Qualitätssicherungskonzept

MSP-13 – Integration eines Semantischen Tagging Systems in Microsoft Sharepoint

Projektbezeichnung	MSP-13 - Integration eines Semantischen Tagging Systems in Microsoft Sharepoint	
Projektleiter	Martin John	
Verantwortlich	Qualitätssicherung, Dokumentation, Tests	
Erstellt am	17.1.2013	
Zuletzt geändert	10:26 21.1.2013	
Bearbeitungszustand	X	in Bearbeitung vorgelegt fertig gestellt
Dokumentablage		
V-Modell-XT Version	1.4	

Dokumentationskonzept **2**

Testkonzept **4**

Organisatorische Festlegungen **5**

1. Dokumentationskonzept

Eine gute Dokumentation trägt wesentlich zur Qualität eines Softwareproduktes bei. Sie verkürzt die Einarbeitungszeit projektfremder Entwickler in den Quellcode und erleichtert damit die Wartung und Weiterentwicklung der Software. Ohne programmbezogene Dokumentation ist Zweck des Programms nicht problemlos erkennbar.

Dokumentation kann in unterteilt werden in:
- Beschreibung der Benutzerschnittstellen + Anwendung der Teilbereiche der Software
- Benutzerhilfe, Handbücher und Hilfefunktion
- Programmarchitektur und Systemvoraussetzungen
- Quelltextdokumentation

Die Gründe zum Kommentieren sind
- Dokumentation wichtiger Entwurfsentscheidungen, z.B. wieso welcher Datentyp genutzt wurde
- Informationen und Erklärungen zu Programmfragmenten welche nicht aus Quelltext heraus ersichtlich sind
- Verbesserung der optischen Gliederung
- Beschreibung des Einsatzes von Programmiertricks

1.1. Kommentare

Es wird direkt während der Programmierung kommentiert, damit der Quelltext verständlich ist und wichtige Details nicht verloren gehen. Zu Beginn jeder Funktion steht eine kurze Beschreibung was die Funktion macht. Außerdem werden die übergebenen Parameter und die Rückgabewerte festgehalten. Bei Bedarf können weitere Tags angegeben werden.

C# kennt 3 Typen von Kommentaren:

`//`	für einen einzeiligen Kommentar
`/* */`	für einen mehrzeiligen Kommentar
`///`	als Einleitung für einen einzeiligen XML Dokumentationskommentar

Beispiel:

```
1/**
2 * kurze Beschreibung
3 * @param uebergebene Werte
4 * @return return Wert
5 * @TAG
weitere Tags bei Bedarf
6 */
```

Algorithmen und sich nicht selbsterklärende Codeblöcke sind durch Kommentare zu erklären. Beispiel:

```
1/** Beschreibung */
2 Code
```

Variablendeklarationen und Kontrollstrukturen sind zu erläutern. Beispiel:

```
1 Variable , Code
// Erklaerung
```

1.2. Einrückung

Um eine gute Lesbarkeit zu sichern, wird jeder Unterbefehl im Rumpf einer Funktion eingerückt. Einrückungen haben eine Länge von vier Leerzeichen. Öffnende und schließende geschweifte Klammern des Funktionsrumpfes auf separate Zeilen gesetzt und jeder Befehl steht in einer extra Zeile.

Beispiel:

```
1 function functionname ( varX )
2 {
3 .... if ( condition ) // '. ' steht fuer ein Leerzeichen
4     {
5     .... then
6     }
7 }
```

1.3. Bezeichner

Bei der Implementierung wird die Regel der Verbalisierung eingehalten. Es werden also sprechende Namen für Variablen und Funktionen verwendet. Als einheitliche Bezeichnungssprache ist Englisch festgelegt.

Für die Namensgebung gelten außerdem folgende Regeln:
Variablen- und Funktionsbezeichner werden klein geschrieben
Bei zusammengesetzten Bezeichnern wird der Anfangsbuchstabe jedes neuen Wortes groß geschrieben
Konstantenbezeichner bestehen komplett aus Großbuchstaben

1.4. Dokumentation

C# untertsützt eine XML basierte Dokumentation. Es existieren verschiedene Dokumentationstags welche hier nachzulesen sind
http://openbook.galileocomputing.de/csharp/kap31.htm#t2t311
Als Dokumentationssoftware wird Mdoc benutzt

- Mdoc liest die externen XML-Dokumentationsdateien aus, verarbeitet diese weiter (legt z.B. einen Dateibaum an) und gibt diese anschließend als HTML Dateien aus Mdoc existiert für Windows, Mac OS X und Linux
- Möglichkeit die Dokumentation synchron zu halten

1.5. Externe Dokumentation

Die externe Dokumentation ermöglicht es, das Programm unabhängig vom Quellcode zu verstehen. Daher werden eine Designbeschreibung und ein Benutzerhandbuch in deutscher Sprache erstellt und diese im Verlauf des Projektes aktuell gehalten.

2. Testkonzept

Die Komplexität dieses Projekts macht es erforderlich die Implementierungsphase (bzw. Einführungsphase) mit einem umfangreichen Tests zu begleiten. Hierbei sollen mittels Komponententests (unit tests) die korrekte Funktionsweise jeder Methode und Klasse sichergestellt werden. Integrationstests sollen die richtige Zusammenarbeit der einzelnen Komponenten sicherstellen und Systemtests das Verhalten der kompletten Software überwachen. Zum Schluss soll der Auftraggeber der Software durch einen Abnahmetest die korrekte Funktionsweise bestätigen.
Bei der Programmierung mit C# und der damit verbunden Verwendung von Visual Studio (Microsoft) greift man in der Regel auf die integrierte Testumgebung von Visual Studio zurück.

2.1. Komponententests

Mit den folgenden Tools: Test-Explorer, Microsoft-Komponententest-Framework für verwalteten Code, Codeabdeckungstools. Microsoft Fakes-Isolationsframework, stellt Visual Studio ein Reihe Möglichkeiten für ausführliche Komponententests zur Verfügung.
Ziel ist es den Programmierer schnell auf logische Fehler im Quellcode hinzuweisen und Funktionsmerkmale einzelner Komponenten unabhängig vom Gesamtsystem zu testen.
Die angestrebte Automatisierung dieser Tests wir bei Visual Studio über parallele Testklassen mit dem Komponententest-Framework, deren Verwaltung und Ausführung mittels des Test-Explorers und, falls nötig, künstliche Isolation von Komponenten mit Hilfe des Fakes-Isolationsframework sichergestellt.
Dabei können benutzerdefinierte verwendet, oder zufällige Parameter erzeugt werden.

2.2. Integrationstests

In dieser Phase werden insbesondere Funktions- und Schnittstellentests mit den zuvor in Komponententests bestätigten Einzelkomponenten durchgeführt und auf ihre Interaktionsfähigkeit im Softwaresystem untersucht. Integrations- und Systemtest werden einfach über eine Erweiterung eines Testfalls auf mehrere Komponenten im Test-Explorer realisiert.

2.3. Systemtest

In dieser Phase wird das vollständige Produkt in einer Testumgebung die den Anforderungen des späteren Anwendungsgebietes genügen muss, getestet. Hierbei werden nicht mehr einzelne Elemente betrachtet, sondern die Gesamtfunktion des aus den einzelnen Komponenten zusammengesetztes Produkts überprüft, wobei beachtet werden muss, dass dem Benutzer alle vom Pflichtenheft geforderten Funktionen vollständig und fehlerfrei zur Verfügung stehen.

2.4. Abnahmetest

Beim Abnahmetest wird ein letzter Testlauf gemeinsam mit allen Beteiligten durchgeführt. Als Testumgebung sollte möglichst genau die Benutzerumgebung simuliert werden, um Kompatibilitätsprobleme vollständig auszuschließen.
Alle Anforderungen des Pflichtenheftes werden der Reihe nach überprüft und auf Vollständigkeit getestet. Am Ende dieser Phase steht ein fehlerfreies und dem Pflichtenheft genügendes Gesamtprodukt.

3. Organisatorische Festlegungen

Es gibt ein wöchentliches Gruppentreffen, bei dem im Regelfall alle Mitglieder anwesend sind. Dort werden Fortschritte vorgestellt, neue Aufgaben verteilt und auftretende Probleme besprochen. Parallel dazu und verstärkt in der vorlesungsfreien Zeit wird viel über E-mails besprochen. In einem Wiki werden die Beschlüsse und Ansagen aus den Emails festgehalten um die Übersicht zu gewährleisten.

Ein GIT-Repository wurde angelegt, um den Quelltext nachvollziehbar zu machen. Jede abgeschlossene (Teil-)Arbeit am Code muss mit Kommentar eingepflegt werden. Zusätzlich wird die gesamte Dokumentation dort eingepflegt und so jedem zugänglich gemacht.

Mit Hilfe von Diagrammen, wie UML- und Sequenzdiagrammen werden Modellierungsschritte verdeutlicht. Diese sollten besonders übersichtlich und kompakt gehalten werden.

Die in diesem Dokument festgelegten Dokumentationsrichtlinien sind der Standard für den erstellten Programmcode. Dies wird von der Qualitätssicherung vor der Veröffentlichung geprüft.