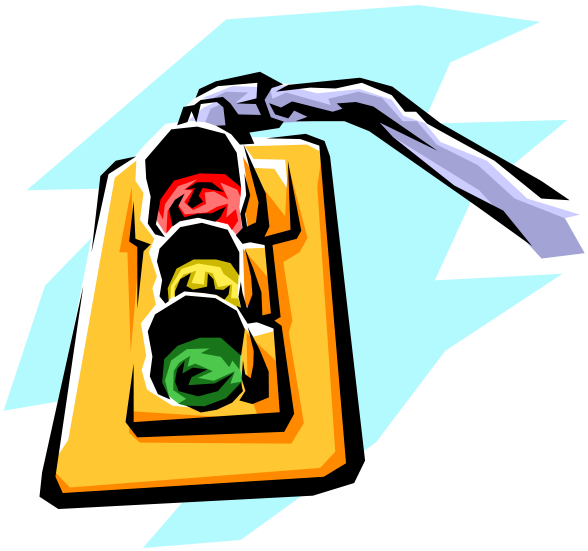
Softwarepraktikum

Teil: Eingebettete Systeme

Sommersemester 2003

Implementierung I:

Struktur



Aufgabe 3

Implementierung I: Struktur

Umfang: **1 Woche**
Punkte: **50 P.**

In den ersten beiden Aufgaben wurden die Struktur und das Verhalten des Systems *modelliert*. Ausgehend von diesen Modellen soll nun in den folgenden Aufgaben die Ampelsteuerung *implementiert* werden.

Den Anfang macht diese Aufgabe, in welcher zunächst die strukturellen Modelle (Klassen- und Instanzendiagramme) herangezogen werden, um die strukturellen Aspekte des Systems zu implementieren. Dazu gehört die Abbildung der Modellklassen auf Klassen der Programmiersprache Java, aber auch die Realisierung der Relationen zwischen diesen Klassen.

1 Umsetzung der Klassen- und Instanzendiagramme

Der erste Schritt in der Umsetzung der strukturellen Modelle in ausführbaren Java-Code bildet die Realisierung von Java-**Klassenrahmen**, d.h. von reinen Klassendefinitionen ohne Methodendefinitionen. Diese Klassenrahmen, welche dann in den darauffol-

genden Schritten erweitert werden, können aus dem UML-Klassendiagramm aus der ersten Aufgabe abgeleitet werden.

Ein wichtiger Aspekt bei dieser Umsetzung ist die Berücksichtigung aktiver Klassen. In der Modellierung waren wir davon ausgegangen, dass alle Klassen aktive Klassen sind. Bei der Umsetzung auf Java-Code bedeutet dies, dass jede dieser Klassen einen unabhängigen Kontrollfluss besitzen muss, d.h. dass jede Instanz einer Klasse durch einen Java-Thread realisiert wird[1]. Diese vollständig „parallele“ Realisierung ist unter Umständen aber nicht nötig und kann auch sehr ineffizient werden (wenn die Zahl der aktiven Threads zu groß wird). Daher sollten man sich genau überlegen, für welche Klassen die Realisierung durch einen Thread benötigt wird.

Im Anschluss an die Realisierung der Klassenrahmen lässt sich mit den Instanzendiagrammen die Frage klären: „Wer ist für die Erzeugung einer Instanz verantwortlich?“ Der Code für solche Instanziierungen sollte schon (wenn möglich) bei der Implementierung der Struktur erstellt werden, s.d. die Ausführung des Codes die gewünschte Zahl von Instanzen liefert.

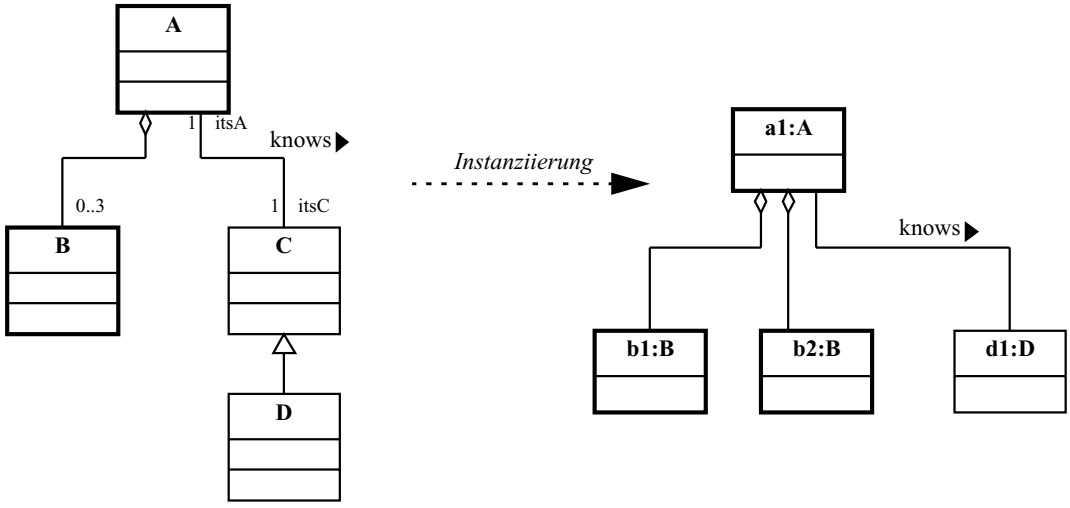


Abbildung 16 Beispielhafte Klassen- und Instanzendiagramme

1. Dies soll bei der Implementierung dadurch gekennzeichnet werden, dass die betroffene Java-Klasse das `java.lang.Runnable` Interface implementiert.

Um die obigen Schritte zu verdeutlichen, zunächst einmal ein Beispiel: Gegeben seien die Klassen- und Instanzendiagramme in Abbildung 16 (auf die Realisierung der Relationen wird in Abschnitt 1.1 eingegangen). Wie in der Abbildung gekennzeichnet, handelt es sich bei Klassen A und B um aktive Klassen, welche also das `Runnable` Interface implementieren müssen. Zur Implementierung des `Runnable` Interfaces muss die Methode `run()` definiert werden. Wir erhalten zunächst also den folgenden Code für die Klasse A (analoges gilt für die Klasse B):

```
public class A implements Runnable {

    public A () {
    }

    public void run() {
    }
}
```

Angenommen, die Instanz `a1` würde in der `main()`-Methode instanziiert, dann bietet es sich an die anderen Instanzen, die in irgendeiner Weise über Relationen von A abhängen, im Konstruktur der Klasse A zu instanziieren. Damit ergibt sich der folgende Code:

```
public class A implements Runnable {

    public A () {
      B b1 = new B();
      B b2 = new B();
      D d1 = new D();
    }

    public void run() {
    }
}
```

1.1 Abbildung von Relationen

Im Anschluss an die obigen Schritte folgt nun die Abbildung der Relationen, wie sie in den Klassendiagrammen modelliert sind, auf Code.

Die Generalisierungsrelation (Vererbung) lässt sich dabei direkt auf ein Java-Konstrukt abbilden. Mit der Verwendung des `extends` Keywords erlaubt Java nämlich die Vererbung zwischen Klassen zu beschreiben.

Für obiges Beispiel ergibt sich somit für die Klasse D die folgende Klassendeklaration:

```
public class D extends C { ...
```

Die Assoziation und die Aggregation lassen sich auf ein einheitliches Konzept zurückführen, welches im Folgenden beschrieben wird.

Prinzipiell werden für die Realisierung dieser Relationen sogenannte **Referenzattribute** eingeführt, die eine Referenz auf die Instanz (oder Instanzen) auf der anderen Seite der Relation beinhalten. Wichtig ist in diesem Zusammenhang zu unterscheiden, ob eine Relation **gerichtet** ist (d.h. die Relation nur in einer Richtung verfolgt werden kann) oder ob es sich um eine **bidirektionale** Relation handelt. Ist letzteres der Fall, so muss gewährleistet werden, dass wenn das Referenzattribut der einer Instanz, die an der Relation beiteiligt ist, belegt wird, auch das entsprechende Referenzattribut der Instanz auf der anderen Seite der Relation belegt wird. Im Beispiel aus Abbildung 16 bedeutet dies, dass wenn `a1` die Instanz `d1` kennt (`knows`) auch `d1` die Instanz `a1` kennen muss. Die folgenden beiden Code-Fragmente zeigen, wie man solch bidirektionale Relationen systematisch implementieren kann. Zunächst Klasse A:

```
class A {

    protected C itsC;

    public C getItsC() {
        return itsC;
    }

    public void __setItsC(C p_C) {
        itsC = p_C;
    }

    public void _setItsC(C p_C) {
        if(itsC != null)
            itsC.__setItsA(null);
        __setItsC(p_C);
    }

    public void setItsC(C p_C) {
        if(p_C != null)
            p_C._setItsA(this);
        _setItsC(p_C);
    }

    public void _clearItsC() {
        itsC = null;
    }
```

```
}
```

Und hier für Klasse C (symmetrisch zu Klasse A):

```
class C {

    protected A itsA;

    public A getItsA() {
        return itsA;
    }

    public void __setItsA(A p_A) {
        itsA = p_A;
    }

    public void _setItsA(A p_A) {
        if(itsA != null)
            itsA.__setItsC(null);
        __setItsA(p_A);
    }

    public void setItsA(A p_A) {
        if(p_A != null)
            p_A._setItsC(this);
        _setItsA(p_A);
    }

    public void _clearItsA() {
        itsA = null;
    }

}
```

Wie man sieht, ist die Realisierung bidirektionaler Relationen nicht ganz einfach, wenn man – wie oben – auch das Ändern von Relationen zur Laufzeit berücksichtigt. Da häufig sowieso gerichtete Relationen ausreichen, sollte man sich an dieser Stelle nochmals Gedanken machen, wie die Relationen genutzt werden und sich evtl. für eine gerichtete Relation entscheiden.

Im obigen Beispiel war schon gezeigt, wie ein Referenzattribut für eine Relation mit der Multiplizität 1 aussehen kann – man verwendet ein ganz normales Attribut. Für die Abbildung höherer Multiplizitäten gibt es verschiedene Möglichkeiten, die in der folgenden Tabelle 1 nach den Eigenschaften der Multiplizitäten gegliedert sind.

Eigenschaft der Multiplizität	**Realisierung**
fest: 0..1	explizites Attribut, z.B. `C itsC;`
fest:0.. n (n > 1)	Array, z.B. `B itsB[3];`

Tabelle 1 Abbildung von Multiplizitäten

Eigenschaft der Multiplizität	Realisierung
dynamisch	Liste, z.B. `Vector itsB = new Vector();`

Tabelle 1 Abbildung von Multiplizitäten

1.2 Aufgabe (50 Punkte)

Erstellen Sie ausgehend von Ihren strukturellen Modellen den Java-Code für Ihr System, der die Code-Rahmen nach obigem Schema, die Instanziierung der Klassen und die Abbildung der Relationen (und wenn möglich deren initiale Belegung) beinhaltet.

1.3 Abgabe

Abzugeben sind:

- der aussagekräftig dokumentierte Java-Code
- die generierte Dokumentation (javadoc)