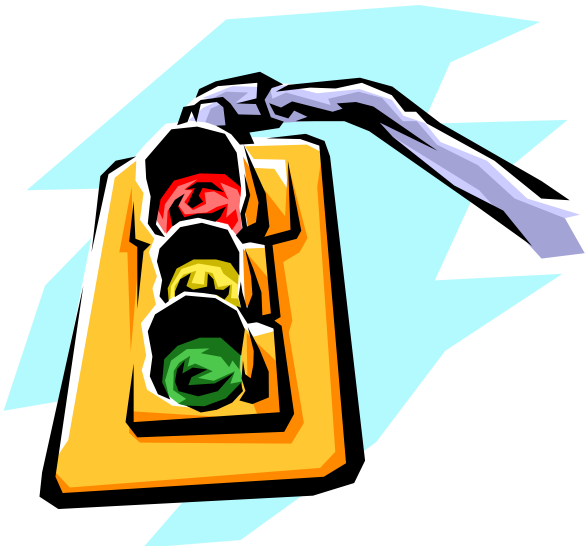
Softwarepraktikum

Teil: Eingebettete Systeme

Sommersemester 2003

Implementierung II:

Verhalten (Teil 1)



Aufgabe 4

Implementierung II: Verhalten (Teil 1)

Umfang: **1 Woche**

Punkte: **50 P.**

In den ersten beiden Aufgaben wurde die Modellierung des Systems durchgeführt. Die entstandenen Dokumente (Klassen-, Instanzen- und Zustandsdiagramme) dienen der anschließenden Implementierung als Ausgangspunkt. Dabei muss die Verhaltensbeschreibung mittels Zustandsdiagrammen nicht unbedingt in der Implementierung beibehalten werden (weil diese vielleicht zu ineffizient ist), solange die Implementation der Spezifikation (den Modellen) genügt.

Die Abbildung der Struktur, die in den Klassen-/Objektdiagrammen festgehalten ist, auf Java-Kode wurde in der letzten Aufgabe bereits durchgeführt. Nun folgt die Implementierung des Verhaltens. Eine mögliche Vorgehensweise ist dabei die Transformation der Zustandsdiagramme in Java-Kode. Dazu existieren verschiedene Möglichkeiten. Im Folgenden wird ein objektorientierter Ansatz propagiert. Es existieren aber genauso Lösungen, die mit prozeduralen Sprachkonstrukten auskommen. Ein typischer prozeduraler Ansatz ist die Verwendung des Switch-Konstrukts und Variablen, die den aktuellen Zustand speichern.

Bevor wir mit der Abbildung der Zustandsdiagramme beginnen können, stellen wir die Implementierung der Ereignisse (Events oder Signale) vor und beschreiben die Event-Handling-Mechanismen.

1 Konzepte

1.1 Events und Event-Handling

Die Ereignisse, die in einem Zustandsdiagramm zum Wechsel in einen Folgezustand führen, werden in Java als sog. Events realisiert. Zunächst existieren drei Typen von Events in unserem System: Timer-[1], Detektor- und Fehler-Events. Später kommen noch selbstdefinierte Events hinzu, um die Kommunikation zwischen den Zustandsdiagrammen der aktiven Klassen (bzw. Instanzen) zu realisieren.

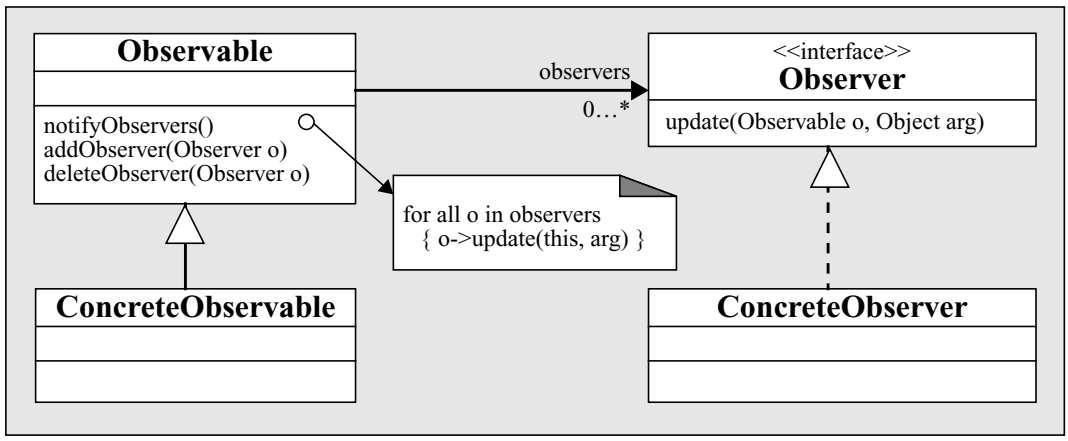


Abbildung 17 Observer Pattern

4.1.1.1 Observer-Pattern

Der Mechanismus der Events steht in Java seit JDK 1.1 zur Verfügung. Das Konzept basiert auf sog. Event-Sources, die Events auslösen und Event-Listenern, die auf solche Events (asynchron)

1. In den Zustandsdiagrammen wurden die Timer als Bedingung (z.B. [10s]) modelliert. Dies wird in Java nun auf Timer-Events abgebildet. Das ist ganz analog zu der Darstellung des Timers als separates Objekt in Abbildung 8.

reagieren. Zu Grunde liegt dabei das sogenannte *Observer-Pattern* [ObPattern]. Es wird verwendet, um Abhängigkeiten zwischen Objekten zu verwalten und um Änderungen mitzuteilen. Kern des Patterns ist ein `Observer`, der Änderungen eines `Observable`s überwachen will. Die Struktur sieht dabei folgendermaßen aus (Abbildung 17):

Um Änderungen zu Überwachen, muss sich der Observer bei seinem Observable mit der Methode `addObserver()` registrieren. Jetzt wird bei jeder Änderung des Observables automatisch die `update()`-Methode des Observers aufgerufen (s. Abbildung 18)[1]:

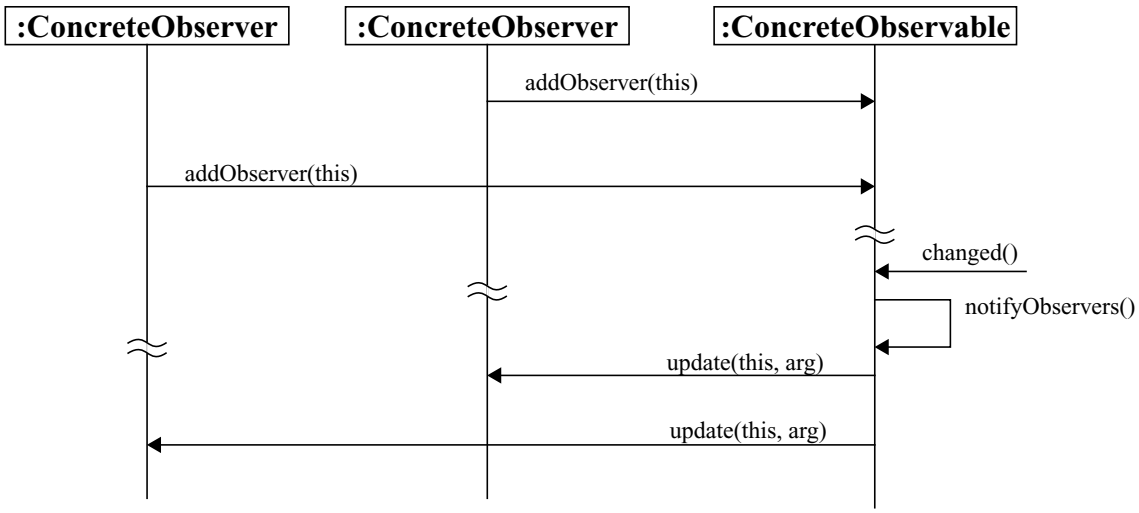


Abbildung 18 Ablauf des Observers

Die Klasse `Observable` und das Interface `Observer` stehen bereits in Java zur verfügung [APIObs].

Der Event-Mechanismus von Java baut nun auf diesem Observer-Prinzip auf. Event-Listener (Observer) registrieren sich bei der Klasse, die den Event auslöst (Event-Source, Observable) mit einer

1. Dieses Sequenzdiagramm zeigt eine Abstraktion des realen Verhaltens insofern, dass die Methodenaufrufe durch Signale beschrieben werden. Bei Methodenaufrufen kommt zu dem „aufrufenden Pfeil“ auch immer noch ein „Rückgabe-Pfeil“ hinzu, der das Ende des Methodenaufrufes kennzeichnet.

`add...Listener()`-Methode. Wird das entsprechende Ereignis ausgelöst, werden automatisch alle Listener benachrichtigt. Dabei wird als Argument das auslösende Event übergeben. In [Event], [EHandler] findet man weitere Details zu den Events.

4.1.1.2 Das NamedEventObject

Alle Events, die wir in diesem Praktikum verwenden, sollen von `environment.event.NamedEventObject` (siehe Abbildung 19 und [EventDoc]) abgeleitet werden. Zusätzlich zur Quelle des Events wird in diesem Objekt noch der Name der auslösenden Instanz mitgegeben. Für DetektorEvents ist das der Name des Detektors wie er im System bekannt ist. Neben dem DetektorEvent sind auch Timer-Events, FehlerEvents und eventuell vom Programmierer definierte Events von NamedEventObject abgeleitet. Um den Namen der Event-Quelle zu bekommen, verwendet man die `getName()`-Methode.

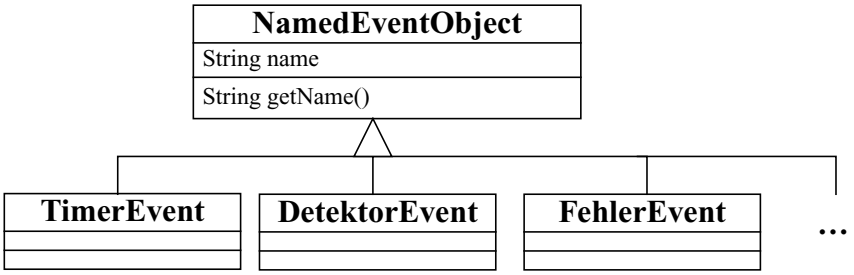


Abbildung 19 Klassendiagramm der verwendeten Events

4.1.1.3 TimerEvent

Viele der Zustandsübergänge werden von Timer-Ereignissen ausgelöst. Mit Hilfe der vorgegebenen Klasse `environment.timer.Timer` können Sie solche Events erzeugen. Dabei müssen Sie zunächst ein Timer-Objekt instanziieren (z.B. `new Timer("T1")`), die entsprechende TimerListener-Klasse registrieren (`addTimerListener()`) und dann den Timer „aufziehen“ (`setTimeout()`).

Beispiel:

```
import environment.timer.*;
```

```
import environment.event.*;

class TimerBeispiel implements TimerListener {
    Timer t1;

    /** creates a new Timer, adds “this” as TimerListener and
     * starts the Timer with a timeout of 3 seconds.
     */
    TimerBeispiel() {
        t1=new Timer("T1");                    // neuer Timer
        t1.addTimerListener(this);             // Listener registrieren
        t1.setTimeout(3000);                   // Timer starten, 3 s. Timeout
    }

    /** is called when the timer expires.
     * @param e the TimerEvent that was issued
     */
    public void timerExpired(TimerEvent e) {
        System.out.println("Hallo Welt");
        t1.setTimeout(3000);                   // Timer neu starten
    }

    public static void main(String args[]) {
        TimerBeispiel bsp=new TimerBeispiel();
    }
}
```

Weitere Details finden Sie in [TimerDoc].

Bemerkung: In Packages `javax.swing` und `java.util` existieren ebenfalls Timer-Klassen. Prinzipiell hätte man auch eine dieser Klassen verwenden können. Allerdings erzeugen diese keine `NamedEventObject` und sollten deshalb nicht verwendet werden.

4.1.1.4 DetektorEvent

Genau wie das Timer-Event können auch die Detektor-Ereignisse verwendet werden. Hierzu existiert die DetektorListener-Klasse und entsprechend die DetektorEvent-Klasse. Detektor-Ereignisse werden von Komponenten in der Umgebung des Eingebetteten Systems ausgelöst (in unserem Fall vom Simulator generiert) und über die Schnittstelle ausgetauscht. Diese wird in der nächsten Aufgabe vorgestellt.

4.1.1.5 FehlerEvent

Fehler-Events werden verwendet, um Fehlfunktionen von Lichtsignalanlagen mitzuteilen. Ebenso wie die DetektorEvents werden die FehlerEvents von der Umgebung ausgelöst. Alle Komponenten, die auf Fehler reagieren sollen, müssen sich also als FehlerListener bei der Umgebung registrieren.

4.1.1.6 Vom Programmierer definierte Events

Für die Kommunikation zwischen den einzelnen Steuerprozessen der Signalanlagen können Sie sich weitere Events, die von NamedEvent-Object abgeleitet sind, definieren. Sollen zwei Zustandsdiagramme miteinander kommunizieren (realisiet durch zwei Java-Klassen), muss sich die eine bei der anderen registrieren. Beim Senden des Events wird dann die gewählte Methode in der Listener-Klasse aufgerufen. Für die korrekte Implementierung des Observables („Sender“ des Signals) wird auf [ObImpl] verwiesen.

1.2 Vom Zustandsdiagramm zum Java-Kode

Bei der Ausführung von Zustandsdiagrammen ergibt sich abhängig vom aktuellen Zustand ein unterschiedliches Verhalten. Eine direkte Umsetzung der Zustandsdiagramme führt sehr schnell zu verschachtelten `case` oder `if-then` Konstrukten. Hauptnachteile dieser Lösung sind vor allem Unübersichtlichkeit, schlechte Erweiterbarkeit und Probleme bei asynchronen Events. Dank der Objektorientierung können Zustandsdiagramme auch eleganter implementiert werden. Wir wollen im Praktikum das State Pattern [StPattern] zur Umsetzung verwenden.

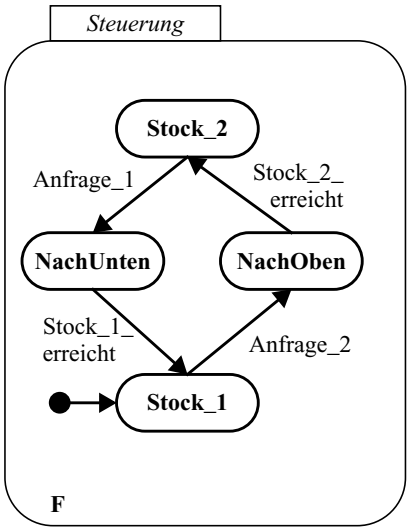


Abbildung 20 Einfaches Zustandsdiagramm zur Fahrstuhlsteuerung

Jeder Zustand eines Zustandsdiagrammes wird dabei auf eine eigene Instanz einer Klasse abgebildet. Alle Zustände sind dabei von einer gemeinsamen Oberklasse abgeleitet. Das Zustandsdiagramm merkt sich jeweils den aktuellen Zustand und gibt eingehende Events an diesen weiter.

Am Beispiel einer einfachen Fahrstuhlsteuerung (Abbildung 20) soll dieses Konzept nun verdeutlicht werden. Dazu zunächst einmal ein Klassendiagramm (s. Abbildung 21).

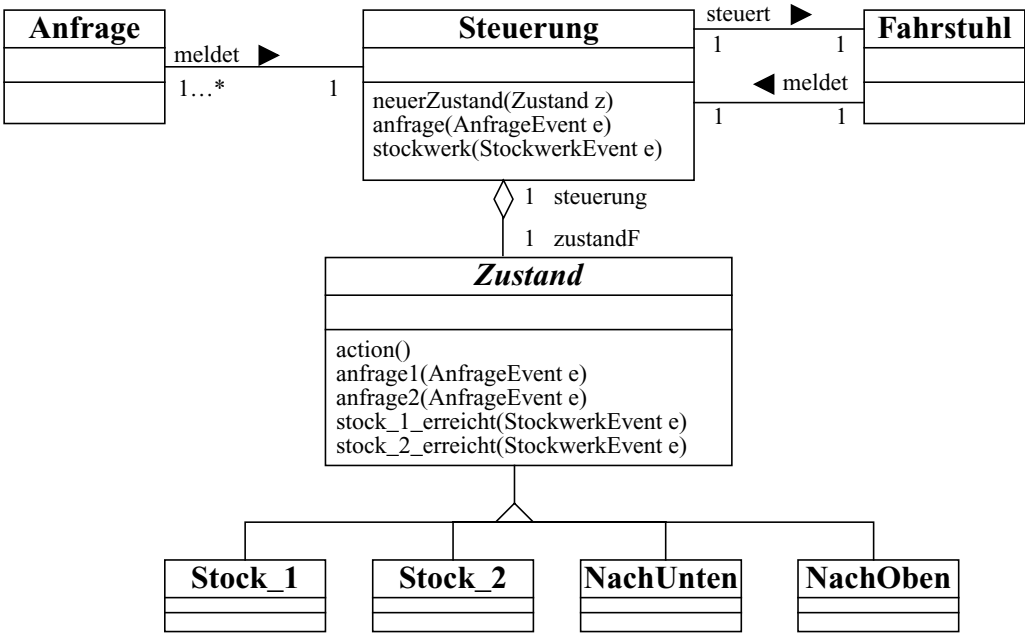


Abbildung 21 Klassendiagramm für Implementierung eines Zustandsdiagramms

Die Klasse `Steuerung` verwaltet den aktuellen Zustand und gibt Events in Form von Methodenaufrufen an diesen weiter (Methoden `anfrage1/2()` und `stock_1/2_erreicht()`). Die Implementierung sieht wie folgt aus:

```
public class Steuerung implements
    FahrstuhlListener, AnfrageListener {
    private Zustand zustandF;                    //    aktueller Zustand
    public final Zustand stock1;                 //    mögliche Zustände
    public final Zustand stock2;
    public final Zustand nachUnten;
    public final Zustand nachOben;
    public Anfrage anfrage1;                     //    Umgebung
    public Anfrage anfrage2;
    public Fahrstuhl fahrstuhl;

    public LSA1() {
            stock1=new Stock1(this);             //    Zustaende anlegen
```

```
            stock2=new Stock2(this);
            nachUnten=new NachUnten(this);
            nachOben=new NachOben(this);
            anfrage1=new Anfrage("1");          //  Listener registrieren
            anfrage1.addAnfrageListener(this);
            anfrage2=new Anfrage("2");
            anfrage2.addAnfrageListener(this);
            fahrstuhl=new Fahrstuhl();
            fahrstuhl.addStockwerkListener(this);
            zustandF=stock1;                    //  Anfangszustand
            zustandF.action();                  //  Aktion ausfuehren
    }

    public Zustand getZustandF() {
            return zustandF;
    }

    public void anfrage(AnfrageEvent e) {
            if (e.getName().compareTo("1") == 0)
                    zustandF.anfrage1(e);
            if (e.getName().compareTo("2") == 0)
                    zustandF.anfrage2(e);
      }

    public void stockwerk(StockwerkEvent e) {
            if (e.getName().compareTo("Stock_1") == 0)
                    zustandF.stock_1_erreicht(e);
            if (e.getName().compareTo("Stock_2") == 0)
                    zustandF.stock_2_erreicht(e);
    }

    public synchronized void neuerZustandF(Zustand z) {
            zustandF = z;
            zustandF.action();
    }
}
```

Initial befindet sich `Steuerung` im Zustand Stock1 (Startzustand in Abbildung 20). Tritt ein Anfrage- oder ein Stockwerk-Ereignis auf, wird die entsprechende Methode des aktuellen Zustandsobjektes aufgerufen und diese bestimmt dann den Folgezustand. Die Methode `neuerZustandF()` dient dazu, einen Zustandswechsel auszuführen. Direkt nach dem Zustandswechsel wird die in dem Zustand spezifizierte Aktion ausgeführt. Diese Methode wird hier als `synchronized` definiert. Dieses Schlüsselwort bedeutet, dass immer nur ein Thread zur selben Zeit die Methode ausführen darf. Ansonsten könnte es passieren, dass während der Abarbeitung dieser Methode ein zweiter Thread, z. B. auf Grund eines TimerEvents, einen Zustandswechsel durchführen will und auch `neuerZustandF()` aufruft. Dann könnte die Aktion `zustandF=z;` des ersten Threads schon ausgeführt sein, ohne dass die zugehörige Aktion (`zustandF.action();`) durchgeführt wurde. Werden sämtliche Zustandswechsel nur über eine Methode abgewickelt,

erhält man mit dem `synchronized`-Schlüsselwort einen einfachen Mechanismus zur Prozesssynchronisation.

Da die Klasse `Steuerung` sowohl von `AnfrageListener` als auch von `StockwerkListener` erbt, kann sie direkt (`this`) als Listener registriert werden (z.B. in `addAnfrageListener(this)`).

Die Oberklasse `Zustand` wird wie folgt implementiert:

```
public abstract class Zustand {
    protected Steuerung steuerung;
    public Zustand(Steuerung s) {
            steuerung = s;
    }

    public void action() {}
    public void anfrage1(AnfrageEvent e) {}
    public void anfrage2(AnfrageEvent e) {}
    public void stock_1_erreicht(StockwerkEvent e) {}
    public void stock_2_erreicht(StockwerkEvent e) {}
}
```

Diese Klasse wird als `abstract` definiert, da von ihr keine Objekte direkt instanziiert werden sollen. `Zustand` liefert Defaults für alle möglichen Events und für die bei einem Zustandswechsel auszuführende Aktion. Die Vorgabe hier ist, jeweils nichts zu tun.

Von dieser Klasse leiten wir vier Zustandsklassen ab. In der Methode `action()` steht die Ausgabe, die in diesem Zustand erfolgt, `anfrage1/2()` und `stock_1/2_erreicht()` werden aufgerufen, wenn das entsprechende Event eintrifft und wechseln möglicherweise den Zustand.

```
public class Stock_1 extends Zustand {
    public Stock_1(Steuerung s) {super(s);}

    public void action() {
            steuerung.stopMotor();                  //      Fahrstuhlmotor anhalten
}

    public void anfrage2(AnfrageEvent e) {
            steuerung.neuerZustandF(steuerung.nachOben);
    }
}

public class Stock_2 extends Zustand {
    public Stock_2(Steuerung s) {super(s);}

    public void action() {
            steuerung.stopMotor();                  //      Fahrstuhlmotor anhalten
}

    public void anfrage1(AnfrageEvent e) {
            steuerung.neuerZustandF(steuerung.nachUnten);
    }
```

```
}

public class NachOben extends Zustand {
   public NachOben(Steuerung s) {super(s);}

   public void action() {
          steuerung.motorHoch();              //     Fahrstuhlmotor nach oben
}

   public void stock_2_erreicht(StockwerkEvent e) {
          steuerung.neuerZustandF(steuerung.stock2);
   }
}

public class NachUnten extends Zustand {
   public NachUnten(Steuerung s) {super(s);}

   public void action() {
          steuerung.motorRunter();            //     Fahrstuhlmotor nach unten
}

   public void stock_1_erreicht(StockwerkEvent e) {
          steuerung.neuerZustandF(steuerung.stock1);
   }
}
```

1.3 Umsetzung paralleler Zustandsdiagramme

Parallele Zustandsdiagramme können einfach implementiert werden, in dem statt einem, einfach mehrere Zustände verwaltet werden. Folgendes Beispiel (Abbildung 22) soll dieses verdeutlichen:

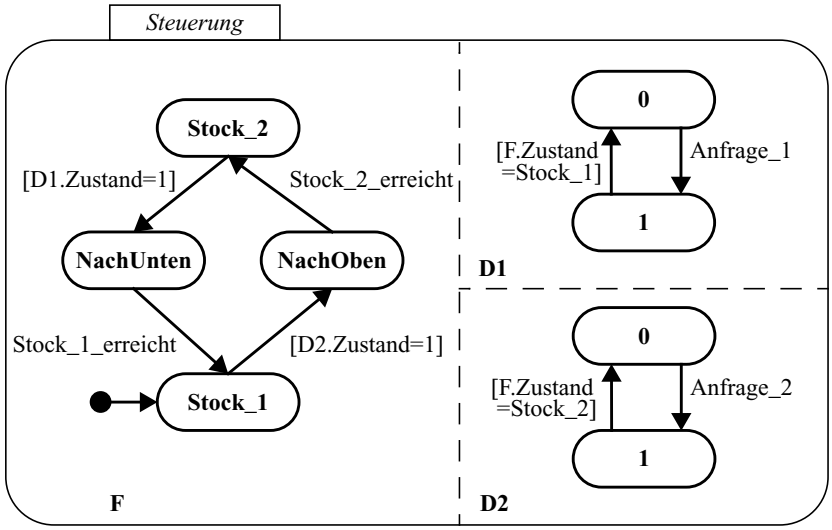


Abbildung 22 Paralleles Zustandsdiagramm

Bei der Umsetzung in Kode erhält die Klasse `Steuerung` für jeden Teilautomaten eine Instanzvariable vom Typ Zustand:

```
public class Steuerung implements
      FahrstuhlListener, AnfrageListener {
   private Zustand zustandF;
   private Zustand zustandD1;
   private Zustand zustandD2;
   …
```

Ebenso gibt es jetzt drei Methoden um einen Zustandswechsel durchzuführen:

```
public synchronized void neuerZustandF(Zustand z) {
     zustandF=z;
     zustandF.action();
     zustandF.check();
     zustandD1.check();
     zustandD2.check();
}

public synchronized void neuerZustandD1(Zustand z) {
     zustandD1=z;
     zustandD1.action();
     zustandF.check();
     zustandD1.check();
     zustandD2.check();
}


public synchronized void neuerZustandD2(Zustand z) {
    … // analog
}
```

Der Aufruf der check()-Methoden dient der Überprüfung von Transitionen, die nicht direkt durch Events ausgelöst werden. Beispielsweise kann der Übergang vom Zustand `D1.1` zu `D1.0` wie folgt ablaufen:

```
public class D1_0 extends Zustand {
   …

   public void check() {
           if (steuerung.getZustandF() == steuerung.stock_1) {
               steuerung.neuerZustandD1(steuerung.d1_0);
           }
}
```

Dabei wird davon ausgegangen, dass alle Bedingungen nur durch Zustandswechsel wahr oder falsch werden können. Um diese Einschränkung aufzulockern und z. B. Bedingungen periodisch abzufragen, kann ein zweiter Timer eingesetzt werden:

```
public class Steuerung implements TimerListener, …
    …
    public Steuerung() {
        …
        t1=new Timer("T1");
        t1.addTimerListener(this);
        t1.setTimeout(3000);
        …

    public void timerExpired(TimerEvent e) {
           …
           if (e.getName().compareTo("T1") == 0) {
                 zustandF.check();
```

```
                zustandD1.check();
                zustandD2.check();
                t1.setTimeout(3000);
            }
        }
```

Bemerkung: Um Platz zu sparen und wegen der textuellen Beschreibung sind die Kodebeispiele in diesem Kapitel nur unzureichend kommentiert. Kommentieren Sie Ihren Kode ausreichend im javadoc-Format!

2 Aufgaben (50 P.)

2.1 Implementierung der Signalanlagen

Bilden Sie zunächst Ihre Zustandsdiagramme für die *einzelnen* Signalanlagen auf Java-Kode ab (die Erweiterung auf die kommunizierenden und die übergreifenden Diagramme erfolgt in der nächsten Aufgabe). Gehen Sie dabei nach dem oben beschriebenen Schema (State-Pattern) vor.

Da die Umgebung (d.h. der „Ampelsimulator“) noch nicht zur Verfügung steht, schreiben Sie sich zum Testen Ihrer Implementierung eigene Detektor- und Signalgeber-Klassen, welche die Detektor- und Fehler-Ereignisse (z.B. zufällig) erzeugen.

Der DetektorListener und der FehlerListener benutzt eine `RemoteException`, die später erklärt wird. Lassen Sie diese einfach stehen (vor allem bei der Definition der Methode `vehicleDetected()` in Ihrer Implementierung von DetektorListener).

Wichtig: Kommentieren Sie Ihren *gesamten* Kode mit javadoc-Kommentaren, so dass sich die prinzipielle Funktionsweise der einzelnen Klassen auch mit Hilfe der generierten Dokumentation verstehen lässt! Verwenden Sie mindestens 2 Sätze zur Kommentierung von Methoden, mind. 1 Satz zu Attributen und mind. 5 Sätze als Kommentar für Klassen (siehe dazu auch [Ambler]).

Protokollieren Sie wichtige Testläufe. Besonders die als Ausgangspunkt der Modellierung eingesetzten Sequenz-Diagramm sollten nachvollzogen werden und die Ergebnisse der Testläufe gegen diese Diagramme validiert werden. Ein ausführlicherer Test der Zustandsdiagramme kommt in der nächsten Aufgabe.

2.2 Abgabe

Abzugeben sind

- die Java-Sourcen für die implementierten Zustandsdiagramme (inkl. vernünftiger javadoc-Kommentare)
- die generierten javadoc-Dokumente (vollständig und ausführlich)
- die Protokolle der durchgeführten Testläufe (z.B. als Sequenzdiagramm oder Text)

3 Literatur

[APIObs] *Java API.* `Observer` und `Observable`.
—zu finden unter [JavaAPI] im Package java.util

[Event] *Event Handling.* The Java Tutorial.
—Link ist auf der Praktikums-Site vorhanden

[EHandler] *General Information about Writing Event Listeners*. The Java Tutorial
—Link ist auf der Praktikums-Site vorhanden

[StPattern] *State Pattern*.
—Link ist auf der Praktikums-Site vorhanden

[ObPattern] *Observer Pattern*.
—Link ist auf der Praktikums-Site vorhanden

[ObImpl] *Observer.java* Java-Code der Observer-Klasse
—Link ist auf der Praktikums-Site vorhanden

[EventDoc] *environment.event Package Documentation*
—Link ist auf der Praktikums-Site vorhanden

[TimerDoc] *environment.timer Package Documentation*
—Link ist auf der Praktikums-Site vorhanden

[JavaAPI] Java 1.3. API Dokumentation
—z.B. unter http://java.sun.com/j2se/1.3/docs/api/index.html

[Ambler] Scott W. Ambler. AmbySoft Inc. Coding Standards for Java
—wird in der Vorbesprechung ausgeteilt