Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

INFORMATIK

Objektorientierte Programmierung

Event/Listener-Pattern

Observer-Pattern in Java

Roland Gisler

Inhalt

- Zirkuläre Beziehungen
- Observer-Pattern in Java: Event/Listener-Pattern
- Beispiel: **`PropertyChangeListener`**
- Innere Klassen und anonyme innere Klassen
- Kurzer Ausblick: Lambda-Expressions
- Zusammenfassung

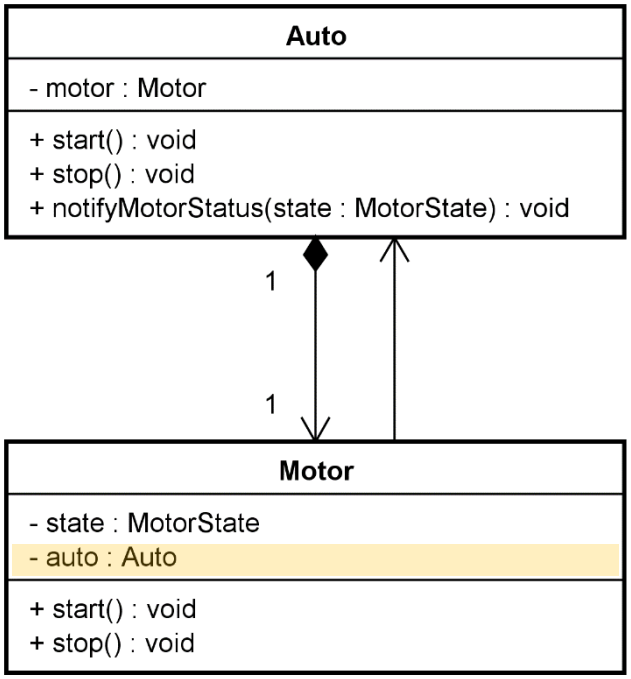
Lernziele

- Sie verstehen das Observer- bzw. Event/Listener-Pattern in Java.
- Sie können ein Listener-Interface implementieren und nutzen.
- Sie können eigene Event-Typen und Eventquellen implementieren.
- Sie erkennen das Potenzial des Event/Listener-Patterns, um damit die Kopplung zwischen Softwareeinheiten massiv zu senken.
- Sie haben eine Vorstellung, was (anonyme) innere Klassen sind.
- Sie können einfache (anonyme) innere Klassen zur Event-Behandlung implementieren.

Zirkuläre Beziehungen

Zirkuläre Beziehung zwischen Klassen

- Gegeben sind die folgenden Anforderungen für ein Auto mit Motor:
  - Das Auto kann mitsamt Motor ein- und ausgeschaltet werden.
  - Der Motor soll seinen Status und allfällige Störungen (z.B. Überhitzung) an das Auto melden.
- Das führt zu nebenstehendem, vereinfachten Modell eines Autos.
  - Ein **`Auto`** «hat» einen **`Motor`**.
  - Damit der **`Motor`** seinen Status über **`notifyMotorStatus(…)`** melden kann, benötigt er eine Referenz auf das **`Auto`**.



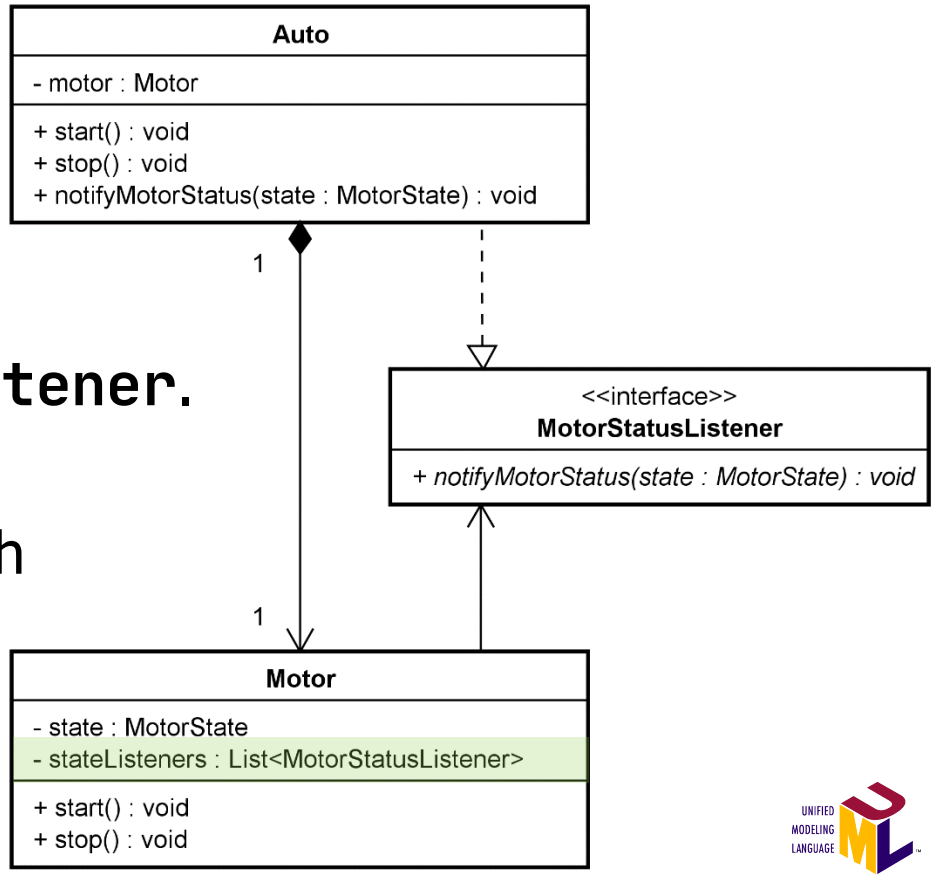
- Das führt nicht nur zu einer **starken Kopplung**, sondern sogar zu einer **zirkulären Beziehung!**

Vorsicht bei zirkulären Beziehungen!

- Zirkuläre Beziehungen verursachen nicht nur eine sehr starke Kopplung, sondern haben noch weitere, gravierende Nachteile:
  - Die zirkulär verbundenen Elemente sind **nicht mehr teilbar**, und können somit nicht mehr einzeln verwendet werden.
  - Sind nur innerhalb der selben Releaseunit (Projekt) überhaupt realisierbar (keine unabhängige Kompilation bei Änderungen).
- Interessant: Der Zyklus ist bei unserem «**`Auto`**/**`Motor`**»-Beispielmodell gar nicht nötig/gewollt, denn dem **`Motor`** kann es eigentlich «egal» sein, wem er seinen Status meldet!
- Eine mögliche Lösung: **Observer**-Pattern (➔ Designpattern, GoF)
  - Details dazu erfahren Sie im Modul VSK im 3./4. Semester.

Auflösen von zirkulären Beziehungen

- Zirkuläre Beziehungen können wir mit Hilfe von Interfaces brechen.
- Erklärung:
  - **`Auto`** «hat» noch immer einen **`Motor`**.
  - **`Auto`** implementiert neu das Interface **`MotorStatusListener`**.
  - Der **`Motor`** kennt das **`Auto`** nicht mehr, sondern nur noch den Typ des Interfaces!



➔ **Kein Zyklus mehr**!

➔ Nebeneffekt: Es kann also verschiedene **`MotorStatusListener`**(-Implementationen) geben (alles was einen Motor «brauchen» kann)!

Event/Listener Pattern von Java

Event-Handling – Analogie «Abonnieren einer Zeitung»

Vergleichbar mit dem Abonnement einer Zeitung:
Verschiedene **Leser** abonnieren bei einem **Verlag** eine **Zeitung**.

- Die Zeitungsleser (Abonnementen) abonnieren eine Zeitung:
  - Der Listener registriert sich bei Event-Quelle.
- Verlag verwaltet die Abonnementen:
  - Die Event-Quelle verwaltet (merkt sich) alle Listener.
- Die Zeitung wird produziert und versendet:
  - Der Event wird von der Event-Quelle produziert und «gefeuert».
- Briefkasten beim Abonnementen:
  - Die Schnittstelle für den Funktionsaufruf.

➔ Der Verlag braucht nur sehr wenig über seine Leser zu wissen!

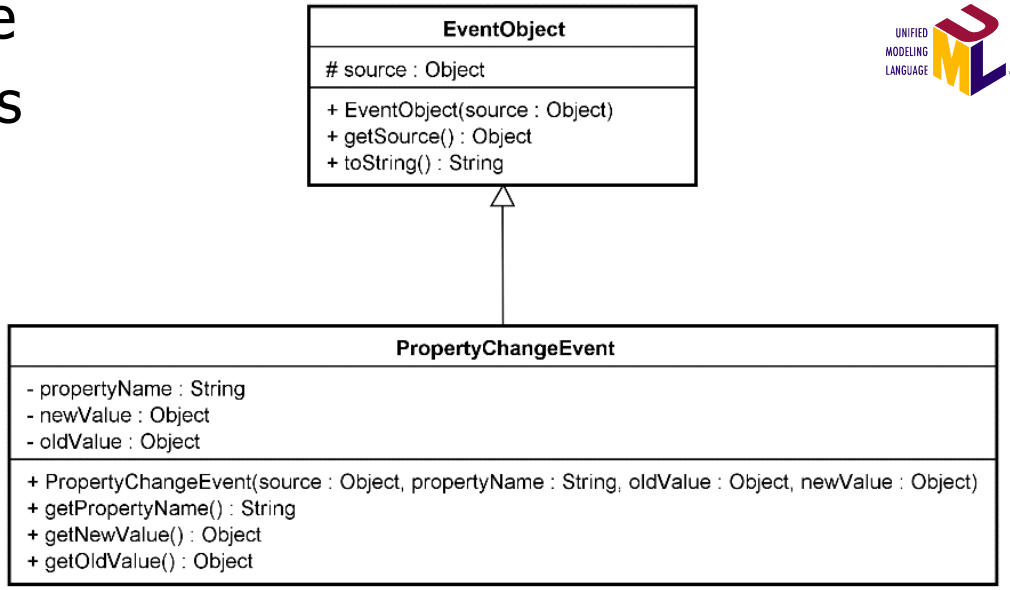
Event/Listener-Pattern in Java - Grundlagen

- Java verwendet dieses Prinzip häufig, u.a. für die Java-Beans Technologie (z.B. Properties) und für die ereignisorientierte Programmierung z.B. bei ➔ **G**raphical **U**ser **I**nterfaces (GUI).
- Es gibt eine Reihe von Konventionen, um dieses Pattern zu implementieren. Die zwei wesentlichsten Teile sind:
  - **`Listener`**-Interfaces (häufig mit nur einer Methode).
  - **`Event`**-Objekte (zur Zusammenfassung der relevanten Daten).
- Dazu kommen mindestens vier verschiedene Methoden:
  - Registrierung eines Listeners für einen Event (auf Quelle).
  - Deregistrierung eines Listeners (auf Quelle).
  - Methode zur Verteilung eines Events (auf Quelle).
  - Methode für Notifikation des Events (auf Empfänger).

Beispiel: `PropertyChangeEvent`

Beispiel: `PropertyChangeEvent` - Event

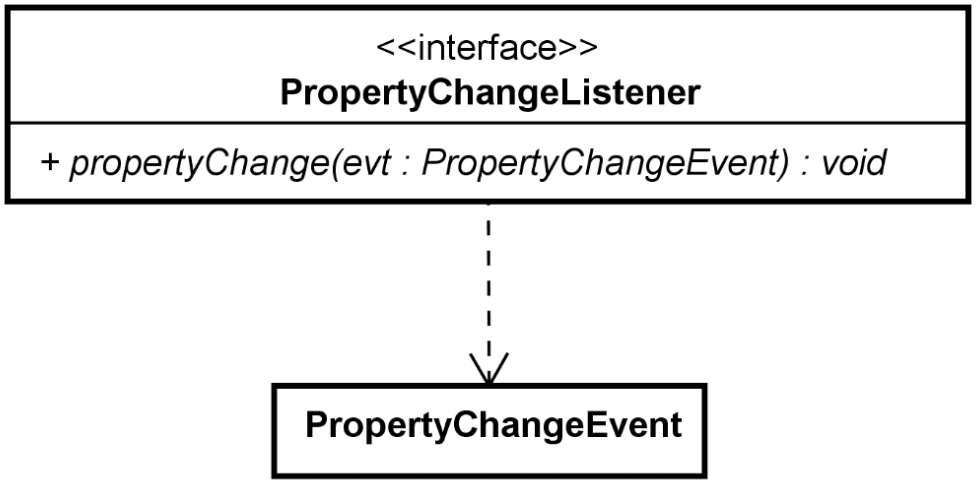
- Wird verwendet, um über die Veränderung eines Properties (vereinfacht: Attribut) eines Objektes zu Informieren.
- Enthält Informationen über die Quelle (Source), den Namen des Properties, sowie dessen alten und neuen Wert.



- Hinweis: Alle Event-Klassen (auch selber implementierte) werden von der Basisklasse **`EventObject`** vererbt.

Beispiel: `PropertyChangeListener` - Interface

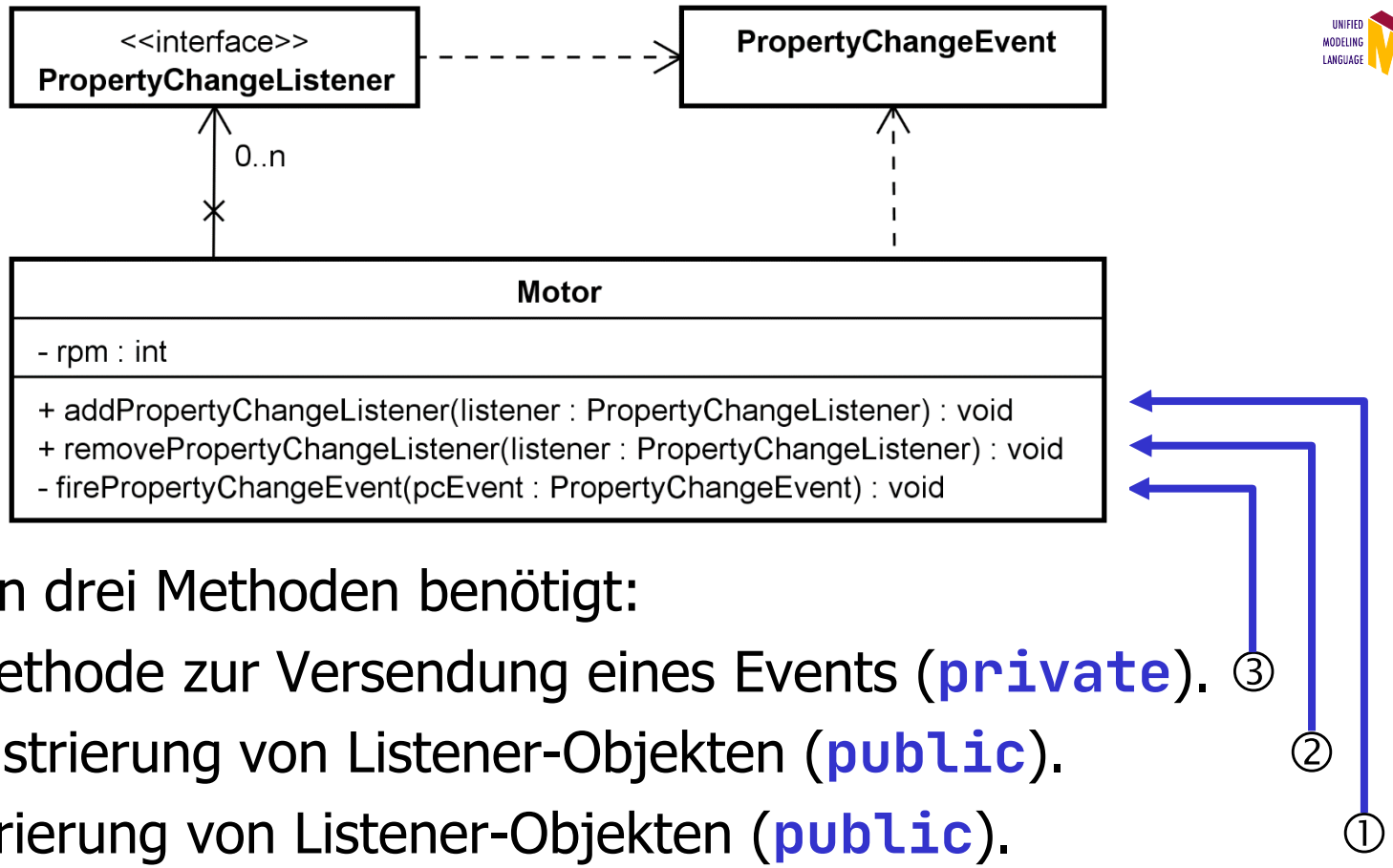
- Das zum **`PropertyChangeEvent`** passende Interface für die Listener (Beobachter) heisst **`PropertyChangeListener`**.
- Es enthält eine einzige Methode:



- Klassen, welche **`PropertyChangeEvents`** empfangen möchten, **müssen** dieses Interface implementieren und bei der Quelle dafür registriert werden.

Beispiel: `PropertyChangeListener` – Event-Quelle

- Beispiel: Ein **`Motor`**, welcher über seinen Betriebszustand informieren will:



- Es werden drei Methoden benötigt:
  - Hilfsmethode zur Versendung eines Events (**`private`**).
  - Deregistrierung von Listener-Objekten (**`public`**).
  - Registrierung von Listener-Objekten (**`public`**).

Beispiel: PropertyChangeListener – Event Quelle – 1

```
// Datenstruktur zur Speicherung aller Listener.
private final List<PropertyChangeListener> changeListeners =
                                                    new ArrayList<>();

/**
 * Registriert einen PropertyChangeListener.
 * @param listener PropertyChangeListener.
 */
public void addPropertyChangeListener(
                            final PropertyChangeListener listener) {
   this.changeListeners.add(listener);
}

/**
 * Deregistriert einen PropertyChangeListener.
 * @param listener PropertyChangeListener.
 */
public void removePropertyChangeListener(
                            final PropertyChangeListener listener) {
   this.changeListeners.remove(listener);
}
```

Beispiel: PropertyChangeListener – Event-Quelle – 2

- Codefragment der Klasse **Motor**: Versenden eines Events.

```
public void switchOn() {
  if (isSwitchedOff()) {
    this.state = MotorState.ON;
    final PropertyChangeEvent pcEvent = new PropertyChangeEvent(
                          this, "state", MotorState.OFF, MotorState.ON);
    this.firePropertyChangeEvent(pcEvent);
  }
}

/**
 * Informiert alle PropertyChangeListeners über PropertyChangeEvent.
 * @param pcEvent PropertyChangeEvent.
 */
private void firePropertyChangeEvent(
                                    final PropertyChangeEvent pcEvent) {
  for (final PropertyChangeListener listener : this.changeListeners) {
    listener.propertyChange(pcEvent);
  }
}
```

Beispiel: `PropertyChangeListener` – Event-Empfänger

- Klasse **`Car`**, registriert sich bei **mehreren** seiner Komponenten:

```
public final class Car implements PropertyChangeListener {

   public Car(final String model) {

      // Komponenten erzeugen.
      this.motor = new Motor();
      this.lightFrontLeft = new Light();
      this.lightFrontRight = new Light();

      // Sich selber als Listener registrieren.
      this.motor.addPropertyChangeListener(this);
      this.lightFrontLeft.addPropertyChangeListener(this);
      this.lightFrontRight.addPropertyChangeListener(this);
      …
   }
   …
}
```

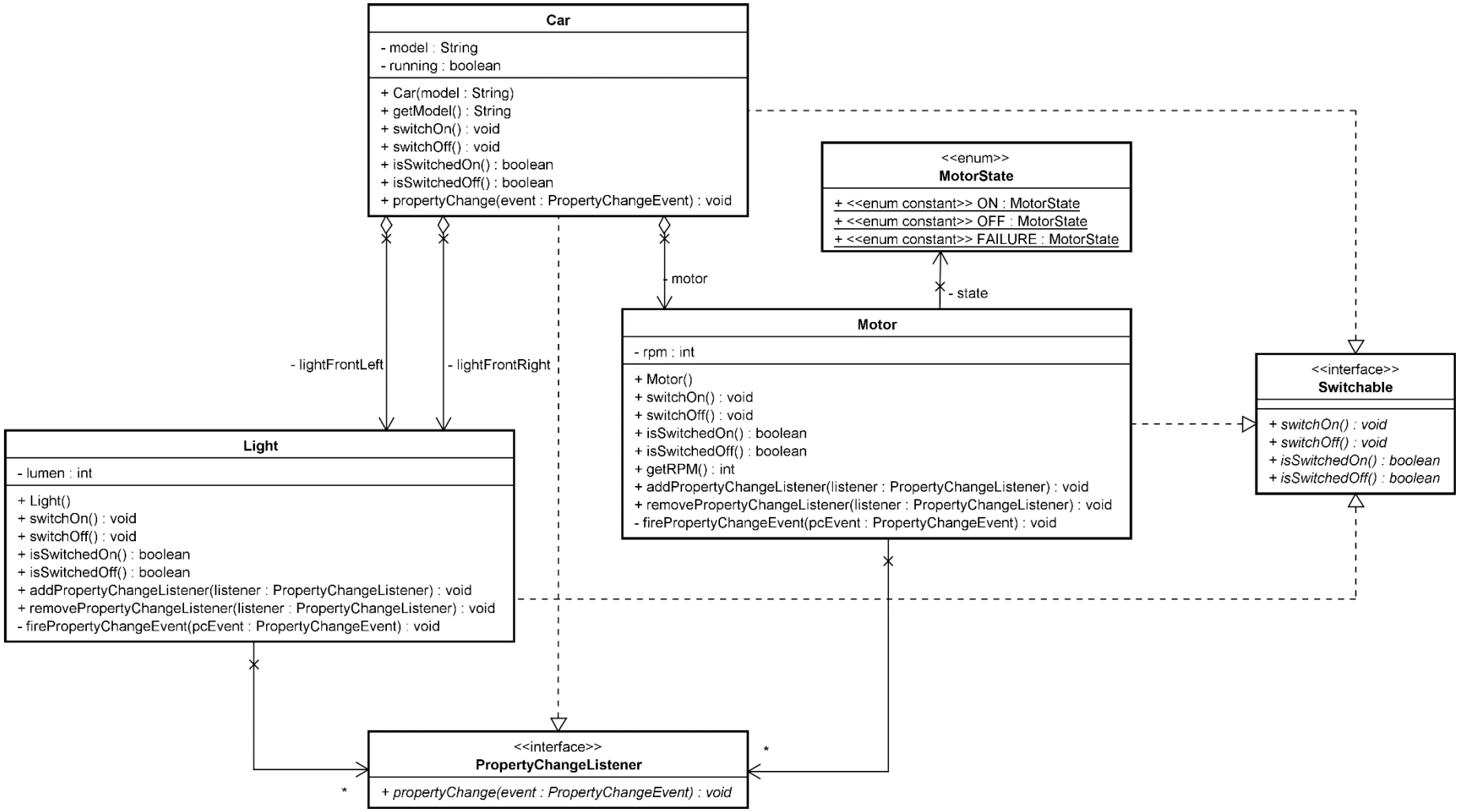
Beispiel: `PropertyChangeListener` – Event-Behandlung

- Listener, welcher Events von **mehreren** Komponenten empfängt:

```
public final class Car implements PropertyChangeListener {
  …
  @Override
  public void propertyChange(final PropertyChangeEvent event) {
    if (event.getSource() == this.motor) {
      this.handleMotorEvent("Motor", event);
    }
    if (event.getSource() == this.lightFrontLeft) {
      this.handleLightEvent("Scheinwerfer Links", event);
    }
    if (event.getSource() == this.lightFrontRight) {
      this.handleLightEvent("Scheinwerfer Rechts", event);
    }
  }
}
```

- Eigentliche Behandlung wird in Methoden ausgelagert:
  Code wird übersichtlicher; Methoden sind wiederverwendbar.

Beispiel: `PropertyChangeListener` – UML-Diagramm



Empfehlungen zum Event-Handling

- So weit möglich und sinnvoll immer bereits bestehende Eventklassen und Listener-Interfaces verwenden.
  - weniger Programmieraufwand, bekannte Semantik.
  - eigene Event-Klassen immer von **`EventObject`** ableiten.
- Behandlung von Events am besten an (private) Methoden delegieren, Code wird dadurch übersichtlicher.
- Halten Sie sich an die Namenskonventionen:
  - **`XxxEvent`** – Klasse für den Event
  - **`XxxListener(…)`** – Interface für den Listener
  - **`Xxx[Performed](…)`** – Methode auf Listener-Interface
  - **`addXxxListener(…)`** – Registrierung von Listenern auf Quelle
  - **`removeXxxListener(…)`** – Deregistrierung von Listenern
  - **`fireXxxEvent(…)`** – Versenden von Events von Quelle

Innere Klassen

Unschönheit beim Event-Handling

- Beim Event-Handling haben wir festgestellt, dass es viele verschiedene Quellen für gleichartige Events geben kann.
  - Registriert sich ein Objekt bei mehreren Quellen für die gleichen Events, treffen diese somit alle in der selben Methode ein!
- Das Event-Objekt enthält als Information auch die Quelle (**`source`**), was zwar die Differenzierung ermöglicht, aber:
  - Auswertung ist über ein **`if`**-Konstrukt nötig.

  ➔ SRP nicht eingehalten, unübersichtlich und fehleranfällig!
- Schöner wäre es doch, wenn man bei jeder Quelle ein eigenständiges Objekt registrieren könnte.
  - Dafür müsste man aber für jede Quelle wiederum eine **eigenständige** Listener-Klasse implementieren!

  ➔ geringe Kohäsion, gehört ja eigentlich **in** die Listener-Klasse.

Lösung: Innere Klassen (inner classes)

- Wir können in Java **Klassen innerhalb von Klassen** definieren.
  - Nebenbei: Auch Interfaces in Interfaces sind möglich.
- Sinnvolle Anwendung: Damit kann man «private» (Hilfs-)Klassen definieren, welche **nur innerhalb** einer Klasse sichtbar sind.
  - können aber auch **`public`** und **`static`** sein ➔ zurückhaltend einsetzten, weil dann wird es komplizierter!
- Beispiele aus den Datenstrukturen:
  - Das Interface **`Map`** definiert ein internes Interface **`Map.Entry`**.
  - Die Klasse **`HashMap`** definiert eine interne Klasse **`Node`** (welche **`Map.Entry`** implementiert).
- Wir können die Technik der inneren Klassen u.a. bei Event-Listeners gut einsetzen, um weniger Code schreiben zu müssen, und eine **höhere Kohäsion** zu erreichen.

Beispiel: Innere Klasse - Eventhandling

- **Innere** Klasse `MotorPropertyListener` in der Klasse `Car`:

```
public final class Car {
    …
    private class MotorPropertyListener
                                    implements PropertyChangeListener {
        @Override
        public void propertyChange(final PropertyChangeEvent event) {
            handleMotorEvent("Event vom Motor", event);
        }
    }

    public Car(final String model) {
        …
        this.motor = new Motor();
        this.motor.addPropertyChangeListener(
                                    new Car.MotorPropertyListener());
    }
}
```

- Vorteil: **Individuelle** Listener **pro** Registrierung/Quelle möglich!

Java – Anonyme innere Klassen (anonymous inner classes)

- Wenn wir wissen, dass wir eine innere Klasse genau **nur einmal** und **nur an einer einzigen** Stelle instanziieren, können wir diese auch als **anonyme innere Klasse** definieren.
- Eigenschaften von anonymen innere Klassen
  - Sie haben **keinen** Namen.
  - Sie werden **an Ort und Stelle** ihrer Verwendung definiert!
- **Achtung**: Die Syntax ist zwar logisch, aber vielleicht etwas gewöhnungsbedürftig.
  - Geschweifte Klammern innerhalb runder Klammern **`({…})`**

Beispiel – Anonyme innere Klasse für Eventhandling

- **Anonyme** innere Klasse als `MotorPropertyListener` in `Car`:

```
public final class Car {
  …
  public Car(final String model) {
    …
    this.motor = new Motor();
    this.motor.addPropertyChangeListener(new PropertyChangeListener() {

      @Override
      public void propertyChange(final PropertyChangeEvent event) {
        handleMotorEvent("Event vom Motor", event);
      }
    });
  }
  …
}
```

- Anonyme Klassen werden auf Basis von Interfaces oder Klassen definiert und dabei **an Ort und Stelle** implementiert bzw. spezialisiert!

Kurzer Ausblick: Eventhandling mit Java Lambdas

- **Lambdas** sind ein Konzept der **funktionalen Programmierung**.
  - Wurden bei Java (erst) mit der Version **8** eingeführt.
  - Methoden können damit quasi als Objekte behandelt werden.
- Damit können wird die Listener-Implementationen abermals drastisch verkürzen:

```
…
this.motor.addPropertyChangeListener(
                e -> handleMotorEvent("Event von Motor", e));
…
```

- Wichtige Bedingung: Listener-Interface muss ein sogenanntes «functional interface» mit nur **einer abstrakten Methode** sein.

➔ Mehr dazu folgt in ➔ SW12 und optional im Wahl-Modul «Programming Concepts & Paradigms» (PCP).

Zusammenfassung

- Observer-Pattern, in Java als Event/Listener-Pattern implementiert:
  - Grundlage der ereignisgesteuerten Programmierung.
  - Ziel: Lose Kopplung von Klassen.
- Events, Event-Quellen und Event-Empfänger (Listener)
- Behandlung von Events über Listener-Interfaces.
- Konzept der inneren Klassen.
- Konzept der anonymen, inneren Klassen.

Fragen?

Fragen bitte im
ILIAS-Forum