15 Klassen und Vererbung, weitere Details

15.1 Abstrakte Klassen

Abstrakte Klassen sind Zwitter zwischen normalen Klassen und Interfaces:

- Zum einen benennen sie nicht ausimplementierte Methoden (wie Interfaces).
- Zum anderen können sie ausimplementierte Methoden enthalten (wie Klassen).
  Man kann auch nur von maximal *einer* abstrakten Klasse erben (wie bei Klassen).
- Anders als bei Interfaces sind die Methoden nicht automatisch `public`.

```
abstract class Agent {
   private String id ;
   Agent( String id ) {
      this.id = id ;
   }
   getId() { return id ; }
   abstract void doSimStep() ; // force derived classes to implement this!
}
class Person extends Agent {
   ...
   @Override
   void doSimStep() {
      ...
   }
   ...
}
```

Abstrakte Klassen sind zur Typ-Definition ungeeignet (wg. der fehlenden Mehrfach-Vererbung); verwende dafür Interfaces.

Quelle: Bloch, J. Effective Java. Second Edition Addison-Wesley, 2008, Item 18 "Prefer interfaces to abstract classes".

Abstrakte Klassen *sind* geeignet als "skeletal implementations" von Interfaces:

```
interface List extends Collection {
    ...
}
abstract class AbstractList extends AbstractCollection implements List {
    ...
}
class ArrayList extends AbstractList {
    ...
}
```

Das ist aber bereits fortgeschrittenes objekt-orientiertes Design, und (für Objekte, die auch von anderen verwendet werden) unserer Erfahrung nach nach schwieriger als es vielleicht aussieht.

15.2 Implementations-Vererbung: Nutzen und Probleme

Vorteil: Auf bereits Vorhandenes zurückgreifen und dieses nur noch zu erweitern ...

- ... spart Aufwand
- ... vermeidet Redundanz

Früher hat man Implementations-Vererbung zentraler gesehen. Inzwischen:

"parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation' [Sny86]. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change."

"our experience is that designers overuse inheritance as a reuse technique"

Quelle: Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley, 2001, Chapter 1.

[Sny86]: Alan Snyder. Encapsulation and inheritance in object-oriented languages. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 38–45, Portland, OR, November 1986. ACM Press.

...

...

"Design and document for inheritance or else prohibit it".

Quelle: Bloch, J. Effective Java. Addison Wesley, 2008. Item 17.

Das GridWorld Framework in den Übungen fällt in die erste Kategorie: designed and documented for inheritance.

Sie selber sollten vorzugsweise in Ihrer package bleiben:

```
class MyActor extends Actor {
// (note missing ``public'')
   @Override
   public void act()
   // (public enforced by Actor)
      ...
   }
   ...
}
```

15.3 Vererbung, Umgangssprache vs. Programmiersprache

Ich finde das Java-Schlüsselwort **extends** sehr passend: Eine abgeleitete Klasse ist eine *Erweiterung* des Basis-Klasse: Sie hat zusätzliche Variablen, zusätzliche Methoden, ...

Manchmal wird Vererbung als **"isa"-Relation** (isa = is a = ist ein) beschrieben: Ein Erwachsener ist eine Person, aber eine Person ist nicht unbedingt erwachsen.

```
class Person {
   int getAge() { ... }
}
class Adult extends Person {
   String getEmployer() ;
}
```

`getEmployer()` macht (auch inhaltlich) keinen Sinn für eine Person, die nicht erwachsen ist.

Leider kann “isa” in die Irre führen:

Ein Quadrat ist ein Rechteck; ein Rechteck ist nicht unbedingt quadratisch. Daraus sollte folgen:

```
class Rectangle {
   double getEdgeLength1() { ... }
   double getEdgeLength2() { ... }
}
class Square extends Rectangle {
   ?????
}
```

Ein Rechteck hat zwei Kantenlängen, ein Quadrat nur eine.

Ein Quadrat hat also eine Variable und damit einen getter = eine Methode *weniger* als ein Rechteck.

Es ist somit nicht sinnvoll, Quadrat von Rechteck abzuleiten.

Wie gesagt, “extends” im Sinne von “erweitert” bezeichnet es m.E. viel besser als “isa” ...

... die erweiternde Klasse hat *mehr* Funktionalität (mehr (public) Methoden) als die Ausgangsklasse.

15.4 “Design for inheritance or prohibit it”

Quellen:

- Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley, 2001, Chapter 1, Abschnitt “Putting Reuse Mechanisms to Work”.
- Bloch, J. Effective Java. Second Edition Addison-Wesley, 2008, Item 17 “Design and document for inheritance or else prohibit it”.

Nehmen wir an, ich implementiere

```
public class AClass {
   public void run() {
      aMethod() ;
      bMethod() ;
   }
   public void aMethod() {
      doA() ;
      doB() ;
   }
   public void bMethod() {
      doC() ;
      doD() ;
   }
}
```

Jemand anders implementiere

```
public class MyClass extends AClass {
   @Override
   public void aMethod() {
      doA() ;
      myExtensionMethod() ;
      doB() ;
   }
}
```

Nun kann ich in `AClass` nicht mehr `doC()` von `bMethod()` nach `aMethod()` verschieben,

... ohne das Verhalten der Unterklasse zu verändern (`doC()` nicht mehr aufgerufen).

Und es gibt nirgendwo eine Fehlermeldung!

Copy-and-paste (hier des Inhaltes von `aMethod()`, bevor `myExtensionMethod()` hinzugefügt wird) ist immer eine schlechte Idee.

Ein regression test würde evtl. anschlagen.

Die Verwendung von `super` macht es etwas besser

```
public class MyClass extends AClass {
   @Override
   public void aMethod() {
      super.aMethod() ;
      myExtensionMethod() ;
   }
}
```

Bem.:

- Restriktiver (man kann die zusätzliche Methode nicht mehr zwischen den beiden anderen Methoden einsetzen).

  (Sollte man ohnehin vermeiden.)

- Die/der EntwicklerIn von `AClass` kann den Aufruf von `super` nicht erzwingen.

Daraus folgt:

Design for inheritance or prohibit it.

15.4.1 Design for inheritance ...

Make (public or protected) methods final or empty (incl. abstract).

Dann können nur noch leere (oder noch gar nicht implementierte) Methoden überschrieben werden; damit kann es aber auch kein copy-and-paste des Methodeninhaltes geben.

Und finale Methoden kann man ohnehin nicht überschreiben.

15.4.2 .. or prohibit (public) inheritance

Make classes either final or package-private (= default).

Make methods either final or package-private (= default).

Eigentlich ist der Java default (wenn man private, protected, public komplett weglässt), ziemlich gut.

Man bleibt dann innerhalb von einer Package, die oft nur von einer Person gewartet wird, und dann bleibt das überschaubar.

Leider ist es nicht der Eclipse default.

15.5 Das `protected` Schlüsselwort

Wir hatten bereits folgende Tabelle:

	class	package	subclass	world
private	x			
(default)	x	x		
protected	x	x	x	
public	x	x	x	x

Wenn wir also von package-private (default) ausgehen, dann ist **`protected`** die nächstmögliche Öffnung für Methoden und Variablen (für Klassen geht **`protected`** nicht).

*In Java ist **`protected`** weniger stark geschützt als "default".*

Z.B. Anwendungen, bei denen man eine *interne* Funktionalität ausdifferenzierbar machen will, ohne sie öffentlich zur Verfügung zu stellen.

```
public class AClass {
   public run() {
      methodA() ;
      methodB() ;
      methodC() ;
   }
   protected void methodB() {} ;
   ...
}

public class MyClass extends AClass { // in a different package
   @Override
   protected void methodB() {
      // implement some material
   }
}
```

Damit kann man AClass konfigurierbar machen, ohne die öffentliche Schnittstelle von AClass um methodB erweitern zu müssen.

—

Ich würde eher vorschlagen, die Verwendung von ***protected*** *zu vermeiden.*

15.6 “Favor delegation (≈ composition) over inheritance”

Quellen:

- Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley, 2001, Chapter 1, Abschnitt “Putting Reuse Mechanisms to Work”.
- Bloch, J. Effective Java. Second Edition Addison-Wesley, 2008, Item 16 “Favor composition over inheritance”.

Nehmen wir folgendes an:

```
interface PersonI {
   public int getAge() ;
}
interface ChildI extends PersonI {
   public String getSchool() ;
}
final class Person implements PersonI { ... }
```

Nun wollen wir die Implementierung von `Person` wiederverwenden (re-use), aber wir wollen Vererbung vermeiden, was der Programmierer bereits durch `final class Person` ausgedrückt hat.

Eine gute Alternative ist **Delegation**:

```
final class Child implements ChildI {
   PersonI delegate ;
   Child( ... ) {
      delegate = new Person(...) ;
      ...
   }
   @Override
   public int getAge() {
      return delegate.getAge() ; // delegation!
   }
   @Override
   public String getSchool() {
      // do your ``additional'' implementation here
   }
}
```

Bem:

- Funktioniert auch, wenn die Klassen, von denen man erben will, `final` sind.

  Das wird einem bei modernem Code oft so gehen.

- Funktioniert nicht (oder nur mit Gemurkse), wenn die Klasse, von der man erben will, kein Interface implementiert.

  Das wird einem bei älterem Code oft so gehen.

“Composition”

```
class MyCar implements Engine, Brake, Steering {
   Engine engineDelegate = new EngineType89() ;
   Brake brakeDelegate = new BrakeType123() ;
   Steering steeringDelegate = new SteeringType554() ;
   @Override
   public void accelerate( double val ) {
      engineDelegate.accelerate( val ) ;
   }
   @Override
   public void brake( double val ) {
      brakeDelegate.brake( val ) ;
   }
   @Override
   public void steer( double val ) {
      steeringDelegate.steer( val ) ;
   }
   ...
}
```

Man kann sich komplexe Objekte aus einfacheren Objekten “zusammenbauen” (= composition).

15.7 Effective Java

Go through Chapter 4 Bloch, J. Effective Java. Second Edition Addison-Wesley, 2008.

15.8 Zusammenfassung

Design for ("beyond-package") inheritance (a) or prohibit it (b).

- (a) $\approx$ methods either final or abstract/empty.
- (b) $=$ class and/or methods final

—

Consider replacing beyond-package inheritance by delegation.

15.9 Verdeutlichung: Vererbung und FarmWorld Hausaufgabe

Oben stehen einige OOP Design-Regeln, u.a.:

- "Design for inheritance ..."
  - "Make methods final or abstract/empty."
- "... or prohibit it."
  - "Make classes either final or package-private (= default)." (*)
  - "Make methods either final or package-private (= default)."

Weg (*) konnten Sie mit den Hausaufgaben gehen. Implementation inheritance ist dann problemlos, und in diesem Fall auch sinnvoll (da durch das verwendete Framework nahegelegt).

Wenn die Klasse einmal final oder package-private ist, dann ist es auch kein Problem mehr, wenn einige Methoden non-final und/oder public sind; insbesondere letzteres wird auch durch das GridWorld Framework erzwungen.