Prinzipien Objektorientierter Programmierung

Valerian Wintner

Inhaltsverzeichnis

1 Vorwort

Ich möchte in dieser „Mini-Seminararbeit“ kurz auf die wichtigsten Aspekte der *Objektorientierten Programmierung* eingehen.

2 Kapselung

Programmteile und Daten werden in der *Objektorientierten Programmierung* in Klassen und deren Instanzen (Objekte) gepackt. Der Zugriff auf deren Inhalte kann nach außen hin eingeschränkt werden.
Das hat folgende Vorteile:

- Es kann gewährleistet werden, dass der Objektzustand konsistent bleibt. Wenn Attribute für die Korrektheit des Programmes nur bei Ausführung bestimmter Methoden verändert werden dürfen, können diese Attribute nach außen verborgen werden. Typisch dafür wären Zählvariablen. Ein anderes Beispiel wäre, wenn Attribute nur gemeinsam verändert werden dürfen, oder erst beim Abfragen berechnet werden.
- Der Nutzer muss nur mit den Methoden vertraut sein, die nach außen gestellt werden.

- Die eigentliche Implementierung mit zusätzlichen Funktionen und Attributen wird austauschbar, solange die versprochene Schnittstelle gestellt wird.

```
public class Counter{
  public String name = "Default";
  private int counter;

  public int count(){
    return counter++;
  }
}
```

Abbildung 1: Ein Minimalbeispiel der Kapselung

Abb. 1 zeigt eine Java-Klasse, deren Instanzen das Attribut *name* nach außen hin frei zugänglich machen, aber das Attribut *counter* nur durch die öffentliche Methode *count*() verändert werden kann.

3 Polymorphie

Polymorphie bedeuted *Vielgestaltigkeit*: Die Eigenschaft, Programmteile für mehrere Typen zu verwenden.
Es gibt verschiedene Formen der Polymorphie:

3.1 Dynamische Polymorphie

Zur Laufzeit kann festgelegt werden, welche Implementierung verwendet wird. Das kann genutzt werden, wenn verschiedene Klassen die geforderte Spezifikation stellen und daher untereinander ausgetauscht werden können.
In Java wäre der Typ einer Klasse mit allen Unterklassen, die von dieser erben, austauschbar, da garantiert wird, dass diese wegen der Vererbung die Methoden stellen.

Abb. 2 zeigt Dynamische Polymorphie am Beispiel von Java-Code. Der Variable *variable* können sowohl Instanzen der Klasse *One* als auch Instanzen der Klasse Two zugewiesen werden, da beide die Spezifikation erfüllen und die Methode $foo()$ stellen.

3.2 Statische Polymorphie

Diese Polymorphie findet nicht zur Laufzeit statt, sondern zur Compilierzeit. Es gibt zwei Einteilungen, die hier getroffen werden können:

```
public class Test{
  static interface I{
    public int foo();
  }
  static class One implements I{
    public int foo(){ return 1; }
  }
  static class Two implements I{
    public int foo(){ return 2; }
  }

  public static void main(String[] args){
    I variable;
    variable = new One();
    variable = new Two();
    variabble.foo();
  }
}
```

Abbildung 2: Ein Minimalbeispiel Dynamischer Polymorphie

3.2.1 Parametrische Polymorphie

Parametrische Polymorphie ermöglicht es, Klassen derart zu schreiben, dass bei der Instanziierung oder beim Methodenaufruf der Typ, der verwendet werden soll, angegeben wird. In Java ist das durch die Übergabe eines Typparameters möglich, der beim Compilieren durch Casts ersetzt wird.

```
public class Test{
  static class Para<T>{
    private T attribute;
    public T getAttribute(){ return attribute; }
    public void setAttribute(T attribute){
      this.attribute = attribute;
    }

    public static void main(String[] args){
      Para<Integer> para =
      new Para<Integer>();
      Integer i = para.getAttribute();
    }
  }
```

Abbildung 3: Ein Minimalbeispiel Parametrischer Polymorphie

Abb. 3 zeigt Parametrische Polymorphie in Java anhand einer Klasse, die Typisierung zulässt.

3.2.2 Ad-Hoc-Polymorphie

Diese beschreibt das *Überladen von Methoden*, d.h. die Möglichkeit, Methoden gleich zu benennen, aber im Typ der Argumente zu unterscheiden. Es wird die jeweils passenste Methode beim Aufruf gewählt.

```
public class Test<T>{
  public int foo(){ return 0; }
  public int foo(int arg){ return arg; }
}
```

Abbildung 4: Ein Minimalbeispiel der Ad-Hoc-Polymorphie

Abb. 4 zeigt Ad-Hoc-Polymorphie in Java anhand einer Klasse, die eine Methode überlädt.

4 Vererbung

Vererbung beschreibt im Allgemeinen, dass Programmteile einer Klasse übernommen werden. Vererbung kann auch verhindert werden. Es können mindestens zwei Unterteilungen getroffen werden, zusätzlich zu den Mischformen:

4.1 Vererbung der Spezifikation

Bei der reinen Vererbung der Spezifikation einer Klasse müssen die Programmteile von der erbenden Klasse implementiert werden.
In Java entspricht das der Implementierung eines Interfaces. Siehe Abb. 2.

4.2 Vererbung der Implementierung

Hier wird die Funktionalität der Oberklasse gestellt. Dies ist zumeist der Fall.

Abb. 5 zeigt *Vererbung der Implementierung* in Java. Der Unterklasse wird die Implementierung der Methode der Oberklasse gestellt.

5 Schlusswort

Die *Objektorientierte Programmierung* stellt hochsprachige Konzepte, die es ermöglichen, Programme effizient zu strukturieren. Die Möglichkeiten gehen aber weit über die in dieser Arbeit angestreiften Grundzüge hinaus.

```
public class Test{
  static class Oberklasse{
      public int foo(){ return 0; }
  }
  static class Unterklasse extends Oberklasse{
      public int bar(){ return 1; }
  }

  public static void main(String[] args) {
      Unterklasse u = new Unterklasse();
      u.foo();
      u.bar();
  }
}
```

Abbildung 5: Ein Minimalbeispiel von Vererbung der Implementierung

6 Literaturverzeichnis

Literatur

[1] Prof. Dr. Arnd Poetzsch-Heffter. Konzepte objektorientierter programmierung. Springer-Verlag Berlin Heidelberg, 2009.

[2] Bernhard Lahres und Gregor Rayman. Objektorientierte programmierung. Galileo Computing, 2009.