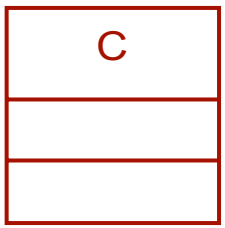
Implementierung I

Struktur

Implementierung II

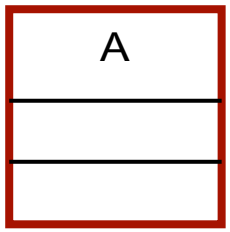
Verhalten
(Zustandsautomaten)

Java-Klassenrahmen

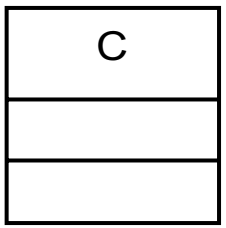



```
public class C {

   public C() { }

}
```

Aktive Klassen

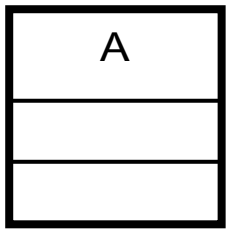



```
public class A implements Runnable {

   public A() { }

   public void run() { /* ... */ }
}
```

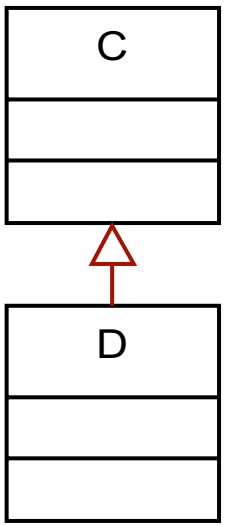



```
public class C {

   public C() { }

}
```

Relationen: Generalisierung

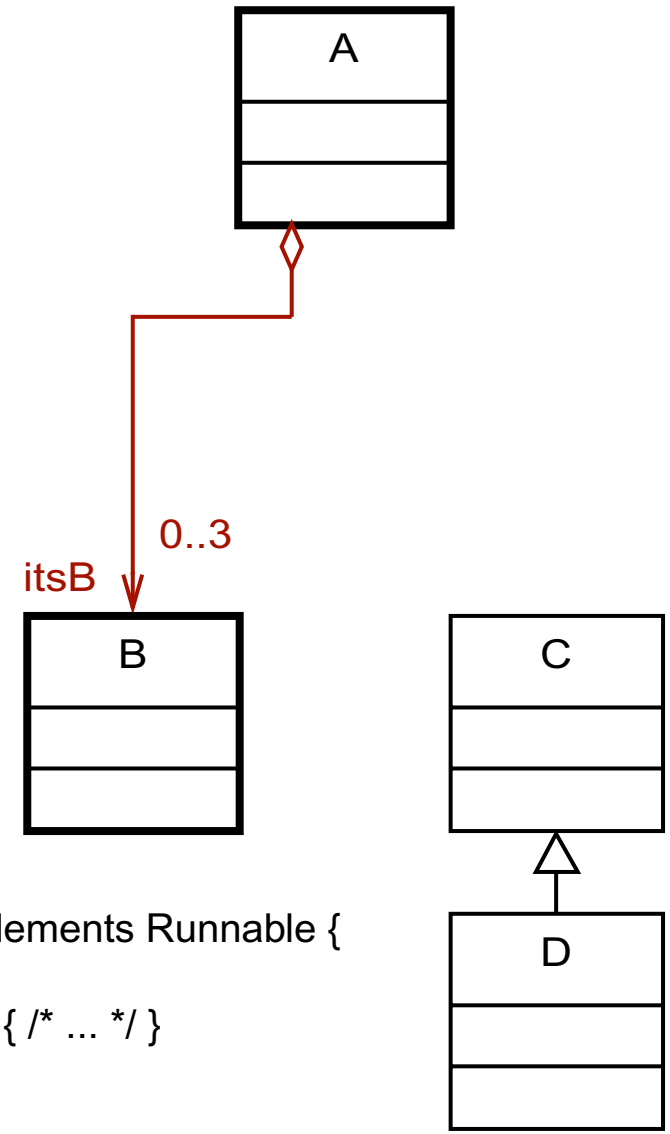



```
public class A implements Runnable {

  public A() { }

  public void run() { /* ... */ }
}
```




```
public class C {

  public C() { }

}

public class D extends C {
  public D() { }
}
```

Relationen: Gerichtete Assoziation/Aggregation

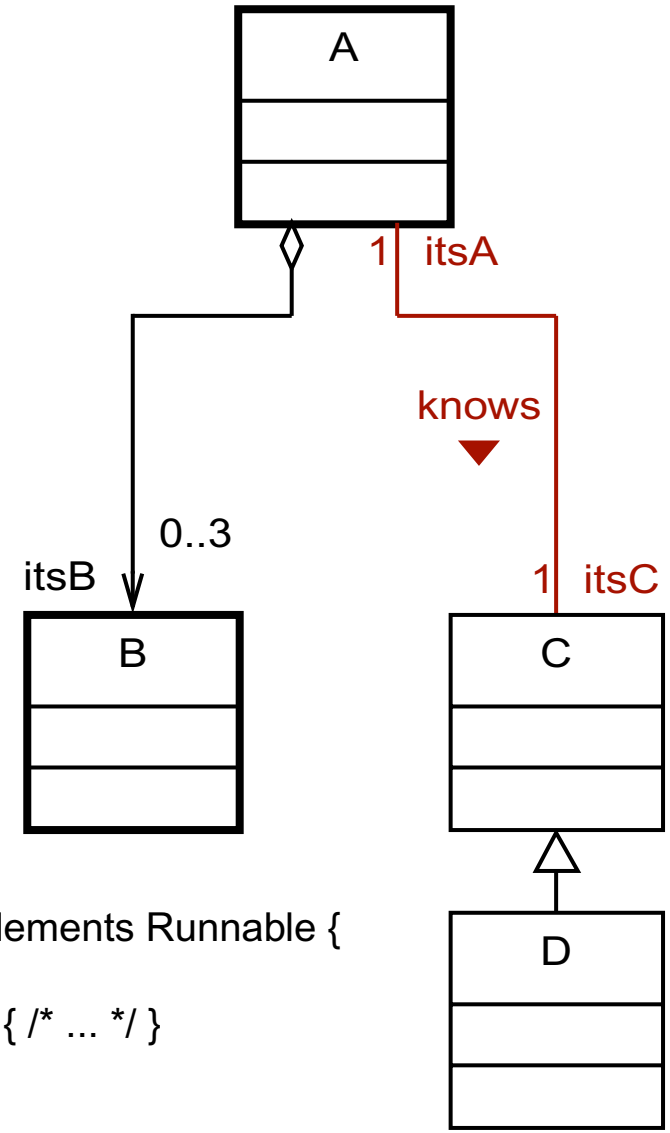



```
public class A implements Runnable {

  protected B itsB[3];
  public A() { }

  public void run() { /* ... */ }

  public void addItsB(B p_B) { /* ... */ }
}
public class C {

  public C() { }

}
```

```
public class B implements Runnable {
  public B() { }
  public void run() { /* ... */ }
}
```

```
public class D extends C {
  public D() { }
}
```

Relationen: Bidirektionale Assoziation/Aggregation



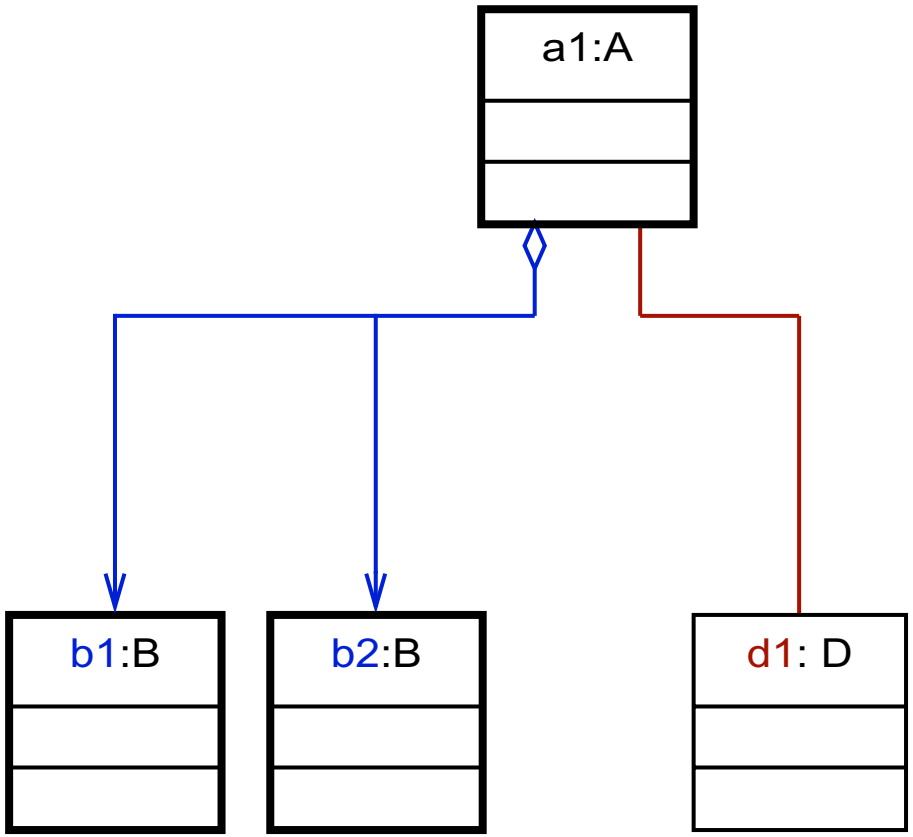

```
public class A implements Runnable {
   protected C itsC;
   protected B itsB[3];
   public A() { }

   public void run() { /* ... */ }
   public void setItsC(C p_C) { /* ... */ }
   public void addItsB(B p_B) { /* ... */ }
}
```

```
public class C {
   protected A itsA;
   public C() { }
   public void setItsA(A p_A) { /* ... */ }
}
```

```
public class B implements Runnable {
   public B() { }
   public void run() { /* ... */ }
}
```

```
public class D extends C {
   public D() { }
}
```

Instanziierung




```
public class A implements Runnable {
   protected C itsC;
   protected B itsB[3];
   public A() {
      setItsC(new D());
      B b1 = new B();
      new Thread(b1).start();
      addItsB(b1); /* analog für b2 */
   }
   public void run() { /* ... */ }
   public void setItsC(C p_C) { /* ... */ }
   public void addItsB(B p_B) { /* ... */ }
}
public class C {
   protected A itsA;
   public C() { }
   public void setItsA(A p_A) { /* ... */ }
}
```

```
public class B implements Runnable {
   public B() { }
   public void run() { /* ... */ }
}
```

```
public class D extends C {
   public D() {
   }
}
```

Aufgaben

- Implementierung der **Java-Coderahmen** aus den **strukturellen UML-Diagrammen**
  - Abbildung der Relationen
  - Instanziierung der Klassen und Belegung der Relationen (wenn möglich)
  - Start der Threads (aktive Klassen)
- Dokumentation des Codes
  - Coding Standards beachten!

Transformation der Zustandsdiagramme in Java-Code

- Implementierung der Zustände
- Implementierung der Transitionen

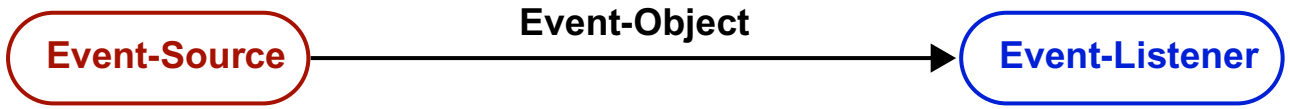
→ z.B. State Pattern

- Implementierung der Ereignisse

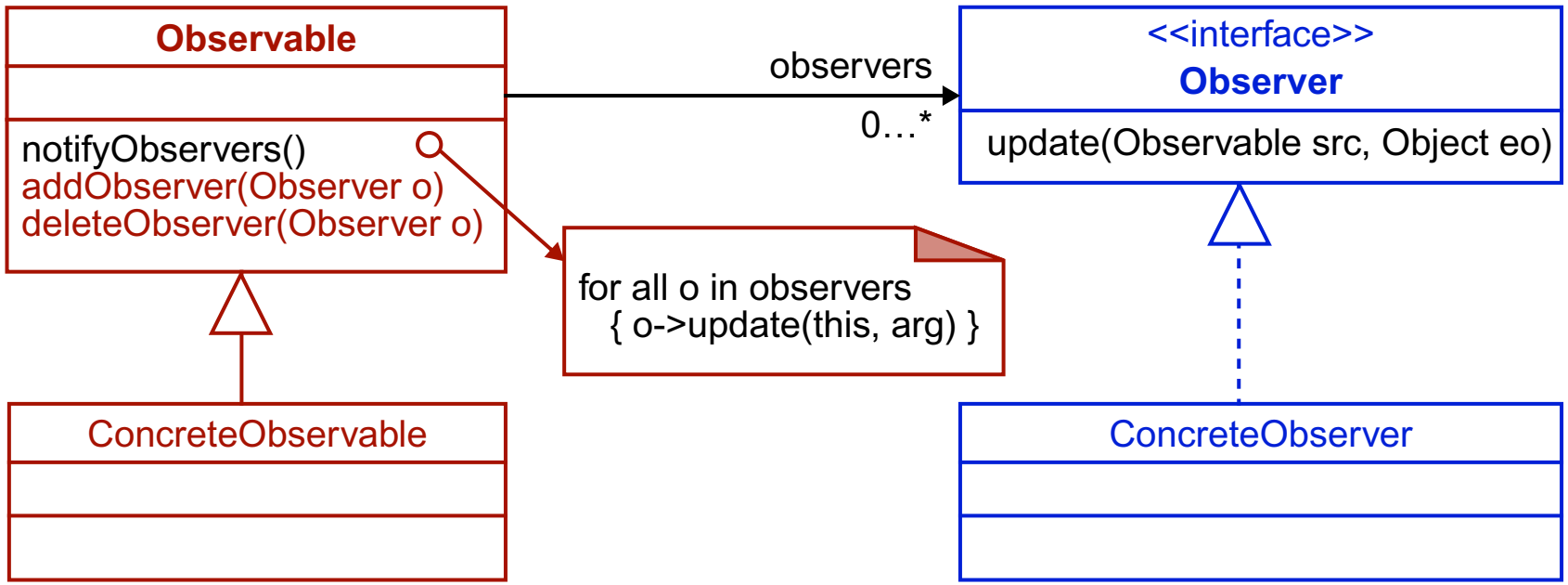
→ Observer Pattern

- Implementierung der Timer

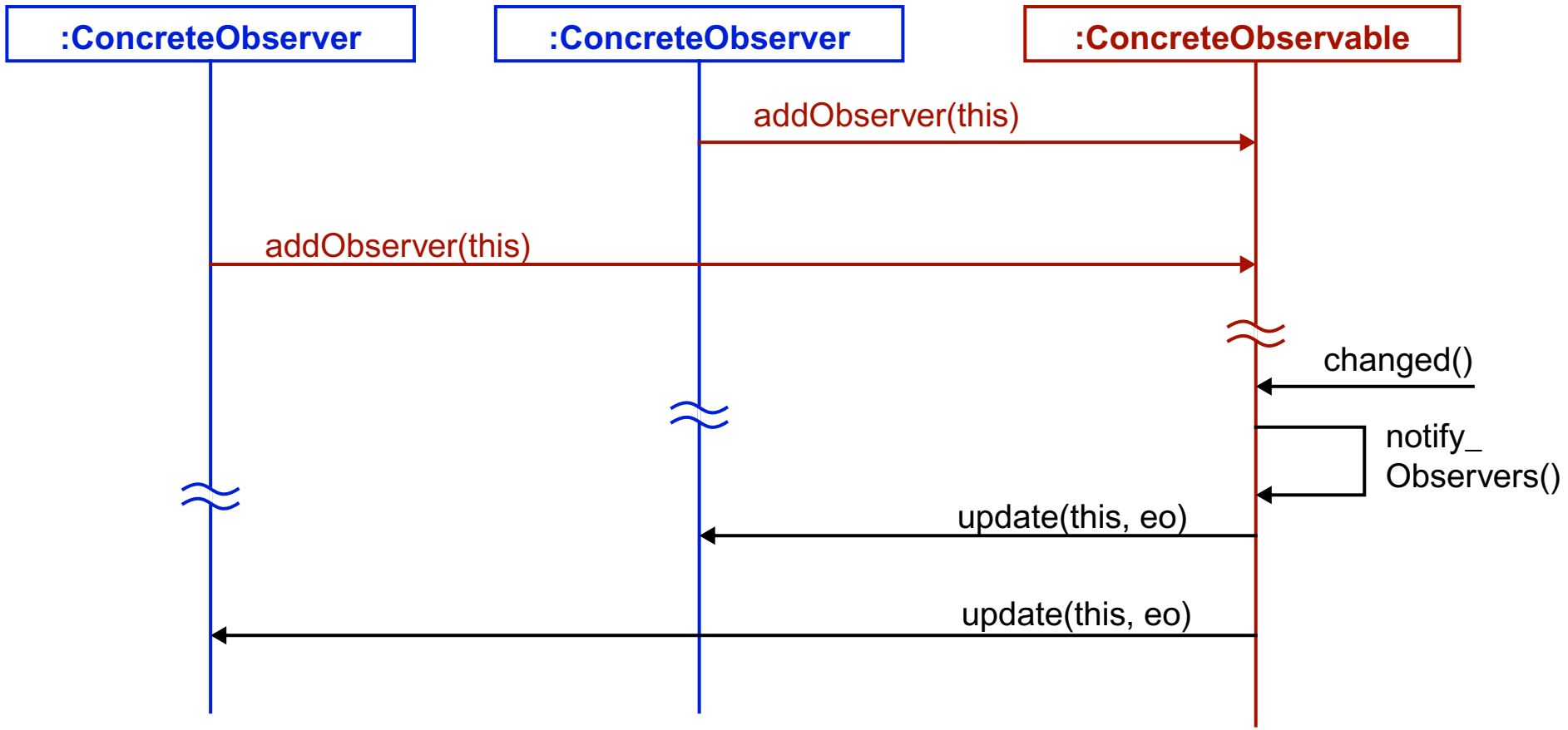
Event-Konzept von Java



Grundlage: Observer Pattern



Ablauf



in Java:

- addObserver() ⇒ add...Listener()

NamedEventObject

- im **`environment.event`** Package (auf Praktikums-WWW-Seite)

- Oberklasse für alle im Praktikum verwendeten Events!
  - selbst eine Spezialisierung von **`java.util.EventObject`**

- besitzt zusätzlich zu Event-Source-Objekt auch „lesbaren“ Name der Event-Source
  - auslesbar über **`getName()`**-Methode
  - Parameter für Konstruktor: **`java.lang.Object source, java.lang.String name`**

Existierende Event-Typen

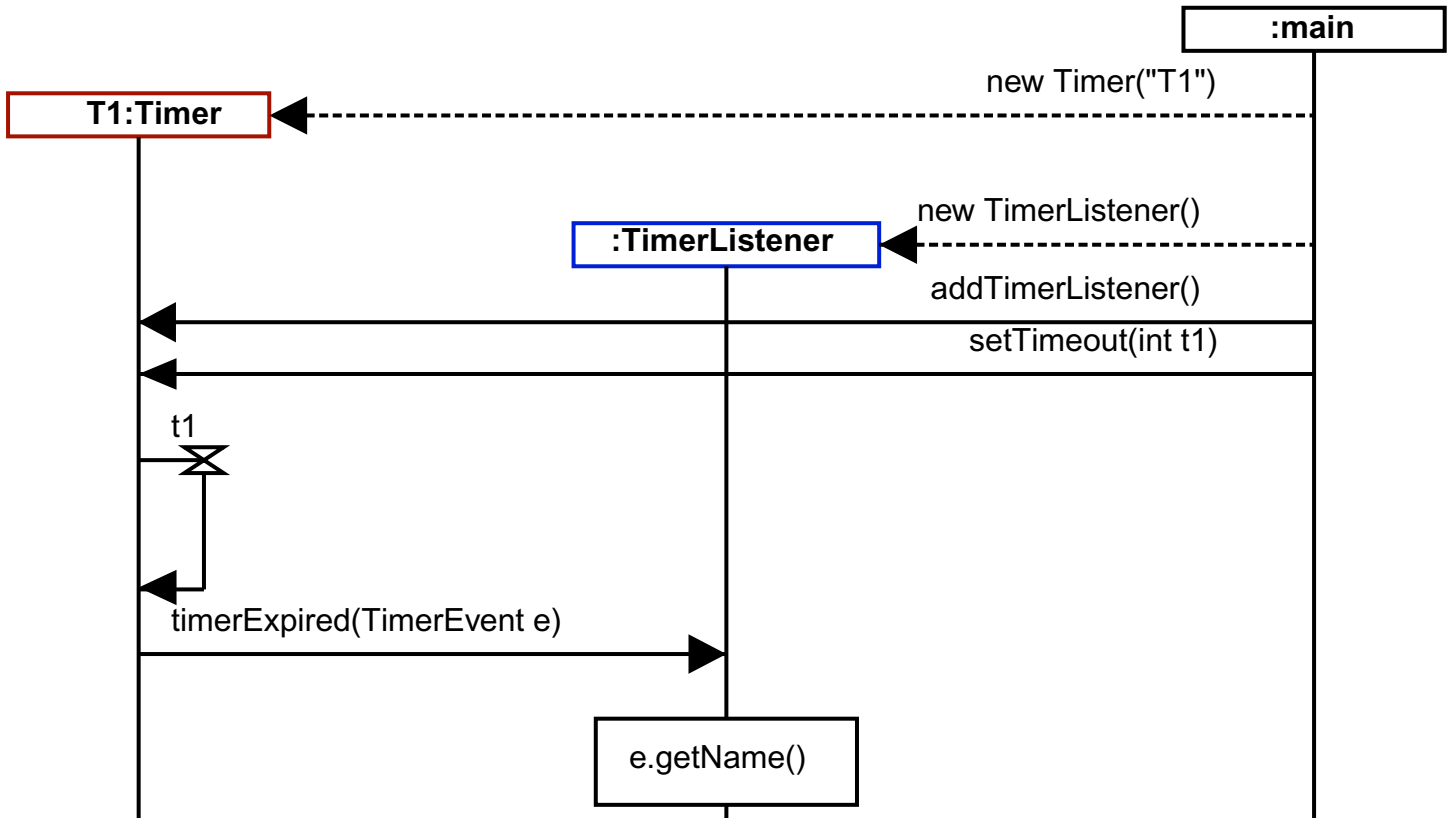
Event-Name	Auslöser	zugehöriger Listener (Observer)	update-Methode
DetektorEvent	Detektion eines KFZ	`DetektorListener`	`vehicleDetected()`
FehlerEvent	Auftreten eines Fehlers in einem Signalgeber	`FehlerListener`	`errorDetected()`
TimerEvent	Ablauf eines Timers	`TimerListener`	`timerExpired()`

Definition eigener Events:

- Spezialisierung von `NamedEventObject`

Timer

- im **`environment.timer`** Package
- **`javax.swing.Timer`** und **`java.util.Timer`** liefern keine **`NamedEventObjects`**
- „Aufziehen“ mit **`setTimeout(t1)`**
- „Abbrechen“ (Cancel) mit **`stopTimer()`**



Realisierungsmöglichkeiten:

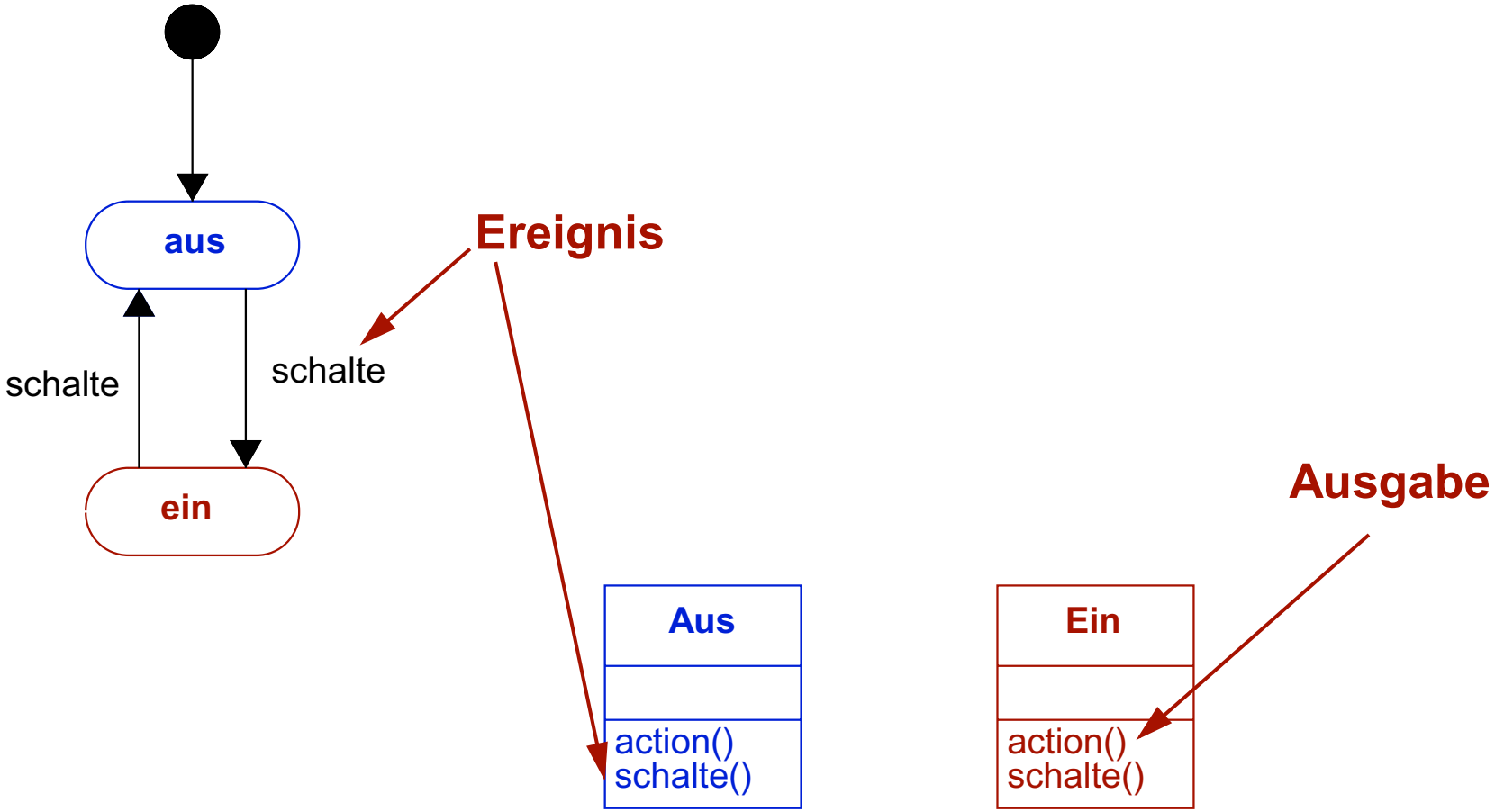
- **Zustandsübergangstabellen**
  - Vorteile: einheitlicher Code für alle Klassen, nur Tabelle ändert sich
  - Nachteile: Aufstellen der Tabelle aufwändig (Zustandsexplosion bei parallelen Automaten)

- **logische Ausdrücke**
  - Vorteile: äußerst kompakt, ideal für Hardwarerealisierung
  - Nachteile: Minimierung der Zustandsübergangstabelle nötig, schlecht erweiterbar

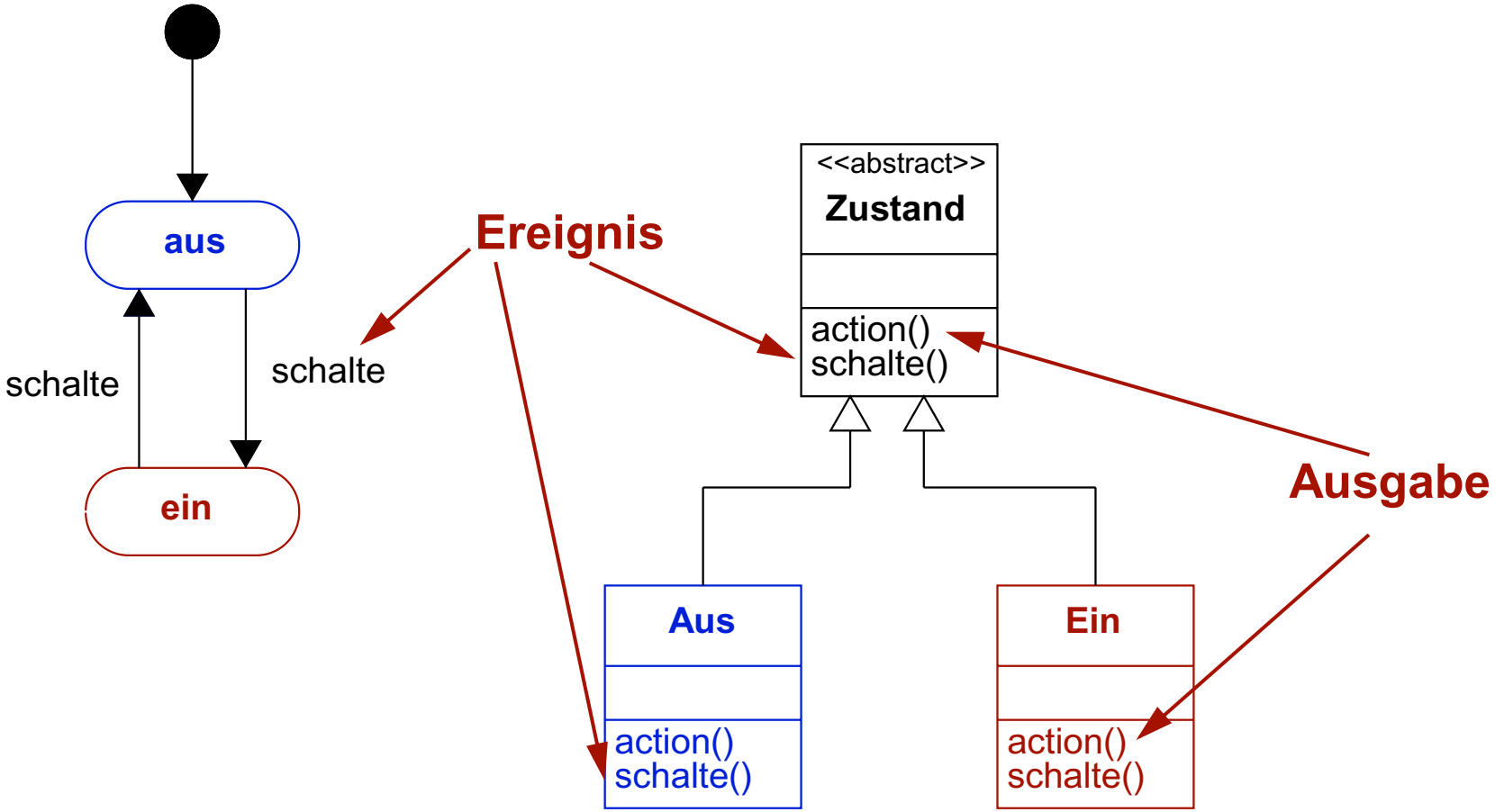
- **Case-Anweisungen**
  - Vorteile: geeignet für prozedurale Sprachen, relativ kompakt
  - Nachteile: verschachtelte Case/If-Then-Konstrukte, unübersichtlich, schlecht erweiterbar

- **State-Pattern**
  - Vorteile: geeignet für objektorientierte Sprachen, gut erweiterbar, übersichtlich, hohe Verfolgbarkeit von Zustandsdiagramm zu Code
  - Nachteile: viele Klassen (Code) mit hoher Ähnlichkeit (Hilfe: Wiederverwendung!)

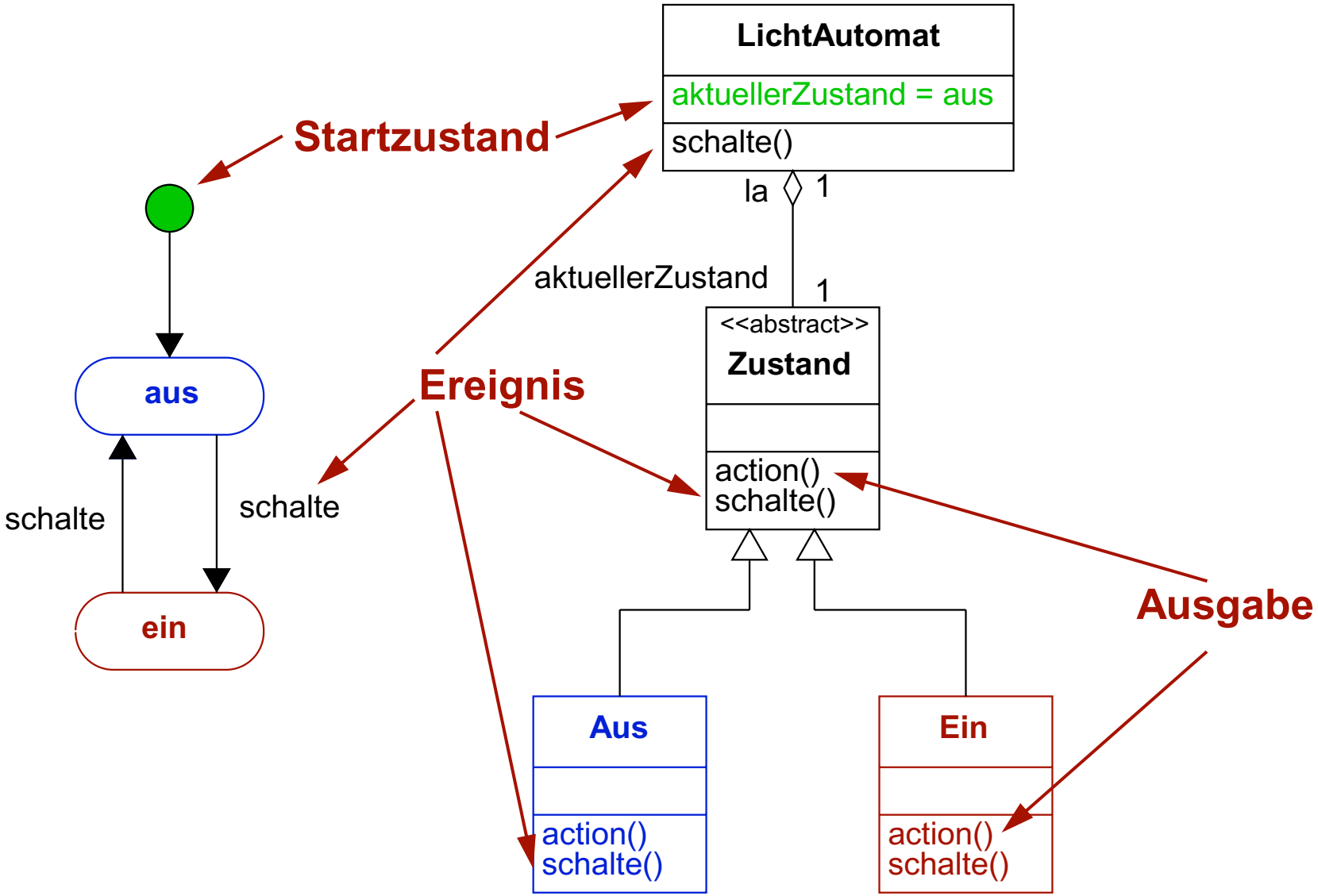
State-Pattern: Zustand = Klasse



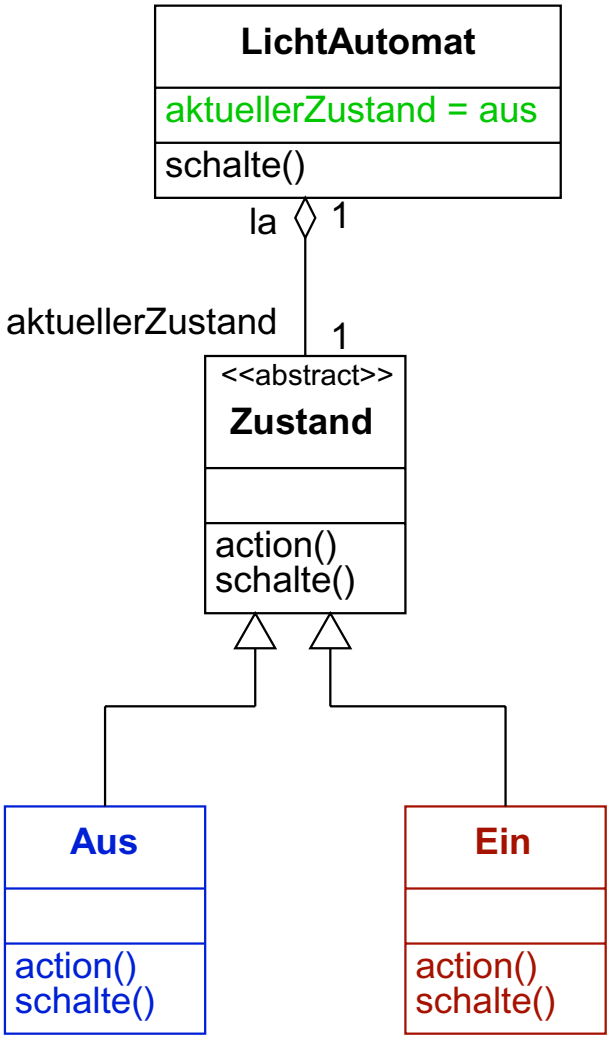
State-Pattern: Verallgemeinerung



State-Pattern: „Aggregation“ zur Automaten-Klasse

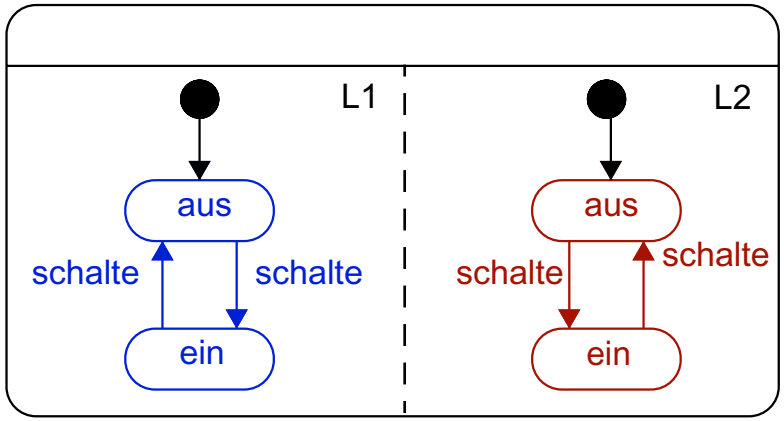


State-Pattern: Java-Code




```
public class LichtAutomat implements SchalterListener {
    private Zustand aktuellerZustand;
    public final Zustand aus; public final Zustand ein;

    public LichtAutomat(Schalter schalter) {
        aus = new Aus(this); ein = new Ein(this);
        schalter.addSchalterListener(this);
        aktuellerZustand=aus; aus.action(); }

    public void schalte() { aktuellerZustand.schalte(); }

    public void synchronized neuerZustand(Zustand p_Zustand) {
      aktuellerZustand = p_Zustand; aktuellerZustand.action(); }
}

public abstract class Zustand {
    protected LichtAutomat la;

    public Zustand(LichtAutomat p_LichtAutomat) { la = p_LichtAutomat; }

    public void action() {}
    public void schalte() {}
}

public class Aus extends Zustand {
    public Aus(LichtAutomat p_LichtAutomat) { super(p_LichtAutomat); }

    public void action() { /* Lampe ausschalten */ }

    public void schalte() { la.neuerZustand(la.ein); }
}
```

State-Pattern: Parallele Zustandsdiagramme




```
public class LichtAutomat implements SchalterListener {
        private Zustand aktuellerZustandL1; private Zustand aktuellerZustand L2;
        // ...

        public synchronized neuerZustandL2(Zustand p_Zustand) {
                aktuellerZustandL2 = p_Zustand;
                aktuellerZustandL2.action();
                aktuellerZustandL1.check();
                aktuellerZustandL2.check();
        }
}
public class L2_Aus extends Zustand {
        // ...

        public void check() {
                // Prüfe Bedingungen und wechsle
                // dann evtl. den Zustand
        }
}
```

Aufgaben

- Implementierung der ***lokalen* Zustandsdiagramme**
  - möglichst unter Verwendung des *State-Pattern*
  - Erweiterung auf kommunizierende Automaten i.d. nächsten Aufgabe

- Für den Test der Implementierung:
  - eigene Detektor- und Signalgeber-Klassen schreiben (als „*Ersatz*“ für die späteren „echten“ Klassen)
  - **`RemoteException`** von Detektor- und FehlerListener erst später wichtig

- Dokumentation des Codes
  - Bei Verwendung des State-Pattern: Verfolgbarkeit durch gleiche Benennung der Zustände und Ereignisse gewährleisten (dann geringerer Dokumentationsaufwand)
  - Coding Standards beachten!