Programmieren in Java

Vorlesung 02: Einfache Klassen (Fortsetzung)

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Inhalt

Einfache Klassen

- Testen für einfache Klassen
- Vermeiden von sinnlosen Objekten
- Benennen von Konstanten
- Vermeiden von magischen Werten

Zusammengesetzte Klassen

- Merkmale
- Fahrkarten mischen

Zusammenfassung: Arbeiten mit einfachen Klassen

Executive Summary

1. Testen
   - Synthetische Tests aus Spezifikation
   - Analytische Tests aus Fehlerberichten
2. Es sollte nicht möglich sein, sinnlose Objekte zu erzeugen.
   - Factory-Methoden
   - Aufzählungstypen
3. Zur Vermeidung von Missverständnissen und zur Dokumentation sollte jede Konstante sinnvoll benannt werden.
4. Der Wertebereich eines Felds sollte nicht künstlich beschränkt werden. (Keine magischen Werte.)

Testen für einfache Klassen

Testen

- Ansatz: *Unit Testing*
- separate Testsuite für jede Programmkomponente (z.B. Klasse)
- separate Tests für jede Methode
- zwei Arten von Tests
  - synthetische Tests abgeleitet von der Spezifikation
    - Demonstration der normalen Funktion
    - Randfälle
  - Tests, die Fehler dokumentieren
    - Bug report

    → fehlschlagender Testfall

    → Bug fix

    → funktionierender Testfall
- Tests werden nicht wieder entfernt

Testen mit Werkzeugen

JUnit

- Werkzeug zum Unit Testing
- Eclipse-Integration
- Rahmen zum automatischen Testen auf Knopfdruck

Testen mit Werkzeugen

JUnit

- Werkzeug zum Unit Testing
- Eclipse-Integration
- Rahmen zum automatischen Testen auf Knopfdruck

Vorgehensweise

- Zu jeder "normalen" Klasse erstelle eine Testklasse
- Für jede signifikante Methode erstelle mindestens eine synthetische Testmethode
- Für jeden nachvollziehbaren Bug erstelle eine Testmethode

Testklasse zu **SimpleTicket**

```
1 package lesson_01;
2
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5
6 public class SimpleTicketTest {
7   // Testmethoden
8 }
```

- Testklasse im gleichen Paket
- Die beiden **import**s machen **public** Klassen und Methoden des Testframeworks JUnit sichtbar
  - org.junit ist ein Paket, das zu JUnit gehört
  - org.junit.Test und org.junit.Assert sind die Klassen Test und Assert in diesem Paket
  - Erklärung von **static** folgt ...
- Namenskonvention für Testklasse: `<IhreKlasse>Test`

Testen von Methoden: Beispiel isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Testen von Methoden: Beispiel isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
1  @Test
2  public void testIsUsable() {
3    SimpleTicket st = new SimpleTicket(1, 1);
4    assertTrue("Ticket should be usable", st.isUsable());
5  }
```

Testen von Methoden: Beispiel isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
1  @Test
2  public void testIsUsable() {
3    SimpleTicket st = new SimpleTicket(1, 1);
4    assertTrue("Ticket should be usable", st.isUsable());
5  }
```

- @Test: Markierung (Annotation) für eine Testmethode
- SimpleTicket st: Deklaration einer lokalen Variable mit Typ
- **new** SimpleTicket (1, 1): Erzeuge ein neues Objekt der Klasse **SimpleTicket** und rufe den Konstruktor mit den Parameter (1, 1) auf
- st.isUsable(): Aufruf der Method auf dem Objekt st
- assertTrue(message, condition):
  Prüft, ob condition erfüllt ist. Sonst Exception.

Testen von Methoden: isUsable(), Teil 2

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Testen von Methoden: isUsable(), Teil 2

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
1  @Test
2  public void testIsUsable2() {
3    SimpleTicket st = new SimpleTicket(1, 1);
4    st.stamp(4711, 1);
5    assertFalse("Ticket should not be usable", st.isUsable());
6    st.stamp(5000, 2);
7    assertFalse("Ticket should still be unusable", st.isUsable());
8  }
```

Testen von Methoden: isUsable(), Teil 2

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
1  @Test
2  public void testIsUsable2() {
3    SimpleTicket st = new SimpleTicket(1, 1);
4    st.stamp(4711, 1);
5    assertFalse("Ticket should not be usable", st.isUsable());
6    st.stamp(5000, 2);
7    assertFalse("Ticket should still be unusable", st.isUsable());
8  }
```

- assertFalse(message, condition):
  Prüft, ob condition *nicht* erfüllt ist. Sonst Exception.

Vermeiden von sinnlosen Objekten

Vermeiden von sinnlosen Objekten

Erinnerung (Felder und Konstruktor von SimpleTicket)

```
1  // Preisstufe 1, 2, 3
2  private int level;
3  // Kind = 0, Erwachsener = 1
4  private int category;
5  // ...
6  public SimpleTicket(int level, int category) {
7    this.level = level;
8    this.category = category;
9  }
```

Vermeiden von sinnlosen Objekten

Erinnerung (Felder und Konstruktor von SimpleTicket)

```
1  // Preisstufe 1, 2, 3
2  private int level;
3  // Kind = 0, Erwachsener = 1
4  private int category;
5  // ...
6  public SimpleTicket(int level, int category) {
7    this.level = level;
8    this.category = category;
9  }
```

Sinnvolle und sinnlose Objekte

- Keine Restriktion der Argumente beim Aufruf des Konstruktors
- Sinnvoll: **new** SimpleTicket(1, 1); **new** SimpleTicket(1, 0)
- Sinnlos: **new** SimpleTicket(0, 3); **new** SimpleTicket(−1, 42); **new** SimpleTicket(−999, −999)

Vermeiden der Konstruktion von sinnlosen Objekten

Wanted

Sinnlose Objekte können nicht erzeugt werden, d.h.,
alle Objekte sind sinnvoll.

Zwei Strategien

1. Fehlermeldung beim Erzeugen (Factory Method)
   - \+ geht immer, auch bei komplizierten Bedingungen an das Objekt
   - – potentieller Fehler zur Laufzeit
2. Einschränkung des Wertebereichs
   - – nur Bedingungen an einzelne Konstruktorargumente
   - \+ keine Laufzeitfehler

Strategie #1: Absicherung durch Factory Method

- Eine Java Methode kann einen Fehlerzustand durch eine *Exception* melden.
- Eine Exception wird durch eine **throw**-Anweisung ausgelöst und kann durch eine **catch**-Anweisung abgefangen werden.
- Jede Exception wird durch ein geeignetes Exception-Objekt repräsentiert

$\Rightarrow$ Die **throw**-Anweisung nimmt ein Exception-Objekt als Parameter

- Konvention: vermeide Exceptions in Konstruktoren

Strategie #1: Absicherung durch Factory Method

- Eine Java Methode kann einen Fehlerzustand durch eine *Exception* melden.
- Eine Exception wird durch eine **throw**-Anweisung ausgelöst und kann durch eine **catch**-Anweisung abgefangen werden.
- Jede Exception wird durch ein geeignetes Exception-Objekt repräsentiert

$\Rightarrow$ Die **throw**-Anweisung nimmt ein Exception-Objekt als Parameter

- Konvention: vermeide Exceptions in Konstruktoren

Lösung: Das Pattern *factory method*

Verstecke den Konstruktor und definiere (eine) Methode(n) zur Konstruktion von Objekten. Diese wirft eine Exception, falls die Argumente fehlerhaft sind.

Strategie #1: Absicherung durch Factory Method

Implementierung Teil 1

```
1  private SimpleTicket(int level, int category) {
2    this.level = level;
3    this.category = category;
4  }
5  // next up: constructor method create()
```

- Sichtbarkeit des Konstruktors wird **private**.
- Anstelle von **new** SimpleTicket(l, c) verwende Aufrufe der factory-Methode create (nächste Folie):
  SimpleTicket.create (l, c)
- Die factory-Methode hat die gleichen Parameter wie der Konstruktor, aber sie prüft ihre Legalität bevor das Objekt erzeugt wird.
- Die factory-Methode ist eine **static** Methode: Sie hängt nur von der Klasse ab, nicht von einem spezifischen Objekt.

Strategie #1: Absicherung durch Factory Method

Implementierung Teil 2

```
1  /**
2   * Safe constructor method for simple tickets.
3   * @param level must be 1,2,3
4   * @param category must be 0,1
5   * @return a new legal SimpleTicket
6   */
7  public static SimpleTicket create(int level, int category) {
8    if (level < 1 || level > 3 || category < 0 || category > 1 ) {
9      throw new IllegalArgumentException("Illegal level or category");
10   }
11   return new SimpleTicket(level, category);
12 }
```

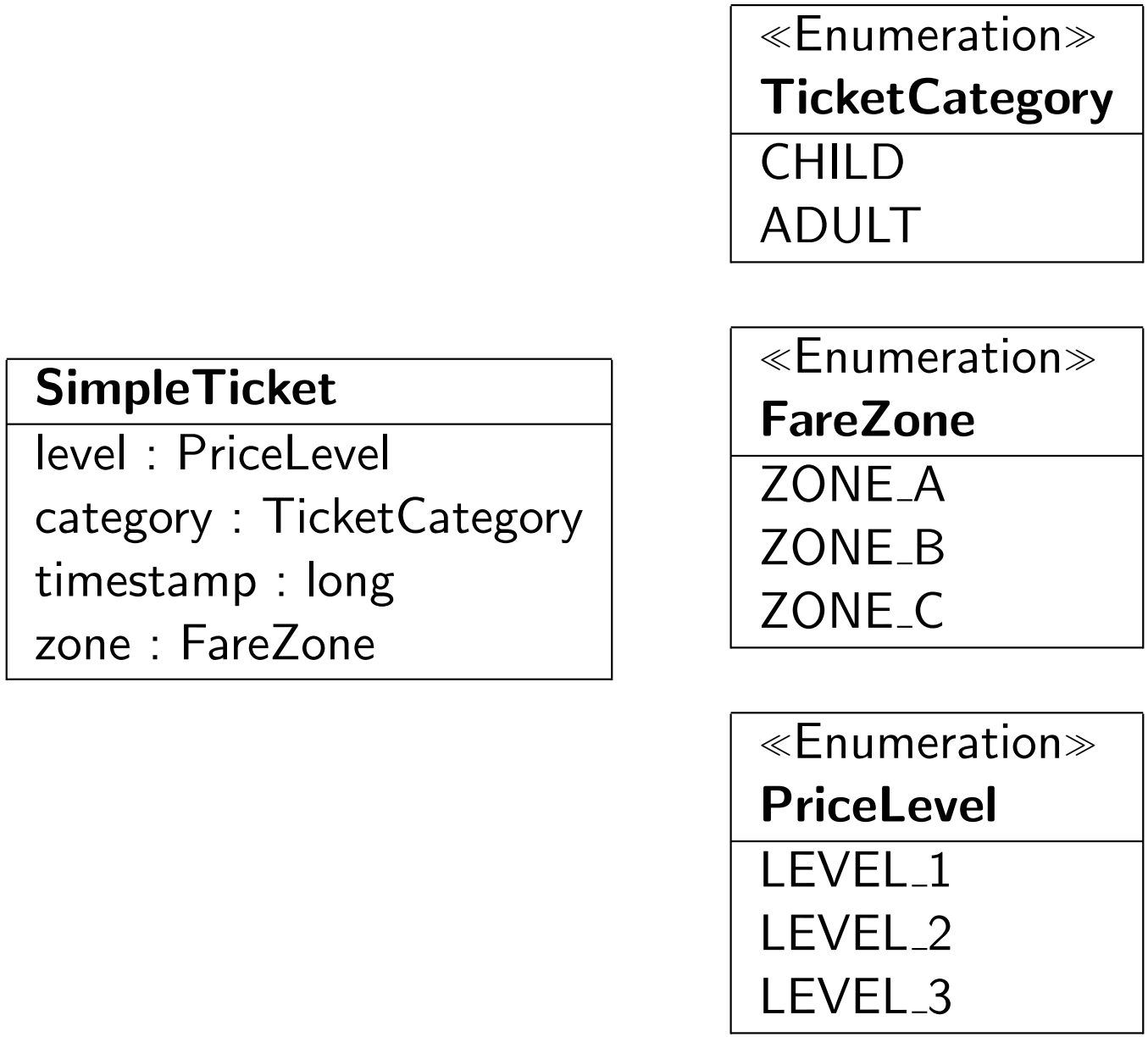
- Eine **static** Methode hängt nicht an einem Objekt, sondern kann direkt über die Klasse als SimpleTicket.create() aufgerufen werden.
- IllegalArgumentException ist ein vordefinierter Konstruktor für ein Exception Objekt. Das String-Argument ist Teil der Fehlermeldung.

Strategie #2: Einschränkung des Wertebereichs

Aufzählungstyp

- Anstatt die Kategorie “Kind” oder “Erwachsener” durch Zahlen zu kodieren, verwende einen *Aufzählungstyp* (*enum*).
- Ein Aufzählungstyp
  - ist definiert durch die Liste seiner benannten Werte, *Elemente* genannt
  - dient der Modellierung einer endlichen Zustandsmenge
- Definierbar im Klassendiagramm und im Java-Code.

Klassendiagramm (Revision mit Aufzählungstyp)



Aufzählungstypen **TicketCategory** und **FareZone**

```
1 package lesson_02;
2
3 public enum TicketCategory {
4         CHILD, ADULT;
5 }
```

```
1 package lesson_02;
2
3 public enum FareZone {
4         ZONE_A, ZONE_B, ZONE_C;
5 }
```

- Syntax: Sichtbarkeit, Schlüsselwort enum, Name (groß)
- Elemente des Aufzählungstyps: Bezeichner, nur Großbuchstaben oder Unterstriche

Änderungen in **SimpleTicket**

Vorher

```
9       // Preisstufe 1, 2, 3
10      private int level;
11      // Kind = 0, Erwachsener = 1
12      private int category;
13      // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
14      // nicht entwertet=0, ungültig=1
15      private long timestamp;
16      // Ort der Entwertung: Zone A=1, B=2, C=3
17      private int zone;
```

Änderungen in **SimpleTicket**

Nachher

```
4       // new: Preisstufe: LEVEL_1, LEVEL_2, LEVEL_3
5       private PriceLevel level;
6       // new: Kind = CHILD, Erwachsener = ADULT
7       private TicketCategory category;
8       // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
9       // nicht entwertet=0, ungültig=1
10      private long timestamp;
11      // new: Ort der Entwertung: ZONE_A, ZONE_B, ZONE_C
12      private FareZone zone;
```

Änderungen in **SimpleTicket.validate()**

Vorher

```
1  public boolean validate(int c, long t, int z) {
2    // 1. stamped exactly once?
3    boolean result = (this.timestamp != 0) && (this.timestamp != 1);
4    // 2. passenger category less than ticket category?
5    result = result && (c <= category);
6    // 3. ticket expired?
7    long timediff = t - timestamp;
8    result = result && (timediff <= level * 60 * 60 * 1000);
9    // 4. ticket used in valid zone?
10   int leveldiff = Math.abs(zone - z);
11   result = result && (leveldiff < level);
12   return result;
13 }
```

- Operationen in Zeile 5 (numerischer Vergleich), 8 (Rechnung), 11 (numerischer Vergleich) sind auf Aufzählungstypen nicht definiert.

Änderungen in **SimpleTicket.validate()**

Nachher

```
1  public boolean validate(TicketCategory c, long t, FareZone z) {
2  ...
3     // 2. passenger category less than ticket category?
4     result = result && (c.ordinal() <= category.ordinal());
5  ...
6     // 4. ticket used in valid zone?
7     int leveldiff = Math.abs(zone.ordinal() - z.ordinal());
8  ...
9     return result;
10 }
```

- Nachher: geänderte Signatur
- *Abhilfe*: auf Aufzählungstypen ist Methode **int** ordinal() definiert.
- ordinal() bildet auf $0, 1, 2, 3, \ldots$ ab

Änderungen in **SimpleTicket.validate()**

Vorher

```
1 public boolean validate(int c, long t, int z) {
2 ...
3   // 3. ticket expired?
4   long timediff = t - timestamp;
5   result = result && (timediff <= level * 60 * 60 * 1000);
6 ...
7   return result;
8 }
```

- Fürs Rechnen mit level ist ordinal() nicht passend.
- Besser wäre LEVEL_1 $\mapsto$ 1, LEVEL_2 $\mapsto$ 2 etc
- Das kann durch einen *Aufzählungstyp mit Attributen* erreicht werden.

Einschub: Aufzählungstyp mit Attribut

```
2  public enum PriceLevel {
3      LEVEL_1(1), LEVEL_2(2), LEVEL_3(3);
4      // corresponding numeric level
5      private final int level;
6      private PriceLevel(int level) {
7          this.level = level;
8      }
9      // getter method
10     public int getLevel() {
11         return this.level;
12     }
13 }
```

- Für jedes Attribut wird ein **private final** Feld definiert.
- **final** bedeutet, dass sich der Wert des Feldes nach der Initialisierung durch den Konstruktor nicht mehr ändern kann.
- Konstruktoraufruf mit Parametern in Klammern (Zeile 4)
- **public** *Getter-Methode* nur zum Lesen des Feldes

Änderungen in **SimpleTicket.validate()**

Nachher

```
1 public boolean validate(TicketCategory c, long t, FareZone z) {
2 ...
3   // 3. ticket expired?
4   long timediff = t - timestamp;
5   result = result && (timediff <= level.getLevel() * 60 * 60 * 1000);
6 ...
7   return result;
8 }
```

- Erstes Ziel erreicht: keine sinnlosen **SimpleTicket** Objekte mehr möglich!
- Aber die Klasse enthält weitere *Anti-Pattern*, die wir eliminieren wollen ...

Benennen von Konstanten

Benennen von Konstanten

Betrachte

```
1  // 3. ticket expired?
2  long timediff = t - timestamp;
3  result = result && (timediff <= level.getLevel() * 60 * 60 * 1000);
```

- Was ist die Bedeutung der Konstante $60 * 60 * 1000$?

Benennen von Konstanten

Betrachte

```
1  // 3. ticket expired?
2  long timediff = t - timestamp;
3  result = result && (timediff <= level.getLevel() * 60 * 60 * 1000);
```

- Was ist die Bedeutung der Konstante $60 * 60 * 1000$?
- Natürlich ist es die Anzahl der Millisekunden pro Stunde!
- Zur Vermeidung von Missverständnissen und zur Dokumentation sollte *jede Konstante sinnvoll benannt* werden.
- Ausnahmen: 0, 1, **true**, **false** etc

Benennen von Konstanten

Betrachte

```
1  // 3. ticket expired?
2  long timediff = t - timestamp;
3  result = result && (timediff <= level.getLevel() * 60 * 60 * 1000);
```

- Was ist die Bedeutung der Konstante $60 * 60 * 1000$?
- Natürlich ist es die Anzahl der Millisekunden pro Stunde!
- Zur Vermeidung von Missverständnissen und zur Dokumentation sollte *jede Konstante sinnvoll benannt* werden.
- Ausnahmen: 0, 1, **true**, **false** etc
- Sinnvolles Muster: Lege separate Klasse zur Aufnahme der Konstantendefinitionen an.

Klasse für Konstantendefinitionen

```
1 public class Tickets {
2   public static final int MILLISECONDS_PER_HOUR = 60 * 60 * 1000;
3 }
```

- **static** — gehört zu keinem Objekt
- **final** — Konstante, die nach Initialisierung nicht mehr verändert wird
- Name in Großbuchstaben oder Unterstrichen
- Initialisiert durch konstanten Ausdruck
- Verwendung durch Tickets.MILLISECONDS_PER_HOUR

Vermeiden von magischen Werten

Vermeiden von magischen Werten

```
1    // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
2    // nicht entwertet=0, ungültig=1
3    private long timestamp;
```

- Die Werte 0 und 1 sind *magisch* für timestamp.
- Sie sind prinzipiell gültige Zeitstempel, bei denen es höchst unwahrscheinlich ist, dass sie auftreten.
- Besser und sicherer: verfahre nach dem Prinzip „Nichts ist unmöglich“

Vermeiden von magischen Werten

Lösung

```
1    // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
2    private long timestamp;
3
4    private static final int MAX_RIDES = 1;
5    private int nrOfStamps;
```

- Neues Feld nrOfStamps, das die Anzahl der Stempel zählt.
- MAX_RIDES ist die maximale Anzahl von Stempeln, bevor der Fahrschein nicht mehr entwertet werden kann.
- timestamp unterliegt keinerlei Einschränkungen mehr.

Vermeiden von magischen Werten

Anpassung der Methoden

```
1  public boolean isUsable() {
2    return nrOfStamps < MAX_RIDES;
3  }
```

```
1  public void stamp(long t, FareZone z) {
2    if (isUsable()) {
3      zone = z;
4      timestamp = t;
5    }
6    nrOfStamps++;
7  }
```

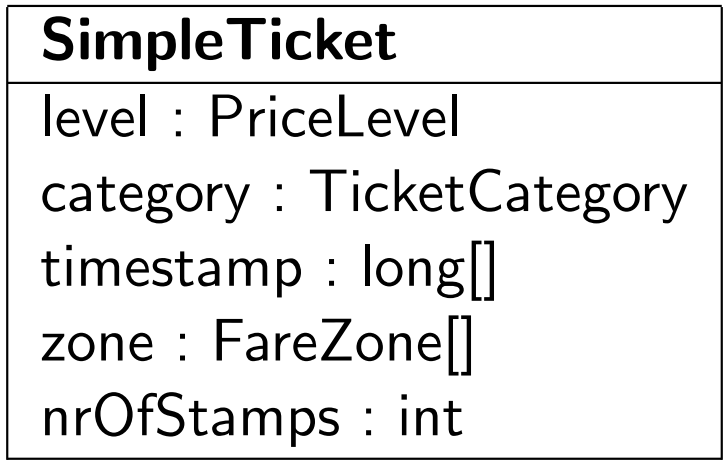
Zusammengesetzte Klassen

Erweiterung: Mehrfahrtenkarte

Spezifikation

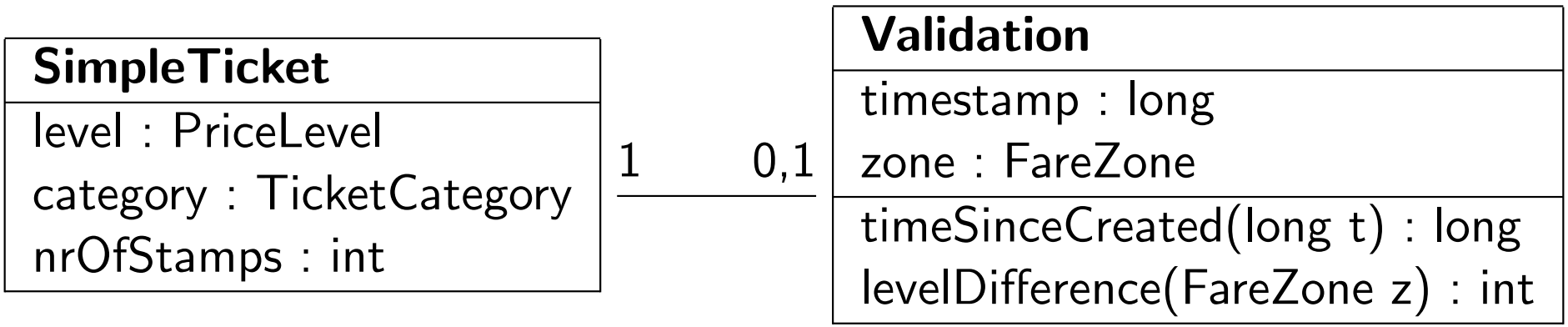
Das Verkehrsunternehmen möchte nun 2x4-Fahrtenkarten ausgeben. Die 2x4-Fahrtenkarte hat eine Preisstufe (1, 2, 3), sie ist für Erwachsene oder Kinder und sie kann acht Mal entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. (usw. wie gehabt)

Klassendiagramm (Versuch)



- Erste Idee
  - Erweitere timestamp und zone zu Arrays
  - Korrespondierende Einträge bilden einen Stempel ab.
  - Nachteil: Korrespondenz ist implizit

Validation wird Klasse



Besser

- Eigene Klasse **Validation**
- *Assoziation* zwischen **SimpleTicket** und **Validation**
- Jedes **SimpleTicket** besitzt null oder eine **Validation**
- Ein **Validation** Objekt kapselt einen Stempel
- Operationen auf Zeitstempeln *müssen* in die Klasse **Validation** verlagert werden!

Validation in Java

```
1  public class Validation {
2    // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
3    private final long timestamp;
4    // Ort der Entwertung
5    private final FareZone zone;
6
7    public Validation(long timestamp, FareZone zone) {
8      this.timestamp = timestamp;
9      this.zone = zone;
10   }
11   /** ... */
12   public long timeSinceCreated(long t) {
13     return t - timestamp;
14   }
15   /** ... */
16   public int levelDifference(FareZone z) {
17     return Math.abs(zone.ordinal() - z.ordinal());
18   }
19 }
```

Anpassung in **SimpleTicket**

```
1   /**
2    * Stamp the ticket.
3    * @param t time of validation (in millisec)
4    * @param z location of validation (zone)
5    */
6   public void stamp(long t, FareZone z) {
7     if (this.isUsable()) {
8       validation = new Validation(t, z);
9     }
10    nrOfStamps++;
11  }
```

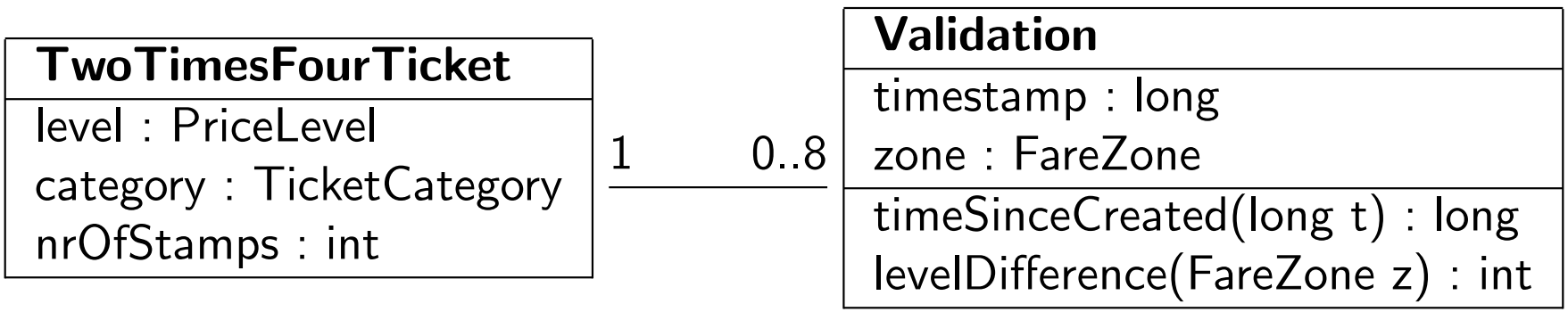
- Die stamp() Methode erzeugt ein neues **Validation** Objekt.

Anpassung in **SimpleTicket**

```
1   /**
2    * Check if a ticket is valid given the passenger's category, ...
3    */
4   public boolean validate(TicketCategory c, long t, FareZone z) {
5     boolean result = nrOfStamps == 1;
6     result = result && (c.ordinal() <= category.ordinal());
7     result = result && (validation.timeSinceCreated(t) <=
8                           level.getLevel() * Tickets.MILLISECONDS_PER_HOUR);
9     result = result && (validation.levelDifference(z) < level.getLevel());
10    return result;
11  }
```

- validate() bearbeitet nur eigene Felder
- Alle Berechnungen an Zeitstempeln sind in die Klasse **Validation** verschoben

Mehrfahrtenkarte mit **Validation**



Änderung

- *Assoziation* zwischen **TwoTimesFourTicket** und **Validation**
- Jedes **TwoTimesFourTicket** besitzt *null bis acht* **Validation** Objekte
- (Vielfachheit / Multiplicity einer Assoziation)
- Mögliche Implementierung: mit Array

Implementierung der Mehrfahrtenkarte (Auszug)

```
1  public class TwoTimesFourTicket {
2    // ...
3    private Validation[] validation;
4
5    public TwoTimesFourTicket(PriceLevel level, TicketCategory category) {
6            this.level = level;
7            this.category = category;
8            this.validation = new Validation[MAX_RIDES];
9            this.nrOfStamps = 0;
10   }
11
12   public void stamp(long t, FareZone z) {
13     if (isUsable()) {
14       validation[nrOfStamps] = new Validation(t, z);
15     }
16     nrOfStamps++;
17   }
18 }
```

Merkmale zusammengesetzter Klassen

Merkmale

- **SimpleTicket** und **TwoTimesFourTicket** sind Beispiele für *zusammengesetzte Klassen*.
- Zusammengesetzte Klassen sind mit anderen Klassen assoziiert oder besitzen Attribute, die selbst Objekte sind
- Methodenentwurf:
  Eine Operation auf einer zusammengesetzten Klasse verwendet
  - Attribute und Operationen der Klasse
  - Operationen auf Objekten von assoziierten Klassen
  - Operationen auf Objekten in Attributen

Testen für zusammengesetzte Klassen

- Entwurf von Operationen geschieht top-down
  1. Entwerfe Operation auf zusammengesetzter Klasse unter Annahme von Operationen auf assoziierten Klassen
  2. Entwerfe geforderte Operationen auf den assoziierten Klassen
- Testen geschieht bottom-up
  1. Erstelle Tests für untergeordnete und assoziierte Klassen
  2. Erstelle Tests für zusammengesetzte Klasse
- Im Beispiel
  1. Erstelle Tests für **Validation**
  2. Erstelle Tests für **SimpleTicket** und **TwoTimesFourTicket**

Fahrkarten mischen

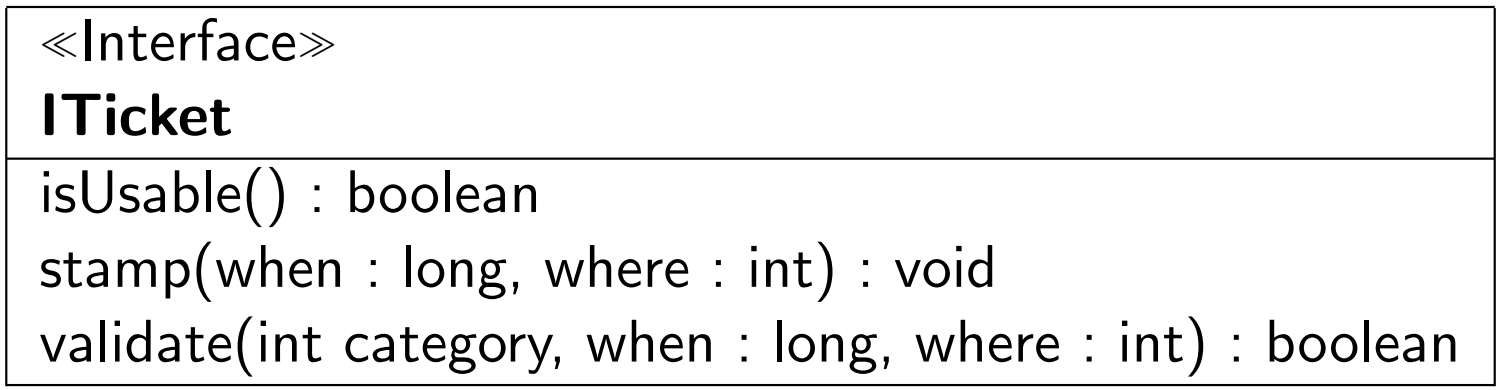
Erweiterung: Fahrkarten mischen

Spezifikation

Das Verkehrsunternehmen möchte Einzelfahrscheine und auch 2x4-Fahrtenkarten ausgeben. (usw. wie gehabt)
Ein Fahrgast möchte mit Einzelfahrscheinen und 2x4-Fahrtenkarten gleichermaßen umgehen.

- D.h., ein Einzelfahrschein und eine 2x4-Fahrtenkarte muss sich auf eine gewisse Art gleich verhalten.
- Sie müssen beide das gleiche *Interface* implementieren!

Klassendiagramm: Interface für Fahrscheine



- Wie Klasse, aber Stereotyp „Interface“
- Keine Attribute (Konstanten möglich)
- Operationen wie Klasse

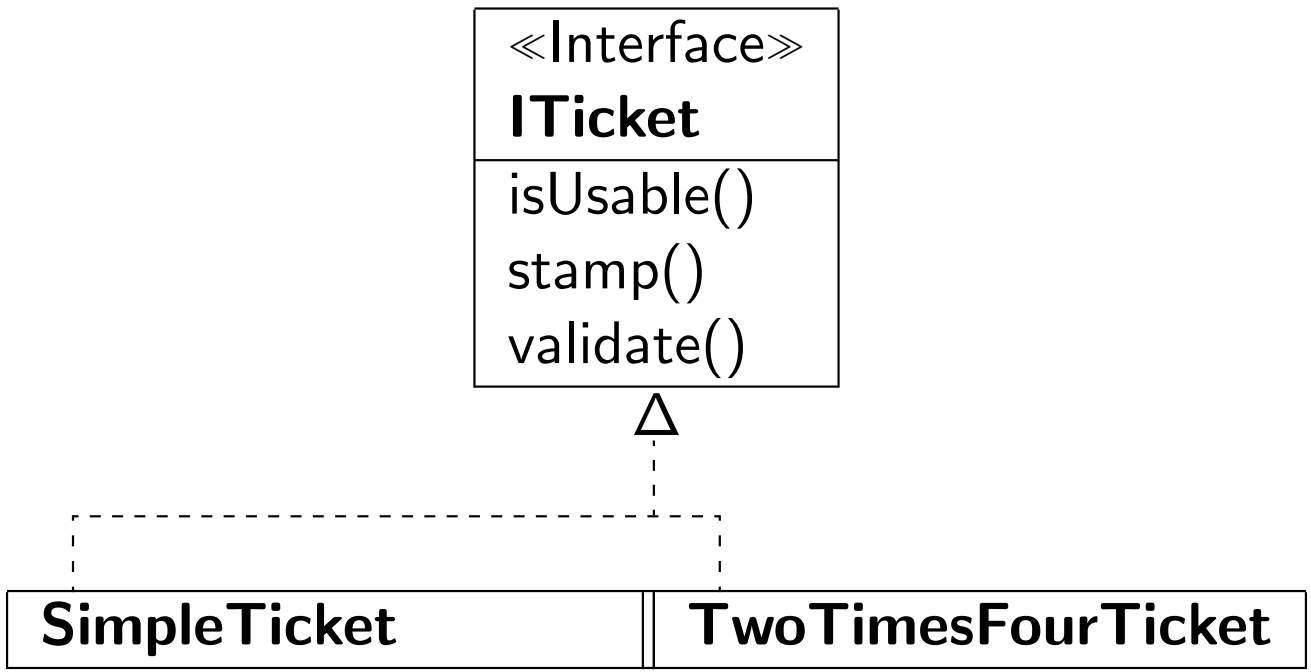
Interface in Java

```
1 package lesson_02;
2
3 public interface ITicket {
4   public boolean isUsable();
5   public void stamp(long t, FareZone z);
6   public boolean validate(TicketCategory c, long t, FareZone z);
7 }
```

- Jede Klasse, die diese Methoden **public** implementiert, kann das Interface implementieren
- Intention durch die **implements**-Klausel im Klassenheader deklariert.
- Dann müssen diese Methoden vorhanden sein.

```
1 public class SimpleTicket implements ITicket { ... }
2 public class TwoTimesFourTicket implements ITicket { ... }
```

Implementierungsbeziehung im Klassendiagramm



- Gestrichelte Linie mit offenem Pfeilkopf von Klasse zu implementiertem Interface

Interfacetypen

- Ein Interface definiert einen *Referenztypen*, genau wie eine Klassendeklaration.
- Ein Interface (-typ) kann überall benutzt werden, wo ein Typ erwartet wird: lokale Variable, Felder, Parameter, usw.
- Wenn ein Interfacetyp als Parameter erwartet wird, dann darf ein Objekt einer beliebigen Klasse, die das Interface implementiert, übergeben werden.
- Gleiches gilt für Zuweisungen an Variable oder Felder vom Interfacetyp.
- Jede implementierende Klasse ist ein *Subtyp* des Interfacetyps. Also
  - **SimpleTicket** ist Subtyp von **ITicket** und
  - **TwoTimesFourTicket** ist Subtyp von **ITicket**.

Verwendung von Interfacetypen

Vereinigung von Klassen

- Ein Typ I ist eine *Vereinigung von Klassen* $C_1, \ldots, C_n$, falls jedes Objekt vom Typ I genau eine der Klassen $C_1, \ldots, C_n$ besitzt.
- Die Operationen auf I sind die gemeinsamen Operationen von $C_1, \ldots, C_n$.
- Implementierung durch *Composite Pattern*.
- Der Vereinigungstyp I wird durch ein Interface implementiert.

Beispiel für Vereinigung

ITicket ist Vereinigung von **SimpleTicket** und **TwoTimesFourTicket**