstool “Slack” festgehalten. Architekturentscheidungen und Entwurfsaspekte werden im Modellierungskonzept begründet festgehalten.

1.5 Sonstige Dokumentationsartefakte

1.5.1 Endnutzer Dokumentation - GUI

Dem Endbenutzer wird im Zusammenhang mit der GUI ein Hilfesystem zur Verfügung gestellt, in dem die grundlegende Nutzung des Produktes beschrieben wird. Dieses soll zudem über Beispieleingaben und ein FAQ verfügen.

2 Testkonzept

Während der Softwareentwicklung ist es wichtig, dass auf jeder Granularitätsebene alle Bestandteile regelmäßig auf Fehler überprüft werden. Dazu werden stets folgende Tests durchgeführt:

2.1 Durchgeführte Tests

2.1.1 Komponententests

Komponententests sichern die Korrektheit von einzelnen Methoden und Klassen. Hierbei wird darauf geachtet, dass jede Codezeile aller Funktionen von mindestens einem Test ausgeführt wird. Mit dieser vollständigen Überprüfung wird die Richtigkeit des gesamten Codes sichergestellt.

2.1.2 Integrationstests

Beim Integrationstest werden größere, logisch zusammengehörige Teile des Codes als Einheit getestet. Die korrekte Interaktion aller Methoden in einem ganzen Pakete oder mit einer Schnittstelle wird geprüft.

2.1.3 Systemtests

Systemtests sollen die Nutzung durch den Anwender simulieren und feststellen, ob das Programm erwartungsgemäße Ergebnisse erzielt. Um die Ergebnisse des Programms zu untersuchen, werden verschiedene Situationen zusammen mit erwarteten Ergebnissen modelliert und in xml-Dateien festgehalten. Diese werden mit jedem neuen Build ausgeführt und prüfen dieses damit auf Richtigkeit.

2.1.4 Abnahmetests

Beim Abnahmetest wird das fertige Produkt den Betreuern vorgestellt. Es wird festgestellt, ob alle Anforderungen erfüllt worden sind. Gegebenenfalls wird bei auftretenden Mängeln dokumentiert, in welcher Art und Weise die Software abzuändern oder zu erweitern ist.

2.2 Genutzte Tools

Alle Tests werden mit dem Framework 'JUnit' implementiert. Tests werden bei jedem Build automatisch ausgeführt. Der Ablauf wird durch 'maven' festgelegt. Um die vollständige Abdeckung des Codes zu überwachen soll ein Code coverage Tool in maven eingebunden werden. Auf dem Projektserver läuft 'Jenkins' und führt in regelmäßigen Abständen einen neuen Build durch.

2.3 Erfahrungen aus dem Vorprojekt

Im Vorprojekt wurde klar, dass die getrennte Entwicklung von Code und Tests zwar funktioniert, aber nicht zweckmäßig ist. Deshalb werden diese in Zukunft als Einheit implementiert und dann nur noch von einer zweiten Person überprüft. Weiterhin war die Abdeckung des Codes nur spärlich. Ursache dafür war vor allem der große Overhead. Es existierten viele Klassen um Daten zu speichern und variable Strukturen zu unterstützen, jedoch war nur wenig Funktionalität in diesen enthalten.

3 Organisatorische Festlegungen

3.1 Treffen

Jede Woche findet ein Weekly-Scrum Treffen des Teams mit dem Stakeholder und dem Tutor statt.

Das Team kann in diesem Rahmen vereinbaren, wann es sich außerhalb des Weekly-Scrums trifft.

3.2 Entwicklung

Entwickelte Klassen und/oder Module dürfen erst ins git-Verzeichnis gepushed werden, wenn sie erfolgreich kompilieren.

Eventuell können geeignete Branches verwendet werden, um das Implementieren unterschiedlicher Funktionalitäten zu trennen. Die Arbeit auf den verschiedenen Branches wird nicht anders organisiert als die auf dem master-Branch. Tests müssen auf allen Branches durchgeführt werden.

3.3 Tests

Komponententests Komponententests werden mit dem ersten Push ins git-Verzeichnis durch den Verantwortlichen auf den Server geladen. Diese werden automatisch bei jeder Änderung im Modul ausgeführt und vom Verantwortlichen überwacht.

Integrationstests Integrationstests werden automatisiert auf dem Server durchgeführt. Dies geschieht jeden Morgen. Die Verantwortung, dass diese Tests geordnet stattfinden, liegt beim Verantwortlichen für Tests.

Systemtests Systemtests werden jede Woche durchgeführt, damit Ergebnisse beim Weekly Scrum vorliegen. Die Verantwortung hierfür liegt beim Verantwortlichen für Tests.

Abnahmetest Der Abnahmetest findet eigens abgesprochen statt.

3.4 Aufgabenteilung

Hauptaufgaben sind die im Taiga eingetragenen User Stories. Diese werden vom Teamleiter erstellt und dann einem verantwortlich Entwickler zugewiesen. Dieser kann weitere Unteraufgaben (Tasks) anlegen und diese wiederum anderen Entwicklern zuweisen. User Stories müssen immer mit einer aussagekräftigen Beschreibung versehen werden.

3.5 Erfahrungen aus dem Vorprojekt

Im Vorprojekt mangelte es uns an einer strukturierten und organisierten Arbeitsweise. Teilweise war unklar, wem eine Aufgabe zugewiesen wurde und welche Schritte für die Erfüllung dieser Aufgabe notwendig waren. Deshalb haben wir uns dazu entschieden die komplette Aufgabenverteilung über unser Scrum Tool zu verwalten. Außerdem ist die Aufgabenverteilung dadurch streng hierarchisch aufgebaut und der Teamleiter ist immer über alle Aufgaben im Bilde.

4 Coding Standards für Java

Konsistenz der Quelltextformatierung und das Verwenden der Kommentier- und Programmierparadigmen ist wünschenswert, um die Leserlichkeit und Wartbarkeit des Codes zu erhöhen. Die hier vorgestellten Richtlinien sollen dabei helfen.

Dieses Konzept orientiert sich an den "Google Java Style", den Coding Standards für Java von Google Inc.

https://google.github.io/styleguide/javaguide.html

In Situationen, die nicht durch die hier vorgestellten Konzepte abgedeckt werden, wird Rücksprache mit dem Team gehalten. Die Diskussion sollte sich an den Richtlinien von Google orientieren. Ist eine neue Richtlinie gefunden, wird diese zu unseren Standards hinzugefügt.

4.1 Dateien

Alle Textdateien nutzen die Dateicodierung UTF-8. Jede Klasse, sofern diese keine *nested*-Klasse ist, wird in einer eigenen Datei implementiert. Klassen- und Dateiname müssen identisch sein. Dieser sollte möglichst kurz und prägnant gewählt werden. Übereinstimmung mit den in der Modellierungsphase gefundenen Objektnamen ist wünschenswert.

Besteht der Dateiname aus einem Wort, wird nur der Anfangsbuchstabe groß geschrieben. Besteht der Dateiname aus mehreren Wörtern, sind die Wörter nicht zu trennen. Jeder Anfangsbuchstabe wird groß geschrieben. Kommt in dem Dateinamen ein Akronym vor, wird nur der Anfangsbuchstabe groß geschrieben. (PascalCase)

```
Person                  -> Person.java
GUI                     -> Gui.java
Random number generator -> RandomNumberGenerator.java
IOs6 main               -> Ios6Main.java
Database                -> Database.java oder Db.java
```

Die Zeilenanzahl ist unbeschränkt. Die Spaltenanzahl wird nicht beschränkt, sollte aber aufgrund der Lesbarkeit gering gehalten werden. Extrem verschachtelte Befehle sind zu vermeiden. Ist dies nicht möglich, sollten die Befehle an sinnvollen Stellen in die nächste Zeile umgebrochen werden.

4.2 Dateistruktur

Alle Kommentare sind folgendermaßen aufgebaut:

```
/**
 * Kommentar
 * Kommentar 2
 * /
```

oder (nur für einzeilige Kommentare):

```
// Kommentar
```

Für auskommentierten Quellcode sind auch andere Blockkommentararten erlaubt.
Der Dateiaufbau ist wie folgt festgelegt:

```
/**
 * Beschreibung der Klasse
 * @author Name des Autors
 * @version Version
 * /
(leerzeile)
package statement
(leerzeile)
import statements
(leerzeile)
public class Name{
...
}
```

Öffnende geschweifte Klammern stehen hinter dem Konstrukt, welches sie öffnet. Schließende geschweifte Klammern stehen in einer eigenen Leerzeile. Ausnahme sind das *else*-Statement und leere Blöcke (s.u.). Diese können so positioniert werden, dass die Lesbarkeit des Codes erhöht wird. Einzeilige Kommentare hinter schließenden Klammern sind wünschenswert, um die Lesbarkeit zu erhöhen.

```
public Object(){
    if(var1){
        switch(var2){
            case 1: break;
        }// end switch var
    }else{}
}// end constructor
```

Wenn eine Situation das Wegfallen der geschweiften Klammern erlaubt, wird trotzdem geklammert. Wenn z.B. auf ein *if*-Statement nur eine Anweisung folgt, wird diese geklammert.
Methoden Aufrufe und Kontrollstrukturen werden ohne das Verwenden von Leerzeichen zwischen Schlüsselwort, Argumenten und Code geschrieben.

```
richtig -> public void fnc(){
...
}
richtig -> if(cond){
...
```

```
}
falsch  -> public void fnc () {
...
}
falsch -> if (cond) {
...
}
```

Es werden keine Tabs verwendet. Das Einrücken findet über Verwendung von vier Leerzeichen statt.

4.3 CODING STYLE

Die Sprache in Dokumentation und Code ist Englisch.

Es gibt keine Wildcard-Imports. Organisation der Importe kann in der Regel die IDE übernehmen.

Escape-Sequenzen werden durch entsprechende plattformunabhängige Javakonstrukte ersetzt:

```
"\r\n" , "\n" -> System.getProperty("line.separator").toString();
"\" , "/"     -> System.getProperty("file.separator").toString();
```

Jede Variablendeklaration deklariert genau eine Variable.

```
richtig -> int intName1;
richtig -> int intName2;
falsch  -> int intName1, intName2;
```

Variablennamen beginnen immer mit einem Kleinbuchstaben. Sind Namen zusammengesetzt, wird jedes folgende Wort mit Großbuchstaben begonnen. (CamelCase)

Einzeichen-Deskriptoren sollten nur in lokal begrenztem Kontext benutzt werden (z.B. Laufvariable in kleinen Schleifen). Bindezeichen sind in der Regel nicht erlaubt.

Konstanten werden komplett in Großbuchstaben geschrieben. Besteht der Konstantenname aus mehreren Wörtern, sind diese mit Unterstrich zu trennen.

```
int index;
int varName,
int dasIstEinName;
final int MY_NAME;
```

Bei Felddeklarationen sind die eckigen Klammern nicht Teil des Variablennamens.

```
richtig -> String[] args
falsch  -> String args[]
```

Bei *switch*-Statements folgt in der Regel auf jedes *case* ein *break*. Wird ein *break* weggelassen (fall-trough), muss das im *case* dokumentiert werden. Der *default* case ist immer implementiert.

Wird eine Methode einer Superklasse überschrieben, wird das mit *@Override* über dem Methodenkopf kommentiert.

Bei mehreren, nah beieinander stehenden Befehlen, die einen *try-catch*-Block erfordern, sollten diese zusammen in einen solchen Block geschrieben werden. Trotzdem gilt: so viel wie nötig, so wenig wie möglich in *try-catch*. Darüber hinaus sollte jede Fehlerart, der von einem Block geworfen werden kann, mit einem eigenen *catch* behandelt werden. Damit sind Fehler-Wildcards wie:

```
catch(Exception e){ //doSomething}
```

nur in Ausnahmesituationen zu verwenden.

Kein *catch*-Block darf leer sein.