Einstieg in die Programmierung von Computern – Teil VI

Im Teil 6 behandeln wir das Thema „abstrakte Klassen“ und „Interfaces“ noch etwas genauer, denn diese beiden Programmier-Paradigmen stellen ein wichtiges Hilfsmittel für den objektorientierten Entwurf von komplexen Softwarearchitekturen dar. Ein weiteres Thema werden die sogenannten „Pakete“ zur besseren Sourcecode-Strukturierung kennenlernen.

Konstanten in Interfaces

Außer den abstrakten Methoden können in Interfaces auch **Konstanten** (auch als Membervariablen bezeichnet) mit den Attributen **static** und **final** deklariert sein. Eine Klasse, die dann das Interface mit den definierten Konstanten implementiert, erbt automatisch alle Interface-Konstanten. Ein Interface nur mit Konstanten ist ebenfalls erlaubt. Wann ist so etwas sinnvoll? Ganz einfach: Ein größeres Programm kann somit umfangreiche Konstantendefinitionen in ein separates Interface auslagern. Klassen, die dann diese Konstanten benötigen, brauchen lediglich das entsprechende Interface implementieren und es stehen sofort alle Konstanten zur Verfügung und können ohne vorangestellten Klassennamen aufgerufen werden. Schauen wir uns dazu ein kleines Beispielprogramm an (siehe Kasten 1).

Interfaces als Datentyp

Ein großer Vorteil von Interfaces ist, dass sie als Datentypen verwendet werden können. Das heißt einer Referenz vom Typ eines Interfaces kann als Wert eine Referenz auf ein Objekt zugewiesen werden, dessen Klasse die Schnittstelle implementiert. In unserem Beispiel aus Kasten 1 machen wir dafür im Programm „TestMeldungen.java“ nur eine kleine Änderung:

```
MeldungenInterface2 spezielleMeldung = new SpezielleMeldung();
```

anstatt

```
SpezielleMeldung spezielleMeldung = new SpezielleMeldung();
```

Beachte: Von einem Interface kann kein Objekt mit dem new-Operator erzeugt werden kann. Eine Instanz eines Interfaces kann nur über Zuweisungen erstellt werden, indem ein Objekt einer das Interface implementierenden Klasse einer Referenzvariablen, die das Interface zum Datentyp hat, zugewiesen wird.

Welche Vorteile bringen nun Interfaces?

- Eine vollständige Trennung von Spezifikation und Implementierung
- Verschiedene Implementierungen der gleichen Spezifikation
- Austauschbarkeit ist deutlich einfacher

Die Java-Klassen „ArrayList“ und „LinkedList“ sind gute Beispiele verschiedener Implementierungen, denn beide Klassen implementieren das Interface „List“. Das Interface „List“ stellt eine vollständige Funktionalität einer Liste bereit, jedoch ohne Implementierung. Das überlässt das Interface den Klassen, also hier „ArrayList“ und „LinkedList“. Die beiden Klassen „ArrayList“ und „LinkedList“ stellen verschiedene Implementierungen der Schnittstelle „List“ dar.

Entwurfsmuster mit Interfaces und abstrakten Klassen

Entwurfsmuster *(engl. Design Patterns)* kommen ursprünglich aus der Architektur. In der Softwareentwicklung baut man auch aus vielen Dateien eine Anwendung zusammen. Die Software wird zuerst entworfen (Softwaredesign), d. h. es wird u. a. auch die Softwarearchitektur festgelegt. Dieser Prozess ist sehr entscheidend im Softwareentwicklungsprozess und bestimmt letztendlich auch die Qualität und Performance einer Anwendung. Ein Haus wird ja auch zuerst von einem Architekten entworfen, bevor es letztendlich von Baufachleuten vor Ort gebaut wird. Man kann hier leicht die Ähnlichkeit zur Architektur ausmachen. In der Software sind Sourcecode-Fragmente, auch Code-Schnipsel genannt, die ein Softwareentwickler immer wieder verwendet, bereits Muster bzw. Pattern. Sie sollen die Entwicklung vereinfachen bzw. beschleunigen und man kann leicht die Ähnlichkeit zur Architektur ausmachen. Entwurfsmuster sind dokumentierte und etablierte Konventionen für unterschiedliche Anwendungsfälle. In unserer Artikelserie möchte ich nur das **Brücken-Entwurfsmuster** *(engl. Bridge Design Pattern)* detailliert behandeln, da dieses Entwurfsmuster sehr gut die Entkopplung einer Abstraktion von ihrer Implementierung zeigt. Im Wikipedia steht dazu: Eine **Brücke** *(engl. bridge pattern)* ist in der Softwareentwicklung ein Strukturmuster *(engl. structural pattern)* und dient zur Trennung der Implementierung von ihrer Abstraktion (Schnittstelle), wodurch beide unabhängig voneinander verändert werden können. Eine Brücke findet Anwendung, wenn:

- sowohl Abstraktion als auch Implementierung erweiterbar sein sollen und eine dauerhafte Verbindung zwischen Abstraktion und Implementierung verhindert werden soll,
- Änderungen der Implementierung ohne Auswirkungen für den Klienten sein sollen,
- die Implementierung vor dem Klienten verborgen bleiben soll, oder
- die Implementierung von verschiedenen Klassen gleichzeitig genutzt werden soll.

Schauen wir uns dazu ein konkretes Beispiel an (siehe Kasten 2). Mit dem **Brücken-Entwurfsmuster** kann das Gelernte über „abstrakte Klassen“ und „Interfaces“ sehr gut angewendet und veranschaulicht werden. Nicht zuletzt wird dieses Entwurfsmuster sehr häufig in der professionellen Programmierung eingesetzt, um eine Trennung der Implementierung von ihrer Abstraktion (Schnittstelle) zu erreichen. In den Klassen „EineMeldung“ und „AndereMeldung“ kann man gut erkennen, dass keine Kenntnis über die genaue Implementierung über die Art der Meldung existiert. Das ist genau die Entkopplung. In der Methode „main“ der Klasse „TestMeldungen“ sieht man auch, dass auch zur Laufzeit die Implementierung einfach ausgetauscht werden kann.

```
// Datei: MeldungenInterface1.java
public interface MeldungenInterface1 {

  public static final String MELDUNGSTEXT_1 = "Heute scheint die Sonne.";
  public static final String MELDUNGSTEXT_2 = "Heute regnet es.";
}

// Datei: MeldungenInterface2.java
public interface MeldungenInterface2 {

  public static final String SPEZIELLE_MELDUNG = "Heute schneit es.";
  public oid melden();
}

// Datei: SpezielleMeldung.java
public class SpezielleMeldung implements MeldungenInterface2 {

  // Implementierte Methode aus dem Interface MeldungenInterface2
  public void melden() {

    System.out.println(SPEZIELLE_MELDUNG);
  }
}

// Datei: TestMeldungen.java zum Ausgeben verschiedener Meldungen!
public class TestMeldungen implements MeldungenInterface1 {

  // Methode zum Starten eines Java-Programms.
  public static void main(String[] args) {

    System.out.println(MELDUNGSTEXT_1);
    System.out.println(MELDUNGSTEXT_2);

    SpezielleMeldung spezielleMeldung = new SpezielleMeldung();
    spezielleMeldung.melden();
  }
}
```

Kasten 1: Interfaces mit Konstanten.

```
// Datei: MeldungInterface.java
public interface MeldungInterface {

  public void melden(String meldung);
}

// Datei: Meldung.java
public abstract class Meldung {

  protected MeldungInterface meldung;

  public Meldung(MeldungInterface meldung) {
    this.meldung = meldung;
  }

  public abstract void melden();

  public MeldungInterface getMeldung() {
    return this.meldung;
  }

  public void setMeldung(MeldungInterface meldung) {
    this.meldung = meldung;
  }
}

// Datei: EineMeldung.java
public class EineMeldung extends Meldung {

  public EineMeldung(MeldungInterface meldung) {
    super(meldung);
  }

  public void melden() {
    meldung.melden("Heute scheint die Sonne.");
  }
}

// Datei: AndereMeldung.java
public class AndereMeldung extends Meldung {

  public AndereMeldung(MeldungInterface meldung) {
    super(meldung);
  }

  public void melden() {
    meldung.melden("Heute regnet es.");
  }
}

// Datei: NormaleMeldung.java
public class NormaleMeldung implements MeldungInterface {

  public void melden(String meldung) {
    System.out.println(meldung);
  }
}

// Datei: HTMLMeldung.java
public class HTMLMeldung implements MeldungInterface {

  public void melden(String meldung) {
    System.out.println("<h1>" + meldung + "</h1>");
  }
}

// Datei: TestMeldungen.java zum Ausgeben verschiedener Meldungen!
public class TestMeldungen {

  // Methode zum Starten eines Java-Programms.
  public static void main(String[] args) {

    Meldung meldung;

    MeldungInterface normaleMeldung = new NormaleMeldung();
    MeldungInterface htmlMeldung = new HTMLMeldung();

    meldung = new EineMeldung(normaleMeldung);
    meldung.melden();
    meldung.setMeldung(htmlMeldung); // zur Laufzeit die Meldungsart austauschen
    meldung.melden();

    meldung = new AndereMeldung(normaleMeldung);
    meldung.melden();
    meldung.setMeldung(htmlMeldung); // zur Laufzeit die Meldungsart austauschen
    meldung.melden();
  }
}
```

Kasten 2: Brücken-Entwurfsmuster.

Von der abstrakten Klasse „Meldung“ kann kein Objekt direkt erstellt werden. Erst die Subklassen „EineMeldung“ und „AndereMeldung“, die von der abstrakten Klasse „Meldung“ erben, implementieren die Methode „melden“. Mit der Anweisung `super(meldung);` im Konstruktor der Klasse „EineMeldung“ und „AndereMeldung“ wird der Konstruktor der Vaterklasse, also abstrakte Klasse „Meldung“, aufgerufen.

Pakete

Bislang haben wir alle Java-Dateien in einem einzigen Verzeichnis gespeichert und dort kompiliert. Das ist prinzipiell erst einmal kein Fehler. Man kann sich aber sehr

leicht vorstellen, dass bei einer größeren Anwendung mit vielen Java-Dateien der Überblick verloren geht. Chaotisch wird es dann, wenn in der Anwendung Java-Dateien (Java-Bibliotheken) von anderen Herstellern benötigt und im selben Verzeichnis gespeichert werden. So funktioniert keine professionelle Softwareentwicklung. Die Lösung sind sogenannte „Pakete", die eine hierarchische Strukturierung der Java-Dateien ermöglichen. *Merke:* Ein Java-Paket ist eine logische Gruppierung von Klassen. Pakete können in Hierarchien geordnet werden, sodass in einem Paket wieder ein anderes Paket liegen kann. Pakete stellen die gröbsten Strukturierungseinheiten der objektorientierten Technik dar und werden in der Entwurfsphase erstellt. Man kann Java-Pakete auch gut mit einer Verzeichnisstruktur vergleichen. Der Name des Pakets ist dann gleich dem Namen des Verzeichnisses (und natürlich umgekehrt). Statt des Verzeichnistrenners (etwa »/« oder »\«) steht ein Punkt.

Beispiel:
de/burgardsoft/TestMeldungen.java

Hierbei ist der Paketname „de.burgardsoft" und damit die Verzeichnisstruktur „de/burgardsoft". Sonderzeichen sowie Umlaute sind unbedingt zu vermeiden, da sie auf den unterschiedlichen Dateisystemen oft zu Problemen führen. Außerdem sind Paketnamen immer kleingeschrieben.

Aufbau von Paketnamen
Zwar können die Paketnamen beliebig gewählt werden, doch es haben sich die umgedrehten Domänennamen durchgesetzt. Aus der Website-Domäne „http://www.burgardsoft.de" ergibt sich somit die oben beschriebene Paketstruktur „de.burgardsoft". Man hat somit auch eine Eindeutigkeit geschaffen.

Wie werden nun die Pakete (engl. packages) in der Java-Programmierung verwendet?
Die Klasse „TestMeldungen" wird dazu in das Paket „de.burgardsoft" gesetzt, d. h. die Datei „TestMeldungen.java" muss im Verzeichnis „de/burgardsoft" abgespeichert werden und als erste Anweisung in der Datei „TestMeldungen.java" muss die „package-Deklaration" stehen. Sollte keine package-Deklaration am Anfang stehen, obwohl sich die Datei im Verzeichnis „de/burgardsoft" befindet, kommt es bei der Übersetzung (Kompilierung) zu einem Übersetzungsfehler.

```
package de.burgardsoft; // Deklaration des Paketnamens

public class TestMeldungen {
  …
}
```

Verwendung von Paketen
Bislang haben wir die Annahme getroffen, dass sich alle .java-Dateien in einem Paket, also in einem Verzeichnis, befinden. Das muss aber so nicht sein. In der eigenen Anwendung können sich die Java-Dateien in verschiedenen Paketen befinden. Werden fremde Hersteller-Bibliotheken in der Anwendung verwendet, befinden sich diese Java-Dateien sowieso in anderen Paketen.

Beispiel:

```
// Datei HTMLMeldung.java im Verzeichnis de/burgardsoft/meldungsart
package de.burgardsoft.meldungsart;

public class HTMLMeldung {
  …
}

// Datei TestMeldungen.java im Verzeichnis de/burgardsoft/meldungen
package de.burgardsoft.meldungen;

public class TestMeldungen {
  …
}
```

Möchte man nun in der Klasse „TestMeldungen" die Klasse „HTML Meldung" verwenden, so ist der Punktoperator anzuwenden, also

```
package de.burgardsoft.meldungen;

public class TestMeldungen {

de.burgardsoft.meldungsart.HTHL Meldung = new de.burgardsoft. meldungsart.HTHLMeldung();
}
```

Das ist ziemlich lästig auf Dauer. Zum Glück geht das deutlich einfacher! Wir können die sogenannten **„Import-Anweisungen"** verwenden. Der Code wird dann wesentlich einfacher.

```
package de.burgardsoft.meldungen;

import de.burgardsoft.meldungsart.HTMLMeldung;

public class TestMeldungen {

 HTHLMeldung = new HTHLMeldung();
}
```

Es ist zu beachten, dass die „Import-Anweisungen" immer nach der „Package-Anweisung" stehen müssen. Müssen mehrere Klassen aus einem Paket verwendet werden, geht das mit **„paketname.*"**:

```
import de.burgardsoft.meldungsart.*;
```

Das funktioniert natürlich auch mit nur einer benötigten Datei aus dem fremden Paket. ZT

ZT Adresse

Thomas Burgard Dipl.-Ing. (FH)
Softwareentwicklung & Webdesign
Bavariastraße 18b
80336 München
Tel.: 089 540707-10
Fax: 089 540707-11
info@burgardsoft.de
www.burgardsoft.de