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!

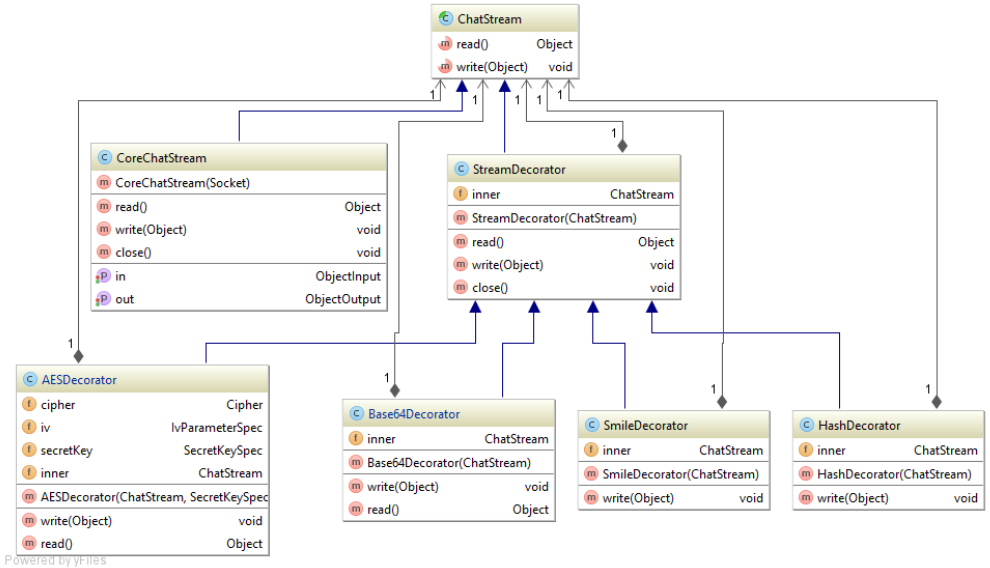
2.2 Umsetzung

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.
- ✓ **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**

Literatur

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

Tabellenverzeichnis

Listings

Abbildungsverzeichnis

