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

1. Prüfer: Prof. Dr. B. Rumpe

2. Prüfer: Dipl.-Inform. Deni Raco

Betreuer: Marlene Lutz

Diese Arbeit wurde vorgelegt am Lehrstuhl für Software Engineering

Aachen, den 20. Dezember 2017

Zusammnefassung

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 Schluss wird noch einmal ein Überblick über das Thema gegeben und der Zusammenhang untereinander und zur konkreten Anwendung hergestellt.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Idee / Herkunft

Innere bzw. 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 Haupsächlich von der Existens von Funktionszeigern in anderen Programmiersprachen, wie z.B. C oder Lisp. Die Möglichkeit, Innere Klassen verschiedenster Art zu benutzen, macht es für die Programmiere einfacher Callbacks und Iteratoren als Adapterklassen zu implemtieren, was vorher umständlich und mit relativ viel aufwand verbunden war. Es gibt insgesamt vier verschiedene Arten von Eingebetteten Klassen, die Inneren Klassen, die Statischen Inneren Klassen, die Lokalen Klassen und die Anonymen Klassen, welche sich jeweils Leicht in der Implementierung, in der Datenkapselung und in den Möglichen Funktionalitäten unterscheiden. Speziell sind Innere- und Statische Innere Klassen nicht sehr unterschiedlich, unterscheiden sich jedoch zusammen relativ stark von den Lokalen Klassen. Die Anonymen Klassen sind eine Allgemeine Unterart der Inneren / Lokalen Klassen, wobei sie sich besonders in der Implementierung von den anderen Eingebetteten Klassen unterscheidet.

Kapitel 2

Hauptteil

2.1 (Statische) Innere Klassen

2.1.1 Implementierung

```
1  //Listing 2.1: Generelle Implementierung
2
3  class OuterClass{
4
5      int b;
6      //wird vom b der RandomClass verdeckt
7
8      class InnerClass extends RandomClass{
9      //kann auch private oder protected sein,
10     //darf nicht OuterClass heissen, darf erben
11
12         static int a;
13         //a wuerde hier einen Fehler hervorrufen,
14         //da die Variable statisch ist
15
16         int b;
17         //b ist erlaubt, verdeckt b der OuterClass/RandomClass
18     }
19
20     static class StaticInnerClass{
21
22         //beide Variablen sind erlaubt
23         static int c;
24         int d;
25     }
26 }
27
28 public class RandomClass{
29     int b;
30     //wird vom b der InnerClass verdeckt
31 }
```

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 Modifiers 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 in Java nicht möglich ist, ist es inneren Klassen erlaubt, von anderen Klassen zu erben. Falls es gleichnamige Attribute gibt, wird das Attribut der äußeren Klasse vom Attribut der Oberklasse 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 eigentlich ein statisches Attribut, 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 aufgerufen werden, jedoch über den Namensraum der äußeren Klasse. In beiden Fällen werden die Objektdeklarationen über den Namensraum der äußeren Klasse geregelt (Listing 2.2) [GH13].

```
1 //Listing 2.2: Instanziierung
2
3 //Instanziierung einer inneren Klasse "innen"
4 OuterClass aussen = new OuterClass();
5 OuterClass.InnerClass innen = aussen.new InnerClass();
6
7 //Instanziierung einer statischen inneren Klasse
8 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  //Listing 2.3: Sichtbarkeit & Zugriff
2
3  public class OuterClass{
4
5      private int a;
6      //nur fuer InnerClass sichtbar
7
8      private static int b;
9      //fuer InnerClass sowie StaticInnerClass sichtbar
10
11     private class InnerClass{
12     //nur fuer eine Instanz der OuterClass sichtbar,
13     //maximale Kapselung der InnerClass
14     }
15
16     public static StaticInnerClass{
17     //fuer alle sichtbar
18     }
19 }
```

Listing 2.3: Sichtbarkeitsbeispiel für Innere Klassen in Java

2.1.2 Vorteile / Nutzen

Ein Grund für die Benutzung der Inneren Klassen ist die Möglichkeit, innerhalb der Inneren Klasse auf die privaten Atribute der äußeren Klasse zugreifen und diese modifizieren zu können. Dies ist insbesonders ein Vorteil für die Datenkapselung, da man innerhalb der Inneren Klasse Mithilfe von Funktionen der Oberklasse alle Atribute 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 Äuseren 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 das diese direkt von ihr erbt. Dies kann jedoch auch, wenn man die Möglichkeit nicht betrachtet, zu Problemen führen.

Dadurch entsteht jedoch die Möglichkeit, dass durch die mögliche Sichtbarkeit der Inneren Klasse nach außen hin es zu Zyklischen Vererbungen kommen kann, also dass eine Klasse (indirekt) von sich selbst erbt und somit die Eigenen Methoden möglicherweise überschreibt ([Tee13]) (Listing 2.4). Im Beispiel 2.4 wird die Zyklische Vererbung nicht erkannt, da die Klasse innerA die Klasse A nicht extended und somit nicht direkt von ihr erbt, ud somit der Java-Compiler die Tatsache, dass innerA durch A von B erbt, nicht erkennt und den Code somit ihne Fehler kompiliert. 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, welche das Potential beschreibt, dass Vererbungsprobleme bei Inneren Klassen aufteten könnte.

```
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

Dies erleichtert die Implementierung einer Adapterklasse, welche genau an der Stelle mit den Entsprechenden Methoden einer Externen Klasse deklariert werden kannn, an welcher sie gebraucht wird, und man nicht notwendigerweise Umwege über Interfaces gehen muss, um die Entsprechende Beziehungen zwischen den Klassen zu bekommen.

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 Möglich, dass der Code durch zu viele bzw. zu tief ineinander verschachtelte Innere Klassen unübersichlich und somit unangenehm zu Lesen wird. Allgemein gilt dazu, dass so wenig 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 (Statische) Innere Klasse jedoch über Ihre Äußere Klasse Instanziiert werden muss 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 einer weiteren benutzt wird.

Beispiel: Der Ladeadapter

In diesem simplen Beispiel geht es darum, einen Adapter für den Ladevorgang eines Handys der Marke Samsong zu bauen. Wir benötigen für den Ladevorgang ein Objekt, welches das Interface Ladegeraet implementiert. Wir haben jedoch kein passendes Ladegeraet von Samsong zur Verfuegung, sondern nur ein Ladegeraet eines EiPhones, welches das Interface nicht implementiert. Das jedoch auch dieses im Prinzip laden kann, versuchen wir, diese Funktionalität für unser Handy zu benutzen. Dies koennen wir durch die Einbindung der innere Klasse Adapter erreichen, welche das Interface implementiert. Jedoch steht uns hier auch das Ladegeraet des EiPhones zur Verfuegung. Dieses instanzieren wir und ueberschreiben die Methode des Interfaces dadurch, dass wir hier die Lademethode des EiPhones benutzen. Damit koennen wir eine Instanz des Adapters als Ladegeraet für unser Handy benutzen, obwohl uns kein rrichtigesLLadegeraet zur Verfuegung stand.

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

Listing 2.5: Beispielanwendug von Inneren Klassen in einem Adapter

2.2 Anonyme & Lokale Klassen

2.2.1 Implementierung

Anonyme Klassen

Implementierung und Instanziierung

Eine anonyme Klasse wird in Java nach Konstruktoraufruf einer beliebigen Klasse / Interfaces definiert (Listing 2.6). 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 Instaziierten 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 deklarierten 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 arbeiten bzw. denen der Oberklasse.

```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();s
        //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 2.6: Beispielimplementierung & -instanziierung einer Anonymen Klasse in Java

Lokale Klassen

Implementierung

Lokale Klassen sind Innere Klassen, welche in Methoden eingebunden werden (Listing 2.7). Im Gegensatz zu den Inneren Klassen können hier keine zusätlichen Modifiers verwendet werden, da die lokale Klasse an sich kein 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.

Sichtbarkeit & 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 Modifier 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 als final deklariert. Falls die lokale Klasse erbt, verdecken gleichnamige Attribute der Oberklasse die der umschließenden Klasse.

```
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 2.7: Beispielimplementierung einer Lokalen Klasse in Java

2.2.2 Vorteile / Nutzen

Anonyme Klassen

Anonyme Klassen werden Haupsätzlich und im Idealfall nur benutzt, wenn man eine modifizierte Instanz einer Klasse bzw. eine Instanz eines Interfaces / eines nur einmal gebraucht wird und somit das übertragen der Modifikationen in eine (Innere) Klasse 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. Somit 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 (Statisch) Innere Klasse ausgewichen werden, um die lesbarkeit des Codes zu erhöhen.

Lokale Klassen

Die Lokale Klasse kann nur im Berreich ihrer Methode eingesetzt werden und nicht als Oberklasse dienen Lokale Klassen sollten nur benutzt werde, wenn die neu Implementierten Methoden nur in der entsprechenden Methoden 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

```
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 2.8: Mögliche GUI-Implementierung mit Inneren Klassen

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 Windows Abstract 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 (ButtonPressing), 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 2.8). 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 Ëinweginstanzenßu 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 das Drücken eines bestimmten Buttons auslösen soll (Listing 2.9).

```
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 2.9: Mögliche GUI-Implementierung mit Anonymen statt Inneren Klassen

Beispiel für lokale Klassen: Kuchenverkostung

In diesem Beispiel soll das Essen eines Kuchens modelliert werden. Dabei gibt es die Klasse Kuchen. Der Kuchen kann je nach seinem Geschmack Feedback bekommen. Dieses Feedback kommt von verschiedenen Menschen, die unterschiedliche Geschmäcker haben. Da die Kuchentester jedoch nur in dieser Methode gebraucht werden, implementieren wir sie auch nur in einer lokalen Klasse in getFeedback.

```
1  import java.util.Random;
2
3  public class Kuchen{
4      private Geschmack g;
5      public Kuchen(Geschmack g){
6          this.g = g;
7      }
8      void getFeedback(){
9          class Mensch{
10             Geschmack g;
11             public Mensch(){
12                 Random rand = new Random();
13                 int r = rand.nextInt(3);
14                 if(r == 0){
15                     this.g = Geschmack.Kuchen;
16                 }else if(r == 1){
17                     this.g = Geschmack.Apfel;
18                 }else{
19                     this.g = Geschmack.Schokoladen;
20                 }
21             }
22             void kuchenEssen(Kuchen k){
23                 if(k.g == g  g == Geschmack.Kuchen){
24                     System.out.println("Mhm, dieser Kuchen mit " + k
                           .g.toString() + "geschmack schmeckt aber gut.
                           ");
25                 }else{
26                     System.out.println("Ich mag keine Kuchen mit " +
                           k.g.toString() + "geschmack.");
27                 }
28             }
29         }
30         Mensch m1 = new Mensch();
31         Mensch m2 = new Mensch();
32         Mensch m3 = new Mensch();
33         Mensch m4 = new Mensch();
34         m1.kuchenEssen(this);
35         m2.kuchenEssen(this);
36         m3.kuchenEssen(this);
37         m4.kuchenEssen(this);
38     }
39 }
40 enum Geschmack{
41     Kuchen, Apfel, Schokoladen
42 }
```

Listing 2.10: Beispiel für die Anwendung von lokalen Klassen

Kapitel 3

Zusammenfassung und Ausblick

Im Algemeinen kann man zusammenfassend sagen, dass Lokale Klassen eher weniger gebraucht werden, Anonyme Klassen aufgrund derer Spezifität häufiger und an verschiedenen Stellen, und Innere Klassen sind in bestimmten Anwendungsbereichen wie GUIs fast Standard, aber sonst auch nicht so vielsetig einsetzbar. Insgesamt werden aber alle Konstrukte in ähnlichen Anwendungsbereichen eingesetzt, und unterscheiden sich somit maximal in der Größenordnungen des Einsatzes. Weiterhin sind die Konzepte aus Gründen der Sichtbarkeit und Ästhetik relevant, in wenigen Fällen ist tatsächlich die Codeefizienz höher, ausgenommen sind in diesem Fall die Anonymen Klassen. Dies kann man auch an der Tatsache erkenne, dass für die Implementierung der eingebetteten Klassen die JVM nicht verändert werden musste, da sich die Eingebetteten Klassen sich soweit nur in der Implemtierung und leicht im Verhalten von den Top-Level Klasse unterscheiden.

Abbildung 3.1: Das SE Logo

Literaturverzeichnis

[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` december 1997.

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