CS108 Programmier-Projekt
Fachbereich Informatik
Universität Basel, FS 2014

Projekt-Handbuch

Reto Furger
Maximilian Kratt
Sebastian Sacher
Jannis Vamvas

Tutor: Alexander Stiemer

Abgabe: 14. Mai 2014

Inhalt

1 Spielbeschreibung

1.1 Spielregeln

Imuri ist ein Strategie-Brettspiel für zwei bis vier Spieler, erschienen 1985. Unsere Umsetzung in Java hält sich – bis auf die variable Spielfeldgrösse – ganz an die analoge Vorlage.

Auf einem Raster von $n \times n$ Punkten platzieren die Spieler abwechselnd Türme. Die Türme werden automatisch durch ein Mauerstück verbunden, falls zwei Bedingungen erfüllt sind:
Erstens kann ein Turm *A* nur mit Turm *B* verbunden werden, wenn ausschliesslich Türme, die auf gleicher Reihe, Linie oder Diagonale mit *A* liegen, näher an *A* sind als *B*. Anschaulich heisst dies, dass *A* und *B* in Springer-Distanz zueinander liegen müssen. Alle Mauerverbindungen, welche diese Bedingung erfüllen, sind auf dem Spielbrett eingezeichnet.
Zweitens darf die Linie zwischen den Türmen nicht durch eine andere Mauer blockiert werden.

Bevor sie ihren Turm setzen, können die Spieler eine beliebige Zahl ihrer Türme wieder abreissen. Abhängige Mauern werden automatisch entfernt.

Gewonnen hat der erste Spieler mit einer durchgehenden, beide Ränder des Spielbretts miteinander verbindenden Mauer. Die Spieler bzw. Teams haben unterschiedliche Zielvorgaben: Team 1 muss den oberen Spielbrettrand mit dem unteren verbinden, Team 2 den linken mit dem rechten.

1.2 Teams

Team 1 und Team 2 sind die Kontrahenten in jeder Imuri-Partie. Jedes Team sollte einen oder zwei Spieler umfassen. Folglich sind drei Arten von Imuri-Partien denkbar: 1 Spieler gegen 1, 2 gegen 1 oder 2 gegen 2.

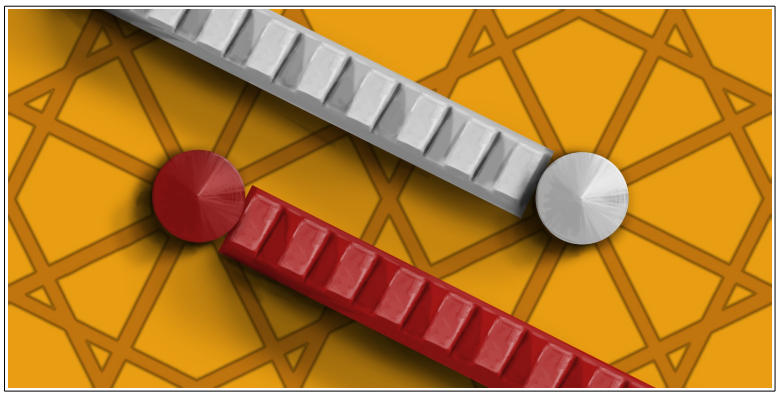
Beim Beitritt zu einer Partie wird ein Spieler automatisch dem kleinsten Team mit der niedrigsten Ordnungszahl zugewiesen. Beispiel:

1. Spieler 1 eröffnet die Partie und wird Team 1 zugewiesen.
2. Spielerin 2 tritt der Partie bei und wird Team 2 zugewiesen.
3. Spieler 3 tritt der Partie bei und wird Team 1 zugewiesen.

In Teams mit zwei Spielern wechseln sich die Spieler mit ihren Zügen ab.

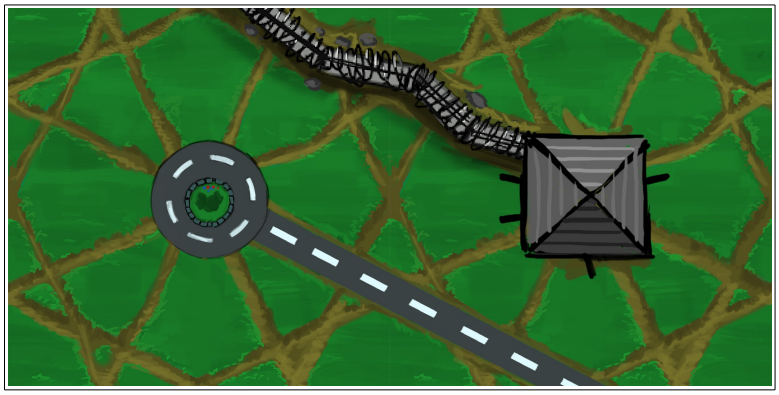
1.3 Designvarianten

Wer eine Partie eröffnet, kann zwischen zwei verschiedenen Spielbrettdesigns auswählen.
Das Design *Classic* orientiert sich am Original von 1985:



Team 1 spielt hier stets rot, Team 2 stets weiss.

Das Design *Streets vs. Walls* stellt die Partie als Wettlauf zwischen Strassenbauern und Isolationisten dar:



Team 1 baut stets Türme und Mauern, Team 2 stets Kreisel und Strassen.

2 Struktur des Programms

Unsere Implementierung von Imuri besitzt eine Client-Server-Architektur. Ein separat betriebenes Server-Programm dient der Verwaltung und Modifikation des Spielerliste, des Chats und der Imuri-Partien. Der Client fungiert als Schnittstelle zwischen Spieler und Server. Er stellt die grafische Benutzerschnittstelle (GUI) bereit. Die Kommunikation zwischen Client und Server wird durch ein textbasiertes Netzwerk-Protokoll ermöglicht.

2.1 Hauptklasse

Beim Starten des Jar-Archivs wird zunächst die Main-Methode der Hauptklasse `Imuri` aufgerufen. Falls die Kommandozeilenargumente korrekt formatiert sind, wird nun entweder die Main-Methode von `Server` oder von `ClientCore` aufgerufen.

2.2 Client

`ClientCore` stellt mithilfe der von Imuri übergebenen Daten eine Socket-Verbindung zum Server her, indem eine Instanz des Threads `SocketHandler` gestartet wird. Ferner verwaltet `ClientCore` ein Verzeichnis `games` der Partien, denen der Spieler beigetreten ist.

Textzeilen, die über die Socket-Verbindung eingehen, werden von `SocketHandler` geparst. Falls sie dem im enum `NetProtocol` definierten Format entsprechen, ruft `SocketHandler` entsprechende spezielle Methoden auf, um den Input weiter zu verarbeiten.

Bezieht sich der Input des Servers auf eine bestimmte Partie, ermöglicht die Hashmap `ClientCore.games` den Zugriff auf den zugehörigen `GameController`. Dieser stellt Methoden für die Darstellung und Veränderung der Spiel-GUI bereit. Er ist der Controller im Model-View-Controller-Schema der Spiel-GUI und verwaltet eine Instanz von `ClientGame` als Model und eine Instanz von `GameFrame` als View.

Ein `GameFrame` setzt sich aus den JPanels `BoardPanel` und `StatusPanel` zusammen. Das `BoardPanel` dient der Darstellung des Spielbretts im oberen Teil des `GameFrame`; das `StatusPanel` stellt links (im JPanel `messages`) spielbezogene Nachrichten des Servers dar. Rechts (im JPanel `numbers`) wird die Zahl verbliebener Turm- und Mauerstücke angezeigt.

Allgemeine Inputs des Servers werden an den `LobbyController` weitergeleitet, der wiederum eine Instanz von `ClientLobbyGUI` als View verwaltet.

2.3 Server

Auf Server-Seite sind Instanzen von `ClientThread` für die Verwaltung der Socket-Verbindungen zu den Clients zuständig. Die Hauptklasse `Server` führt Verzeichnisse für die einzelnen `ClientThread`s sowie für die Spieler (`Player`) und Imuri-Partien (`ServerGame`).

ClientThread parst eingehende Anfragen des Clients und verarbeitet sie – falls sie dem im enum `NetProtocol` definierten Format entsprechen – mithilfe spezieller Methoden. So reagiert zum Beispiel die Methode `ClientThread.placeTower(NetProtocol)` auf einen Befehl der Form `NetProtocol.placeTower`, indem sie prüft, ob der zum Thread gehörige Spieler im genannten Spiel am Zug ist und ob das Setzen des Turms gemäss `Rules` regelkonform ist. Anschliessend wird das `ServerGame` entsprechend modifiziert, mithilfe von `BoardSearch` nach einem Gewinner gesucht und allenfalls eine nächste Runde eingeleitet.

2.4 Spiele

`ClientGame` und `ServerGame` sind Unterklassen von `Game`. Jede Instanz von `Game` repräsentiert eine einzelne Imuri-Partie. Sie speichert den Zustand des Spielbretts als `Board`, eine Liste von `Teams` sowie Metadaten wie den Namen der Partie, die Spielbrettgrösse, das gewählte Design usw.

Die Methode `Game.toStringArray()` kodiert den aktuellen Spielzustand in Form eines String-Arrays. Dessen Elemente entsprechen den Parametern des Netzwerkprotokoll-Befehls `NetProtocol.gameState`. Damit kann der Server nach jedem Spielzug den kompletten Spielzustand an die Clients übermitteln. Der Konstruktor `Game(String[])` erlaubt es, die Spielbeschreibung wieder auszulesen.

Ein `Team` definiert sich durch eine Liste von Spielern sowie eine `id` (1 oder 2). Die `id` wird vor allem von `Board` genutzt, damit die Turm- und Mauerdaten einfach in Integer-Matrizen gespeichert werden können.

Für weitere Details verweisen wir auf das Javadoc-Dokument.

3 Benutzeranleitung

3.1 Systemanforderungen

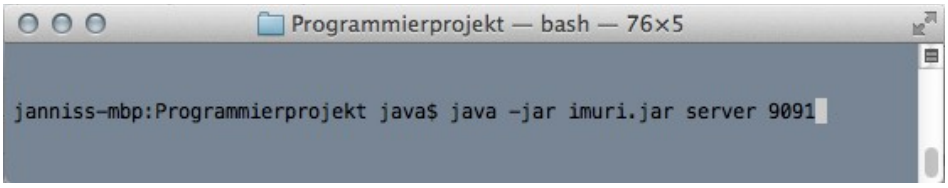
- Java SE Runtime Environment 7

3.2 Server starten

Starten Sie `imuri.jar` mit folgendem Kommandozeilenbefehl:

```
java -jar imuri.jar server <listenport>
```

Ersetzen Sie dabei `<listenport>` durch eine freie Portnummer. Beispiel:

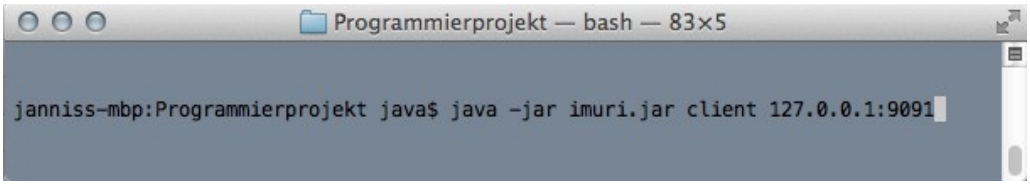


3.3 Client starten

Starten Sie `imuri.jar` mit folgendem Kommandozeilenbefehl:

```
java -jar imuri.jar client <serverip>:<serverport>
```

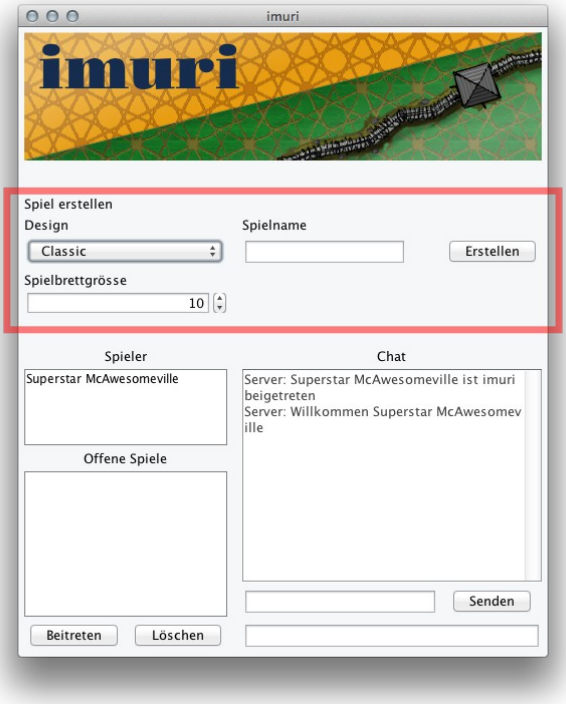
Ersetzen Sie `<serverip>` durch die IP des Servers und `<serverport>` durch die oben gewählte Portnummer. Beispiel:



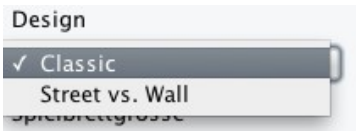
Falls eine Verbindung zum Server hergestellt werden konnte, werden Sie aufgefordert, Ihren Nickname zu wählen. Wählen Sie einen Namen und klicken Sie auf „Ok“:

3.4 Eine Partie eröffnen

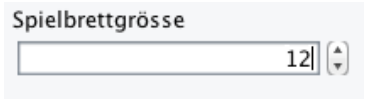
Sie können das mittlere Drittel der Lobby benutzen, um eine neue Imuri-Partie zu eröffnen:



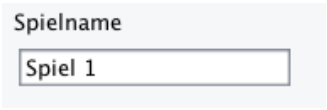
1. Wählen Sie in der Drop-Down-Liste das gewünschte Spielbrett-Design aus:



2. Wählen Sie die gewünschte Seitenlänge des Spielbretts in Türmen aus. Praktikabel sind Werte zwischen 7 und 20.



3. Geben Sie einen Spielnamen ein:

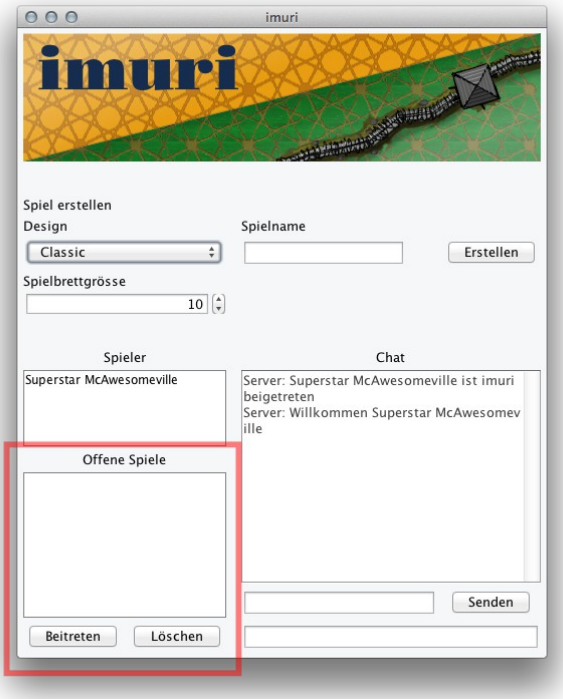


4. Klicken Sie auf „Erstellen“.

Beim Erstellen eines Spiels treten Sie dem Spiel automatisch bei. Sie werden Team 1 zugeteilt und spielen somit auf der Vertikalen mit Rot (*Classic*-Design) bzw. Mauern (*Streets vs. Walls*).

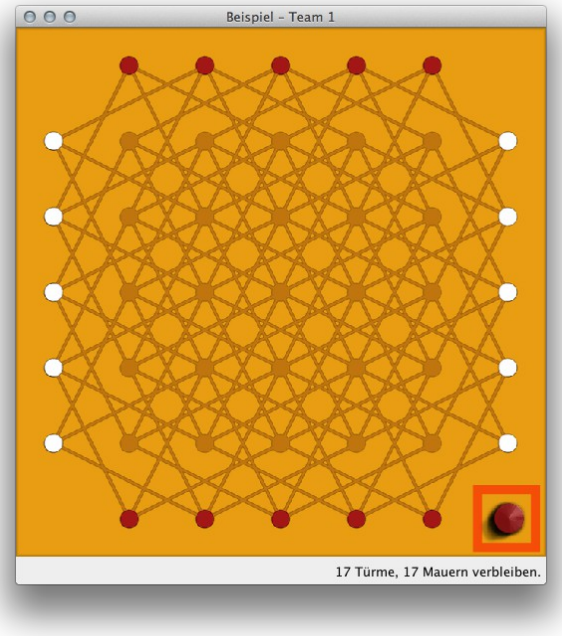
3.5 Einer Partie beitreten

Benutzen Sie die linke untere Ecke der Lobby, um einer Partie beizutreten:



Wählen Sie in der Liste der offenen Spiele das gewünschte Spiel aus und klicken Sie auf „Beitreten“.

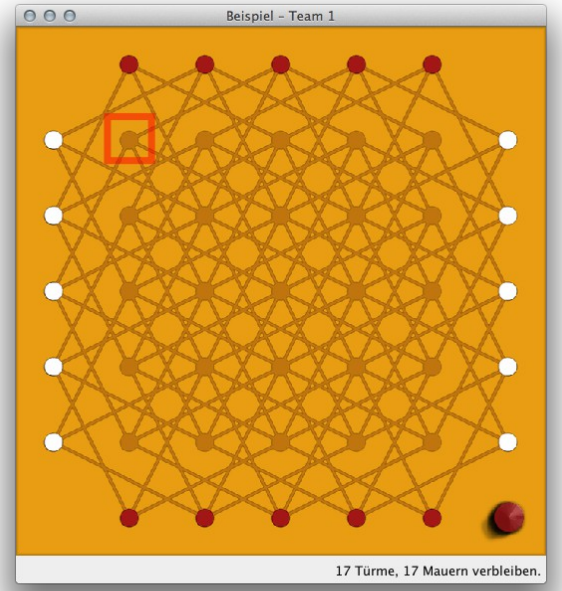
Welchem Team Sie zugeteilt werden, ist vom Zeitpunkt des Beitretens abhängig (vgl. Abschnitt 1.2). Sie können Ihre Team-Nummer dem Chat oder dem Titel des Spiel-Fensters entnehmen. Ferner wird Ihre Spielfarbe in der unteren linke Ecke des Spielbretts angezeigt:



3.6 Einen Turm bauen

Stellen Sie sicher, dass jedes Team mit mindestens einem Spieler besetzt ist und dass alle Spieler aktuell mit dem Server verbunden sind.

Klicken Sie dann auf einen unbebauten Punkt:



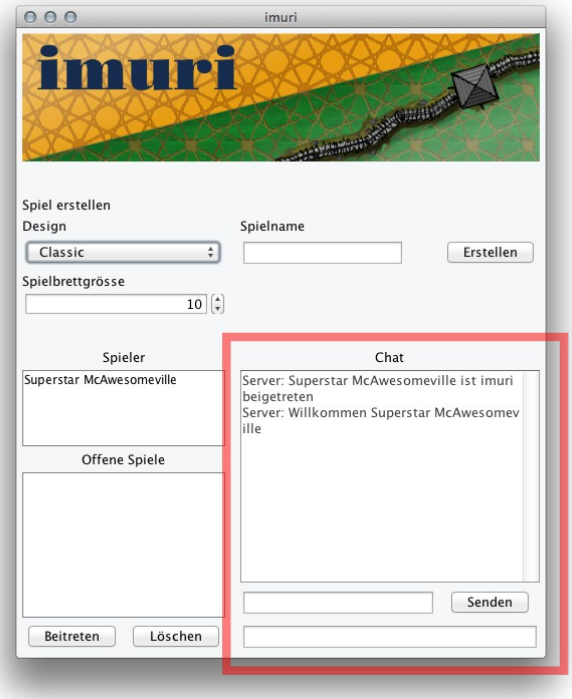
3.7 Einen Turm entfernen

Stellen Sie sicher, dass jedes Team mit mindestens einem Spieler besetzt ist und dass alle Spieler aktuell mit dem Server verbunden sind.

Klicken Sie dann auf einen Ihrer Türme, um ihn zu entfernen.

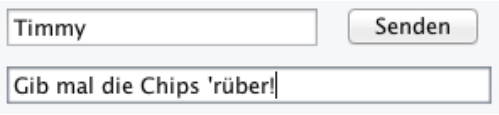
3.8 Eine Chat-Nachricht schreiben

Benutzen Sie die rechte untere Ecke der Lobby, um eine Chat-Nachricht zu schreiben:



1. Geben Sie im oberen Textfeld den Namen des Empfängers ein oder schreiben sie „all“, falls die Nachricht an alle gehen soll:

2. Schreiben Sie die Chat-Nachricht in das untere Textfeld und klicken Sie auf „Senden“:

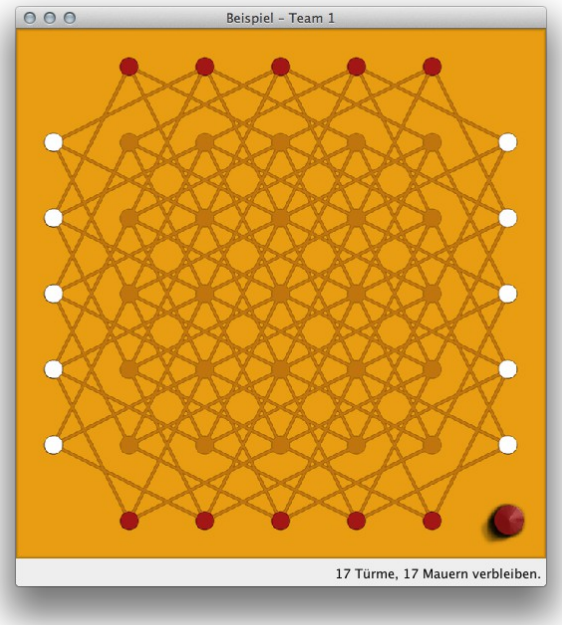


4 Beispielsession

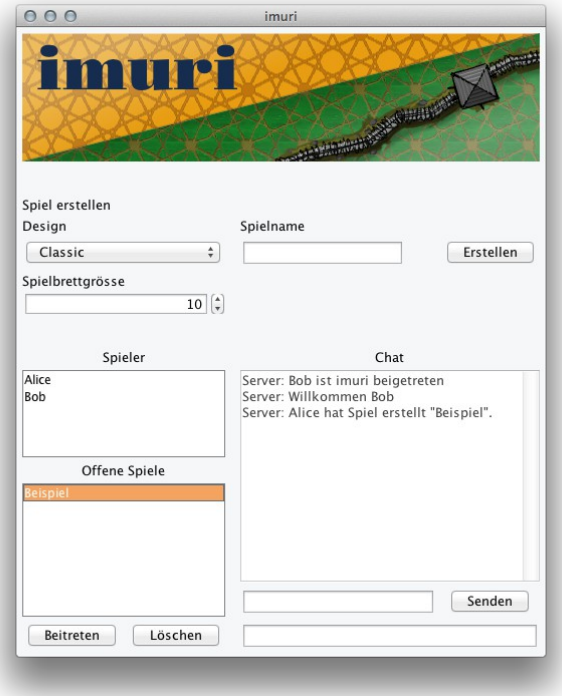
- Alice startet den Server.
- Alice startet den Client und meldet sich unter dem Namen „Alice“ an. Es öffnet sich die Lobby:

- Bob startet den Client und meldet sich unter dem Namen „Bob“ an:

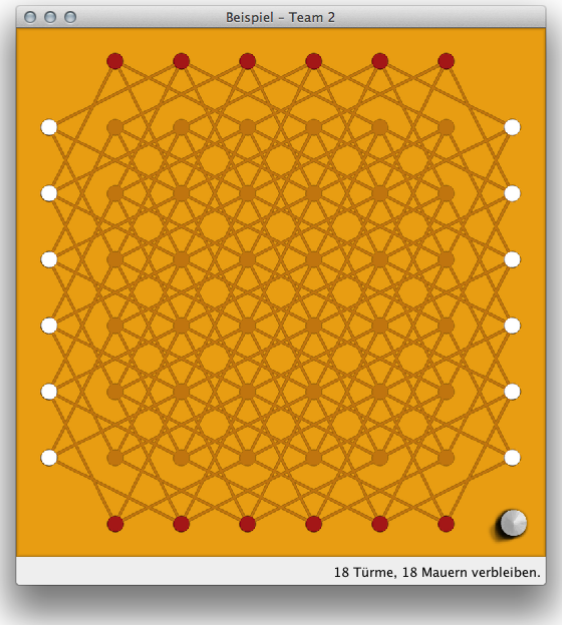
- Alice erstellt das Spiel „Beispiel“. Das Spielfenster erscheint.



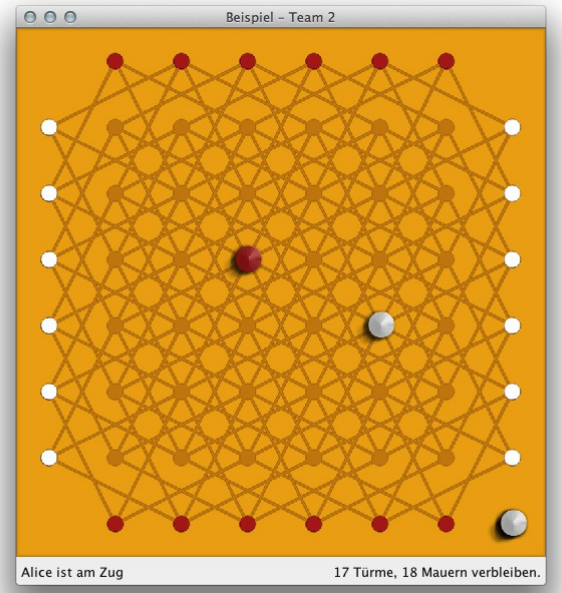
- Bob sieht im Chatfenster und in der Liste der Spiele, dass Alice das Spiel „Beispiel“ erstellt hat. Er tritt dem Spiel bei.



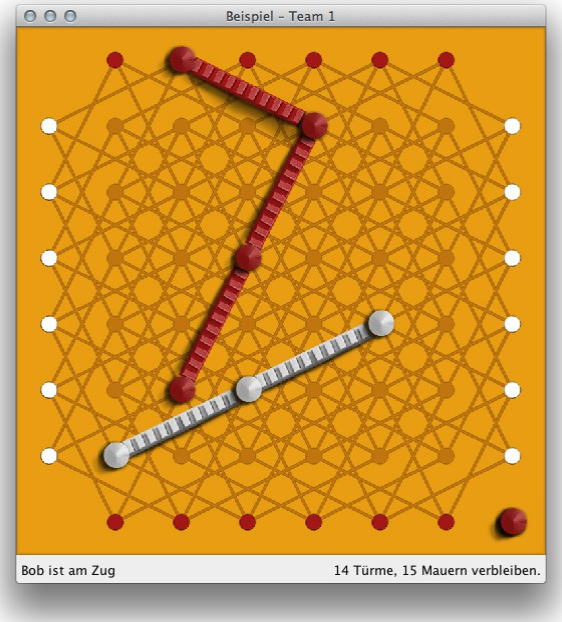
- Bob tritt dem Spiel bei. Sein Spielfenster öffnet sich:



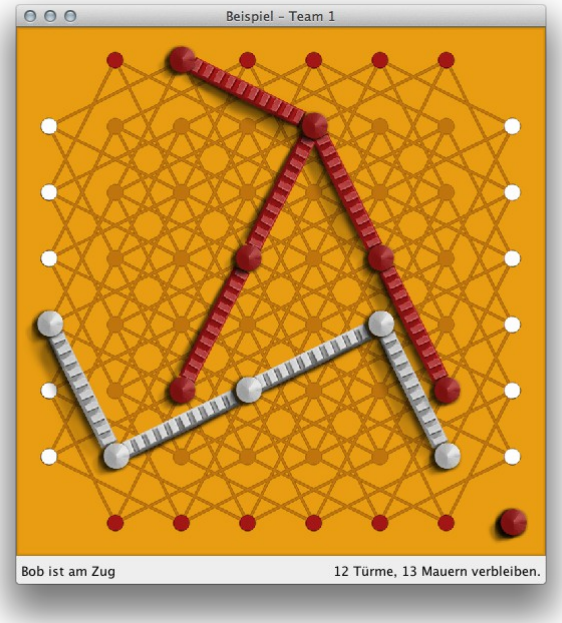
- Alice und Bob machen je einen Zug:



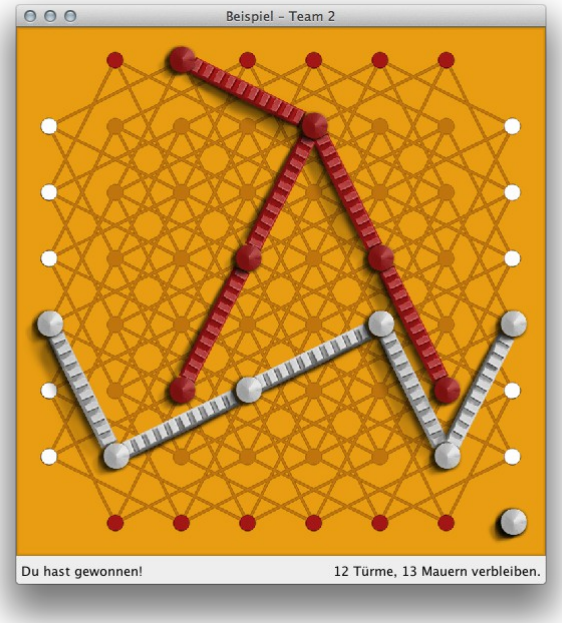
- Alice und Bob spielen weiter.



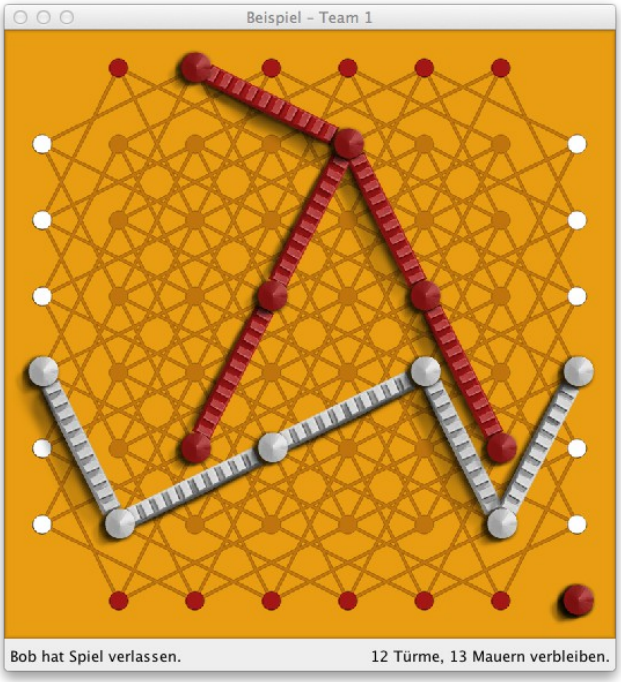
- Alice hat einen Fehler gemacht. Das Spiel entwickelt sich zu Bobs Gunsten.



- Obwohl Bob nicht optimal spielt, hat Alice keine Möglichkeit mehr, seinen Sieg aufzuhalten.



- Bob schliesst das Spielfenster. Alice wird in der Statusleiste darüber benachrichtigt:



- Alice schliesst das Spielfenster ebenfalls. Das Lobby-Fenster ist weiterhin geöffnet:

- Alice und Bob beenden ihre Client-Programme, indem sie die Lobby-Fenster schliessen.

5 Software-Qualitätsmanagement

Um die Qualität der Software sicherzustellen und um Programmierfehler zu vermeiden, haben wir mehrere verschiedene Techniken angewendet.

5.1 Team-Kommunikation

Zunächst unterzogen wir Code, den wir alleine geschrieben haben, in der Gruppe einen informellen Review. Wir hielten einander ständig über die wesentlichen Ideen und Konventionen der Programmelemente auf dem Laufenden. Wir führten ferner ein Projekt-Tagebuch in Form eines Google-Dokuments, um die Arbeit im Team zu organisieren.

Wir bemühten uns, sämtliche Methoden und erklärungsbedürftige Datenfelder in einem Javadoc-kompatiblen Format zu kommentieren. Zugleich nahmen wir uns vor, wichtige Änderungen in Stichworten im Changelog des Repository zu beschreiben. Diesen Vorsatz haben wir allerdings, wie wir rückblickend feststellen, nicht immer konsequent eingehalten.

5.2 Tests

Um unsere Programme zu testen und Fehler aufzuspüren, schrieben wir einerseits massgefertigte, nicht-standardisierte Testklassen. So bedienten wir uns beispielsweise in einem frühen Entwicklungsstadium der Methode `ClientGame.printOutBoard()`. Damit konnten wir das Spielbrett als „ASCII art“ auf der Konsole skizzieren. Fast alle dieser Testklassen haben wir nach Gebrauch wieder gelöscht und nie ins Repository hochgeladen.

Zugleich verwendeten mit dem JUnit-Test `BoardTest` eine standardisierte und kommentierte Testklasse, um die Funktionalität der Spielbrettrepräsentation zu testen. Mittels `BoardTest` konnten wir alle public-Methoden der Klasse `Board` auf charakteristische Situationen sowie auf unerlaubte Eingabewerte überprüfen.

Als wesentlichen Vorteil des JUnit-Tests sehen wir seine Wiederverwendbarkeit. Wenn wir Änderungen am Code vorgenommen hatten, konnten wir uns mittels des Unit-Tests vergewissern, dass die Funktionalität des Codes die gleiche geblieben war. Eigentliche Implementierungsfehler haben wir mit dem Unit-Test zwar nicht mehr gefunden. Allerdings half uns der Test, Lücken in der Definition der public-Methoden zu erkennen. Zum Teil hatten wir vorher nicht festgelegt, wie die Methode auf ungültige Argumente reagieren soll.

Im Rückblick erscheint uns der JUnit-Test gegenüber den nicht-standardisierten Tests besser geeignet für die Softwareentwicklung im Team. Die Vollständigkeit und die Wiederverwendbarkeit rechtfertigen den erhöhten Schreibaufwand. Es bleibt aber festzuhalten, dass sich nicht alle Arten von Klassen gleichermassen für einen JUnit-Test eignen.