Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

Best Practices of Modern and Efficient Software Engineering

Anonyme und innere Klassen in Java und ihre Verwendungsmuster

Proseminar

von

Birk, Peter Jakobi, Felix

Prüfer: Prof. Dr. B. Rumpe

Betreuer: Dipl.-Inform. Deni Raco

Diese Arbeit wurde vorgelegt am Lehrstuhl für Software Engineering

Aachen, den 23. Januar 2018

Überblick

Dieses Paper stellt alle gängigen Arten von eingebetteten Klassen vor. Nach einer kurzen Einleitung, die einen ersten Überblick ober die verschiedenene Arten gibt, wird ausführlich auf die Implementierung der verschiedenen Arten eingegangen. Außerdem werden noch technische Details konkretisiert. Weiterhin werden Vorteile und Nutzen erläutert. Danach kommt jeweils ein Codebeispiel, in dem eine vorteilhafte Implementierung vorgestellt wird. Zum Ende des Abschnitts werden die eingebetteten Klassen untereinander verglichen. Im vorletzten Abschnitt werden auch noch kurz ähnliche Konzepte in anderen Sprachen beschrieben. Zum Schluss wird noch einmal ein Überblick über das Thema gegeben sowie die wichtigsten Aspekte hervorgehoben.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Idee / Herkunft

Eingebettete Klassen wurden in Java 1.1 eingeführt. [www97] Sie wurden hauptsächlich eingeführt, da die Java-Designer erkannt hatten, dass die Sprache eine Art "Funktionszeiger" benötigt, welcher jedoch mit der Sprachkonstruktion von Java nicht zu realisieren ist. Diese Notwendigkeit kam unter anderem von der Existenz der Funktionszeiger in anderen Programmiersprachen, wie z.B. C oder Lisp. Auch andere Konzepte wie Callbacks oder Adapterklassen lassen sich elegant mit dem Werkzeug der eingebetteten Klassen lösen.

Es gibt insgesamt vier verschiedene Arten von eingebetteten Klassen: Die *inneren Klassen*, die *statischen inneren Klassen*, die *lokalen Klassen* und die *anonymen Klassen*. Diese unterscheiden sich jeweils in der Implementierung, in der Datenkapselung und in den möglichen Funktionalitäten. Aufgrund der semantischen Nähe werden die statischen inneren Klassen sowie die inneren Klassen jedoch zusammen vorgestellt.

Kapitel 2

Eingebettete Klassen

2.1 (Statische) Innere Klassen

2.1.1 Implementierung / Instanziierung

```
1  class OuterClass{
2
3      int b;
4      //wird vom b der RandomClass verdeckt
5
6      class InnerClass extends RandomClass{
7      //kann auch private oder protected sein,
8      //darf nicht OuterClass heissen, darf erben
9
10         static int a;
11         //a wuerde hier einen Fehler hervorrufen,
12         //da die Variable statisch ist
13
14         int b;
15         //b ist erlaubt, verdeckt b der OuterClass/RandomClass
16     }
17
18     static class StaticInnerClass{
19
20         //beide Variablen sind erlaubt
21         static int c;
22         int d;
23     }
24 }
25
26 public class RandomClass{
27     int b;
28     //wird vom b der InnerClass verdeckt
29 }
```

Listing 2.1: Beispielimplementierung von inneren Klassen in Java

Eine innere Klasse wird in Java durch das Erstellen einer Klasse innerhalb eines bestehenden Klassenblocks implementiert. Der Name muss unterschiedlich sein. Dabei kann diese

mit allen Modifikatoren versehen werden, die auch einer äußeren Klasse zur Verfügung stehen - sie darf aber auch *private*, *protected* oder *static* sein, da sie letzten Endes ein Attribut der äußeren Klasse ist. Obwohl klassische Mehrfachvererbung (wie in C++) in Java nicht möglich ist, ist es inneren Klassen erlaubt, von einer weiteren Klasse (ausgenommen ihrer äußeren Klasse) zu erben. Falls es gleichnamige Attribute gibt, wird das Attribut der äußeren Klasse vom Attribut der Oberklasse, von welcher die innere Klasse erbt, verdeckt. Außerdem können in einer inneren Klasse Attribute implementiert werden. Diese verdecken wiederum gleichnamige Attribute der Ober- oder äußeren Klasse. Bei einer nicht-statischen inneren Klasse erlaubt Java keine statischen Attribute, da die innere Klasse an sich ein Attribut der äußeren Klasse ist und sonst ein nicht-statisches Attribut statische Elemente enthalten würde. Falls die Klasse statisch ist, verhält sich diese ähnlich einer normalen Klasse: Es dürfen statische sowie instanzgebundene Attribute verwendet werden, da die Klasse, obwohl sie eigentlich ein statisches Attribut ist, instanziiert werden kann (Listing 2.1).

Instanziierung

Eine normale innere Klasse ist an eine Instanz ihrer äußere Klasse gebunden. Bevor der Konstruktor der inneren Klasse aufgerufen werden kann, muss also zuerst eine Instanz der äußeren Klasse generiert werden. Dann wird der Konstruktor der inneren Klasse von der Instanz der äußeren Klasse aus aufgerufen. Im Gegensatz dazu benötigt eine statische innere Klasse keine Instanz der äußeren Klasse. In diesem Fall darf einfach der Konstruktor der statischen inneren Klasse über den Namensraum der Äußeren Klasse aufgerufen werden. In beiden Fällen werden die Objektdeklarationen über den Namensraum der äußeren Klasse geregelt (Listing 2.2) [GH13].

Dies Erklärt sich durch Speicherung des Bytecodes der Inneren Klassen, welcher nach der Kompilierung als .class Datei im gleichen Ordner wie die Äußere Klasse abgelegt und mit *ÄußereKlasse$InnereKlasse*.class bezeichnet wird, und nicht in der gleichen Datei gespeichert wird.

```
1 //Instanziierung einer inneren Klasse "innen"
2 OuterClass aussen = new OuterClass();
3 OuterClass.InnerClass innen = aussen.new InnerClass();
4
5 //Instanziierung einer statischen inneren Klasse
6 OuterClass.StaticInnerClass statischInnen = new OuterClass.
    StaticInnerClass();
```

Listing 2.2: Beispielinstanziierung von inneren Klassen in Java

Sichtbarkeit & Zugriff

Generell gilt, dass die innere (statische) Klasse je nach Modifier genau wie ein Attribut der äußeren Klasse gesehen wird, was dann implizit auch für alle Attribute der inneren Klasse gilt, falls diese einer weniger starken Modifier benutzen. Auch muss stets der Namensraum der äußeren Klasse benutzt werden, sollte auf Attribute der inneren Klasse zugegriffen werden. Weiterhin hat die innere Klasse im Normalfall Zugang zu allen Methoden und Variablen der äußeren Klasse - sogar zu denen, die als private deklariert wurden. Dies gilt andersherum ebenso: Auch die äußere Klasse kann auf die komplette innere Klasse zugreifen, gleich ob diese privat ist oder nicht. Eine große Unterscheidung muss jedoch gemacht werden wenn eine innere Klasse statisch ist. Da in diesem Fall keine Instanz der äußeren Klasse verbunden ist, können auch lediglich statische Attribute der

äußeren Klasse benutzt werden (Listing 2.3).

```
1  public class OuterClass{
2
3      private int a;
4      //nur fuer InnerClass sichtbar
5
6      private static int b;
7      //fuer InnerClass sowie StaticInnerClass sichtbar
8
9      private class InnerClass{
10     //nur fuer eine Instanz der OuterClass sichtbar,
11     //maximale Kapselung der InnerClass
12     }
13
14     public static StaticInnerClass{
15     //fuer alle sichtbar
16     }
17 }
```

Listing 2.3: Sichtbarkeitsbeispiel für Innere Klassen in Java

2.1.2 Vorteile und Nutzen

Ein Grund für die Benutzung der inneren Klassen ist die Möglichkeit, innerhalb der inneren Klasse auf die privaten Attribute der äußeren Klasse zugreifen und modifizieren zu können. Dies ist insbesonders ein Vorteil für die Datenkapselung, da man innerhalb der inneren Klasse mithilfe von Funktionen der Oberklasse alle Attribute der äußeren Klasse bearbeiten kann, ohne dass die Oberklasse (und somit weitere Klassen) Zugriff auf diese Daten haben können.

Ein Grund dafür ist die Möglichkeit, dass eine innere Klasse, welche implizit die Methoden und Variablen der äußeren Klasse erbt, eine weitere Oberklasse erweitern kann und somit auch deren Variablen und Methoden erbt. Außerdem können Klassen, welche von der inneren Klasse erben (was möglich ist, wenn diese nicht als private, protected oder final markiert sind) über die innere Klasse auf die Attribute und Methoden der äußeren Klasse zugreifen, ohne dass diese direkt von ihr erbt. Dies kann jedoch auch zu Problemen führen.

Es entsteht dadurch die Möglichkeit, dass es durch die Sichtbarkeit der inneren Klasse nach außen zyklischen Vererbungen kommt. Dies bedeutet, dass eine Klasse (indirekt) von sich selbst erbt und somit die eventuell eigene Methoden überschreibt ([Tee13]) (Listing 2.4). Im Beispiel 2.4 wird die zyklische Vererbung nicht erkannt, da die Klasse *innerA* die Klasse *A* nicht erweitert und somit nicht direkt von ihr erbt. Damit kann der Java-Compiler die Tatsache, dass *innerA* durch *A* von *B* erbt, nicht erkennen, und der Code wird fehlerlos kompliiert. Dies kann, wenn es nicht bemerkt wird, zu falschem, und aus der Sicht der Entwickler, sonderbarem Verhalten führen, obwohl die Implementierung semantisch korrekt ist. Um die Möglichkeit zu verringern, dass eine zyklische Vererbung auftreten kann, wurde das sogenannte PICIP (Eine Kurzform für "Potential Inner Class Inheritance Problem") enwickelt, welches das Potential beschreibt, mit dem Vererbungsprobleme bei inneren Klassen auftreten könnten.

```
1  class A  extends B{
2
3      class innerA{
4
5      }
6  }
7
8  class B extends innerA{
9
10 }
```

Listing 2.4: Von Java nicht erkannte Zyklische Vererbung

Weiterhin wird durch Einsatz innerer Klassen die Implementierung einer Adapterklasse erleichtert, welche genau an der Stelle mit den entsprechenden Methoden einer externen Klasse deklariert werden kann.

Man kann durch die Deklaration einer benötigten Klasse als innere Klasse an der entsprechenden Stelle die Lesbarkeit/Übersichtlichkeit des Codes erhöhen. Dadurch ist es jedoch auch möglich, dass der Code durch zu viele bzw. zu tief ineinander verschachtelte innere Klassen unübersichlich und somit unangenehm zu lesen wird. Allgemein gilt daher, dass so wenig innere Klassen wie möglich ineinander geschachtelt werden sollten, um besonders die Fehlererkennung zu verbessern.

Außerdem werden Namenskonflikte mit anderen Klassen im Paket vermieden, da man mehrere Klassen mit gleichem Namen haben kann, die (statischen) inneren Klassen jedoch über ihre äußere Klasse instanziiert werden müssen und es somit nicht zu Verwechslungen und Konflikten mit gleichnamigen Klassen im Paket kommen kann.

2.1.3 Anwendungen / Best Practices

Innere Klassen sollten hauptsächlich dann verwendet werden, wenn eine Klasse ausschließlich von der Oberklasse benutzt wird. Ein weiterer Anwendungsfall wäre die Ausnutzung der Mehrfachvererbung oder die Möglichkeit, neuen Methoden Zugriff auf die privaten Attribute der äußeren Klasse zu erlauben.

Beispiel 2.5: Der Ladeadapter

In diesem simplen Beispiel geht es darum, einen Adapter für den Ladevorgang eines Handys der Marke A zu bauen. Dabei benutzen wir eine Adapterklasse, welche ein Strukturmuster aus der Softwaretechnik ist. Wenn eine Klasse Methoden einer anderen Klasse benötigt, jedoch nur ein Objekt hat, welches ein Interface implementiert, so kann man eine Adapterklasse benutzen. Wir benötigen für das HandyA ein Objekt, welches das Interface *Ladegeraet* implementiert. Wir haben jedoch kein passendes Ladegeraet von A zur Verfuegung, sondern nur ein Ladegeraet einer Marke B, welches das Interface nicht implementiert. Da jedoch auch dieses im Prinzip laden kann, versuchen wir, diese Funktionalität für unser Handy zu benutzen. Dies können wir durch die Einbindung der innere Klasse Adapter erreichen, welche das Interface implementiert. Jedoch steht uns hier auch das Ladegeraet der Marke B zur Verfügung. Dieses instanziieren wir und überschreiben die Methode des Interfaces dadurch, dass wir hier die Lademethode des Handy B benutzen.

Damit können wir eine Instanz des Adapters als Ladegeraet für unser Handy benutzen, obwohl uns kein “korrektes“ Ladegeraet zur Verfügung stand.

```
1  public class HandyLaden{
2      public static void main(String[] args){
3          ALadevorgang lv = new ALadevorgang();
4
5          Ladegeraet l = lv.new Adapter();
6          lv.setLadegeraet(l);
7          lv.aufladen();
8      }
9  }
10 public class HandyA{
11     private Ladegeraet l;
12
13     public Ladegeraet getLadegeraet(){
14         return l;
15     }
16
17     public void setLadegeraet(Ladegeraet l){
18         this.l = l;
19     }
20
21     public void aufladen(){
22         l.laden();
23         System.out.println("Vollstaendig aufgeladen!");
24     }
25
26     class Adapter implements Ladegeraet{
27         LadegeraetB lp = new LadegeraetB();
28         public void laden(){
29             lp.BLaden();
30         }
31     }
32 }
33
34 interface Ladegeraet{
35     public void laden();
36 }
37
38 class LadegeraetB{
39     public void BLaden(){
40
41     }
42 }
```

Listing 2.5: Beispielanwendug von Inneren Klassen in einem Adapter

2.2 Lokale Klassen

2.2.1 Implementierung

Implementierung

Lokale Klassen sind innere Klassen, welche in Methoden eingebunden werden (Listing 2.6). Im Gegensatz zu den inneren Klassen können hier keine zusätzliche Modifikatoren verwendet werden, da die lokale Klasse nicht das Attribut einer Klasse ist. Weiterhin darf sie nicht den Namen der umschließenden Klasse haben. Die lokale Klasse darf jedoch von einer anderen Klasse erben und nicht-statische Attribute haben.

```
1  class Outer{
2      int a;
3      //regulaer benutzbar in MyLocalClass, falls doSomething
           instanzgebunden
4      static int f;
5      //immer benutzbar in MyLocalClass
6      void doSomething(){
7          final int b;
8          //wird von Java akzeptiert, da b final ist
9          int c;
10         //wird akzeptiert, da c effektiv final ist (wird nirgends
               veraendert)
11         int d;
12         //wird nicht akzeptiert, da d potenziell veraendert werden
               kann
13         class MyLocalClass{ //darf nicht "Outer" heissen, keine
               Modifier, darf erben
14             int e;
15             //kann ganz normal ueber die Instanz aufgerufen werden
16             void setExternalD(int n){
17                 //falls diese Methode entfernt wird, ist die
                       Deklaration von d legitim
18                 d = n;
19             }
20             void setExternalA(int n){
21                 //Methode wird akzeptiert, da eine Variable
                       ausserhalb der Methode gesetzt wird
22                 a = n;
23             }
24         }
25         MyLocalClass local = new MyLocalClass();
26     }
27 }
```

Listing 2.6: Beispielimplementierung einer Lokalen Klasse in Java

Sichtbarkeit und Zugriff

Die lokale Klasse an sich ist nur innerhalb der Methode sichtbar. Welche Art von Zugang die lokale Klasse zur umschließenden Klasse hat, hängt vom Zugriffsmodifikator der einschließenden Methode ab, verläuft aber sonst parallel zu den inneren Klassen. Ist die Methode statisch, kann die lokale Klasse somit auch nur auf statische Attribute der umschließenden Klasse zugreifen. Ist die Methode wiederum instanzgebunden, dann hat diese auf alle Attribute der umschließenden Klasse Zugriff. Auf Methodenvariablen selbst hat die Klasse keinen Zugriff, außer diese wurden (effektiv) final deklariert. Falls die lokale Klasse erbt, verdecken gleichnamige Attribute der Oberklasse die der umschließenden Klasse.

2.2.2 Vorteile / Nutzen

Die lokale Klasse kann nur im Bereich ihrer Methode eingesetzt werden und nicht als Oberklasse dienen. Lokale Klassen sollten nur benutzt werden, wenn die neu implementierten Methoden nur in der entsprechenden Methode gebraucht werden. Wenn diese Funktionen jedoch möglicherweise in mehreren Methoden der äußeren Klasse benötigt werden, sollte jedoch eine innere Klasse bzw. “normale“ Klasse benutzt werden.

2.2.3 Anwendung / Best Practices

Beispiel 2.7: Anwendung von lokale Klassen: Studenten und Klausuren

In diesem Beispiel geht es um die Klasse *Student*. Ein Student kann eine gewisse Anzahl an Klausuren schreiben, aber sonst beschäftigt er sich nicht damit. Je nachdem, wie gut er sich vorbereitet hat, kann er die Klausur bestehen oder durchfallen. Jedoch richtet sich das Bestehen zusätzlich noch nach dem Schwierigkeitsgrad der Klausur. Bei guter Vorbereitung besteht der Student immer, sonst nur wenn die Schwierigkeit der Klausur mindestens dem Vorbereitungsgrad entspricht. Hier können wir passend eine lokale Klasse benutzen, da die KLasse *Klausur* sonst nirgendwo genutzt würde, während die Methode *klausurenSchreiben* diese mehrmals instanziieren und benutzen muss. Weiterhin kann so problemlos auf das private Attribut Schwierigkeit zugegriffen werden.

```
1  import java.util.Random;
2
3  public class Student{
4      private Schwierigkeit vorbereitung;
5      public Student(Schwierigkeit v){
6          this.vorbereitung = v;
7      }
8      void klausurenSchreiben(int anzahl){
9          class Klausur{
10             Schwierigkeit s;
11             public Klausur(){
12                 Random rand = new Random();
13                 int r = rand.nextInt(3);
14                 switch(r){
15                     case 0: this.s = Schwierigkeit.LEICHT;
16                             break;
17                     case 1: this.s = Schwierigkeit.MITTEL;
18                             break;
19                     case 2: this.s = Schwierigkeit.SCHWER;
20                             break;
21                 }
22             }
23             void klausurSchreiben(){
24                 if(vorbereitung == Schwierigkeit.SCHWER){
25                     System.out.println("Ich habe bestanden, da meine
                           Vorbereitung gut war.");
26                 }else if(vorbereitung == Schwierigkeit.MITTEL){
27                     if(s != Schwierigkeit.SCHWER){
28                         System.out.println("Ich habe bestanden, weil
                               die Klausur nicht allzu schwer war.");
29                     }else{
30                         System.out.println("Die Klausur war viel zu
                               schwer.");
31                     }
32                 }else{
33                     if(s == Schwierigkeit.LEICHT){
34                         System.out.println("Zum Glueck war die
                               Klausur leicht.");
35                     }else{
36                         System.out.println("Ich habe mich nicht
                               genug vorbereitet.");
37                     }
38                 }
39             }
40         }
41         for(int i = 0; i < anzahl; i++){
42             Klausur klausur = new Klausur();
43             klausur.klausurSchreiben();
44         }
45     }
46 }
47 enum Schwierigkeit{
48     LEICHT, MITTEL, SCHWER
49 }
```

Listing 2.7: Beispiel für die Anwendung von lokalen Klassen

2.3 Anonyme Klassen

2.3.1 Implementierung

Anonyme Klassen

Eine *anonyme Klasse* wird in Java nach Konstruktoraufruf einer beliebigen Klasse definiert (Listing 2.8). Dabei ist es aufgrund der Klassenerweiterung auch möglich, ein Interface zu instanziieren. Dazu wird nach dem Aufruf ein Codeblock mit geschweiften Klammern geöffnet, in dem die instanziierte Klasse beliebig um Methoden und Datenfelder erweitert werden kann, welche nicht statisch sein dürfen. Da die anonyme Klasse von der eigentlich instanziierten Klasse erbt und somit eine Unterklasse dieser ist, können dabei auch Methoden überschrieben werden. Die konkrete Instanz kann jedoch selbst nur auf schon in der Oberklasse deklarierte Methoden und Variablen zugreifen, da das Objekt der anonymen Klasse automatisch zu einem Objekt der Oberklasse gecastet wird, welche jedoch selbst, wenn sie überschrieben werden, auch auf die Methoden der anonymen Klasse Zugreifen dürfen. Da diese eigentlich neue Klasse keinen Namen zugewiesen bekommen hat, wird sie anonym genannt. Aufgrund des einmaligen, instanzgebundenen Auftretens einer anonymen Klasse sind weitere Modifikatoren weder sinnvoll noch möglich ([GH13]) ([www17]).

Sichtbarkeit & Zugriff

Es liegt in der Natur der anonymen Klassen, dass auf sie lediglich über die mit ihr verbundenen Instanz zugegriffen werden kann. Weiterhin kann die anonyme Klasse auch nur mit ihren eigenen Attributen bzw. denen der Oberklasse arbeiten.

2.3.2 Vorteile / Nutzen

Anonyme Klassen werden hauptsächlich und im Idealfall nur benutzt, wenn man eine modifizierte Instanz einer Klasse bzw. eines Interfaces nur einmal gebraucht wird und somit das Übertragen der Modifikationen in eine (innere) Klasse oder Unterklasse nicht sinnvoll wäre. Außerdem werden sie benutzt, um Instanzen von abstrakten Klassen zu erzeugen, da man direkt die abstrakten Methoden überschreiben kann, ohne vorher eine neue Unterklasse der entsprechenden abstrakten Klasse zu deklarieren. Weiterhin hilft die Benutzung von anonymen Klassen der Übersichtlichkeit, Lesbarkeit und Verständnis des Codes, da Veränderungen an den Klassen direkt an den benötigten Stellen durchgeführt werden. Wenn die Modifikationen jedoch in verschiedenen Klassen bzw. Methoden gebraucht werden, sollte jedoch auf eine innere oder lokale Klasse ausgewichen werden, um die Lesbarkeit des Codes zu erhöhen.

```java
public class Main{
    public static void main(String[] args){

        PrintNumbers a = new PrintNumbers(){
            //alle Attribute gelten strenggenommen
            //nur fuer die Methoden der Instanz a

            int y = 20;
            //a.y kann nicht abgerufen werden,
            //jedoch koennen die Methoden y verwenden.

            public void printY(){
                //kann nicht als "a.printY()" aufgerufen werden,
                //muss in eine andere Methode eingebunden werden
                System.out.println(y);
            }

            public void printVariables(){
                //printVariables wird ueberschrieben
                System.out.println(x + " " + y);
                printY();
            }
        };
        //Da printVariables auch innerhalb der Oberklasse existiert,
        //kann sie von der Instanz a aufgerufen werden
        a.printVariables();
        //Ausgabe:"10 20" \n "20"
    }
}
class PrintNumbers{
    //alle Attribute sind in der anonymen Klasse sichtbar

    int x = 10;

    public void printVariables(){
        System.out.println(x);
    }
}
```

Listing 2.8: Beispielimplementierung und -instanziierung einer Anonymen Klasse in Java

2.3.3 Anwendung / Best Practices

Beispiel: Eine einfache GUI mit Inneren / Anonyme Klassen

In diesem einfachen Anwendungsbeispiel soll die Anwendung in einem Graphical User Interface (GUI) modelliert werden, die ereignisgesteuert ist. Es gibt zwei Buttons, die nach Drücken etwas tun sollen. Die Ereignisbehandlung an sich wird hierbei durch Elemente des Abstract Windows Toolkit (aus jawa.awt) geregelt. Um nun die Ereignisbehandlung zu implementieren, brauchen wir eine Klasse, welche das ActionListener-Interface aus dem Toolkit implementiert und die Methode dem Aufrufer entsprechend ändert. Insgesamt benötigen wir also 3 Klassen: Die *Anwendung* (PressButtons), die *GUI* (GUI) und die *Ereignisbehandlung* (MyEventHandler). Dabei wird die Ereignisbehandlung nur von der GUI benutzt, die GUI wiederum nur von der Anwendung. Dabei benutzt die Ereignisbehandlung Methoden, die in der Anwendung aufgerufen werden müssen. Die Klassen sind

also semantisch stark miteinander verbunden und brauchen teilweise vollen Zugriff aufeinander. Aus den Umständen folgt, dass hier eine entsprechende Schachtelung der Klassen von Vorteil wäre, was sich sehr gut mit den inneren Klassen realisieren lässt. Dabei wird die Ereignisbehandlung eine innere Klasse des GUI, die ihrerseits eine innere Klasse der Anwendung wird (Listing 2.9). So lässt sich der Code prägnant an der Stelle formulieren, wo er gebraucht wird. Währenddessen sind die betroffenen Klassen sonst komplett unsichtbar. Doch hier können auch die anonymen Klassen genutzt werden, denn momentan agiert nur eine Instanz des EventHandlers, die je nach Button einen Unterschied macht. Es kann jedoch deutlich eleganter sein, mehrere "Einweginstanzen" zu haben, die dann nur auf einen Button reagieren und dementsprechend nur eine Methode überschreiben müssen. Dadurch fällt der Overhead durch if-Cases weg und man kann noch klarer sehen, was durch das Drücken eines bestimmten Buttons ausgelöst werden soll (Listing 2.10). Weiterhin kann man sich das Schreiben einer weiteren Klasse sparen.

```
1  import javax.swing.JOptionPane;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4  import java.awt.FlowLayout;
5  import java.awt.event.*;
6
7  public class PressButtons{
8      void doSomethingRight(){
9          JOptionPane.showMessageDialog(null, "Du hast auf den rechten
               Button geklickt!");
10         //do something
11     }
12     void doSomethingLeft(){
13         JOptionPane.showMessageDialog(null, "Du hast auf den linken
               Button geklickt!");
14         //do something else
15     }
16     public static void main(String[] args){
17         PressButtons myApp = new PressButtons();
18         PressButtons.GUI myGUI = myApp.new GUI();
19         myGUI.setVisible(true);
20         myGUI.setSize(300, 100);
21         myGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22     }
23     private class GUI extends JFrame{
24         private JButton left;
25         private JButton right;
26         public GUI(){
27             super("Buttons klicken!");
28             setLayout(new FlowLayout());
29             left = new JButton("left");
30             right = new JButton("right");
31             add(left);
32             add(right);
33             MyEventHandler handler = new MyEventHandler();
34             left.addActionListener(handler);
35             right.addActionListener(handler);
36         }
37         private class MyEventHandler implements ActionListener{
38             public void actionPerformed(ActionEvent event){
39                 if(event.getSource() == left){
40                     doSomethingLeft();
41                 }else if(event.getSource() == right){
42                     doSomethingRight();
43                 }
44 }}}}
```

Listing 2.9: Mögliche GUI-Implementierung mit Inneren Klassen

```
1  private class GUI extends JFrame{
2
3      private JButton left;
4      private JButton right;
5
6      public GUI(){
7          super("Buttons klicken!");
8          setLayout(new FlowLayout());
9
10         left = new JButton("left");
11         right = new JButton("right");
12         add(left);
13         add(right);
14
15         left.addActionListener(new ActionListener(){
16             public void actionPerformed(ActionEvent e){
17                 doSomethingLeft();
18             }
19         });
20
21         right.addActionListener(new ActionListener(){
22             public void actionPerformed(ActionEvent e){
23                 doSomethingRight();
24             }
25         });
26 }
```

Listing 2.10: Mögliche GUI-Implementierung mit Anonymen statt Inneren Klassen

Kapitel 3

Verwandte Konzepte in anderen Sprachen

3.1 In C++

Das Konzept der inneren Klassen wurde erstmals 1989 in C++ eingeführt. Diese sind sehr ähnlich zu Javas heutigen statischen inneren Klassen, wobei der größte Unterschied in der Möglichkeit Javas besteht, auf private, für andere Klassen nicht einsehbare Attribute zuzugreifen, was in C++ nicht möglich ist. Also sind Javas innere Klassen eine Art Erweiterung der inneren Klassen von C++. In C++11 wurden die inneren Klassen dann um die Möglichkeit einer "normalen" inneren Klasse Javas, nämlich auch auf nicht-statische Attribute der äußeren Klasse zugreifen zu können, erweitert ([EA07]). Es existiert auch die Möglichkeit, eine Namenlose und somit anonyme Klasse oder *struct* zu erstellen, jedoch ist dies nur in Verbindung mit einem *typedef* möglich. Anonmye Klassen sind in C++ auch noch nützlich, um den Verweis auf einen Klasseninhalt so aussehen zu lassen, als ob das referenzierte Objekt in der gleichen Klasse liegen würde.

3.2 In C#

Derartige Strukturen existieren auch in C#, welche eine von Microsoft produzierte Weiterentwicklungen von C++ ist. Damit folgen diese auch fast den gleichen Regeln wie die inneren Klassen in C++. In C# existiert überdies die Möglichkeit, auf als *protected* oder *privat* deklarierte Attribute zuzugreifen. In C# existiert zusätlich die Möglichkeit ein Namenlose Klasse zu definieren, welche direkt von *Object* erbt. Diese Anonyme Klasse hatt ähnliche Eigenschaften wie die Anonymem Klassen Javas. Sie werden haupsächlich genutz, um "einen Satz schreibgeschützter Eigenschaften in einem Objekt zu kapseln, ohne zuerst explizit einen Typ für diese Daten definieren zu müssen". ([www18b])

3.3 In D

In der Sprache D, welche 1999 ebenfalls als Weiterentwicklung von C++ entwickelt wurde, existiert das gleiche Konzept der inneren Klassen. Es besteht bei als statisch markierten inneren Klassen nur die Möglichkeit auf als statisch deklarierte Attribute der äußeren Klasse zuzugreifen. Jedoch wird nicht unterschieden, ob die Klasse innerhalb einer Methode oder Klasse deklariert wurde. Es existiert zusätzlich noch die Möglichkeit, Anonyme Klassen zu deklarieren, welche sehr große Ähnlichkeiten zu Javas Anoynmen Klassen besitzen.([www18a])

3.4 In Pyton und Ruby

Die Idee des Definierens von Klassen innerhalb anderer Klassen existiert auch in Ruby und Python. Dabei unterscheiden sich die eingebetteten Klassen in diesen Sprachen jedoch maximal durch den Namensraum, über den Sie angesprochen werden können, von den “normalen“ Klassen.

Kapitel 4

Zusammenfassung und Fazit

Eingebettete Klassen in Java kann man grob in 4 Arten charakterisieren. Dabei sind die statischen inneren Klassen das selbstständigste Konzept, sie verbindet hauptsächlich der Namensraum mit einer anderen Klassen, verhalten sich aber sonst wie eine ganz normale Klasse. Eine innere Klasse ist an eine Instanz einer äußeren Klasse gebunden und kann folglich mehr Attribute benutzen. Beide werden vor allem dann benutzt, wenn eine Klasse nur von einer anderen benutzt wird und dementsprechend semantisch verbunden ist. Eine ähnliche Motivation gibt es für lokale Klassen. Auch hier ist es wichtig, dass eine Klasse nur an einer Stelle benutzt wird. Diese befindet sich im Gegensatz zu den inneren Klassen jedoch innerhalb einer Methode, und ist damit nur dort benutzbar. Einen sehr präzisen Einsatz erlaubt das Benutzen von anonymen Klassen, welche das Modifizieren einer Klasse für genau eine Instanz bei Konstruktoraufruf ermöglichen.

Diese Konzepte sind so, wie sie in Java funktionieren, weitergehend als ähnliche Strukturen in anderen Sprachen, da sie teilweise mächtiger sind und mehr Zugriffsrechte besitzen. In der Grundidee sind sie trotzdem ähnlich, geht es doch meist darum, Datenkapselung zu verbessern.

Insgesamt haben die verschiedenen Arten eingebetteter Klassen einige weitere Vorteile, welche in jeweils unterschiedlichen Einsatzbereichen resultieren. Dabei sind vor allem die anonymen Klassen durch Vielseitigkeit und Codeeffizienz interessant. Dennoch muss auch hier betont werden, dass eingebettete Klassen in Java oftmals zwar eine sinnvolle Alternative sind, jedoch ebenso vermieden werden können. Dies zeigt sich auch darin, dass die Java Virtual Machine bei Einführung der eingebetteten Klassen nicht verändert werden musste, da Java auch aus eingebetteten Klassen nach Kompilation normale Klassen macht. Letzten Endes ist dies

Literaturverzeichnis

[EA07] Ellis Ellis and Ellis Margaret A. *The Annotated C++ Reference Manual* -. Pearson Education, Amsterdam, 2007.

[GH13] J. Goll and C. Heinisch. *Java als erste Programmiersprache: Ein professioneller Einstieg in die Objektorientierung mit Java*. Springer Fachmedien Wiesbaden, 2013.

[Tee13] Sim-Hui Tee. Problems of inheritance at java inner class. *CoRR*, abs/1301.6260, 2013.

[www97] JDK 1.1.4 Documentation `https://www.cs.princeton.edu/courses/archive/fall97/cs461/jdkdocs/ind` December 1997.

[www17] Java Language Specification `https://docs.oracle.com/javase/specs/`, December 2017.

[www18a] D Language Specification `https://dlang.org/spec/class.html`, January 2018.

[www18b] Microsoft C# Programming Guide `https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/`, January 2018.