

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

Kurzfassung

Eine kurze Zusammenfassung der Arbeit.

Inhaltsverzeichnis

1	Einführung	1
1.1	Idee / Herkunft	1
2	Hauptteil	3
2.1	(Statische) Innere Klassen	3
2.1.1	Implementierung	3
2.1.2	Vorteile / Nutzen	5
2.1.3	Anwendungen / Best Practices	6
2.2	Anonyme & Lokale Klassen	8
2.2.1	Implementierung	8
2.2.2	Vorteile / Nutzen	11
2.2.3	Anwendung / Best Practices	12
3	Zusammenfassung und Ausblick	15
	Literaturverzeichnis	17

Kapitel 1

Einführung

1.1 Idee / Herkunft

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 wurde 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 äußeren 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: Beispielinstantziierung 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 Attribute der äußeren Klasse zugreifen und diese modifizieren zu können. Dies ist insbesondere 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, 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, und somit der Java-Compiler die Tatsache, dass `innerA` durch `A` von `B` erbt, nicht erkennt und den Code somit ohne 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") entwickelt, welche das Potential beschreibt, dass Vererbungsprobleme bei Inneren Klassen auftreten 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 kann, 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 unübersichtlich 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. Im Folgenden wird ein Codebeispiel mit Codebeispiel präsentiert, in denen das der Fall ist und in denen die vorhin beschriebenen Vorteile eindeutig zur Geltung kommen.

```

1 public class HandyLaden{
2     public static void main(String[] args){
3         SamsungLadevorgang lv = new SamsungLadevorgang();
4         Ladegeraet l = lv.new Adapter();
5         lv.setLadegeraet(l);
6         lv.aufladen();
7     }
8 }
9 public class SamsungLadevorgang{
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 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 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.

```

1 public class Main{
2     public static void main(String[] args){
3
4         PrintNumbers a = new PrintNumbers(){
5             //alle Attribute gelten strenggenommen
6             //nur fuer die Methoden der Instanz a
7
8             int y = 20;
9             //a.y kann nicht abgerufen werden,
10            //jedoch koennen die Methoden y verwenden.
11
12            public void printY(){
13                //kann nicht als "a.printY()" aufgerufen werden,
14                //muss in eine andere Methode eingebunden werden
15                System.out.println(y);
16            }
17
18            public void printVariables(){
19                //printVariables wird ueberschrieben
20                System.out.println(x + " " + y);
21                printY();
22            }
23        };
24        //Da printVariables auch innerhalb der Oberklasse existiert,
25        //kann sie von der Instanz a aufgerufen werden
26        a.printVariables();s
27        //Ausgabe:"10 20" \n "20"
28    }
29 }
30 class PrintNumbers{
31     //alle Attribute sind in der anonymen Klasse sichtbar
32
33     int x = 10;
34
35     public void printVariables(){
36         System.out.println(x);
37     }
38 }

```

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 zusätzlichen 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 hauptsächlich 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 (statische) innere Klasse ausgewichen werden, um die Lesbarkeit des Codes zu erhöhen.

Lokale Klassen

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: Eine einfache GUI

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. 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 `EventHandler`s, 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 das Drücken eines bestimmten Buttons auslösen soll.

```

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
10         Button geklickt!");
11         //do something
12     }
13     void doSomethingLeft(){
14         JOptionPane.showMessageDialog(null, "Du hast auf den linken
15         Button geklickt!");
16         //do something else
17     }
18     public static void main(String[] args){
19         PressButtons myApp = new PressButtons();
20         PressButtons.GUI myGUI = myApp.new GUI();
21         myGUI.setVisible(true);
22         myGUI.setSize(300, 100);
23         myGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24     }
25     private class GUI extends JFrame{
26         private JButton left;
27         private JButton right;
28         public GUI(){
29             super("Buttons klicken!");
30             setLayout(new FlowLayout());
31             left = new JButton("left");
32             right = new JButton("right");
33             add(left);
34             add(right);
35             MyEventHandler handler = new MyEventHandler();
36             left.addActionListener(handler);
37             right.addActionListener(handler);
38         }
39         private class MyEventHandler implements ActionListener{
40             public void actionPerformed(ActionEvent event){
41                 if(event.getSource() == left){
42                     doSomethingLeft();
43                 }else if(event.getSource() == right){
44                     doSomethingRight();
45                 }
46             }
47         }
48     }
49 }

```

Listing 2.8: 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 2.9: Mögliche GUI-Implementierung mit Anonymen statt Inneren Klassen

Kapitel 3

Zusammenfassung und Ausblick



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