Abteilung: Höhere Lehranstalt für Informatik

Handout

Mediator

Thema:	POSE Mediator Design-Pattern Handout
Fach:	POSE
Erstelldatum:	February 25, 2023
Letzte PDF-Erstellung	February 25, 2023
Ersteller:	Simon Fischer 5AHIF

Contents

1 Motivation

In komplexerer (objektorientierter) Software kann es recht schnell vorkommen, dass Objekte Abhängigkeiten / Verbindungen zu vielen anderen Objekten besitzen, was gegen das lose Kopplungs Prinzip spricht und somit die Wiederverwendbarkeit einzelner Teile des Programms verhindert oder zumindest stark erschwert. Im Extremfall würde dies (wie in der Grafik 1) so aussehen, dass jede Klasse eine Referenz auf jede andere Klasse besitzt.

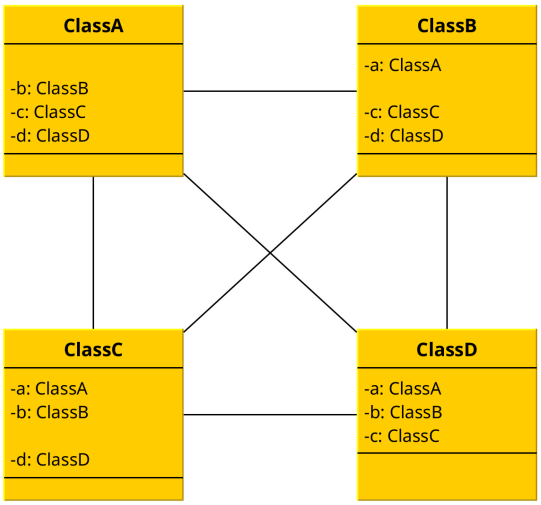

Figure 1: kein Mediator - Extremfall

[vgl. 1, S. 273ff]

2 Einleitung

Um die, bereits in der Motivation erwähnten, vielen Abhängigkeiten zwischen den einzelnen Klassen aufzulösen und somit lose Kopplung herzustellen, wird ein Mediator eingesetzt. Wie auch in der Grafik 2 veranschaulicht, interagieren Objekte dann nicht mehr direkt miteinander, sondern immer über den Mediator. Dies führt sowohl zu einigen Vor- als aber auch Nachteilen (siehe: Vor- und Nachteile vom Mediator). Weiters gehört der Mediator zur Gruppe der Behavioral-Pattern.

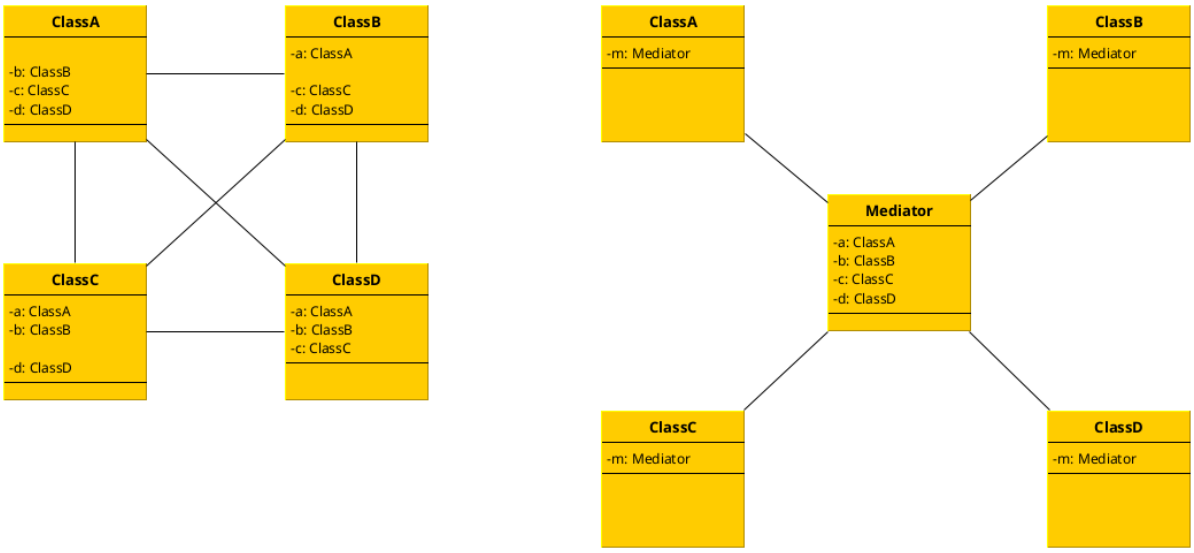

Figure 2: Vergleich kein Mediator / Mediator - Extremfall

[vgl. 1, S. 273ff]

2.1 zusätzlichen Themen

Um das Mediator Design-Pattern besser verstehen zu können, benötigt es auch das Verständnis folgender Konzepte:

2.1.1 lose Kopplung

Bei Kopplung im Software-Bereich unterscheidet man meist zwischen starker und loser Kopplung. Als Kopplung wird der Grad der Ab- /bzw. Unabhängigkeit zwischen zwei Komponenten beschrieben. Ziel ist es, möglichst unabhängige Software-Komponenten und somit modulare Software zu erstellen. Die Änderung einer Komponente sollte möglichst keine Auswirkung auf andere haben. Dies hat unter anderem folgende Vorteile:

- bessere Wartbarkeit: nicht jede kleine Änderung benötigt Änderung am "gesammten" System. z.B. Datenquelle kann einfach an einer **einzigen** Stelle ausgetauscht werden und alle funktioniert weiterhin
- Wiederverwendbarkeit: einzelne Komponenten können in neuen Projekten verwendet werden ohne das "komplette" alte Programm kopieren zu müssen
- Testbarkeit: Während Unit-Test können einzelne Komponenten (oft z.B. Datenbank) gegen emulierte Varianten ausgetauscht werden.
- ...

[vgl. 2], basierend auf [vgl. 3, S. 101ff]

Lose Kopplung in Java:
Lose Kopplung in Java (und auch vielen anderen Sprachen) wird oft über Interfaces (bzw. in anderen Sprachen dazu äquivalentes) realisiert. Hier ein simples Beispiel eines Taschenrechners:

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }
}

@RequiredArgsConstructor
public class CalculartorApp {
    private final Calculator calculator;

    public int add(int a, int b) {
        return calculator.add(a, b);
    }

    public int multiply(int a, int b) {
        return calculator.multiply(a, b);
    }
}

public class Main {
    public static void main(String[] args) {
        var calcApp = new CalculartorApp();
        System.out.println(calcApp.add(3, 14));

        System.out.println(calcApp.multiply(4, 2));
    }
}
```

Listing 1: starke Kopplung Beispiel

Änderungen in der Calculator-Klassen (ändern der Methoden-Namen / Ändern der Parameter: z.B. Änderung auf Array, damit beliebige Anzahl übergeben werden kann), benötigen in diesem simplen Beispiel zumindest Änderung der CalcularorApp-Klasse oder im schlimmsten Fall sogar bis zur Main-Methode, wenn die CalculatorApp-Klasse nicht als Facade agiert und dann z.B. ebenfalls ein Array erwartet.

Eine bessere Lösung dieser Problemstellung könnte wie folgt aussehen:

```
1  public interface ICalculator {
2      public int calculate(int a, int b);
3  }
4
5  public class Adder implements ICalculator {
6      public int calculate(int a, int b) {
7          return a + b;
8      }
9  }
10
11 public class Multiplier implements ICalculator {
12     public int calculate(int a, int b) {
13         return a * b;
14     }
15 }
16
17 @RequiredArgsConstructor
18 public class CalculartorApp {
19     private final ICalculator calculator;
20
21     public int calculate(int a, int b) {
22         return calculator.calculate(a, b);
23     }
24 }
25
26 public class Main {
27     public static void main(String[] args) {
28         var adder = new CalculatorClient(new Adder());
29         System.out.println(adder.calculate(3, 14));
30
31         var multiplier = new CalculatorClient(new Multiplier());
32         System.out.println(multiplier.calculate(4, 2));
33     }
34 }
```

Listing 2: lose Kopplung Beispiel

Jedes "new" sorgt wieder für einen gewissen Grad an starker Kopplung, daher wird das Konzept der Interfaces dann noch mit Dependency-Injection kombiniert, um einen maximal möglichen Grad an Unabhängigkeit zu gewährleisten.

2.2 Verwendungsbeispiele aus der Praxis

Das Mediator Pattern ist in der Praxis häufig im Einsatz. Unter anderem findet es bei GUI-Anwendungen (auch im JDK) sehr gern, aber natürlich auch bei anderen Problemstellungen Anwendung.

- MVC: in fast allen Fällen agiert der Controller beim MVC Pattern als Mediator zwischen Model und View. z.B. in einer Swing-Applikation (oder auch Web-Applikation wie z.B. Angular) kümmert er sich um Data-Binding mit dem View, aktiviert / deaktiviert z.B. einen "absenden" Button basierend darauf, ob bereits alle notwendigen Eingaben getätigt wurden und sendet dann beim Klick dieses die Daten weiter z.B. Persistierung der Daten in Datenbank (bei "HTML-Generatoren", wie z.B. JSF wird die Backing-Bean teils als Mediator gesehen / teils nicht, da sie oft "nur" Daten für den JSF->Html Generator bereitstellt)
- Swing: Event Dispatch Thread: 1 Thread, der sich um sämtliche GUI-Relevante dinge, kümmert - alles wird NUR von diesem Thread ausgeführt: Änderungen / Benutzer-Events / ... → sicheres mutlithreading (wenn mehrere Threads am UI Änderungen durchführen, führt die meist nur zu Problemen)
- AWT ActionListener: auch hier wird intern ein Mediator verwendet, der bei Events im GUI benachrichtigt wird und dann entsprechend das Event an die einzelnen ActionListener weiterleitet
- java.util.concurrent: Mediator koordiniert Kommunikation zwischen Threads. Synchronisation durch z.B. Blocking-Queues, ...
- Javascript event loop: Der Javascript Event Loop kann ebenfalls als Mediator gesehen werden. Er erhält von Objekten Ereignisse / Aufgaben wie z.B. senden einer asynchronen HTTP-Request, und kümmert sich darum, sobald das Ergebnis des Servers da ist, eine entsprechende Callback-Funktion aufzurufen
- Linux: D-Bus
- ...

2.2.1 konkreten Anwendung: Smart-Home-Hub

In Grafik 3 ist eine einfache konkrete Anwendung des Mediator Design Patterns zu sehen: Ein zentraler Smart-Home-Hub wie z.B. Home-Assistant / ..., der als Mediator zwischen den einzelnen Komponenten dient.

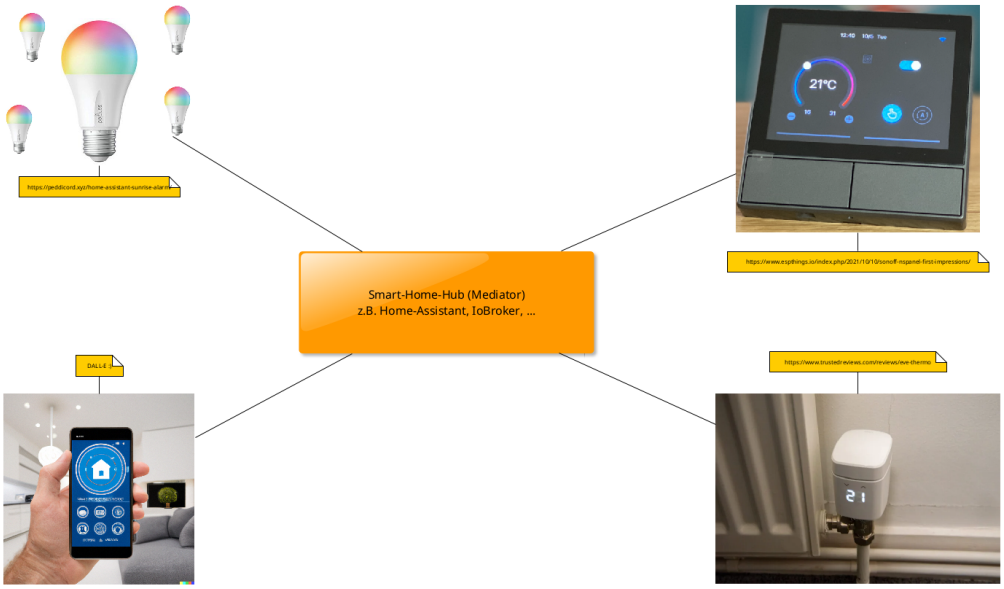

Figure 3: grafische Darstellugn Smart-Home-Hub-Mediator

Beispielsweise könnte hier folgende Kommunikation ablaufen:

- **Smartphone App**: interagiert mit Zentrale:
  - steuert Komponenten: Licht, Heizung, ...

  - erhält Informationen: Information über aktuelle Temperatur → Push-Benachrichtigung, wenn zu kalt, ...

- **smarter Heizkörperregler**: wird vom Hub gesteuert und stellt die entsprechend gewünschte Temperatur ein
- **smarte Lampe**: wird vom Hub gesteuert, wenn Person auf Lichtschalter drückt aber auch z.B. automatisch durch Bewegungsmelder, automatische Aufwachroutine, ...
- **smarter Lichtschalter** (hier z. B.: Sonoff-NSPanel): lässt sowohl Licht als auch andere Dinge steuern / Informationen anzeigen

2.3 Vor- und Nachteile vom Mediator

2.3.1 Vorteile

- Lose Kopplung: Da der Mediator lose Kopplung fördert, sind die Vorteile dieses (Vorteile llose Kopplung) auch auf den Mediator anwendbar.
  - Wartbarkeit: Änderungen in einer Colleague-Klasse benötigt nur Änderung im Mediator. Andere Colleague-Klassen bedürfen keinen Änderungen
  - Wiederverwendbarkeit: Colleague und Mediator Klassen können unabhängig wiederverwendet / ausgetauscht werden
  - Testbarkeit: Es kann einfach eine andere Implementierung des Mediator-Interfaces verwendet werden.
- Änderbarkeit / Erweiterbarkeit: Anderes Verhalten bzw. weitere Funktionen benötigt nur Änderung / erstellen einer Sub-Klasse bzw. Implementierung des Interfaces des Mediators. Colleague-Klassen können oft 1 zu 1 wiederverwendet werden,
- zentrale Stelle zur Verwaltung der Interaktions-Logik zwischen den Objekten. Auch sind dadurch die Komplexität der Colleague Objekte selbst,
  - bessere Übersicht / Lesbarkeit / Verständnis,
  - bessere Wartbarkeit,
  - vereinfachte Interaktionslogik: vorher Many-To-Many Beziehungen zwischen Colleagues - jetzt nur mehr One-To-One/Many: Colleague -(one)> Mediator -(many)> anderer Colleague
- ...

2.3.2 Nachteile

- Auslagerung der Komplexität der Interaktion zwischen Objekten in Mediator lässt diese nicht magisch verschwinden. Oft wird der Mediator selbst dann sehr komplex → wird zum Monolith, ohne den nichts geht → wieder nicht gut für Wiederverwendbarkeit
- "god object": im Worstcase wird der Mediator zu einem oft als "God Object" bezeichneten Objekt. Also einem Objekt, dass alles kann / macht. Eine Klasse mit **allen (tausenden) Funktionen** der Applikation.
- geringere Flexibilität: Interaktion muss / sollte immer über Mediator passieren
- Mediator verletzt 1. SOLID Design Prinzip (Single-responsibility): gibt meist deutlich mehr als einen Grund den Mediator zu ändern / hat mehr als 1 Aufgabe
- Skalierbarkeit: Mediator kann zum Bottleneck werden
- einige kleinere ("unwichtige") Nachteile:
  - KISS-Prinzip / Overengineering: für kleine Applikationen erhöht der Mediator (wie auch andere Design-Patterns) nur Sinnlos den Aufwand der Entwicklung. Es bedarf hier generell immer einer Abwägung: Ja, Design-Patterns sind wichtig und sinnvoll, aber man kann es auch übertreiben
  - (zusätzliches Objekt -> "höherer" Ressourcenbedarf)
- ...

[vgl. 1, S. 273ff] + eigene Ergänzungen

2.4 Verwandte Design-Patterns

Der Mediator ähnelt dem Facade-Pattern. Allerdings unterscheidet er sich wie folgt: Einen Mediator kann man sich ein bisschen wie eine BI-direktionale-Fassade vorstellen. Eine Facade stellt z.B. eine einfachere Schnittstelle für eine komplexe Library bereit. Ein Mediator vereinfacht die Kommunikation **zwischen** (also in beide Richtungen!) zwei Objekten. Zusätzlich kann ein Mediator zusätzliche Funktionalität bereitstellen, die von den einzelnen Colleague Objekten selbst nicht bereitgestellt wird / bereitgestellt werden kann. [vgl. 1, S. 273ff]

3 UML-Diagramm und Erklärung des Patterns

Wie bereits in der Einleitung erwähnt, ist das Ziel, dass die Colleague Objekte nicht mehr direkt miteinander, sondern über den Mediator kommunizieren. Dies gelingt durch folgenden Aufbau. In folgender Grafik 4 ist der grundlegende Aufbau eines Mediator dargestellt.

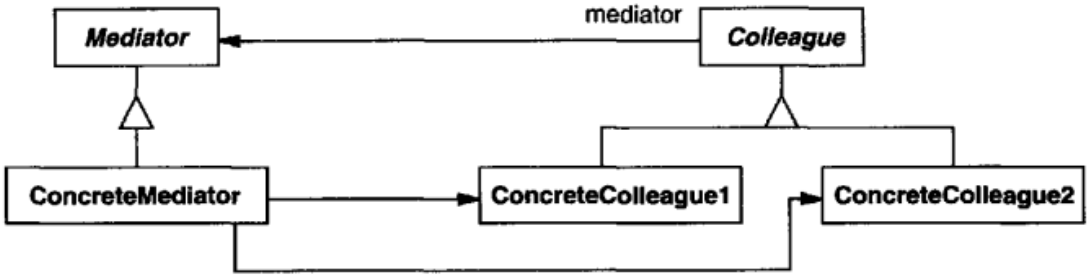

Figure 4: Mediator UML-Diagramm von: [1]

- **Mediator** (auch oft als "IMediator" benannt): meist ein Interface (oder abstrakte Klasse) definiert die Schnittstelle, über die Colleague-Objekte mit dem Mediator und darüber auch mit anderen Colleague-Objekten kommunizieren können.
- **ConcreteMediator** (oder nur "Mediator", wenn Interface "IMediator" ist):
  - eine konkrete Implementierung des Interfaces / der abstrakten Klasse. Hier ist die Logik des Mediators implementiert: simple Kommunikation zwischen Colleagues, neu-e Funktionalität (nicht in Colleague Klassen), ...
  - hält alle notwendigen (müssen nicht alle sein. z. B.: "Mediator-Clients", also Colleague Objekte, die nur auf Funktionen des Mediators zugreift, von diesem aber nicht aufgerufen werden) Referenzen auf mit ihm verbundenen Colleague-Objekte
- **Colleague** ("IColleague"): Noch eine weitere Abstraktions-Schicht (oft abstrakte Klasse), wenn mehrere Colleague-Objekte gleiche Funktionalitäten besitzen.
- **ConcreteColleague**: Ein konkretes Colleage-Object (bzw. Standard Business-Logik-Objekt)
  - **jedes** Colleague Objekt kennt / hat eine Referenz auf den Mediator (anders als in andere Richtung: Mediator muss nicht unbedingt jeden Colleague kennen)
  - kommunizieren jetzt nur mehr mit Mediator (anstatt direkt mit anderen Colleague-Objekten): verwenden Funktionalität vom und stellen Funktionalität für den Mediator bereit.

[vgl. 1, S. 273ff]

4 Darstellung des selbst programmierten Beispiels

INFO: Das gesamte Beispiel (samt dieser Ausarbeitung) ist auch zu finden unter: https://github.com/SimonFischer04/POSEMediator

4.1 Einleitung

Die Haupt-Funktionalität ist eigentlich in einem Bild 5 zusammengefasst:

Figure 5: Overlay - Seite

Durch Klick auf den Button der aktuelle Mikrofon-Status zwischen stumm / nicht stumm umgeschaltet. Die Komplexität liegt im Detail. Das Besondere:

- Animation beim Umschalten,
- Handling im Backend: Der Button führt einfach nur einen HTTP Request aus → könnte auch über jede andere Quelle geschehen (z.B. Remote per "Funk-Knopf"), zum Testen aber z.B. auch über Swagger-UI siehe Grafik 9.
- Live-Update: Das UI ist (über SSE, mit JSF Backing Bean Logik und dann mittels Websockets) zum Backend verbunden. Beispielhafte Anwendung: man hat nur 1 Monitor, kann am Handy immer noch aktuellen Status sehen / schnell wechseln, obwohl man gerade mit anderen Dingen beschäftigt ist → schneller, kein Fokuswechsel, ...
- Discord Integration (über Proxy): Ziel ist es das Mikrofon in Discord umzuschalten

4.2 Weitere UI - Seiten

Um auf Discord zugreifen zu können, sind Credentials nötig. Diese können auf, die in Grafik 6 ersichtlichen, Settings-Seite sicher (standardmäßig ausgeblendete Felder) konfiguriert und in einer Datenbank gespeichert werden.

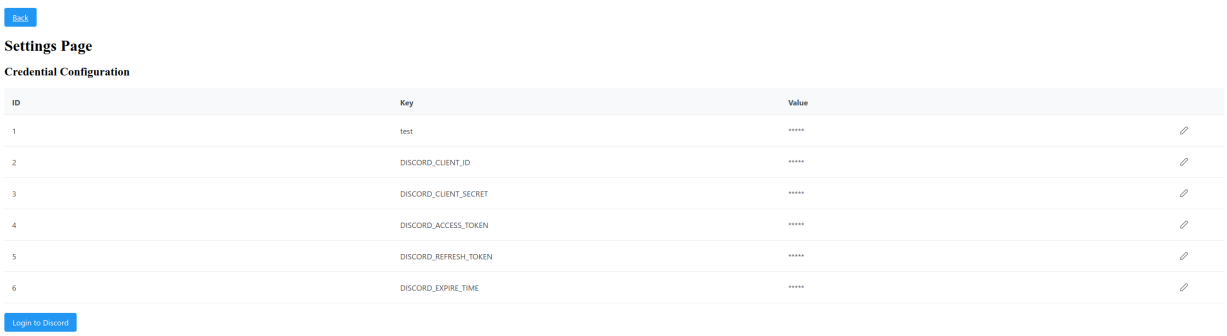

Figure 6: Settings - Seite

Zu guter Letzt noch die Startseite 7, wo einfach auf die bereits erwähnten Seiten (und eine Testseite zum Debuggen) navigiert werden.

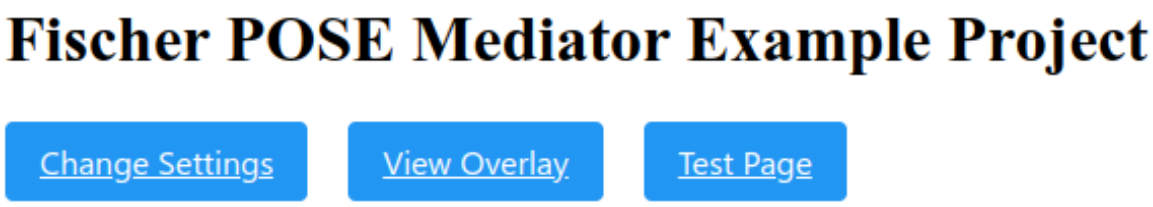

Figure 7: Startseite

4.3 Backend

Zuer Realisiertung dieses Beispiels war **einiges** notwendig. (Fast) das ganze UML-Diagramm ist in Grafik 8 ersichtlich. Hierbei gibt es folgende Gruppen an Klassen:

- Logik: Business-Logik der Anwendung:
  - Haupt bzw. Mediator - Logik: hier ist das Design Pattern implementiert: Der Mediator kümmert sich um Eingang HTTP Event (RemoteController) → Verbindung zum UI (UIConnector) und auch handling von Discord (DiscordConnector). Den DiscordConnector gibt es hierbei in 2 Varianten: einen "echten" und eine Demo zum Testen ohne Discord. Die tatsächliche Kommunikation mit Frontend passiert dann im *WSService* bzw. der Implementierung dieses Interfaces.
  - weitere Logik wie Credential-Verwaltung / bereitstellen dieser über API, ...
- Model: Model- Klassen bzw. Records. (Die Verbindungen / Abhängigkeiten zu den einzelnen Logik-Klassen wurden hier zur besseren Übersichtlichkeit weggelassen)
- Konfiguration: einige Klassen, die zur Konfiguration der Spring Applikation / Websocket-Verbindung / MapStruct-Mapper notwendig sind

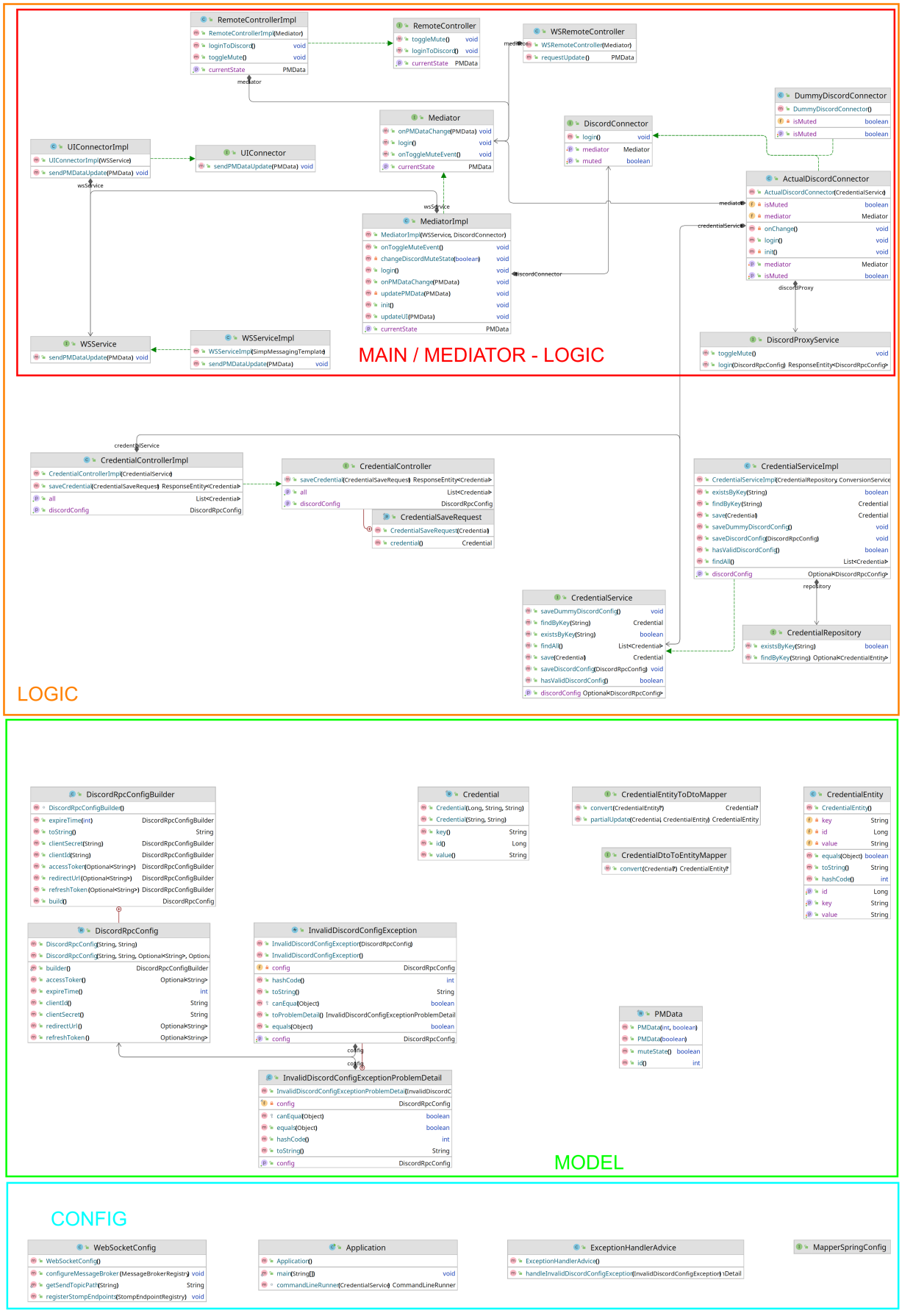

Figure 8: Backend UML-Diagramm

Auch ist ein Swagger-UI zum Testen implementiert:

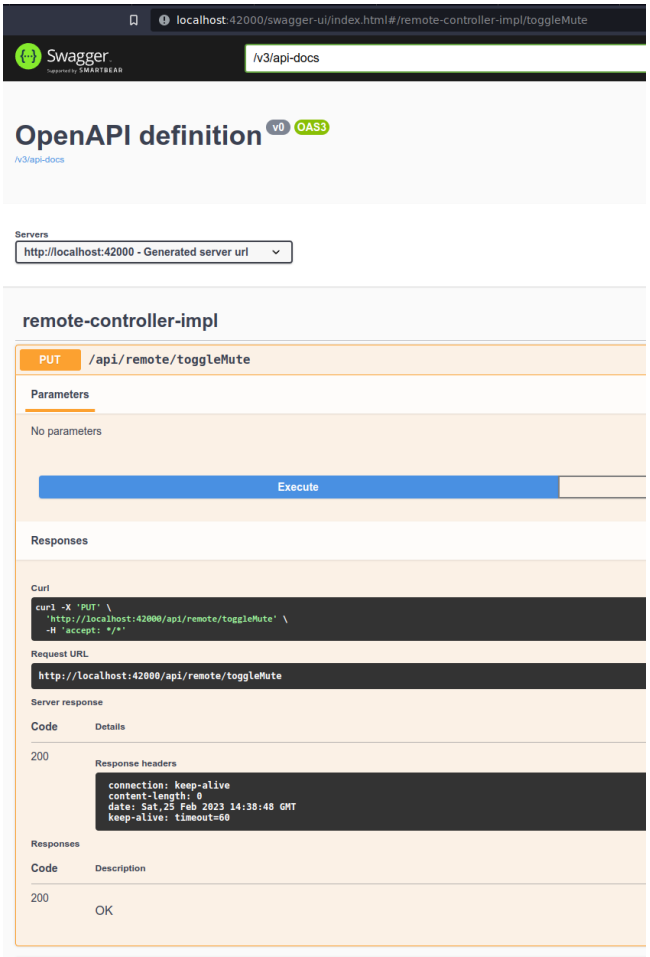

Figure 9: Swagger -UI

4.4 Frontend

Im Frontend sieht der Aufbau wie folgt aus:

Logik : auch im Frontend ist bewusst ein Mediator implementiert (oder mehr, wenn man wie in Verwendungsbeispiele aus der Praxis erwähnt, die Backing-Beans auch als Mediator zählt). Zwischen Backing-Bean und Browser besteht eine Server-sent events (SSE) Verbindung (SSEServlet), damit dieses (unter Zuhilfenahme von UpdateManager, ...) eine Live-Änderung im Browser bewirken kann, wenn über Websocket (WS) ein Event hereinkommt. Zusätzlich noch ganze Credential Handling für Discord mit Microprofile Rest-Client, ...

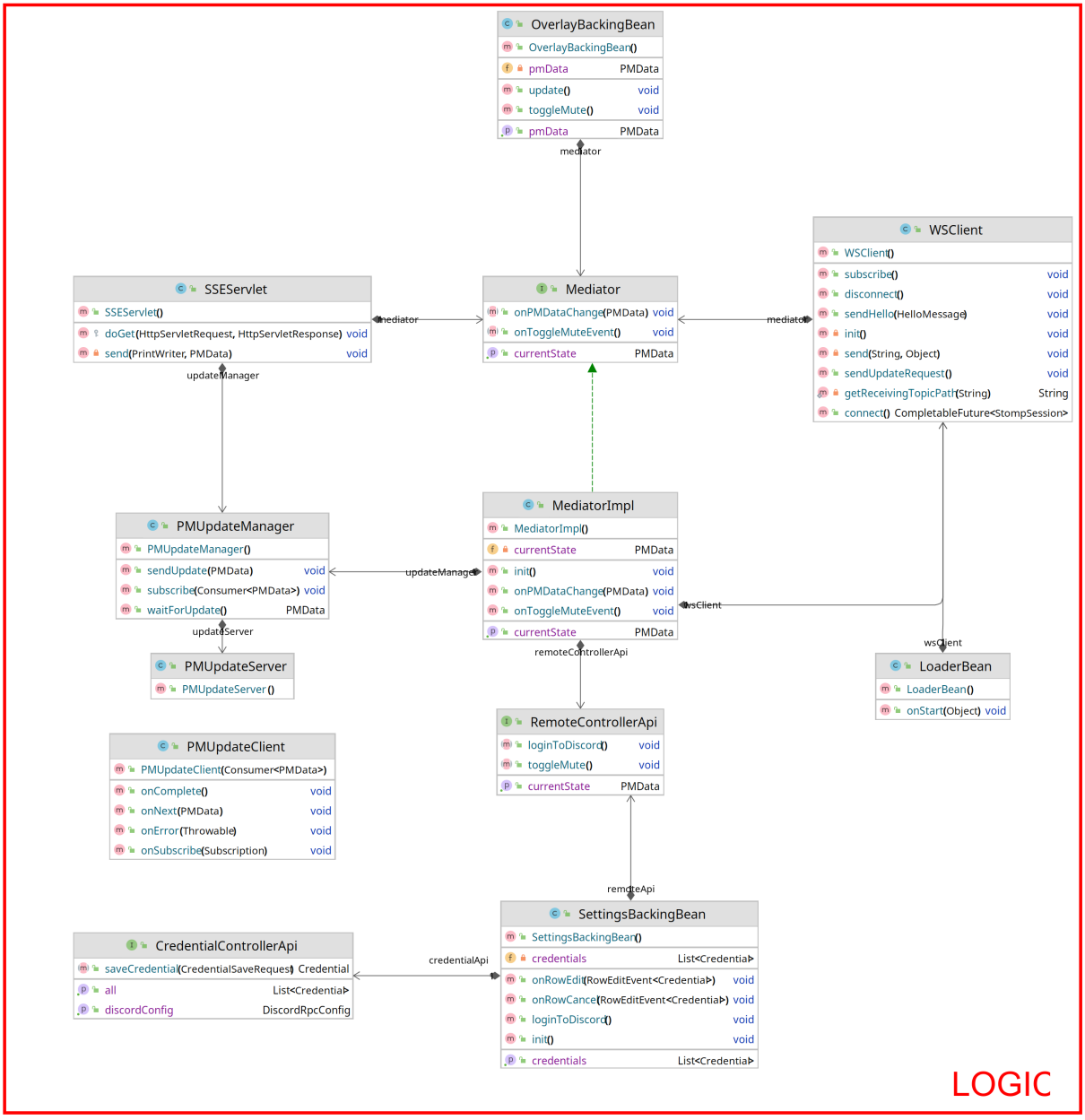

Figure 10: Auszug aus Frontend UML-Diagramm - Teil 1: Logik

Model / Andere : dann hier noch die Modellklassen + ein paar andere. (auch hier wurde die Verbindungen zu den Modellklassen zur Übersichtlichkeit weggelassen). Hierbei noch auffälig ist, dass es viele Model-Klassen "doppelt" gibt: einmal die tatsächlichen Model-Klassen des Frontends (meist Records) und 1-mal die generierten Models aus der API (POJO-Klassen).

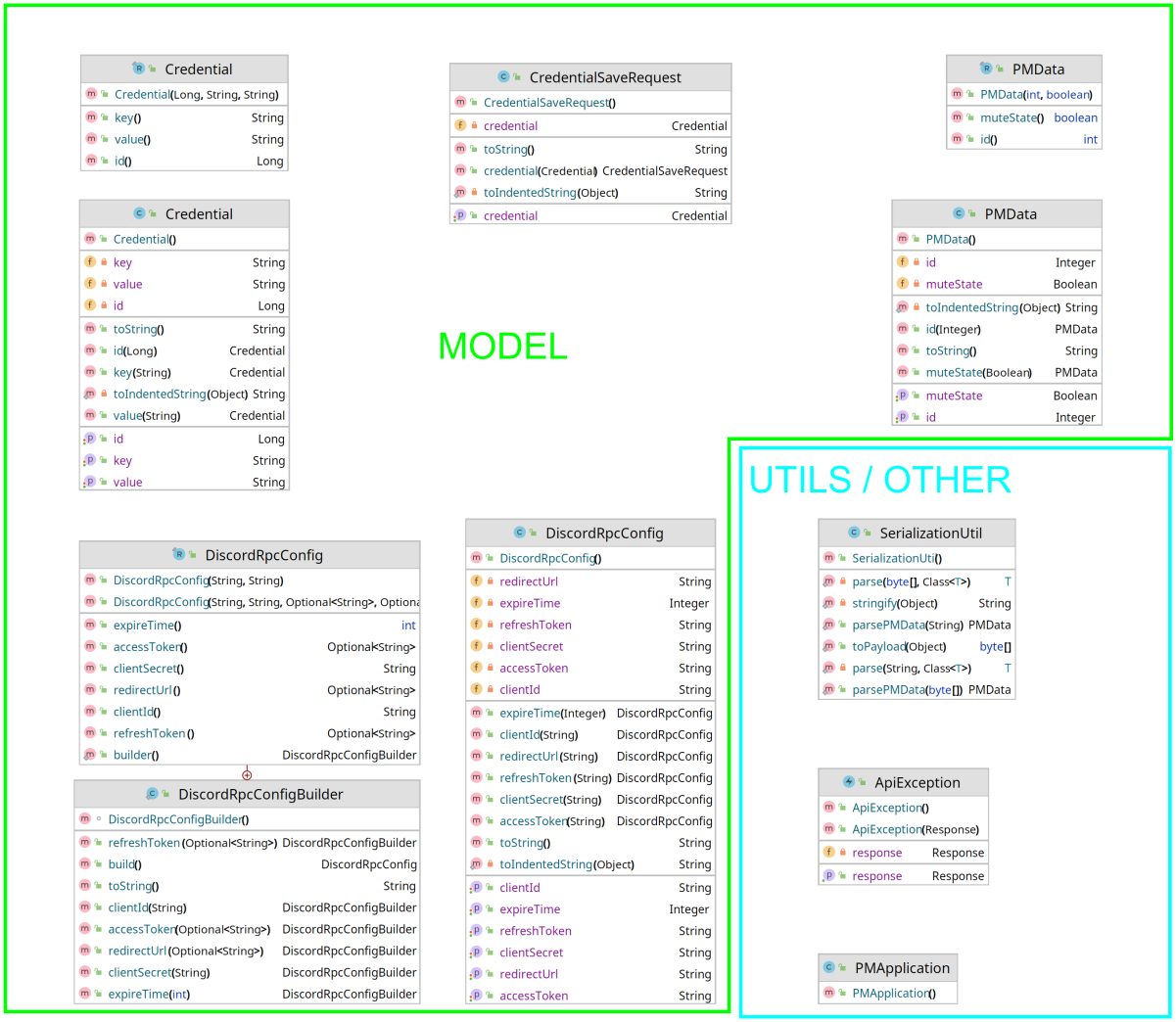

Figure 11: Auszug aus Frontend UML-Diagramm - Teil 2: Model + Andere

4.5 weitere Sub-Projekte

: Neben Backend und Frontend mussten noch weitere kleine "Sub-Projekte" implementiert werden. Siehe Resümee

5 Resümee

Beim Implementieren dieses Beispiels gab es **einige** Herausforderungen, die **VIEL** Zeit in Anspruch genommen haben!

- JSF:
  - Import von CSS (und JavaScript files) in JSF ...
  - SSE in JSF: Servlet → View-Verbindung → update zurück zur Backing Bean (RemoteCommand)
    - → ohne komische Render-Bugs: rendered attribute auf einem output-label lässt komplett woanders Elemente verschwinden, update Attribute ändert auch komplett andere Ids Div-Elemente → Animation wird abgebrochen, ...
- keine (+ mit Jarkarta EE / Wildfly kompatiblen) Java Websocket-Client / STOMP Libraries
  - → Sub-Projekt: eigene Websocket-Library (Wrapper) für mehr Informationen siehe https://github.com/SimonFischer04/POSEMediator/tree/main/StompLib
- Discord Integration ...:
  - → Sub-Projekt: Discord-Proxy: Proxy Applikation, die sich tatsächlich um die direkte Discord-Interaktion kümmert. Simple NodeJS / Express Anwendung mit REST-API, dass im Backend dann verwendet wird. :
  - *Warum?* TLDR: keine vernünftigte Library für Java. Für mehr Informationen siehe: https://github.com/SimonFischer04/POSEMediator/tree/main/DiscordProxy
- (vorher Neuland - und daher auch großen Ziel dieses Beispiels gewesen: Spring Boot 3 "und Dinge drum herum")
  - Java 17 (Records, ...)
  - Tests mit "alten" (noch nicht gelernten) Java Features: verschiedenste Queues und Dequeues → migriert zu Java 9 Flow-API, ThreadPools / Futures / CompleteableFutures, ...

Acronyms

SSE Server-sent events. 13

WS Websocket. 13

Besondere Begriffe

Behavioral-Pattern	"Behavioral-Pattern", zu Deutsch "Verhaltensmuster", sind Design Pattern, welche die Interaktion zwischen Objekten beschreiben. 3
Design Pattern	"Design Pattern", zu Deutsch "Entwurfsmuster", beschreiben den Aufbau von Software-Systemen. 17
lose Kopplung	"Lose Kopplung", Englisch: "loose coupling", beschreibt eine Architektur, in der Komponenten unabhängig voneinander funktionieren und Änderungen an einer Komponente keine Auswirkungen auf andere Komponenten haben. Für weitere Informationen siehe: lose Kopplung. 3, 7

References

[1] R. J. u. J. V. Erich Gamma Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, ISBN: 0-201-63361-2 (cit. on pp. 3, 7, 8).

[2] J. D. Orton, J. Douglas, and K. Weick, "Loosely coupled systems: A reconceptualization," *Academy of Management Review*, vol. 15, pp. 203–223, Apr. 1990. DOI: 10.2307/258154 (cit. on p. 4).

[3] M. Page-jones, "The practical guide to structured systems design," Jan. 1980 (cit. on p. 4).

List of Figures

List of Tables

List of source codes