

Tutorial für AIGS

Version für Eclipse



Institut für Wirtschaftsinformatik, Fachhochschule Nordwestschweiz

Autor	Raphael Stöckli raphael.stoeckli@students.fhnw.ch
Version	1.1e (Eclipse)
Datum	10.09.2014

Inhalt

1.	Funktionsprinzip AIGS.....	3
1.1.	Einleitung	3
1.2.	System.....	3
2.	Struktur eines Spiels	4
2.1.	Package-Struktur.....	4
2.2.	Vorbereitete Packages (Bibliotheken)	4
2.3.	Spiel-Packages.....	5
3.	Architektur.....	6
3.1.	Swing vs. JavaFX.....	6
3.2.	Assets	7
3.3.	Menschliche Gegenspieler vs. Computergegner (AI).....	7
4.	Tutorial – Schere, Stein, Papier	7
4.1.	Spielkonzept.....	7
4.2.	Voraussetzungen.....	8
4.3.	Schritt 1: Neues Projekt anlegen.....	9
4.4.	Schritt 2: Bibliotheken einbinden	10
4.5.	Schritt 3: Anlegen der Package-Struktur.....	11
4.6.	Schritt 4: Grundgerüst vorbereiten.....	12
4.7.	Schritt 5: main-Methode setzen und Client testen.....	14
4.8.	Schritt 6: Erstellen der gemeinsamen Komponenten (Commons)	15
4.9.	Schritt 6: Erstellen der Client-Logik.....	17
4.10.	Schritt 7: Erstellen der Server-Logik.....	23
4.11.	Schritt 8: JAR-Datei erzeugen, Anwendung verteilen und testen.....	27
5.	Hilfe bei der Entwicklung.....	31
5.1.	GUI-Editoren	32
5.2.	Schnellkompilierung.....	33
5.3.	Debuggen der Client-Logik.....	34
5.4.	Debuggen der Server-Logik.....	36
6.	Git-Repository.....	37
6.1.	Repository-Inhalt	37
6.2.	Software und Informationen zu Git	38
6.3.	Klonen des Git-Repository	38
6.4.	Erstellen eines Eclipse-Projektes aus dem Repository.....	39
7.	Troubleshooting	40
8.	Glossar	44

1. Funktionsprinzip AIGS

1.1. Einleitung

AIGS steht für „AI Game Server“ und ist ein von der FHNW entwickeltes System zur standardisierten Entwicklung von Spielen in Java. AIGS wurde für die Ausbildung konzipiert und soll Studenten helfen komplexere Java-Programme mit einem Praxisnutzen zu erstellen. Um den Einstieg zu erleichtern wurde der eigentliche Server und Grundgerüste zur Spieleprogrammierung bereits erstellt. Es ist möglich Spiele mit nur einem Spieler oder beliebig vielen Gegenspielern zu konzipieren. An Stelle von Gegenspielern können auch Computer-Gegner (AI) implementiert werden.

1.2. System

AIGS ist ein klassisches Server-Client-System. Verteilte Clients kommunizieren dabei mit einem zentralen Server. Beim Server handelt es sich um ein Programm (nicht Hardware), welches entweder auf einem speziellen Remote-Computer oder auf dem lokalen Rechner laufen kann. Der Client ist das Programm, welches das Spiel darstellt. Clients laufen jeweils auf den lokalen Rechnern. Zu Test- und Entwicklungszwecken kann der Server und der Client auf demselben Rechner betrieben werden. Die Serveradresse lautet dann [localhost](#). Mittels eines Ports (eine Art Softwarekanal) wird die Kommunikation zwischen Client und Server sichergestellt. Der Standardport bei AIGS ist [25123](#). Diese Nummer kann allerdings frei gewählt werden. Gültige Werte liegen zwischen 0 bis 65535. Es empfiehlt sich aber einen Port über der Nummer 1023 zu wählen, da darunterliegende Portnummern Konflikte mit Betriebssystemkomponenten und Diensten, wie FTP, WWW, SSH oder SMTP auslösen können.

Um ein Spiel auf AIGS zu nutzen, muss es sowohl auf dem Server, als auch auf den Clients vorhanden sein. AIGS wurde so konzipiert, dass dasselbe Programm als Informationsquelle für den Server, als auch als Client auf einem lokalen Rechner dienen kann. Bei der Entwicklung muss also nur eine Applikation erstellt werden. Die beim Kompilieren entstandene JAR-Datei kann anschliessend als Client-Applikation und als Server-Komponente verwendet werden. Dazu muss sie auf dem Server in ein spezielles Verzeichnis (Standardmässig in Unterverzeichnis [gamelibs](#)) abgelegt werden. Beim Starten des Servers ist das Spiel dann sofort verfügbar. Fehlt das Spiel auf dem Server, können sich die Clients nicht verbinden. Bei einer Änderung am Spiel müssen jeweils die JAR-Datei auf dem Server und auf den Clients ersetzt werden.

AIGS teilt seine Funktionalität in zwei Bestandteile auf:

- Client-Logik (Darstellung, Eingabe und Ausgabe)
- Server-Logik (Datenverarbeitung, Spielregeln und Verwaltung)

Auf der Client-Seite gibt es daher keine logische Verarbeitung der Spielvorgänge. Die Auswertung von Zügen und die Spielregeln allgemein werden nur auf dem Server verarbeitet. Im Gegenzug kann der Server die Art der Darstellung auf dem Client nicht. Er kümmert sich nur um die Datenverarbeitung und gibt Nachrichten an die Clients aus. Diese entscheiden anhand der Nachrichten, was darzustellen ist. Eine Ausnahme bildet das Setup. Über den Client können beim Starten des Spiels der GameMode (Multiplayer oder Singleplayer), sowie die etwaige Anzahl von Spielfeldern oder Objekten auf den Spielfeldern bestimmt werden. Dies ist allerdings nicht zur Laufzeit des Programms änderbar. Dazu muss das Programm neu kompiliert werden. Dies könnte allerdings mittels einer Serialisierung dieser Parameter in eine XML-Datei (wie bei den Settings) geändert werden.

2. Struktur eines Spiels

2.1. Package-Struktur

Das AIGS-System hat eine hierarchische Package-Struktur. Alle Packages beginnen mit [org.fhnw.aigs](#). Das dient zur besseren Organisation von Packages in einer Organisation. So könnten bei der FHNW zum Beispiel diverse Packages wie [org.fhnw.demos](#) oder [org.fhnw.dreamhome](#) gesammelt und Bibliotheken programmübergreifend verwendet werden. Ähnliche Package-Strukturen finden sich in Java-Programmen oder dem JDK immer wieder, wie beispielsweise [org.w3c.dom](#) oder [com.oracle.util](#).

Physisch – also die Dateien des Java-Projekts – sind die Packages in Unterordnern abgelegt. Im Ordner [org](#) befindet sich also ein Unterordner [fhnw](#), welcher wieder einen Unterordner [aigs](#) besitzt. IDEs wie Eclipse oder NetBeans organisieren diese Ordnerstruktur automatisch.

2.2. Vorbereitete Packages (Bibliotheken)

Das AIGS-System basiert auf einem vorbereiteten Grundgerüst. Ein Spiel muss also nicht von Grund auf neu programmiert werden. Mechanismen, wie die Darstellung des Spielfensters, die Kommunikation oder die Benutzerverwaltung bestehen bereits. Diese Mechanismen sind in zwei Dateien abgelegt, welche in jedem Spiel als JAR-Bibliotheken (Libs) eingebunden werden müssen. Es handelt sich um:

JAR-Bibliothek [AIGS_Commons.jar](#) mit den enthaltenen Packages:

- [org.fhnw.aigs.common](#)s
- [org.fhnw.aigs.common](#)s.communication

JAR-Bibliothek [AIGS_BaseClient.jar](#) (falls JavaFX verwendet wird), beziehungsweise [AIGS_SwingBaseClient.jar](#) (falls Swing verwendet wird) mit den enthaltenen Packages:

- [Assets.Basepatterns](#)
- [Assets.Fonts](#)
- [Assets.Stylesheets](#) (nur bei JavaFX)
- [org.fhnw.aigs.client.GUI](#)
- [org.fhnw.aigs.client.communication](#)
- [org.fhnw.aigs.client.gameHandling](#)

2.2.1. Packages der AIGS-Commons

Das Package [org.fhnw.aigs.common](#)s mit dem Unterpackage [communication](#) enthält Klassen, Variablen und Methoden, welche sowohl für die Clients, als auch für die Server-Applikation verwendet werden. Sie enthalten vor allem grundsätzliche Klassen, wie zum Beispiel [Game](#) oder [Player](#) um ein Spiel beziehungsweise ein Spieler als Objekt abzubilden. Ausserdem wird in den Klassen die Kommunikation zwischen Clients und Server definiert. Damit es zu keinen Fehlern in der Kommunikation kommt, ist es wichtig, dass sowohl Clients, als auch der Server die identischen Commons-Bibliotheken verwenden.

2.2.2. Packages der AIGS-Assets

Die Packages [Assets.BasePattern](#), [Assets.Fonts](#) und [Assets.Stylesheets](#) enthalten Ressourcen für die Client-Applikation. Der Einfachheit halber wurde hier auf die [org.fhnw.aigs](#)-Struktur verzichtet, da die Assets spezifisch für den AIGS-Client gedacht sind. In [BasePattern](#) befinden sich Bilddateien (Skins), welche für die Fensterdarstellung zuständig sind. In [Fonts](#) befinden sich Schriftarten zur Textgestaltung um Fenster. Das Unterpackage [Stylesheets](#) kommt nur in der Bibliothek [AIGS_BaseClient.jar](#) (für JavaFX) vor, da Swing kein CSS verarbeitet. Bei JavaFX dienen die CSS-Dateien im Package zur genaueren Steuerung der Darstellung des Fensters, ähnlich wie bei einer Webseite.

2.2.3. Packages des AIGS-Clients

Die Packages [org.fhnw.aigs.client.GUI](#), [org.fhnw.aigs.client.communication](#) und [org.fhnw.aigs.client.gameHandling](#) enthalten Klassen, Variablen und Methoden, welche zur Darstellung eines Spiels (Client) notwendig sind. Diese Klassen verweisen auf die Assets-Packages und die Commons-Packages. Hier werden zum Beispiel der Ladebildschirm, das Fenster für die Eingabe von Einstellungen (Settings) oder der grundsätzliche Fensteraufbau definiert.

2.3. Spiel-Packages

Ein Spiel besteht aus drei bis vier Java-Packages. Der Ausdruck *SPIELNAME* steht für den jeweiligen Namen eines Spiels (zum Beispiel „TicTacToe“ oder „Minesweeper“).

- [org.fhnw.aigs.SPIELNAME.client](#)
- [org.fhnw.aigs.SPIELNAME.common](#)s
- [org.fhnw.aigs.SPIELNAME.server](#)
- Optional: [Assets](#)

Die Client-, Commons- und Server-Packages arbeiten prinzipiell unabhängig voneinander und könnten sogar in eigenen Java-Projekten gepflegt werden. Um den Entwicklungsprozess zu vereinfachen wurden in den Beispielen alle Packages in einem Projekt zusammengefasst.

2.3.1. Client-Package

Im Package [org.fhnw.aigs.SPIELNAME.client](#) befinden sich Klassen, Variablen und Methoden, welche nur den Client des Spiels betreffen. Dazu gehören auch grafische Elemente und der Aufruf von lokalen Daten, wie die Settings. In diesem Package wird nicht die Spiellogik verarbeitet, sondern nur die Darstellung des Spiels im Client-Programm. Die Wichtigsten Klassen sind [SPIELNAMEClientGame](#), abgeleitet aus der Klasse [ClientGame](#) aus dem Package [org.fhnw.aigs.client.gameHandling](#), sowie [Main](#) beziehungsweise [SPIELNAMEMain](#). Diese beiden Klassen müssen zwingend vorkommen. Letztere ist dafür verantwortlich, dass der Client überhaupt startet und angezeigt wird. Sie enthält die [main](#)-Methode.

2.3.2. Server-Package

Im Package [org.fhnw.aigs.SPIELNAME.server](#) befinden sich Klassen, Variablen und Methoden, welche die Spiellogik verarbeiten. Die wichtigste Klasse ist dabei [GameLogic](#). Diese muss in jedem Spiel vorkommen und ist von der abstrakten Klasse [Game](#) aus dem Package [org.fhnw.aigs.common](#)s abgeleitet.

2.3.3. Commons-Package

Im Package [org.fhnw.aigs.SPIELNAME.common](#)s befinden sich nur Klassen, welche gemeinsam in Clients und Server verwendet werden. Es handelt sich dabei meistens um Klassen, welche von der Klasse [Message](#) aus dem Package [org.fhnw.aigs.common.communication](#) abgeleitet sind. Hier werden auch spielspezifische Klassen definiert. Zum Beispiel kann in einer Klasse definiert werden, welche Eigenschaften ein Spielfeld besitzt. Diese Eigenschaften können auch über Enumeratoren bestimmt werden, welche ebenfalls in diesem Package abgelegt werden können. Als Faustregel gilt: Alles, was sowohl der Server, als auch der Client wissen muss, gehört in dieses Package.

2.3.4. Assets-Package(s)

Im Package [Assets](#), beziehungsweise in dessen Unterpackages befinden sich Grafiken, Schriftarten, Stylesheets oder Sounds, welche auf dem Client zur Darstellung des Spiels benötigt werden. Ähnlich wie bei den AIGS-Assets wurde der Einfachheit halber auch hier auf die [org.fhnw.aigs.SPIELNAME](#)-Struktur verzichtet, da die Assets spezifisch für den Client und nur für ein bestimmtes Spiel gedacht sind. Hat ein Spiel keine Multimedia-Inhalte (zum Beispiel nur textbasiert) kann auf die Assets-Packages verzichtet werden.

3. Architektur

3.1. Swing vs. JavaFX

Grundsätzlich funktioniert der AIGS-Server unabhängig von der Darstellungstechnologie. Um die Entwicklung von Spielen zu vereinfachen, wurden – wie im letzten Kapitel beschrieben – zwei verschiedenen BaseClient-Bibliotheken erstellt. Eine basiert auf **Swing**, die andere auf **JavaFX**. Zwar können diese Technologien teilweise gemischt werden (es ist zum Beispiel möglich Swing-Dialoge in JavaFX aufzurufen), trotzdem muss man sich vor Beginn der Entwicklung auf eine Technologie festlegen und den entsprechenden BaseClient (JAR-Bibliothek) ins Projekt einbinden.

3.1.1. Swing

Swing ist in Java seit vielen Jahren etabliert und es gibt unterdessen einige gute GUI-Editoren zum Gestalten von grafischen Oberflächen. Das Anpassen von Inhalten zur Programmlaufzeit ist in Swing unproblematisch. Elemente können hinzugefügt, geändert oder gelöscht werden. Allerdings ist die Möglichkeit mit Swing etwas begrenzt. Mit Grafiken kann Swing nur bedingt umgehen. Diese müssen auf JButtons oder JLabel oder ähnliche Container gelegt werden. Um neuartige Elemente zu kreieren müssen relativ aufwändige Klassen und Methoden definiert werden. Swing ist zwar stabil, besitzt aber geringere Flexibilität als JavaFX und hat bei vielen Leuten das Image von „alten“ Programmoberflächen der späten 90er-Jahre. Für Programme oder Spiele ohne grössere Anforderungen an die Oberfläche ist Swing aber eine durchaus brauchbare Lösung.

3.1.2. JavaFX

JavaFX ist eine Neuentwicklung und erst seit kurzem offiziell im Umfang von Java SE enthalten. Es ist sehr flexibel und hat im Gegensatz zu Swing diverse moderne Möglichkeiten zur Manipulation von Fensterinhalten und Bildern. Elemente können wie bei einem HTML per CSS formatiert werden. Der Nachteil von JavaFX gegenüber Swing, ist der tendenziell etwas komplexere Aufbau, da zu Java noch CSS

hinzukommt. Ausserdem muss beim Anpassen von Inhalten während der Programmlaufzeit darauf geachtet werden, dass dies im richtigen Kontext – über sogenannte Invoking- oder runLater-Methoden – passiert, da es sonst zu einem Absturz des Programms kommen kann. Erschwerend kommt hinzu, dass die Fehlersuche in CSS relativ mühsam sein kann, da nicht korrekte Angaben, wie beim Browser, standardmässig ignoriert werden und die IDE oft nichts davon erfährt. Für die Spielentwicklung ist JavaFX aber tendenziell zu bevorzugen, da Spiele oft grafikorientiert sind.

3.2. Assets

Assets, also Bilder, Tondateien, 3D-Modell oder Texte sind ein wesentlicher Bestandteil von Spielen. Nichtsdestotrotz werden für AIGS-Spiele nicht zwingend Assets benötigt. Einige Assets zur Darstellung des Spielfensters sind bereits im BaseClient enthalten und ein textbasiertes Spiel könnte ohne Bilder, nur mit Hilfe von JavaFX- oder Swing-Elementen aufgebaut werden. Sollen Assets eingebaut werden, empfiehlt es sich diese in ein Package [Assets](#) zu legen. Bei der Spielprogrammierung ist dann auf den relativen Pfad zu achten. In den Demo-Spielen, wie „TicTacToe“ oder „Minesweeper“ ist im Code des Client-Packages zu sehen, wie die Bilder einzubinden sind. Bei JavaFX können zusätzlich noch CSS-Dateien zur Gestaltung eingebunden werden. Der CSS-Code könnte aber auch direkt in Java gepflegt werden, was allerdings nicht zu empfehlen ist. Tondateien oder 3D-Daten könnten ebenfalls eingebunden werden. Dazu werden aber weitere Bibliotheken für Java benötigt, welche die Verarbeitung ermöglichen.

3.3. Menschliche Gegenspieler vs. Computergegner (AI)

Bei der Entwicklung von Spielen kann bestimmt werden, wie viele menschliche Gegenspieler für ein Spiel benötigt werden, oder ob der Computer als Gegenspieler agieren soll. In der Regel wird beim Starten des Spiels bestimmt, ob dieses nur ein Spieler hat (Singleplayer) oder mehrere (Multiplayer). Die minimale Anzahl von Spielern, welche für eine Partie benötigt werden, wird aber auf der Server-Logik des Spiels hinterlegt. Wichtig zu erwähnen ist, dass ein Computergegenspieler (AI) programmiert werden muss. Die Logik wird nicht von AIGS bereitgestellt. AI-Logik muss in den Server-Komponenten des Spiels implementiert werden.

4. Tutorial – Schere, Stein, Papier

In den folgenden Kapiteln wird eine AIGS-Version von „Schere, Stein, Papier“ Schritt für Schritt aufgebaut. Die Ressourcen liegen im Git-Repository zu finden, damit alles nachvollzogen werden kann. Das Git-Repository wird in einem späteren Kapitel erklärt.

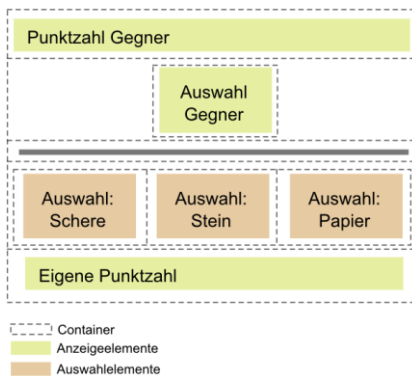
4.1. Spielkonzept

Das Grundprinzip von „Schere, Stein, Papier“ ist sehr einfach:

		Spieler 1		
		Schere	Stein	Papier
Spieler 2	Schere	Unentschieden	Stein gewinnt	Schere gewinnt
	Stein	Stein gewinnt	Unentschieden	Papier gewinnt
	Papier	Schere gewinnt	Papier gewinnt	Unentschieden

Theoretisch kann das Spiel mit mehr als zwei Personen gespielt werden. Um das Tutorial etwas einfacher zu halten gibt es aber nur zwei Spieler. Es wird auch keine KI-Logik implementiert. Damit die Kommunikation zwischen Client und Server etwas besser gezeigt werden kann, wird nicht nur ein Zug, sondern drei gemacht. Jeder gewonnene Zug bedeutet einen Punkt für den Gewinner. Ein Spieler kann am Ende des Spiels also zwischen 0 bis 3 Punkte haben. Der Spieler mit der höheren Punktzahl gewinnt das Spiel. Bei gleicher Punktzahl ist unentschieden. Nach dem dritten Zug wird also der Gewinner über alle Züge ermittelt. Danach wird das Spiel beendet.

Grafisch ist das Spiel nicht sehr komplex. Es müssen Auswahlfelder für jeweils Schere, Stein und Papier vorhanden sein und eine Anzeige was der Gegner gewählt hat. Diese Anzeige darf natürlich gegenseitig erst erfolgen, wenn beide Spiele ihre Züge gemacht haben. Weiter sollte es eine Anzeige über die Punktzahl und den Stand der Züge geben.



Da sowohl bei Swing, als auch bei JavaFX mit verschachtelten Containern gearbeitet werden muss, wird das Spiel auf Basis von verschachtelten Grids (ähnlich einem Tabellenlayout auf einer Webseite) konzipiert.

Die drei Auswahlfelder dienen auch als Anzeige der eigenen Auswahl. Zwischen der eigenen Auswahl und der des Gegners wird ein horizontaler Trenner zur besseren Darstellung eingefügt.

Abbildung 1: Layout des Spiels

Nach jedem Zug soll eine Meldung über den Gewinner erscheinen. Nach dem Quittieren dieser Meldung beginnt der nächste Zug.

Die gesamte Logik läuft in den Server-Komponenten ab. Einzig die richtige Darstellung von Bildern und Texten wird auf dem Client erledigt.

4.2. Voraussetzungen

Zur Entwicklung wird neben Java eine IDE benötigt. Es wird mit der folgenden Umgebung gearbeitet. Es muss nicht zwingend genau mit dieser Umgebung gearbeitet werden. Es erleichtert aber die Suche nach Fehlern, bei Problemen.

- JDK 8 (Update 11 oder neuer) → <http://www.oracle.com/technetwork/java/javase/downloads/>
- JRE 8 (Update 11 oder neuer, in JDK 8 enthalten)
- Eclipse 4.4 „Luna“ (Standard oder für Java Developers) → <https://www.eclipse.org/downloads/>

An Stelle von Eclipse kann auch NetBeans oder eine andere IDE verwendet werden. Im Tutorial werden aber Vorgänge und Bilder anhand von Eclipse erklärt. Diese weichen bei anderen IDEs in der Regel nicht gross ab. Für NetBeans gibt es aber ein identisches Tutorial.

Es werden während des Tutorials gewisse Grundkenntnisse über die IDE und ihre Bestandteile vorausgesetzt.

Beim Download der Komponenten ist darauf zu achten, dass alle Komponenten dieselbe Prozessorarchitektur haben. Entweder alles 32 Bit oder alles 64 Bit. Eine Mischung ist nicht zu empfehlen.

Das Tutorial wurde auf einem Windows-System erstellt. Natürlich kann das Ganze auch auf Mac- oder Linux-Rechnern realisiert werden, da sich weder Ordnerstruktur, noch etwas bei Java ändert. Einzig die Installation der oben genannten Komponenten variiert von System zu System.

Das Spiel wird im Tutorial mittels JavaFX aufgebaut. Daher sind folgende vorbereitete AIGS-Bibliotheken notwendig:

- [AIGS_Commons.jar](#)
- [AIGS_BaseClient.jar](#)

4.3. Schritt 1: Neues Projekt anlegen

4.3.1. Workspace definieren

Bevor ein Projekt angelegt werden kann, muss der Workspace in Eclipse definiert werden. Im Workspace werden danach alle AIGS-Projekte angelegt. Wird mit dem Git-Repository von AIGS gearbeitet (wird in einem späteren Kapitel erläutert), kann der Workspace auf das Verzeichnis des Repository ([ai-game-server](#)) gelegt werden.

In Eclipse wird standardmässig bei jedem Start nach dem Workspace gefragt. Wurde diese Abfrage ausgeschaltet, wird der Workspace gewählt durch: [File](#) → [Switch Workspace](#) → [Other...](#)

Über [Browse...](#) wird das Verzeichnis ausgewählt. Existiert es noch nicht, muss es ausserhalb von Eclipse (z.B. im Windows Explorer) angelegt werden.

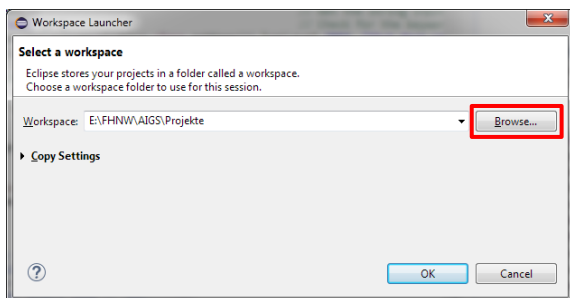


Abbildung 2: Auswahl des Workspace in Eclipse

Eclipse startet nach dem Auswählen des Workspace neu. Der Projekt-Explorer ist danach noch leer, sollten im gewählten Verzeichnis noch keine Eclipse-Projekte angelegt worden sein.

4.3.2. Projekt anlegen

In Eclipse wird ein neues Projekt über den Menüpunkt angelegt: [File](#) → [New](#) → [Java Project](#)

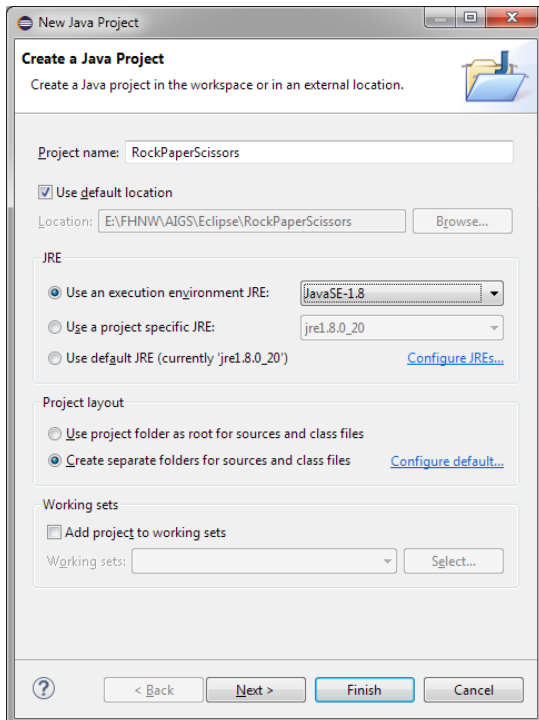


Abbildung 3: Projekt-Assistent in Eclipse

Auf dem ersten Bildschirm des Projekt-Assistenten wird der Name des Projekts neben *Project name* angegeben. Hier „RockPaperScissors“. Im Bereich *JRE* muss normalerweise nichts eingestellt werden. Das neueste, verfügbare JRE (hier *JavaSE-1.8*) wird standardmässig ausgewählt. Im Bereich *Project layout* muss die Option *Create separate folders for sources and class files* ausgewählt sein.

Auf dem nächsten Bildschirm könnten Source-Dateien, weitere Projekte und Bibliotheken definiert werden. Dies wird im Tutorial aber in den nächsten Schritten erledigt. Daher kann das Projekt jetzt mit einem Klick auf *Finish* angelegt werden.

Im Projekt-Explorer von Eclipse ist nun das leere Projekt zu sehen. Im Dateisystem wurden einige Dateien angelegt, welche nur zur Verwaltung des Projekts benötigt werden. Aktuell sind auch noch keine Bibliotheken verknüpft.

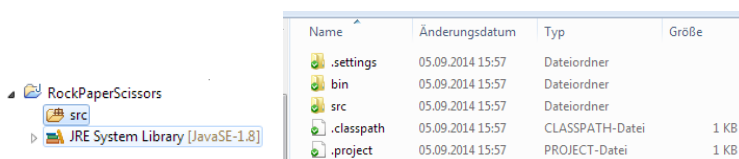


Abbildung 4: Ansicht Eclipse Projekt-Explorer (links), Windows-Explorer (rechts)

4.4. Schritt 2: Bibliotheken einbinden

Nun müssen die im Vorbereitungskapitel genannten JAR-Dateien ins Projekt eingebunden und verknüpft werden.

Als erstes wird ein neuer Ordner names *lib* im Projektverzeichnis angelegt. Dieses Verzeichnis darf sich aber nicht im *src*-Verzeichnis befinden, sondern muss im Stammverzeichnis des Projekts liegen.

In dieses Verzeichnis werden nun die zwei Dateien *AIGS_Commons.jar* und *AIGS_BaseClient.jar* kopiert, welche bei AIGS mitgeliefert wurden.

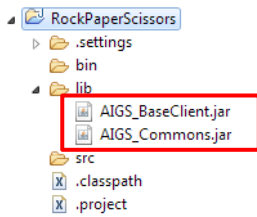


Abbildung 5: Dateiansicht über Navigator in Eclipse

Diese zwei Bibliotheken müssen nun in Eclipse zu den Libraries hinzugefügt werden. Dazu wird im Projekt-Explorer ein Rechtsklick auf das Projektsymbol (Offener Ordner mit einem „J“), beschriftete mit RockPaperScissors) gemacht und *Properties* ausgewählt. Anschliessend wird zur Kategorie *Java Build Path* gewechselt und zum Reiter *Libraries* gewechselt. Mit einem Klick auf *Add JARs...* erscheint ein Auswahlfenster, welches die Dateien des Projekts anzeigt. Als Pfad wird er Ordner *lib* im Projektverzeichnis gewählt. Dort sind nun beider JAR-Dateien zu sehen. Die Dateien werden mit Control-Taste beide ausgewählt und der Dialog bestätigt (OK). Die beiden Dateien sind nun unter *JARs and class folders on the build path* aufgelistet und im Projekt-Explorer unter *Referenced Libraries* (Der Eintrag wird unterhalb von *JRE System Library* automatisch angelegt) zu sehen. Mit einem Klick auf *OK* wird der Properties-Dialog geschlossen.

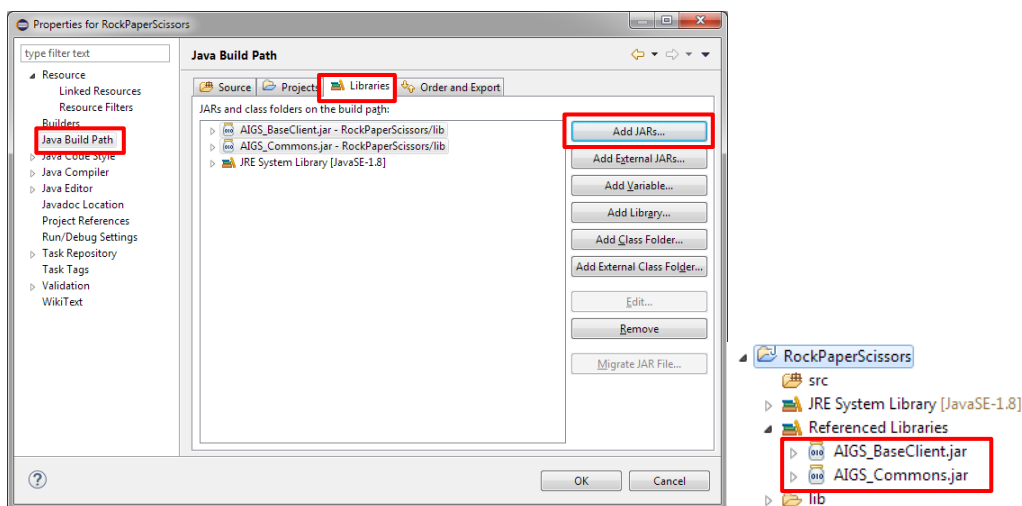


Abbildung 6: Properties-Dialog in Eclipse (links), Projekt-Explorer in Eclipse (rechts)

Das Projekt besitzt nun bereits alle Informationen um das Grundgerüst des AIGS-Spiels darzustellen. Allerdings fehlen noch Klassen und eine *main*-Methode zum Starten des Programms.

4.5. Schritt 3: Anlegen der Package-Struktur

Als nächstes werden die Packages für das Spiel vorbereitet. Wie Anfangs erwähnt werden zwingend die drei Packages *client*, *commons* und *server* benötigt. Zusätzlich wird ein Package für Assets angelegt, da das Spiel Grafiken enthält.

Dazu wird auf dem Projektsymbol ein Rechtsklick gemacht, dann: *New* → *Package*

Nacheinander können nun fünf Packages mit den nachfolgenden Namen angelegt werden. Die Namen werden in *Package Name* eingetragen. Mit einem Klick auf *Finish* wird ein Package angelegt. Die Packages sind:

- [org.fhnw.aigs.RockPaperScissors.client](#)
- [org.fhnw.aigs.RockPaperScissors.server](#)
- [org.fhnw.aigs.RockPaperScissors.common](#)
- [Assets.Images](#)
- [Assets.Stylesheets](#)

Hinweis:

Assets könnten auch in einem Package abgelegt werden. Die Trennung in [Images](#) und [Stylesheets](#) soll der Übersicht dienen, falls sehr viele Assets vorhanden sind.

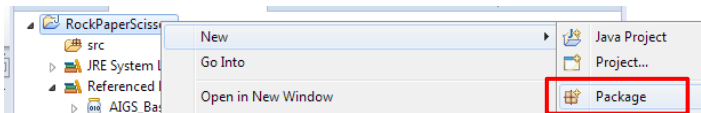


Abbildung 7: Anlegen eines neuen Packages in Eclipse

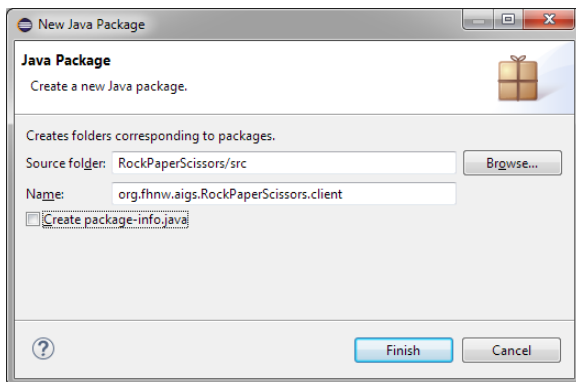
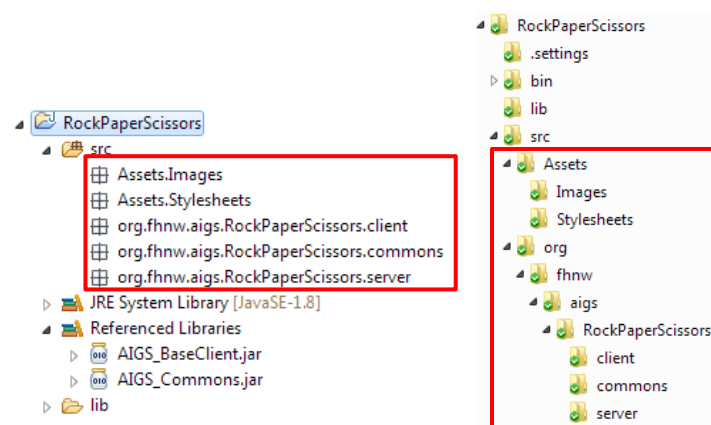


Abbildung 8: Dialog zum Anlegen von Packages in Eclipse

In Eclipse sind nach dem Anlegen die fünf neu erstellten Packages unter [Source Packages](#) zu sehen. Automatisch wurden hierarchisch gegliederte Verzeichnisse erstellt. Jeder Punkt im Package-Namen stellt eine Verzeichnisebene dar.



Hinweis:

Zusätzliche Ordner wie [.settings](#), [bin](#) oder [build](#) beeinflussen das Programm nicht. Sie werden von der IDE intern zur Projektverwaltung oder zur Kompilierung verwendet. In Eclipse wird zusätzlich der Ordner [lib](#) im Projekt-Explorer angezeigt. Die JAR-Dateien sind aber damit nicht automatisch ins Projekt eingebunden. Siehe dazu **Schritt 2**.

Abbildung 9: Ansicht im Projekt-Explorer in Eclipse (links), Ansicht im Windows-Explorer (rechts)

4.6. Schritt 4: Grundgerüst vorbereiten

Das Grundgerüst jedes Spiels in AIGS ist mehr oder weniger identisch. Daher können die wichtigsten Klassen aus anderen Projekten kopiert werden. Der Stand dieses Schritts wird dem Tutorial zusätzlich beigelegt. Somit können eigene Projekte schneller aufgesetzt werden. Einzig Spiel- und Package-Namen müssen jeweils angepasst werden.

Nach Abschluss dieses Schritts kann das Spiel das erste Mal getestet werden. Es enthält allerdings keinerlei Funktionalität.

Zum Anlegen einer neuen Klasse wird in Eclipse im Projekt-Explorer auf dem gewünschten Package (z.B. [org.fhnw.aigs.RockPaperScissors.client](#)) ein Rechtsklick gemacht und ausgewählt: [New → Class](#). Der Klassenname (Eingabe beim Feld [Name](#)) ist automatisch der Dateiname. Die Klasse [Main](#) hat also den Namen **Main.java**. In der Regel muss der [Modifier](#) nicht geändert werden. Er kann auf [public](#) stehen bleiben. Andere Angaben sind in der Regel auch nicht notwendig. Nach dem Bestätigen des Dialogs ([Finish](#)) wird die neue Klasse im Package angelegt. Sie enthält aber erst die Klassendefinition ([public class Main {}](#)).

Für das Grundgerüst wurden total drei Klassen angelegt:

- **Main.java**
 - Klasse: [Main](#) im Package [org.fhnw.aigs.RockPaperScissors.client](#)
 - Abgeleitet von Klasse [Application](#) (JavaFX)
 - Die Klasse wird vor allem für JavaFX benötigt. Falls Swing verwendet werden soll, wird eine gleichartige Klasse definiert, welche nur leicht von der JavaFX-Version abweicht. Sie beinhaltet die [main](#)-Methode
 - Bis auf ein paar weitere Zeilen Code (später) ändert sich an dieser Klasse im weiteren Verlauf des Tutorials nicht mehr viel
- **RockPaperScissorsClientGame.java**
 - Klasse: [RockPaperScissorsClientGame](#) im Package [org.fhnw.aigs.RockPaperScissors.client](#)
 - Abgeleitet von Klasse [ClientGame](#) aus Package [org.fhnw.aigs.client.*](#) (Bibliothek)
 - Diese Klasse ist später für die Verarbeitung von Nachrichten vom Server verantwortlich. Sie setzt die Befehle vom Server um und delegiert Veränderungen des Spielfensters (GUI)
 - Momentan beinhaltet sie noch keine Logik. Diese muss im weiteren Verlauf angepasst werden
 - Die Klasse beinhaltet noch (abstrakte) Methoden, welche implementiert werden müssen
- **GameLogic.java**
 - Klasse: [GameLogic](#) im Package [org.fhnw.aigs.RockPaperScissors.server](#)
 - Abgeleitet von Klasse [Game](#) aus Package [org.fhnw.aigs.commons.*](#) (Bibliothek)
 - Diese Klasse ist später für die eigentliche Spiellogik auf dem Server verantwortlich. In ihrer Instanz werden auch die Parameter einer Spielpartie gespeichert
 - Momentan beinhaltet sie noch keine Logik. Diese muss im weiteren Verlauf angepasst werden
 - Die Klasse beinhaltet noch (abstrakte) Methoden, welche implementiert werden müssen

Im Package [org.fhnw.aigs.RockPaperScissors.commons](#) wurden momentan noch keine Klassen angelegt. Die später dort platzierten Klassen variieren von Spiel zu Spiel. Um das Grundgerüst für ein anderes Spiel zu nutzen, muss [RockPaperScissorsClientGame](#) umbenannt, sowie alle Verweise auf die Klasse in [Main](#) angepasst werden. Ausserdem muss der Spielname in [Main](#) und [GameLogic](#) geändert werden.

Tipp:

Mit der **Refactoring**-Funktion **Rename** können Klassennamen und Variablenamen bequem und sicher innerhalb eines ganzen Projekts geändert werden.

4.7. Schritt 5: main-Methode setzen und Client testen

Bevor es mit der Logik des Spiels weitergeht, wird das Spiel das erste Mal gestartet um sicherzustellen, dass es zumindest clientseitig überhaupt läuft. Dazu muss die *main*-Methode definiert werden. Die Methode befindet sich in der Klasse *Main* (Main.java) im Package *org.fhnw.aigs.RockPaperScissors.client*. Sie ist allerdings leer, da JavaFX automatisch zur *start*-Methode wechselt. Java ist aber auf eine *main*-Methode angewiesen, da das Programm ansonsten gar nicht startet.

Bevor die *main*-Methode gesetzt werden kann, sollten alle Java-Dateien in einem frischen Projekt einmal gespeichert worden sein. Ansonsten kann es vorkommen, dass keine *main*-Methode gefunden wird.

Wird die *main*-methode in Eclipse nicht eingestellt, wird nach ihr (je nach Einstellung) bei jedem Start gefragt. Das kann mit einer **Run-Configuration** einmalig definiert werden.

Um ins Konfigurationsmenü zu kommen wird Im Menü gewählt: *Run* → *Run Configurations...*

Im Dialog wird der Eintrag *Java Application* ausgewählt und auf das Symbol für ‚New‘ geklickt. 

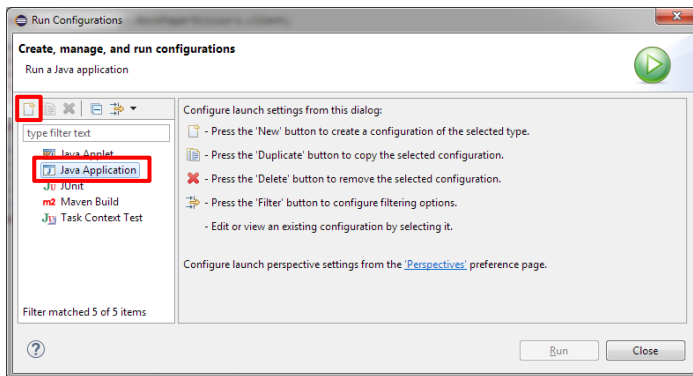


Abbildung 10: Konfigurationsmenü in Eclipse

Nun kann bei *Name* eine Bezeichnung für die Konfiguration (hier „Run RockPaperScissors“) eingegeben werden. Unter *Main class* wird mit einem Klick auf die Schaltfläche *Search...* der Auswahldialog für die *main*-Klasse geöffnet. Hier wird die Klasse *Main* aus dem Package *org.fhnw.aigs.RockPaperScissors.client* ausgewählt und mit *OK* bestätigt.

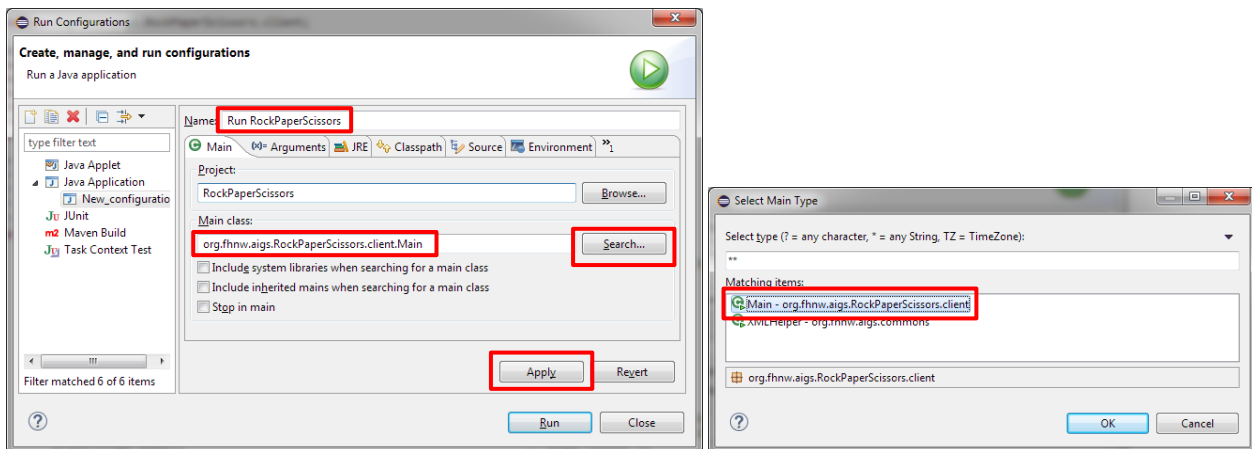


Abbildung 11: ausgefülltes Konfigurationsmenü in Eclipse (links), Auswahldialog für main-Klasse in Eclipse (rechts)

Die Einstellungen können nun über [Apply](#) gespeichert werden. Über [Close](#) wird der Dialog verlassen.

Nun ist das Projekt kompilierbar. In Eclipse wird die Kompilierung standardmässig automatisch erledigt. Das Programm muss also nur noch ausgeführt werden.

Das Programm kann nun also gestartet werden.

Entweder über: [Run](#) → [Run](#)

...oder über das Dreieck-Symbol.



Tipp:

Sollte es in Eclipse trotz automatischer Kompilierung Probleme geben, können mit dem Befehl [Project](#) → [Clean](#) alle Dateien gelöscht und neu kompiliert werden. Die automatische Kompilierung kann über den Menüpunkt [Project](#) → [Build Automatically](#) ein- oder ausgeschaltet werden.

Nun sollte als erstes das Settings-Fenster auftauchen und nach Benutzernamen, Identification Code (Passwort), Serveradresse und Port fragen. Für den Test können hier beliebige Werte eingetragen werden. Einzig der Port muss eine gültige Nummer sein. Nach einem Klick auf [Check and create configuration](#) (falls alles OK ist) wird das Settings-Fenster geschlossen und der Ladebildschirm des Spiels ist zu sehen. Allerdings wird die Meldung „Connecting...“ nicht verschwinden, da weder ein Server läuft, noch eine Startaktion definiert wurde. Das Spiel kann nun wie ein normales Windows/Mac/Linux-Programm beendet werden.

Die Settings wurden als XML-Datei **ClientConfig.xml** im Projektverzeichnis angelegt. Die Datei kann gefahrlos gelöscht werden. Beim nächsten Programmstart würde sich dann wieder das Settings-Fenster öffnen. Um die Settings im Spiel zu bearbeiten, kann auf das Zahnrad-Symbol geklickt werden.



Die Änderungen werden aber erst beim nächsten Start des Programms aktiv.

Die Benutzernamen und Identification Codes müssen separat in einer Datei im Serververzeichnis gepflegt werden, um sich später mit dem Server verbinden zu können.

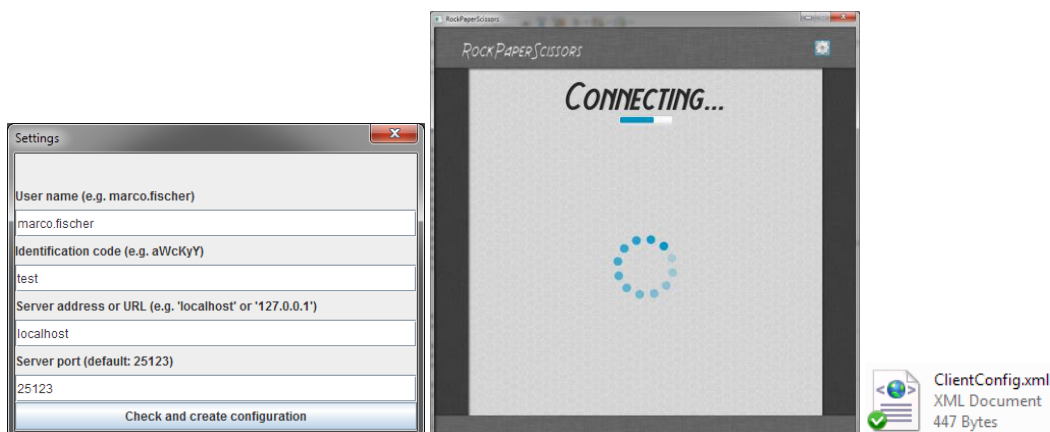


Abbildung 12: Settings-Fenster (links), Spielfenster (Mitte), Angelegte XML-Datei (rechts)

4.8. Schritt 6: Erstellen der gemeinsamen Komponenten (Commons)

Es gibt zwar kein Patentrezept ob die gemeinsamen Komponenten, die Server- oder Client-Komponenten zuerst erstellt werden sollen. In der Praxis wird dies oft gemeinsam erledigt. Oft entstehen während den Arbeiten an der Client- oder Server-Logik neue Bedürfnisse in den Commons, welche dann vor zu umgesetzt werden. Aus praktischen Gründen der Dokumentation werden hier die Commons zuerst erstellt, da diese keine Abhängigkeiten zu den anderen zwei Packages des Spiels haben.

Im Package [org.fhnw.aigs.RockPaperScissors.common](#)s werden im Folgenden zwei **Enumeratoren** und drei **Klassen** angelegt. Diese werden später sowohl von den Clients, als auch vom Server verwendet. Die Klassen enthalten keine Funktionen, sondern dienen als reine Informationsträger. Der Zweck der Enumeratoren wird im nächsten Abschnitt erläutert.

4.8.1. Enumerator: GameState

Die Datei **GameState.java** enthält den Enumerator [GameState](#). Dabei handelt es sich um eine Art Auswahlliste von Konstanten, welche als Texte dargestellt werden, aber intern wie Nummern behandelt werden. Somit gibt es, im Gegensatz zur Benutzung von Integern oder Strings (als Flags), keine Missverständnisse wenn verschieden Zustände abgebildet werden sollen.

[GameState](#) soll das Resultat eines Zuges für einen Spieler abbilden. Mögliche Werte sind also gewonnen ([Win](#)), verloren ([Lose](#)), unentschieden ([Draw](#)) oder nicht definiert ([None](#)), für alle anderen Fälle. Wichtig zu erwähnen ist noch die Zeile mit dem Befehl [@XmlElement](#). Es handelt sich dabei um eine sogenannte **Annotation**. Dieser Befehl aus dem Package [javax.xml.bind.annotation.XmlRootElement](#) stellt sicher, dass der Enumerator in XML umgewandelt werden kann. Dies ist wichtig, da AIGS zwischen Server und Clients mit XML kommuniziert. Fehlt der Befehl, kann es zu einem Absturz des Spiels kommen. Das Attribut [name](#) des Befehls entspricht immer dem Namen der Klasse oder des Enumerators, hier also [GameState](#).

4.8.2. Enumerator: RockPaperScissorsSymbol

Die Datei **RockPaperScissorsSymbol.java** enthält den Enumerator [RockPaperScissorsSymbol](#). Er soll das Gewählte Symbol eines Zuges abbilden. Also Schere ([Scissors](#)), Stein ([Rock](#)), Papier ([Paper](#)) oder nicht definiert ([None](#)), für alle anderen Fälle.

Der Enumerator enthält dieselben Elemente, wie [GameState](#) und muss auch als XML darstellbar sein. Daher auch wieder der Befehl [@XmlElement](#).

4.8.3. Klasse : RockPaperScissorsParticipantsMessage

Die Datei **RockPaperScissorsParticipantsMessage.java** enthält die Klasse [RockPaperScissorsParticipantsMessage](#). Diese ist von [Message](#) aus dem Package [org.fhnw.aigs.common.communication.Message](#) abgeleitet. Auch sie muss in XML darstellbar sein. Bei den get-Methoden der Variablen ist jeweils ein Befehl [@XmlElement](#) (aus Package [javax.xml.bind.annotation.XmlElement](#)) vorangestellt. Dieser hat denselben Zweck wie [@XmlElement](#) für die ganze Klasse, jedoch für die eine einzelne Variable. Zur Definition reicht es den Befehl nur bei den get-Methode der einzelnen Variablen anzugeben. Wichtig ist, dass die [name](#)-Attribute der Annotationen bei den einzelnen get-Methoden eindeutig sind. Wird ein Name mehr als einmal verwendet, kann es zu einem Absturz oder unvorhersehbarem Programmverhalten kommen.

Der Zweck der Klasse ist, am Spielanfang beiden Spielern den Namen des Gegenspielers anzuzeigen. Ein Objekt dieser Klasse beinhaltet zwei Variablen ([playerOne](#) und [playerTwo](#) vom Typ [String](#)). Die Erste ist der Name des ersten Spielers, die zweite, der Name des zweiten Spielers. Die Reihenfolge ist unerheblich, da auf den Clients ermittelt wird, wer der zwei Namen der Gegenspieler ist. Die Nachricht wird gleichzeitig an alle Spieler geschickt. Neben dem parameterlosen Konstruktor gibt es einen weiteren Konstruktor, welcher die beiden Namen annimmt.

4.8.4. Klasse: *RockPaperScissorsSelectionMessage*

Die Datei **RockPaperScissorsSelectionMessage.java** enthält die Klasse *RockPaperScissorsSelectionMessage*. Diese ist von *Message* aus dem Package *org.fhnw.aigs.commons.communication.Message* abgeleitet. Auch sie muss in XML darstellbar sein.

Der Zweck dieser Klasse ist, dem Server Mitzuteilen, welches Symbol (Schere, Stein oder Papier) ein Spieler ausgewählt hat. Nachrichten dieses Typs werden also nur von Clients in Richtung Server gesendet. Die Klasse besitzt eine Variable. Und zwar das gewählte Symbol vom Typ *RockPaperScissorsSymbol* (Enumerator). Dieses kann mit dem zweiten Konstruktor übergeben werden.

4.8.5. Klasse: *RockPaperScissorsResultMessage*

Die Datei **RockPaperScissorsResultMessage.java** enthält die Klasse *RockPaperScissorsResultMessage*. Diese ist von *Message* aus dem Package *org.fhnw.aigs.commons.communication.Message* abgeleitet. Auch sie muss in XML darstellbar sein. Es handelt sich um die umfangreichste Nachrichten-Klasse des Spiels.

Ihr Zweck ist den Clients mitzuteilen, wie das Resultat eines Zuges ist. Ein Zug ist beendet, nachdem beide Spieler ihre Auswahl getroffen haben. Das heisst: beide haben eine Nachricht vom Typ *RockPaperScissorsSelectionMessage* an den Server gesendet. Danach sendet der Server Nachrichten mit dem Resultat an die Clients. Für jeden Spieler muss eine eigene Nachricht gesendet werden. Die Klasse besitzt neun **Variablen**:

- *mySymbol* (vom Typ *RockPaperScissorsSymbol*) zeigt an, wie das gewählte Symbol des angesprochenen Spielers ist (also der Spieler, welcher die Nachricht erhält)
- *opponentSymbol* (vom Typ *RockPaperScissorsSymbol*) zeigt an, wie das gewählte Symbol des Gegners Spielers ist
- *turn* (vom Typ *int*) zeigt die Nummer des aktuellen Zuges an (1-3)
- *myPoints* (vom Typ *int*) zeigt die Punktzahl des angesprochenen Spielers über alle Züge an (0-3)
- *opponentPoints* (vom Typ *int*) zeigt die Punktzahl des Gegners über alle Züge an (0-3)
- *opponentName* (vom Typ *String*) zeigt den Namen des Gegenspielers an
- *isLastTurn* (vom Typ *boolean*) zeigt an, ob es sich um den letzten Zug handelt
- *turnMessage* (vom Typ *String*) zeigt eine Nachricht an, welche auf dem Client ausgegeben werden soll
- *myState* (vom Typ *GameState*) zeigt an, wie der Status des angesprochenen Spielers für den Zug ist (gewonnen, verloren oder unentschieden)

Neben dem parameterlosen, gibt es noch eine Konstruktor, welcher alle Parameter annimmt.

4.9. Schritt 6: Erstellen der Client-Logik

Als nächstes soll die clientseitige Logik, welche vor allem für die Darstellung zuständig ist, erstellt werden. Dazu müssen ein paar Klassen definiert, Assets erstellt, eingebunden und alles mit dem Grundgerüst verknüpft werden.

Tipp:

Dateien können entweder per Drag&Drop in Eclipse ins Package hineingezogen oder direkt im Dateiverzeichnis abgelegt werden. Im ersten Fall wird gefragt, ob die Dateien kopiert oder verknüpft werden sollen. Im zweiten Fall nimmt Eclipse die Dateien automatisch ins Package auf.

4.9.1. Assets

Als Assets werden in einem ersten Schritt sechs Bilder benötigt. Es werden jeweils drei Bilder für Schere, Stein und Papier, sowie ein Bild mit dem leeren Hintergrund, ein Bild mit einem Haken (gewonnen) und eins mit einem Kreuz (verloren) benötigt. Für das Tutorial wurden Bilder aus dem Internet mit Public Domain-Lizenz verwendet und mit einem Editor (Photoshop oder Gimp) bearbeitet. Wichtig ist vor allem, dass alle Bilder dieselben Dimensionen haben um Verzerrungen zu vermeiden. Ausserdem sollten alle Bilder nicht zu gross (Dateigrösse) in einem gängigen Format vorliegen. Für das Tutorial wurden die Bilder mit einer Auflösung von 150 x 150 Pixel im PNG-Format (24 Bit) abgespeichert. Die zwei Bilder mit dem Kreuz und Haken wurden zusätzlich mit Transparenz (PNG 32 Bit) abgespeichert, da diese später über die anderen Bilder gelegt werden sollen.

Alle Bilder werden im Package `Assets.Images` abgelegt. Ihre Namen werden im Anschluss von der Client-Logik referenziert (siehe Abschnitt `RockPaperScissorsBoardPane`).



Abbildung 13: Verwendete Bilder als Assets

Im Package `Assets.Stylesheets` wird eine CSS-Datei mit dem Namen **RockPaperScissors.css** abgelegt. In ihr befinden sich unter anderem Informationen, wie ein Feld (Schere, Stein, Papier oder leer) dargestellt werden soll. Unter anderem auch, wie sich ein Feld verändern soll, wenn darauf geklickt wird. Die meisten CSS-Definitionen sind speziell auf JavaFX abgestimmt. Dies sieht man am Ausdruck „-fx-...“. Es können aber durchaus auch standardisierte CSS-Definitionen gemäss W3C verwendet werden. Die auf JavaFX zugeschnittenen Befehle sind auf der folgenden Seite dokumentiert:

<http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>

Sollen also Veränderungen an der Darstellung der einzelnen Felder vorgenommen werden, kann dies über eine Anpassung der CSS-Klassen (Gekennzeichnet durch vorangestellten Punkt → Beispiel: `.separator`) in der CSS-Datei erfolgen. Das Programm muss bei einer Änderung aber trotzdem neu kompiliert werden, da die Assets-Packages mitkompiliert werden.

4.9.2. Klasse: Main

Die Datei **Main.java** enthält die bereits im Grundgerüst erwähnte Klasse **Main**, welche auch die `main`-Methode des Clients enthält.

Für das Tutorial wurden zwei Konstanten definiert. Ausserdem enthält die Klasse zwei vordefinierte Methoden. Bei den **Konstanten** handelt es sich um folgendes:

- **GAMENAME** (vom Typ `String`): In ihr ist der Name des Spiels abgespeichert. Wird dieser im Client irgendwo benötigt, kann er mittels `Main.GAMENAME` ermittelt werden.
- **GAMEMODE** (vom Typ `GameMode`): Definiert, ob das Spiel Singleplayer oder Multiplayer ist. Wird der Wert im Client irgendwo benötigt, kann er mittels `Main.GAMEMODE` ermittelt werden.

Die zwei **Methoden** sind:

- **start**: Die Methode überschreibt die Methode der Super-Klasse (*Application*), zu sehen an der Annotation *@Override*. Sie entspricht in JavaFX in etwa der main-Methode und wird ausgeführt, sobald das Programm (Client) startet. Der Ablauf sieht wie folgt aus:
 - Eine Instanz von *BaseGameWindow* aus dem Package *org.fhnw.aigs.client.GUI.BaseGameWindow* wird instanziiert und der Spielname übergeben. Hiermit wird das Hauptfenster definiert.
 - Die CSS-Datei aus dem Package *Assets.Stylesheets* wird in das Programm eingebunden.
 - Der Titel des Spiels wird im Hauptfenster über Methode *setTitle* gesetzt.
 - Das Hauptfenster wird mit der Methode *show* eingeblendet.
 - Eine Neue Instanz der Klasse *RockPaperScissorsClientGame* wird gebildet und Spielname und Spielmodus übergeben. Hierbei handelt es sich um die Clientseitige Logik des Spiels. Die Klasse wird später erklärt.
 - Eine neue Instanz der Klasse *RockPaperScissorsBoard* wird gebildet und im Hauptfenster eingebunden. Hierbei handelt es sich um das eigentliche Spielfeld. Die Klasse wird später erklärt.
 - Der Ladebildschirm (neue Instanz der Klasse *LoadingWindow* aus dem AIGS BaseClient) wird mit der Methode *setOverlay* eingeblendet.
 - Die Settings des Clients werden geladen. Sind keine Settings vorhanden, wird das Settings-Fenster angezeigt.
 - Mit den Informationen aus den Settings wird versucht den Server zu erreichen. Dazu dient die Methode *setCredentials*. Sind die Anmeldeinformationen falsch, wird eine Fehlermeldung vom Server zurückgegeben. Wird der Server nicht erreicht, ist dies in den Logfiles des Clients zu sehen. Es passiert aber vordergründig nichts.
 - Ein neuer Thread für die Kommunikation wird erstellt und gestartet. Die GUI und die Kommunikation laufen nun in getrennten Threads. Damit wird verhindert, dass die Programmoberfläche einfriert, wenn keine Nachricht ankommt. Das Programm ist nun betriebsbereit und wartet nun auf eine Meldung des Servers.
- **main**: Die *main*-Methode des Client-Programms
 - Der Befehl (Methode) *launch* startet das JavaFX-Programm und indirekt auch die Methode *start*, welche zuvor erklärt wurde. Weitere Befehle sind hier nicht mehr notwendig.

4.9.3. Klasse: *RockPaperScissorsBoard*

Die Datei *RockPaperScissorsBoard.java* enthält die Klasse *RockPaperScissorsBoard*. Sie ist zur Darstellung des Spielfeldes verantwortlich. Eine Instanz dieser Klasse wird im Hauptfenster eingebunden. Dies passiert in der Methode *start* der Klasse *Main*.

Die Klasse, ist wie jedes Spielfeld in AIGS von der Klasse *GridPane* (aus JavaFX) abgeleitet. Es sind diverse Variablen und Methoden definiert. Sehr wichtig sind auch die Aktionen im Konstruktor.

Als **Variablen** sind definiert:

- **clientGame** (vom Typ *RockPaperScissorsClientGame*): Dient zum einfacheren Zugriff auf die clientseitige Logik.

Hinweis:

Alternativ zu einem *GridPane*, bietet der AIGS BaseClient die Klasse *BaseBoard*. Von dieser Klasse kann abgeleitet werden. *BaseBoard* stellt klassische Spielfelder, wie Schach oder Mühle dar. In den Demo-Spielen „TicTacToe“ und „Minesweeper“ kann die Funktionsweise von *BaseBoard* angeschaut werden.

- **fieldsPane** (vom Typ **GridPane**): Ist eine Container, in welchem die drei Auswahlfelder (Schere, Stein und Papier) eingebunden werden. Der Container wird zur Darstellung benötigt.
- **opponentPane** (vom Typ **GridPane**): Ist ebenfalls ein Container, in welchem die Auswahl des Gegners angezeigt wird. Der Container enthält aber nur ein einziges Feld.
- **opponentLabel** (vom Typ **Label**): Dient zur Darstellung des Gegnernamens und seiner Punktzahl
- **myLabel** (vom Typ **Label**): Dient zur Darstellung des eigenen Namens (Ich) und der eigenen Punktzahl
- **separator** (vom Typ **Separator**): Dient als visuelles Stilmittel. Zwischen der Auswahl des Gegners und der eigenen Auswahl soll ein Trenner dargestellt werden.
- **opponentField** (vom Typ **RockPaperScissorsBoardPane**): Dient zur Darstellung der Auswahl des Gegners.
- **rockField** (vom Typ **RockPaperScissorsBoardPane**): Dient zur Darstellung des eigenen Auswahlfeldes für Stein.
- **paperField** (vom Typ **RockPaperScissorsBoardPane**): Dient zur Darstellung des eigenen Auswahlfeldes für Papier.
- **scissorsField** (vom Typ **RockPaperScissorsBoardPane**): Dient zur Darstellung des eigenen Auswahlfeldes für Schere.

Es gibt einen Konstruktor in der Klasse, welcher eine Instanz von **ClientGame** als Parameter annimmt. Dieser Konstruktor wird aus der Klasse **Main** in der Methode **start** aufgerufen. Dort ist auch das zentrale **ClientGame**-Objekt vorhanden. Der **Konstruktor** erledigt folgende Schritte:

- das **ClientGame**-Objekt wird aus den übergebenen Parametern gesetzt
- Von allen oben genannten Variablen werden mittels des **new**-Befehls neue Instanzen dieser Objekte erstellt.
 - Bei Variablen vom Typ **Label** wird ein Text beim Instanziiieren definiert
 - Bei der Variable vom Typ **Separator** wird **HORIZONTAL** (Enumerator-Wert) als Argument übergeben
- Bei den Objekten vom Typ **RockPaperScissorsBoardPane** wird mit der **setSymbol**-Methode bestimmt, welches Symbol angezeigt werden soll
- Den Objekten vom Typ **Label** und **Separator** wird mittels der Methode **getStyleClass().add** eine Stylesheet-Klasse zugewiesen, welche in **RockPaperScissors.css** definiert wurde. Andere Elemente könnten bei Bedarf auf dieselbe Weise formatiert werden
- Die Elemente werden mittels der Methode **add** ineinander verschachtelt:
 - **opponentField** wird in **opponentPane** hinzugefügt. Mittels der Methode **setAlignment** und dem Argument **CENTER** (Enumerator-Wert) wird definiert, dass das Feld zentriert werden soll
 - **rockField**, **paperField** und **scissorsField** werden in **fieldsPane** hinzugefügt. Mit den Methoden **vgapProperty().set**, **hgapProperty().set** und **setPadding** werden Abstände zwischen den Elementen definiert
 - **opponentLabel**, **opponentPane**, **separator**, **fieldspane** und **myLabel** werden dem Fenster (**this**) hinzugefügt. Mit den Methoden **vgapProperty().set** und **setAlignment** werden wie oben bereits beschrieben Abstände und Zentrierung für alles Fensterelemente bestimmt
- Drei Eventhandler, ausgelöst von der Action **setOnMouseClicked** werden für **rockField**, **paperField** und **scissorsField** definiert
 - Beim Anklicken wird jeweils zuerst die Methode **setSelectedPane** mit dem entsprechenden Feld aufgerufen
 - Eine Neue Message vom Typ **RockPaperScissors-SelectionMessage** wird erstellt und das jeweilige Feld als Argument übergeben

- Über das *clientGame*-Objekt wird die Methode *sendMessageToServer* aufgerufen und die zuvor erstellte Nachricht gesendet

Nach diesen Aktionen ist das Spielfeld komplett definiert und darstellbar. Wichtig ist vor allem, dass die EventHandler korrekt definiert worden sind. Die Methode *setSelectedPane* muss also bereits erstellt worden sein, oder zumindest als leere Hülle vorhanden sein.

In der Klasse sind sechs weitere **Methoden** definiert:

- ***setSelectedPane***: Diese Methode stellt das angeklickte Symbol (Scher, Stein, Papier) beim Anklicken über den Austausch der Style-Klasse anders dar (Highlighting). Alle Änderungen an den JavaFX-Elementen müssen zur Laufzeit in einer Methode *Platform.runLater* aufgerufen werden um Abstürze zu verhindern. Die eigentlichen Befehle befinden sich verschachtelt in einer *run*-Methode.
 - Zuerst werden alle Style-Klasse von den drei Feldern *rockField*, *paperField* und *scissorsField* entfernt
 - Danach wird die Style-Klasse für das Highlighting auf das gewünschte Feld mittels der Methode *getStyleClass().add* angewendet
 - Wurde weder *Rock*, *Paper*, noch *Scissors* übergeben (also *None*) wird kein Feld als selektiert angezeigt (kein Highlighting)
- ***manipulateGUI***: Diese Methode ruft einerseits eine Überladung der Methode, andererseits die Methode *manipulateText* auf
- ***manipulateGUI*** (Überladung): Diese Methode leitet Veränderungen der Bilder auf dem Spielfeld ein. Die eigentlichen Veränderungen findet aber in der Klasse *RockPaperScissorsBoardPane* in der Methode *setSymbol* statt
 - Es wird jeweils unterschieden, ob der Spieler gewonnen oder verloren hat, oder ob unentschieden steht
 - Je nach ausgewähltem Feld wird entweder ein Haken (für gewonnen), ein Kreuz (für verloren) oder nichts (für unentschieden) über das Bild gelegt
- ***manipulateText***: In dieser Methode wird der Text der zwei Statuslabel *myLabel* und *opponentLabel* angepasst. Die Aktionen für die Anpassung müssen wie bereits weiter oben beschrieben in der Methode *Platform.runLater* verschachtelt werden.
- ***manipulateHeader***: Diese Methode ändert den Statustext des Spielfensters (rechts neben Spielname). Dabei werden Methoden aus dem AIGS BaseClient aufgerufen
- ***nextTurn***: Diese Methode bereitet das Spielfeld auf den Nächsten Zug oder die Bekanntgabe des Siegers vor
 - Das ganze Programm wird mittels eines *JOptionPane* angehalten. Erst nach einem Klick auf *OK* fährt die Methode fort
 - Alle Auswahlen, sowie Kreuz und Haken werden mit den Methoden *manipulateGUI* und *setSelectedPane* entfernt
 - Der Statustext des Programms wird angepasst (warten auf nächsten Zug)

Hinweis:

Werden in JavaFX Elemente ausserhalb des GUI-Threads geändert, kann das zu einem Absturz führen. Ein Hinweis darauf ist eine **IllegalStateException** mit dem Text „**Not on FX application thread**“. Um dies zu verhindern wird die Methode *Platform.runLater* verwendet. Alle kritischen Aktionen werden darin in einer verschachtelten *run*-Methode gekapselt.

4.9.4. Klasse: *RockPaperScissorsBoardPane*

Die Datei **RockPaperScissorsBoardPane.java** enthält die Klasse *RockPaperScissorsBoardPane*. Sie ist zur Darstellung eines Symbols (Schere, Stein, Papier) verantwortlich. Auf dem Spielfeld gibt es total vier Elemente dieser Klasse. Das Feld des Gegners (nur als Anzeige) und die drei eigenen Felder (als Anzeige und zum anklicken). Die Klasse ist von *Pane* (JavaFX) abgeleitet und ihre Instanzen können daher in anderen Panes verschachtelt werden (siehe Klasse *RockPaperScissorsBoard*).

Die Klasse ist mit zwei Variablen, dem Konstruktor und zwei Methoden relativ überschaubar.

Als **Variablen** sind definiert:

- *symbolImageView* (vom Typ *ImageView*): Dient zur Darstellung des Symbols, respektive des Hintergrunds
- *overlayImageView* (vom Typ *ImageView*): Dient zur Darstellung des über das Symbol gelegten Bildes (Kreuz oder Haken)

Im parameterlosen **Konstruktor** werden folgende Schritte erledigt:

- Initialisieren der Super-Klasse
- Erzeugen einer neuen Instanz der *ImageView*-Objekte
- Den Objekten vom Typ *ImageView* wird mittels der Methode *getStyleClass().add* eine Stylesheet-Klasse zugewiesen, welche in **RockPaperScissors.css** definiert wurde
- Dem aktuellen Pane (*this*) werden die beiden *ImageView*-Objekte mittels der *add*-Methode hinzugefügt

Als **Methoden** wurden definiert:

- *setSymbol*: Diese Methode erzeugt das gewünschte Hintergrund- und Overlay-Bild, und nimmt als Parameter das Symbol (Schere, Stein, Papier oder nichts) und den Status (gewonnen, verloren oder unentschieden beziehungsweise nichts) an
 - Zwei Bilder (Typ *Image*) werden definiert, sowie Breite und Höhe des Panes ermittelt
 - Das Hintergrund- respektive Symbolbild wird anhand des übergebenen Parameters aus dem Asset-Package erstellt
 - Das Overlay-Bild wird anhand des übergebenen Parameters aus dem Asset-Package erstellt. Soll kein Overlay-Bild angezeigt werden, wird das Bild auf *null* gesetzt und somit später nicht dargestellt
 - Die beiden zuvor erstellten Bilder werden über die Methode *setImage* gesetzt
- *setImage*: Diese Methode ist für die eigentliche Darstellung von Bildern im Pane verantwortlich. Wie bereits bei der Klasse *RockPaperScissorsBoard* (Methoden *setSelectedPane* und *manipulateText*) müssen die Aktionen zum Darstellen der Bilder in JavaFX in *Platform.runLater* gekapselt sein. Wird als Bild *null* übergeben, wird dieses ignoriert und nicht dargestellt, respektive als transparente Fläche, falls es sich um das Overlay-Bild handelt

4.9.5. Klasse: *RockPaperScissorsClientGame*

Die Datei **RockPaperScissorsClientGame.java** enthält die Klasse *RockPaperScissorsClientGame*. Sie ist für die clientseitige Logik verantwortlich. Die Klasse ist von der Klasse *ClientGame* aus dem Package *org.fhnw.aigs.client.gameHandling.ClientGame* abgeleitet. In Ihr wird alles auf Seiten des Clients erledigt, was mit Kommunikation und Steuerung zu tun hat. Direkte grafische Manipulation ist in dieser Klasse nicht vorgesehen. Das ist den Klassen *RockPaperScissorsBoard* und *RockPaperScissorsBoardPane* vorbehalten. Es gibt nur eine Variable, einen Konstruktor und zwei Methoden, welche implementiert werden müssen (abstrakte Methoden aus Super-Klasse *ClientGame*).

Die einzige **Variable** ist `clientBoard` (vom Typ `RockPaperScissorsBoard`): Die Referenz zum Spielfeld wird in den Methoden der Klasse benötigt um andere Methoden von `RockPaperScissorsBoard` ansprechen zu können.

Der **Konstruktor** nimmt als Parameter den Spielnamen und den GameMode an. Er wird aus der Klasse `Main` in der Methode `start` aufgerufen um eine neue Instanz für das Spiel zu bilden. Im Konstruktor wird nur die Super-Klasse (ebenfalls mit Spielname und GameMode) initialisiert. Dieser Schritt ist wichtig, damit das Spiel überhaupt korrekt läuft.

Die zu implementierenden **Methoden** sind:

- **`processGameLogic`**: In dieser Methode werden vom Server ankommende Nachrichten (abgeleitet von Klasse `Message`) empfangen und verarbeitet. Es könnten im Prinzip beliebig viele Nachrichtenarten verarbeitet werden. Zur Unterscheidung wird immer der Befehl `instanceof` verwendet. Damit kann getestet werden, von welcher Klasse eine Nachricht stammt. Für eine korrekte Verarbeitung der Nachricht muss sie nach der Identifizierung gecastet werden. In der vorliegenden Methode werden drei Nachrichtenarten unterschieden
 - Nachrichten vom Typ `RockPaperScissorsParticipantsMessage` werden dazu verwendet den Namen des Gegenspielers bei Spielbeginn ins Spielfeld einzutragen. Dabei wird der eigene Namen mit den von der Nachricht übergebenen Namen verglichen. Weicht der Name vom eigenen ab, handelt es sich um den Namen des Gegners
 - Nachrichten vom Typ `RockPaperScissorsResultMessage` werden dazu verwendet das Resultat eines Zuges darzustellen. Dabei wird zuerst der Text im Header angepasst. Danach wird die Methode `manipulateGUI` (aus Instanz von `clientBoard`) aufgerufen um die Symbole auf dem Spielfeld anzupassen. Zum Schluss wird die Methode `nextTurn` aufgerufen um den nächsten Zug vorzubereiten
 - Nachrichten vom Typ `GameEndMessage` werden dazu verwendet das Spiel korrekt zu beenden. Zuerst werden alle Eingaben beim Client gesperrt, damit keine Nachrichten mehr an den Server gesendet werden. Danach wird ein `JOptionPane` mit der vom Server gesendeten Nachricht angezeigt. Zum Schluss wird das Programm mit der Methode `System.exit(0)` beendet
- **`onGameReady`**: Diese Methode wird automatisch ausgeführt, sobald das Spiel startklar ist. Damit das Spiel überhaupt startet muss aber darin die Methode `startGame` aus der Super-Klasse aufgerufen werden. Gleichzeitig wird noch der Text im Header angepasst

4.10. Schritt 7: Erstellen der Server-Logik

Zum Schluss wird die Logik auf dem Server erstellt. Die Arbeiten hier sind relativ umfangreich, auch wenn es nur zwei Klassen gibt. Die Server-Logik ist für die Spielregeln und die korrekte Kommunikation zwischen Server und Clients verantwortlich.

4.10.1. Klasse: `GameLogic`

Die Datei `GameLogic.java` enthält die Klasse `GameLogic`. Diese Klasse ist für die gesamte Verarbeitung auf der Serverseite verantwortlich. Theoretisch würde sie bei einem AIGS-Spiel als einzige Klasse im Server-Package ausreichen. Sie ist abgeleitet von der Klasse `Game` aus dem Package `org.fhnw.aigs.common.Game`. Sie enthält drei Konstanten, drei Variablen einen parameterlosen Konstruktor und sieben Methoden, wobei drei davon implementiert (abstrakte Methoden aus Klasse `Game`) worden sind.

Als **Konstanten** sind definiert:

- **GAMENAME** (vom Typ *String*): In ihr ist der Name des Spiels abgespeichert. Er muss Identisch mit dem Namen sein, welcher in der Klasse *Main* in der Konstante **GAMENAME** definiert wurde
- **MINNUMBEROFPLAYERS** (vom Typ *int*): In ihr die notwendige (oder minimale) Anzahl von Spielern für eine Partie gespeichert. Der Wert ist für diesen Spielertyp auf **2** gesetzt
- **NUMBEROFTURNS** (vom Typ *int*): In ihr die Anzahl von Zügen gespeichert, bis die Partie beendet wird. Als Wert wurde **3**, also drei Züge bis Spielende definiert

Als **Variablen** sind definiert:

- **turnPlayers** (vom Typ ArrayList von *RockPaperScissorsTurn*-Objekten): Diese ArrayListe verwaltet die Spieler. Die Liste enthält so viele Objekte, wie Spieler vorhanden sind, in der Regel also zwei. Jedes Objekt speichert ausserdem den aktuellen Stand des Spiels aus Sicht der einzelnen Spieler. Die Klasse *RockPaperScissorsTurn* wird im Anschluss erklärt
- **turnNumber** (vom Typ *int*): Diese Variable speichert den aktuellen Zug der Partie
- **lastTurn** (vom Typ *boolean*): Diese Variable speichert, ob es sich beim aktuellen Zug um den letzten der Partie handelt

Der parameterlose **Konstruktor** initialisiert nur die Super-Klasse über den Befehl *super*. Dieser Aufruf ist wichtig, damit das Spiel korrekt starten kann.

Als **Methoden** sind definiert:

- **initialize**: Diese implementierte Methode wird aufgerufen, sobald genügend Spieler an der Partie beteiligt (angemeldet) sind und das Spiel somit starten kann
 - Die Spieler werden als Objekte vom Typ *RockPaperScissorsTurn* angelegt und in die ArrayList gespeichert
 - Die Variable **turnNumber** (Nummer des Zugs) wird **1** und die Variable **lastTurn** auf **false** gesetzt.
 - Die Methode *setCurrentPlayer(getRandomPlayer())* aus der Super-Klasse wird aufgerufen. Dies ist sehr wichtig, da AIGS-intern ein Spieler mit dem Spiel beginnen muss. Für dieses Spielmodell ist das allerdings unerheblich, da sowieso beide Spieler den Zug beendet haben müssen, bevor ein Resultat berechnet werden kann. Der Methodenaufruf hat also keine Auswirkungen auf das Spiel, ist für den Programmablauf aber wichtig
 - Die Methode *startGame* aus der Super-Klasse wird aufgerufen um das Spiel auf Serverseite zu starten
 - Eine Neue Message vom Typ *RockPaperScissorsParticipantsMessage* wird erstellt und mit der Methode (aus Super-Klasse) *sendMessageToAllPlayers* an alle Clients gesendet
- **processGameLogic**: Diese implementierte Methode verarbeitet die von den Clients gesendeten Nachrichten (abgeleitet von der Klasse *Message*). Es könnten im Prinzip beliebig viele Nachrichtenarten verarbeitet werden. Zur Unterscheidung wird immer der Befehl *instanceof* verwendet. Damit kann getestet werden, von welcher Klasse eine Nachricht stammt. Für eine korrekte Verarbeitung der Nachricht muss sie nach der Identifizierung gecastet werden. Hier wird nur eine einzige Nachrichtenart verarbeitet. Und zwar Nachrichten vom Typ *RockPaperScissorsSelectionMessage*
 - Zuerst wird überprüft ob es sich um den letzten Zug handelt und die variable **lastTurn** auf **true** gesetzt, falls zutreffen. Dieser Wert wird an anderer Stelle zur Unterscheidung verwendet

- Die ArrayList der Spieler (*turnPlayers*) wird in einer for-Schleife durchgegangen und der Spieler gesucht, von welchem die Nachricht stammt. Für diesen Spieler wird dann das gewählte Symbol gesetzt und der Zug (für ihn) als beendet gekennzeichnet
- **checkForWinningCondition**: Diese implementierte Methode überprüft nach jeder empfangenen Nachricht ob ein Zug beendet worden ist, respektive ob die Bedingungen für das Spielende vorliegen
 - In einem ersten Schritt wird überprüft, ob überhaupt alle Spieler den Zug beendet haben. Dazu wird die Methode *allPlayersFinished* aufgerufen. Ist dies nicht der Fall, wird die gesamte Methode verlassen
 - Der Gewinner des Zuges wird mit der Methode *calculateTurnWinner* ermittelt. Als Resultat wird der Index in der ArrayList zurückgegeben
 - Die ArrayList wird mit einer for-Schleife durchgegangen und jeder Spieler der Partie behandelt
 - Wurde als Index (von Methode *calculateTurnWinner*) -1 zurückgegeben, steht unentschieden. Entspricht der zurückgegebene Index dem aktuellen Index in der for-Schleife, ist der Spieler der Gewinner, ansonsten der Verlierer
 - Für die drei möglichen Konditionen werden die Ausgabetexte für die Clients erstellt
 - Eine Neue Nachricht vom Typ *RockPaperScissorsResultMessage* wird mit allen Parametern erstellt und danach den einzelnen Spieler gesendet. Für jeden Spieler wird also eine eigene Nachricht erstellt, da die Texte bei allen Spielern abweichen
 - Nachdem alle Nachrichten gesendet wurden, werden alle Spieler in der ArrayList mit der Methode *nextTurn* aus der Klasse *RockPaperScissorsTurn* zurückgesetzt und für den nächsten Zug vorbereitet
 - Die Nummer des Zuges wird um 1 (++) hochgezählt
 - Falls der letzte Zug erreicht wurde (*lastTurn* ist *true*) wird der Gewinner über alle Züge mittels der Methode *calculateGameWinner* ermittelt. Anschliessend wird eine neue Nachricht vom Typ *GameEndsMessage* erstellt und über die Methode *sendMessageToAllPlayers* an alle Clients gesendet. Das Spiel ist danach beendet
- **allPlayersFinished**: Diese Methode überprüft ob alle Spieler ihren Zug abgeschlossen haben. Dazu wird die ArrayList *turnPlayers* in einer for-Schleife durchgegangen und die Variable *hasTurnFinished* pro Spieler ausgelesen. Ist die Variable bei allen Spielern auf *true*, ist der Zug abgeschlossen und es wird *true* zurückgegeben. Ansonsten wird *false* zurückgegeben
- **calculateTurnWinner**: Diese Methode ist die umfangreichste der Klasse und berechnet den Sieger eines aktuellen Zuges
 - Alle Spieler werden in einer verschachtelten for-Schleife durchgegangen.
 - Alle möglichen Kombinationen zum Gewinnen, Verlieren oder für unentschieden werden getestet
 - Mit der Methode *setTurnState* aus der Klasse *RockPaperScissorsTurn* wird gesetzt ob der Spieler verloren oder gewonnen hat, oder ob unentschieden steht. Gleichzeitig wird mit diesem Methodenaufruf beim Gewinner auch noch die Punktzahl um 1 erhöht
 - Als Resultat wird der Index der ArrayList zurückgegeben, an welchem sich der Gewinner (Objekt des Spielers) befindet
- **calculateGameWinner**: Diese Methode ermittelt den Gewinner aus allen Zügen
 - Als Ausgangspunkt wird die Punktzahl des ersten Spielers der ArrayList *turnPlayers* als grösste Punktzahl genommen. Der Index (0) wird vorderhand als Gewinner-Index (Variable *winnerIndex*) definiert

Hinweis:

Bei nur zwei Spielern hätte die Überprüfung der Gewinn-Konditionen auch ohne for-Schleifen funktioniert. Sobald mehr als zwei Spieler in einem Spiel vorhanden sind, muss der Gewinner in der Regel über Schleifen ermittelt werden.

- Zusätzlich wird grundsätzlich von unentschieden (Variable *draw*) ausgegangen
- In einer for-Schleife werden alle Spieler der ArrayList durchgegangen. Dabei wird beim zweiten Spieler (*i = 1*) angefangen, da der Erste bereits behandelt wurde. Ist die Punktzahl des aktuellen Spielers höher oder kleiner als die bisherige Maximalpunktzahl, wird unentschieden (*draw*) auf *false* gesetzt. Ist die Punktzahl höher, wird die neue Maximalpunktzahl mit dieser Zahl gesetzt und der Gewinner-Index auf den aktuellen Spieler gesetzt
- Ist am Schluss *draw* immer noch *true*, besteht unentschieden und es wird *null* zurückgegeben. Ansonsten wird das Objekt (vom Typ *RockPaperScissorsTurn*) des Gewinners zurückgegeben
- **printSymbol**: Diese Methode ist eine Hilfsmethode (und daher auch *static*). Sie gibt einen Text anhand des übergebenen Enumeratorwertes aus. Diese Methode wird vor allem von der Methode *checkForWinningCondition* aus aufgerufen und soll die Erstellung von Texten für die Ausgabe an die Clients vereinfachen

4.10.2. Klasse: *RockPaperScissorsTurn*

Die Datei **RockPaperScissorsTurn.java** enthält die Klasse *RockPaperScissorsTurn*. Diese Klasse dient zur Speicherung des Spielverlaufes. Jede Instanz der Klasse stellt einen Spieler zum aktuellen Zeitpunkt des Spiels dar. Theoretisch hätte das auch ohne diese Klasse realisiert werden können. Es ist allerdings einfacher die Züge mit einer einzigen ArrayList dieser Objekte zu verwalten, als in mehreren unabhängigen ArrayLists mit Spielern, Punktzahlen, Zügen und gewählten Symbolen.

Die Klasse besitzt sechs Variablen einen Konstruktor und neben diversen getter- und setter-Methoden, drei einfache Methoden.

Als **Variablen** sind definiert:

- **player** (vom Typ *Player*): Speichert das *Player*-Objekt des Spielers, welches vom AIGS-Server beim Spielstart erzeugt wurde. Darin sind zum Beispiel Name und ID des Spielers gespeichert
- **hasTurnFinished** (vom Typ *boolean*): Gibt an, ob der Spieler den aktuellen Zug bereits abgeschlossen hat, oder nicht
- **turnSymbol** (vom Typ *RockPaperScissorsSymbol*): Speichert das gewählte Symbol (Scher, Stein, Papier oder noch nichts) des Spielers beim aktuellen Zug
- **points** (vom Typ *int*): Speichert die Punktzahl des Spielers über alle Züge
- **turnState** (vom Typ *GameState*): Speichert den Status des Spielers im aktuellen Zug. Mögliche Status sind gewonnen, verloren, unentschieden oder noch nicht abgeschlossen (*None*)
- **opponentIndex** (vom Typ *int*): Speichert den Index des Gegners aus der ArrayList. Diese Variable kann aber nur bei maximal zwei Spielern verwendet werden. Bei mehr als zwei Spielern müsste eine andere Lösung gesucht werden. Die Variable dient beim Ermitteln des Gewinners als Hilfe, um schneller auf das Objekt des Gegners zugreifen zu können

Dem **Konstruktor** wird das *Player*-Objekt des Spielers übergeben. Ansonsten wird nur die Punktzahl auf *0* gesetzt, das Symbol und der *turnState* auf *None*, sowie *hasTurnFinished* auf *false*.

Als **Methoden** sind definiert:

- **nextTurn**: Diese Methode setzt die Variable *hasTurnFinished* auf *false*, sowie das Symbol und der *turnState* auf *None*. Wird diese Methode angewendet, bedeutet dies, dass der Spieler im aktuellen Zug noch nichts gewählt hat

- **getPlayerName**: Diese Methode gibt den Namen aus dem *Player*-Objekt zurück. Sie dient als Abkürzung
- **getPlayerID**: Diese Methode gibt die ID aus dem *Player*-Objekt zurück. Sie dient als Abkürzung

4.11. Schritt 8: JAR-Datei erzeugen, Anwendung verteilen und testen

Das Programm kann nun erneut kompiliert, verteilt und getestet werden. Die **Kompilierung** läuft genauso ab, wie in **Schritt 5** erläutert. Allerdings muss in Eclipse zusätzlich noch eine JAR-Datei erzeugt werden. Der Test erfolgt vorerst auf einem Computer.

4.11.1. JAR-Datei erzeugen

Um eine JAR-Datei in Eclipse zu erzeugen gibt es zwei Möglichkeiten:

1. Nutzung von Ant-Scripts mit Integration beim Kompilieren
2. Nutzung der Export-Funktion von Eclipse

Die erste Variante kann den Prozess der JAR-Erstellung automatisieren, ist aber relativ kompliziert und würde dieses Tutorial sprengen. Mehr Informationen dazu gibt es unter:

http://help.eclipse.org/luna/topic/org.eclipse.platform.doc.user/gettingStarted/qs-80_ant.htm?cp=0_1_2

Im Tutorial wird daher die zweite Variante beschrieben.

Im Projekt-Explorer wird ein Rechtsklick auf den Projektordner gemacht und *Export...* ausgewählt. Im Export-Dialog wird der Bereich *Java* aufgeklappt und *Runnable JAR File* ausgewählt.

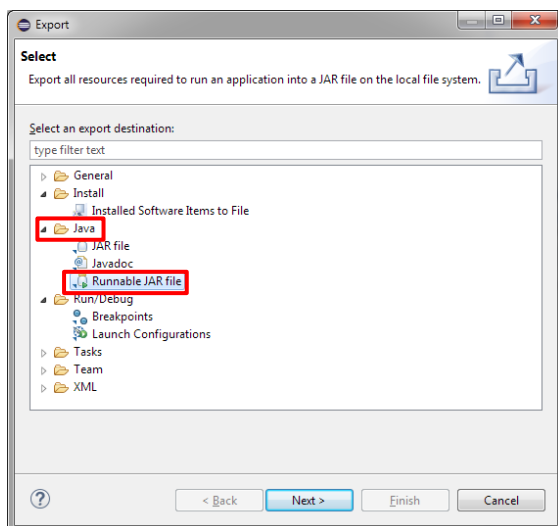


Abbildung 14: Export-Dialog in Eclipse

Mit einem Klick auf *Next* wechselt der Dialog zur nächsten Seite.

Unter *Launch Configuration* wird der Eintrag ausgewählt, welcher in **Schritt 5** (Run-Configuration) erstellt wurde. Ist kein Eintrag vorhanden, muss zuerst **Schritt 5** nachgeholt werden.

Unter *Export destination* kann mit *Browse...* der Ordner *dist* im Projektverzeichnis ausgewählt werden. Ist noch kein Ordner *dist* vorhanden, kann er über das Dateisystem angelegt werden. Als Dateiname wird der Projektname gewählt. Hier also **RockPaperScissors.jar**.

Unter *Library handling* wird der Eintrag *Copy required libraries into sub-folder next to the generated JAR* ausgewählt. Somit können die Bibliotheken wie AIGS Commons oder der AIGS BaseClient unabhängig vom

Spiel ausgetauscht werden. Wird dagegen *Extract required libraries into generated JAR* gewählt, muss das Programm bei einer Änderung der AIGS Commons oder des AIGS BaseClient neu kompiliert werden.

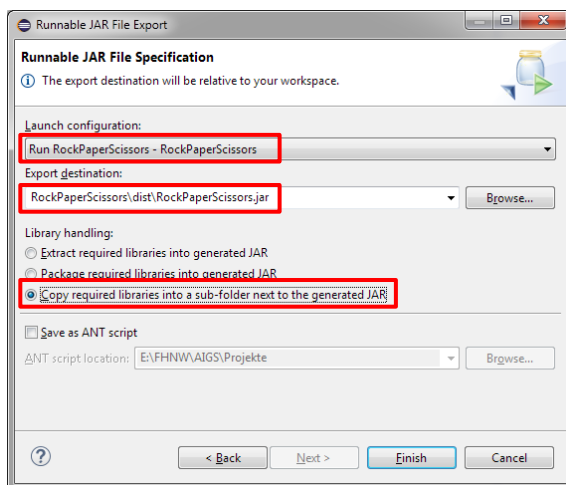


Abbildung 15: Export-Dialog in Eclipse

Tipp:

Für eine raschere Entwicklung in Eclipse könnte die **Schnellkompilierung**, welche in einem späteren Kapitel beschrieben wird, eine Möglichkeit sein.

Mit einem Klick auf *Finish* wird die JAR-Datei im Ordner *dist* erstellt.

4.11.2. Programm verteilen

Um das Programm zu verteilen werden zwei Schritte benötigt.

Als erstes muss die kompilierte JAR-Datei (**RockPaperScissors.jar**) aus dem Verzeichnis *dist* auf dem AIGS-Server im Verzeichnis *gamelibs* abgelegt werden. Ausser der JAR-Datei muss auf den AIGS-Server nichts Weiteres kopiert werden.

Als zweites muss der gesamte Inhalt des Verzeichnisses *dist* in ein anderes Verzeichnis kopiert werden. Dieses Verzeichnis kann an einem beliebigen Ort liegen (zum Beispiel *C:\temp\RockPaperScissors*, */tmp/RockPaperScissors* oder */home/<username>/RockPaperScissors*). Von diesem Verzeichnis aus wird später einer der zwei Clients gestartet. Der andere Client und der Server kann direkt aus Eclipse gestartet werden.

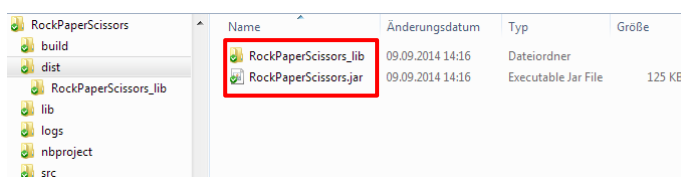


Abbildung 16: Projektverzeichnis nach Kompilierung

Hinweis:

JAR-Dateien, welche mit Eclipse erstellt wurden sollten nicht mit Dateien gemischt werden, welche mit NetBeans erstellt wurden, da die Bibliotheken beim kompilieren anders eingebunden werden.

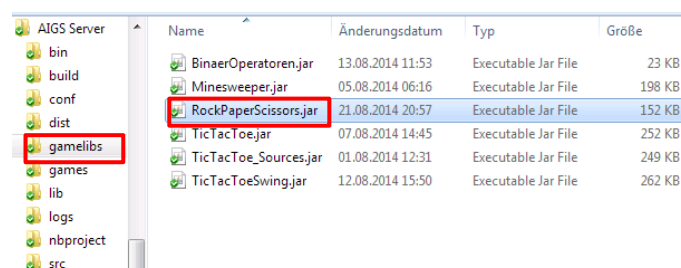


Abbildung 17: Bibliotheksverzeichnis für Spiele auf dem AIGS-Server

Hinweis:

Für die Entwicklung macht es Sinn den AIGS-Sever als Eclipse-Projekt anzulegen. Dieser liegt im Git-Repository auch als vorbereitetes Projekt vor.

4.11.3. AIGS-Server starten

In Eclipse muss für jedes geöffnete Projekt eine Run-Configuration, wie in **Schritt 5** beschrieben vorhanden sein, wenn die einzelnen Programme ausgeführt werden sollen. Für den AIGS-Server muss also eine **neue Run-Configuration** erstellt werden.

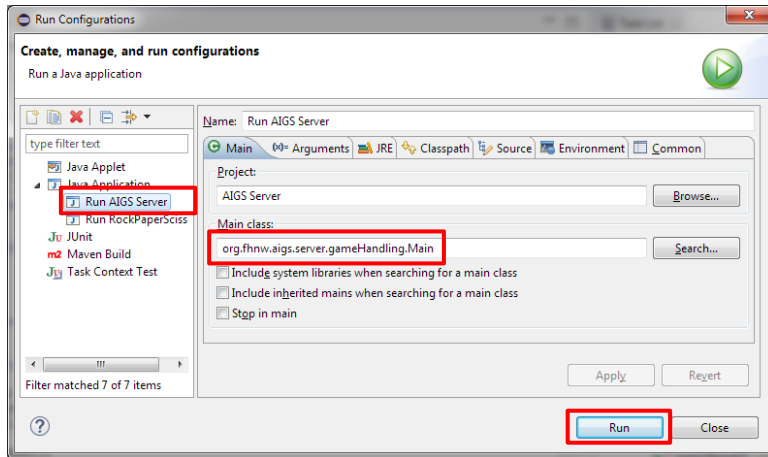


Abbildung 18: Konfigurationsmenü in Eclipse

In Eclipse kann nun der AIGS-Server über: [Run](#) → [Run Configurations...](#) → [Run AIGS Server](#) → [Run](#) ...oder über das grüne Dreieck-Symbol gestartet werden. Dazu muss auf das schwarze (kleine) Dreieck, rechts neben dem Symbol geklickt werden und die Konfiguration (hier also „Run AIGS Server“) ausgewählt werden.

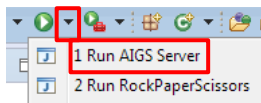


Abbildung 19: Programmstart in Eclipse über eine Konfiguration

Nach dem Programmstart kann als erstes in der Server-Oberfläche überprüft werden, ob das Spiel vom Server erkannt wurde. Ist es in der Liste [Available Games](#) aufgelistet, steht einem Test nichts im Weg. Der Server muss nun noch über die Schaltfläche [Start AIGS](#) gestartet werden. Erst danach empfängt und sendet er Nachrichten.

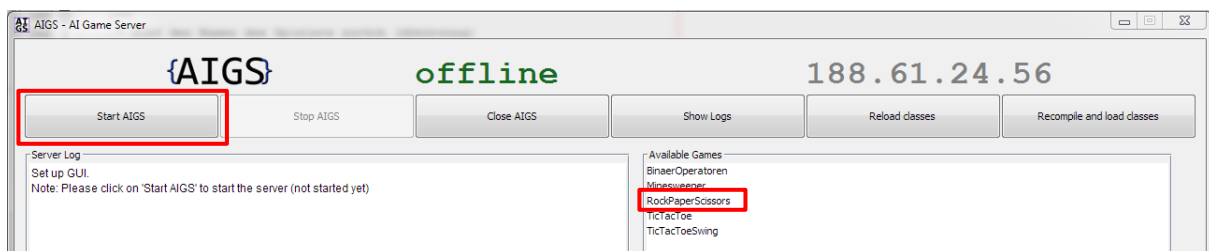


Abbildung 20: AIGS-Server nach Programmstart

Nach dem Start des Servers wird im Fenster [Server Log](#) der Verbindungsstatus, der vorgesehene Port und die externe IP angezeigt. Für den Test wird aber nicht die externe IP, sondern [localhost](#) verwendet.



Abbildung 21: AIGS-Server nach Start

Um das Spiel zu testen werden nun noch **Benutzer-Accounts** benötigt. Diese sind im Verzeichnis des AIGS-Servers im Unterverzeichnis *conf* in der Datei **usersXML.xml** zu finden.

Tipp:

Die folgenden zwei vorbereiteten Benutzer können für die Tests verwendet werden:

Name	Identification Code
test	1
test2	2

4.11.4. Clients starten

Nun können zwei Clients gestartet werden. Ein Client wurde bereits in ein anderes Verzeichnis kopiert und der andere Client kann in Eclipse direkt gestartet werden. Für den Start des Clients in Eclipse muss wie schon beim Server beschrieben, die richtige Run-Configuration ausgewählt werden.

Der Client in Eclipse wird gestartet, wie in **Schritt 5** beschrieben. Dabei ist auf die richtige Run-Configuration (hier also „Run RockPaperScissors“) zu achten.

Sollte das Settings-Fenster nun nicht erscheinen, muss das Zahnrad-Symbol angeklickt werden.



Im Fenster werden nun ein **Benutzername**, der dazugehörige **Identification Code**, die **Server-Adresse** (*localhost*) und der **Port** (*25123*) eingetragen. Danach wird das Fenster über die Schaltfläche *Check and create configuration* geschlossen.

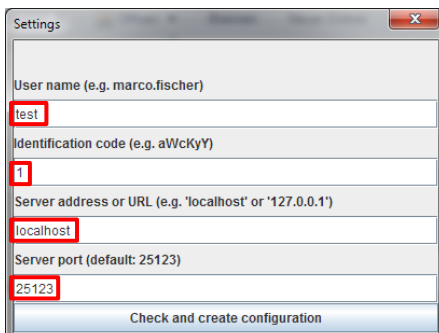


Abbildung 22: Settings-Fenster mit korrekten Angaben

Eine Meldung erscheint, dass die Änderungen erst nach dem Neustart des Programms wirksam werden. Das Programm wird also über die *X*-Schaltfläche geschlossen und in Eclipse erneut gestartet.

Beim aller ersten Programmstart muss das Programm nicht neugestartet werden. Sollte sich das Settings-Fenster nach einem Neustart erneut öffnen stimmen Benutzername oder Identification Code nicht. Der Client zeigt nun den Ladebildschirm an. Auf dem AIGS-Server ist das begonnene Spiel unter *Waiting Games* aufgelistet.

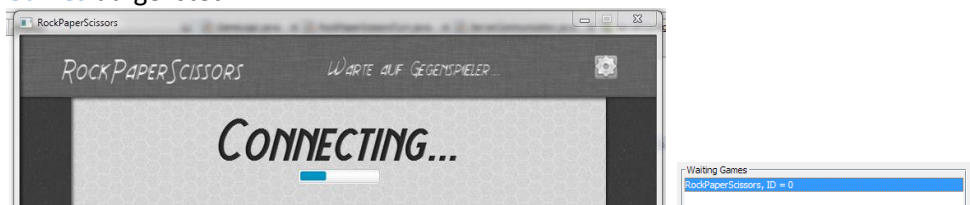


Abbildung 23: Ladebildschirm (links), Anzeige auf der Warteliste des AIGS-Servers (rechts)

Nun wird der zweite Client gestartet, welcher zuvor in ein anderes Verzeichnis kopiert wurde. Dazu kann ein Doppelklick auf die JAR-Datei gemacht werden.

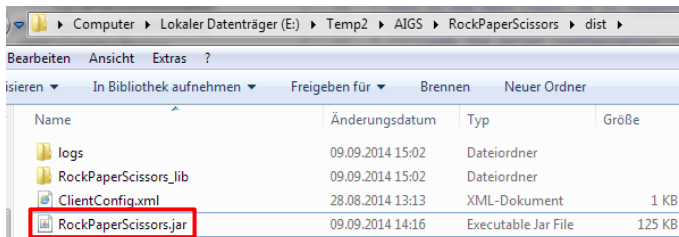


Abbildung 24: Kopiertes dist-Verzeichnis des Projekts

Auch hier müssen die Einstellungen, wie oben beschrieben vorgenommen werden. Bei der Eingabe ist darauf zu achten, dass nicht zweimal derselbe Benutzer gewählt wird, da das Programm Spieler und Gegenspieler nicht mehr richtig unterscheiden kann.

Nachdem beide Clients laufen, wird bei beiden der Ladebildschirm entfernt und das Spiel kann starten. Auf dem Server ist nun das Spiel unter *Active Games* gelistet.

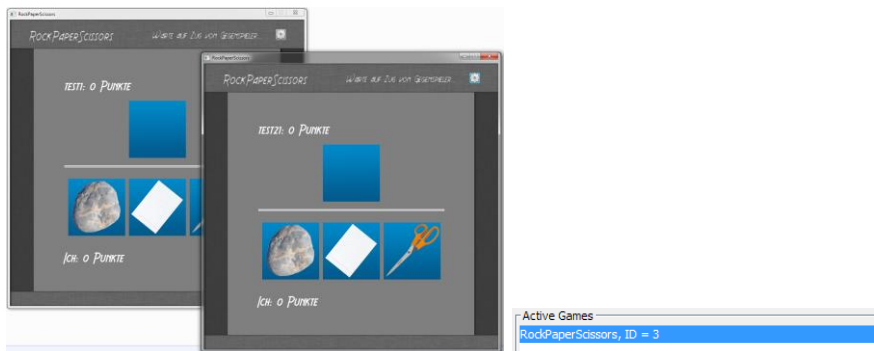


Abbildung 25: Beide Clients nach dem Start (links), Anzeige der Liste aktiver Spiele auf dem AIGS-Server (rechts)

4.11.5. Spiel testen

Das Spiel kann nun lokal getestet werden. Dabei wird abwechselungsweise auf die Auswahlfelder der Clients geklickt. Nach einem beendeten Zug wird das Resultat bekannt gegeben und der nächste Zug vorbereitet. Da sich beide Clients auf demselben Computer befinden erscheint die Meldung nach Beenden des Zuges auch zweimal (einmal pro Client).

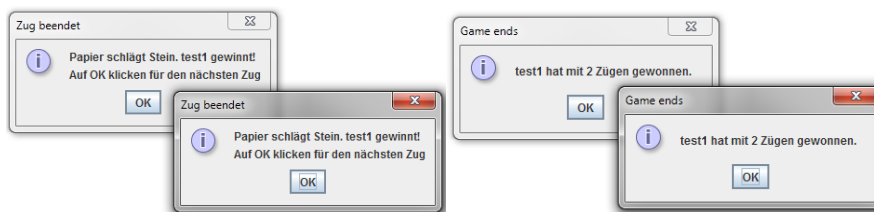


Abbildung 26: Dialogboxen nach einem Zug (links), Dialogboxen nach dem letzten Zug (rechts)

Nach dem letzten Zug erscheint die Meldung über den Gesamtsieger. Nachdem beide Dialogboxen geschlossen wurden, haben sich auch beide Clients selbständig geschlossen. Auf dem AIGS-Server wird das Spiel nun nicht mehr unter *Active Games* oder *Waiting Games* gelistet.

5. Hilfe bei der Entwicklung

Der im letzten Kapitel beschriebene Prozess der Entwicklung kann für jedes AIGS-Spiel angewendet werden. Trotzdem gibt es noch einige Möglichkeiten den Entwicklungsprozess zu vereinfachen. In diesem Kapitel sollen ein paar dieser Möglichkeiten aufgezeigt werden. Ausserdem wird beschrieben, wie nach Fehlern gesucht werden kann (Debugging).

5.1. GUI-Editoren

Der Code für die Programmoberfläche von AIGS (Server und BaseClients), sowie für das Spiel im letzten Kapitel wurden von Hand geschrieben. Dies benötigt aber relativ viel Erfahrung und kann oft mühsam sein. Insbesondere bei JavaFX kann die Suche nach den gewünschten CSS-Befehlen mitunter zeitaufwendig sein. Abhilfe können da GUI-Editoren schaffen. Mit ihnen kann eine Programmoberfläche per WYSIWYG erstellt werden. Der daraus entstehende Code kann später in ein AIGS-Spiel eingebaut werden.

Eclipse bietet im Gegensatz zu NetBeans keinen integrierten GUI-Editor für Swing an. Es gibt allerdings ein Eclipse-Plugin namens „**WindowBuilder**“. Informationen zur Installation gibt es hier:

<http://www.eclipse.org/windowbuilder/>

Die URL für das Softwarepaket in **Eclipse 4.4** ist:

<http://download.eclipse.org/windowbuilder/WB/release/R201406251200/4.4/>

Über den Menüpunkt (in Eclipse) *Help* → *Install New Software...* wird der Installationsassistent aufgerufen. Dort wird die URL unter *Work with* eingetragen und die Pakete *Swing Designer* und *WindowBuilder Engine (Required)* ausgewählt. Mit einem Klick auf *Next* wird die Installation angestoßen. Nach dem Akzeptieren der Softwarelizenz wird die Installation über *Finish* abgeschlossen. Eclipse muss danach neu starten.

Um eine neue GUI-Klasse anzulegen muss im gewünschten Package der ein Rechtsklick gemacht werden und *New...* → *Other...* ausgewählt werden. Anschliessend kann Unter *WindowBuilder* → *Swing Designer* beispielsweise *JPanel* ausgewählt werden. Nach einem Klick auf Next wird der Name der Klasse definiert und mit Finish angelegt.

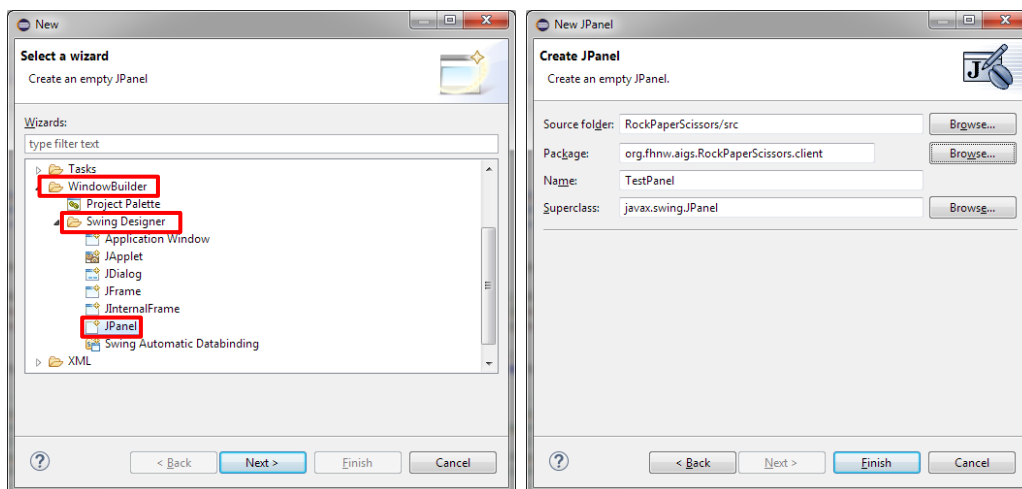


Abbildung 27: Assistent zum neu Anlegen von Klassen in Eclipse

Der Code zur erstellten Oberfläche ist ersichtlich, wenn unter dem Editor auf die Schaltfläche *Source* geklickt wird. Mit der Schaltfläche *Design* kehrt man zurück zum Editor.

Der Code kann aber jederzeit kopiert und in einer anderen Datei oder einem anderen Projekt verwendet werden. Die erstellte Klasse kann aber auch als Ganzes ins Spiel übernommen werden, wenn sie von einer kompatiblen Klasse (z.B. *JPanel*) abgeleitet wurde.

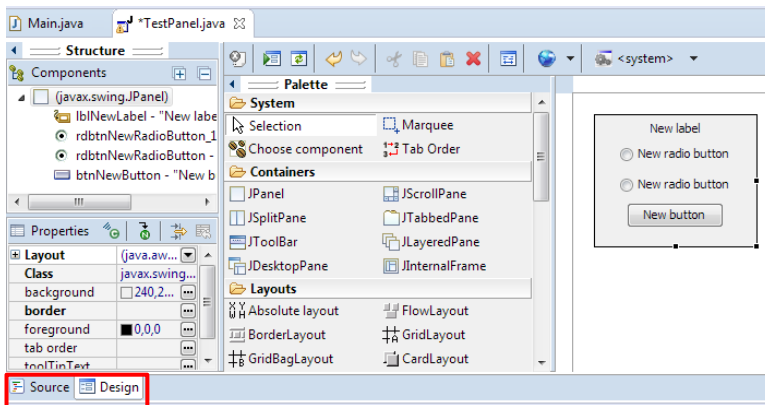


Abbildung 28: GUI-Editor in Eclipse

Für JavaFX stellt Oracle einen kostenlosen Editor namens „**JavaFX Scene Builder**“ bereit. Dieser kann unter der folgenden Adresse im Bereich „**Additional Resources**“ für Windows, Mac oder Linux heruntergeladen werden: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Allerdings ist die Übernahme von Code mit dem JavaFX Scene Builder etwas umständlicher als die Swing-Variante. Dazu muss eine Datei mit der Endung „**fxml**“ ins Programm eingebunden werden. Anstelle von Java-Code befindet sich XML mit der genauen Beschreibung der erstellten Oberfläche in dieser Datei. Artikel zum Thema JavaFX in Kombination mit dem Scene Builder und fxml finden sich hier:

https://blogs.oracle.com/jmxc/etc/entry/connecting_scenebuilder_edited_fxml_to

<http://www.oracle.com/technetwork/articles/java/javafx-fast-1873375.html>

http://docs.oracle.com/javafx/scenbuilder/1/use_java_ides/jsbpub-use_java_ides.htm

5.2. Schnellkompilierung

Der AIGS-Server wurde so konzipiert, dass er Spiele unter bestimmten Umständen selbständig, also ohne IDE, kompilieren kann. Dieses Feature muss allerdings nicht zwingendermassen verwendet werden und ist nur für die Entwicklung, aber nicht für die endgültige Kompilierung eines Spiels vorgesehen. Der Vorteil dieser Methode ist aber, dass die kompilierte JAR-Datei nicht nach jeder Änderung erneut auf den Server kopiert werden muss. Dazu muss ein **Spiel-Projekt** auf dem Server im Verzeichnis *games* abgelegt werden. Für jedes Spiel muss es in diesem Ordner *games* ein eigenes Unterverzeichnis (z.B. *RockPaperScissors*) geben. Wichtig ist, dass das gesamte Projekt, insbesondere die Verzeichnisse *lib* und *src*, in diesem Unterverzeichnis liegen.

Beim Anlegen von Spiel-Projekten kann der Speicherort von Anfang an so gewählt werden, dass sich die Dateien des Projekts in einem Unterverzeichnis auf dem Server befinden. Beim automatischen Kompilieren wird allerdings nur eine auf dem Server lauffähige Version erstellt. Diese wird automatisch nach *gamelibs* kopiert. Um einen lauffähigen Client zu erstellen, muss das Programm einfach wie in **Schritt 5** kompiliert und wie im **Schritt 8** (Unterkapitel „JAR-Datei erzeugen“) erstellt werden.

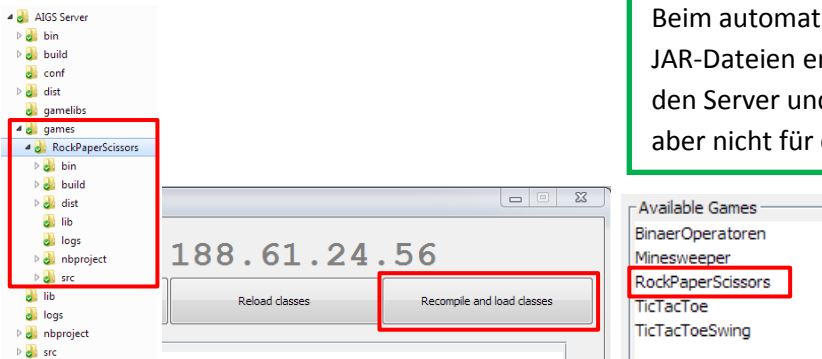
Um eine automatische Kompilierung durchzuführen sind folgende Schritte notwendig:

1. Der AIGS-Server wird gestartet. Der Programmstart reicht aus. Der Service (Start AIGS) muss dafür nicht laufen.

Hinweis:

Beim Starten des AIGS-Servers (Programmstart) werden alle in *games* liegenden Spiele automatisch kompiliert

2. Mit einem Klick auf die Schaltfläche *Recompile and load classes* wird die automatische Kompilierung gestartet
 - a. Der Server kompiliert die JAR-Datei
 - b. Die JAR-Datei wird vom Server ins Verzeichnis *gamelibs* kopiert
 - c. Falls der Eintrag in der Liste *Available Games* noch nicht vorhanden war, taucht nun das Spiel darin auf



Hinweis:

Beim automatischen Kompilieren werden zwei JAR-Dateien erstellt. Das eigentliche Programm für den Server und ein Source-Package. Letzteres wird aber nicht für den Betrieb benötigt.

Abbildung 29: "games"-Verzeichnis auf AIGS-Server (links), Schaltfläche zum Rekompilieren (Mitte), Liste verfügbarer Spiele auf AIGS-Server nach Kompilierung (rechts)

Ein Spiel, welches nur einen Spieler benötigt, respektive der Gegenspieler die KI ist, kann somit ohne manuelles Kopieren von JAR-Dateien entwickelt und getestet werden. Für den Server reicht das automatische Kopieren und für den Client wird die Kompilierfunktion aus der IDE benötigt. Die Quellen sind dabei identisch. Für ein Spiel mit mehr als einem Spieler (Gegenspieler) muss trotzdem noch das *dist*-Verzeichnis aus dem Spielprojekt an einen andern Ort kopiert werden.

5.3. Debuggen der Client-Logik

Das Debuggen der Client-Logik ist relativ einfach, da der Client in der IDE (z.B. Eclipse) als eigenständiges Programm laufen kann. Bei der Server-Logik verhält sich etwas anders. Das debuggen der Server-Komponenten wird später erklärt.

Zum Debuggen muss mindestens ein **Breakpoint** im Programm gesetzt werden, da das Programm ansonsten einfach ausgeführt wird. Ideale **Einstiegspunkte** sind entweder **Konstruktoren** von Klassen oder **Methoden**. Um beispielsweise ankommende Nachrichten vom Server zu debuggen wird ein Breakpoint in der Klasse *RockPaperScissorsClientGame* (Package *org.fhnw.aigs.RockPaperScissors.client*), in der Methode *processGameLogic* gleich nach dem Methodenkopf gesetzt.

Um einen Breakpoint in Eclipse zu setzen muss mit der linken Maustaste auf die Zeilennummer in der Klasse ein Doppelklick gemacht werden. Ein blauer Kreis erscheint neben der Zeilennummer und. Ein weiterer Doppelklick entfernt den Breakpoint wieder. Wird das Programm nun im Debug-Modus gestartet hält es genau an dieser Stelle an, sobald die Programmlogik hier angelengt ist – also wenn eine Nachricht vom Server auf dem Client ankommt. Es können im Übrigen beliebig viele Breakpoints in beliebigen Klassen gesetzt werden.

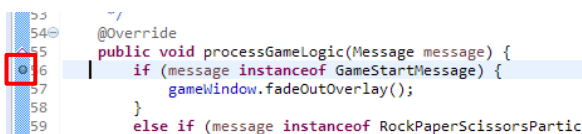



Abbildung 30: Setzen eines Breakpoints in Eclipse

Um das Programm (Client) kann nun im Debug-Modus gestartet werden. Entweder über: [Run → Debug](#) ...oder über das Debug-Symbol. 

Wird das Debug-Symbol verwendet, kann wie bei [Run](#) die Konfiguration mit einem Klick auf das schwarze Dreieck rechts vom Symbol gewählt werden. Somit kann analog zu Run das Programm ausgewählt werden, welches im Debug-Modus gestartet werden soll.

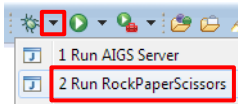



Abbildung 31: Auswahl des Programms im Debug-Modus in Eclipse

Der Debug-Prozess kann mit einem Klick auf das Symbol  jederzeit abgebrochen werden. Damit wird das laufende Programm sofort beendet.

Beim Starten des Debuggers – spätestens aber, wenn ein Breakpoint erreicht wird – schaltet Eclipse die Perspektive (Ansicht) von Java zu Debug um. Dazu kommt ein Infodialog. Die Perspektive kann jederzeit über die zwei Schaltflächen im oberen, rechten Rand von Eclipse umgeschaltet werden.

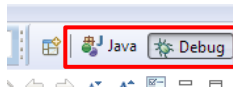
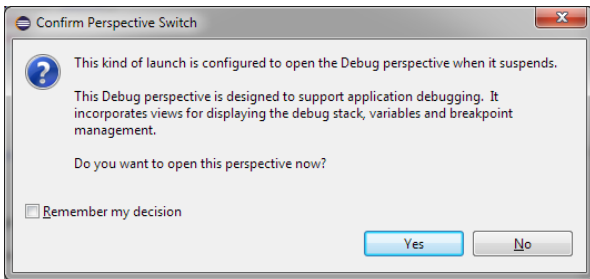


Abbildung 32: Infodialog zum Perspektivenwechsel in Eclipse (links), Schaltflächen zum Umschaltend der Perspektive in Eclipse (rechts)

Die restlichen Programme (zweiter Client und Server) können ganz normal gestartet werden. Sobald eine Nachricht vom Server auf dem Client ankommt, springt der Debugger an die Stelle des Breakpoints. Die Zeile am Breakpoint wird grün eingefärbt.

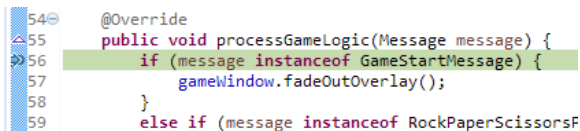





Abbildung 33: Programmstopp an einem Breakpoint in Eclipse

Von diesem Punkt aus kann das Programm Schritt für Schritt ausgeführt werden. Dazu sind vor allem drei Befehle wichtig:

1. **Step over:** Mit Step over wird zur nächsten Programmzeile gesprungen. Dabei wird eine Methode auf der Zeile übersprungen. Somit kommt man relativ schnell im Programm vorwärts. Step Over wird entweder mit der Taste **F6** oder dem Symbol  ausgeführt.
2. **Step into:** Step Into springt ebenfalls zur nächsten Programmzeile. Allerdings wird bei einer Methode auf der Zeile diese mit dem Debugger angesteuert. Somit kann man auch verschachtelte Methodenaufrufe debuggen ohne mehrere Breakpoints zu setzen. Step into wird entweder mit der Taste **F5** oder dem Symbol  ausgeführt.
3. **Resume:** Um einen Breakpoint zu überspringen oder um das Programm fortzufahren kann einfach die Taste **F8** oder das Symbol  angeklickt werden.

Wichtigste Informationsquelle während dem Debuggen ist die Ansicht der Variablen. Diese wird beim Debuggen automatisch im oberen Bildschirmbereich in Eclipse geöffnet. Es werden alle vom aktuellen Punkt aus erreichbaren (im Kontext stehenden) Variablen angezeigt. Also alle „private“ Variablen der aktuellen Methode (oder des aktuellen Konstruktors). In der Variablen-Ansicht werden der Name und der aktuelle Wert dargestellt. Handelt es sich bei einer Variable nicht um einen primitiven Datentyp, kann die Variable mit dem Dreieck-Symbol neben der Variable

aufgeklappt und die Unterobjekte angezeigt werden.

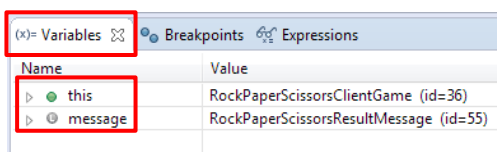


Abbildung 34: Ansicht der Variablen während dem Debuggen in Eclipse

Tipp:

Unter „**Expressions**“ können selbst Parameter oder sogar Methodenaufrufe (Return-Werte) definiert und beobachtet werden. Diese werden dann immer angezeigt, können aber auch ungültig werden, wenn in einer Methode keine entsprechende Variable existiert. Expressions müssen zuerst unter *Window* → *Show View* → *Expressions* eingeblendet werden.

Tipp:

Um *public* und *static* Variablen zu debuggen muss ein Rechtsklick auf den Variabelnamen gemacht werden und der Befehl *Watch* ausgewählt werden. Die Variable wird dann unter Expressions angezeigt.

5.4. Debuggen der Server-Logik

Das Debugging der Server-Logik funktioniert prinzipiell gleich, wie das Debugging der Client-Logik.

Allerdings muss hier der AIGS-Server selbst im Debugging-Modus gestartet werden und nicht der Client.

Das Setzen des Breakpoints an der richtigen Stelle ist hier der wichtigste Punkt.

In Eclipse ist Projektübergreifendes Debugging möglich. Das bedeutet, dass ein im Spiel-Projekt gesetzter Breakpoint im Package *org.fhnw.aigs.RockPaperScissors.server* das Server-Programm anhält, wenn der Server (im Debug-Modus) darauf trifft. Das erscheint auf den ersten Blick unlogisch, da ja nicht das Spiel im Debug-Modus ausgeführt wird, sondern der Server, der Breakpoint sich aber im Spiel-Projekt befindet. Die IDE ist aber so intelligent, dass sie die Zusammengehörigkeit bemerkt und das Server-Programm trotzdem anhält.

Damit das Debugging überhaupt funktioniert muss sowohl das Spiel-Projekt, als auch das Server-Projekt in Eclipse geladen und geöffnet sein. Beide müssen sich also im selben Workspace befinden.

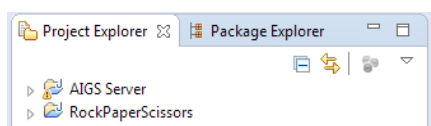


Abbildung 35: Server- und Spiel-Projekt in Eclipse

Nun kann im Spiel-Projekt ein Breakpoint gesetzt werden. Um beispielsweise ankommende Nachrichten vom Client an den Server zu debuggen wird ein Breakpoint in der Klasse *GameLogic* (Package *org.fhnw.aigs.RockPaperScissors.server*), in der Methode *processGameLogic* gleich nach dem Methodenkopf gesetzt.

Um das Debugging zu starten muss nun das Serverprogramm im Debug-Modus gestartet werden, nicht der Client. Nach dem Start des Servers passiert erstmals gar nichts. Der Dienst wird mit *Start AIGS* gestartet. Danach können die beiden Clients (falls zwei Spieler benötigt werden) normal ohne Debug-Modus gestartet werden.

Sobald sie erste Nachricht von einem Client an den Server gesendet wird, stoppt das Server-Programm und

der Breakpoint (im Spiel-Projekt) wird in der IDE angezeigt. Von hier aus funktioniert das Debugging identisch mit dem Client-Debugging.

Server und Client können auch gleichzeitig debugged werden. Allerdings ist es dann schwierig den Programmfluss richtig zu verfolgen, da der Debugger zwischen den verschiedenen Klassen und Projekten hin und herspringt und man sich zuerst orientieren muss ob gerade der Client oder der Server debugged wird.

6. Git-Repository

Sämtliche AIGS-Projekte inklusive AIGS-Server, Commons, BaseClients, vorbereiteten Spielen und Tutorial-Spiel sind auf einem Git-Repository der FHNW verfügbar. Das Repository ist öffentlich zugänglich (read-only) und kann mit einem Git-Client geklont werden.

6.1. Repository-Inhalt

Im Repository sind folgende Verzeichnisse zu finden:

- **AIGS BaseClient:** Darin befinden sich die Quelldateien und Bibliotheken zum AIGS BaseClient (JavaFX). Es ist der Ausgangspunkt für die Bibliothek „AIGS_BaseClient.jar“
- **AIGS Commons:** Darin befinden sich Quelldateien für die Commons. Es ist der Ausgangspunkt für die Bibliothek „AIGS_Commons.jar“
- **AIGS SwingBaseClient:** Darin befinden sich die Quelldateien und Bibliotheken zum AIGS BaseClient (Swing). Es ist der Ausgangspunkt für die Bibliothek „AIGS_SwingBaseClient.jar“
- **BinaerOperatoren:** Darin befinden sich Quelldateien und Bibliotheken für das Demo-Spiel „BinaerOperatoren“
- **Javadoc:** Darin befindet sich eine Zusammenstellung von Javadoc-Dokumentationen aller vorhandenen AIGS-Projekte inklusive des Tutorials
- **Minesweeper:** Darin befinden sich Quelldateien und Bibliotheken für das Demo-Spiel „Minesweeper“
- **Ready-to-go:** darin befindet sich nochmals eine vollständige Kopie aller Projekte. Diese beinhalten aber neben den Quell-Dateien und Bibliotheken noch Projekt-Dateien für NetBeans und Eclipse, sowie die kompilierten JAR-Dateien. Diese Dateien können für einen sehr schnellen Einstieg verwendet werden, oder wenn aus irgendeinem Grund eine JAR-Datei an einem anderen Ort fehlt
- **RockPaperScissors:** Darin befinden sich Quelldateien und Bibliotheken für das Tutorial-Spiel „RockPaperScissors“
- **Schemata:** Hier befinden sich XSD-Schemadateien. Diese werden für die Spiele oder den Server nicht benötigt
- **TicTacToe:** Darin befinden sich Quelldateien und Bibliotheken für das Demo-Spiel „TicTacToe“
- **TicTacToeSwing:** Darin befinden sich Quelldateien und Bibliotheken für das Demo-Spiel „TicTacToeSwing“, welches die Swing-Variante von „TicTacToe“ ist.
- **Tutorial:** Darin befinden sich neben diesem Dokument die Quelldaten für das Tutorial, sowie das Projekt mit dem Grundgerüst. Das fertige Tutorial-Projekt ist im Verzeichnis RockPaperScissors respektive Ready-to-go

6.2. Software und Informationen zu Git

Es gibt diverse frei verfügbare Git-Clients. Eine umfangreiche Liste ist auf der Herstellerseite von Git verfügbar: <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

Für Windows wird **msysGit** empfohlen: <https://msysgit.github.io/>

Für eine angenehme Verwaltung über den Windows Explorer kann zusätzlich noch **TortoiseGit** installiert werden: <https://code.google.com/p/tortoisegit/>

Für Mac wird **SourceTree** empfohlen: <http://www.sourcetreeapp.com/>

Als Alternative kann (unter anderem) **GitX** verwendet werden: <http://gitx.laullon.com/>

Für Linux bietet sich Git aus der Quelle der Distribution (über apt-get, YUM, RPM Package Manager etc.) an.

Als visueller Client kann zum Beispiel **git-cola** verwendet werden: <https://git-cola.github.io/>

Eine Alternative dazu ist **gitg**: <https://wiki.gnome.org/Apps/Gitg/>

Git wurde ursprünglich nur für die Konsole entwickelt. Das heisst: Alle Befehle von Git können über die Kommandozeile aufgerufen werden. Visuelle Tools, wie TortoiseGit oder GitX kamen erst später hinzu, bauen aber ebenfalls im Hintergrund auf die Konsole auf. Da es sehr viele Konsolenbefehle zu Git gibt, würde eine Beschreibung aller Befehle dieses Dokument sprengen. Eine gute Kurzbeschreibung zu den wichtigsten Befehlen ist hier verfügbar:

https://wiki.archlinux.org/index.php/Super_Quick_Git_Guide

Weitere Ressourcen finden sich hier:

<https://www.kernel.org/pub/software/scm/git/docs/gittutorial.html>

<https://www.kernel.org/pub/software/scm/git/docs/everyday.html>

6.3. Klonen des Git-Repository

Nach erfolgreicher Installation von Git kann das Repository von AIGS geklont werden. Hierzu werden keine Berechtigungen wie Username und Passwort benötigt.

1. Auf dem lokalen Computer muss ein Verzeichnis erstellt werden, wo das Repository und später die gesamten AIGS-Projekte liegen sollen (Beispielsweise: `C:\Java\AIGS` oder `/home/myUser/Java/AIGS`, wobei `myUser` für den Benutzernamen steht)
2. Eine Konsole wird geöffnet (Unter Windows `cmd.exe`, unter Mac/Linux `bash`)
 - Zum Testen, ob Git korrekt läuft kann der Befehl „`git`“ eingegeben werden. Der Befehl wird mit der Eingabetaste abgeschlossen. Meldet die Konsole, dass der Befehl nicht gefunden wurde, muss die Installation überprüft werden. Wurde Git richtig installiert, erscheinen diverse Befehle zu Git in der Konsole.
3. In der Konsole wird ins neu erstellte Verzeichnis gewechselt werden
 - Unter Windows: „`cd Java\AIGS`“. Der Befehl wird mit der Eingabetaste abgeschlossen. Befindet sich das Verzeichnis nicht auf dem C-Laufwerk, muss zuerst das Verzeichnis gewechselt werden. Beispielsweise für Laufwerk E wird „`E:`“ gefolgt von der Eingabetaste eingegeben.
 - Unter Mac oder Linux: „`cd home/myUser/Java/AIGS`“. Der Befehl wird mit der Eingabetaste abgeschlossen.

4. Das Repository wird mit folgendem Befehl geklont: „[git clone ssh://git_projects@ol19ns11008.fhnw.ch:50022/matthias.stoeckli/ai-game-server.git](https://git.projects@ol19ns11008.fhnw.ch:50022/matthias.stoeckli/ai-game-server.git)“, wobei hinter „git“ und „clone“ ein Leerzeichen steht. Der Befehl wird mit der Eingabetaste abgeschlossen.
5. Wurde der Befehl und die Adresse richtig eingegeben beginnt der Download des Repository in das angegebene Verzeichnis.
 - Nach dem alle Dateien heruntergeladen wurden, überprüft Git das lokal kopierte Repository und schliesst mit der Ausgabe „**Checking out files: 100% (SOLL/IST), done**“ ab, wobei die Zahlen von *SOLL* und *IST* übereinstimmen sollten.

6.4. Erstellen eines Eclipse-Projektes aus dem Repository

Mit Ausnahme des Verzeichnis „Ready-to-go“ befinden sich in den einzelnen Verzeichnissen des Repository keine Projektinformationen. Nur Quelldateien (Unterordner *src*), Bibliotheken (Unterordner *lib*) und eine Datei namens **build.xml**, welche Informationen zur Kompilierung enthalten kann. Beim AIGS-Server sind zusätzlich noch weitere Verzeichnisse und Dateien enthalten, welche im Betrieb benötigt werden, aber nicht als Bibliotheken eingebunden werden müssen.

Um ein Projekte in den Unterverzeichnissen des Repository anlegen zu können sind nur wenige Schritte notwendig. Alternativ können Projekte aber auch manuell angelegt werden, wie in **Schritt 1** (neues Projekt anlegen) beschreiben.

Name	Änderungsdatum	Typ	Größe
lib	03.09.2014 11:01	Dateiordner	
src	03.09.2014 11:01	Dateiordner	
build.xml	03.09.2014 11:01	XML-Dokument	4 KB

Abbildung 36: Unterordner vom Repository nach Klonen

Im nachfolgenden Beispiel soll das Spiel „TicTacToe“ aus dem Git-Repository erstellt werden.

Zuerst muss wieder der **Workspace** in Eclipse definiert werden. Dieser muss auf das Hauptverzeichnis des Git-Repository gelegt werden. Also auf den Ordner *ai-game-server*. Der Workspace ist momentan noch leer.

Nun wird ein neues Projekt über den Menüpunkt angelegt: **File → New → Java Project**

Der Name des Projekts (*Project name*) muss genau dem Namen des Projektordners entsprechen. Hier also „TicTacToe“. Der Haken bei *Use default location* bleibt gesetzt. Wurde der Name richtig eingegeben, erscheint eine Meldung „The wizard will automatically configure the JRE and the layout based on the existing source“ am unteren Rand des Projektassistenten.

Mit einem Klick auf *Next* geht es zur nächsten Seite.

In den Reitern *Source* und *Project* muss nichts verändert werden. Im Reiter *Libraries* müssen aber die Bibliotheken überprüft werden.

Hier ist einerseits das aktuelle **JRE** verknüpft. Andererseits sind alle gefundenen **JAR-Dateien** verknüpft. Es dürfen nur das JRE, sowie die beiden Bibliotheken *AIGS_BaseClient.jar* und *AIGS_Commons.jar* aus dem Unterordner *lib* eingebunden sein.

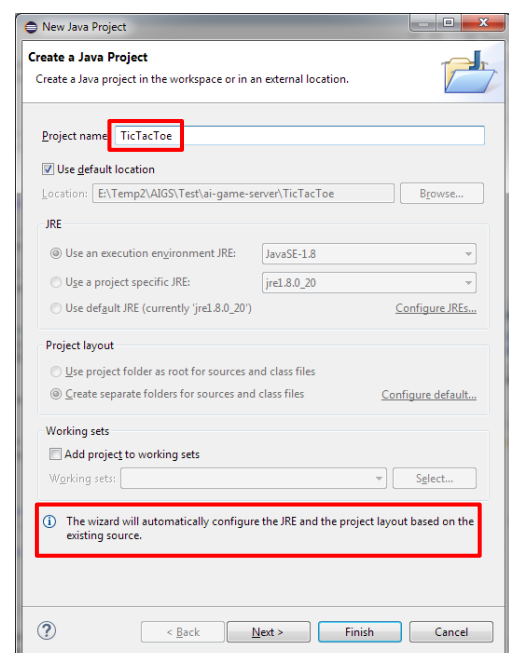


Abbildung 37: Projektassistent nach Eingabe von Name

Sollten noch andere JAR-Dateien (zum Beispiel aus einem Ordner *dist*) vorhanden sein, müssen diese mit der Schaltfläche *Remove* entfernt werden.

Sollten JAR-Dateien aus dem Ordner *lib* fehlen, können diese über *Add JARs...* hinzugefügt werden.

Hinweis:

Beim **AIGS-Server** werden beim Anlegen des Projekts alle JAR-Dateien aus dem Ordner *gamelibs* hinzugefügt. Diese müssen unbedingt entfernt werden. Am besten werden alle Dateien bis auf das JRE markiert, mit *Remove* entfernt und dann der Inhalt vom Ordner *lib* mit *Add JARs...* wieder hinzugefügt.

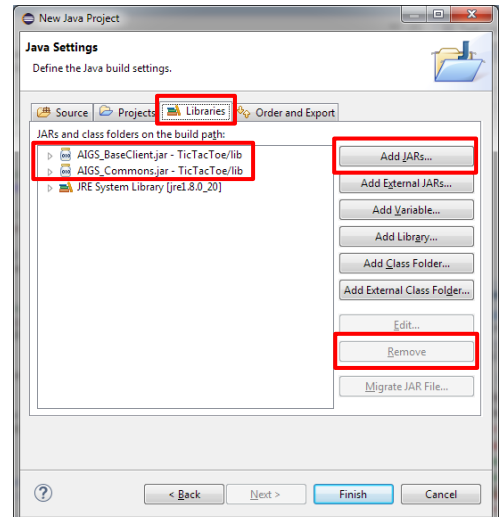


Abbildung 38: Projektassistent, Ansicht Libraries

Mit einem Klick auf *Finish* wird der Assistent geschlossen.

Das Projekt ist nun fertiggestellt und kann nach dem Setzen der main-Methode bereits kompiliert und gestartet werden. Die *main*-Methode wird wie in **Schritt 5** beschrieben definiert.

Hinweis:

Sollten im neuen Projekt Umlaute falsch dargestellt werden, muss das Projekt auf den Zeichensatz **utf-8** umgestellt werden. Dazu wird auf dem Projektordner ein Rechtsklick gemacht und *Properties* gewählt. Dann wird unter *Resource* der Eintrag unter Text *file encoding* umgestellt auf *Other*. Im Auswahlfeld wird dann *UTF-8* ausgewählt und mit *OK* bestätigt.

7. Troubleshooting

Hier werden ein paar gängige Probleme beschrieben, welche während der Entwicklung auftreten können. Die Liste hat keinen Anspruch auf Vollständigkeit.

Problem:

Trotz richtig eingebundener Bibliotheken und keinen Syntaxfehlern kompiliert das Programm nicht. Es werden unklare Fehlermeldungen ausgegeben

Lösungsansätze:

- Wurde der *build*-Ordner im Projektverzeichnis gelöscht, findet Eclipse die Main-Klasse nicht mehr. Um das zu korrigieren muss das Projekt einmalig über *Project* → *Clean...* neu kompiliert werden.
- Unter Eclipse kann es zu Fehlermeldungen in der Art „Access restriction: The type 'Application' is not API“ kommen. Dabei handelt es sich um einen Verarbeitungsfehler in der IDE. Die Fehlermeldungen können entfernt werden, indem unter *Properties* → *Java Build Path* → *Libraries* der Eintrag *JRE System Library [jre1.8.0_NUMMER]* über *Remove* entfernt wird, und danach über *Add Library* wieder eingefügt wird.
- Es werden Bibliotheken verwendet, welche mit einem anderen JDK kompiliert wurden. Bibliotheken welche mit Java 7 kompiliert wurden, können unter Java 8 Probleme machen und umgekehrt. Möglicherweise müssen alle Bibliotheken (z.B. Commons und BaseClient) neu kompiliert und dann die daraus erstellten JAR-Dateien neu als Bibliotheken eingebunden werden. Dazu müssen die

beiden Projekte „AIGS BaseClient“ (respektive „AIGS SwingBaseClient“) und „AIGS Commons“ geladen und kompiliert werden.

- Es wurde eine Projektdatei geöffnet, welche unter einem anderen JDK erstellt wurde. Um das zu korrigieren muss in den Projekt-Einstellungen, unter *Java Compiler* der richtige *Compiler compliance level* eingestellt werden. Dieser sollte auf **1.8** stehen, wenn mit Java 8 gearbeitet wird.
- Sollte nichts funktionieren, empfiehlt es sich den Workspace auf einen neuen Ordner zu setzen und ein neues Projekt zu erstellen. Die Java-Dateien (in *src*) und JAR-Dateien (in *lib*) müssen dabei kopiert werden.

Problem:

Beim Erstellen der JAR-Datei des Projekts (Runnable JAR File Export) werden Fehlermeldungen ausgegeben: „Resource is out of sync with the file system“

Lösungsansätze:

- Aus Performancegründen werden in Eclipse Dateien nicht automatisch aktualisiert, wenn sie von ausserhalb der IDE geändert wurden. Dies kann z.B. der Fall sein, wenn sich ein Java-Projekt in einem Ordner befindet, welcher mit einer Cloud (z.B. Dropbox) gesynct ist. Um die automatische Aktualisierung zu aktivieren wird im Menü gewählt: *Window → Preferences* Dann auf *General → Workspace* und dort den Haken setzen bei *Refresh using native hooks or polling*.

Problem:

Nach einem Doppelklick auf die JAR-Datei des Spiels passiert nichts (kein Fenster öffnet sich).

Lösungsansätze:

- Die JRE-Installation könnte fehlerhaft sein. Damit eine JAR-Datei mit Java geöffnet wird, muss dem System bekannt sein, wo sich die JRE-, respektive JDK-Installation befindet. Unter Windows, Mac und Linux wird dies im „Path“ definiert. Das Eintragen des JDK-Standortes (z.B. *C:\Program Files\Java\jdk1.8.0_11\bin*) in den Path oder eine Neuinstallation des JDK kann Abhilfe vom Problem schaffen.
- Im Programm könnte es beim Programmstart zu einer Exception gekommen sein, welche zum Absturz führt. Um eine etwaige Exception sichtbar zu machen muss die JAR-Datei über die Konsole (unter Windows *cmd*) gestartet werden. Der notwendige Befehl sieht in etwa so aus: *„java -jar E:\Java\RockPaperScissors\dist\RockPaperScissors.jar“*, wobei der Pfad natürlich anzupassen ist. Bei einem Absturz wird die Konsole nach dem Programmende nicht geschlossen und die Exception kann angeschaut werden.
- Anstelle der von der IDE kompiliert JAR-Datei, wurde die Datei verwendet, welche durch die automatische Kompilierung auf dem AIGS-Server erstellt wurde. Diese enthält keine Assets und ist daher nicht als Client-Programm geeignet.
- Es wurde eine JAR-Datei verwendet, welche mit Eclipse erstellt wurde nun nicht auf den Ordner *SPIELNAME_lib* (im Ordner *dist*) zeigt sondern auf ein anderes Verzeichnis. Es sollen bei den Spielen keine JAR-Dateien gemischt werden. Bei der Verteilung von Spielen sollte der komplette Inhalt von *dist* verteilt werden (egal ob Eclipse oder NetBeans).

Problem:

Nachdem der AIGS-Server gestartet wurde scheinen alle zuvor gemachten Programmänderungen am Spiel rückgängig gemacht worden zu sein.

Lösungsansätze:

- Beim Starten des AIGS-Serverprogramms werden alle Spiele im Verzeichnis [games](#) kompiliert und die JAR-Datei ins Verzeichnis [gamelibs](#) kopiert. Wird eine JAR-Datei von einem anderen Ort nach [gamelibs](#) kopiert, wird diese Datei beim Start überschrieben. Um das zu verhindern, kann Das Spiel-Projekt aus dem Verzeichnis [games](#) an einen anderen Ort, ausserhalb des AIGS-Servers verschoben werden.
- Es wurde vergessen die kompilierte JAR-Datei aus dem Spiel-Projekt in den Server ins Verzeichnis [gamelibs](#) zu kopieren.

Problem:

Beim Ausführen oder Debuggen wird ständig nach der Main-Klasse gefragt.

Lösungsansätze:

- Für den Programmstart muss eine Main-Klasse definiert werden. Diese wird in den [Run-Configurations](#) eingestellt. Siehe dazu **Schritt 5** des Tutorials.

Problem:

Beim Debuggen hält das Programm nicht am definierten Breakpoint an, sondern läuft einfach weiter.

Lösungsansätze:

- Das Programm könnte aus Versehen nur gestartet (Run) und nicht im Debug-Modus ausgeführt worden sein.
- Das falsche Programm wurde im Debug-Modus gestartet. Um eine die Client-Logik zu debuggen muss der Client mit dem Debugger gestartet werden. Um die Server-Logik zu debuggen muss der AIGS-Server im Debug-Modus gestartet worden sein.
- Das Programm wird nicht angehalten, weil der Breakpoint gar nicht erst erreicht wird. Breakpoints sollten zu Beginn der Fehlersuche möglichst nahe am Methodenkopf, oder möglichst in der Nähe von der main-Methode positioniert werden. Sobald sichergestellt ist, dass das Programm wirklich anhält, können Breakpoints an „tieferen“ Punkten im Programm positioniert werden.

Problem:

Beim Schnellkompilieren auf dem AIGS-Server kommt es zu einem Absturz. Es wird gemeldet, dass ein oder mehrere Dateien oder Ordner nicht gefunden wurden

Lösungsansätze:

- Bei der Schnellkompilierung wird ein Ant-Script gestartet. Dieses Script sucht nach diversen Java-Dateien. Unter anderem auch nach den Dateien der AIGS Commons. Das Projekt zu AIGS Commons muss sich im selben Hauptordner wie das Projekt des AIGS-Servers (z.B. in [ai-game-server](#)) befinden. Ausserdem muss sich im Projektverzeichnis von AIGS Commons unbedingt ein Ordner [bin](#) befinden.

Problem:

Javadoc kann nicht erstellt werden. Es werden stattdessen sehr viele Warn- und Fehlermeldungen ausgegeben.

Lösungsansätze:

- Seit Java 8 ist wird Javadoc sehr strikt behandelt. Es wird fehlerfreies HTML verlangt. Ausserdem müssen alle Referenzen auf Klassen oder Methoden vorhanden sein. Um wieder das tolerantere Verhalten von Java 7 zu erhalten, kann bei den Javadoc-Optionen (Menüeintrag *Project* → *Generate Javadoc...* → dritte Seite des Assistenten) das Flag *-Xdoclint:none* unter *VM options* angegeben werden. Weitere Informationen: <http://blog.joda.org/2014/02/turning-off-doclint-in-jdk-8-javadoc.html>

Problem:

Beim Starten des AIGS-Servers über einen Doppelklick auf die JAR-Datei tritt ein Fehler auf. Das Programm startet nicht oder nur fehlerhaft.

Lösungsansätze:

- Vermutlich wurde die JAR-datei im *dist*-Verzeichnis gestartet. Dort fehlen aber die Verzeichnisse *games*, *gamelibs*, *conf* und *logs* (falls vorhanden). AIGS-Server sucht diese Verzeichnisse auf einem relativen Pfad. Für einen korrekten Start müssen diese Ordner ins *dis*-Verzeichnis, oder die JAR-Datei ins Projektverzeichnis kopiert werden

Problem:

Beim Testen des Spiels öffnet zwar der Client, aber auch wenn genügend Spieler vorhanden sind verschwindet der Ladebildschirm nicht (Partie beginnt nicht).

Lösungsansätze:

- Der AIGS-Server muss gestartet worden sein, damit Spiele laufen. Es reicht nicht nur das Server-Programm zu starten. Der Dienst muss auch noch über den Button *Start AIGS* gestartet werden
- Clients und Server befinden sich vielleicht nicht im Selben Netzwerk (LAN oder WLAN). Vielleicht verhindert eine Firewall die Verbindung zwischen Clients und Server. Dann muss entweder der Port im AIGS-Server und den Clients geändert werden, oder der Port in der Firewall (oder dem Router) freigegeben werden.
- Beim Entwickeln wurde vielleicht vergessen das Spiel mittels der Methode *startGame* (in der Klasse *GameLogic* auf dem Server oder in der Klasse *ClientGame* auf dem Client) zu starten. Dann wird zwar der Client angezeigt, aber die Spiellogik wird nicht verarbeitet.
- Vielleicht ist es beim Start zu einer Exception auf dem Client gekommen. Hier müsste die Konsole überprüft werden. Für diesen Fall sollte das Spiel entweder in der IDE oder über die Kommandozeile (falls nur die JAR-Datei gestartet werden soll) gestartet werden.

Problem:

Beim Klicken auf ein Element im Fenster stürzt das Spiel plötzlich ab.

Lösungsansätze:

- Falls JavaFX verwendet wurde, wurde vielleicht eine Operation aufgerufen, welche den Fensterinhalt ändern soll. Z.B. wenn ein Bild, ein Text oder ein CSS-Style angepasst werden soll. Diese Operationen müssen in einer gekapselten Methode [Platform.runLater](#) aufgerufen werden. Diese Methode ist im Tutorial beschrieben.
- Vielleicht wurde in der Methode [processGameLogic](#) (auf dem Server oder Client) eine Nachricht ins falsche Format gecastet, was nun eine Exception ausgelöst hat.

Problem:

Der Client startet zwar ordnungsgemäss, es werden aber keine Bilder oder die definierten CSS-Styles (JavaFX) angezeigt.

Lösungsansätze:

- Vielleicht liegen die Assets an einem falschen Ort oder in einem falschen Package.
- Vielleicht wurde der Aufruf zum Laden der Assets nicht richtig gemacht. Eventuell stimmt der Pfad nicht. Als Referenz können hier die in AIGS vorbereiteten Spiele dienen.
- Vielleicht wird ein Bildformat referenziert, welches JavaFX nicht versteht. Unterstützt wird zum Beispiel .png, .bmp, aber nicht .psd oder .tif.

Problem:

Es werden nicht die richtigen oder zu wenig Daten vom Server zu den Clients oder von den Clients zum Server übertragen. Einzelne Variablen fehlen im übertragenen XML oder beinhalten falsche Werte

Lösungsansätze:

- Es wurden Variablen mit Datentypen in Klassen der Commons (des Spiels) definiert, welche nicht in XML umgewandelt werden können. Beispielsweise kann ein Stream (als Objekt) nicht als XML abgespeichert werden. Auch Kollektionen mit Generics können Probleme machen. Diese Datentypen müssen zur Übertragung zuerst in sinnvolle Formate, wie Strings oder numerische Werte umgewandelt werden.
- Vielleicht wurde eine Annotation in einer Klasse der Commons (des Spiels) doppelt oder falsch definiert. Diese Annotationen steuern die Umwandlung von Java-Objekten nach XML und von XML zurück zu den Java-Objekten.

8. Glossar

Ableiten / Ableitung	<p>Als Ableitung wird das Wiederverwenden von Klassen bezeichnet. In Java wird dies durch das Schlüsselwort „extends“ angezeigt. Abgeleitete Klassen erben alle Eigenschaften und Methoden von der Basisklasse. Zusätzlich können aber weitere Eigenschaften und Methoden hinzugefügt werden. Die Basisklasse enthält dabei allgemeingültige, die abgeleiteten Klassen spezialisierte Methoden und Eigenschaften.</p> <p>Beispiele:</p> <pre>public class Hund extends Tier public class Fichte extends Baum</pre>
-------------------------	--

Abstrakt	Abstrakt bedeutet in der objektorientierten Programmierung (z.B. Java), dass Klassen oder Methoden nicht direkt verwendet werden können. Von einer abstrakten Klasse kann kein Objekt erstellt werden. Eine abstrakte Methode kann nicht aufgerufen werden. Sie dienen nur als eine Art Schablone und müssen zuerst zwingend abgeleitet oder implementiert werden. Eine abstrakte Methode macht z.B. dann Sinn, wenn es keine allgemeingültige Methode für die Basisklasse gibt, aber die Methode in jeder abgeleiteten Klasse vorkommen muss. Abstrakte Klassen können dazu verwendet werden, dass gleichartige Klassen davon abgeleitet werden, welche dann immer dieselbe Struktur und identische Grundeigenschaften haben. In AIGS leiten alle Spielklassen von der abstrakten Klasse „Game“ ab.
AIGS BaseClient	Der AIGSBaseClient stellt das Grundgerüst für ein Spiel im AIGS-System dar. Neue Spiele können auf diesem Grundgerüst aufbauen. Enthalten sind unter anderem die Methoden für Kommunikation zwischen Client und Server
AIGS Commons	Bibliotheken für den Betrieb des AIGS-Servers und der einzelnen Clients. Die Commons-Bibliotheken müssen daher sowohl unverändert auf dem Server, als auch auf jedem Client vorhanden sein. In den AIGSCommons werden keine Daten von Spielen, sondern nur grundsätzliche Prozesse verwaltet.
Annotation	Annotationen in Java werden bei Klassen, Konstruktoren, Methoden oder Variablen vorangestellt. Sie beeinflussen nicht direkt den Programmfluss, können aber Auswirkungen auf die Kompilierung oder andere Operationen haben. Eine Annotation beginnt immer mit dem @-Zeichen. Gängigsten Annotationen sind zum Beispiel @Override oder @Deprecated. Sie werden aber auch häufig für die XML-Serialisierung (siehe JAXB) benötigt um XML-Tags korrekt definieren zu können. Hier gibt es zum Beispiel die Annotationen @XmlElement oder @XmlRootElement.
Ant	XML-basierte Script-Sprache, welche vor und nach dem Kompilieren eines Java-Programmes diverse Operationen ausführen kann. Ant kann z.B. dazu verwendet werden Libraries zu kopieren, Verzeichnisse zu erzeugen oder Dateien zu verschieben. Viele IDEs (z.B. NetBeans) erstellen für ihre Projekte automatisch Ant-Scripte, welche angepasst werden können.
Asset(s)	Als Assets werden speziell im Gamedesign Ressourcen wie Grafiken, Soundeffekte, Schriftarten, 3D-Modelle oder Musik bezeichnet. Assets machen in der Regel den grössten Teil eines Spiels aus und können in modernen Spielen viele Gigabyte umfassen.
Bibliothek(en)	Siehe Libraries
Client	Als Client wird bei AIGS das Programm bezeichnet, welches lokal beim Spieler läuft. Alle Clients sind mit dem zentralen AIGS-Server verbunden.
Common(s)	Bibliotheken des AIGS-Servers oder eines bestimmten Spiels, welche sowohl auf dem Server, als auch auf dem Client vorhanden sein müssen. Die Common-Bibliotheken dienen meistens dazu die Kommunikation zwischen Client und Server sicherzustellen. Sie dürfen niemals einseitig (auf Server oder Client) geändert werden. Bei einer Änderung müssen die Bibliotheken sowohl auf dem Server, als auch auf allen Clients getauscht werden. Es gibt für jedes Spiel Commons und auch für den Betrieb des AIGS-Systems selbst (AIGS Commons).
Debugg(er)	Als Debugging wird die Fehlersuche in einem Programm bezeichnet. Der Debugger ist eine speziell angepasste Programmumgebung, welche es erlaubt das Programm an beliebigen Stellen anzuhalten, Variablen zu überprüfen oder in den Programmverlauf einzugreifen. Jede moderne IDE besitzt einen oder mehrere Debugger. Ein Debugger muss aber für jede Programmiersprache speziell erstellt werden. Java-Debugger können nicht mit C#, PHP oder C++ umgehen und umgekehrt.
Enumerator	Fest definierte Auswahlliste von Werten. Enumeratorwerte werden als Texte

	dargestellt, werden aber bei Java-Intern wie Nummern (Integer) behandelt. Jeder Enumeratorwert stellt eine andere Nummer dar. Es können keine ungültig (nicht definierten) Enumeratorwerte benutzt werden. Somit können Missverständnisse vermieden werden, welche bei der Benutzung von Strings oder Nummern als Auswahlwerte entstehen könnten.
Exception	Ausnahme während Programmausführung. Dabei muss es nicht zwingend zum Programmabsturz kommen. Viele Exceptions können abgefangen und behandelt werden. Wird z.B. eine Datei nicht gefunden, welche über einen Stream eingelesen werden sollte, kann eine Meldung ausgegeben werden, anstelle eines Programmabsturzes. AIGS nutzt Exception-Handling intensiv.
Git	Git ist eine Sourcecode-Verwaltung und Versionierungssoftware. Es handelt sich um ein Server-Client-System, welches beliebige Dateien verwalten und versionieren kann. Git basiert auf sogenannten Repositories. Es handelt sich dabei um Sammlungen oder Projekte. Git-Clients können Repositories von Git-Servern auf einen lokalen Rechner klonen und (falls die Berechtigung besteht) Änderungen am wieder ins Repository zurückspielen (Commit). Alternativen zu Git sind unter anderem SVN, Mercurial oder Microsoft Team Foundation Server.
IDE	Integrated Development Environment. Zusammenstellung von Entwicklungswerkzeugen, meist in einem umfangreichen Editor. Für die Java-Entwicklung sind unter anderem NetBeans und Eclipse sehr verbreitete IDEs.
JAR	JAR steht für Java Archive. Es handelt sich grundsätzlich nur um eine Zip-Datei. Damit Java weiss, was sich in einer JAR-Datei befindet ist, enthält diese Metadaten (in Datei MANIFEST.MF). In JAR-Dateien befinden sich oft Libraries, aber auch kompilierte Java-Programme, Sourcecode oder Javadoc. Eine JAR-Datei kann mit einem Zip-Programm, wie 7Zip geöffnet und angeschaut werden.
Java EE	Java Enterprise Edition. Hier sind dem Java-Kern neben den grundlegenden auch für das Unternehmensumfeld wichtige Java-Libraries beigelegt. EE gegenübergestellt sind SE und ME.
JavaFX	GUI-Framework für Java, ähnlich Swing, SWT oder AWT. JavaFX bietet viele moderne Ansätze, wie XML zur Beschreibung oder CSS zur Gestaltung von Programmoberflächen. Seit Java 8 ist JavaFX Bestandteil von Java SE und könnte Swing zukünftig ablösen.
Java ME	Java Micro Edition. Speziell für mobile Endgeräte optimierte Java-Version. ME gegenübergestellt ist EE und SE.
Java SE	Java Standard Edition. Hier sind dem Java-Kern grundlegende Java-Libraries beigelegt. Für die meisten Programmieraufgaben reicht SE völlig aus. SE gegenübergestellt ist ME und EE.
JAXB	Java Architecture for XML Binding. Bibliothek (Standardmässig in Java SE und EE) zum Umwandeln von Java-Objekten in XML und Rückumwandeln von XML in Java-Objekte. Dies wird oft auch Serialisierung und Deserialisierung, beziehungsweise Marshalling und Unmarshalling genannt. Somit ist es möglich Objekte als XML-String über ein Netzwerk zu versenden um sie auf der Gegenseite zu verarbeiten. JAXB wird in AIGS zur Kommunikation zwischen Server und Clients verwendet.
JDK	Java Development Kit (SDK). Sammlung von Entwicklungswerkzeugen (z.B. Compiler, Debugger etc.) für die Java-Entwicklung. Das JDK wird zwar für die Entwicklung (z.B. mit NetBeans oder Eclipse), aber nicht zum Ausführen von Java-Programmen benötigt. Dafür wird nur das JRE benötigt, welches aber ebenfalls im JDK enthalten ist.
JRE	Java Runtime Environment. Java-Kern, welcher benötigt wird um ein Java-Programm ausführen zu können. Das JRE ist die Mindestvoraussetzung um Java verwenden zu können. Für die Entwicklung wird jedoch das JDK benötigt, welches automatisch ein JRE mit sich bringt.

Libraries (Libs) (Bibliotheken)	Libs oder Libraries sind Programmbibliotheken, welche Java-Klassen, Methoden und Enumeratoren enthalten. Diese Bibliotheken können auch von anderen Stellen (z.B. von Oracle) kommen und sind in der Regel in JAR-Dateien gepackt. Libraries werden in der Regel bei der Entwicklung nicht verändert, sondern dienen als Quelle für bestehende Programmbestandteile.
Message (Nachricht)	Eine Message sind über ein Netzwerk übertragene Informationen vom Server zum Client oder umgekehrt. Der Inhalt der Messages muss von beiden Seiten interpretiert werden können, was in AIGS durch die Common-Bibliotheken ermöglicht wird.
Port	Verbindungskanal, dargestellt als Nummer (0 – 65535) zur Kommunikation über Netzwerke. Mit einem Port ist es möglich, dass nur bestimmte Programme auf bestimmte Nachrichten aus dem Netzwerk hören. Bei AIGS ist der Standard-Port: 25123.
Server	Als Server wird bei AIGS das Programm bezeichnet, welches an zentraler Stelle mit allen Clients kommuniziert. Der Server muss allerdings nicht auf einem Web-Server installiert werden. Er kann auf einem normalen PC laufen. Client und Server können auch auf demselben Computer laufen. Es handelt sich dabei aber jeweils um ein eigenständiges Java-Programm. Um mit anderen Clients kommunizieren zu können muss sich der Computer in einem Netzwerk (online) befinden.
WYSIWYG	What you see is what you get: Prinzip für Editoren. Während zum Beispiel Word genau das darstellt, was eingegeben wird, ist dies bei Notepad nicht der Fall, wenn damit HTML bearbeitet wird. WYSIWYG-Editoren haben den Anspruch der vollkommenen Exaktheit. In der Regel gibt es aber trotzdem immer ein paar kleinere Abweichungen zwischen Editor und Endprodukt. So kann ein aus Word generiertes PDF, oder eine mit NetBeans erstellte Swing-GUI von dem abweichen, was ursprünglich editiert wurde.