QS-Konzept

1. Dokumentationskonzept

Die Dokumentation erfolgt in deutscher Sprache, da OLAT größtenteils im deutschsprachigen Raum Anwendung findet. Die Dokumentationsrichtlinien wurden an der bestehenden OLAT Entwicklung ausgerichtet, so dass sich das Projekt nahtlos in das bestehende OLAT einfügen lässt. Weiterhin werden die Java Code-Conventions eingehalten.

1.1. Aufbau von Java Dateien

Alle Java Quelldateien sollen folgendes Format haben:

- Anfangskommentar (/** Copyright */)
- Zuerst Package und dann Import Anweisungen
- Klassen- und Interfacedeklarationen

1.2. Änderungsdokumentation

Jede Änderung des Programmes muss sorgfältig im SVN kommentiert werden, damit die anderen Programmierer wissen welche Änderungen vorgenommen wurden. Jeder Programmierer sollte außerdem alle Kommentare sorgfältig lesen, bevor er beginnt eine Datei zu bearbeiten, damit er immer weiß, was am Programm verändert wurde.

1.3. Allgemeines zu Kommentaren

Die Kommentare zur Implementierung (/*...*/) sollen Teile des jeweiligen Quelltextes beschreiben, während Kommentare zur Dokumentation (/**...*/) die Eigenschaften des Quelltextes beschreiben sollen, welche auch ohne Zugriff auf den Quelltext verständlich sein müssen.

Die Dokumentation findet mit JavaDoc statt. Jede Klassenvariable, Methode und Klasse ist mit JavaDoc zu dokumentieren.

Immer wenn man einen Kommentar verfassen möchte, sollte man überlegen, ob nicht eher der Code verbessert werden könnte, um ihn verständlicher zu machen.

1.4. Namenskonventionen

Alle Namen werden in der englischen Sprache verfasst und sollten "sprechende Namen" sein. Klassennamen sollen Substantive sein (bzw. aus Substantiven zusammengesetzt sein) und mit einem Großbuchstaben beginnen. Jedes Wort innerhalb des Klassennamens beginnt ebenfalls mit einem Großbuchstaben. Interfaces werden wie Klassen geschrieben.

Methoden sollen Verben sein, wobei der erste Buchstabe klein geschrieben wird und Wörter innerhalb des Methodennamens mit einem Großbuchstaben geschrieben werden. Variablen unterliegen derselben Groß- und Kleinschreibung wie Methoden.

1.5. Deklarationen

Funktionen und Variablen werden niemals in einer Zeile deklariert. Deklarationen werden immer am Anfang von Blöcken ({...}) angeordnet. Ausnahmen hierbei sind Indizes in Schleifen.

Bei Klassen- und Interfacedeklarationen werden keine Leerzeichen zwischen Methodenname und "(" am Start der Parameterliste eingefügt.

Methoden werden mit einer Leerzeile voneinander getrennt.

1.6. Benutzerdokumentation

Während des Entwicklungsprozesses wird zeitgleich ein Benutzerhandbuch erstellt, welches die Funktionsweise des Produkts beschreibt und erklärt.

1.7. Entwicklerdokumentation

Es wird eine Entwicklerdokumentation erstellt, damit auch Programmierer, die mit dem Projekt nicht vertraut sind, sich einfach in das Programm einarbeiten können. Dies erfolgt mit einer Designdokumentation, welche die Softwarearchitektur beschreibt und außerdem ihre Funktionen beschreibt.

Der Verantwortliche für Qualitätssicherrung und Dokumentation ist für die Erstellung dieser Dokumentation verantwortlich, auch wenn sie unter Mitwirkung aller Projektmitarbeiter zusammengestellt wird.

1.8. Allgemeine Sachen

- Dateien sind in Abschnitte zu unterteilen, welche mit Leerzeilen voneinander getrennt und möglicherweise durch einen Kommentar beschrieben werden
- Nur eine Anweisung pro Zeile
- Zum Einrücken werden Tabs verwendet (keine Leerzeichen)
- Nach Kommas (z.B. bei Parameterlisten) bzw. Semikola (z.B. bei Schleifen) wird ein Leerzeichen gesetzt
- Vor und hinter Vergleichs- und Rechenoperatoren wird ein Leerzeichen gesetzt
- Die öffnende geschweifte Klammer "{" steht stets in der selben Zeile wie die Methode/Anweisung und zwar ganz am Ende der Zeile
- die schließende geschweifte Klammer "}" steht in einer eigenen Zeile auf derselben Höhe, auf der die Anweisung begonnen hat. (Ausnahme: Bei einer leeren Anweisung steht sie direkt hinter "{".)

2. Testkonzept

2.1. Einleitung

Die Entwicklung von Software erfordert ein Testkonzept, welches die Fehlerfreiheit des Codes und der Funktionalität garantieren kann und eventuell auftretende Bugs identifiziert. Zu diesem Zweck wird bei der Schöpfung umfangreicher Softwareprodukte, während der Implementierung, auf Hilfsmittel wie zum Beispiel JUnit zurückgegriffen. Die Tests des Projektes werden in vier Phasen gegliedert:

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

2.2. JUnit

JUnit ist ein Framework zum Testen von Javaprogrammen und basiert auf Konzepten die ursprünglich unter dem Namen SUnit für Smalltalk entwickelt wurden. Entwickler des JUnit-Frameworks sind Erich Gamma und Kent Beck. JUnit spezialisiert sich auf automatische Unit-Tests (für Klassen und Methoden). Solch ein Test kennt nur zwei mögliche Ergebnisse, das Gelingen bzw. das Misslingen des Tests. Ein Misslingen liegt bei einem Fehler oder bei einem falschen Ergebnis vor, dessen Ursache durch eine Exception signalisiert wurde. Beim Auftreten von keinem Fehler gilt der Test als gelungen. JUnit unterstützt in diesem Zusammenhang die Idee des Extreme-Testings. Dabei schreibt das Programmierteam zuerst einen automatisch wiederholbaren JUnitTest und dann den zu testenden Code.

2.3. Komponententest

Der Komponententest bezieht sich auf das Testen von Klassen und Methoden. Dafür werden vor der Implementierung geeignete Testszenarien festgelegt und später die Klassen auf ihre Spezifikation überprüft. Die Testszenarien werden mittels des JUnit Frameworks in eine Testklasse implementiert. Die einzelnen Testklassen werden zu einer Testsuite zusammengefügt. Folgende Notation ist dabei einzuhalten:

- Testklassen heißen „Test“ + Klassenname
- Testsuiten heißen „Testsuit“ + Paketname

Das heißt in dieser Phase werden die einzelnen Module (der Converter, die Modelkommunikation, …) auf „Herz und Nieren“ getestet.

2.4. Integrationstest

Der Integrationstest wird in zwei Kategorien unterteilt. Zum einen wird das MVC-Konzept der OLAT-Extension getestet. Zum anderen soll die integrierte Extension im Zusammenspiel mit OLAT überprüft werden. Als weiteres Hilfsmittel kann dafür der OLAT-Debugger herangezogen werden, um zu überprüfen ob alle Module fehlerfrei funktionieren. Im Speziellen, wird in dieser Phase überprüft, ob die Backupfunktionen mit OLAT richtig korrespondieren.

2.5. Systemtest

Der Systemtest prüft auf Basis des Pflichtenheftes das Softwareprodukt. In dieser Phase wird aus der Sicht des Anwenders geprüft ob alle Anforderungen zufriedenstellend umgesetzt wurden. Funktionstests, Sicherheitstests und Interoperabilitätstest treten dabei besonders in den Vordergrund. Anschließend werden die nicht-funktionalen Anforderungen geprüft:

- Vollständigkeit
- Leistung
- Performance
- Zuverlässigkeit
- Robustheit
- Benutzbarkeit

2.6. Abnahmentest

Der Abnahmetest stellt den Produktabschluss dar. In dieser Phase wird dem Kunden das entwickelte Softwareprodukt vorgestellt und sichergestellt, dass die Anforderungen des Kunden umgesetzt wurden.

2.7. Testbeispiel

In diesem Abschnitt wird versucht zu erklären wie JUnit an einem Beispiel funktioniert, um die Programmiererteams in das Testkonzept einzuweisen.

Das Prinzip des Testens liegt im Vergleichen von SOLL- mit IST-Werten. Stimmen beide Werte überein so war der Test erfolgreich, anderenfalls ist der Test fehlgeschlagen. Um diese Vergleiche durchzuführen stellt JUnit folgende Methoden zur Verfügung:

- Testen auf Gleichheit

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

- • Spezialfälle von Gleichheit-Tests

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

- Assert-Methoden müssen statisch importiert werden

```
import static org.junit.Assert.*
```

2.8. Beispiel

Als einfaches Beispiel betrachten wir eine Klasse zum multiplizieren zweier Zahlen.

```
public class Rechnen {
  public int multiplizieren(int ersterWert, int zweiterWert ) {
   return ersterWert * zweiterWert;
  }
}
```

Die dazugehörige JUnitTestklasse sieht wie folgt aus:

```
public class TestRechnen extends TestCase {
  public void testMultiplizieren {
   int ersterWert = 6;
   int zweiterWert = 2;
   int erwartetesErgebnis = 12; // 2x6=12
   Rechnen r = new Rechnen();
   int ergebnis = r.multiplizieren(ersterWert, zweiterWert);
   assertTrue(erwartetesErgebnis == ergebnis);
  }
}
```

Mit beiden Codestrukturen können wir nun einen Test durchführen, indem wir eine Instanz der TestRechnen-Klasse erzeugen und die Methode testMultiplizieren ausführen:

```
TestCase test = new TestRechnen("testMultiplizieren");
test.run();
```

Wenn die Ergebnisse nicht übereinstimmen, wirft JUnit eine Execption.

3. Organisatorische Festlegungen

Die komplette Dokumentation wird am Ende zu einer Gesamtdokumentation zusammengefügt, so dass man sich später leicht in das Projekt einarbeiten kann bzw. das Produkt einfach benutzen kann.

Verantwortlichkeiten und Termine:

Für die Dokumentation ist jeder Programmierer selbst verantwortlich. Der Verantwortliche für Dokumentation und Qualitätssicherung ist lediglich für die Erstellung und Kontrolle der Richtlinien zuständig. Außerdem ist er für die Erstellung des Benutzerhandbuchs zuständig. Termine sind stets einzuhalten und deshalb wird es intern einen Abgabetermin geben der noch vor dem eigentlichen Abgabetermin steht.