

Softwareentwicklung II (IB)

Ableiten konkreter Klassen

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 31.05.2018 14:37

Inhaltsverzeichnis

Idee	1
Beispiel: Zähler	1
Erweitern der Funktionalität	2
Kopierter Code	3
Begriffe	3
Definition	3
Beispiel: Zähler mit Merker	4
Zugriffsschutz <code>protected</code>	4
Stufen des Zugriffsschutzes	5
Datenkapselung der Basisklasse	5
Aufruf ererbter Methoden	6
Ableiten zur Modifikation	6
Beispiel: Zähler mit Anschlag	6
Redefinition einer Methode	7
Überladen vs. Redefinition	8
Einschränken einer Funktionalität	8
Dynamisches Binden redefinierter Methoden	9
Statisches Binden	9
Binden von Objektvariablen	10
Aufruf des Basisklassen-Konstruktors	10
Beispiel: Zähler mit Rücksetzen	11
Bezug auf die Basisklasse mit <code>super</code>	12
Endlosrekursion wegen fehlendem <code>super</code>	13

Rückgabe des eigenen Objektes	13
Covarianter Ergebnistyp	14

Idee

- Interfaces isolieren gleiche Eigenschaften verwandter Klassen
- Andere Beziehung zwischen Klassen: Erweiterung oder Modifikation vorhandener Eigenschaften
- Erweitern und Modifizieren mit **konkreten Basisklassen**

Beispiel: Zähler

- Ein Zähler hat einen Zählerstand (ganze, nicht-negative Zahl), dieser kann ...
 - weitergezählt,
 - abgelesen und
 - auf 0 zurückgestellt werden.

- Klasse OpenCounter:

```
class OpenCounter {
    private int count = 0; // Zählerstand
    void step() {          // weiterzählen
        count++;
    }
    int read() {           // ablesen
        return count;
    }
    void reset() {        // zurückstellen
        count = 0;
    }
}
```

- Anwendungsbeispiel:

```
class CounterApplication {
    public static void main(String[] args) {
        OpenCounter c = new OpenCounter();
        for(int i = 0; i < 10; i++) {
            c.step();
            System.out.printf("%d ", c.read());
        }
        System.out.println();
    }
}
```

```
}  
}
```

Ausgabe:

```
1 2 3 4 5 6 7 8 9 10
```

- `OpenCounter` muss mutable sein (= Referenzsemantik haben), nicht immutable (Wertesemantik)

Erweitern der Funktionalität

- Annahme: Neue Art von Zählern gefordert, die sich zusätzlich einen Zählerstand merken kann
- Neue Methoden

mark() aktuellen Zählerstand merken

recall() auf zuletzt gemerkten Stand zurücksetzen (null, wenn vorher kein mark)

- Naive Idee: Code von `OpenCounter` kopieren und verlängern:

```
class MemCounter {  
    ... Kopie von OpenCounter ...  
    private int memory = 0; // gemerkter Zählerstand  
    void mark() {           // merken  
        memory = count;  
    }  
    void recall() { // zurücksetzen auf gemerkten Stand  
        count = memory;  
    }  
}
```

Kopierter Code

- Ergebnis: Zwei isolierte Klassen ohne Bezug
- Problem: Fehler im Code von `OpenCounter` auch in `MemCounter`, muss zweimal identisch korrigiert werden
- Idee: Gemeinsames Interface `CounterInterface` für `OpenCounter` und `MemCounter`
- Jetzt: Verwandtschaft im Code dokumentiert (beide geben an `implements CounterInterface`)
- Aber: weiterhin duplizierter Code, keine Hilfe bei Fehlerkorrektur

Begriffe

- Neuer Mechanismus: **Ableitung** (derivation) einer zugrunde gelegten, einfacheren Klasse zu einer erweiterten Klasse
- Bezeichnungen:
 - zugrunde liegende Klasse = **Basisklasse** (base class, super class)
 - erweiterte Klasse = **abgeleitete Klasse** (derived class)

OpenCounter = Basisklasse, MemCounter = abgeleitete Klasse
- Auch **Vererbung** (inheritance): MemCounter erbt von OpenCounter

Definition

- Definition einer abgeleiteten Klasse nennt lediglich Unterschiede zur Basisklasse
- Syntax einer abgeleiteten Klasse **Derived**:


```
class Derived extends Base {...}
```
- Ableitung ist asymmetrisch:
 - Die abgeleitete Klasse kennt ihre Basisklasse (siehe Syntax)
 - Eine Basisklasse weiß nichts von abgeleiteten Klassen

Beispiel: Zähler mit Merker

- Modifizierte Fassung des vorhergehenden Beispiels, jetzt ohne den Abschnitt „... Kopie von OpenCounter ...“:

```
class MemCounter extends OpenCounter {
    private int memory = 0; // gemerkter Zählerstand
    void mark() {           // merken
        memory = count;
    }
    void recall() { // zurücksetzen auf gemerkten Stand
        count = memory;
    }
}
```

Klasse:	OpenCounter	MemCounter
Konstruktor:	(automatisch)	(automatisch)
Objektvariablen:	int count	(← erbt) int memory

Methoden:	void step()	(← erbt)
	void reset()	(← erbt)
	int read()	(← erbt)
		void mark()
		void recall()

Zugriffsschutz `protected`

- Objektvariable `count` ist `private` in `OpenCounter`: Zugriff nur in `OpenCounter`
- Problem: `MemCounter` braucht `count`, hat aber keinen Zugriff!
- Lösung: Zugriffsschutz „`protected`“
- `protected`-Objektvariablen, -Klassenvariablen und -Methoden in allen abgeleiteten Klassen verfügbar
- Modifizierte Fassung von `OpenCounter`:

```
class OpenCounter {
    protected int count = 0; // statt private
    ...
}
```

Stufen des Zugriffsschutzes

- Java kennt vier Stufen des Zugriffsschutzes

<code>private</code>	nur die eigene Klasse
<code>package</code> (Voreinstellung, ohne Modifier)	alle Klassen im gleichen Package
<code>protected</code>	wie <code>package</code> + alle abgeleiteten Klassen
<code>public</code>	jede Klasse, ohne Einschränkung

- `protected` öffnet den Zugriff auch für
 - ... indirekt (über mehrere Stufen) abgeleiteten Klassen
 - ... abgeleiteten Klassen in anderen Packages
 - ... Klassen im gleichen Package, ob abgeleitet oder nicht
 ⇒ möglicherweise freizügiger, als beabsichtigt
- `protected` weniger nützlich, als es hier den Anschein hat

Datenkapselung der Basisklasse

- Direkter Zugriff auf Objektvariablen weiterhin fragwürdig, ob ererbt oder nicht
- Kapseln hinter Gettern und Settern empfehlenswert, auch gegenüber abgeleiteten Klassen
- Beispiel MemCounter: Methode mark besser mit Aufruf von read (Getter):

```
class MemCounter extends OpenCounter {
    ...
    void mark() {
        memory = read(); // vorher: memory = count;
    }
    ...
}
```

- Methode recall: Neuer Setter setCount der Basisklasse erforderlich

```
class MemCounter extends OpenCounter {
    ...
    void recall() {
        setCount(memory); // vorher count = memory;
    }
    ...
}
```

- protected ausreichend für setCount in OpenCounter

Aufruf ererbter Methoden

- Für Anwendungen: Ererbte Methoden und Methoden einer Klasse selbst nicht unterscheidbar

```
MemCounter c = new MemCounter();
c.step(); // ererbt von OpenCounter
c.mark();
c.reset(); // ererbt von OpenCounter
c.recall();
```

- Methodenaufruf: JVM sucht zuerst in der Klasse selbst, dann in der Basisklasse, dann in deren Basisklasse usw.
- Compiler stellt sicher, dass die JVM in jedem Fall eine Methode findet

Ableiten zur Modifikation

- Abgeleitete Klasse kann Methoden **redefinieren** (= neu definieren, überschreiben), die bereits in der Basisklasse definiert sind
- Regeln:
 - Name und Parameterliste müssen exakt übernommen werden
 - der Zugriffsschutz darf gelockert werden
 - Der Ergebnistyp darf vom Ergebnistyp der redefinierten Methode abgeleitet sein
 - Der Rumpf kann komplett ersetzt werden
- Die Basisklasse wird hier nicht erweitert (wie im Beispiel `MemCounter`), sondern verändert

Beispiel: Zähler mit Anschlag

- Beispiel: Neue Variante von Zählern die nur bis zu einem bestimmten Anschlag laufen und dort stehen bleiben
- Neue Klasse `LtdCounter` (limited counter)
- `LtdCounter` erbt von `OpenCounter`
- Zusätzlich:
 - `final`-Objektvariable `limit` zum Speichern des Anschlags
 - Getter für den Anschlag
 - Konstruktor zum Initialisieren des Anschlags
- Implementierung:

```
class LtdCounter extends OpenCounter {
    private final int limit;
    LtdCounter(int l) {
        limit = l;
    }
    int getLimit() {
        return limit;
    }
    ...
}
```

Redefinition einer Methode

- `LtdCounter` erbt `reset`, `read`, `step`, `setCount` von `OpenCounter`

- Unverändert brauchbar, außer `step`: Soll nicht endlos weiterzählen, sondern am Anschlag stoppen
- `LtdCounter` redefiniert `step`:

```

class LtdCounter extends OpenCounter {
    ...
    @Override
    void step() {           // unveränderter Kopf
        if(read() < limit) { // neuer Rumpf
            count++;
        }
    }
    ...
}

```

(Zur Anweisung `count++` später mehr)

Klasse:	<code>OpenCounter</code>	<code>LtdCounter</code>
Konstruktor:	(automatisch)	<code>LtdCounter(int)</code>
Objektvariablen:	<code>int count</code>	(← erbt) <code>int limit</code>
Methoden:	<code>void step()</code> <code>void reset()</code> <code>int read()</code>	<code>void step()</code> (← erbt) (← erbt) <code>int getLimit()</code>

Überladen vs. Redefinition

- Bei abweichender Parameterliste Überladen einer ererbten Methode, keine Redefinition!

```

class OpenCounter {
    @Override
    void step()
    {...}
}

class LtdCounter extends OpenCounter {
    void step(int i)
    {...}
}

```

In `LtdCounter` zwei Methoden `step`: Eine erbt, die andere neu definiert

- Typischer und tückischer Fehler im Zusammenhang mit `equals`

Einschränken einer Funktionalität

- Abgeleitete Klassen können die Funktionalität Basisklasse erweitern oder ändern, aber keinesfalls einschränken
- Kein Sprachmittel zum Ausblenden ererbter Methoden oder Objektvariablen
- Beispiel: Redefinition mit reduziertem Zugriffsschutz unzulässig:

```
class OpenCounter {  
    void step()  
    {...}  
}
```

```
class LtdCounter extends OpenCounter {  
    @Override  
    private void step() // Fehler  
    {...}  
}
```

- Redefinition mit gelockertem Zugriffsschutz erlaubt:

```
class OpenCounter ...
```

```
class LtdCounter extends OpenCounter {  
    @Override  
    public void step() // ok  
    {...}  
}
```

- Fazit: Ein abgeleitetes Objekt kann alles, was ein Basisklassenobjekt kann, möglicherweise auch mehr, aber keinesfalls weniger

Dynamisches Binden redefinierter Methoden

- Abgeleitete Klassen sind kompatibel zu Basisklassen (vgl. auch Interfaces)
- Folge: Objekt einer abgeleiteten Klasse kann ein Basisklassenobjekt in jedem Kontext ersetzen
- Redefinierte Methoden werden dynamisch gebunden
- Beispiel: Erzeugen eines `LtdCounter` statt eines `OpenCounter` in der Beispielanwendung, `step` und `read` werden dynamisch gebunden:

```
class CounterApplication {
    public static void main(String[] args) {
        OpenCounter c = new LtdCounter(5);
        for (int i = 0; i < 10; i++) {
            c.step();
            System.out.printf("%d ", c.read());
        }
        System.out.println();
    }
}
```

Ausgabe:

```
1 2 3 4 5 5 5 5 5 5
```

Statisches Binden

- Java bindet im Allgemeinen dynamisch
- In einigen Fällen wird **statisch gebunden** (= der Compiler ordnet Aufrufe und Methoden fest zu):

Statische Methoden Kein Zielobjekt, richten sich an eine ganze Klasse. Ohne Zielobjekt kein dynamischer Typ, keine Entscheidungsgrundlage für dynamisches Binden.

Konstruktoren Kein Zielobjekt, der Konstruktor soll ja erst eines liefern. Siehe vorhergehender Punkt.

Private Methoden Außerhalb der eigenen Klasse nicht sichtbar. Stehen überhaupt nicht zur Wahl.

Können in abgeleiteten Klassen neu definiert werden. Das ist keine Redefinition.

Binden von Objektvariablen

- Objektvariablen werden immer statisch gebunden
- Compiler legt beim Übersetzen endgültig fest, welche Objektvariablen benutzt werden. Zur Laufzeit keine Entscheidung mehr.

```
class Base {
    int data = 1;
}
```

```
class Derived extends Base {
```

```
    int data = 2;
}
```

- Statischer Typ von `x` ist `Base`, dessen Objektvariable wird ausgegeben. Der dynamische Typ von `x` (`Derived`) wird ignoriert:

```
public static void main(String[] args) {
    Base x = new Derived();
    System.out.println(x.data); // gibt 1 aus
}
```

- Unabhängig davon werden Objektvariablen vererbt (siehe `MemCounter`)

Aufruf des Basisklassen-Konstruktors

- Jeder Konstruktor einer abgeleiteten Klasse muss zuerst einen Basisklassen-Konstruktor aufrufen
- Folge: Basisklassenobjekt vollständig initialisiert, wenn abgeleiteter Konstruktor abläuft
- Voreinstellung: Default-Konstruktor der Basisklasse
- Beispiel: Custom-Ctor von `LtdCounter` ruft automatisch definierten Default-Ctor von `OpenCounter`:

```
class LtdCounter extends OpenCounter {
    LtdCounter(int l) {
        // Automatischer Aufruf von OpenCounter()
        limit = l;
    }
    ...
}
```

- Basisklassen-Default-Konstruktor explizit aufrufen mit `super()`;
- Beispiel: äquivalent zum vorhergehenden:

```
class LtdCounter extends OpenCounter {
    LtdCounter(int l) {
        super(); // Expliziter Aufruf von OpenCounter()
        limit = l;
    }
    ...
}
```

- Einschränkungen des Aufrufs von `super`:
 - nur ein Aufruf
 - erste Anweisung im Konstruktorrumpf

Beispiel: Zähler mit Rücksetzen

- Beispiel: Klasse LoopCounter: Zähler laufen bis zum Anschlag (wie LtdCounter), springen dann auf null zurück
- Ableiten von LtdCounter, step erneut redefinieren:

```

class LoopCounter extends LtdCounter {
    LoopCounter(int l)
    {...}
    void step() {
        if(read() == getLimit()) {
            reset(); // Anschlag erreicht → Rücksetzen auf 0
        } else {
            count++;
        }
    }
    ...
}

```

(Zur Anweisung count++ später mehr)

- Basisklasse LtdCounter hat keinen Default-Konstruktor \Rightarrow super() kann nicht aufgerufen werden, weder implizit noch explizit
- super mit Argumentliste: Aufruf eines **Basisklassen-Custom-Konstruktors**

```

class LoopCounter extends LtdCounter {
    LoopCounter(int l) {
        super(l); // Expliziter Aufruf von LtdCounter(int)
    }
    ...
}

```

Klasse:	OpenCounter	LtdCounter	LoopCounter
Konstruktor:	(automatisch)	LtdCounter(int)	LoopCounter(int)
Objektvariablen:	int count	(← erbt) int limit	(← erbt)
Methoden:	void step() void reset() int read()	void step() (← erbt) (← erbt)	void step() (← erbt) (← erbt)
		int getLimit()	(← erbt)

Bezug auf die Basisklasse mit `super`

- `super` in normalen Methoden referenziert Basisklassenobjekt als Zielobjekt (weitere Nutzung von `super`, unabhängig vom Aufruf eines Basisklassen-Konstruktors)
- `super` startet die Suche nach einer passenden Methode (dynamisches Binden) in der Basisklasse, statt in der eigenen Klasse
- Beispiel: Redefiniertes `step` von `LoopCounter` mit expliziten Aufrufen der Basisklassenmethoden:

```
class LoopCounter extends LtdCounter {
    @Override
    void step() {
        if(super.read() == super.getLimit()) {
            super.reset();
        } else {
            super.step();
        }
    }
}
```

- `super` im Beispiel unnötig für `read`, `reset`, `getLimit`: Dynamisches Binden trifft, mit und ohne `super`, auf die gleichen Definitionen (in `OpenCounter` bzw. `LtdCounter`)

Endlosrekursion wegen fehlendem `super`

- `super` im vorhergehenden Beispiel unverzichtbar bei `super.step()`

```
class LoopCounter extends LtdCounter {
    @Override
    void step() {
        if(read() == getLimit()) {
            reset();
        } else {
            step(); // statt super.step();
        }
    }
    ...
}
```

- Aufruf von `step`:
 - Suche nach `step` startet in der eigenen Klasse (dynamisches Binden)
 - findet die eigene Definition

- ruft sich selbst auf
- neuer Aufruf von `step`
- Suche nach `step` startet in der eigenen Klasse
- ...
- Praxis: Nur begrenzt Speicherplatz für noch nicht zurückgekehrte Aufrufe (Call-Sequence)
- Speicherplatz erschöpft \Rightarrow Programmabbruch
- Keine Verkettung von `super`: spricht nur das unmittelbare Basisklassenobjekt an, kann die Basisklasse der Basisklasse nicht erreichen

Rückgabe des eigenen Objektes

- `step` liefert nichts zurück (siehe `OpenCounter`):

```
void step() {  
    count++;  
}
```

- Alternative: Sich selbst (= `this`, eigenes Objekt) zurückliefern

```
OpenCounter step() {  
    count++;  
    return this;  
}
```

- Ermöglicht Kettenaufrufe in einer Anweisung:

```
OpenCounter c = new OpenCounter();  
c.step().step().step(); // 3x hochzählen
```

- Statt `void` das eigene Objekt zurückgeben \Rightarrow Methode flexibler einsetzbar
- Beispiel: `StringBuilder`

Covarianter Ergebnistyp

- **Covarianter Ergebnistyp**: Redefinition von Methoden mit kompatiblen Ergebnistypen
- Beispiel: Redefinierte Fassungen von `step` mit Rückgabe des eigenen Objektes
- Ergebnistyp in jeder abgeleiteten Klasse anders:

```
class OpenCounter {  
    OpenCounter step()  
    {...}  
}  
class LtdCounter extends OpenCounter {  
    @Override  
    LtdCounter step()  
    {...}  
}  
class LoopCounter extends LtdCounter {  
    @Override  
    LoopCounter step()  
    {...}  
}
```