Einführung in Software Engineering WS 10/11

Fachbereich Informatik

Dr. Michael Eichberg
eichberg@informatik.tu-darmstadt.de

Assistent: Ralf Mitschke
mitschke@st.informatik.tu-darmstadt.de

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 10 (10 Punkte): Design Patterns

Abgabeformat: Reichen Sie ihre Lösung per SVN ein. Jede Übung muss in einem eigenen Ordner ***ex<Number>*** *(<Number> = 01, 02, ...)* in Ihrem Gruppenverzeichnis eingereicht werden. Während der Übungsbearbeitung können Sie Ihre Lösungen beliebig oft in das SVN hochladen (per Commit). Wir prüfen die Zeit der Einreichung Ihrer Lösungen unter der Benutzung des SVN Zeitstempels.

Erstellen Sie für Lösungen der Aufgaben, die keinen Quelltext erfordern, eine PDF-Datei mit dem Dateinamen ***solution.pdf***. Dies gilt auch für alle UML-Diagramme, die Sie erstellen. Die Basisanwendung wird als Eclipse-Projekt vorgegeben. Ihr eigener Code muss entsprechend in den dafür vorgesehenen Verzeichnissen (***/src*** oder ***/test***) erstellt werden.

Abgabetermin: 02.02.2011 - 24:00 Uhr

Aufgabe 1 (3 Punkte)

Ziel: *Verwendung von Design Patterns im JDK*

a) `java.rmi.server.RMIClientSocketFactory` (1 Punkt)

Studieren Sie die Verwendung von Design Patterns in der Klasse: `java.rmi.server.RMIClientSocketFactory` des JDKs.

Welche Rolle hat diese Klasse im Rahmen der Implementierung welches Patterns inne? Ziehen Sie bei Bedarf weitere Klassen in Betracht mit denen diese Klasse in Beziehung steht.

Hinweis: Folgender Link ist beim Verständnis der Zusammenhänge ggf. hilfreich:
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/socketfactory/index.html

```
/**
 * An RMIClientSocketFactory instance is used by the RMI runtime in order to
 * obtain client sockets for RMI calls.  A remote object can be associated
 * with an RMIClientSocketFactory when it is created/exported via the constructors
 * or exportObject methods of java.rmi.server.UnicastRemoteObject and
 * java.rmi.activation.Activatable.
 *
 * An RMIClientSocketFactory instance associated with a remote object will be
 * downloaded to clients when the remote object's reference is transmitted in an
 * RMI call.  This RMIClientSocketFactory will be used to create connections to
 * the remote object for remote method calls.
 *
 * An RMIClientSocketFactory instance can also be associated with a remote object
 * registry so that clients can use custom socket communication with a remote
 * object registry.
 *
 * [...]
*/
public interface RMIClientSocketFactory {

    /**
     * Create a client socket connected to the specified host and port.
     */
   public Socket createSocket(String host, int port)
     throws IOException;
}
```

b) `java.util.AbstractList.Itr` (1 Punkt)

Studieren Sie die Implementierung des Iterator Patterns in der Klasse `java.util.AbstractList` des JDKs. Der Iterator wird in der inneren Klasse `Itr` implementiert. Der Code der Klasse ist im Folgenden Auszugsweise angegeben.

Handelt es sich um einen „robusten“ Iterator? Diskutieren Sie was passiert wenn sich die zugrunde liegenden Daten ändern?

```
private class Itr implements Iterator<E> {
    /**
     * Index of element to be returned by subsequent call to next.
     */
    int cursor = 0;

    /**
     * Index of element returned by most recent call to next or
     * previous.  Reset to -1 if this element is deleted by a call
     * to remove.
     */
    int lastRet = -1;

    /**
     * The modCount value that the iterator believes that the backing
     * List should have.  If this expectation is violated, the iterator
     * has detected concurrent modification.
     */
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor != size();
    }

    public E next() {
        checkForComodification();
        try {
          E next = get(cursor);
          lastRet = cursor++;
          return next;
        } catch (IndexOutOfBoundsException e) {
          checkForComodification();
          throw new NoSuchElementException();
        }
    }

[...]

    final void checkForComodification() {
        if (modCount != expectedModCount)
          throw new ConcurrentModificationException();
    }
}
```

c) `java.lang.Runtime` (1 Punkt)

Studieren Sie die Implementierung der Klasse `java.lang.Runtime`. Welche(s) Pattern(s) erkennen Sie? Geben Sie genau an welche Klasse(n) welche Rolle(n) inne hat/haben (insbesondere mit Blick auf die Klasse `java.lang.Runtime`). Geben Sie weiterhin an wie welche Methoden der Klasse auf welche Methoden des von Ihnen identifizierten Patterns abgebildet werden können.

Aufgabe 2 (3 Punkte)

Ziel: *Design Patterns diskutieren*

a) Singletons und Testen (1 Punkte)

Erläutern Sie kurz, warum „Singletons“ die Testbarkeit einer Anwendung behindern (können)?

b) Factory Method und Abstract Factory (2 Punkte)

Abb.1 zeigt mehrere zusammengehörige Vererbungshierarchien der Implementierung eines WindowManager-Frameworks (ähnlich den UNIX Window Managern). Die Oberklassen bieten ein allgemeines Interface und sind öffentlich, so dass von jedem Interessierten eine eigene Implementierung gegeben werden kann. Konkrete `WindowManager` sind z.B. `FVWM` und `SawFish`. Die konkreten Subklassen des `Window` und andere Bestandteile der GUI (z.B. `Scrollbar` etc.) müssen dabei immer konsistent genutzt werden. Es darf z.B. keine `FVWMWindow` mit einer `SawFishScrollbar` genutzt werden.

Als Designer des Frameworks (der Oberklassen) möchten Sie die Konsistenz der Klassen sicherstellen. Erläutern Sie, wie Sie diese Anforderung mit Hilfe des Factory Method Patterns und/oder mit Hilfe des Abstract Factory Patterns umsetzten würden. Diskutieren Sie in Ihrer Erläuterung welche Klassen und Methoden Sie einführen würden und wo diese platziert sind. Welche Klasse hat welche Rolle des jeweiligen Patterns inne? Welche Klasse hat in Ihrem Design die Verantwortlichkeit welche Objekte zu erstellen? Diskutieren Sie die Erweiterbarkeit Ihres Designs im Hinblick auf neue GUI Elemente und im Hinblick auf neue WindowManager. Erläutern Sie hierzu welche Klassen angepasst werden müssen, bzw. welche Verantwortlichkeiten von neu hinzugefügten Klassen implementiert werden müssen.

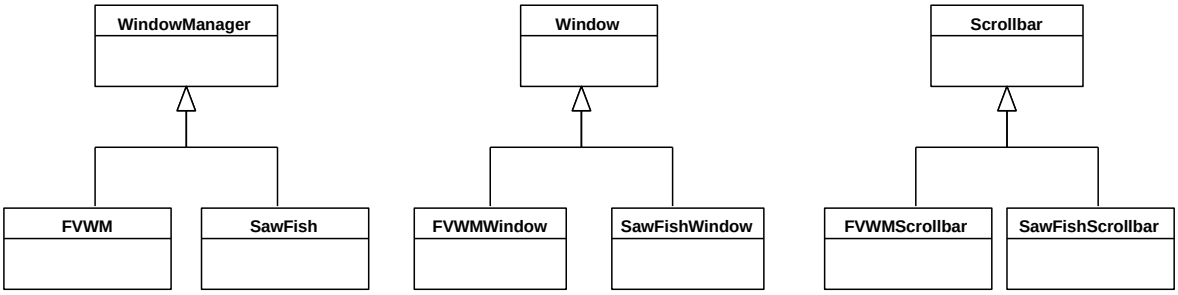

Abb. 1 Klassen eines WindowManager-Frameworks mit verschiedenen Implementierungen

Aufgabe 3 (4 Punkte)

Ziel: *Übung im Umgang mit Design Patterns*

In der letzten Übung wurden verschiedene Lernstrategien implementiert.

a) Erläuterung des bestehenden Designs (0,5 Punkte)

Erklären Sie kurz welche Methode (in welcher Klasse) in Ihrem Design die Verantwortlichkeit hat eine Lernstrategie auszuwählen und daraufhin die entsprechende konkrete Lernstrategie zu instanziieren.

b) Anwendung des Factory Method Patterns (2,5 Punkte)

Eine Möglichkeit die Verantwortung zur Auswahl der Strategie zu platzieren ist die Methode `FlashcardsWindow.learn()`. Dies hat zur Folge, dass immer wenn eine Lernstrategie ergänzt wird auch das `FlashcardsWindow` angepasst werden muss.

Nutzen Sie das Factory Method Pattern, um - im Falle von Änderungen - nur noch an einer Stelle innerhalb des Domänenmodels Lernstrategien hinzuzufügen. Innerhalb der GUI sollte danach nur noch ein Aufruf an die Factory Methode stattfinden.

Hinweis: *Um eine Liste aller vorhandenen Lernstrategien zu präsentieren, sollte die GUI den gesamten Umfang aller Strategien auch aus dem Domänenmodell erfragen.*

c) Dokumentation des Factory Method Patterns (1 Punkt)

Erstellen Sie ein UML Klassendiagramm das Ihre Implementierung des Factory Method Patterns dokumentiert. Die Klassen sollten alle für das Pattern relevanten Methoden zeigen. Weitere Methoden sind nicht aufzuführen. Notieren Sie welche Klassen welche Rollen des Patterns inne haben (entweder im Diagramm, cf. „Design Pattern – Introduction“ Folie 27, oder extern in Ihrem Lösungsdokument).