6 Abstrakte Klassen und Schnittstellen

6.1 Abstrakte Klassen und Methoden

6.1.1 Grundsätzliches

Abstrakte Klassen

- Eine Klasse kann mit dem Schlüsselwort `abstract` deklariert werden, um anzuzeigen, dass sie (technisch oder logisch) unvollständig ist.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- Eine abstrakte Klasse kann trotzdem Konstruktoren besitzen, die (nur) von Konstruktoren ihrer Unterklassen aufgerufen werden können (und daher sinnvollerweise `protected` sind).
- Aufgrund der üblichen Untertyp-Polymorphie, können Variablen einer abstrakten Klasse Objekte von (konkreten) Unterklassen referenzieren.

Abstrakte Methoden

- Eine Methode kann mit dem Schlüsselwort `abstract` deklariert werden, um lediglich ihre Signatur und ihren Resultattyp zu vereinbaren, aber noch keine Implementierung anzugeben. (Private, unveränderliche und statische Methoden können nicht abstrakt sein.)

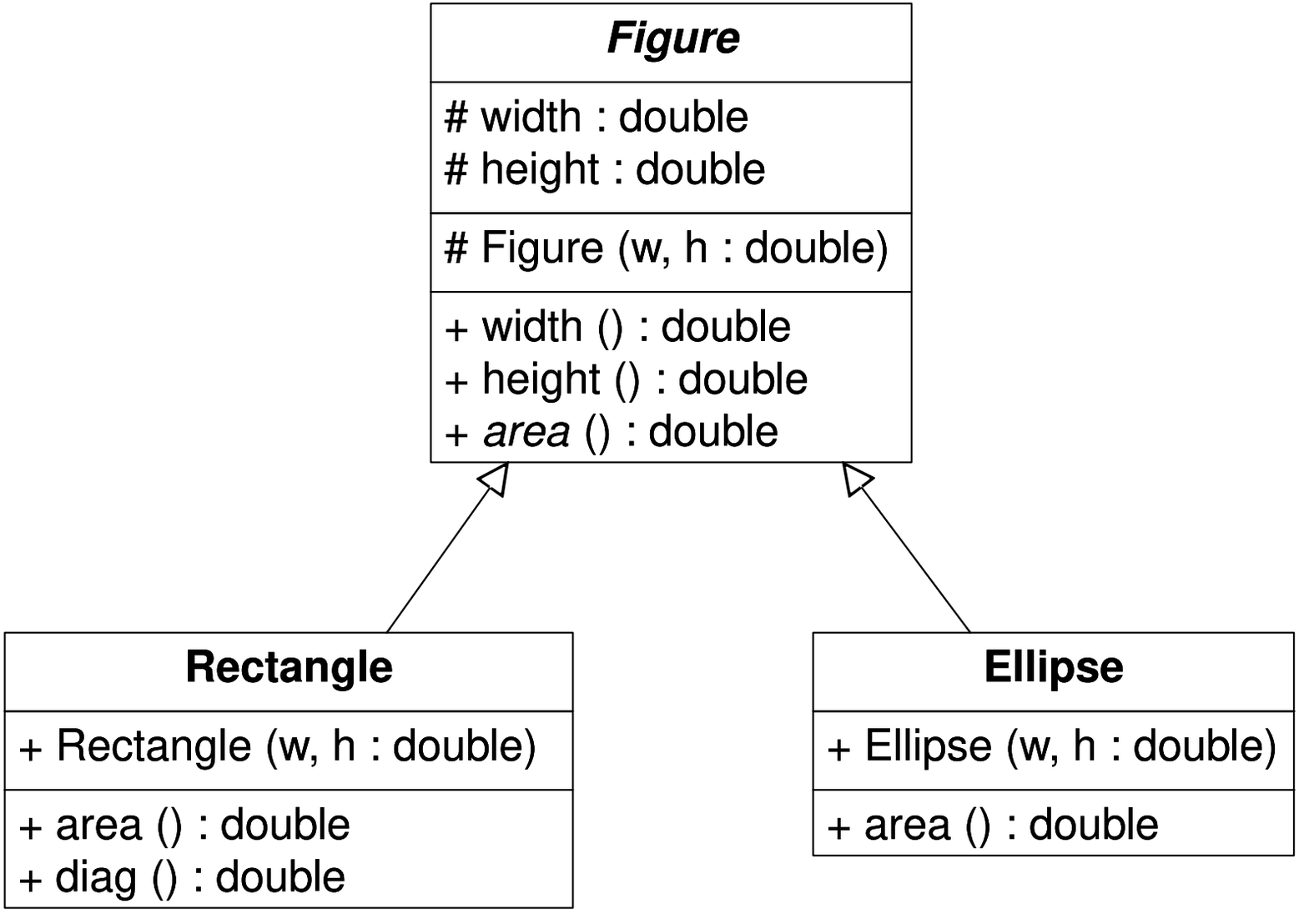
- Abstrakte Methoden können in Unterklassen durch konkrete Methoden mit derselben Signatur überschrieben (d. h. implementiert) werden.

- Eine Klasse, die abstrakte Methoden deklariert oder erbt, aber nicht implementiert, muss abstrakt sein.

6.1.2 Beispiel

```
// Abstrakte Klasse: Allgemeines geometrisches Objekt.
abstract class Figure {
  // Unterklassenöffentliche Objektvariablen: Breite und Höhe.
  protected double width, height;

  // Unterklassenöffentlicher Konstruktor:
  // Breite und Höhe initialisieren.
  protected Figure (double w, double h) {
    width = w;
    height = h;
  }

  // Öffentliche Objektmethoden: Breite und Höhe abfragen.
  public double width () { return width; }
  public double height () { return height; }

  // Öffentliche abstrakte Methode: Fläche berechnen.
  public abstract double area ();
}
```

```
// Konkrete Unterklasse von Figure: Rechteck.
class Rectangle extends Figure {
  // Öffentlicher Konstruktor.
  public Rectangle (double w, double h) { super(w, h); }

  // Implementierung der geerbten abstrakten Methode area.
  public double area () { return width * height; }

  // Zusätzliche Objektmethode: Diagonale berechnen.
  public double diag () {
    return Math.sqrt(width*width + height*height);
  }
}

// Konkrete Unterklasse von Figure: Ellipse.
class Ellipse extends Figure {
  // Öffentlicher Konstruktor.
  public Ellipse (double w, double h) { super(w, h); }

  // Implementierung der geerbten abstrakten Methode area.
  public double area () { return Math.PI/4 * width * height; }
}
```



Die Namen von abstrakten Klassen und Methoden werden in UML kursiv geschrieben.

```
// Testklasse (kann prinzipiell abstrakt sein).
abstract class Test {
  // Aufruf zum Beispiel: java Test r 2.5 3.2 e 1.7 5 ...
  public static void main (String [] args) {
    // Unterschiedliche geometrische Objekte erzeugen.
    Figure [] figs = new Figure [args.length/3];
    for (int i = 0; i < args.length; i += 3) {
      char x = args[i].charAt(0);
      double w = Double.parseDouble(args[i+1]);
      double h = Double.parseDouble(args[i+2]);
      if (x == 'r') figs[i/3] = new Rectangle(w, h);
      else figs[i/3] = new Ellipse(w, h);
    }
    // Fläche und ggf. Diagonale aller Objekte ausgeben.
    for (Figure fig : figs) {
      System.out.print(fig.area());
      if (fig instanceof Rectangle r) {
        System.out.print(" " + r.diag());
      }
      System.out.println();
    }
  }
}
```

6.2 Schnittstellen

6.2.1 Aufbau einer Schnittstelle

- Eine *Schnittstelle* (*interface*) ist ähnlich zu einer abstrakten Klasse, die jedoch nur folgende Elemente enthalten darf:
  - Objektmethoden ohne Implementierung, die implizit `public abstract` sind (d. h. diese Schlüsselwörter können, müssen aber nicht angegeben werden)
  - Konstanten, d. h. Felder, die implizit `public static final` sind und einen Initialisierungsausdruck besitzen müssen, der einmalig beim Laden/Initialisieren der Schnittstelle (d. h. normalerweise zu Beginn der Programmausführung) ausgewertet wird
  - Geschachtelte Klassen und Schnittstellen

  Seit Java 8 zusätzlich:
  - Vorimplementierte Objektmethoden (default methods) mit Schlüsselwort `default` und Implementierung, die implizit `public` sind
  - Klassenmethoden (mit Schlüsselwort `static` und Implementierung), die implizit `public` sind

  Seit Java 9 zusätzlich:
  - Private Objekt- und Klassenmethoden (mit Schlüsselwort `private` und ggf. `static` sowie Implementierung)

- Dementsprechend darf eine Schnittstelle folgende Elemente nicht enthalten:
  - Felder, die nicht `public static final` sind, d. h. insbesondere Objektvariablen
  - Konstruktoren
    (es gibt auch keinen implizit deklarierten parameterlosen Konstruktor)
  - Initialisierungsblöcke
  - paket- oder unterklassenöffentliche Elemente
- Häufig enthält eine Schnittstelle nur öffentliche abstrakte Methoden.
- Eine Schnittstelle kann auch leer sein, d. h. insbesondere keine Methoden deklarieren.
  Wenn eine Klasse eine solche Schnittstelle implementiert (vgl. § 6.2.2), zeigt sie damit lediglich an, dass sie logisch eine bestimmte Eigenschaft besitzt (vgl. § 8.8).

6.2.2 Vererbung und Untertypbeziehungen zwischen Schnittstellen und Klassen

- Eine Schnittstelle kann eine oder mehrere andere Schnittstellen erweitern (Schlüsselwort `extends`), um von ihnen zu erben, d. h. Schnittstellen erlauben Mehrfachvererbung.

- Eine (abstrakte oder konkrete) Klasse kann eine oder mehrere Schnittstellen *implementieren* (Schlüsselwort `implements`), um anzuzeigen, dass sie die durch die Schnittstellen repräsentierten Eigenschaften besitzt. Wenn die Klasse konkret ist, muss sie Implementierungen für alle abstrakten Methoden enthalten oder erben.

- `implements` stellt, genauso wie `extends`, eine Vererbungsbeziehung dar und definiert deshalb genauso eine Ober-/Untertypbeziehung.
  Außerdem gilt jede Schnittstelle als Untertyp der Wurzelklasse `Object`.

- Dementsprechend kann eine Variable, deren Typ eine Schnittstelle ist, Objekte aller Klassen referenzieren, die die Schnittstelle (direkt oder indirekt) implementieren.

- Typtests und Typumwandlungen (vgl. § 5.8) funktionieren auch für Schnittstellen. Anders als in § 5.8, wo nur Klassen betrachtet wurden, sind mit Schnittstellen auch *Querumwandlungen* (cross-casts) möglich, d. h. Umwandlungen, bei denen Ursprungs- und Zieltyp in keiner Ober- oder Untertypbeziehung zueinander stehen.

- Statische Methoden von Schnittstellen werden nicht vererbt.

6.2.3 Überschreiben von Objektmethoden

- Wenn eine Klasse oder Schnittstelle eine Objektmethode (mit oder ohne Implementierung) definiert, überschreibt diese Methode alle geerbten Methoden, die dieselbe Signatur (d. h. denselben Namen und dieselben Parametertypen) besitzen.

  Eine abstrakte Methode kann also auch eine Methode mit Implementierung überschreiben. Damit kann eine abstrakte Klasse ihre Unterklassen „zwingen“, Methoden wie z. B. `toString` oder `equals` zu überschreiben, wenn die von `Object` geerbten Implementierungen nicht sinnvoll sind.

- Der Resultattyp der überschreibenden Methode muss ein trivialer, direkter oder indirekter Untertyp der Resultattypen aller überschriebenen Methoden sein (vgl. § 5.5).

- Eine überschriebene Methode `objmeth` kann, sofern sie nicht abstrakt ist, entweder wie gewohnt (vgl. § 5.5) mit `super.objmeth(arguments)` (wenn sie von der Oberklasse geerbt wird) oder mit `superinterface.super.objmeth(arguments)` (wenn sie von der Ober-Schnittstelle `superinterface` geerbt wird) aufgerufen werden. (`superinterface` muss eine direkte Ober-Schnittstelle sein, die nicht gleichzeitig eine indirekte Ober-Schnittstelle ist.)

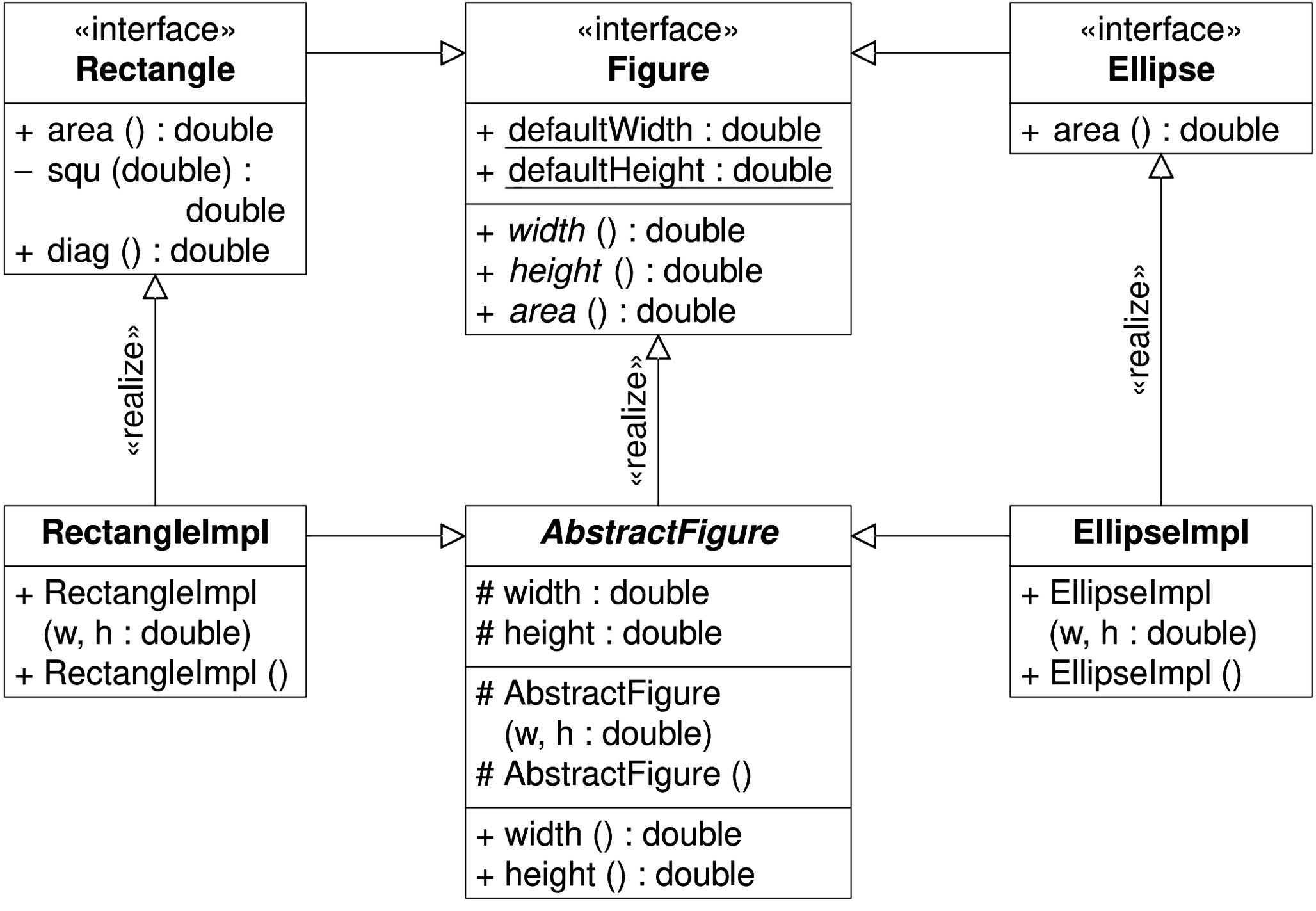
6.2.4 Beispiel (vgl. § 6.1.2)

```
// Schnittstelle für beliebige geometrische Objekte.
interface Figure {
  // Objektmethoden, die implizit public abstract sind.
  double width ();                  // Breite.
  double height ();                 // Höhe.
  double area ();                   // Fläche.

  // Felder, die implizit public static final sind.
  double defaultWidth = 4;          // Standardbreite.
  double defaultHeight = 3;         // Standardhöhe.
}
```

```
// Schnittstelle für Rechtecke.
interface Rectangle extends Figure {
  // Vorimplementierung der geerbten Methode area.
  default double area () { return width() * height(); }

  // Private Hilfsmethode.
  private double squ (double x) { return x * x; }

  // Zusätzliche vorimplementierte Objektmethode,
  // die implizit public ist.
  default double diag () { // Diagonale.
    return Math.sqrt(squ(width()) + squ(height()));
  }
}

// Schnittstelle für Ellipsen.
interface Ellipse extends Figure {
  // Vorimplementierung der geerbten Methode area.
  default double area () { return Math.PI/4 * width() * height(); }

  // Keine zusätzlichen Methoden.
}
```

```
// Abstrakte Klasse zur teilweisen Implementierung
// der Schnittstelle Figure.
abstract class AbstractFigure implements Figure {
  // Unterklassenöffentliche Objektvariablen: Breite und Höhe.
  protected double width, height;

  // Unterklassenöffentliche Konstruktoren:
  // Breite und Höhe initialisieren.
  protected AbstractFigure (double w, double h) {
    width = w;
    height = h;
  }
  protected AbstractFigure () {
    this(defaultWidth, defaultHeight);
  }

  // Implementierung der Schnittstellen-Methoden width und height.
  public double width () { return width; }
  public double height () { return height; }

  // Die Schnittstellen-Methode area wird nicht implementiert.
}
```

```
// Konkrete Klasse zur Implementierung von Rechtecken.
class RectangleImpl extends AbstractFigure implements Rectangle {
  // Öffentliche Konstruktoren.
  public RectangleImpl (double w, double h) { super(w, h); }
  public RectangleImpl () { super(); }

  // Alle abstrakten Methoden sind bereits in AbstractFigure
  // oder Rectangle implementiert.
}

// Konkrete Klasse zur Implementierung von Ellipsen.
class EllipseImpl extends AbstractFigure implements Ellipse {
  // Öffentliche Konstruktoren.
  public EllipseImpl (double w, double h) { super(w, h); }
  public EllipseImpl () { super(); }

  // Alle abstrakten Methoden sind bereits in AbstractFigure
  // oder Ellipse implementiert.
}
```


«interface»
Rectangle
+ area () : double
– squ (double) : double
+ diag () : double
«interface»
Figure
+ defaultWidth : double
+ defaultHeight : double
+ width () : double
+ height () : double
+ area () : double
«interface»
Ellipse
+ area () : double
«realize»
«realize»
«realize»
RectangleImpl
+ RectangleImpl (w, h : double)
+ RectangleImpl ()
AbstractFigure
width : double
height : double
AbstractFigure (w, h : double)
AbstractFigure ()
+ width () : double
+ height () : double
EllipseImpl
+ EllipseImpl (w, h : double)
+ EllipseImpl ()

Erläuterungen zum UML-Diagramm:

- Die Namen von abstrakten Klassen und Methoden werden kursiv geschrieben.
- Schnittstellen werden mit «interface» gekennzeichnet, aber ihre Namen werden nicht kursiv geschrieben, obwohl sie ebenfalls abstrakt sind.
- `extends`-Beziehungen zwischen zwei Klassen oder zwischen zwei Schnittstellen werden durch Pfeile ohne Beschriftung dargestellt.
- `implements`-Beziehungen zwischen einer Klasse und einer Schnittstelle werden durch Pfeile mit Beschriftung «realize» dargestellt.
- Die Namen von statischen Variablen und Methoden werden unterstrichen.

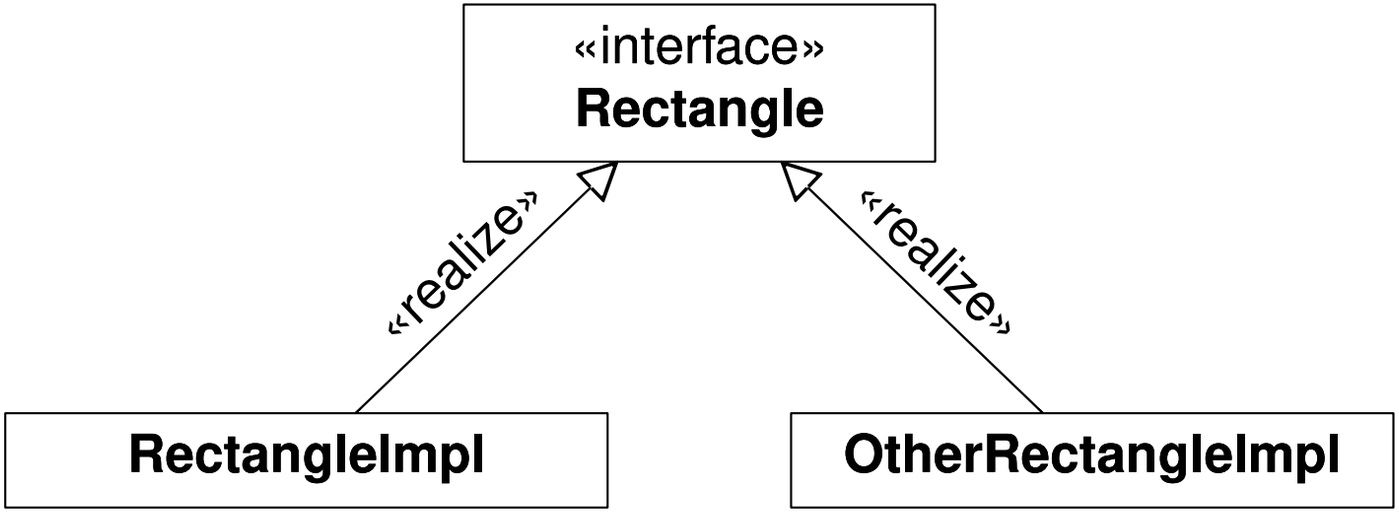
```
// Hauptprogramm (kann auch in einer Schnittstelle definiert sein).
interface Test {
  // Aufruf zum Beispiel: java Test r 2.5 3.2 e 1.7 5 ...
  public static void main (String [] args) {
    // Unterschiedliche geometrische Objekte erzeugen.
    Figure [] figs = new Figure [args.length/3];
    for (int i = 0; i < args.length; i += 3) {
      char x = args[i].charAt(0);
      double w = Double.parseDouble(args[i+1]);
      double h = Double.parseDouble(args[i+2]);
      if (x == 'r') figs[i/3] = new RectangleImpl(w, h);
      else figs[i/3] = new EllipseImpl(w, h);
    }
    // Fläche und ggf. Diagonale aller Objekte ausgeben.
    for (Figure fig : figs) {
      System.out.print(fig.area());
      if (fig instanceof Rectangle r) {
        System.out.print(" " + r.diag());
      }
      System.out.println();
    }
  }
}
```

Anmerkungen

- Obwohl die Implementierung mit Schnittstellen umfangreicher und „komplizierter“ ist als die Implementierung in § 6.1.2, besitzt sie mindestens zwei wichtige Vorteile:

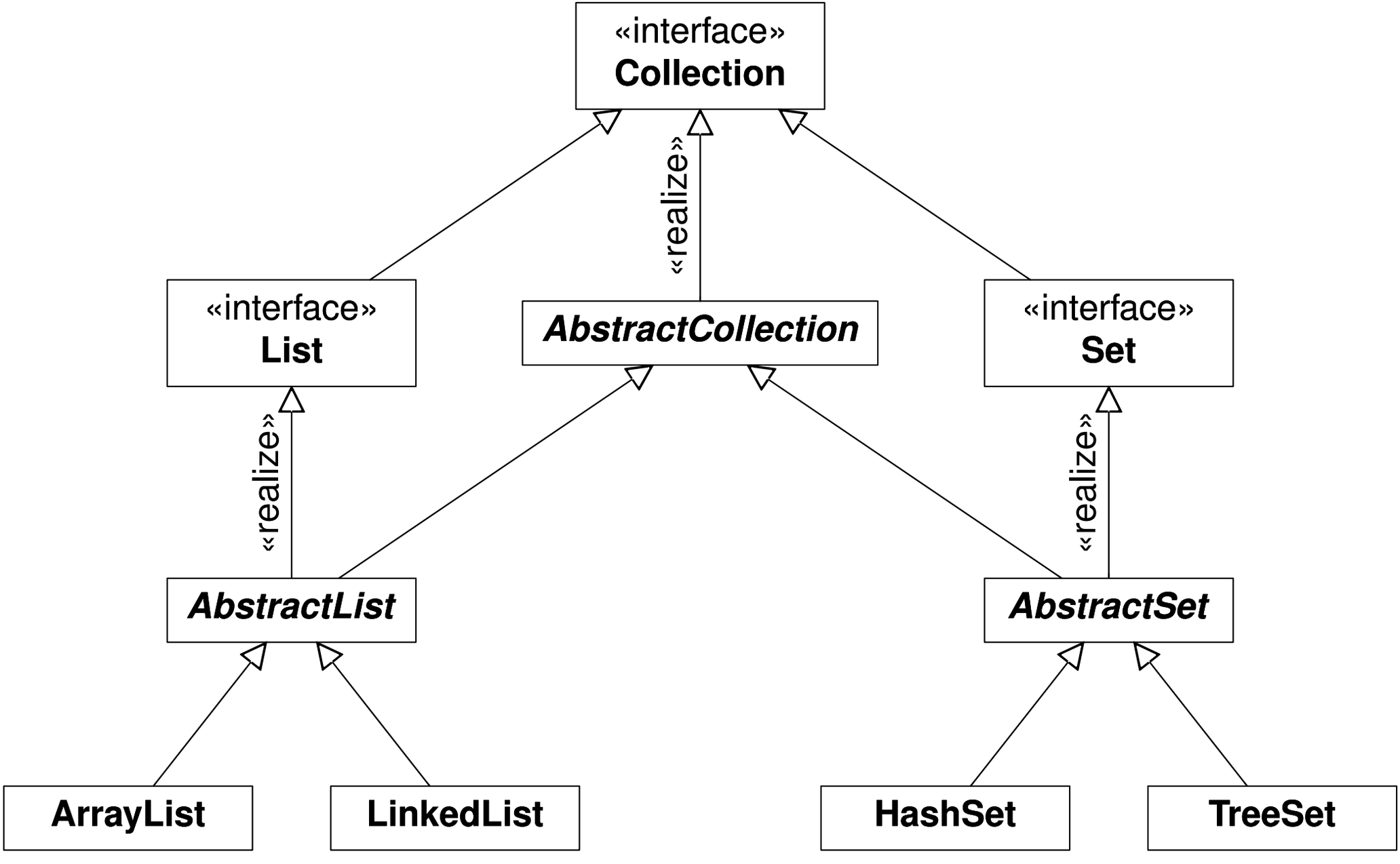
- Schnittstellen (interfaces) und zugehörige Implementierungen (Klassen) sind vollständig voneinander getrennt, was zu einer flexibleren und leichter erweiterbaren Systemarchitektur führt.

- Zu einer Schnittstelle kann es mehrere unterschiedliche Implementierungsklassen geben, wobei Code, der nur die Schnittstellenmethoden verwendet, mit jeder dieser Klassen unverändert funktioniert. Zum Beispiel:



```
// Andere Klasse zur Implementierung von Rechtecken.
class OtherRectangleImpl extends AbstractFigure
                                          implements Rectangle {
  // Zusätzliche Objektvariablen
  // zur Speicherung von Fläche und Diagonale.
  private double area, diag;

  // Konstruktor berechnet sofort Fläche und Diagonale.
  public OtherRectangleImpl (double w, double h) {
    super(w, h);
    area = Rectangle.super.area();
    diag = Rectangle.super.diag();
  }
  public OtherRectangleImpl () {
    this(defaultWidth, defaultHeight);
  }

  // Objektmethoden liefern nur die zuvor berechneten Werte.
  public double area () { return area; }
  public double diag () { return diag; }
}
```

6.2.5 Beispiel aus der Java-Standardbibliothek



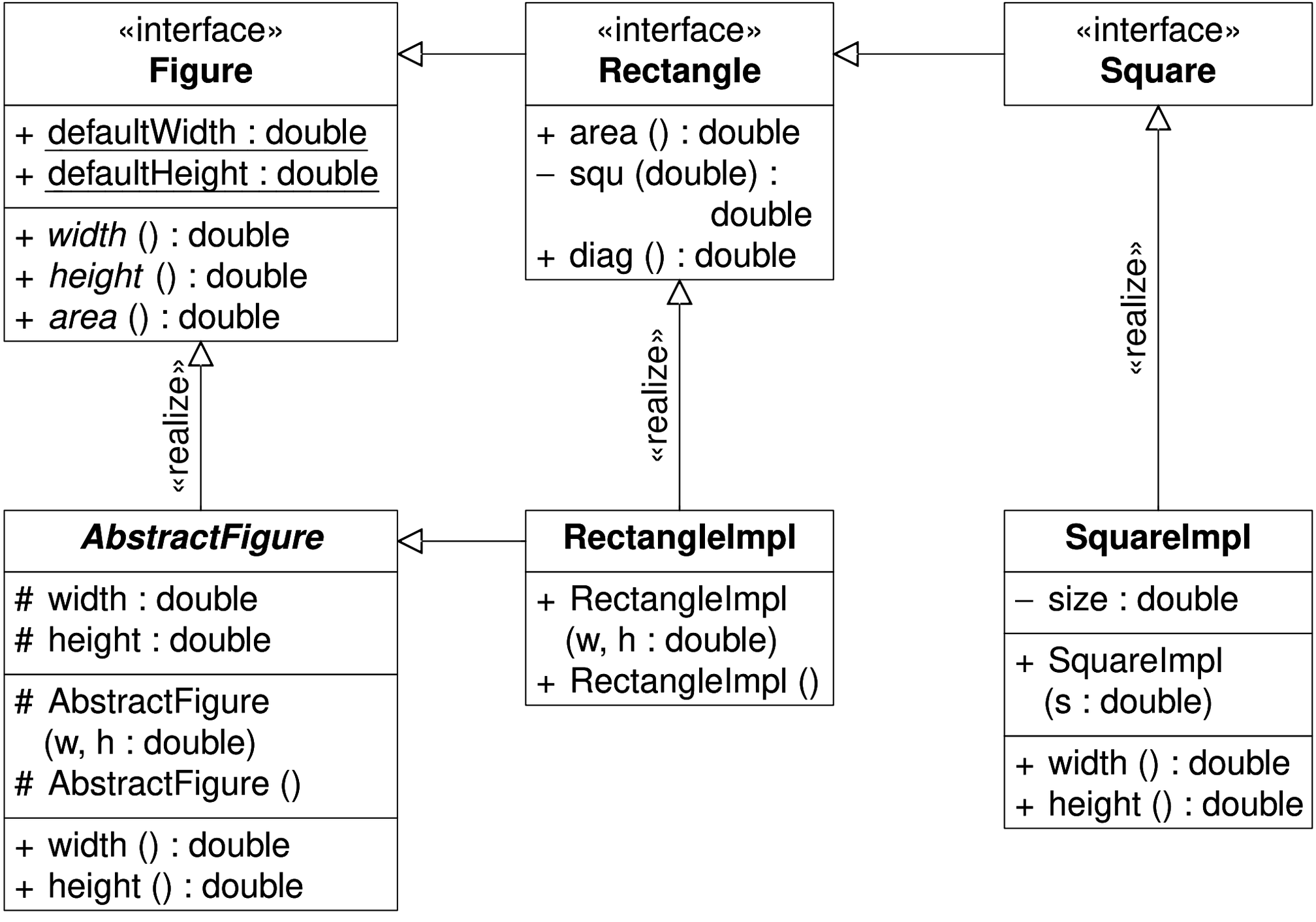
6.2.6 Typ- und Implementierungshierarchie

- Wenn man das Beispiel aus § 6.2.4 noch um Quadrate erweitert, sollte `Square` ein Untertyp von `Rectangle` sein, weil ein Quadrat ein Spezialfall eines Rechtecks ist, das die gleichen Eigenschaften (Methoden `width`, `height`, `area` und `diag`) besitzt und daher überall verwendet werden kann, wo ein Rechteck erwartet wird.

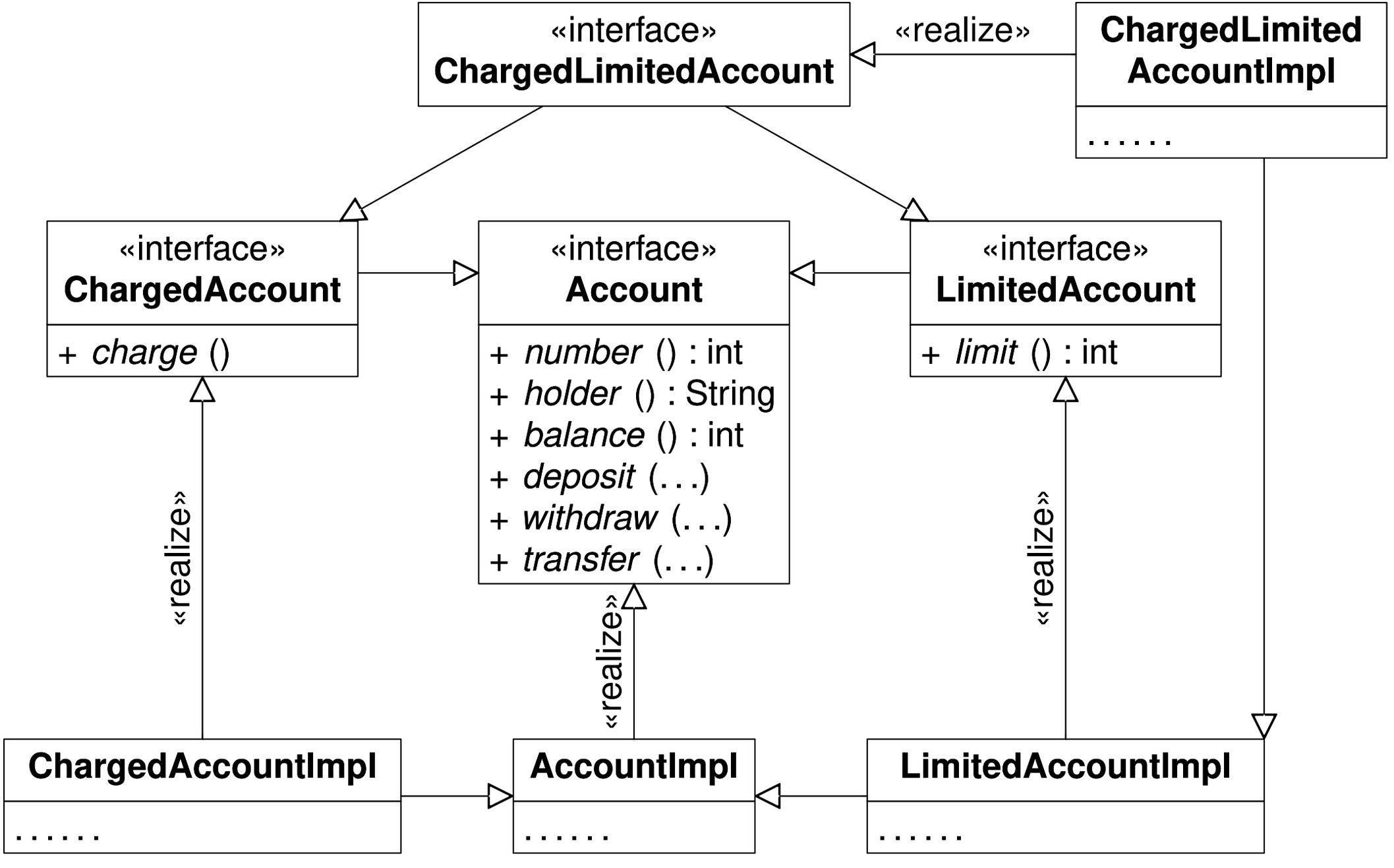
- Wenn man nur mit Klassen arbeitet (§ 6.1.2), muss `Square` dann zwangsläufig auch die Objektvariablen `width` und `height` von `Rectangle` erben, obwohl zur Implementierung eines Quadrats eine einzige Objektvariable `size` ausreichen würde.

- Wenn man Schnittstellen verwendet (§ 6.2.4), können die `extends`-Beziehungen zwischen ihnen unabhängig von den Beziehungen zwischen den zugehörigen Implementierungsklassen sein, d. h. Typ- und Implementierungshierarchie können unterschiedlich strukturiert sein.

- Insbesondere muss `SquareImpl` nicht von `RectangleImpl` erben.


«interface»
Figure
+ defaultWidth : double
+ defaultHeight : double
+ width () : double
+ height () : double
+ area () : double
«interface»
Rectangle
+ area () : double
– squ (double) : double
+ diag () : double
«interface»
Square
«realize»
«realize»
«realize»
AbstractFigure
width : double
height : double
AbstractFigure (w, h : double)
AbstractFigure ()
+ width () : double
+ height () : double
RectangleImpl
+ RectangleImpl (w, h : double)
+ RectangleImpl ()
SquareImpl
– size : double
+ SquareImpl (s : double)
+ width () : double
+ height () : double

6.2.7 Mehrfachvererbung mit Schnittstellen



Erläuterungen

- Auf Schnittstellen- bzw. Typebene lässt sich wie gewünscht ausdrücken, dass ein `ChargedLimitedAccount` *sowohl* die Eigenschaften von `ChargedAccount` (zusätzliche Methode `charge` gegenüber `Account`) *als auch* die Eigenschaften von `LimitedAccount` (zusätzliche Methode `limit`) besitzt.

- Auf Klassen- bzw. Implementierungsebene kann `ChargedLimitedAccountImpl` aber nur *entweder* von `ChargedAccountImpl` *oder* von `LimitedAccountImpl` erben.

- Daher wählt man als Oberklasse sinnvollerweise diejenige, von der es „mehr zu erben“ gibt, im Beispiel `LimitedAccountImpl`, weil die Implementierung des Kreditlimits aufwendiger ist als die Implementierung des Gebührenzählers.
  Die Funktionalität der anderen Klasse muss dann notgedrungen repliziert werden.

- Eventuell kann die störende Code-Verdopplung reduziert werden, indem ein Teil der Funktionalität einer Klasse in die zugehörige Schnittstelle verlagert wird, z. B. in Form einer Methode mit Vorimplementierung.

  Konkret könnte z. B. die Methode `check` von der Klasse `LimitedAccountImpl` in die Schnittstelle `LimitedAccount` verlagert werden, wo sie dann auch von der Klasse `ChargedLimitedAccountImpl` verwendet werden kann, die dann alternativ `ChargedAccountImpl` als Oberklasse besitzen könnte.

6.2.8 Namenskonflikte

Objektmethoden

- Wenn eine Klasse oder eine Schnittstelle mehrere Objektmethoden mit derselben Signatur (von ihrer Oberklasse und/oder einer oder mehreren Ober-Schnittstellen) erbt und selbst keine Methode mit derselben Signatur definiert, werden zunächst diejenigen Methoden entfernt, die bereits von einer anderen geerbten Methode überschrieben werden.

  Zum Beispiel:

```
interface A {
  void m (); // m könnte auch eine Vorimplementierung besitzen.
}
interface B extends A {      // B könnte auch eine Klasse sein,
  default void m () { ..... }          // die A implementiert.
}
interface C extends A, B {}  // C könnte auch eine Klasse sein,
     // die A implementiert und B implementiert bzw. erweitert.
```

  Da die Methode `m` aus `A` in `B` überschrieben wird, erbt `C` letztlich nur die Überschreibung aus `B` und nicht zusätzlich die Methode aus `A`, sodass kein Konflikt entsteht.

- Wenn es dann immer noch mehr als eine Methode mit derselben Signatur gibt:
  - Wenn alle Methoden abstrakt sind, werden sie quasi zu einer einzigen abstrakten Methode zusammengefasst.
    Wenn die Methoden unterschiedliche Resultattypen besitzen, muss einer von ihnen ein trivialer, direkter oder indirekter Untertyp aller anderen sein, und die verschmolzene Methode besitzt diesen Typ als Resultattyp (vgl. § 6.2.3).
  - Wenn eine der Methoden konkret ist, d. h. eine „vollwertige“ Implementierung in einer Klasse besitzt, überschreibt sie alle anderen Methoden, d. h. in diesem Fall besitzt die Klasse genau eine konkrete Methode mit der entsprechenden Signatur. (Weil eine Klasse nur eine direkte Oberklasse besitzen kann, kann es höchstens eine solche konkrete Methode geben. Alle anderen sind dann entweder abstrakt oder besitzen nur eine Vorimplementierung.)
    Wenn die Methoden unterschiedliche Resultattypen besitzen, muss der Resultattyp der konkreten Methode ein trivialer, direkter oder indirekter Untertyp aller anderen Resultattypen sein (vgl. § 6.2.3).
  - Andernfalls, d. h. wenn eine oder mehrere Methoden eine Vorimplementierung, aber keine eine vollwertige Implementierung besitzen, ist die Klasse oder Schnittstelle fehlerhaft.
    Um den Konflikt aufzulösen, muss sie selbst eine Methode mit der entsprechenden Signatur definieren, die dann alle geerbten Methoden überschreibt.

Felder

- Wenn eine Klasse oder eine Schnittstelle mehrere gleichnamige Felder (Objekt- oder Klassenvariablen) erbt, ist der Zugriff auf diese Felder mit dem einfachen Feldnamen `field` *mehrdeutig*.

- Eine solche Mehrdeutigkeit kann mit einer der folgenden Möglichkeiten aufgelöst werden:
  - `super.field`
    (Zugriff auf ein Feld der Oberklasse des aktuellen Objekts)
  - `((supertype)object).field`
    (Zugriff auf ein Feld der Oberklasse oder einer Ober-Schnittstelle `supertype` des Objekts `object`)
  - `type.field`
    (Zugriff auf ein statisches Feld der Klasse oder Schnittstelle `type`)