IET-GIBB

Entwicklungsportfolio Teilkomponente Network

Modul 326: Objektorientiert entwerfen und implementieren

Mirio Eggmann, INF.5G
Version 1.0, 23.12.2016

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellen

- https://de.wikipedia.org/wiki/Entwurfsmuster_(Buch)#cite_note-1 (22.11.2016)
- https://de.wikipedia.org/wiki/Fabrikmethode (22.11.2016)
- https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster) (23.12.2016)
- Head First Design Patterns - Buch (22.11.2016)
- Entwurfsmuster (GoF) - Buch (23.12.2016)

1 Entwurfsprinzip /-muster 1 – Observer [Bewerten]

1.1 Problemstellung

Änderungen bei einem Objekt auch bei anderen Objekten direkt übernehmen zu können wäre sehr nützlich. Dies ist mit dem Observer Pattern möglich. Sonst muss jedes Mal mühsam, bei jeder Änderung von Hand auch noch alles in den anderen Objekten geändert werden. Somit weiss dann eine Klasse viel zu viel von der gesamten Logik und eine spätere Änderung am Code ist sehr zeitaufwändig.

1.2 Auswahl des passenden Entwurfsprinzips / -musters

Das Beobachter-Muster ist ein Entwurfsmuster und gehört zur Kategorie Verhaltensmuster. Es dient zur Weitergabe von Änderungen bei einem Objekt zu allen Beobachtern (observer). Durch den Einsatz von diesem Pattern muss der Beobachter (subscriber, Abonnent) die Struktur der Komponente nicht kennen, denn sie werden vom Subjekt (publisher, beobachtetes Objekt) benachrichtigt, falls eine Änderung vorgenommen wurde. Somit ist auch die Konsistenz zwischen den verschiedenen Instanzen gewährleistet. Dies aber nur, falls der Beobachter dies möchte, er kann auch bei Laufzeit des Programmes aufhören zuzuhören.

Es gibt folgende Arten des Beobachter-Musters:

Push Notification

Wenn sich das Subjekt verändert werden alle Beobachter benachrichtigt. Jedoch werden bei dieser Art des Beobachter-Musters keine Daten mitgesendet, dies führt dazu das diese Methode sehr schnell ist.

Push-Update Notification

Wenn sich das Subjekt verändert werden alle Beobachter benachrichtigt. Bei dieser Art des Musters können die Update-Daten zusätzlich an die Beobachter weitergeleitet werden. Dies führt aber dazu, dass es weniger schnell ist.

Pull Notification

Der Beobachter fragt selbstständig nach dem Zustand des beobachteten Objekts nach.

Die folgende Grafik zeigt die verschiedenen Teile des Observer-Patterns auf:

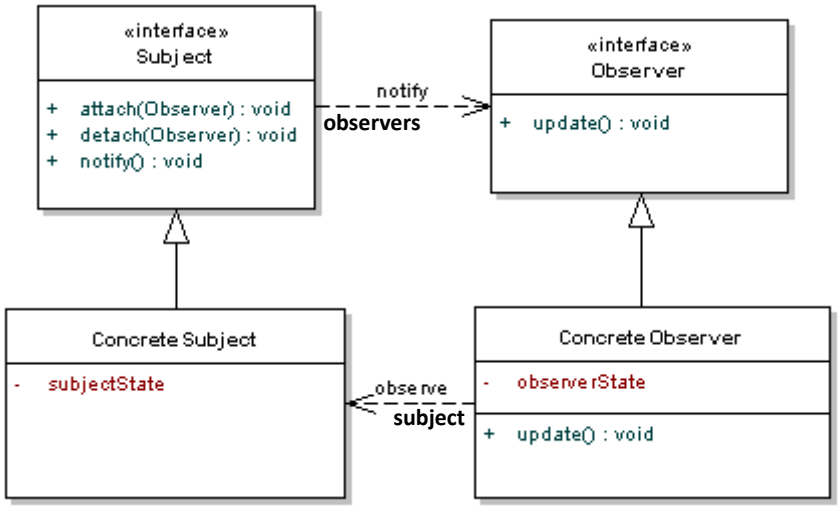


Abbildung 1 Observer Pattern - Diagramm

Subject

- Ein Interface zum Registrieren und Abmelden von Beobachtern
- Kennt alle seine Beobachter (observer).

Das folgende Code-Snipped zeigt wie das Subjekt in der notify()-Methode die update()-Methode von all den Beobachtern (observern) ausführt.

```
for observer in observers {
  observer.update()
}
```

Concrete Subject

- Speichert den Subjektstatus, der Status der für die Beobachter von Interesse ist
- Informiert die Beobachter über alle Änderungen die für sie von Interesse sind

Observer

- Stellt ein Interface zur Verfügung für Objekte die informiert werden sollen bei Änderungen vom Subjektstatus.

Concrete Observer

- Implementiert das Observer Interface um immer auf dem neusten Stand des Subjektes zu sein.

Die folgende Codezeile zeigt wie in der update() Methode des konkreten Beobachters der Status vom Subjekt im Beobachterstatus gespeichert wird.

```
observerState = subject.subjectState
```

1.3 Anwendung auf das ursprüngliche Problem

In der folgenden Grafik sieht man eine sehr vereinfachte Version unseres Observer Patterns. Hauptsächlich soll es zeigen, was es mit dem hinzufügen zu einer Liste auf sich hat und wie man in Java mithilfe von Observer und Observable Informationen an alle Abonnenten übergeben kann. Eine Übersicht was die verschiedenen Teile, wie „konkretes Subjekt“ bedeuten, ist auf der vorherigen Seite beschrieben.

```
// Konkretes Subjekt
class PostServer extends Observable {
  private String message;

  public String getMessage() {
    return message;
  }

  public void changeMessage(String message) {
    this.message = message;
    setChanged();
    notifyObservers(message);
  }

  public static void main(String[] args) {
    // Konkretes Subjekt erstellen (Observable)
    PostServer server = new PostServer();

    // Konkrete Observer erstellen
    PostClient client1 = new PostClient();
    PostClient client2 = new PostClient();
    PostClient client3 = new PostClient();
    PostClient client4 = new PostClient();

    // Die Clients in die Observer-Liste packen
    server.addObserver(client1);
    server.addObserver(client2);
    server.addObserver(client1);
    server.addObserver(client1);

    // nun bekommen dies alle hinzugefügten Clients mit und
    // geben es schlussendlich in der Kommandozeile aus weil ihre
    // update Methode aufgerufen wird
    server.changeMessage("Hello Players!");
  }
}

// Konkreter Beobachter
class PostClient implements Observer {
  public void update(Observable o, Object arg) {
    System.out.println(arg);
  }
}
```

Abbildung 2 Observer Pattern - Code Beispiel

1.4 Lernprozess

Das Observer Pattern habe ich vorher selbst noch nie verwendet. In unserer Network Teilkomponente haben wir das Pattern nun eingesetzt und ich habe es somit gut kennengelernt. Es ist sehr nützlich wenn es mehrere Klassen hat, die von einer Klasse informiert werden wollen, sobald eine Änderung vorgenommen wird. Bei unserer Network Teilkomponente war das praktisch, weil wir 4 Clients haben die vom Server informiert werden müssen bei Änderungen. In Java ist es praktisch, weil es bereits Interfaces hat (Observer, Observable) die das Subjekt und den Beobachter vorgeben.

2 Entwurfsprinzip /-muster 2 – Factory Method

2.1 Problemstellung

Bei grösseren Programmen kommt immer wieder das Problem, dass man ganz viele neue Objekte erzeugt, ganz viele «new» Schlüsselwörter im Programm findet. Dies hat den Nachteil, dass bei einer Änderung überall diese kleinen Codeschnipsel gesucht und geändert werden müssen. Somit ist der Code schlecht Änderbar und auch die Wiederverwendbarkeit ist nicht gewährleistet.

2.2 Auswahl des passenden Entwurfsprinzips / -musters

Grundsätzlich war von Anfang an klar, dass wir eine Factory verwenden. Das Factory Method Pattern zeigt sich als passend. Am Ende war es jedoch nicht mehr so ganz klar, ob es nicht auch ein paar Züge vom Abstract Factory Pattern hat in unserer Implementierung. Nun wurde das Ganze noch umgeschrieben, damit es sicher dem Factory Method Pattern entspricht. Die Implementierung im Code konnte aber noch nicht geändert werden, wird aber so bald als möglich vorgenommen.

Das Factory Method Pattern gehört zu den sogenannten Erzeugungsmustern. Es wird manchmal ebenfalls als «virtueller Konstruktor» bezeichnet. Es definiert ein Interface zum Erstellen von Objekten und lässt dann die Unterklassen entscheiden wie sie erstellt werden sollen. Es reduziert die Anzahl «new» Schlüsselwörter in einem Programm auf ein Minimum und macht den Code somit viel konsistenter.

2.3 Anwendung auf das ursprüngliche Problem

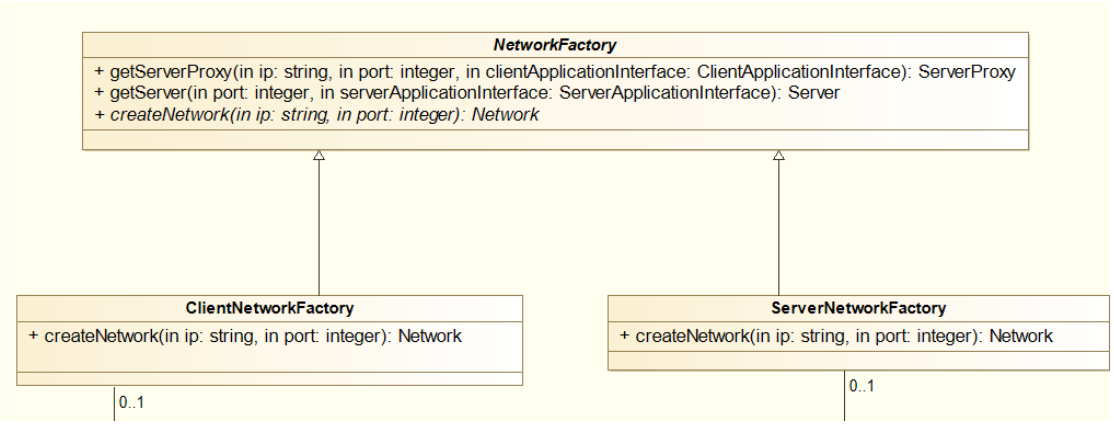


Abbildung 3 Factory Pattern - Erzeuger Teil

NetworkFactory (Erzeuger)

Der Erzeuger, bei uns NetworkFactory stellt eine abstrakte Methode zur Verfügung, welche dann von den konkreten Erzeugern implementiert werden muss. Hier die Methode «createNetwork(…)».

ClientNetworkFactory (KonkreterErzeuger)

Muss die Methode von NetworkFactory implementieren. Anschliessend erzeugt es ein konkretes Produkt, in diesem Fall ein ClientNetwork.

ServerNetworkFactory (KonkreterErzeuger)

Muss die Methode von NetworkFactory implementieren. Anschliessend erzeugt es ein konkretes Produkt, in diesem Fall ein ServerNetwork.

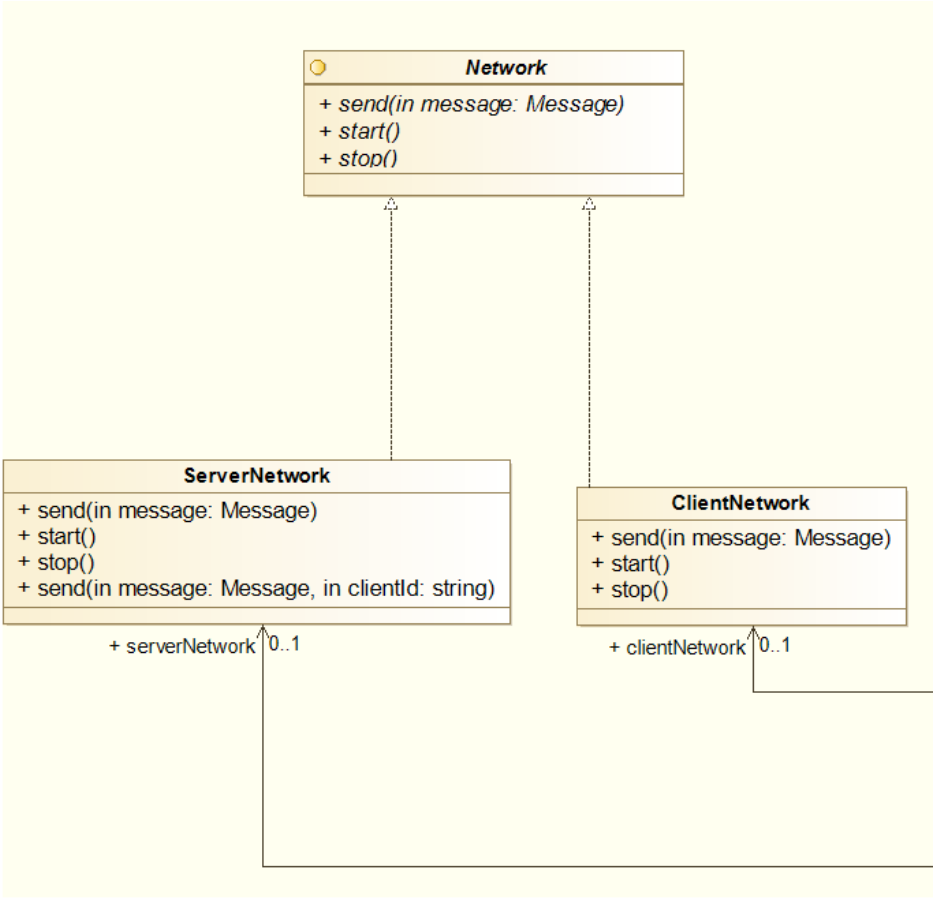


Abbildung 4 Factory Pattern - Produkt Teil

Network (Produkt)

Das ist das Basisprodukt für das konkrete Produkt.

ServerNetwork (Konkretes Produkt)

Erbt die Methoden vom Network und muss diese implementieren. Dies ist ein konkretes Produkt.

ClientNetwork (Konkretes Produkt)

Erbt die Methoden vom Network und muss diese implementieren. Dies ist ein konkretes Produkt.

2.4 Lernprozess

Ich habe beim Suchen des passenden Patterns herausgefunden, dass es nicht immer leicht ist genau auf ein Pattern schliessen zu können. Bei uns ist es jetzt grundsätzlich das Factory Method Pattern, welches wir verwendet haben, trotzdem sind einige Züge vom Abstract Factory Pattern vorhanden. Weiter musste ich am Schluss feststellen, dass ich das Factory Method Pattern nicht ganz richtig implementiert hatte und somit noch Änderungen am Code vornehmen werden muss. Ich habe es nun bereits in den UML Diagrammen richtig aufgebaut und nun werde ich den Code noch anpassen.

3 Entwurfsprinzip /-muster 3 - MVC

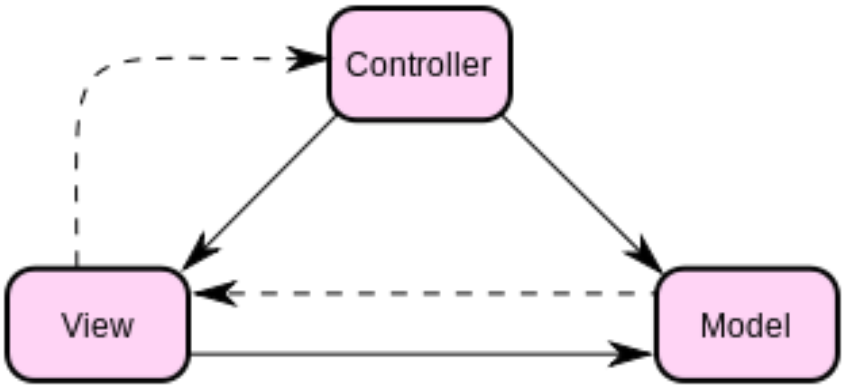
3.1 Problemstellung

Als Entwickler hat man immer wieder das Problem, dass man ein Produkt warten muss, welches ein anderer Programmierer vor Jahren geschrieben hat. Da taucht meistens das Problem auf, dass man nicht weiss, wo der vorherige Entwickler eine bestimmte Sache platziert hat und somit verbringt man Stunden damit die Stelle zu finden. Daher ist es sinnvoll ein Programm nach einem bestimmten Muster aufzubauen, damit die nächste Person, die den Code warten muss einen deutlich einfacheren Einstieg hat und nicht unnötig stundenlang verbringt etwas zu suchen, was man mit einem gewissen Muster sinnvoll platzieren könnte. Zum Beispiel wenn der vorherige Entwickler eine DB Abfrage, die Verarbeitung der Abfrage und anschliessend die Darstellung des Ergebnisses alles an einem Ort macht, muss man zuerst die Datei finden, wo das alles zusammen gemacht wird und weiter ist auch die Wiederverwendbarkeit bei null, wenn man ein Problem so löst.

3.2 Auswahl des passenden Entwurfsprinzips / -musters

Model (Modell) View (Präsentation) Controller (Steuerung), kurz MVC ist ein Muster zur Strukturierung der Software-Entwicklung. Dadurch sind spätere Wartungen am Programm deutlich leichter, weil der Code sinnvoll nach einer bestimmten Struktur angelegt wurde und somit schneller zu finden ist. Durch den Einsatz von MVC steigt auch die Wiederverwendbarkeit der Komponenten, da die Datenhalter, die Logik und die Präsentationsschicht getrennt sind. Dies ermöglicht es Applikationen für diverse Plattformen zu entwickeln und zum Beispiel überall dieselbe Datenstruktur (Modelle) zu nutzen. Dadurch müssen nur noch das vom User genutzte Frontend angepasst werden und eventuell die Steuerung durch die Controller. Die Modelle bleiben bestehend. Das MVC-Muster gilt als Architekturmuster oder auch als Entwurfsmuster.

In folgendem Bild sieht man wie der Controller, Model und View zusammen kommunizieren.



Controller
Nimmt Benutzeraktionen entgegen, verarbeitet diese und handelt entsprechend. Beim Handeln, kann es Daten aus den Models holen, aktualisieren oder auch hinzufügen. Weiter aktualisiert der Controller auch die angezeigten Views.

View
Stellt die Daten für den Benutzer grafisch dar. Die Views sind selber relativ dumm und machen das was ihnen der Controller sagt.

Model
Die Models halten die Daten der Applikation. Diese werden meistens von Controller verwaltet.

3.3 Lernprozess

Ich habe dieses Pattern bereits relativ häufig verwendet und daher schon gekannt. Trotzdem habe ich durch das repetieren nochmals einige neue Sachen gelesen und werde diese in Zukunft anders machen.

4 Entwurfsprinzip /-muster 4 - Singleton

4.1 Problemstellung

Gerade wenn man zum Beispiel ein Spiel entwickelt, welches Daten speichern soll, welche nur während der Spiellaufzeit verfügbar sein sollen, ist es nicht sinnvoll mehrere verschiedene Objekte zu haben. Sonst müssen die Daten unter Umständen noch an einem Zwischenort, wie in einer Textdatei gespeichert werden, damit nicht plötzlich inkonsistente Daten Zustände entstehen. Auch dies kann wieder Probleme verursachen mit dem Datei Zugriff. Daher ist es viel sinnvoller eine Klasse zu haben, von dem es in der ganzen Applikation nur ein Objekt hat, welches immer weiterverwendet werden kann und somit in der Applikation verfügbar sind und keine Unterschiede von der Aktualität mehr entstehen.

4.2 Auswahl des passenden Entwurfsprinzips / -musters

Das Singleton Design Pattern ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster. Singleton stellt sicher, dass es von einer Klasse genau ein Objekt gibt und dadurch von der ganzen Applikation aus auf dasselbe zugegriffen wird.

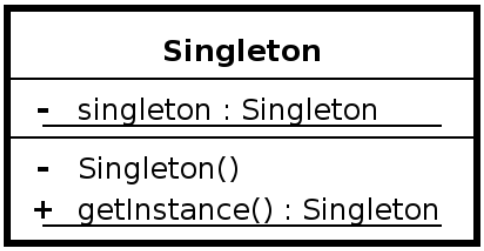
Das Singleton findet Verwendung, wenn

- nur ein Objekt zu einer Klasse existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird oder
- das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

Anwendungsbeispiele sind

- ein zentrales Protokoll-Objekt, das Ausgaben in eine Datei schreibt.
- Druckaufträge, die zu einem Drucker gesendet werden, sollen nur in einen einzigen Puffer geschrieben werden.

Auf folgendem Bild kann man gut erkennen, dass der Konstruktor private ist, damit nicht beliebig viele Objekte erstellt werden können. Dies wird in der getInstance() Methode geregelt. Diese erstellt ein neues Singleton Objekt, falls noch keines besteht und sonst gibt es das bestehende zurück.



4.3 Lernprozess

Das Singleton Pattern habe ich bereits in einem Programm verwendet. Dort hatte ich aber noch keine Ahnung, dass ich ein Design Pattern eingesetzt habe. Nun weiss ich genau wie es funktioniert und in Zukunft kann ich es im richtigen Moment einsetzen.