|  |
| --- |
| 4AHIT |
| S04: Chat |
| Chat für Schwerhörige |
|  |
| **Paul Kalauner, Seyyid Tiryaki** |
| **19.11.2014** |

|  |
| --- |
| Erstellt ein einfaches Chat-Programm für "Schwerhörige", mit dem Texte zwischen zwei Computern geschickt werden können. |

Inhaltsverzeichnis

[Aufgabenstellung/Requirementsanalyse 2](#_Toc404153240)

[Arbeitsaufteilung 3](#_Toc404153241)

[Zeitabschätzung & Zeitaufzeichnung 3](#_Toc404153242)

[Zeitschätzung 3](#_Toc404153243)

[Zeitaufzeichnung 3](#_Toc404153244)

[Gesamt 3](#_Toc404153245)

[Designüberlegung 4](#_Toc404153246)

[Klassendiagramm 5](#_Toc404153247)

[Implementierung 6](#_Toc404153248)

[Aufrufen des Programmes: 7](#_Toc404153249)

[Unittests 8](#_Toc404153250)

[Lessons Learned 9](#_Toc404153251)

[Quellen 12](#_Toc404153252)

# Aufgabenstellung/Requirementsanalyse

Erstellt ein einfaches Chat-Programm für "Schwerhörige", mit dem Texte zwischen zwei Computern geschickt werden können.  
  
Dabei soll jeder gesendete Text "geschrien" ankommen (d.h. ausschließlich in Großbuchstaben, lächelnd wird zu \*lol\*, Buchstaben werden verdoppelt, … - ihr dürft da kreativ sein)  
  
Zusätzlich sollen "böse" Wörter ausgefiltert und durch "$%&\*" ersetzt werden. Diese Funktionalität soll aber im Interface jederzeit aktiviert und deaktiviert werden können.  
  
Verwende dafür ausgiebig das Decorator-Pattern.

# Arbeitsaufteilung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Name** | **geschätzter Zeitaufwand** |
| UML-Diagramm | Paul Kalauner, Seyyid Tiryaki | 30 Minuten |
| Server/Client Implementierung | Seyyid Tiryaki | 90 Minuten |
| GUI Implementierung | Seyyid Tiryaki | 30 Minuten |
| MessageFilter Implementierung | Paul Kalauner | 40 Minuten |
| Testing | Paul Kalauner, Seyyid Tiryaki | 90 Minuten |
| Protokoll | Paul Kalauner, Seyyid Tiryaki | 90 Minuten |

# Zeitabschätzung & Zeitaufzeichnung

## Zeitschätzung

Schätzung: 6 Stunden, 10 Minuten

## Zeitaufzeichnung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Name** | **Zeitaufwand** |
| UML-Diagramm erstellt | Paul Kalauner, Seyyid Tiryaki | 25 Minuten |
| Server/Client implementiert | Seyyid Tiryaki | 60 Minuten |
| GUI | Seyyid Tiryaki | 30 Minuten |
| MessageFilter implementiert (Decorator) | Paul Kalauner | 30 Minuten |
| Testcases | Paul Kalauner, Seyyid Tiryaki | 90 Minuten |
| Codeoptimierung | Paul Kalauner, Seyyid Tiryaki | 30 Minuten |
| Protokoll | Paul Kalauner(70%), Seyyid Tiryaki(30%) | 60 Minuten |

## Gesamt

5 Stunden, 25 Minuten

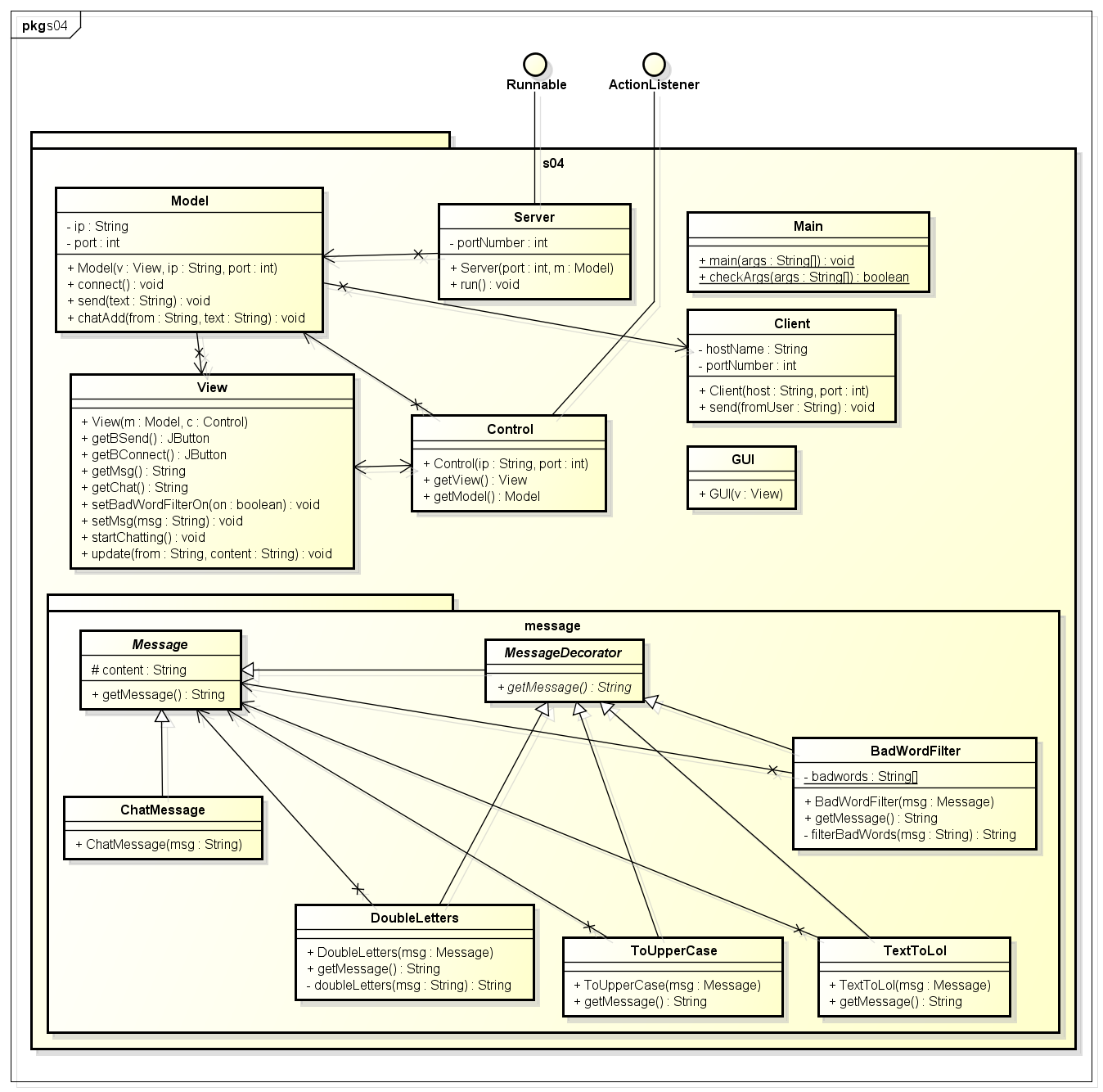
# Designüberlegung

Da es keine Rolle spielen sollen, welche Person die Server oder Client Rolle einnimmt, sind zum Entschluss gekommen, dass bei beiden Benutzern jeweils ein Server und ein Client in zwei verschiedenen Threads arbeiten. Dabei soll der Server Nachrichten empfangen, der Client ist für die Versendung von Nachrichten zuständig.

Für die GUI verwenden wir das MVC Pattern, wobei im Model die Verbindung hergestellt wird. In der View wird der Inhalt der Textfelder gesetzt und die Oberfläche erstellt, im Controller werden die Button-Klicks gehandhabt.

Das Decorator-Pattern wird für die Änderung (Verdoppelung der Buchstaben, Badword-Filter, usw.) der empfangenen Nachrichten benötigt. Der Badword-Filter stellt einen Spezialfall dar, da die Nachricht nur dann dekoriert wird, wenn dieser mittels einer Checkbox auf der GUI eingeschalten ist.

# Klassendiagramm



# Implementierung

**- Server/Client Kommunikation:**

Server und Client haben ihre eigenen Klassen, wobei der Server als eigener Thread ausgeführt wird

**- GUI:**

Bei der GUI haben wir das (uns bekannte) MVC Pattern verwendet:

Model: Server/Client Kommunikation

View: Grafische Oberfläche

Control: Steuerung des Programmes

**- Nachrichten-Modifizierung:**

Hierfür wurde das Decorator-Pattern verwendet. Nach dem Empfangen einer Nachricht, wird ein ChatMessage Objekt mit dem Inhalt der empfangenen Nachricht initialisiert. Danach wird dieses Objekt mit den verschiedenen Filtern dekoriert.

**- Main:**

In der Main Klasse werden die übergebenen Konsolenargumente überprüft.

Bei ungültigen Argumenten wird eine Fehlermeldung ausgegeben. Ansonsten wird das Programm ausgeführt.

# Aufrufen des Programmes:

- Starten über die Konsole mittels:

java –jar Filename.jar <IP des Partners> <Port des Partners>

- Wenn beide Benutzer das Programm gestartet haben, muss der Verbinden-Button gedrückt werden.

- Danach kann gechattet werden.

# Unittests

Die Unittests befinden sich im JAR-File und wurden mit Javadoc ausführlich kommentiert.

# Lessons Learned

**- Verwendung des Decorator Patterns [1]:**

Beispiel:

public abstract class Message {

**protected** String content;

**public** String getMessage() {

**return** **this**.content;

}

}

**public** **class** ChatMessage **extends** Message {

**public** ChatMessage(String msg) {

content = msg;

}

}

**public** **abstract** **class** MessageDecorator **extends** Message {

**public** **abstract** String getMessage();

}

**public** **class** TextToLol **extends** MessageDecorator {

**private** Message msg;

**public** TextToLol(Message msg) {

**this**.msg = msg;

}

@Override

**public** String getMessage() {

**return** msg.getMessage().replace(":)", "\*lol\*");

}

}

**- Client/Server Kommunikation [2]:**

Server:

Zuerst muss ein Server Socket erstellt und auf einen bestimmten Port gebunden werden:

ServerSocket serverSocket = **new** ServerSocket(portNumber);

Client-Verbindungen werden mit folgendem Befehl akzeptiert:

Socket clientSocket = serverSocket.accept();

Um vom Client lesen zu können, wird ein Reader benötigt:

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(clientSocket.getInputStream()));

Danach können Nachrichten vom Client empfangen werden:

String inputLine;

**while** ((inputLine = in.readLine()) != **null**) {

m.chatAdd("Partner", inputLine);

}

Client:

Verbindung mit Server herstellen:

Socket socket = **new** Socket(ip, port);

Um Nachrichten an den Server zu schicken, wird ein PrintStream benötigt:

PrintStream out = **new** PrintStream(socket.getOutputStream());

Danach können Nachrichten übermittelt werden:

out.println("Hallo");´

**- Ant Buildfile Beispiel:**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<project basedir="." default="compile" name="Chat">

<property environment="env" />

<property name="src.dir" value="src" />

<property name="build.dir" value="bin" />

<property name="lib.dir" value="libs" />

<property name="junit.output.dir" value="junit" />

<property name="debuglevel" value="source,lines,vars" />

<path id="JUnit4.libraryclasspath">

<pathelement location="${lib.dir}/junit.jar" />

<pathelement location="${lib.dir}/org.hamcrest.core\_1.3.0.v201303031735.jar" />

</path>

<path id="project.classpath">

<pathelement location="${lib.dir}/log4j-api-2.1.jar" />

<pathelement location="${lib.dir}/log4j-core-2.1.jar" />

<path refid="JUnit4.libraryclasspath" />

</path>

<target name="clean">

<delete dir="bin" />

</target>

<target name="compile">

<mkdir dir="${build.dir}" />

<javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="project.classpath" debug="true" deprecation="true" includeantruntime="false" />

</target>

</project>

# Quellen

[1]: PDF Decorator Pattern „Vererbst du noch oder designst du schon?“

[2]: Java Tutorials: All About Sockets. Verfügbar unter:

<https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>

[abgerufen am 12.11.2014]