JUST DO IT
HÖHERE ABTEILUNG FÜR
INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Protokoll Softwareentwicklung

Design Patterns

SEW

5bHITT 2016/17

Philipp Kogler

Begonnen am 26. Dezember 2016

Betreuer: Prof. Rafeiner, Prof. Dolezal

Beendet am 26. Dezember 2016

Inhaltsverzeichnis

1 Einführung

Diese Übung soll einen Einblick in die zahlreichen Design Patterns der Softwareentwicklung geben. Sowie die dementsprechend meist verbreitetsten kurz beschreiben und mit Beispielen erläutern.

1.1 Ziele

Das Ziel dieser Ausarbeitung ist ein allgemeines Verständnis für die wichtigsten DesignPatterns. Sowhol in der Verwendung als auch im theoretischen Verständnis soll es möglich sein diese umzusetzen.

1.2 Voraussetzungen

- Grundlagen in Java & Python
- Grundlagen UML

1.3 Aufgabenstellung

Folgende Punkte müssen in der Ausarbeitung / dem Protokoll enthalten sein.

- UML-Klassendiagramm der verwendeten Architektur inkl. Beschreibung
- Kurze allgemeine Ausarbeitung zu Design Patterns
  - ✓ Wie können Design Patterns unterteilt werden
  - ✓ Wozu Design Patterns
  - ✓ Übersicht existierender Design Patterns
- Ausarbeitung zum Decorator Pattern
  - ✓ Allgemeines Klassendiagramm
  - ✓ Grundzüge des Design Patterns (wichtige Operationen etc.) am Beispiel des implementierten Programms inkl. spezielles Klassendiagramm
  - ✓ Vor- und Nachteile
- Ausarbeitung zu einem der folgenden Design Patterns: Observer, Abstract Factory, Strategy
  - ✓ siehe oben -> Decorator Pattern

2 Architektur - A09 Server-Client-Chat

2.1 Aufgabenstellung

Verwende das Decorator Pattern, um die Socket-Kommunikation zwischen einem einfachen Server und einem simplen Client (Konsole genügt) zu dekorieren! Die Plaintext-Kommunikation kann z.B. mit einer RSA- oder AES-Verschlüsselung, einer BASE64-Codierung, einem Hashwert / Fingerprint dekoriert werden.

Zeige im Code, dass die unterschiedlichen Dekorierer miteinander kombiniert werden können!

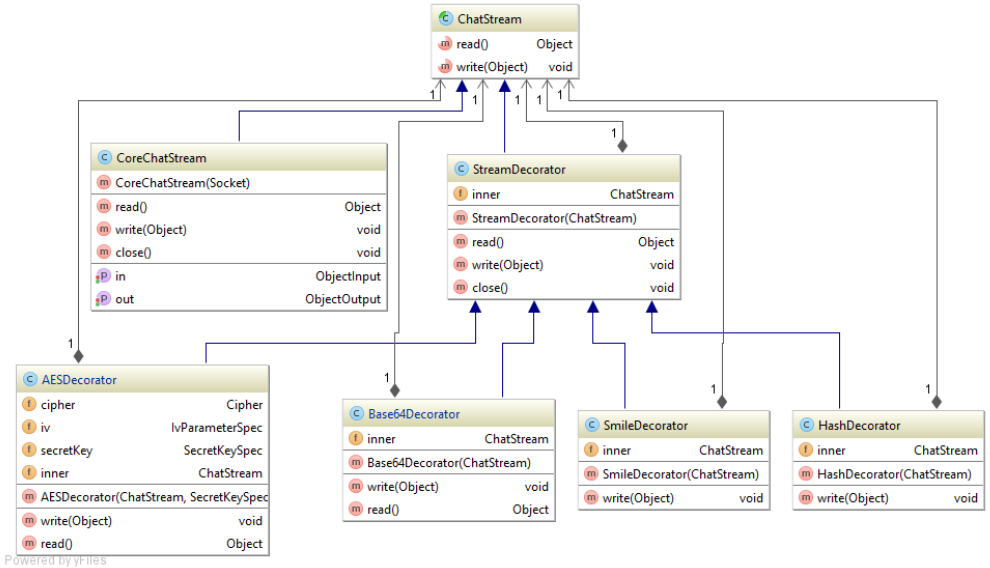
2.2 Umsetzung

Für die Umsetzung wurde wie vorausgesetzt ein Decorator Pattern verwendet. Hierbei wird ein angepasster Stream dekoriert. Dieser wird mit einem Socket ausgestattet und dekoriert somit immer die `write` bzw. die `read` Methode.

Somit kann bsp. ein `AESDecorator` erstellt werden, welcher in der `write` Methode mithilfe eines *Block - Ciphers* den Text verschlüsselt und anschließend in der zugehörigen *read* Methode wieder entschlüsselt.

2.2.1 UML - Klassendiagramm

Das Decorator Pattern wurde nach folgender Struktur umgesetzt.



ChatStream - Komponente

Zu dekorierende Komponente.
Hier wird eine abstrakte Komponente als dekorierende Klasse verwendet, da ein spezielles Interface implementiert werden muss. Die Komponente bietet zwei abstrakte Methoden welche dekoriert werden.

- **`public Object read()`**
  Daten von dem *ObjectInputStream* lesen und dementsprechend an den Aufrufer zurückliefern.
- **`public void write(Object o)`**
  Ein bestimmtes Objekt mithilfe des *ObjectOutputStream* schreiben.

CoreChatStream - Konkrete Komponente

Konkrete Implementierung des *ChatStreams* / Komponente.
Diese Klasse extended von der zu dekorierenden Komponente und muss aufgrund der abstrakten Methoden diese auch überschreiben und mit Funktionalität füllen. Wie der Name schon beschreibt ist dies die nötigste Grundfunktionalität. In diesem Fall wird ausschließlich ein *Object* gelesen bzw. mit `read` geschrieben.

```java
private ObjectOutput out;
private ObjectInput in;

/**
* {@link CoreChatStream} Constructor
*
* @param socket
* @throws IOException
*/
public CoreChatStream(Socket socket) throws IOException {
    this.out = new ObjectOutputStream(socket.getOutputStream());
    this.in = new ObjectInputStream(socket.getInputStream());
}
```

Listing 1: Konstruktor and in / out Stream CoreChatStream

```java
/**
 * Read from an InputStream
 *
 * @return @{@link Object} read Object
 * @throws IOException
 */
@Override
public Object read() throws IOException {
    return this.in.readObject();
}
```

Listing 2: read - CoreChatStream

```java
/**
* Write to an OutputStream
*
* @param o @{@link Object} Object to write
* @throws IOException
*/
@Override
public void write(Object o) throws IOException {
    this.out.writeObject(o);
}
```

Listing 3: write - CoreChatStream

StreamDecorator - Decorator Komponente

Diese Klasse erbt ebenfalls von der zu dekorierenden *ChatStream* Klasse.
Zusätzlich wird ein Attribut meist `inner` gennant, erstellt welches von dem gleichen Type wie der Decorator Komponente ist in diesem Fall *ChatStream*. Dieses Attribut ist sozusagen der innere ChatStream. Sobald nun ein ChatStream mittels dem Decorator dekoriert wird kann mit dem Attribute `inner`, die inneren zu dekorierenden Methoden aufgerufen werden und somit rekursiv nach innen durchzuarbeiten.

```java
/**
* Inner ChatStream which will be further decorated
*/
private ChatStream inner;

/**
* {@link StreamDecorator} Constructor
* @param inner
*/
public StreamDecorator(ChatStream inner) {
    this.inner = inner;
}
```

Listing 4: Konstruktor and inner StreamDecorator

```java
/**
* @see ChatStream
*
* @return
* @throws IOException
*/
@Override
public Object read() throws IOException {
    return this.inner.read();
}
```

Listing 5: read - StreamDecorator

```java
/**
* @see ChatStream
*
* @param o @{@link Object} Object to write
* @throws IOException
*/
@Override
public void write(Object o) throws IOException {
    this.inner.write(o);
}
```

Listing 6: write - StreamDecorator

Zudem muss, nachdem der ganze ChatStream *AutoClosable* implementiert, eine `close` Methode existieren, welche alle inneren Streams regelkonform schließt und auf eventuelle Fehler reagiert.

```java
/**
* Closes this resource, relinquishing any underlying resources.
* This method is invoked automatically on objects managed by the
* {@code try}-with-resources statement.
*/
@Override
public void close() throws Exception {
    this.inner.close();
}
```

Listing 7: close StreamDecorator

AES/Base64... - Decorator - Die Decorator Komponenten

Der wichtigste Zweck dieser Klassen ist, die abstrakten Methoden in diesem Fall `read` & `write` zu implementieren und zu ”dekorieren”.
Diese Klassen erben von der DekoratorKomponente *StreamDecorator* und hat somit Zugriff auf den inneren *ChatStream* und somit letztendlich auf den *CoreChatStream*. Somit kann das banale `read` und `write` von der *CoreChatStream* Klasse erweitert oder gar komplett geändert werden.
Beispiel eines Dekorators: *Base64Decorator*

```java
/**
 * @return
 * @throws IOException
 * @throws ClassNotFoundException
 * @see ChatStream
 */
@Override
public Object read() throws IOException {
    Message<String> a = (Message) super.read();
    byte[] msg = a.getMessage().getBytes();
    byte[] decBytes = Base64.getDec().decode(msg);
    System.out.print("base64:" + a.getMessage());
    a.setMessage(new String(decBytes));
    return a;
}
```

Listing 8: read - Base64Decorator

```java
/**
 * @param o @{@link Object} Object to write
 * @throws IOException
 * @see ChatStream
 */
@Override
public void write(Object o) throws IOException {
    byte[] msg = ((Message) o).getMessage();
    byte[] encMsg = Base64.getEnc().encode(msg);
    //System.out.println("base64 encoded Message:
        " + new String(encodedMsg));
    Message<String> old = (Message<String>)o;
    old.setMessage(new String(encMsg));
    this.inner.write(old);
}
```

Listing 9: write - Base64Decorator

Mithilfe des Attributs *inner* kann nun auf die inneren Methoden zugegriffen werden.
Das ermöglicht nun das **Decorator Pattern**. Hier wird in den Methoden `write` & `read` ein Base64 end/dec String erzeugt und dementsprechend mit `inner/super.read` & `inner/super.write` auf den *Input/Output - Stream* geschrieben.

Aufruf / Erstellen eines Stream-Decorators

Hierfür ist vorallem die Reihenfolge wichtig, da ansonsten beim lesen, vor allem beim entschlüsseln und decoding Fehler auftretet können.

```java
try (
    ChatStream stream =
        new Base64Decorator(
            new AESDecorator(
                new CoreChatStream(socket),
                skeySpec,
                iv
        )
    )
) {
```

Listing 10: erstellen / Aufrufen eines Decorators

3 Design - Patterns Allgemein

Es gibt bereits hunderte verschiedene Design-Patterns für alle Anwendungsfälle und Programmiersprachen. Allgemein kann man jedoch den Sinn und Zweck sowie die Hintergrundgedanken als auch Vor & Nachteile anhand dieser Beispiele gut zusammenfassen und erklären. [1]

Prinzipiell werden Design-Patterns verwendet um Programmcode wiederverwendbar und vor allem übersichtlich zu gestalten. Dafür gibt es zahlreiche Vorlagen und Templates. Im Prinzip gibt es zahllose Baupläne für ein passenden Software Design. Die Schwirigkeit hierbei liegt nun das **richtige** Pattern auszwählen und anzuwenden. [1]

Des Weiteren bieten bereits **getestete** Design-Patterns ein zuverlässiges Programmierparadigma und somit können Fehler und Gedankenfehler vorgebeugt werden. Bereits bewiesene Patterns werden mit großer Wahrscheinlichkeit auch für den eigenen Anwendungsfall genügen.

3.1 Klassifizierung Design-Patterns [1]

Prinzipiell können alle Arten von Design-Patterns in 3 Arten unterteilt werden.

- **Creational Patterns - [Erzeugungsmuster]**
  Dienen der Erzeugung von Objekten. Sie entkoppeln die Konstruktion eines Objekts von seiner Repräsentation. Die Objekterzeugung wird gekapselt und ausgelagert, um den Kontext der Objekterzeugung unabhängig von der konkreten Implementierung zu halten, gemäß der Regel: „Programmiere auf die Schnittstelle, nicht auf die Implementierung!“
- **Structural Patterns - [Strukturmuster]**
  Erleichtern den Entwurf von Software durch vorgefertigte Schablonen für Beziehungen zwischen Klassen.
- **Behavioral Patterns - [Verhaltensmuster]**
  Modellieren komplexes Verhalten der Software und erhöhen damit die Flexibilität der Software hinsichtlich ihres Verhaltens.

Zudem kamen später noch weitere Typen für Entwurfsmuster dazu, welche zu keinen der 3 bereits genannten Typen dazu passte. Beispielsweise für Muster für Objekt rationale Abbildungen. Diese dienen der Ablage und dem Zugriff von Objekten und deren Beziehungen in einer relationalen Datenbank

3.2 Warum werden Deisgn-Patterns verwendet ?

Wie bereits angeschnitten sollen Entwurfsmuster uns dabei helfen unser Design flexibler und eleganter zu gestalten, und somit die Software wiederverwendbar machen. Aber warum sollten wir mehr Aufwand bei der Entwicklung treiben, um in der Software die eben genannten Eigenschaften zu realisieren?

Im wesentliche verfolgt die Wiederverwendung von Software 3 wichtigen Zielen.

- ✓ **Reduzierung des Entwicklungsaufwandes**
  Da bereits die schwierige Arbeit des tüftelns getan wurde, muss das Design-Pattern lediglich noch angewendet werden.[3]
- ✓ **Erreichung einer Qualitätsverbesserung**
  Da diese Patterns sich offenbar schon bewahrt haben kann man auf jeden Fall davon ausgehen, dass diese auch ”optimal” umgesetzt sind. Zudem kommt, dass man durch diese Muster gezwungen ist, sich nach diesem Bauplan zu halten, welches selbstverständlich auch die insgesamte Qualität erhöhen kann.
- ✓ **Einheitliche Wartbarkeit**
  Nachdem die Qualität gegeben ist, ist ebenfalls ein einheitlicher Code gegeben. Dieser ist natürlich, wenn mehrere Entwickler an einem Programm arbeiten, um einiges einfacher zu warten. Auch wenn der Code lang funktionieren soll und deshalb des öfteren erweitert und erneuert werden soll, können dies Design-Patterns erleichtert. [3]

Hierbei ist jedoch zu sagen, dass zuerst ein größerer Aufwand nötig ist um das richtige Design auszuwählen und anzuwenden.

Zuerst sollte man sich im klaren sein welche verschiedenen Arten von Design-Patterns verwendet werden und anschließend verschiedene Muster ausarbeiten und dementsprechend umsetzten.[3]

Es muss nicht immer nur ein Entwurfsmuster verwendet werden. Diese können auch kombiniert werden um sein Ziel zu erreichen und alle Aufgabenstellungen zu erfüllen.[3]

3.3 Übersicht aktueller Design-Patterns

Übersicht aktuell verwendeter Design-Patterns nach ihrer Klassifizierung.

3.3.1 Creational - Patterns

Name	Beschreibung
Abstract Factory	Es wird ein Interface zur Verfügung gestellt, um verbundene Objekte zu Erstellen. Ohne Konkrete Klassen zu spezifizieren
Singelton	Es gibt nur eine Instanz dieser Klasse
Factory Method	Mithilfe von dependency Injection kann die subklasse entscheiden wie die methode zu implementieren ist.
Multiton	Im Gegensatz zu Singelton. Versichern, dass eine Klasse einen globalen Zugriffspunkt für seine Instanzen hat.

Tabelle 1: Creational Patterns Übersicht [1]

3.3.2 Structural - Patterns

Name	Beschreibung
Decorator	Dekorieren einer Klasse bzw. von Methoden. Dynamisches anpassen von Funktionalität.
Composite	Objekte in einer Baumstruktur. Partitioning Design-Pattern. Gruppe in gleicher Art behandelt
Marker	Leeres Interfaces um Metadaten zuzuweisen

Tabelle 2: Structual Patterns Übersicht [1]

3.3.3 Behavioral patterns

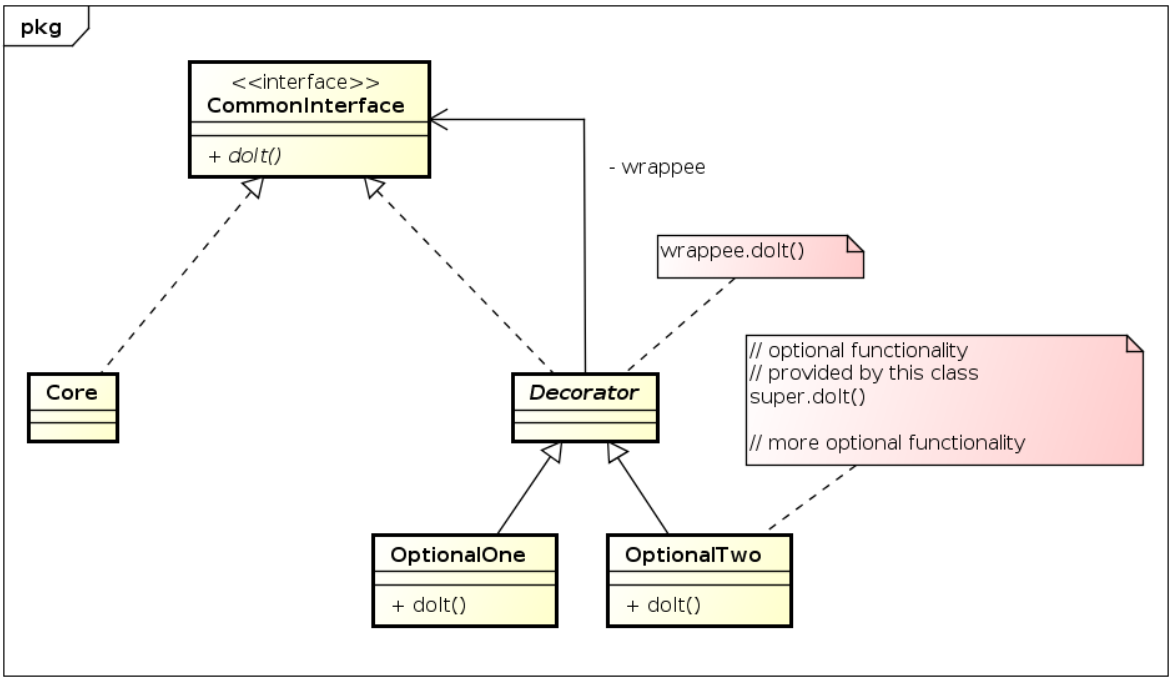
Name	Beschreibung
Command Pattern	Verpacken eines Kommandos in einem Objekt. Support of undoable Operations
Strategy Pattern	Variieren des ALgorithmus, je nach Client. Kapseln jeder Funktionalität und diese untereinander tauschbar gestalten.
Visitor Pattern	Kapselung von Operationen auf Elemente einer Objekt - Struktur. Neue Operationen ohne Veränderung von betroffenen Elementklassen.

Tabelle 3: Behavioral patterns Übersicht [1]

4 Das Decorator Pattern

Das Decorater Pattern ist weit verbreitet und wird selbstverständlich auch von Java verwendet. Dieses Pattern ermöglicht es, Funktionalität während der Laufzeit hinzuzufügen ohne die bereits bestehende Funktionalität eines Objketes zu verändern.

4.1 Allgemeines Klassendiagramm



4.2 Grundzüge des Designpatterns

Das Decorator - Pattern erlaubt uns das Verhalten bzw. die Funktionalität eines Objektes während der Laufzeit zu erweitern, ohne Änderungen direkt am Objekt vorzumehnen Das Dekorieren/Erweitern kann statisch, sowie in manche Fällen auch dynamisch erfolgen. Um dies zu erreichen wird die Basis Klasse als Decorater Klass implementiert. Dies wird erreicht indem wir diese Basis Klasse als Decorater Klasse wrapen. Es gibt 3 Implementations:

- **Inheritance**
- **Wrapper**
- **External**

4.2.1 Inheritance [2]

Es wird von der `Decorater Klasse` geerbt. Alle weiteren Funktionen können anschließend in der Kindklasse implementiert werden.

```
public class DebugButton extends JButton {
    public DebugButton() {
        addActionListener(new ActionListener() {
            System.out.println("debug message");
        });
    }
}
```

Listing 11: Decorator Pattern Inheritance [2]

Als Beispiel die `BufferedInputStream.class` Implementation. Jede `read()` Methode kann den buffer nach der nächsten Portion von Bytes fragen. Falls der Buffer leer ist wurde er wahrscheinlich von der `super.read()` Methode gefüllt.

Vorteile

- Wir können nahezu jeden Decorator implementieren.

4.2.2 Wrapper [2]

Die Idee beim wrapping ist, ein dekoriertes Objekt in dem Decorater Pattern zu verpacken. Das Decorator Pattern leitet requests zu dem gewrapptem Objekt und kann die neue Funktionalität anschließend vor / nach dem forwarden ausführen.

Bsp: **`BufferedReader.class`**

`BufferedReader.class` ist ein Wrapper von `InputStream`. Obwohl es die Klasse `InputStream` extended ist es ein Wrapper. Im Konstruktor von `BufferedReader` wird ein `InputStram` initialisiert und behält diesen als instance Variable. Aufrufe können anschließend an den `InputStream` weitergeleitet werden.

Bsp: **`DebugButton`**

Wir wollen den JButton erweitern um einen DebugButton zu erhalten. Dieser DebugButton hat dieselben Funktionen wie ein noramler JButton, abgesehen davon, dass er bei jedem Druck eine Message "log msgäusgibt.

```
public class DebugButton extends JButton implements ActionListener {
    private JButton butt = null;
    public DebugButton(JButton butt) {
        this.butt=butt;
        butt.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
    }
}
```

Listing 12: Decorator - Wrapper [2]

4.2.3 External [2]

Um die External Implementierung darzustellen, wird wieder das Bsp mit DebugButton hergenommen. Zuerst wird eine `Decorator Class erstellt`.

```
public class DebugDecorator implements ActionListener {
    public void decorateDebug(JButton butt) {
        butt.addActonListenr(this);
    }
    public void undecorateDebug(JButton butt) {
        butt.removeActonListenr(this);
    }
    public void actionPerformed(ActionEvent evt) {
        System.out.println("debug message for button" + src);
    }
}
```

Listing 13: Decorator External [2]

Nun können wir den folgenden Code überall anwenden.

```
DebugDecorator decor = new DebugDecorator();
JButton myButt = ...
// Add external decorator
decor.decorateDebug(myButt);
// Remove external decorator
decor.undecorateDebug(myButt);
```

Listing 14: Decorator External Aufruf[2]

4.3 Vor & Nachteile des Decorator Patterns

4.3.1 Vorteile

Die Vorteile bestehen darin, dass mehrere Dekorierer hintereinandergeschaltet werden können; die Dekorierer können zur Laufzeit und sogar nach der Instanziierung ausgetauscht werden. Die zu dekorierende Klasse ist nicht unbedingt festgelegt (wohl aber deren Schnittstelle). Zudem können lange und unübersichtliche Vererbungshierarchien vermieden werden.

4.3.2 Nachteile

Das Muster hat eine Gefahr: Da eine dekorierte Komponente nicht identisch mit der Komponente selbst ist (als Objekt), muss man beim Testen auf Objekt-Identität vorsichtig sein. (Ein Vergleich kann falsch ausgehen, obwohl dieselbe Komponente gemeint ist.) Zudem müssen bei der Verwendung von dekorierten Komponenten die Nachrichten vom Dekorierer an das dekorierte Objekt weitergeleitet werden.

4.4 Anwendungsfälle in der Java API

Java verwendet in zahlreichen Packages das Decorator Pattern. Das erleichtert dem Entwickler einiges und kann auch einfach umgesetzt werden. Beispielsweise im Package *java.io* oder *java.util.Collection* wird das Decorat-Pattern angewendet.

4.4.1 Package *java.io*

Die Struktur des InputStreams ist wie folgt aufgebaut:[4]

```
class java.io.InputStream
         class java.io.FileInputStream
                 class java.io.BufferedInputStream
                 class java.util.zip.CheckedInputStream
                 class java.io.DataInputStream
                 class java.security.DigestInputStream
                 class java.util.zip.InflaterInputStream
                         class java.util.zip.GZIPInputStream
                         class java.util.zip.ZipInputStream
                 class java.io.LineNumberInputStream
                 class java.io.PushbackInputStream
```


Um nun beispielsweise eine Datei `foo.zip` kann von folgendem Stream gelesen werden:

```
DataInputStream input = new DataInputStream ( new ZipInputStream ( new FileInputStream ("foo.zip") )
    );
```

Listing 15: Example Decorator java.io [2]

`class java.io.FileInputStream` ist ebenfalls in der `class java.io.InputStream` und beinhaltet somit das feld vom Typ `InputStream` zu dem alle Aufrufe delegeirt werden. Einfach gessagt fügt das Decorator Pattern funktonaliät hinzu ohne das Interface zu ändern.

Funktionsweise [7] [8]

Wenn wir uns InputStream und seine Implementationen genauer ansehen, dann bemerken wir, dass die Implementations wie BufferedInputStream / FilterInputStream im Konstruktor eine Instanz von der abstrakten Klasse. Dies ist eins der Merkmale eines typischen Decorater Patterns. (Dies gilt auch für Konstruktoren, welche eine Instanz von demselben Interface nimmt)

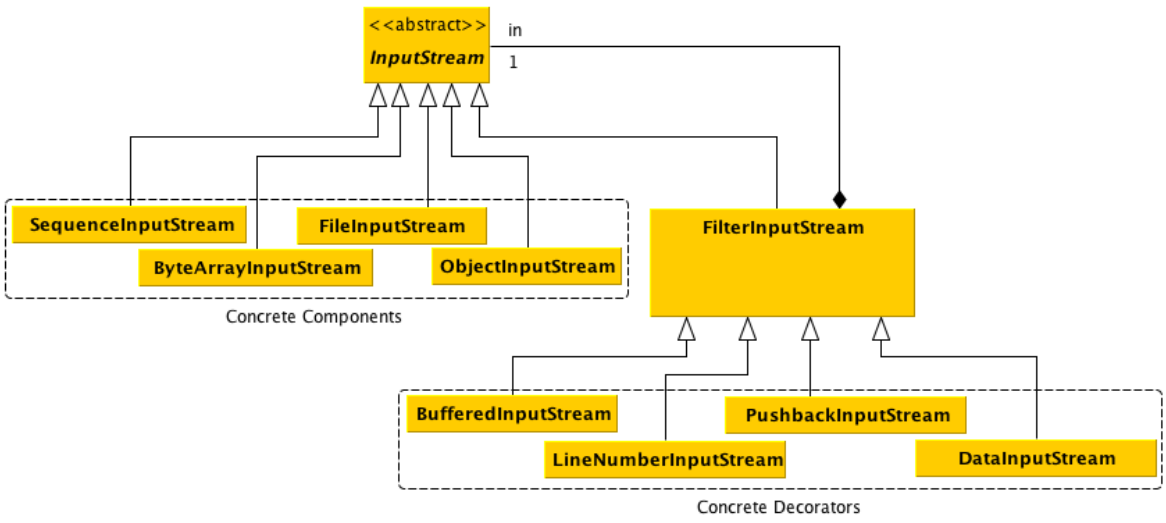


Abbildung 1: InputStream - Uml Class Diagram [3]

4.5 package `javax.swing` [3] [4]

An zahlreichen Stellen in GUI-Bibliotheken werden GUI-Komponenten dekoriert. Ein typisches Beispiel ist die JScrollPane von Swing.

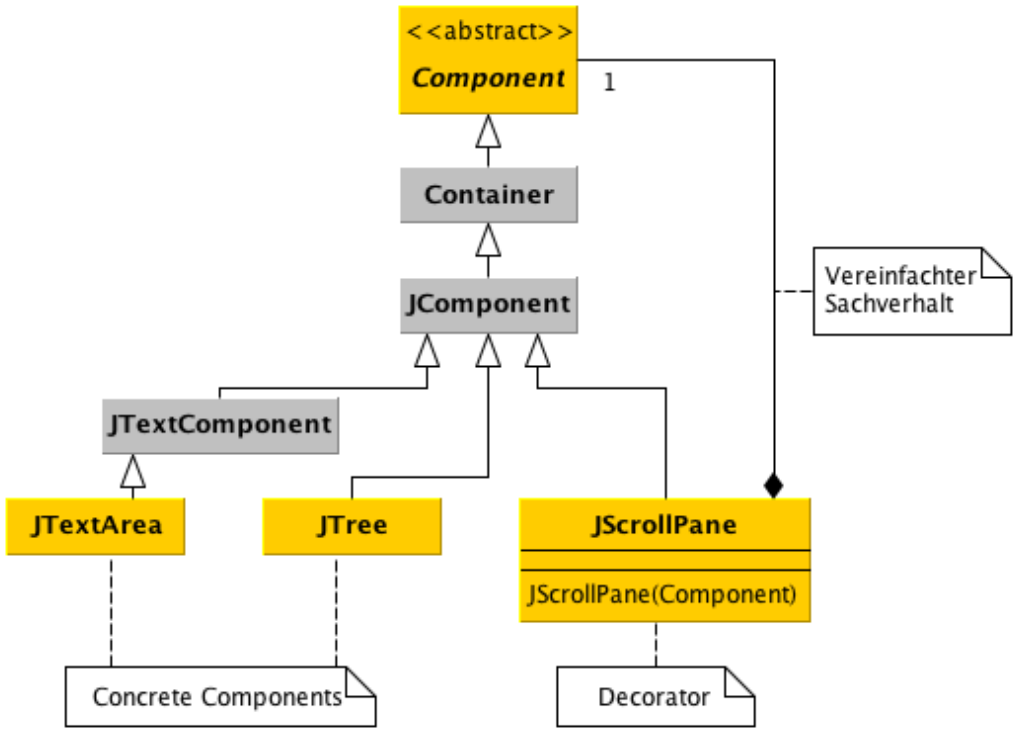


Abbildung 2: JScrollPane - Uml Class Diagram [3]

Soll eine JTextArea oder ein JTree Scrollbalken erhalten, so muss man das Objekt nur einer JScrollPane als Konstruktorparameter übergeben und das JScollPane-Objekt stattdessen (es ist ja auch eine Component) auf den Container (JPanel/JFrame) hinzufügen. Soll nun die Component gezeichnet werden, so fügt das JScrollPane-Objekt dem JTextArea-Objekt das nötige Verhalten für Scrollbalken hinzu.

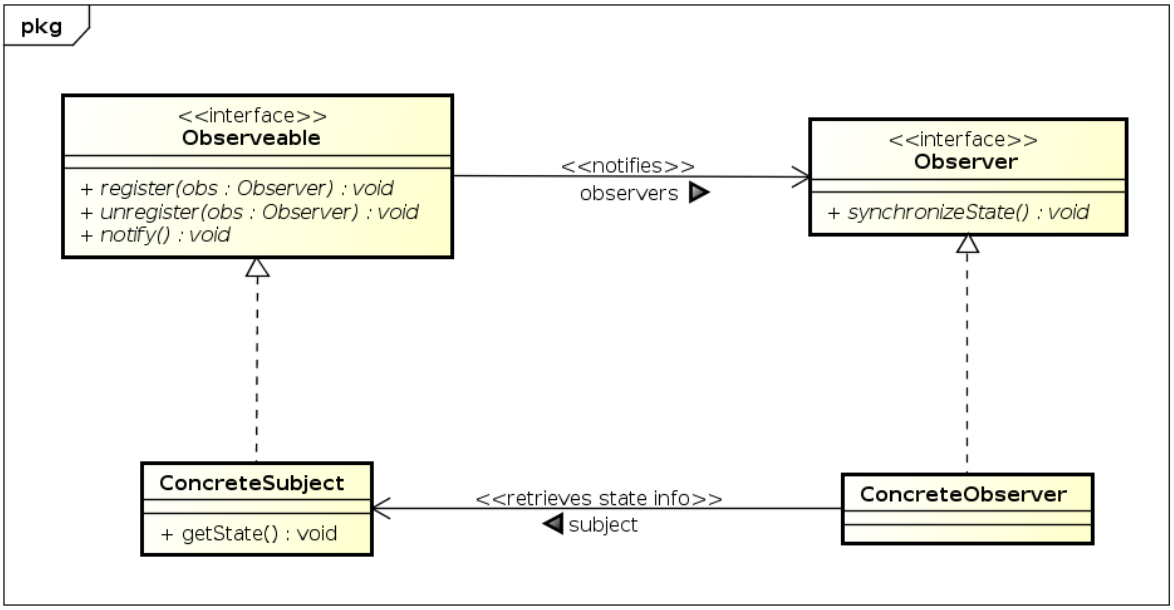
```
1  JTextArea textComponent = new JTextArea(12,20);
   JScrollPane scrollComponent = new JScrollPane(textComponent);
3  add(scrollComponent);
   //oder add(new JScrollPane(new JTextArea(12,20));
```

Listing 16: Example Decorator javax.swing.JScrollPane [3]

5 Das Observer Pattern

Das Observer Pattern auch Beobachter Pattern basiert auf dem Prinzip vom allbekannten abonnieren. Man abonniert von einem *Observable* und wird zum *Observer*. Anschließend kann nun je nach Implmentierung nach dem ***Pull*** oder nach dem ***Push*** Prinzip ein Datenstream geupdatet werden.

5.1 Allgemeines Klassendiagramm



5.2 Grundzüge des Observer Patterns

Es gibt immer einen zu abonnierenden Observable und ein Abonneten den Observer. Hierbei ist es üblich, dass man einen neuen Subsriber mit der `register` Methode hinzufügen kann. Um dies zu gewährleisten werden alle Subscriber anschließend in einer Art Liste verwaltet und somit in der `register` Methode dieser Liste hinzugefügt.

Ebenfalls soll es möglich sein sich zu trennen und unsubscriben. Dies wird mit der Methode `unregister` gewährleistet. Desweiteren ist nun noch abhängig ob nach dem Pull oder Push Prinzip gearbeitet wird. Wird ein Push Prinzip verwendet, so hat meist der Observer eine `update` Methode und sobald das Observable **notified** wird, so wird mittels einer Schleife alle Observer in der Liste iteriert und geupdatet.

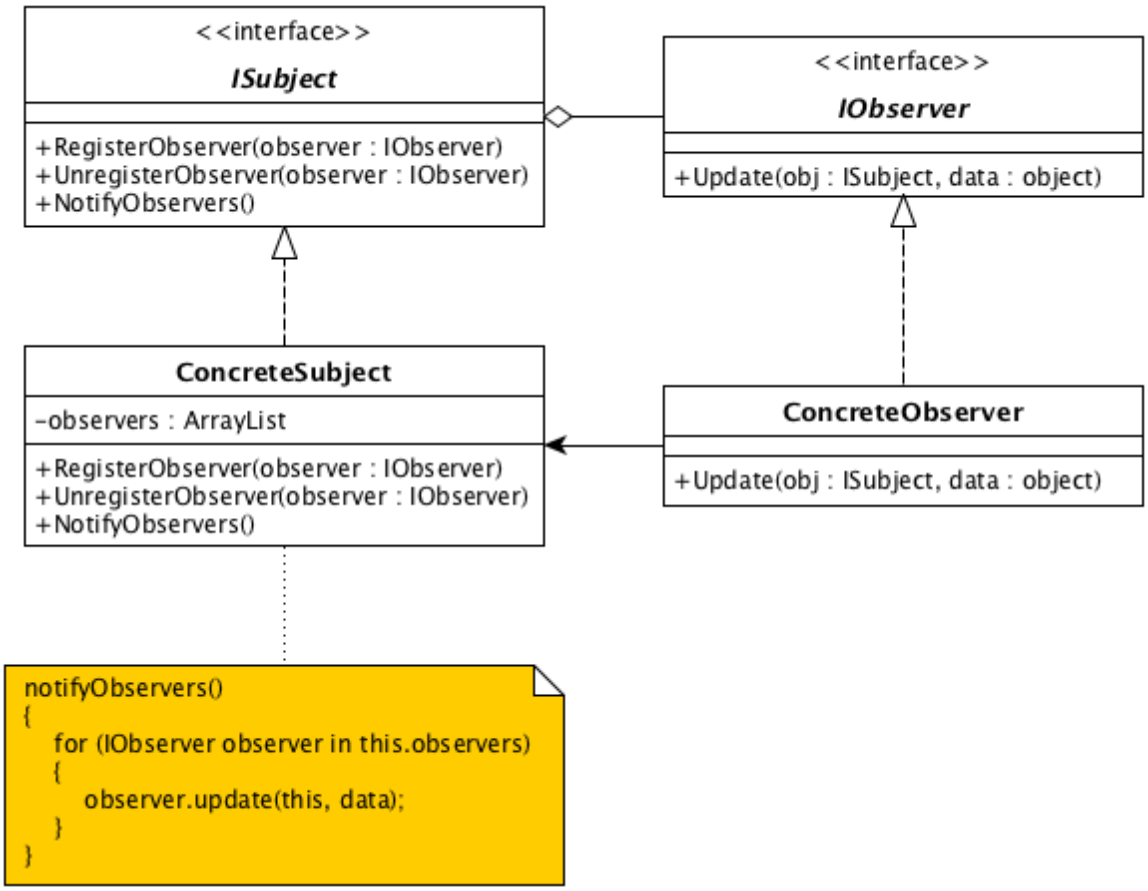
5.2.1 Beispiel: Messstation / Wetterstation [5]

er Entwurf soll es ermöglichen verschiedene Anzeigen zur Darstellung der Wetterdaten einzubinden, ohne grössere Änderungen am bestehenden Code durchzuführen.

Deshalb soll auf eine Schnittstelle anstatt auf eine Implementierung gearbeitet werden. Die Wetterstation ist das **Subjekt**, welches von den Anzeigen als **Beobachter** beobachtet wird und die bei einer Änderung der Daten benachrichtigt werden.

Das Subjekt benötigt daher Methoden, die es ermöglichen Beobachter an- und abzumelden sowie über Änderungen zu benachrichtigen. Die Beobachter selbst benötigen eine Methode, die bei einer Änderung vom Subjekt aufgerufen wird und die Änderung mitteilt. Da in dem Beispiel auch eine Anzeige für die Wetterdaten verwendet wird, wird dafür ebenfalls eine Schnittstelle definiert, die die Anzeige aktualisiert.

Klassendiagramm [5]



Akteuere [5]

- **Subject (Subjekt)**
  Definiert eine Schnittstelle für die An- und Abmeldung von Beobachtern sowie zur Benachrichtigung der Beobachter über Zustandsänderungen.
- **ConcreteSubject (KonkretesSubjekt)**
  Implementiert die Schnittstelle des Subjekts. Speichert den Zustand des Objekts und benachrichtigt alle Beobachter über Zustandsänderungen.
- **Observer (Beobachter)**
  Definiert eine Schnittstelle zur Benachrichtigung über die Zustandsänderung des Subjekts.
- **ConcreteObserver (KonkreterBeobachter)**
  Implementiert die Schnittstelle zur Benachrichtigung über die Zustandsänderung eines konkreten Subjekts.

5.3 Vor & Nachteile des Observer Patterns

5.3.1 Vorteile [6]

Observer und Observables können unabhängig voneinander variiert werden. Diese sind ebenfalls voneinander lose gekoppelt und die Verbindung kann jederzeit getrennt werden.

Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Beobachter zu besitzen, sondern kennt diese nur über die Beobachter-Schnittstelle. Ein abhängiges Objekt erhält die Änderungen automatisch. Es werden auch Multicasts unterstützt.

5.3.2 Nachteile [6]

Änderungen am Subjekt führen bei großer Beobachteranzahl zu hohen Änderungskosten. Außerdem informiert das Subjekt jeden Beobachter, auch wenn dieser die Änderungsinformation nicht benötigt. Zusätzlich können die Änderungen weitere Änderungen nach sich ziehen und so einen unerwartet hohen Aufwand haben.

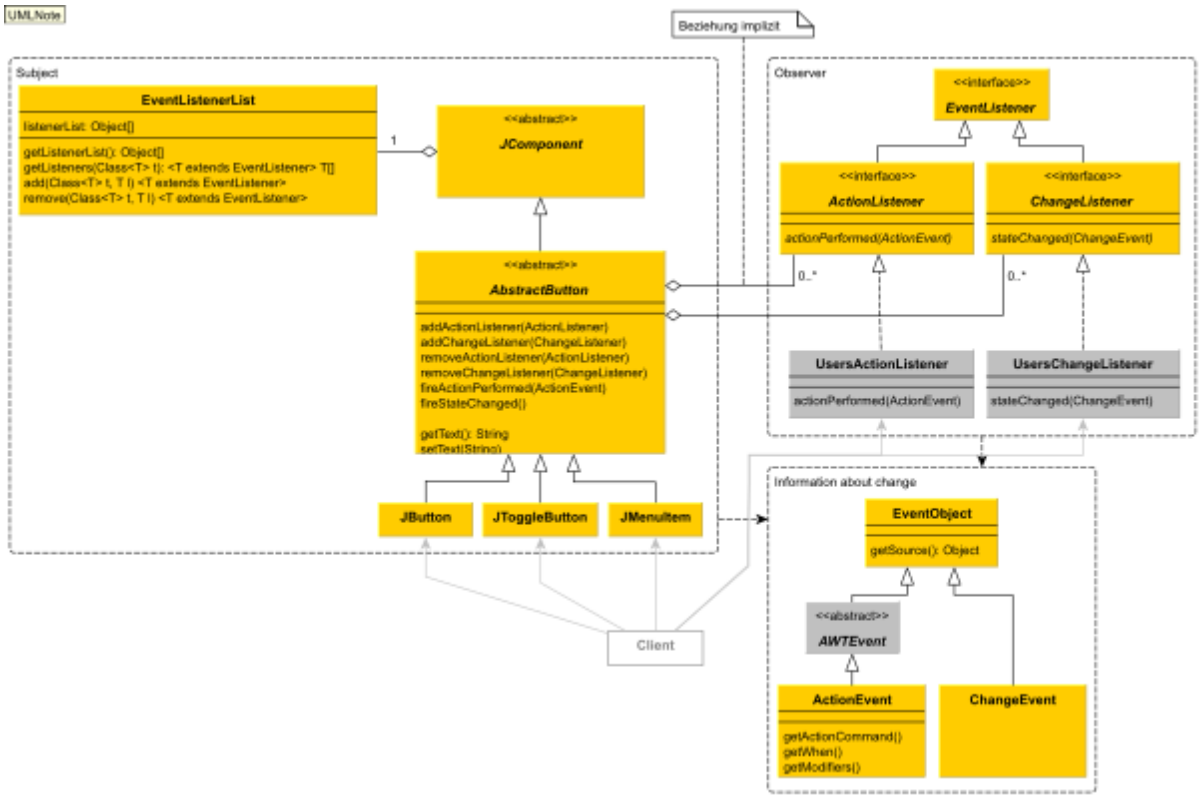
Der Mechanismus liefert keine Information darüber, was sich geändert hat. Die daraus resultierende Unabhängigkeit der Komponenten kann sich allerdings auch als Vorteil herausstellen.

5.4 Anwendungsfälle des Observer / Beobachter Patterns

Das klassische Anwendungsbeispiel in der Java-API ist das Eventhandling von AWT/Swing. Die GUI-Komponenten sind dabei die Subjects, bei den sich Listener, die Observer, registrieren können. Findet eine Userinteraktion auf der Komponente statt, so werden alle registrierten Listener benachrichtigt.

Der Aufbau wird im folgendem am Beispiel von Buttons und dem dazugehörigen Action- und Change-Listenern dargestellt.

Klassendiagramm [9]



Jeder Swing-Button erbt von AbstractButton Methoden zum An- und Abmelden von Listenern, sowie eine Methode fireActionPerformed() (bzw. fireStateChanged()) zur Benachrichtigung der entsprechenden Listener. Diese müssen die Aktualisierungsmethode actionPerformed(ActionEvent) bzw. stateChanged(ChangeEvent)) implementieren.

Literatur

[1] https://de.wikipedia.org/wiki/Entwurfsmuster.

[2] http://www.javaworld.com/article/2072703/design-patterns/three-approaches-for-decorating-your-code.html.

[3] http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/SHUTTLE/seminar/ausarbeitungen/Entwurfsmuster_Ausarbeitung.pdf.

[4] https://docs.oracle.com/javase/7/docs/api/.

[5] http://newsight.de/2015/01/13/das-observer-muster/.

[6] https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)#Vorteile.

[7] http://stackoverflow.com/questions/6366385/decorator-pattern-for-io.

[8] http://cecs.wright.edu/~tkprasad/courses/ceg860/paper/node26.html.

[9] https://www.philipphauer.de/study/se/design-pattern/observer.php#java_api.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis