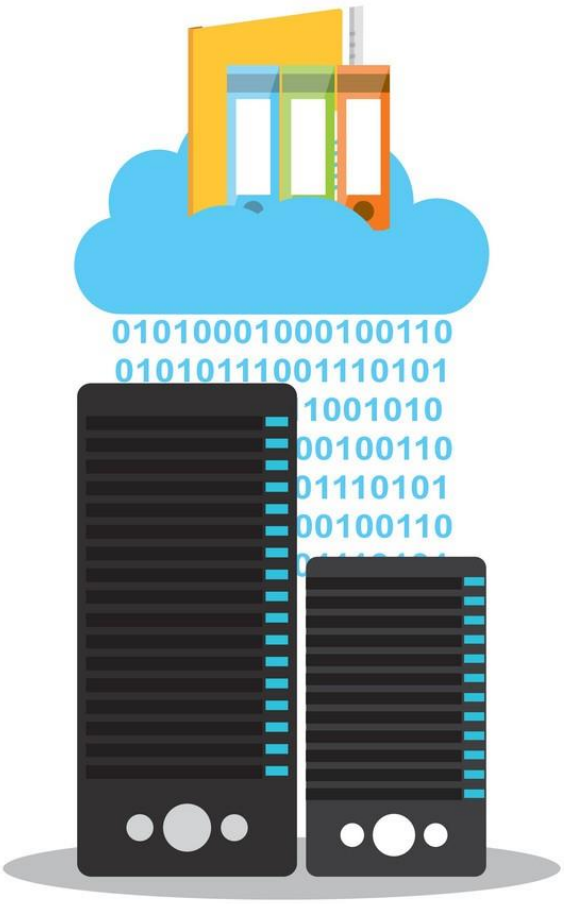
Entwicklungsportfolio Teilkomponente Server

M326 Objektorientiert entwerfen und implementieren

Version 1.0 18.12.2019

Winkler Olivier INF5J



Inhaltsverzeichnis

Abbildungsverzeichnis

1. Lernjournal

1.1 Dienstag 15.10.2019

Heute war der erste Tag nach den Ferien. Ab heute sind wir in der Realisationsphase unseres gemeinsamen Projekts. Als erstes wurde ein Überblick von der Lehrperson gezeigt, was uns bis zu den Weihnachtsferien bevorsteht. Als erstes habe ich das Git-Repository von unserem Gitlabserver geklont. Zu unserer Teilkompetenz haben wir einige Videos für die Umsetzung auf unserem Klassenlaufwerk. Ich habe angefangen unser Projekt auf den Stand der ersten zwei Videos zu bringen. Nächste Woche werde ich mit meinem Entwicklungsteam weiter an der Implementierung weiterarbeiten.

1.2 Dienstag 22.10.2019

Heute habe ich mit der Realisation begonnen. Ich habe begonnen das Projekt korrekt aufzusetzen. Mithilfe von den Videos auf dem Moduleshare habe ich dies erledigt und bin bis und mit Video Nr.3 gekommen. Dabei habe ich die korrekten Verzeichnisse erstellt und die dazugehörenden Klassen. Nächste Woche versuche ich die weiteren Videos zu schauen und auf Basis von diesen den application.server korrekt zu implementieren.

1.3 Dienstag 29.10.2019

Heute habe ich an der Realisation weitergearbeitet. Letzte Woche habe ich das Projekt bis und mit Video Nr.3 aufgesetzt. Diese Woche bin ich bis und mit Video Nr.6 gekommen. Das Projekt ist schon fast fertig aufgesetzt. Wenn das Projekt fertig aufgesetzt ist, kann meine Gruppe mit der Verfeinerung des Programmes beginnen und für den gemeinsamen Merge vorbereiten. Nächste Woche versuche ich die Videos zu beenden. Heute hatte ich keinerlei Schwierigkeiten und konnte mich gut meiner Arbeit widmen.

1.4 Dienstag 05.11.2019

Heute bin ich mit den Screencasts fertig geworden. Ich konnte unsere Applikation auf demselben Stand wie die Videos bringen und habe diese in unser gemeinsames Gitlab-Repository gepusht, damit alle meine Teammitglieder meine Änderungen haben. Somit können wir am heute mit der weiteren Implementierung beginnen um die restliche Funktion des Servers zu erreichen. Zudem bereiten wir uns auf den ersten gemeinsamen Merge vor. Ich konnte heute sehr gut Arbeiten. Zusätzlich hat uns Herr Järmann noch unsere Bewertung gegeben zu dem Dokument, welches wir vor den Ferien abgeben mussten. Die dort genannten Kritikpunkte versucht unser Team nun umzusetzen.

1.5 Dienstag 12.11.2019

Heute ist der erste Abgabetermin dieses Entwicklungsportfolios. In der ersten Leistungsbeurteilung von diesem Dokument werden die Tagesjournale bewertet. Im zweiten Teil der restliche Inhalt inklusive Entwurfsmuster. Zudem habe ich mir heute Gedanken über das weitere Vorgehen gemacht. Ich habe mich über verschiedene Designpatterns in Java informiert, welche man in unserer Applikation implementieren könnte. Auch habe ich versucht die genannten Kritikpunkte von unserem ersten Dokument in dieses hier zu integrieren, um eine bessere Gesamtbewertung zu bekommen.

1.6 Dienstag 19.11.2019

Heute habe ich mit meinem Team an der Weiterentwicklung der Applikation gearbeitet. Wir haben zudem uns über Designpatterns informiert, welche wir noch einbauen könnten. Bei der Weiterentwicklung ging es vor allem über das Handling, wenn alle Spieler beigetreten sind und der Server die Informationen von allen Spielern dem Netzwerk weitersendet. Heute konnte ich gut arbeiten und das Teamwork war sehr gut.

1.7 Dienstag 26.11.2019

Heute habe ich mich mit dem Netzwerkteam ausgetauscht. Zusammen haben wir angeschaut wie unsere beiden Teilkomponente am besten miteinander kommunizieren. Zudem haben wir in den Dokumenten nachgeschaut, ob alle Kriterien von unserer Applikation abgedeckt werden. Unser Team muss noch mit den restlichen Teams zusammenarbeiten und die Applikation selber darauf anpassen.

1.8 Dienstag 03.12.2019

Heute habe ich nicht allzu viel an meiner Applikation geschrieben. Ich habe die meiste Zeit mit dem Netzwerkteam geschaut wie man am besten die Nachrichten der Clients verarbeitet. Momentan ist der Stand eher so, dass wir ein bisschen abhängig vom Netzwerkteam sind, da wir zuerst ihre Methoden benötigten, die sie bei uns dann verarbeiten lassen wollen.

1.9 Dienstag 10.12.2019

Die Applikation endet sich dem Ende. Das Ziel wäre, dass nächstes Mal alle ihre Features in den Master mergen und im besten Fall alles funktioniert. Bei uns sieht das momentan aber noch nicht so gut aus. Wir haben noch einige Probleme mit dem Handling der Nachrichten. Unser Team versucht bis nächste Woche dies zu beheben um eventuell mit der Inbetriebnahme beginnen zu können.

2. Entwurfsmuster

2.1 Singleton-Pattern

Problemstellung

Es leitet einen Entwickler eher dazu in einer objektorientierten Sprache schnell ein neues Objekt zu erstellen. Da dies dazu führen kann, dass von einer Klasse dann mehrere Objekte gleichzeitig erstellt / verwendet werden und dies so die Performance beeinträchtig, gibt es das Singleton-Pattern.

Ohne ein solches Singleton-Pattern müsste man in jeder Klasse eine neue Objekt-Instanz auf die benötigte Klasse erstellen. Da dies zu Laufzeit zu Einschränkungen führen kann, ist in unserer Applikation ein solches Pattern vorhanden.

Warum dieses Muster?

Wir haben während der Entwicklung gemerkt, dass viele «unnötige» Instanzen vorhanden sind. Mit dem Singleton-Pattern kann genau eine Instanz erstellt werden. Diese kann dann von jeder Klasse aufgerufen werden und gibt immer die gleiche eine Instanz zurück. Somit wird die Applikation performanter und stabiler.

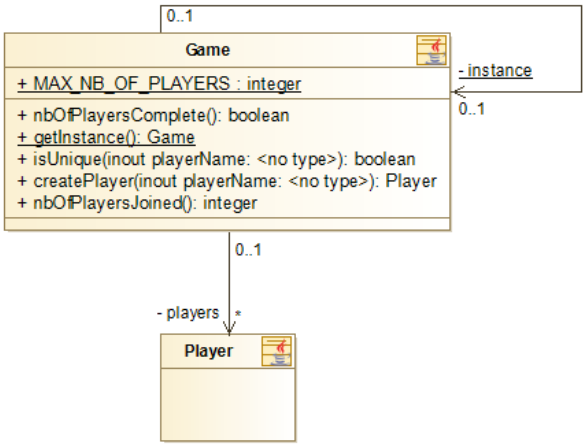


Abbildung 1

Pattern im Code

Static Instanz von Klasse ***«Game»***. Diese kann mit einem ***«Getter»*** angesprochen werden.

```
private static Game instance = new Game();

public static Game getInstance(){
    return instance;
}
```

```
Game game = Game.getInstance();
```

Abbildung 2

(Quelle: https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm)

2.2 Chain of Responsibility-Pattern

Problemstellung

Bei einer Applikation ist es zum Teil vorteilhaft ein gutes Logging zu besitzen. Da unsere Teilkomponenten einen Server verfasst, dient dieses Pattern für ein optimales Logging bei Verwendung.

Warum dieses Muster?

Während der Realisierung ist uns aufgefallen, dass noch ein wenig geloggt wird. Wir haben uns dann erkundigt wie man am besten die verschiedenen Ausgabemöglichkeiten managen könnte. Mit dem Pattern können wir sogenannte «Infos» loggen. Diese werden bei nützlichen Informationen z.B. Eine Nachricht wurde an Client XY gesendet, verwendet. Bei irgendwelchen Errors werden diese dann geloggt.

Im Klassendiagramm sind die verschiedenen Beziehungen und Aufrufe zu erkennen. Die schwarzen Striche symbolisieren Aufrufe in verschiedenen Klassen auf «***AbstractLogger***». Im Dispatcher werden die Logger «**instanziiert**». In den verschiedenen Controller werden Nachrichten geloggt. Dort wird dann ein Aufruf auf die Instanz in der Dispatcher-Klasse gemacht. Danach wird die Nachricht mittels «***logMessage()***» ausgegeben.

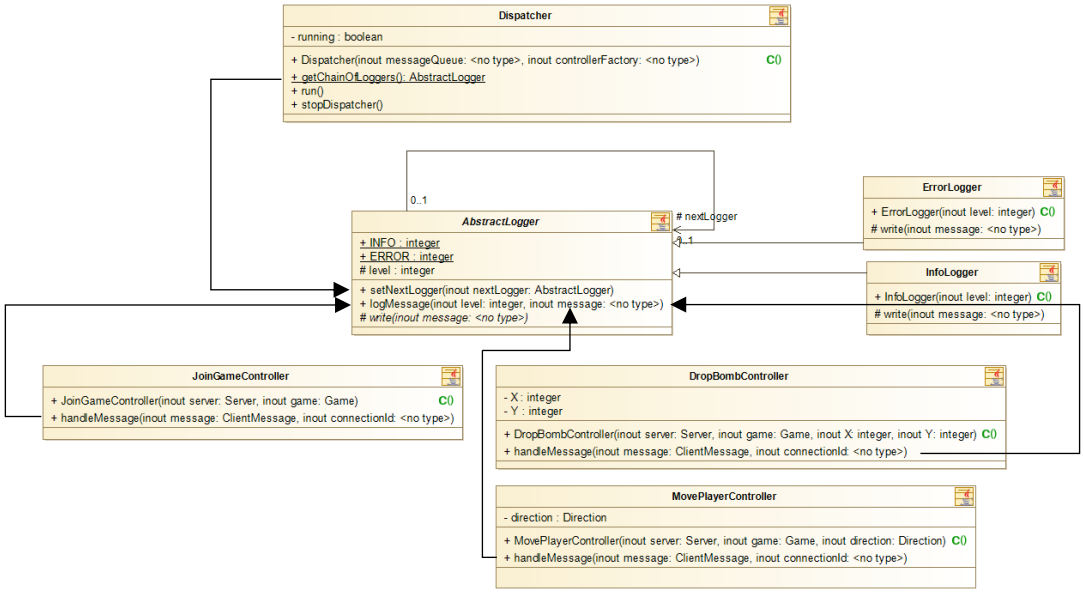


Abbildung 3

Pattern im Code

extends

extends

```
public abstract class AbstractLogger {
    public static final int INFO = 1;
    public static final int ERROR = 2;

    protected int level;

    protected AbstractLogger nextLogger;

    public void setNextLogger(AbstractLogger nextLogger) { this.nextLogger = nextLogger; }

    public void logMessage(int level, String message){
        if(this.level <= level){
            write(message);
        }
        if(nextLogger !=null){
            nextLogger.logMessage(level, message);
        }
    }

    abstract protected void write(String message);
```

```
public class ErrorLogger extends AbstractLogger {

    public ErrorLogger(int level) { this.level = level; }

    @Override
    protected void write(String message) {
        System.out.println("ERROR: " + message);
    }
}
```

```
public class InfoLogger extends AbstractLogger {

    public InfoLogger(int level) { this.level = level; }

    @Override
    protected void write(String message) { System.out.println("INFO: " + message); }
}
```

Aufruf

```
public static AbstractLogger getChainOfLoggers(){

    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
    AbstractLogger consoleLogger = new InfoLogger(AbstractLogger.INFO);

    errorLogger.setNextLogger(consoleLogger);

    return errorLogger;
}
```

```
@Override
public void handleMessage(ClientMessage message, String connectionId) {
    BombDropped bombDropped = new BombDropped(message.getPlayerName(), X, Y);
    Dispatcher.getChainOfLoggers().logMessage( level: 1, bombDropped.toString());
    server.broadcast(bombDropped);
}
```

Abbildung 4

(Quelle: https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm)