1 Entwurf Vortrag Skript

1.1 Einleitende Worte

Design Patterns sind in aller Munde. Ich garantiere euch zu hundert Prozent dass ihr sie im Bewerbungsgespräch in irgend einer Art und Weiße werdet runtereiern können müssen. Die Frage ist warum. Was sind design Patterns? WARUM sind Design Patterns?

Erfunden wurde das Konzept von vier Informatikern, die sogenannte gang of four (originell). Der komplette Titel ihres Buches hieß "Design patterns - elements of reusable object-oriented software". Für uns interessant ist hierbei der Untertitel. Es bezieht sich also grundsätzlich mal auf Objekt-orientierte Software. Also müssen wir uns zunächst darauf einigen was genau DAS nun ist.

1.1.1 Entwurfsmuster sind auch nicht so geil

Zunächst möchte ich auf folgendes hinweisen: Keines der Konzepte, die ich nun besprechen werde ist (meiner Meinung nach) sonderlich tiefgründig. Informatik ist ein junges Fachgebiet und obwohl wir den Vorteil haben, auf einige Jahrhunderte Mathematik zugreifen zu können sind wir weit davon entfernt unsere Version des Grundsatzes der Analysis oder dergleichen zu haben. Ich denke jeder von uns hätte nach ausreichender Zeit auch selbst auf die meisten dieser Pattern kommen können.

Es wird oft über Design Patterns geredet als wären sie in theoretischer Bedeutung auf der Höhe von Mathematischen Definitionen. Dies ist weit von der Wahrheit entfernt. Tatsächlich sind es einfach Lösungen auf Probleme, welche oft in Objekt orientierten Entwürfen aufkommen. In der Praxis ist es viel wichtiger, die unterliegenden Ideen zu verstehen als hirnlos Pattern zu implementieren.

1.1.2 OO ist auch nicht so geil

Objekt- oder Klassenorientierte Programmierung ist schwer in einem Satz zu definieren. Falls ihr jemals JAVA geschrieben habt wisst ihr was es ist - in Java kann man nichtmals eine Funktion deklarieren ohne dass sie in einer Klasse ist. Wie alle Extreme kommt dies mit einigen Einschränkungen. Meiner Meinung nach existieren viele Entwufsmuster nur, um diese Einschränkungen zu umgehen. Ein guter Informatiker sollte mit vielen Programmierparadigmen vertraut sein. Ansonsten seid ihr wie ein Elektriker der außer Seitenschneidern keine Zangen verwendet.

Der Grund aus welchem OO so beliebt ist hat damit zu tun dass es einen guten sales pitch hat, vor allem für Anfänger: "Hey, wir wissen programmieren ist schwer aber schau doch mal hier das: Ein Mercedes ist ein Auto und hat Räder“ und der anfänger denkt "Hey

voll gut ich kenne einige dieser Begriffe“ und man modelliert dann Auto als Hauptklasse, Mercedes erbt von Auto und bekommt als Felder vier Variablen vom Typ Rad und BÄM man programmiert. Dieser naive Ansatz hat das Problem, dass man sehr schnell sehr schlechte Entwürfe schreibt welche schwer zu ändern und langsam sind. Ironischerweise funktionieren viele der Design Patterns gerade daher so gut, dass sie diese naheliegenden OO-Entwürfe (und damit den vermeintlichen Vorteil von OO) komplett ignorieren.

1.1.3 Aber beide sind um einiges geiler als nichts

Ich sprach bisher sehr negativ über Entwurfsmuster und OO. Tatsächlich muss ich sagen, dass ein Informatiker der beide kennt um Mengen produktiver ist als einer, welcher sie nicht kennt. Ich werde jedoch behaupten dass dies hauptsächlich daran liegt dass mit dem Behandeln der Materie miteingeht, sich überhaupt einmal mit dem Thema Entwurf zu befassen und nicht einfach dumm draufzuprogrammieren.

1.2 Also was sind Design Patterns

1.2.1 Grundlegende Ideen

Es gibt einige Ideen welche allen wichtigen Design Pattern unterliegen. Ich werde diese nun beschreiben und anhand von Beispielen verdeutlichen.

One more layer of indirection

Ein kluger Mann sagte einmal “All problems in computer science can be solved by another level of indirection”. Sollte man mal gehört haben. Das Nächste Konzept, Polymorphie ist ein gutes Beispiel dafür. Anstatt die Funktion direkt in der Klasse zu speichern speichert man stattdessen einen Pointer auf die korrekte Implementierung der Funktion.

Polymorphie

Polymorphie klingt kompliziert und ist extrem einfach. Zumindest in der Praxis. In der Theorie kann man zeug sagen wie ”...ist ein Konzept in der objektorientierten Programmierung, das ermöglicht, dass ein Bezeichner abhängig von seiner Verwendung unterschiedliche Datentypen annimmt”. Bei Java ist es im Grunde lediglich: Interfaces und/oder abstrakte Methoden.

Denkt mal drüber nach was ein Interface eigentlich ist: Ihr definiert welche Funktionen mit welchen Rückgabewerten eine Klasse zur Verfügung stellen muss. Und das ist ALLES. Die Klassen, welche das Verhalten schlussendlich implementieren müssen nichts miteinander zu tun haben. Abstrakte Methoden sind ähnlich. Man definiert in einer Oberklasse eine Methode `public abstract void meth()` und überschreibt diese in der Unterklasse. Nun kann man zum Beispiel eine Liste vom Typ List<Oberklasse> haben und diese mit Instanzen der Unterklasse füllen. Sagt man nun etwas wie `for(Oberklasse c : liste) c.meth();`, dann wird für jedes Element dieser Liste automatisch zur Laufzeit die korrekte Version der Methode, nämlich die der Unterklasse aufgerufen.

Beispiel: Das Command Pattern.
Das Command Pattern ist eine Möglichkeit eine Rückgängig Funktionalität zu implementieren. Dabei werden Eingaben des Benutzers in Objekten festgehalten, welche dann gespeichert werden können.

Man nimmt ein Interface, zum Beispiel

```
Interface Command {
    void perform();
    void undo();
}
```

Dann kann man sich beliebig viele Aktionen Ausdenken

```
class AddWordCommand implements Command {
    private String word;
    private TextPosition pos;

    public void perform() {
        pos.add(word);
    }

    public void undo() {
        pos.remove(word);
    }
}

class AddPropertyToListCommand implements Command {
    private Property prop;

    public void perform() {
        list.add(prop);
    }

    public void undo() [
        list.remove(prop);
    }
}
```

Dann hat man noch irgendwo zwei Listen, eine mit gerade vollbrachten und eine mit gerade rückgängig gemachten Commands. Rest ist relativ simpel. Der Teufel liegt natürlich im Detail.

Bündeln von Funktionalität

Funktionalität, welche sich oft gemeinsam verändert, sollte an einem Ort sein. Oft sagt man auch: DRY - dont repeat yourself. Im Grunde gilt: sobald ihr größere Mengen Code kopiert wird es Zeit die gemeinsame Funktionalität zu extrahieren.
Beispiel: Das Policy Pattern
Sagen wir, wir haben eine Hauptklasse, zum Beispiel Employee. Diese hat ein zwei abstrakte Funktionen, pay() und fire(), welche von Unterklassen überschrieben werden sollen. Nun kreieren wir 50 Unterklassen. Viele der Unterklassen überschreiben diese Funktionen auf dieselbe Art und Weiße. das Problem: Verändert sich das Verhalten für eine der Klassen, so muss die Funktion in allen Klassen aktualisiert werden.

Also tut man folgendes. Man definiert zwei Interfaces

```
Interface FireAble() {
   void fire(Employee e);
}

interface PayAble {
    void pay(Employee e);
}
```

Nun nimmt man die verschiedenen Implementationen der Funktionen aus den einzelnen Unterklassen von Employee heraus und gibt jeder eine eigene Klasse, welche das Interface implementiert.

```
class FireNormally : FireAble {...}
class FireHard : FireAble {...}
class PayGood : PayAble {...}
class PayBad : PayAble {...}
```

Und dann gibt man den einzelnen Unterklassen die entsprechende Klasse im Konstruktor mit. Und dann ruft DIESE Klasse wiederrum die Implementierung in der anderen Klasse auf. Verwirrend?

```
class Programmer extends Employee {
    private PayAble payHandler;
    private FireAble fireHandler;

public Programmer(PayAble payHandler, FireAble fireHandler) {
    this.payHandler = payHandler;
    this.fireHandler = fireHandler;
}

public void pay() {
    this.payHandler.pay(this);
}
```

```
public void fire() {
    this.fireHandler.fire(this);
}
}
```

Nicht so komplex. Wenn man jetzt eine gewisse Implementierung verändern will, ist alles dazu schön in einer Klasse aufbewahrt.

Betrachtet man das ganze so fällt nach einer Weile auf, dass im Grunde eine Funktion infusiert wird. Man könnte nun auch im laufenden Programm das Verhalten eines Employee ändern, indem man eine andere Implementierung von PayAble oder FireAble einsetzt. Dies wäre zum Beispiel bei funktionaler Programmierung um einiges einfacher.

Vermeidung von ifs

Oftmals ist es berechtigt und unproblematisch ifs zu verwenden. Allerdings gibt es viele Fälle, in denen dies zu unübersichtlichem Code führen kann.
Beispiel: Sate 'splosion
Sagen wir wir wollen dem C-Editor kontrollieren. Wenn der Benutzer entfernen klickt, soll abhängig davon ob etwas markiert ist oder nicht ein Löschen-Command erstellt und ausgeführt werden. Eine mögliche Implementierung wäre:

```
if(textPane.isSomethingSelected()) {
   RemoveCommand cmd = new Removecommand(textPane.getSelected());
   commandList.add(cmd);
} else {
 //do nothing or maybe somethind else who knows}
```

Des ist soweit alles noch kein Problem. Aber es besteht die Möglichkeit, dass noch zehn andere Eigenschaften existieren welche wir überprüfen müssen. Sobald man mehr als zwei ifs geschachtelt hat sollte man sich Gedanken machen.

Eine Möglichkeit das Problem zu lösen wäre die Verwendung von States:

```
Interface State {
....
    void handleDelete();
}

SomeTextSelectedState implements State {
    void handleDelete() {
        // push back delete command
    }
}

NoTextSelectedState implements State {
```

```
    void handleDelete() {
        //do nothing
    }
}
```

Klickt der User nun delete, so lässt man einfach den momentanen state die sache regeln. Und damit - Puff - keinerlei ifs mehr. Magic! Dies ist auch ein gutes Beispiel für die erste grundlegende Idee: alles was man tut ist eine Schicht Indirektion (die states) einzuführen.