Programmieren in Java

Vorlesung 03: Abstraktion mit Klassen

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

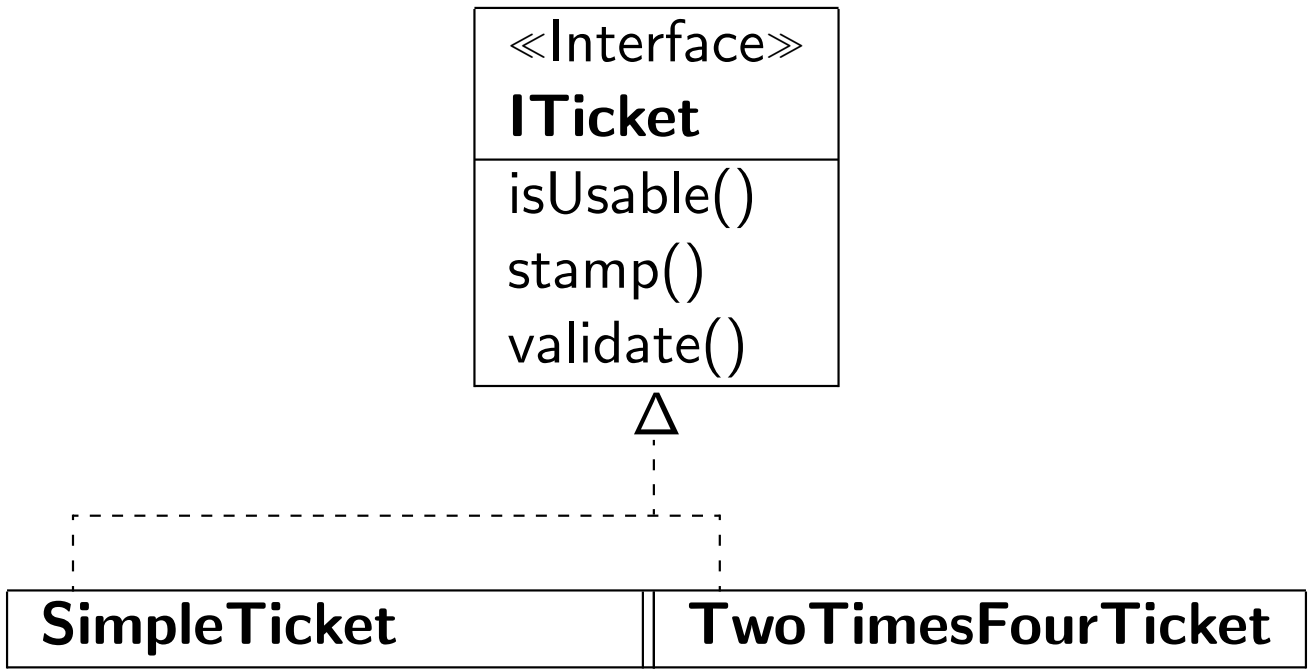
SS 2015

Inhalt

Executive Summary

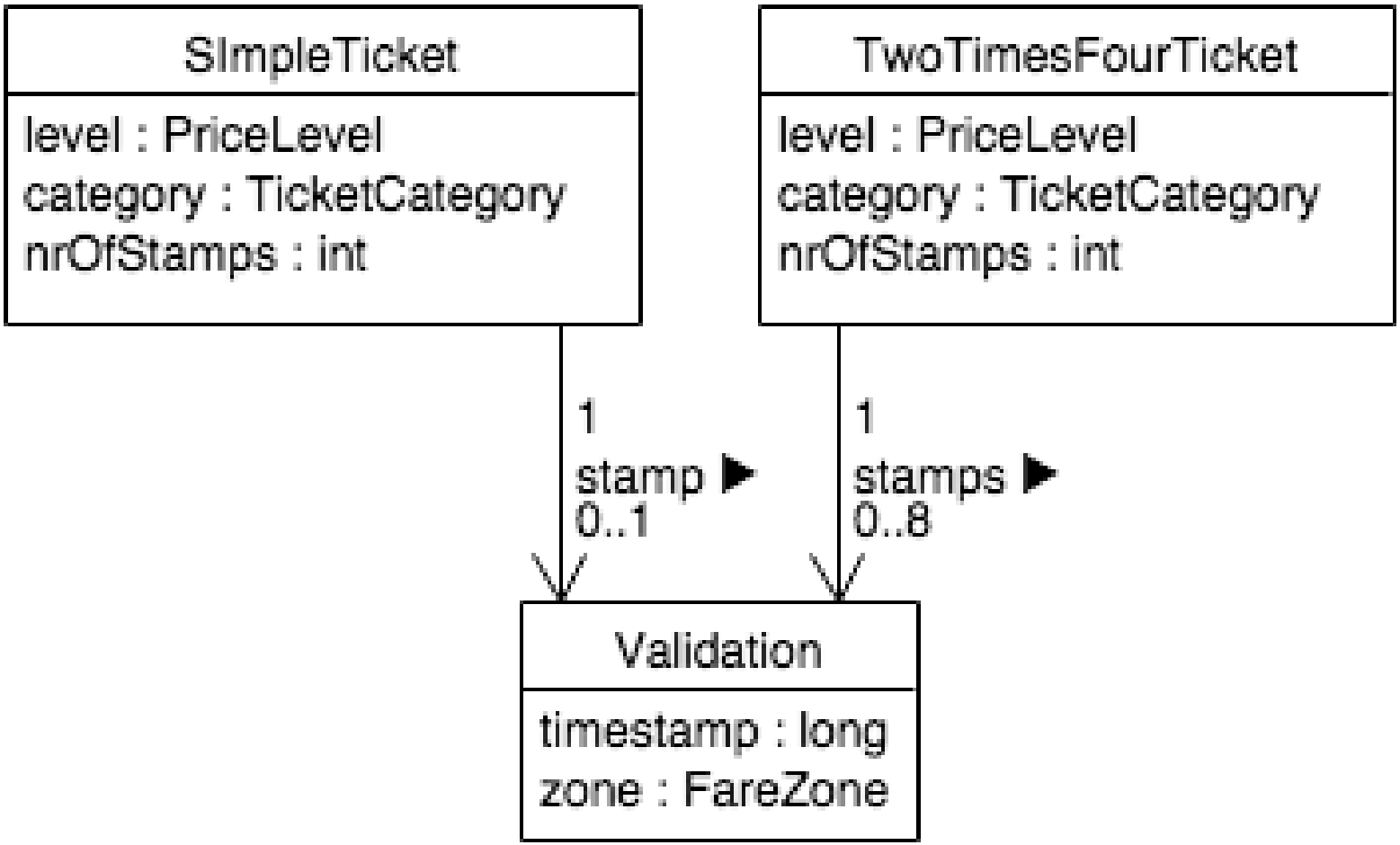
1. Listen und Iteratoren
   - Das Java Collection Framework
   - Interfaces und Klassen (Implementierungen) zum Bearbeiten von Listen, Mengen und ähnlichen Datenstrukturen
2. Refactoring
   - Wiederholtes Anpassen des Designs an veränderte Anforderungen
   - Verbesserungen der Struktur des Codes
3. Abstrakte Klassen
   - Abstraktion durch Zusammenfassen von Merkmalen aus logisch zusammengehörigen Klassen
   - Liften der Definition von gemeinsamem Verhalten (Operationen)
4. Rekursive Assoziation
5. Überladung, statischer und dynamischer Typ

Erinnerung: Vereinigung von Klassen



- Beide Ticket-Klassen implementieren das selbe Interface
- Aber es gibt weitere Gemeinsamkeiten.

Vergleich **SimpleTicket — TwoTimesFourTicket**



Vergleich / Code

```
1 public class SimpleTicket
2    implements ITicket {
3    // attributes
4    private PriceLevel level;
5    private TicketCategory category;
6    private int nrOfStamps;
7    // association
8    private Validation stamp;
9 }
```

```
1 public class TwoTimesFourTicket
2    implements ITicket {
3    // attributes
4    private PriceLevel level;
5    private TicketCategory category;
6    private int nrOfStamps;
7    // association
8    private Validation[] stamps;
9 }
```

- Ziel: Abstraktion der Gemeinsamkeiten
- Hauptunterschied: Unterschiedliche Repräsentation der Assoziation
- Gesucht: Repräsentation für 0, 1 oder mehr **Validation** Objekte

Listen und Iteratoren

Listen

- Das Interface List<X> ist eine Abstraktion zum Bearbeiten von Sequenzen von Elementen vom Typ X.
- List<X> ist ein *generischer Typ*, bei dem für X ein beliebiger Referenztyp (Klasse, Interface, ...) eingesetzt werden kann.
- Beispiele
  - List<Integer> Liste von Zahlen
  - List<Object> Liste von beliebigen Objekten
  - List<Validation> Liste von **Validation** Objekten

Operationen auf Listen (Auswahl)

```
1  package java.util;
2
3  public interface List<X> {
4     // add new element at end of list
5     boolean add (X element);
6     // get element by position
7     X get (int index);
8     // nr of elements in list
9     int size();
10    // further methods omitted
11 }
```

- Weitere Methoden in der *Java API Dokumentation*
- Um eine Liste zu erzeugen, muss eine konkrete Implementierung gewählt werden
- Beispiele: **ArrayList**, **LinkedList**, **Stack**, **Vector**, ...
- Unterschiedliche Eigenschaften, Auswahl nach Anwendungsfall

Beispiel: Liste

```
1  public class ListTest {
2    @Test
3    public void testList() {
4      List<Integer> il = new LinkedList<Integer>();
5      assertEquals(0, il.size());
6      il.add(1);
7      assertEquals(1, il.size());
8      il.add(4);
9      assertEquals(2, il.size());
10     il.add(9);
11     assertEquals(3, il.size());
12     assertEquals((int)1, (int)il.get(0));
13     assertEquals((int)4, (int)il.get(1));
14     assertEquals((int)9, (int)il.get(2));
15   }
16 }
```

Durchlaufen von Listen

- Das Durchlaufen einer Liste kann mittels `get` geschehen.
- Erfordert Manipulation von Indexen und der Länge der Liste
- Generische Möglichkeit: Durchlaufen mittels *Iterator*

Das Interface **Iterable**

```
1 public interface Iterable<X> {
2   Iterator<X> iterator()
3 }
```

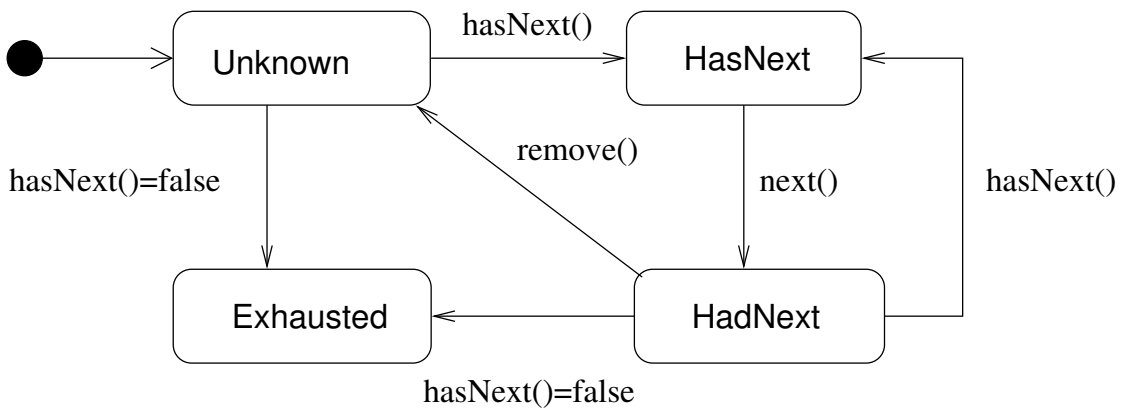
- Jede Liste kann einen *Iterator* liefern, mit dem die Liste durchlaufen werden kann.
- (Dazwischen liegt das **Collection** Interface.)

- Im Klassendiagramm steht der offene Pfeil für *Vererbung*: **List** erbt alle Operationen von **Collection**, das wiederum von **Iterable** erbt.

Das Interface **Iterator**

```
1 public interface Iterator<X> {
2   // true if there is a next element in the list
3   boolean hasNext();
4   // obtain next element and advance
5   X next();
6   // remove the last element returned by next (optional)
7   void remove();
8 }
```

StateChart Diagramm: Korrekte Verwendung von Iterator



Codemuster für **Iterator**

```
1  Iterable<X> collection = ...;
2  Iterator<X> iter = collection.iterator();
3  while (iter.hasNext()) {
4    X element = iter.next();
5    // process element
6    if (no_longer_needed(element)) {
7      iter.remove();
8    }
9  }
```

- Konkretes Beispiel folgt

For-Schleife mit **Iterator**

- Falls Löschen nicht erforderlich ist, kann die explizite Verwendung der **Iterator** Methoden vermieden werden
- Stattdessen: Verwende eine For-Schleife

```
1  Iterable<X> collection = ...;
2  for (X element : collection) {
3    // process element
4  }
```

Abstrakte Klassen

Revision: **SimpleTicket** und **TwoTimesFourTicket**

```java
public class SimpleTicket
   implements ITicket {
   // attributes
   private final PriceLevel level;
   private final TicketCategory category;
   private final int maxNrOfStamps;
   // association
   private final List<Validation> stamps;
}
```

```java
public class TwoTimesFourTicket
   implements ITicket {
   // attributes
   private final PriceLevel level;
   private final TicketCategory category;
   private final int maxNrOfStamps;
   // association
   private final List<Validation> stamps;
}
```

Änderungen

- maxNrOfStamps statt nrOfStamps
- Vorteil: die Felder ändern sich zur Laufzeit nicht mehr und können als **final** deklariert werden
- List<Validation> verallgemeinert die Typen Validation (0 oder 1 Objekt) und Validation[] (0 bis n Objekte für festes n)

Codeanpassung: **SimpleTicket** — Konstruktor

vgl package lesson_03a

```
1  public SimpleTicket(PriceLevel level, TicketCategory category) {
2    this.level = level;
3    this.category = category;
4    this.maxNrOfStamps = 1;
5    this.stamps = new LinkedList<Validation>();
6  }
```

- Alle **final** Felder *müssen* initialisiert werden
- Auswahl der Implementierung von List<Validation>
- **new** LinkedList<Validation>() ruft den *Konstruktor* von LinkedList ohne Parameter auf
- LinkedList ist eine vordefinierte *generische Klasse*
- Mehr dazu nächste Einheit.

Codeanpassung: **SimpleTicket** Methoden

```
1  public boolean isUsable() {
2    return this.stamps.size() < this.maxNrOfStamps;
3  }
4
5  public void stamp(long t, FareZone z) {
6    this.stamps.add(new Validation(t, z));
7  }
```

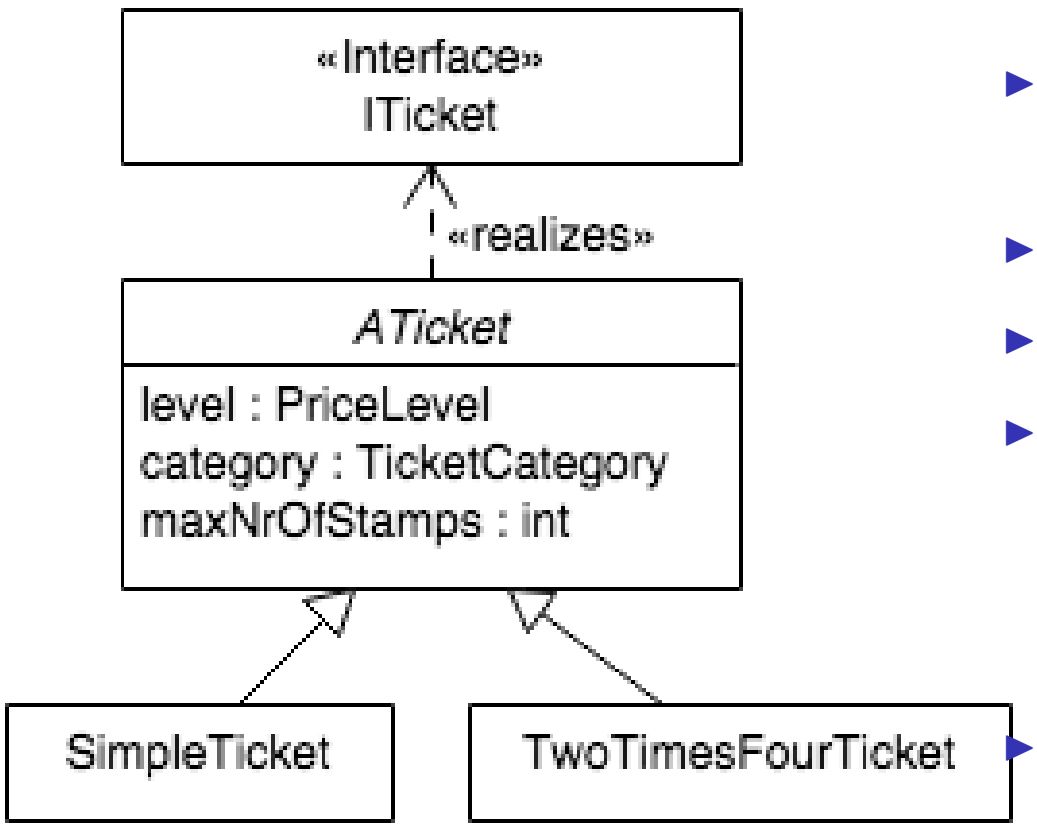
- stamps.size() zählt die Anzahl der Entwertungen. Ein extra Zähler ist nicht erforderlich.
- Es gibt keine Begrenzung der Anzahl der Elemente einer Liste. Daher vereinfacht sich der Code für stamp: Es wird einfach ein neuer Stempel hinzugefügt.
- Beobachtung: Der gleiche Code würde auch für ein **TwoTimesFourTicket** funktionieren.

Abstraktion mit Klassen

Einführen einer abstrakten Klasse

- Eine *abstrakte Klasse* kann gemeinsame Attribute und Operationen für eine Reihe von *konkreten Klassen* aufnehmen.
- Eine abstrakte Klasse besitzt keine eigenen Instanzen (Objekte).
- Eine abstrakte Klasse muss nicht alle Operationen implementieren.

Abstrakte Klasse im Klassendiagramm



- Auch eine abstrakte Klasse kann ein Interface implementieren!
- Name der abstrakten Klasse in *kursiver Schrift*
- Attribute
- (Operationen)
- **SimpleTicket** und **TwoTimesFourTicket** sind *Subklassen* (angezeigt durch den offenen Pfeil) von **ATicket**
- Sie erben alle Attribute und Operationen von **ATicket**.

Abstrakte Klasse im Java Code

```
1  public abstract class ATicket implements ITicket {
2    protected final PriceLevel level;
3    protected final TicketCategory category;
4    protected final List<Validation> stamps;
5    protected final int maxStamps;
6
7    protected ATicket(PriceLevel level,
8                      TicketCategory category,
9                      int maxStamps) {
10     this.level = level;
11     this.category = category;
12     this.maxStamps = maxStamps;
13     this.stamps = new LinkedList<Validation>();
14   }
15   public boolean isUsable() { ... }
16   public void stamp(long t, FareZone z) { ... }
17   // more elided
18 }
```

- Abstrakte Klasse angezeigt durch Schlüsselwort **abstract**
- Sichtbarkeit **protected**: sichtbar in allen Subklassen von **ATicket**
- **protected** Konstruktor kann nur vom Konstruktor einer Subklasse aufgerufen werden
- isUsable() und stamp() wie in letzter Anpassung von **SimpleTicket**

Subklasse im Java Code

```
1 public class SimpleTicket
2   extends ATicket {
3   public SimpleTicket(
4     PriceLevel level,
5     TicketCategory category)
6   {
7     super(level, category, 1);
8   }
9 }
```

```
1 public class TwoTimesFourTicket
2   extends ATicket {
3   public TwoTimesFourTicket(
4     PriceLevel level,
5     TicketCategory category)
6   {
7     super(level, category, 8);
8   }
9 }
```

- ZZTicket **extends** ATicket gibt an, dass ZZTicket Subklasse von ATicket ist
- ATicket heißt auch *Superklasse* von ZZTicket
- *Eine Klasse kann höchstens eine Superklasse haben!*
- Aber: *Eine Klasse kann mehrere Interfaces implementieren!*
- Im Konstruktor kann (nur als erstes) ein Konstruktor der Superklasse durch **super** (...) aufgerufen werden

Refactoring

Neue Anforderung

Spezifikation

Das Verkehrsunternehmen möchte zusätzlich auch Punktekarten ausgeben. Die Punktekarte gibt es mit insgesamt 20 Punkten. [...] Die benötigte Punktezahl richtet sich nach der Anzahl der je Fahrt berührten Tarifzonen (z.B. Preisstufe 1 für eine Tarifzone):

Preisstufe	1 Person
1	3 Punkte
2	5 Punkte
3	7 Punkte

Punktekarten sind ab Entwertung zeitlich gestaffelt gültig für die Fahrt in Zielrichtung [Preisstufe * 60 Minuten].
[Jeder Punkt wird durch einen Entwerterstempel entwertet.]

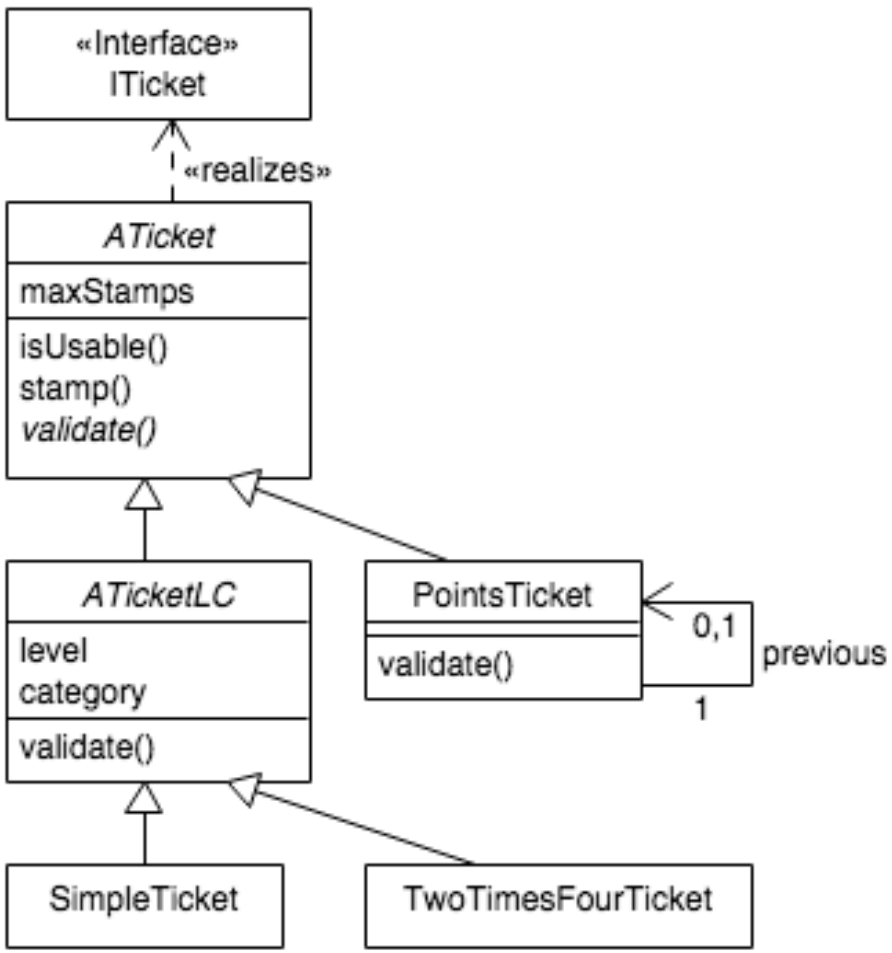
Reaktion: Refactoring

- Neue Klasse **PointsTicket** repräsentiert Punktekarten
- Die Punktekarte soll immer noch das **ITicket** Interface implementieren!
- Grund: der Rest des Programms sollte (nur) vom Interface **ITicket** abhängen, nicht von konkreten Implementierungen
- Die Punktekarte besitzt keine Preisstufe und unterscheidet nicht zwischen Kindern und Erwachsenen.

$\Rightarrow$ **ATicket** muss revidiert werden

- Allen Klassen gemeinsam: stamps und maxStamps
- Nicht in **PointsTicket**: level und category
- Also: zwei abstrakte Klassen ...

Klassendiagramm für Tickets (endgültig)



- ATicket.validate() ist *abstrakte Operation*, zu erkennen an *kursiver Schrift*
- Alle konkreten Subklassen von ATicket müssen validate() imlementieren!
- ATicketLC ist neue abstrakte Subklasse von ATicket mit level und category
- Neue Klasse PointsTicket mit *rekursiver Assoziation* previous auf sich selbst

Javacode für **ATicket**

```
1  public abstract class ATicket implements ITicket {
2    protected final List<Validation> stamps;
3    protected final int maxStamps;
4
5    protected ATicket(int maxStamps) {
6      if (maxStamps < 1) {
7        raise new IllegalArgumentException();
8      }
9      this.maxStamps = maxStamps;
10     this.stamps = new LinkedList<Validation>();
11   }
12   public boolean isUsable() { ... }
13   public void stamp(long t, FareZone z) { ... }
14   // abstract method
15   public abstract boolean
16     validate(TicketCategory c, long t, FareZone z);
17 }
```

Die abstrakte Methode validate . . .

- wird durch das Schlüsselwort **abstract** markiert.
- wird nicht implementiert. Es wird nur die Signatur der Methode angegeben.
- muss von jeder konkreten Subklasse implementiert werden.

Javacode für **ATicketLC**

```
1  public abstract class ATicketLC extends ATicket {
2    private final PriceLevel level;
3    private final TicketCategory category;
4
5    protected ATicketLC(PriceLevel level, ...) { ... }
6
7    public boolean validate(TicketCategory c, ...) {
8      int nrStamps = stamps.size();
9      boolean result = (nrStamps > 0)
10                   && (nrStamps <= maxStamps);
11     if (result) {
12       Validation validation =
13         stamps.get(nrStamps-1);
14       // same as before
15     }
16     return result;
17   }
18 }
```

ATicketLC ...

- ist abstrakt
- ist selbst Subklasse
- (Konstruktor ruft **super** Konstruktor auf)
- implementiert die Methode validate (abstrakt in Superklasse)
- validate testet die letzte Entwertung, falls eine vorhanden ist

Javacode für **PointsTicket**

```
1  public class PointsTicket extends ATicket {
2    private final static int MAX_STAMPS = 20;
3    public PointsTicket() { super(MAX_STAMPS); }
4
5    public boolean validate(TicketCategory c, long t, FareZone z) {
6      int nrStamps = getNrOfStamps();
7      boolean result = (nrStamps > 0) && (nrStamps <= MAX_STAMPS);
8      if (result) {
9        Validation validation = stamps.get(nrStamps-1);
10       int count = countValidations(validation); // *****
11       PriceLevel level;
12       if (count >= Tickets.STAMPS_FOR_LEVEL3) {
13         level = PriceLevel.LEVEL_3;
14       } else if (count >= Tickets.STAMPS_FOR_LEVEL2) {
15         level = PriceLevel.LEVEL_2;
16       } else if (count >= Tickets.STAMPS_FOR_LEVEL1) {
17         level = PriceLevel.LEVEL_1;
18       } else {
19         return false;
20       }
21       result = result && (validation.timeSinceCreated(t) <= level.getLevel() * TicketsM
```

Stempel zählen

- Zum Implementieren von validate() benötigen wir eine Methode, die die Stempel zählt.
- Ansatz zum Entwurf
  - Nehme zunächst an, dass eine (Hilfs-) Methode countValidations existiert
  - Diese Methode ist **private**, also außerhalb der Klasse nicht sichtbar
  - Signatur: **int** countValidations (Validation validation)
  - Gewünschte Funktion: zähle die Anzahl der Stempel die gleich validation sind

Stempel zählen, erster Versuch

```
1  private int countValidations(Validation validation) {
2    int count = 0;
3    if (this.stamps.size() <= MAX_STAMPS) { // ignore if too many stamps
4      for (Validation stamp : this.stamps) {
5        if (validation.equals(stamp)) { count++; }
6      }
7    }
8    return count;
9  }
```

- Eine verstempelte Karte ist komplett ungültig
- Verwende eine for-Schleife um die Liste von Stempeln zu durchlaufen
- Verwende die equals() Methode um validation und stamp zu vergleichen
- Exkurs: Herkunft und Implementierung von equals()

Exkurs: Vergleichen

Die Klasse **Object**

Jede Klasse erbt von der Klasse **Object**, die in Java vordefiniert ist. Dort sind einige Methoden definiert, die für Objektvergleiche relevant sind:

```
1  public class Object {
2    public boolean equals(Object obj) {
3      return this == obj;
4    }
5    public int hashCode() { ... }
6    public final Class<?> getClass() { ... }
7    ...
8  }
```

- *Die Methoden* `equals` *und* `hashCode` *sollten im Normalfall überschrieben werden!*
- (Überschreiben = in der Subklasse erneut definieren)
- `getClass` *kann nicht überschrieben werden*, da mit **final** definiert.

Die `equals` Methode

```
1    public boolean equals(Object obj) { ... }
```

Die `equals` Methode testet, ob `this` "gleich" `obj` ist.
Sie muss eine *Äquivalenzrelation* auf Objekten $\neq$ `null` implementieren.
D.h. für alle Objekte x, y und z, die nicht **null** sind, gilt:

- `equals` muss *reflexiv* sein:
  Es gilt immer x.equals(x).
- `equals` muss *symmetrisch* sein:
  Falls x.equals(y), dann auch y.equals(x).
- `equals` muss *transitiv* sein:
  Falls x.equals(y) und y.equals(z), dann auch x.equals(z).

Die `equals` Methode (Fortsetzung)

Weitere Anforderungen an `equals`:

- `equals` muss *konsistent* sein:
  Wenn Objekte x und y nicht `null` sind, dann sollen wiederholte Aufrufe von x.equals(y) immer das gleiche Ergebnis liefern, es sei denn, ein Gleichheits-relevanter Bestandteil von x oder y hat sich geändert.
- Wenn x nicht `null` ist, dann liefert x.equals(**null**) das Ergebnis **false**.

Wichtig

- Jede Implementierung von `equals` muss auf diese Anforderungen hin getestet werden. Grund: *Manche Operationen im Collection Framework verlassen sich darauf!*
- Die Methode equals(Object other) muss überschrieben werden.
  Typischer Fehler:

```
1  class MyType {
2    public boolean equals (MyType other) { ... }
3  }
```

Typische Implementierung von `equals`

```
1  public class Validation {
2    public boolean equals (Object obj) {
3      if (this == obj) { return true; }
4      if (obj == null) { return false; }
5      if (this.getClass() != obj.getClass()) { return false; }
6      Validation other = (Validation)obj;
7      // compare relevant fields...
8    }
9  }
```

Neuheiten:

- getClass()
- Typcast (Validation)other

Der Typcast-Operator

- Der Ausdruck (*Typcast*)

  (*objekttyp*) *ausdruck*

  hat den statischen Typ *objekttyp*, falls der statische Typ von *ausdruck* entweder ein Supertyp oder ein Subtyp von *objekttyp* ist.
- Zur Laufzeit testet der Typcast, ob der **dynamische Typ** des Werts von *ausdruck* ein Subtyp von *objekttyp* ist und bricht das Programm ab, falls das nicht zutrifft. (Vorher sicherstellen!)
- Angenommen **A extends C** und **B extends C** (Klassentypen), aber **A** und **B** stehen in keiner Beziehung zueinander:

```
1  A a = new A(); B b = new B(); C c = new C(); C d = new A();
2
3  (A)a // statisch ok, dynamisch ok
4  (B)a // Typfehler
5  (C)a // statisch ok, dynamisch ok
6  (B)d // statisch ok, dynamischer Fehler
7  (A)d // statisch ok, dynamisch ok
```

Die `getClass`-Methode

```
1    public final Class<?> getClass() { ... }
```

Liefert ein Objekt, das den Laufzeittyp des Empfängerobjekts repräsentiert. Für jeden Typ T definiert das Java-Laufzeitsystem genau ein Objekt vom Typ Class<T>. Die Methoden dieser Klasse erlauben (z.B.) den Zugriff auf die Namen von Feldern und Methoden, das Lesen und Schreiben von Feldern und den Aufruf von Methoden.

Implementierung von `equals` (Fortsetzung)

```
1   // compare relevant fields; beware of null
2   // int f1; // any non-float primitive type
3   if (this.f1 != other.f1) { return false; }
4   // double f2; // float or double types
5   if (Double.compare (this.f2, other.f2) != 0) { return false; }
6   // String f3; // any reference type
7   if ((this.f3 != other.f3) &&
8       ((this.f3 == null) || !this.f3.equals(other.f3))) {
9     return false;
10  }
11  // after all state-relevant fields processed:
12  return true;
```

- Double.compare: Beachte spezielles Verhalten auf NaN und -0.0

Vollständige Implementierung von equals() für **Validation**

```
1  public boolean equals(Object obj) {
2    if (this == obj)
3      return true;
4    if (obj == null)
5      return false;
6    if (getClass() != obj.getClass())
7      return false;
8    Validation other = (Validation) obj;
9    if (timestamp != other.timestamp)
10     return false;
11   if (zone != other.zone)
12     return false;
13   return true;
14 }
```

- Tipp: Automatisch von Eclipse generieren lassen
- Menü "Source→Generate hashCode and equals"

Die `hashCode`-Methode

- Dient der Implementierung von Hash-Verfahren (siehe V Algorithmen und Datenstrukturen)
- Wird vom Collection Framework zur Implementierung von Mengen und Abbildungen verwendet
- Beispiele: Klassen **HashSet** und **HashMap**

Vertrag von hashCode

- Bei mehrfachem Aufruf auf demselben Objekt muss hashCode() immer das gleiche Ergebnis liefern, solange keine Felder geändert werden, die für equals() relevant sind.
- Wenn zwei Objekte equals() sind, dann muss hashCode() auf beiden Objekten den gleichen Wert liefern.
- Die Umkehrung hiervon gilt nicht.

Rezept für eine brauchbare hashCode Implementierung

Vgl. Joshua Bloch. *Effective Java*.

1. Initialisiere `int result = 17`
2. Für jedes Feld *f*, das durch equals() verglichen wird:
   - 2.1 Berechne einen Hash Code c für das Feld *f*, je nach Datentyp
     - `boolean`: `(f ? 1 : 0)`
     - `byte`, `char`, `short`: `(int)f`
     - `long`: (f ^ (f >>> 32))
     - `float`: `Float.floatToIntBits(f)`
     - `double`: konvertiere nach `long` ...
     - `f` ist Objektreferenz und wird mit equals vergleichen: `f.hashCode()` oder 0, falls `f == null`
     - `f` ist Array: verwende `java.util.Arrays.hashCode(f)`
   - 2.2 `result = 31 * result + c`
3. `return result`

Vollständige Implementierung von hashCode für **Validation**

```
1  public int hashCode() {
2    final int prime = 31;
3    int result = 1;
4    result = prime * result + (int) (timestamp ^ (timestamp >>> 32));
5    result = prime * result + ((zone == null) ? 0 : zone.hashCode());
6    return result;
7  }
```

- Generiert von Eclipse
- Vorige Folie ist ein sogenanntes “Metaprogramm”, d.h. ein Algoritmus um ein Programm zu schreiben
- Implementiert von Eclipse

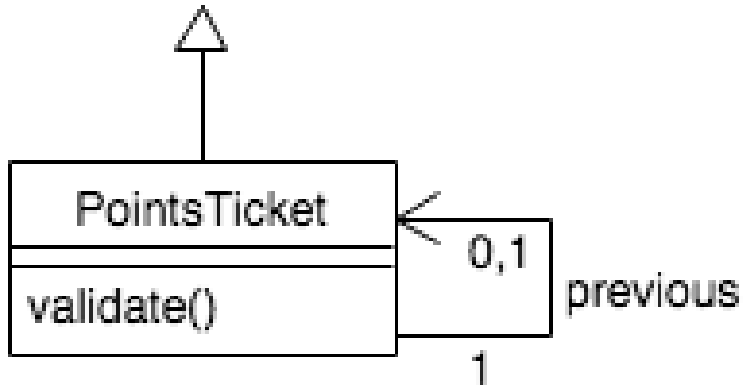
Rekursive Assoziation

Neue Anforderung

Spezifikation

. . . Mehrere Punktekarten können zusammengefasst werden um die notwendige Zahl von Stempeln zu erreichen. . . .

Klassendiagramm für Tickets (Auszug)



- PointsTicket mit *rekursiver Assoziation* previous auf sich selbst
- Intention: Falls mehrere Tickets zusammengefasst werden sollen, hängen sie über die Assoziation previous zusammen.
- previous ist optional
- Implementierung durch Feld previous, das im Konstruktor gesetzt wird

Javacode für **PointsTicket** mit previous

```
1  public class PointsTicket extends ATicket {
2    private final PointsTicket previous;
3    private final static int MAX_STAMPS = 20;
4
5    public PointsTicket(PointsTicket previous) {
6      super(MAX_STAMPS);
7      this.previous = previous;
8    }
9    public PointsTicket() {
10     super(MAX_STAMPS);
11     this.previous = null;
12   }
13   // countValidations ...
14 }
```

Zwei Konstruktoren

- PointsTicket() für neues Ticket ohne Vorgänger
- PointsTicket(previous) für Anschlussticket

Javacode für **PointsTicket**: countValidations

```
1  private int countValidations(Validation validation) {
2    int count = 0;
3    if (this.stamps.size() <= MAX_STAMPS) {
4      for (Validation stamp : this.stamps) {
5        if (validation.equals(stamp)) { count++; }
6      }
7    }
8    if (previous != null) {
9      count += previous.countValidations(validation);
10   }
11   return count;
12 }
```

- Typisches Muster: rekursiver Aufruf auf previous (falls ungleich **null**)

Mini-Exkurs: Überladung

- **PointsTicket** hat mehrere Konstruktoren
- Sie unterscheiden sich in der Anzahl der Argumente

$\Rightarrow$ Der Konstruktor ist *überladen*

- Mehrere Konstruktoren dürfen definiert werden, solange sie unterschiedliche Signaturen haben, d.h. sie müssen sich in der Anzahl oder in den Typen der Argumente unterscheiden.
- Bei einem Aufruf wird *statisch* (d.h., vom Java Compiler) anhand der Anzahl und der statischen Typen der Argumente entschieden, welcher Konstruktor gemeint ist.
- *Genauso können Methoden und statische Methoden überladen werden.*

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Beispiele

- Angenommen **A extends B** (Klassentypen).

```
1   A a = new A (); // rhs: Typ A, dynamischer Typ A
2   B b = new B (); // rhs: Typ B, dynamischer Typ B
3   B x = new A (); // rhs: Typ A, dynamischer Typ A
4   // für x gilt: Typ B, dynamischer Typ A
```

- Bei einem Interfacetyp ist der dynamische Typ **immer** ein Subtyp.
- Im Rumpf einer Methode definiert in der Klasse **C** hat `this` den statischen Typ **C**. Der dynamische Typ kann ein Subtyp von **C** sein, falls die Methode vererbt worden ist.

Regeln für die Bestimmung des statischen Typs

- Falls Variable (Feld, Parameter) `x` durch **ttt x** deklariert ist, so ist der Typ von `x` genau **ttt**.
- Der Ausdruck `new C(...)` hat den Typ **C**.
- Wenn *e* ein Ausdruck vom Typ **C** ist und **C** eine Klasse mit Feld *f* vom Typ ttt ist, dann hat *e.f* den Typ ttt.
- Wenn *e* ein Ausdruck vom Typ **C** ist und **C** eine Klasse oder Interface mit Methode *m* vom Rückgabetyp ttt ist, dann hat *e.m*(...) den Typ ttt.
- Beim Aufruf eines Konstruktors oder einer Funktion müssen die Typen der Argumente jeweils Subtypen der Parametertypen sein.
- Bei einer Zuweisung muss der Typ des Audrucks auf der rechten Seiten ein Subtyp des Typs der Variable (Feld) sein.