P. Thiemann, S. Wehr Wintersemester 2005/2006

Software Praktikum

`http://proglang.informatik.uni-freiburg.de/teaching/sopra/2005ws/`

Übungsblatt 3

22. Dezember 2005

1 JUnit

In Aufgabe 3, Blatt 1 hast du eine minimale Repräsentation eines Dateisystems erstellt. Füge der entsprechenden Klasse für Dateien ein Attribute hinzu, welches die Größe der Datei angibt. Schreibe nun analog zu Aufgabe 4, Blatt 1 einen Visitor, der die Summe der Größen aller Dateien eines Verzeichnisses berechnet. (Die Musterlösung zu den Aufgaben von Blatt 1 sind auf der Webseite des Praktikums erhältlich.)

Erstelle nun mit Hilfe von Eclipse einen JUnit Test, der sowohl den in dieser Aufgabe als auch den in Aufgabe 4, Blatt 1 erstellten Visitor testet. Erstelle dazu in der `setUp` Methode des Tests die Repräsentation eines Dateisystems und schreibe zwei Testmethoden `testNameVisitor` und `testSizeVisitor`, welche den eigentlichen Test mit Hilfe von Assertions spezifizieren. Führe den Test mit Hilfe von Eclipse aus.

(Die restlichen Aufgaben sollen in den Gruppen bearbeitet werden, welche für die einzelnen Komponenten von Comes verantwortlich sind.)

2 Dummy Implementierungen

Unter `https://abacus.informatik.uni-freiburg.de/svn/proglang/sopra0506/common/` steht ein Subversion Repository zur Verfügung, das ein Eclipse Projekt mit den Interfaces für Comes enthält. Darüberhinaus enthält das Archiv eine Klasse `de.uni_freiburg.informatik.comes.Factory`, die konkrete Objekte für diese Interfaces erzeugen soll.[1] Im Moment geben allerdings alle Methoden der `Factory` Klasse `null` zurück.

Schreibt nun für die Interfaces, welche nicht in euren Verantwortungsbereich fallen, sogenannte *Dummy Implementierungen*. Das sind Klassen, welche die entsprechenden Interfaces zwar implementieren, aber nur eine sehr stark eingeschränkte Funktionalität zur Verfügung stellen. Zum Beispiel kann man eine Methode dadurch implementieren, dass immer ein konstanter Wert zurückgegeben wird. Beachtet allerdings, dass ihr (zumindest am Anfang) die Dummy Implementierungen benutzen werdet, um Implementierungen eurer Interfaces zu testen. Das bedeutet, dass es nicht ausreicht, in jeder Methode einer Dummy Implementierungen einfach `null` zurückzugeben.

Jetzt könnt ihr die entsprechenden Methoden der `Factory` Klasse vernünftig implementieren. Achtet im weiteren Verlauf des Praktikums darauf, dass ihr Instanzen für Interfaces immer über die `Factory` Klasse erzeugt. Das gibt den anderen Gruppen die Möglichkeit, die Implementierung eines Interfaces einfacher zu ändern.

[1] Die `Factory` Klasse ist ein Beispiel für das *Factory Design Pattern*.

3 Implementierung

Das Eclipse Projekt aus Aufgabe 2 (inklusive der Dummy Implementierungen) soll als Ausgangspunkt für die Implementierung eurer Interfaces dienen. (Die UI Gruppe sollte den Code in ihr Servlet Projekt hineinkopieren.) In dieser Aufgabe sollt ihr mit der eigentlichen Implementierungsarbeit beginnen. Es ist euch prinzipiell freigestellt, welches Vorgehen ihr dafür wählt. Hier sind ein paar Anregungen, wie die einzelnen Gruppen vorgehen könnten:

Gruppe Dokumentenbaum: Implementiert zunächst die Baumstruktur mittels des Composite Patterns. Macht euch dann mit der Java Bibliothek *Jaxen* (`http://jaxen.org`) vertraut. Mit Hilfe dieser Bibliothek kann man in beliebigen Objektstrukturen mittels XPath Ausdrücken navigieren. Um diese Idee auf den Dokumentbaum anwenden zu können, müsst ihr eine Subklasse der abstrakten Klasse `org.jaxen.DefaultNavigator` erzeugen.

Gruppe Persistenz/Versionierung: Statische Dokumente können nicht als solche gespeichert werden, da zusätzliche Informationen (wie z.B. Berechtigungen, MIME-Typ) mitgespeichert werden müssen. Daher sollen statische Dokumente als XML abgespeichert werden. Entwerft eine DTD, die solche XML Dokumente beschreibt. Implementiert dann eine Klasse, die ein entsprechendes XML Dokument einliest und Zugriff auf die darin gespeicherten Informationen bietet. Zum Parsen von XML könnt ihr z.B. die Klasse `javax.xml.parsers.DocumentBuilder` verwenden.

Als nächsten Schritt könnt ihr dann das Laden eines Dokumentenbaums aus dem Dateisystem implementieren.

Gruppe Zugriffskontrolle: Überlegt euch zunächst eine Darstellung der Rollenhierarchie. Mathematisch gesehen ist die Rollenhierarchie eine Halbordnung $\leq$ (reflexiv, transitiv, antisymmetrisch). Für zwei Rollen $r_1 \leq r_2$ gilt, dass r_1 alle Berechtigungen von r_2 erbt, und dass jedes Mitglied von r_1 auch Mitglied von r_2 ist. Eine Halbordnung wird üblicherweise durch gerichtete, azyklische Graphen (englisch: directed azyclic graph, daher wird als Abkürzung oft DAG verwendet) dargestellt. Es ist also sinnvoll, die Rollenhierarchie ebenfalls als einen DAG zu implementieren.

Als nächstes solltet ihr euch überlegen, was beim Hinzufügen und Entfernen einer Berechtigung für eine Rolle r mit den Berechtigungen für Rollen r', r'' mit $r' \leq r$ und $r \leq r''$ geschieht. Nun könnt ihr eine Methode implementieren, die den Dokumentenbaum von der Wurzel bis zu einem Knoten n durchläuft und für alle vorhandenen Rollen die Berechtigungen für n berechnet.

Gruppe UI: Macht euch zunächst mit dem BEA Applikationsserver vertraut. Danach könnt ihr die Funktionalität zum Anzeigen einer Seite implementieren. Desweiteren solltet ihr die Benutzerschnittstelle zum Editieren eines Dokuments programmieren.

Abgabe: Mittwoch, 11.1.2006.
Aufgabe 1 soll in der Besprechung in Papierform abgegeben werden. Der Code zu Aufgabe 2 und 3 soll ins Subversion Repository eurer Gruppe eingecheckt und in der Besprechung kurz vorgestellt werden.