Qualitätssicherungskonzept

I. Dokumentationskonzept

i) quelltextnahe Dokumentation:
Um gut lesbaren, strukturierten und verständlichen Code zu garantieren und somit die Arbeit im Team zu erleichtern, sind folgende Konventionen zur quelltextnahen Dokumentation einzuhalten, die sich an den JAVA Code-Conventions orientieren.

- Um die Dokumentation Klassen, Klassenvariablen und Methoden leichter zugänglich zu machen, werden die Möglichkeiten von javadoc ausgeschöpft:
- Sämtliche Klassen, Klassenvariablen und Methoden werden mit javadoc Kommentaren /** .. */ so ausführlich wie möglich beschrieben. Der Implementierer versucht dabei von seinem eigenen Verständnis zu abstrahieren um eine verständliche Funktionsbeschreibung zu formulieren. Besondere Aufmerksamkeit ist hier auf die richtige Nutzung der Tags zu setzen. (@author für den Klassenauthor, @return für Rückgaben, @param für Parameter usw)
- Zusätzlich werden schwer verständliche, also nicht-triviale Code-Passagen mit // kommentiert falls es sich um einzeilige Kommentare, mit /* .. */ falls es sich um mehrzeilige Kommentare handelt.
- Namen von Klassen, Methoden und Variablen werden selbsterklärend und in englischer Sprache gewählt. Umlaute werden vermieden. (ae statt ä usw)
- Klassennamen sind Substantive im Singular und beginnen mit Großbuchstaben, wobei der Anfangsbuchstabe jedes inneren Wortes groß geschrieben wird. Sie sind kurz und prägnant, nicht gebräuchliche Akronyme und Abkürzungen werden vermieden.
- Methodennamen sind Verben, deren erster Buchstabe klein geschrieben wird. Der Anfangsbuchstabe jedes inneren Wortes wird auch hier groß geschrieben. Getter und Setter privater Attribute heißen *getAttribute()* bzw *setAttribute()*. Boolsche Attribute werden mit *isAttribute()* abgefragt.
- Variablennamen beginnen klein, jedes innere Wort wird groß geschrieben. Einzelne Buchstaben als Variablen werden vermieden, außer es handelt sich um eine temporäre Variable z.B. als Indexvariable.
- KONSTANTEN werden komplett groß geschrieben, einzelne Wörter werden mit _ abgetrennt.

Zur Übersichtlichkeit des Codes ist dieser wie folgt zu strukturieren:

- Es wird eine Zeile pro Deklaration verwendet, das Mischen von Datentypen ist zu vermeiden. Weiterhin stehen Deklarationen am Anfang eines Blockes.
- Variablen werden dort initialisiert, wo sie deklariert werden, es sei denn ihr Wert hängt von einer Berechnung ab.
- Nach Klassen- und Methodendeklarationen steht die öffnende geschwungene Klammer in der selben Zeile, die schließende geschwungene Klammern in einer eigenen Zeile.
- allgemein wird eine Zeile pro Statement verwendet. Eingeschlossene Blöcke von Statements werden ein Level mehr eingezogen als das umschließende Statement. Die öffnende Klammer steht in der selben Zeile wie das umschließende Statement. Auch um einzelne Statements werden Klammern gesetzt, wenn sie Teil einer Kontrollstruktur wie if-else oder for sind.

ii) Entwurfsdokumentation
Die Entwurfsdokumentation dient dem verständlichen und nachvollziehbaren Überblick über das Projekt. Sie enthält zumindest die Klassendiagramme inklusive Beschreibungen , Sequenz- und/oder Aktivitätsdiagramme wichtiger Abläufe, eine vereinfachte Gesamtübersicht und wenn nötig weitere Diagramme. Entscheidungen hinsichtlich kritischer Entwurfsdetails werden außerdem begründet und der Entwurfsdokumentation beigefügt. Für die Diagramme gilt der UML 2.0 Standard.

iii) Änderungsdokumentation
Nach jeder abgeschlossenen Arbeit am Code ist dieser ins SVN-Repository hochzuladen. Dabei sind die vorgenommenen Änderungen unbedingt über die zur Verfügung stehende Funktion zu kommentieren, um diese Änderungen nachvollziehbar zu machen.

iv) Testdokumentation
Um die Durchführung und den Verlauf nötiger Programmtests zu belegen, werden diese dokumentiert. Testplanung und -dokumentation sind im Testkonzept dargelegt.

II. Testkonzept

i) Einführung
Ein umfangreiches und durchdachtes Testkonzept ist der Schlüssel zur Anfertigung qualitativ hochwertiger Software. Hierbei ist zu beachten, dass unser Projekt eine Erweiterung einer bereits bestehenden Software, der Lernplattform OLAT, ist. Deswegen ist für viele Komponenten unserer Software nicht ausschließlich entscheidend, ob sie die geforderte Funktionalität fehlerfrei erbringen, sondern auch, ob sie dies innerhalb der OLAT Umgebung tun. Unser Testkonzept stützt sich im Wesentlichen auf drei Teile:
Teil 1 sind die Komponententest via JUnit. Diese werden zeitgleich mit der Implementierung des jeweiligen Releases vom zuständigen ProgrammingPair erstellt, jedoch nur für Methoden, die am Datenfluss beteiligt sind, nicht für reine GUI-Elemente.
Teil 2 sind die Integrationstests. Wir werden jegliche Anzeigeelemente eines Releases im Kontext des OLAT einem Sichttest unterziehen. Dadurch sollte nicht nur die korrekte Darstellung der Elemente gewährleistet werden, auch grobe Fehler im Datenfluss sollten dadurch sofort ersichtlich sein.
Teil 3 setzt sich aus dem finalen Systemtest und dem Abnahmetest zusammen. Ersterer besteht wiederum aus getrennten Sichttests durch jedes Mitglied der Gruppe sowie einer JUnit-Testsuite zur Überprüfung des Datenflusses des finalen Releases. Der Abnahmetest erfolgt beim Kunden, der selbst die Einhaltung der geforderten Funktionalität veri- oder falsifiziert.

ii) JUnit-Tests
Aus diversen Gründen können wir nicht für alle Methoden jeglicher Klassen Junit-Testobjekte schreiben. Dies wäre zum Beispiel bei GUI-Elementen auch nicht sinnvoll . Wichtig sind hier aber jene Klassen, die aktiv am Datenfluss beteiligt sind.
Als erstes wären hier die Klassen des DatasourceInteractions-Paketes zu nennen. Da die Klasse ODSparqler bereits im Vorprojekt auftauchte und vom Kunden verifiziert wurde, können wir davon ausgehen, dass diese korrekt arbeitet und die richtige Menge an Datensätzen mit dem geforderten Informationsgehalt aus der externen Datenquelle holt. Zu testen sind aber die beiden Manager CourseTableDBInteractionManagerImpl und BlockTableDBInteractionManagerImpl, die sowohl für die Speicherung der OD-Daten in der SQL-Datenbank von OLAT als auch für Abfragen auf diese und Umwandlung der gefundenen

Datensätze in Java-Objekte verantwortlich sind. Diese beiden Funktionalitäten müssen getrennt voneinander beim jeweiligen Release getestet werden. Dazu bietet es sich an, in die JUnit-Tests eingebettetete mySQL-Befehle zu integrieren und mittels assertEquals() und assertNull() zu prüfen, ob die Zusammensetzung der Tabellen in der Datenbank korrekt ist und ob die von den Managern zurückgegebenen Datenobjekte die geforderten Eigenschaften haben.
Unsere Erweiterung nutzt zudem diverse von uns geschaffene Datenobjekte, das QueryDataObject, das CourseDataObject, das BlockDataobject, sowie das ScheduleEvent. Nachdem deren Funktionalität verifiziert wurde, sind sie hervorragend dazu geeignet, die Konsistenz des Datenflusses an einer bestimmten Stelle zu überprüfen. Sie werden also als Testfälle in unsere JUnit-Testklassen integriert werden. Auch in der finalen Testsuite werden wir unter anderem den Datenfluss dadurch verifizieren, dass durch Einfügen gut gewählter Testdatensätze in die Datenbank unser Produkt in den frontend-nahen Klassen die erwarteten Datenobjekte generiert.

iii) Sichttests
Wie bereits erwähnt, ist die Funktionalität innerhalb von OLAT für unser Produkt entscheidend. Bereits bei der Implementierung des Vorprojektes fiel uns auf, dass ein spezieller geschachtelter Funktionsaufruf vom nativen Javacompiler des Implementierers anders behandelt wurde als von dem in dessen OLAT-Installation integrierten. Ausführliche Sichttests unseres Produktes stellen daher nicht nur sicher, dass die Benutzeroberfläche das „Look and feel“ von OLAT wahrt und die korrekten Ergebnisse angezeigt werden, sondern zeigen auch unvorhergesehene Komplikationen mit OLAT-Spezifika auf. Zu diesem Zwecke melden sich die Implementierer in verschiedenen Systemrollen bei verschiedenen unserer OLAT-Installationen an und testen unser Produkt aus den damit verbundenen Sichtweisen. In der Implementierungsphase eines einzelnen Releases tun dies die ProgrammingPairs der jeweiligen Story in ihren OLAT-Installationen, vor der Veröffentlichung des jeweiligen Releases aber testen auf diese Art aber noch einmal die anderen beiden Pairs die Funktionalität einer Story.

iv) System- und Abnahmetest
Beim Systemtest kommen noch einmal umfangreiche Sichttests ins Spiel. Auch wird, wie schon erwähnt, eine JUnit-Testsuite entwickelt, die nicht nur alle bisherigen Testklassen enthält, sondern speziell auch die Zusammenarbeit aller Pakete mittels ausgewählter Testfälle überprüft.

v) Testdokumentation
Alle JUnit-Testklassen werden von ihrem jeweiligen Schöpfer hinreichend dokumentiert. Die Ergebnisse der Anwendung, der Sichttests und der Durchläufe der Testsuite werden mit Datum, Namen des Ausführenden und Ergebnis in einem Protokoll festgehalten, welches dem finalen Release als Informationsmaterial beigelegt wird.

vi) Planungsrisiken und Unvorhergesehenes
Als gesondertes Risiko sei an dieser Stelle der Mehrbenutzerbetrieb erwähnt. Zwar wurde bei der Modellierung unserer Erweiterung darauf geachtet, dass z.B. nur Klassen, die keine Klassenvariablen besitzen, als Singleton angelegt werden und ansonsten für jede Klasse pro Nutzer mindestens eine Instanz angelegt wird, was aber im Extremfall bei gleichzeitigem Zugriff hunderter Nutzer auf unsere Erweiterung passiert, können wir nicht exakt vorhersagen und aus Personalmangel auch nicht testen.

III. organisatorische Festlegungen

i) Qualitätsstandards
Die Einhaltung der oben geschilderten Konventionen in Bezug auf die Qualität des Codes ist obligatorisch. Die Entwurfsdokumentation wird möglichst verständlich strukturiert, vom allgemeinen Überblick zum komplizierten Einzelfall zunehmend detailliert. Zielgruppe der Dokumentation sind nicht in das Projekt involvierte Entwickler, da unser Projekt unter Open-Source-Lizenz steht. Weiterhin nehmen wir an, dass sie zumindest unser Know How besitzen und UML Diagramme, die zur Anwendung kommen werden, verstehen. Das Testkonzept sichert in Verbindung mit der Story-Struktur und dem PairProgramming die Korrektheit des erstellten Programms. Daher wird neben ausführlichen Tests auch auf die Testdokumentation besonderer Wert gelegt.

ii)Vorgehensweisen
Die Einhaltung des Dokumentationskonzepts in Bezug auf quelltextnahe Dokumentation wird im Quelltext selbst stichprobenartig, in Bezug auf javadoc ausführlich von dem Verantwortlichen für Qualitätssicherung und Dokumentation sowie vom Verantwortlichen für Implementierung überprüft. Dies wird regelmäßig nach erfolgreichen Klassentests, sowie am Ende der Implementierung geschehen. Werden Qualitätsanforderungen nicht eingehalten, wird der verantwortliche Implementierer mit Bitte um Nachbesserung darauf hingewiesen.
Die Entwurfsdokumentation wird phasenweise durch Besprechung im Team auf Vollständigkeit und Verständlichkeit diskutiert und gegebenenfalls angepasst.
Die Einhaltung der Festlegungen des Testkonzepts ist von besonderem Wert für den erfolgreichen Abschluss des Projekts. Besondere Aufmerksamkeit erhält die Erstellung der JUnit-Tests und deren Dokumentation. Der Verantwortliche für Tests wird diese überwachen und die finale sowie die wöchentlichen Test-Suites kontrollieren.

iii) Verantwortlichkeiten
Für die quelltextnahe Dokumentation, die Einhaltung der Code-Conventions, sowie die Bereitstellung der javadoc HTML-Dateien sind die jeweiligen Implementierenden selbst verantwortlich. Dabei wird der Quelltext sofort, also während des Programmierens und nicht erst danach kommentiert, die javadoc Dateien werden ständig aktualisiert und allen zugänglich gemacht. Die Kontrolle wird wie im vorhergehenden Punkt beschrieben stattfinden.
Für die Entwurfsdokumentation ist der Verantwortliche für Modellierung verantwortlich. Hierbei werden der Verantwortlichen für Qualitätssicherung und Dokumentation und der Verantwortliche für Implementierung ihn unterstützen.
Für die Testdokumentation ist der Verantwortliche für Tests zuständig und wird wiederum vom Verantwortlichen für Dokumentation unterstützt.