Schnittstellenbildung und Klassifizierung

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Abstraktion durch Schnittstellenbildung ist neben der Kapselung, Vererbung und Polymorphie ein Grundprinzip der objekt-orientierten Programmierung. In diesem Kapitel werden wir sehen, wie Methodenschnittstellen in Java in Form von Interfaces modelliert und implementiert werden können. Die Verwendung von Interfaces erlaubt die Abbildung von hierarchischen Klassifizierungen auf Typen, so dass Klassen mit gleicher (Teil-)Funktionalität zu Typen zusammengefasst werden können.
So haben beispielsweise alle Listenimplementierungen aus dem letzten Kapitel die gleiche Funktionalität, nämlich das Einfügen von Elementen, Ermitteln von Elementen an bestimmten Positionen, das Berechnen der Länge der Liste etc. Diese gemeinsame Methodenschnittstelle lässt sich in einem Schnittstellen-Typ für Listen zusammenfassen, der durch die einfach-/doppelt-verkettete bzw. Array-Liste implementiert wird. Wir werden in diesem Kapitel die Deklaration und Verwendung von Schnittstellentypen anhand von Beispielen besprechen. Dabei werden wir auch auf einige Eigenheiten der Typhierarchie in Java eingehen.

Lernziele dieses Kapitels:

- Schnittstellen in Java in Form von Interfaces zu modellieren und zu beschreiben.
- Klassen zu definieren, die Interfaces implementieren.
- Klassifizierungen durch Referenztypen zu modellieren.
- Sub-/Supertypbeziehungen auf elementaren Datentypen zu nennen.
- Funktionsweise und Vorteile der dynamischen Methodenauswahl anhand von Beispielen zu erläutern.

1 Objektorientierte Modellierung

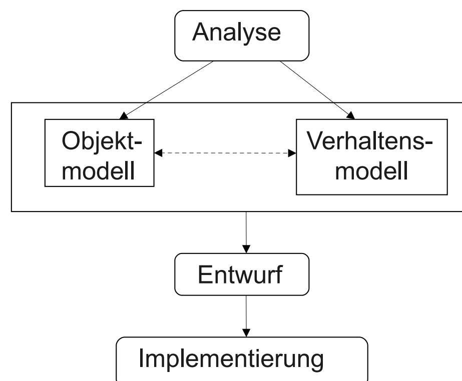
Die Anforderungsanalyse führt zu einem ***Modell*** des Ausschnitts der Welt, der zur Lösung geeignet ist. Das Modell beschreibt die wichtigen Eigenschaften des zu entwickelnden Systems. Es orientiert sich zunächst an der Anwendung und nicht der Realisierung.

Wir geben hier nur eine kurze Einführung; die objektorientierte Modellierung ist zentraler Gegenstand der Vorlesung SE 2.

Aspekte der Modellierung Bei objektorientierter Modellierung müssen wir Antworten auf folgende Fragen finden:

1. Welche Objekte werden benötigt?
2. Welche Beziehungen gibt es zwischen den Objekten?
3. Welche Eigenschaften besitzen die Objekte?
4. Wie lassen sich die Objekte klassifizieren?
5. Wie werden die Objekte angewendet?
6. Was ist das Verhalten der Objekte? Wie reagieren sie auf Nachrichten?

Das ***Objekt-/Klassenmodell*** liefert die Antworten zu 1-4. ***Anwendungsfälle*** beantworten Frage 5. Das ***Verhaltensmodell*** klärt Frage 6.
Im Rahmen der objekt-orientierten Softwareentwicklung ist die objektorientierte Analyse zwischen der Problemanalyse und dem Entwurf anzusiedeln.



Sie liefert zwei Modelle:

- Das ***Objektmodell*** beschreibt
  - die relevanten Klassen von Objekten,
  - deren Attribute,
  - welche Dienste/Nachrichten/Operationen bereitgestellt werden,
  - die Beziehungen zwischen den Objekten.
- Das ***Verhaltensmodell*** beschreibt
  - die Wirkungsweise der Methoden,
  - die möglichen Zustände von Objekten,
  - das Ablaufverhalten und die Interaktion zwischen den Objekten.

1.1 Beispiel

Wir betrachten hier exemplarisch die objektorientierte Analyse zur Aufgabenstellung ein Informationssystem für eine Universität zu entwickeln.
Die Analyse liefert als groben Leistungsumfang folgende Punkte (stark vereinfacht):

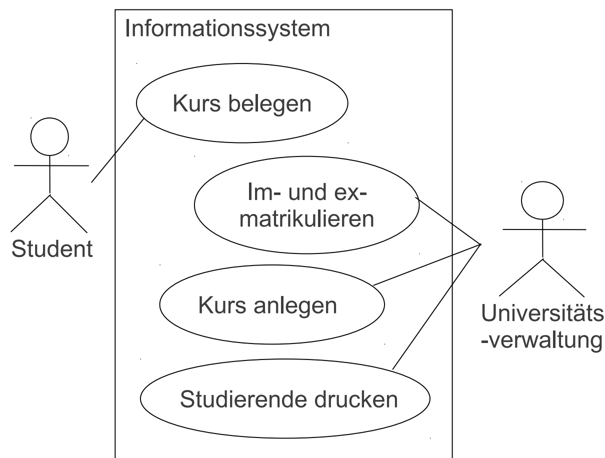
- Universitäten bestehen aus Studierenden und Kursen mit folgenden Beziehungen:
  - Jede(r) Studierende hat einen Namen, eine eindeutige Matrikelnummer und belegt eine Reihe von Kursen.
  - Jeder Kurs hat einen Titel und eine maximale Teilnehmerzahl.
- Typische ***Anwendungsfälle*** (engl. *use cases*) sind:
  - Die Universität kann Studierende im- und exmatrikulieren.
  - Studierende können angebotene Kurse belegen.
  - Man kann eine Liste aller Studierenden der Universität sowie einzelner Kurse ausgeben.

Wir verfolgen hier einen *top-down* Ansatz, d.h. wir modellieren zuerst das Verhalten der Objekte abstrakt, bevor wir uns der Implementierung und den Details der Umsetzung widmen. Das Verhalten von Objekten wird durch das Versenden von Nachrichten und das Bearbeiten der Nachrichten durch Methoden modelliert.
Wir betrachten hier zwei unterschiedliche Möglichkeiten, Verhaltensaspekte zu beschreiben:

- Skizzieren der wesentlichen Anwendungsfälle
- Beschreibung der Nachrichten, die die Objekte verstehen (*Methodenschnittstelle*)

1.1.1 Anwendungsfälle (Use cases)

Ausgangspunkt für die Verhaltensmodellierung sind die wesentlichen Anwendungsfälle des zu entwickelnden Systems. Ein Anwendungsfall ist ein typischer Vorgang, der mit dem zu realisierenden System durchgeführt wird. Anwendungsfälle verdeutlichen auch die Abgrenzung des Systems von seiner Umgebung.
Im Beispiel gibt es vier Anwendungsfälle:



Studierende und Universitätsverwaltung sind Beispiele für ***Akteure***, dargestellt durch Strichmännchen. Ein Anwendungsfalldiagramm (engl. *use case diagram*), wie das oben gezeigte, stellt dar, welche Akteure in welchen Anwendungsfällen involviert sind und gibt somit einen groben Überblick über das System. Im Allgemeinen können Akteure sein:

- Benutzer des Systems
- agierende Softwareteile der Systemumgebung

1.1.2 Methodenschnittstellen

Schnittstellenbeschreibungen sind ein wichtiger Baustein in der Modellierung, aber auch Implementierung von objekt-orientierten Systemen.

- Sie bieten klare Vereinbarungen (*contracts*) über die Interaktion von Klassen und Objekten.
- Sie klassifizieren Objekte nach ihrem Verhalten.
- Sie verbergen die tatsächliche Implementierung (→ Stichwort: Information hiding).
- Sie verhindern (ungewollte) Abhängigkeiten zwischen Klassen.
- Sie vereinfachen die Zusammenarbeit von Entwicklerteams und ermöglichen eine klare Aufgabenteilung.
- Sie erlauben ein einfaches Verändern von Klassenimplementierungen, da nur wenig im Anwendungscode angepasst werden muss, der sich auf die Schnittstellen bezieht und nicht auf die Implementierung.

Für die Schlüsselabstraktionen Student, Universität und Kurse legen wir folgende Methodenschnittstellen fest:

<<interface>> **Student**
getName() : String getMatrikel() : int enroll(String courseTitle) : boolean

<<interface>> **University**
register(String name) : Student deregister(int matrikel) findCourse(String courseTitle) : Course addCourse(Course c) printStudents()

<<interface>> **Course**
getTitle() : String enroll(Student s) : boolean printStudents()

2 Interfaces in Java

Java stellt als Sprachmittel ***Interfaces*** zur Verfügung, um Schnittstellen zu beschreiben.
Ein ***Interface*** deklariert einen Referenztyp T und beschreibt die öffentliche Schnittstelle, die alle Objekte von T haben.
Syntax:

```
InterfaceDeklaration →
        Modifikatorenliste interface ≪ Bezeichner ≫ extendsKlausel {
            KonstantenMethodenDeklarationsListe
        }

extendsKlausel →
        extends TypListe
      | ϵ

TypListe →
        Typ , TypListe
      | Typ

KonstantenMethodenDeklarationsListe →
      | KonstantenDeklaration KonstantenMethodenDeklarationsListe
      | MethodenDeklaration KonstantenMethodenDeklarationsListe
      | ϵ
```

```
KonstantenDeklaration →
    Typ ≪ Bezeichner ≫ = Ausdruck;

MethodenDeklaration →
    TypOderVoid ≪ Bezeichner ≫ (FormaleParameter);
```

Zur Verwendung von `extends` in der Interface-Deklaration verweisen wir auf Abschnitt 3.5.

Für die Schnittstellendeklaration aus dem obigen Beispiel der Universitätsverwaltung ergeben sich die folgenden Implementierungen:

```
interface Course {
  String getTitel();
  boolean enroll(Student s);
  void printStudents();
}
```

```
interface Student {
  String getName();
  int getMatrikel();
  boolean enroll(String courseTitle);
}
```

> **Frage 1:** Geben Sie eine Interface-Implementierung für Universitäten an!

Zu einem Schnittstellentyp T lassen sich keine Objekte erzeugen, die nur zu T gehören. Um eine Schnittstelle verwenden zu können, müssen Klassen diese Schnittstelle implementieren. Eine Klasse stellt für jede Methode, die in dem Interface deklariert wird, eine Implementierung bereit.

In Java wird durch `implements` in der Klassendeklaration angezeigt, welche Schnittstellen durch eine Klasse implementiert werden. Es kann mehr als eine Klasse eine Schnittstelle implementieren.

Beispiel: Implementierung von Schnittstellen Wir wollen in unserem Modell zwei Arten von Kursen unterscheiden, Seminare und Vorlesungen. Seminare unterscheiden sich von Vorlesungen, da sie eine maximale Teilnehmerzahl festlegen. Der Typ `List<Student>` beschreibt dabei eine Liste von Studierenden, denen wir (wie bereits bei den Code-Tagebucheinträgen gesehen) Studierende mittels `add(Student s)` hinzufügen können.

```
import java.util.List;
import java.util.LinkedList;

class Seminar implements Course {
  private String title;
  private int capacity;
  private List<Student> students;

  Seminar(String title, int capacity) {
```

```
    this.title = title;
    this.capacity = capacity;
    this.students = new ArrayList<Student>();
  }
  public String getTitel() {
    return title;
  }
  public boolean enroll(Student s) {
    if (students.size() < capacity) {
      students.add(s);
      return true;
    }
    return false;
  }
  public void printStudents() {
    for(Student s : students) {
      System.out.println(s);
    }
  }
}
```

```
import java.util.List;
import java.util.ArrayList;

class Lecture implements Course {
  private String title;
  private String lecturer;
  private List<Student> students;

  Lecture(String title, String lecturer) {
    this.title = title;
    this.lecturer = lecturer;
    this.students = new ArrayList<Student>();
  }

  public String getTitel() {
    return title;
  }
  public String getLecturer() {
    return lecturer;
  }
  public boolean enroll(Student s) {
    return students.add(s);
  }
  public void printStudents() {
    for(Student s : students) {
      System.out.println(s);
    }
  }
}
```

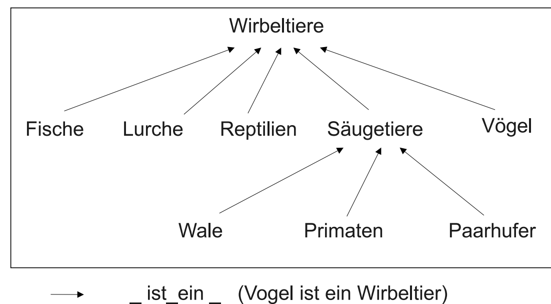
Frage 2: In Kapitel 11 haben wir verschiedene Möglichkeiten gesehen, Listen mit Integer-Elementen zu implementieren.

- Schreiben Sie ein Interface `ListOfInt`, das den Schnittstellentyp für Listen mit Integer-Elementen definiert, das die Methoden `add`, `get`, `size` bereitstellt!
- Wie müssen die Klassen `IntList`, `DLIntList` und `ArrayList` abgeändert werden, damit sie dieses Interface implementieren?

3 Klassifizieren von Objekten

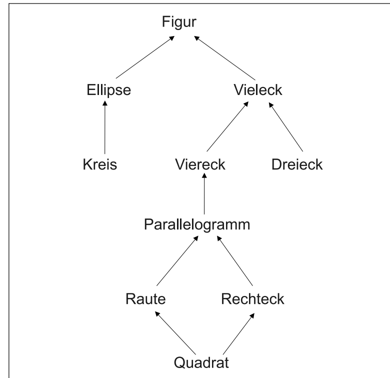
Literaturhinweis: Kapitel 3.1 und 3.2 aus A. Poetzsch-Heffter: Konzepte objektorientierter Programmierung. Springer, 2009.

Klassifikation ist eine zentrale Grundlage der objektorientierten Modellierung und Programmierung. Klassifizieren ist eine allgemeine Technik, mit der Wissen über Begriffe, Dinge und deren Eigenschaften hierarchisch strukturiert wird. Das Ergebnis nennen wir eine ***Klassifikation***.
Klassifikationen werden in vielen Wissenschaften angewandt. Das biologische Fachgebiet der Systematik beschäftigt sich z.B. mit der Einteilung, Benennung und Bestimmung von Lebewesen.



Üblicherweise stehen die allgemeineren Begriffe oben, die spezielleren unten. Es gibt dabei abstrakte Klassen (ohne "eigene" Objekte, z.B. die Klasse "Wirbeltiere") und nicht abstrakte Klassen, von denen Objekte abgeleitet werden können.
Klassifikationen können baumartig oder DAG[1]-artig sein. D.h. Klassifikationen haben eine Richtung ("ist ein"-Beziehung) und dürfen keine Zyklen enthalten. Hier ein weiteres Beispiel einer Klassifikation für geometrische Figuren, die DAG-artig ist:

[1]DAG = directed acyclic graph (dt. gerichteter azyklischer Graph)



3.1 Klassifikation in der Softwaretechnik

In der objekt-orientierten Modellierung klassifizieren wir Objekte auf Grund ihrer Eigenschaften:

- Alle Objekte mit ähnlichen Eigenschaften werden zu einer Klasse zusammen gefasst.
- Die Klassen werden hierarchisch geordnet.

Diese Klassifikation beruht auf den Schnittstellen der Klassen/Objekte sowie den Eigenschaften der Objekte.

3.2 Klassifikationen und Typisierung

Klassifikation und Typisierung sind miteinander verflochten, da ein Typ die Eigenschaften von Objekten bzw. Werten beschreibt. Man versucht daher jede Klasse bzw. jede Art einer Klassifikation in einem Programm durch einen eigenen Typ zu realisieren.

Wiederholung: Typsystem von Java Werte in Java sind

- die Elemente der Basisdatentypen,
- Referenzen auf Objekte,
- der Wert `null`.

Jeder Wert in Java hat (mindestens) einen Typ. Der Typ charakterisiert die Operationen, die auf den Werten des Typs zulässig sind. Man unterscheidet in Java dabei

- die vordefinierten elementaren Basisdatentypen,
- die durch Klassen deklarierten Typen und

- die durch Interfaces deklarierten Typen.

Darüber hinaus gibt es noch die Arraytypen mit Typkonstruktor `[]` . Array-, Klassen- und Interface-Typen fasst man unter dem Namen ***Referenztypen*** zusammen. [2]
Auch jedes Objekt in Java hat (mindestens) einen Typen. Man unterscheidet in Java dabei nicht zwischen dem Typ eines Objekts und dem Typ der Referenz auf dieses Objekt, da es durch den Programmkontext immer klar ist, wann ein Objekt und wann eine Objektreferenz gemeint ist.[3] Klassen- und Schnittstellentypen beschreiben, welche Nachrichten ihre Objekte verstehen bzw. welche Methoden sie besitzen.

3.3 Sub-/Supertypen

Wir führen nun eine partielle Ordnung $\leq$ auf Typen ein, so dass speziellere Typen gemäß dieser Ordnung kleiner als ihre allgemeineren Typen sind und Objekte speziellerer Typen auch zu den allgemeineren gehören:

Wenn $S \leq T$ gilt, dann sind alle Objekte vom Typ S auch vom Typ T.

Wir verwenden folgende Begriffe und Notation:

- S ist ein ***Subtyp*** von T, und T ist ein ***Supertyp*** von S.
- Wenn $S \leq T$ und $S \neq T$, heißt S ein ***echter*** Subtyp von T, in Zeichen $S < T$.
- Wenn $S < T$ und es kein U mit $S < U < T$ gibt, dann heißt S ein ***direkter*** Subtyp von T.

Objekte eines Typs S, der ein Subtyp eines Typs T ist ($S \leq T$), können an allen Stellen verwendet werden, an denen Objekte vom Typ T zulässig sind.

Beispiel Eine Methode, die eine Referenz auf ein `Course`-Objekt als Parameter erwartet, kann auch mit einem Ausdruck vom Typ `Lecture` aufgerufen werden, da `Lecture` ein Subtyp von `Course` ist:

```
class Modulhandbuch {
  ...
  void addCourse(Course c) { ... }
}

// Benutzung:
Modulhandbuch mh = new Modulhandbuch();
// Hier wird ein Lecture-Objekt übergeben, wo ein Course erwartet wird
mh.addCourse(new Lecture("Compilerbau", "Bieniusa"));
```

[2]Der Wert `null` gehört zu allen Referenztypen.
[3]In anderen Programmiersprachen wie C++ werden Objekte und Objektreferenzen in Bezug auf den Typ klar unterschieden.

Frage 3: `Seminar` ist ein Subtyp von `Course`. Welche der beiden Zuweisungen in der folgenden Methode wird vom Java-Compiler nicht akzeptiert?

```
void test(Course[] courses, Seminar[] seminars) {
    courses[0] = seminars[0];
    seminars[1] = courses[1];
}
```

Typen als Mengen Vereinfachend betrachtet, kann man Typen als die Menge ihrer Objekte bzw. Werte auffassen.
Sei $M(S)$ die Menge der Objekte vom Typ S. In einem Typsystem mit Subtyping gilt dann für Typen S und T:

$$S \leq T \quad \text{impliziert} \quad M(S) \subseteq M(T)$$

D.h. die Subtyp-Beziehung entspricht der Teilmengen-Beziehung auf den zugehörigen Objektmengen. Die Interpretation der Subtyp-Beziehung entspricht auch der informellen und intuitiven Vorstellung, dass in einer Klassifikation die allgemeinere Klasse die Elemente der spezielleren umfassen.

Beispiel: Subtypbeziehungen Für die elementaren Datentypen in Java gelten z. B. die folgenden Subtypbeziehungen:

$$\begin{aligned} \texttt{float} &< \texttt{double} \\ \texttt{int} &< \texttt{long} \\ \texttt{long} &< \texttt{float} \end{aligned}$$

3.4 Die Klasse `Object`

Die Klasse `Object` spielt in Java eine spezielle Rolle. Sie ist die Wurzel der Typhierarchie der Referenztypen und stellt einige wichtige Methoden (z.B. `toString()`) bereit.
In Java gilt:

→ Alle Referenztypen sind Subtypen von `Object`.

→ Alle Objekte (inkl. Arrays) erben die Methoden von `Object` (⇒ nächste Vorlesung "Vererbung").

3.5 Subtyping und Interface-Deklarationen

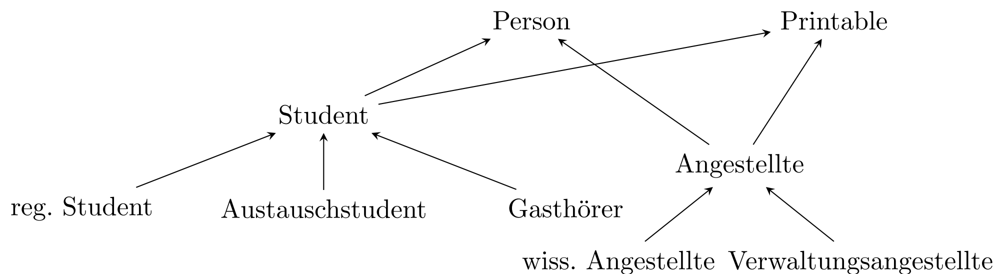
In Java muss bei der Klassen- bzw. Schnittstellendeklarationen für den entsprechenden Referenztyp T explizit angegeben werden, was die direkten Supertypen von T sein sollen. Fehlt die Angabe von Supertypen, wird der Typ `Object` als direkter Supertyp angenommen.
Implementiert eine Klasse ein (oder mehrere) Interface, so sind diese Interface-Typen direkte Supertypen des Klassentyps.

Beispiel Für die Kurse im Universitätsverwaltung bestehen folgende Subtyp-Beziehungen:

$$\texttt{Lecture} < \texttt{Course}$$
$$\texttt{Seminar} < \texttt{Course}$$

Zwischen `Lecture` und `Seminar` gibt es keine Subtyp-Beziehung.
Eine Interface-Deklaration für einen Referenztypen T kann die Interface-Deklaration eines oder mehrerer anderer Typen $S_1, \ldots, S_n$ erweitern. Dazu dient die `extends`-Klausel bei der Deklaration. Es gilt dann, dass T ein direkter Subtyp für jeden der Typen $S_1, \ldots, S_n$ ist. Implizit ist außerdem der Klassentyp `Object` ein Supertyp aller Interface-Typen.

Beispiel Wir ergänzen unser Universitätsinformationssystem um folgende verfeinerte Klassifizierung für Personen:



Das Typ `Printable` umfasst alle Objekte, die eine Methode `print()` für die Ausgabe auf Konsole bereitstellen. Dies können neben den hier aufgelisteten Personen z.B. auch Kurse sein. Der Typ `Person` abstrahiert von den verschiedenen Arten von Studierenden und Angestellten. Wir können dies folgendermaßen in Interface- und Klassendeklarationen umsetzen:

```
interface Printable {
  void print();
}

interface Person {
  String getName();
  String getBirthdate();
}

interface Angestellte extends Person, Printable {
    String getAbteilung();
}
interface Student extends Person, Printable {
    int getMatrikel();
}

class WissAngestellte implements Angestellte { ... }
class VerwAngestellte implements Angestellte { ... }
class RegulaererStudent implements Student { ... }
```

```
class AustauschStudent  implements Student { ... }
class Gasthoerer        implements Student { ... }
```

- Die Typen `Person` und `Printable` haben nur `Object` als Supertypen.
- Der Typ `Angestellter` hat `Person` und `Printable` als direkte Supertypen; das gleiche gilt für `Student`.
- Der Type `Austauschstudent` hat als direkten Supertypen den Typ `Student`; weitere Supertypen sind `Person`, `Printable` und `Object`.
- Methodensignaturen aus den Supertypen müssen in der Interface-Deklaration nicht nochmals aufgeführt werden (*Signaturvererbung*).

3.6 Dynamische Methodenauswahl

Die Auswertung von Ausdrücken vom (statischen) Typ T kann Ergebnisse haben, die von einem Subtyp sind.

Beispiel Folgender Code-Schnipsel gibt die Liste der Teilnehmer für die Veranstaltung "Graphtheorie" aus:

```
University university = ....;

Course c = university.findCourse("Graphtheorie");
c.printStudents();
```

Dabei ist nicht klar, ob die Veranstaltung "Graphtheorie" eine Vorlesung oder ein Seminar ist. Damit stellt sich die Frage, wie Methodenaufrufe (im Beispiel `printStudents()`) auszuwerten sind.
Die auszuführende Methode zu einem Methodenaufruf:

`Ausdruck` . ≪ *Bezeichner* ≫ (`AktuelleParameterListe`)

wird wie folgt bestimmt:

1. Werte zunächst den Ausdruck aus; Ergebnis ist das *Zielobjekt*.
2. Werte dann die aktuellen Parameter aus.
3. Führe die Methode mit dem entsprechenden Methodennamen des Zielobjekts mit den aktuellen Parametern aus.

Dieses Verfahren nennt man ***dynamische Methodenauswahl*** oder ***dynamisches Binden*** (engl. *dynamic method binding*).
Die Unterstützung von Subtypen und dynamischer Methodenauswahl ist entscheidend für die verbesserte Wiederverwendbarkeit und Erweiterbarkeit, die durch Objektorientierung erreicht wird. Zusätzlich werden diese Aspekte auch durch Vererbung unterstützt (⇒ siehe Kapitel "Vererbung").

Beispiel: Erweiterbarkeit Wir gehen von einem Programm aus mit der Methode:

```
void printAll(Printable[] df) {
   int i;
   for(i = 0; i < df.length; i++) {
      df[i].print();
   }
}
```

`Printable` ist dabei Supertyp von `Student`, `Angestellte`, `WissAngestellte` und `VerwAngestellte`. Das Programm soll nun erweitert werden, um auch Professoren/-innen und studentische Hilfskräfte behandeln zu können. Es reicht zwei weitere Klassen hinzuzufügen:

```
class Professor
   implements Person, Printable  {... }

class StudHilfskraft
   extends Student { ... }
```

Eine Änderung der Methode `printAll` ist **NICHT** nötig! Dies ist ein entscheidender Vorteil, in dem die dynamische Methodenauswahl in Java begründet ist.

3.7 Weitere Aspekte der Subtypbildung

Bei der Verwendung von Subtypen gilt es einige weitere Dinge zu beachten. Näheres werden wir in Kapitel 16 und 17 behandeln.

Zyklenfreiheit Die Subtyprelation darf keine Zyklen enthalten.[4] Folgendes Fragment ist also in Java nicht zulässig:

```
interface C extends A { ... }
interface B extends C { ... }
interface A extends B { ... }
```

Die Typhierarchie enthält den Zyklus $A \leq B \leq C \leq A$; dies liefert einen Fehler beim Compilieren.

Subtyprelation bei Arrays Jeder Array-Typ mit Komponenten vom Typ S ist ein Subtyp von `Object`:

$$\texttt{S[]} \leq \texttt{Object}$$

Aus diesem Grund ist folgende Zuweisung zulässig:

```
Object ov = new String[3];
```

Ist $S \leq T$ für Referenztypen S und T, dann ist `S[]` $\leq$ `T[]`. Daher ist folgende Zuweisung zulässig:

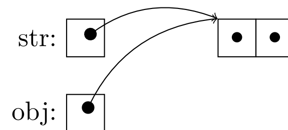
```
Person[] pv = new Student[3];
```

[4] Andernfalls wäre sie keine Ordnung.

Diese Festlegung der Subtypbeziehung zwischen Arraytypen ist in vielen Fällen praktisch, aber birgt auch Probleme. Statische Typsicherheit ist damit nicht mehr gegeben:

```
1 String[] str = new String[2];
2 Object[] obj = str;
3 obj[0] = new Object(); // Laufzeitfehler: ArrayStoreException
4 int strl   =  str[0].length();
```

Nach Ausführen von Zeile 2 liegt folgender Speicherzustand vor:



Die beiden Referenzen `str` und `obj` verweisen auf das String-Array mit den zwei `null` -Elementen. In Zeile 3 wird nun versucht, in das Array ein Objekt vom Typ `Object` einzufügen. Dies schlägt zur Laufzeit fehl. Java überprüft bei jeder Modifikation des Arrays, ob die Elemente des Arrays dem deklarierten Typ (hier: `String`) entsprechen. Andernfalls würden Zugriffe auf die Objekte wie in Zeile 4 undefiniert sein (Objekte vom Typ `Object` haben keine Methode `length()`).

Polymorphie Polymorphie ist die Eigenschaft, verschiedene Formen annehmen zu können. Die Form der Polymorphie in Typsystemen mit Subtypen heißt ***Subtyp-Polymorphie***. Dabei gehören Objekte eines Subtyps auch den entsprechenden Supertypen an. Subtyp-Polymorphie erlaubt es z.B. *inhomogene* Arrays oder Listen zu verwenden, d.h. mit Elementen unterschiedlichen Typs (s.o.). Eine weitere wichtige Art der Polymorphie, parametrische Polymorphie, werden wir im Zusammenhang mit Collections kennenlernen.