Projekt AGB-10

Qualitätssicherungskonzept

Verantwortliche: Josepha Gelfert, Tony Mey
Gruppe: SWP10-9

25. Mai 2010

Inhaltsverzeichnis

1 Dokumentationskonzept **3**
1.1 Interne, quelltextnahe Dokumentation 3
1.2 Externe Dokumentation 5

2 Organisatorische Festlegungen **5**

3 Testkonzept **6**
3.1 Einleitung 6
3.2 Komponententest 6
3.3 Integrationstest 7
3.4 Systemtest 7
3.5 Abnahmetest 8
3.6 Assertion 8
3.7 JUnit 9
3.8 Nameskonventionen 10

1 Dokumentationskonzept

Das Dokumentationskonzept dient zur Qualitätssicherung. An Hand von festgelegten Kriterien soll die Entwicklung eines übersichtlichen, lesbaren und nachvollziehbaren Quellcodes garantiert werden. Eine gute Dokumentation erleichtert die Wartbarkeit, sowie die Einarbeitung fremder Entwickler in ein Produkt.
Ebenso soll das Dokumentationskonzept dem Nutzer ein besseres Arbeiten mit dem Programm ermöglichen.

1.1 Interne, quelltextnahe Dokumentation

Die interne Dokumentation wird nicht veröffentlicht. Sie richtet sich insbesondere an das Entwicklungsteam des Softwareproduktes und hat die Aufgabe Wartbarkeit und Entwicklung des Softwareproduktes zu unterstützen.

Regeln für Bezeichner, um sie selbsterklärend und übersichtlich zu gestalten:

- selbsterklärende Namen für Methoden, Klassen und Variablen, d.h sie entsprechen ihrem Verwendungszweck
- einheitliche Sprache für die Bezeichnung; Verwendung englischer Bezeichner
- Bezeichner beginnen immer mit einem Buchstaben
- Bezeichner enthalten keine Leerzeichen
- in zusammengesetzten Bezeichnern beginnt jedes Wort mit einem Großbuchstaben
- Klassennamen und Interfacenamen beginnen immer mit einem Großbuchstaben
- Namen von Variablen und Methoden fangen mit einem Kleinbuchstaben an
- Konstanten werden komplett groß geschrieben, mehrere Wörter werden durch einen Unterstrich getrennt
- Methoden, die auf Attribute lesend zugreifen, werden mit vorangestelltem „get“ benannt; Methoden, die auf Attribute schreibend zugreifen, mit vorangestelltem „set“

Regeln für die Kommentare des Programms, um anderen Entwicklern das Einarbeiten in den Quellcode zu erleichtern:

- jede Methode und jede Klasse wird durch ein Kommentar beschrieben
- Befehle, die nur schwer nachvollziehbar sind, werden durch Kommentare erklärt
- kurze, zeilengebundene Kommentare werden durch „//“ kommentiert
- detaillierte Kommentare über mehrere Zeilen, werden mit „/**“ versehen, jede neue Zeile in diesem Kommentar beginnt hierbei mit „*“
- der Kommentar jeder Methode enthält die Beschreibung der Methode, die Parameterliste gekennzeichnet mit @param und die Rückgabewerte gekennzeichnet mit @return, falls nötig ist auch @throw anzugeben
- der Kommentar jeder Klasse besteht aus einer Beschreibung der Klasse, der Versionsnummer (gekennzeichnet mit @version) und dem Autor dieser Klasse (gekennzeichnet mit @author)

Regeln für die Struktur des Programms zur Verbesserung der Übersichtlichkeit:

- Klassen beginnen mit einem einleitenden Kommentar (siehe: Regeln für die Kommentare)
- Vor jeder Methode erfolgt die Beschreibung der Methode in Form eines Kommentars (siehe: Regeln für die Kommentare)
- die Parameterliste steht auf derselben Zeile, wie die dazugehörige Methode
- die Klammern des Rumpfes einer Methode sind jeweils auf extra Zeilen in Höhe der Methode
- der Inhalt der Methode ist eingerückt
- Leerzeilen sollen Variablen von Methoden, Methoden von Methoden und lokale Variablen von ersten Programmanweisungen trennen
- Deutsche Umlaute (ä,ö,ü) und ß dürfen nicht verwendet werden, da sie nicht überall akzeptiert werden
- Einrückungen erfolgen durch Tabs
- Zeilen sollten nicht zu lang sein, damit nicht erst nach rechts gescrollt werden muss, um sie zu Ende zu lesen; die maximale Länge betrage die Breite eines A4 Blattes (entspricht ca. 90 Zeichen), andernfalls ist ein Zeilenumbruch angebracht und der Rest des Befehls wird ohne Einrücken fortgesetzt

- zwischen dem Parameter im Funktionskopf werden nach jedem Komma Leerzeichen gesetzt
- pro Zeile wird nur eine Anweisung formuliert
- Deklarationen stehen immer am Anfang von Blöcken
- bei Kontrollstrukturen werden immer geschweifte Klammern gesetzt, „{“ und „}“ stehen dabei auf separaten Zeilen auf der Höhe des ersten Zeichens der Kontrollstruktur

Sollte der Code später noch einmal verändert werden, sollte man darauf achten, dass diese Änderung beim Hochladen in SVN genau dokumentiert wird.
Alle vorgenommenen Tests werden im Testkonzept beschrieben. Damit soll sichergestellt werden, dass alle Programme auch tatsächlich ihren Zweck erfüllen. Die Kommentierung der Testfälle erfolgt intern durch die Quelltext- und Javadoc-Dokumentation.

1.2 Externe Dokumentation

Die externe Dokumentation umfasst also alle produkbegleitenden Informationen für den Kunden und wird im Gegensatz zur internen Dokumentation veröffentlicht. Sie soll dem Anwender als Bedienungsanleitung für die Software dienen.
Die externe Dokumentation besteht aus der Design-Beschreibung und dem Benutzerhandbuch. Die Design-Beschreibung gibt einen Einblick in die grundlegende Architektur des Systems. Sie ermöglicht auch fremden Entwicklern, sich schnell mit dem Projekt vertraut zu machen und gewährt dem Kunden einen Überblick über die grundlegenden Eigenschaften des Systems. Parallel zum Entwicklungsprozess wird ein Benutzerhandbuch geschrieben, in dem die Funktionsweise des Produktes erklärt und beschrieben wird. Das Benutzerhandbuch soll auch online zur Verfügung stehen.

2 Organisatorische Festlegungen

Jeder Programmierer ist für die Dokumentation seines Quellcodes selber zuständig. Dabei soll er sich an die im Dokumentationskonzept festgelegten Richtlinien halten, dies beinhaltet die Wahrung der Form, die richtige Namensgebung und die Kommentierung während des Programmierens. Dem Verantwortlichen für Qualität und Dokumentation unterliegt die Kontrolle der Dokumentationen. Die Erstellung, beziehungsweise das

Zusammentragen der Bestandteile der externen Dokumentation, liegt ebenso in den Händen des Verantwortlichen für Qualität und Dokumentation.
Der Verantwortliche für Tests überwacht die Tests.
Um eine termingerechte Abgabe der Aufgaben zu garantieren, gibt es noch vor dem eigentlichen Abgabetermin gruppeninterne Abgabetermine.

3 Testkonzept

3.1 Einleitung

Das Testkonzept dient dazu, Fehler in der Software möglichst frühzeitig zu erkennen und Folgefehler zu vermeiden. Fehler können zum Beispiel durch Vertippen oder Unwissenheit der Implementierer entstehen oder sogar direkt in der Planung der Software. Dabei sind nur die zur Laufzeit auftretenden Fehler im Blickpunkt. Syntaxfehler werden im Allgemeinen bereits vom Compiler gefunden und angezeigt.
Durch Tests kann eine korrekte Funktionalität der Software mit den verschiedensten Eingaben nahezu garantiert werden. Besonders wichtig ist dabei, die Testläufe sorgfältig zu organisieren und konsequent während der gesamten Entwicklung durchzuführen. Dies verringert die Zeit, in der man die Fehlerursachen sucht, da bereits getestete Elemente funktionieren müssen und sich so der Suchbereich stark einschränken kann.
Für das Testen von Klassen und Methoden kann auf bereits bewährte Hilfsmittel wie Assertions und JUnit zurückgegriffen werden (Siehe 3.5 und 3.6).
Die Tests des Projekts werden in 4 Phasen gegliedert:

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

3.2 Komponententest

Komponententests überprüfen die kleinsten Bausteine der Softwarearchitektur, die Klassen und Funktionen. Im Blickpunkt steht dabei nur, ob die geforderte Aufgabe erwar-

tungsgemäß erfüllt wird, nicht der dahinter stehende Algorithmus oder Code.
Im Idealfall sollten die Tests selbst bereits vor dem Implementieren der Komponenten bereit stehen und werden während der Entwicklung immer wieder automatisch durchgeführt. So können Problemfälle sofort erkannt und behoben werden.
Jede Komponente wird dabei isoliert getestet. Eingabe und sonstige erforderliche Daten werden simuliert, wobei nicht nur die möglichen und Spezialfälle getestet werden, sondern auch falsche Eingaben und Fehlermeldungen, die schlimmstenfalls an die Komponente übergeben werden können.
Mit Abschluss eines erfolgreichen Komponententests sollte der jeweilige Baustein des Programms jede Eingabe korrekt bearbeiten und Fehler abfangen können.

3.3 Integrationstest

Nachdem jede Komponente für sich genommen fehlerfrei funktioniert, wird im Integrationstest das Zusammenspiel der Komponenten miteinander überprüft. Im Blickpunkt steht der korrekte Datenaustausch über die gemeinsam definierten Schnittstellen und die Daten, insbesondere die Datentypen selbst.
Dabei wird immer das Zusammenspiel von je zwei Komponenten getestet. Nur bei größeren Projekten kann es notwendig sein, bereits getestete Elemente als neue Komponente aufzufassen und mit diesen weiter zu testen, um den Aufwand zu reduzieren.
Integrationstests werden im Gegensatz zu den Komponententests manuell durchgeführt.

3.4 Systemtest

Beim Systemtest werden alle Komponenten zusammengesetzt und auf Erfüllung aller im Pflichtenheft festgehaltenen Funktionalitäten geprüft. Wichtig ist dabei festzuhalten inwieweit die Funktionalitäten erfüllt sind und welche verbesserungswürdig sind. Desweiteren müssen die Eingaben und Benutzerhinweise deutlich erkennbar sein und das Layout an die Funktion angepasst sein. Außer der Funktionalität und dem Design muss die Performance der Software den gesetzten Anforderungen entsprechen.
Im Allgemeinen findet diese Phase des Testkonzepts in einer Testumgebung mit Testdaten statt, um wieder alle möglichen und eigentlich unmöglichen Eingaben und Fehler zu testen. Die Testumgebung muss dabei bereits die Arbeitsumgebung des Kunden simulieren, um thematisch relevante Probleme erkennen zu können. Der Kunde selbst kann dabei

bereits mit einbezogen werden.
Weitere wichtige Eigenschaften, die nun zu überprüfen sind:

- Leistung
- Performance
- Zuverlässigkeit
- Robustheit

3.5 Abnahmetest

Beim Abnahmetest wird dem Kunden das fertige Produkt vorgeführt und die von ihm gestellten Erwartungen mit dem Ergebnis der Entwickler verglichen. Dieser Schritt kann bereits mit frühen Versionen des Produkts durchgeführt werden, um die Entwicklung mit Hilfe des Kunden in die richtige Richtung lenken zu können.

3.6 Assertion

Assertions sind Behauptungen die der Programmierer aufstellen und im Quellcode abfragen kann. Jede Behauptung muss zu Wahr ausgewertet werden, ansonsten wird eine entsprechende Exception geworfen. Assertions in Java haben die folgende Form:

assert AssertConditionExpression;

assert AssertConditionExpression : MessageExpresion;

zum Beispiel:

assert eingabe.length() > 0;
assert x = 3: “fehlerhafte Anzahl”;

Assertions können aktiviert und deaktiviert werden. So muss der Quellcode nicht verändert werden um das eigentliche Programm ohne Leistungsverlust durch assert-Überprüfungen zu erstellen. Daher ist es auch wichtig, dass Assertions keine Seiteneffekte haben. Das Programm also auch ohne diese Funktioniert. Assertions werden aktiviert durch den

Befehl:
$ java **-ea** MainProgram

3.7 JUnit

Die JUnit ist ein Framework zum Testen von Java-Programmen. Es ist besonders für das automatisierte Testen einzelner Units, vor allem Klassen und Methoden, gedacht.
Es gibt nur zwei mögliche Ergebnisse eines Tests: Entweder gelingt der Test oder misslingt. Die Ursache für das Misslingen kann dabei sowohl ein Fehler (Failure) als auch ein falsches Ergebnis (Error) sein. Beides wird durch eine Entsprechende Exception von JUnit signalisiert. Fehler werden erwartet, wohingegen Errors eher unerwartet auftreten und auf weitere Testszenarien hinweisen können. Failures werden von JUnit mit der Exception "AssertionFailedError" signalisiert.
JUnit bietet eigene Assertions an die mit import static org.junit.Assert.∗; eingebunden werden. Diese erweitern vereinfachen das Handhaben von Assertions.
Wichtige Assertions die noch viele weitere Überladungen bieten sind:

assertTrue(boolean condition);
assertFalse(boolean condition);
assertNull(Object object);
assertNotNull(Object object);

assertArrayEquals(Object[] expecteds, Object[] actuals);
assertEquals(Object expecteds, Object actuals);
assertSame(Object expected, Object actual);
assertNotSame(Object[] expected, Object[] actual);

Zu jeder Klasse werden mit JUnit eine oder mehrere Testklassen mit Testfunktionen erzeugt, die wiederum Methoden zum Testen enthalten. In den Methoden können die jeweiligen Instanzen erzeugt und die Assertions überprüft werden.

Beispiel:
import static org.junit.Assert.∗;
import org.junit.Test;
public class TestKlassename
@Test

```
public void testMethodenname1()
Klassenname kn = new Klassenname();
int expected = 0;
int result = kn.Methodenname1();
assertTrue( expected == result);

@Test
public void testMethodenname2(){...}
```

Diese Klasse kann man kompilieren und ausführen, wobei jede mit "@Test" gekennzeichnete Funktion ausgeführt und entsprechend der Erfolg oder Misserfolg jedes Tests gemeldet wird.

3.8 Nameskonventionen

- Methoden: "test" + Methodenname
- Testklassen: "Test" + Klassenname