Vorab: Die Regeln von Monopoly – Mega Edition können hier nachgelesen werden!

Inhaltsverzeichnis

1. Aufbau

1.1 Projektstruktur

Das Projekt ist in die Bereiche Client, Common und Server aufgeteilt. Wie die Namen schon sagen, haben Klassen im Package Client nur mit dem Client zu tun, während Klassen im Package Server nur mit dem Server zu tun haben. Klassen im Package Common werden überall genutzt. Alle Packages sind für eine bessere Übersicht noch weiter in verschiedene Bereiche unterteilt, da unser Projekt 107 Klassen zählt.

1.2 Hauptklasse

Die Hauptklasse Monopoly enthält die Main Methode, die beim Start von Monopoly ausgeführt wird. In dieser Methode wird das Programm vorbereitet, der Server des Gerätes gestartet und die GUI erstellt. Außerdem werden hier Server und GameState für alle zugänglich gespeichert.

1.3 Spieldaten

Daten für das Spiel werden im Wesentlichen in acht Klassen gespeichert. In den Enums Street, TrainStation und Plant werden alle Daten zu den kaufbaren Feldern inklusive Namen der Besitzer gespeichert. Spielerspezifische Daten werden in einer Map auf dem Server in Form von Objekten der Klasse Player gespeichert. Daten für die Kartenstapel werden in den Klassen CommunityCard, EventCard und BusCard gespeichert. Alle übrigen Daten werden in einem Objekt der Klasse GameData auf dem Server gespeichert.

2. Graphische Benutzeroberfläche

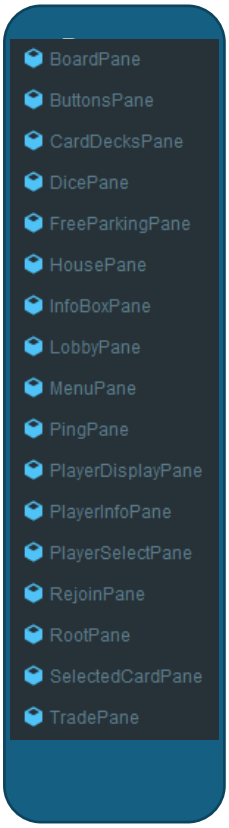
2.1 Texturen und Bilddateien

Wir haben uns bewusst dafür entschieden alle Figuren, Hintergründe, und die meisten sonstigen Texturen im Pixel Art-Stil selbst zu gestalten, da uns dies sehr viel Freiheit lässt das Projekt selbst zu gestalten und eigene Ideen einzubringen. Zudem lassen sich Texturen im „Pixel Art-Stil“ relativ einfach erstellen, was ideal für uns ist, da wir nicht zu viel Zeit in das Aussehen investieren wollten. Zur Implementierung der graphischen Benutzeroberfläche haben wir uns für Java-Swing entschieden.

2.2 Ansatz der Benutzeroberfläche

Unsere erste GUI haben wir PrototypeMenu genannt. Zuerst haben wir versucht, die gesamte Benutzeroberfläche in einer Klasse zu implementieren. Jedoch erwies sich dies schnell als sehr unübersichtlich, da wir eine Vielzahl von Bildern und Texturen benötigten. Beim Aktualisieren der GUI sind wir auf ein weiteres Problem gestoßen: Schon bei der kleinsten Änderung, also wenn sich beispielsweise die Anzahl der Häuser auf einem Feld änderte, wurden die gesamten angezeigten Elemente neu erstellt. Dabei wurden zunächst alle Elemente von der Oberfläche entfernt und anschließend mit den Veränderungen wieder neu hinzugefügt. Diese Vorgehensweise führte dazu, dass Elemente auf dem Bildschirm flackerten.

2.3 Verbesserung der Benutzeroberfläche



Um diese Probleme zu lösen, haben wir die angezeigten Informationen auf verschiedene Panes verteilt. Dadurch können wir die Panes mit separaten Updatemethoden individuell aktualisieren. Dies sorgt nicht nur für Übersichtlichkeit, sondern auch für eine bessere Performance. Die RootPane beinhaltet alle Panes als public Attribute. Diese werden auf verschiedenen Layer angeordnet, da sich einige Panes überschneiden. Die meisten Panes haben eine init(), reset() und update() Methode. Diese werden hauptsächlich im PrototypeMenu und auf Anfrage des Servers ausgeführt. In der neuen Version sind, bis auf kleine Ausnahmen, alle Elemente auf den Panes schon von Anfang an vorhanden und werden nur sichtbar und unsichtbar gemacht, anstatt gelöscht und neu erstellt zu werden. Zusammen sorgt das dafür, dass alles flüssig angezeigt wird. Und ja, unsere Benutzeroberfläche heißt immer noch PrototypeMenu.

2.4 Aufbau des Spiels

Wir haben die Benutzeroberfläche in drei Bereiche aufgeteilt:

Menü:

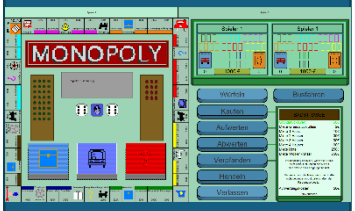
Hier Startet der Spieler, wenn er Monopoly öffnet. In diesem Bereich kann der Spieler eine eigene Runde Monopoly erstellen, oder einer schon existierenden Runde Beitreten.

Lobby:

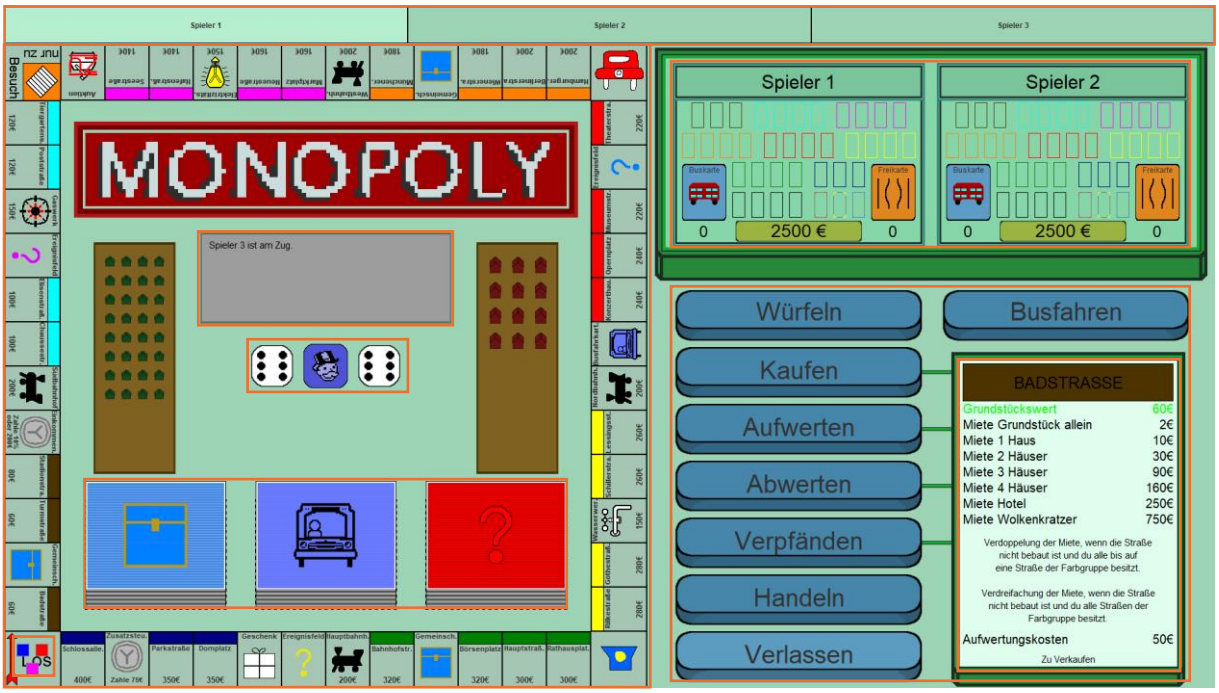
Dieser Bereich dient als Warteberich und dazu, Einstellungen zu ändern, solange noch nicht alle Spieler bereit sind. Dort kann man vom Spielernamen bis zum Startgeld alles Wichtige einstellen.

Spiel:

In diesem Bereich wird dann das eigentliche Spiel gespielt. Hier werden das Spielbrett, mögliche Züge und wichtige Informationen angezeigt.



Die meisten Panes sind im Spiel aktiv. Einige sind hier orange umkreist:



3. Netzwerk

3.1 Server und Client

Unser Mehrspielerkonzept

Monopoly unterstützt mehrere Spieler sowohl an einem als auch an mehreren Geräten in einem Lan-Netzwerk (oder mit einem gemeinsamen VPN wie Hamachi).
Um zu verstehen, warum die Netzwerkprogrammierung so wichtig für Mehrspieler-Spiele über mehrere Geräte ist, muss man zuerst verstehen, was der Unterschied zwischen Server und Client ist.

Aufgaben von Server und Client

Der Client ist sozusagen die Ein- und Ausgabe des Spiels. Er nutzt Bildschirm, Tastatur und Maus zum Anzeigen und Weitergeben von Daten. Der Server hingegen ist für den Nutzer unsichtbar. Er kümmert sich darum, dass das Spiel funktioniert, informiert alle Clients über den aktuellen Spielstand und reagiert auf Anfragen der Clients. Startet Monopoly, ist erstmal weder ein Client noch ein Server vorhanden. Ein Server wird aktiviert, sobald ein Spiel erstellt wurde. Ein neuer Client wird erstellt, sobald einem Spiel beigetreten wird. Das kann das eigene gerade erstellte Spiel oder eins auf einem anderen Gerät sein.

Threads

Sowohl auf dem jedem Client als auch auf dem Server läuft jeweils ein Thread, genannt requestThread, also ein nebenbei laufendes Programm, das sich um alle eingehenden Daten kümmert. Außerdem läuft auf Server und Client noch ein pingThread, welcher sicherstellt, dass dauerhaft eine Verbindung vorhanden ist und die Signalverzögerung herausfindet. Um ein erstes Mal Kontakt mit einem neuen Client aufzunehmen, läuft auf dem Server zusätzlich der connectionThread. Meldet sich ein neuer Client und ist ein neuer Beitritt erlaubt, wird ein neuer Spieler erstellt und die Verbindung auf beiden Seiten in der Form eines Sockets gespeichert. Mit diesem können Server und Client im weiteren Verlauf gegenseitig Daten austauschen.

Kommunikation mit dem Server

Über einen Socket kommt man an einen InputStream und einen OutputStream. Wenn etwas auf einem Gerät in den InputStream eines Sockets geschrieben wird, kommt es in dem OutputStream des dazugehörigen Sockets auf dem andern Gerät an. Aus diesen Streams können DataStreams erstellt werden, welche zwei zusätzliche Methoden, writeUTF und readUTF, aufweisen. Mit diesen Methoden können neben Bytes auch Strings übertragen werden. Bei der Kommunikation werden Objekte in Strings umgewandelt, versendet, und wieder zurückgewandelt.

3.2 Message

Warum wir Messages nutzen

Wenn Daten zwischen Server und Client übertragen werden, muss das Zielgerät auch wissen, was es mit den Daten anfangen soll. Außerdem muss nach einer Umwandlung von Objekten in Strings bei der Rückwandlung die Klasse bekannt sein. Daher werden nur Messages kommuniziert bzw. versendet.

Aufbau der Message Klasse

Die Message Klasse besteht aus einem Array von Objekten und einem MessageType. Durch den MessageType ist der Zweck der Nachricht bekannt und somit auch, wie mit ihr zu verfahren ist.

JSON und Jackson

Objekte der Message Klasse müssen, wie bereits erwähnt, zum Versenden in Strings umgewandelt und anschließend wieder zurückgewandelt werden. Dazu nutzen wir die Jackson-Library. Diese wird in der Json Klasse verwendet. Über mehrere Methoden können Objekte, JsonNodes und Strings gegenseitig umgewandelt werden. Dafür müssen die Objekte jedoch einige Kriterien erfüllen. Zum einen müssen alle Attribute public sein. Zum anderen muss ein Konstruktor ohne Parameter zur Verfügung stehen. Außerdem müssen die Attribute entweder einfache Datentypen wie Integer oder Strings sein, Listen oder Maps einfacher Datentypen sein, oder Klassen sein, die ebenfalls diese zwei Kriterien erfüllen. Aus diesem Grund sind der Array von Objekten und der MessageType in der Message Klasse public.

Senden und Empfangen

Zum Senden ist der zum Ziel gehörige Socket nötig. Über die Methode writeUTF wird dann das Message Objekt im Json Format als String versendet. Dieser kommt dann auf dem Zielgerät an und wird in dem RequestThread auf Server oder Client verarbeitet. Dabei wird die messageReveived Methode aufgerufen, welche den String wieder in ein Message Objekt zurückwandelt. Anschließend wird ein Switch-Case-Statement mit dem MessageType aufgerufen und entsprechend gehandelt.

3.3 UnicastRemote

Warum wir UnicastRemote nutzen

UnicastRemote bietet eine weitere Möglichkeit der Kommunikation. Clients können so direkt Methoden auf dem Server ausführen. Vorteilhaft ist, dass die Rückgabewerte direkt verarbeitet werden können. So können beispielsweise über die Methode getPlayers() auf dem Client alle Spieler vom Server direkt als Objekte erhalten werden. Allerdings sind Messages trotzdem noch nötig, da diese Art der Kommunikation nur einseitig von Client zu Server funktioniert.

Änderungen der Server Klasse

Zuerst einmal wird eine neue Klasse, das IServer Interface benötigt. Die Methoden, die von den Clients ausgeführt werden können, müssen im IServer Interface stehen und eine RemoteException werfen können. Dieses erweitert zudem das Interface Remote. Der Server implementiert dann dieses Interface und die dazugehörigen Methoden. Zudem muss der Server auch die Klasse UnicastRemoteObject erweitern.

Das Serializable Interface

In den Methoden im IServer Interface werden neben einfachen Datentypen, Listen und Maps auch Objekte der Klasse Player und ServerSettings zurückgegeben oder als Parameter angefordert. Damit das möglich ist, müssen beide Klassen das Interface Serializable implementieren. Es enthält allerdings keine zu implementierenden Methoden.

Verbindung mit dem Server

Um die Verbindung zu ermöglichen, wird im Konstruktor des Servers eine Registry auf dem Port 1199 erstellt und an den Server gebunden. Auf dem Client wird über die IP-Adresse und den Port 1199 die Registry erhalten. Dadurch kann über ein IServer Interface auf den Server zugegriffen werden. Dieses wird als Attribut gespeichert. Über die Methode serverMethod kann auch das PrototypeMenu auf das Objekt und somit den Server zugreifen.

3.4 Packet

Warum wir Packets nutzen

Wenn bisher etwas Neues zwischen Server und Client kommuniziert werden sollte, musste ein neuer MessageType zu dem Enum und in den messageReceived Methoden in den Klassen Server und Client ergänzt werden. Beim Abschicken einer Nachricht musste dann der jeweilige MessageType sowie ein Array mit den Daten übergeben werden. Mit einer zunehmenden Anzahl an verschiedenen Nachrichten wurden die messageReceived Methoden immer unübersichtlicher. Daher nutzen wir mittlerweile stattdessen Packets.

Aufbau eines Packets

Eine Klasse, die das Interface Packet implementiert, besteht im Wesentlichen aus vier Methoden: Im Konstruktor wird das Packet, wie jede Klasse, initialisiert. Die Methode serialize wandelt das Packet in einfache Daten um. Diese können dann mithilfe einer Message versendet werden. Die Methode deserialize ist static und wandelt die übergebenen Daten wieder in das Packet um, wenn das Packet auf dem jeweiligen Gerät angekommen ist. Anschließend wird die Methode handle ausgeführt. Diese hat als Parameter eine Side, welche benötigte Daten bereitstellt.

ServerSide und ClientSide

Eine Side ist, je nachdem ob das Packet auf einem Client oder einem Server ankommt, entweder eine ServerSide und enthält den Server oder eine ClientSide und enthält den Client und die RootPane. Daher erweitern alle Packets die abstrakte Klasse S2CPacket oder C2SPacket, anstatt das Packet Interface direkt zu implementieren. Diese Klassen übernehmen die handle Methode. Stattdessen müssen die Packets dann die abstrakte Methode handleOnClient oder handleOnServer überschreiben.

DataWriter und DataReader

Die zu kommunizierenden Daten werden in der Methode serialize in einen DataWriter geschrieben und in der Methode deserialize aus einem DataReader gelesen. Durch diese Klassen ist sichergestellt, dass nur mit einer Message kompatible Datentypen verschickt werden. Die Daten müssen in derselben Reihenfolge gelesen und geschrieben werden. Dadurch muss man keinen Index angeben und es kommt nicht zu Komplikationen. Im DataWriter können die Daten mithilfe der Methode getData in Form eines Arrays von Objekten erhalten und anschließend in einer Message versendet werden. Mit dem Array in der ankommenden Nachricht kann dann ein DataReader konstruiert und die zum Packet zugehörige deserialize Methode ausgeführt werden.

Veränderung der Message Klasse

Bisher bestand eine Message aus einem MessageType und einen Array mit Objekten als Daten, welche, wie schon erwähnt, bestimmte Bedingungen erfüllen müssen. Durch die Einführung von Packets werden nur noch deren Daten über Messages übertragen. Daher beinhaltet die neue Version der Message Klasse nun die Klasse des Packets sowie weiterhin ein Array von Objekten als Daten. Über die Klasse des Packets kann in der Klasse Packets die deserialize Methode erhalten werden.

PacketManager

Packets werden über den PacketManager versendet sowie verarbeitet. Über die Methoden sendS2C kann ein Packet an bestimmte Spieler oder direkt an einen Socket gesendet werden. Über die Methode sendC2S kann ein Packet an den Server gesendet werden. In allen Fällen muss ein Consumer<Exception> angegeben werden. Dieser wird ausgeführt, falls ein Fehler auftritt. Zum Versenden wird eine Message aus der Klasse des Packets und der Daten aus einem DataWriter nach ausführen der serialize Methode konstruiert und versendet. Diese drei Methoden werden an verschiedenen Stellen im Code ausgeführt. Zum Verarbeiten dient die Methode handle. Diese benötigt eine Side und die Message, mit der das Packet ankommt und wird in den Methoden messageReceived in den Klassen Server und Client ausgeführt. Über die Methode packet wird das Packet der Nachricht durch den Aufruf der deserialize Methode mit einem DataReader, erstellt aus den gesendeten Daten, rekonstruiert. Anschließend wird die handle Methode des Packets mit der gegebenen Side ausgeführt.

Beispiel: InfoBoxS2CPacket

Das InfoBoxS2CPacket ist ein S2CPacket, welches der Server an Clients sendet, damit in der grauen Infobox in der oberen Hälfte der Spielbrettmitte eine neue Zeile angezeigt wird. Im Konstruktor wird der Inhalt der Zeile als String übergeben. In der Methode serialize wird dieser in den DataWriter geschrieben und in der Methode deserialize wieder aus dem DataReader gelesen. In der abstrakten Klasse S2CPacket wird die handle Methode überschrieben und die abstrakte Methode handleOnClient mit dem Client und der RootPane ausgeführt. Diese greift über die RootPane auf die InfoBoxPane zu, um die neue Zeile anzuzeigen.

4. Events

4.1 Das Konzept

Warum wir Events nutzen

Der Spielablauf von Monopoly ist sehr komplex. Die Spieler haben viele mögliche Aktionen und viele Regeln sind zu beachten. Daher haben wir uns entschieden, den Spielablauf als Events zu implementieren. Für jedes mögliche Ereignis gibt es eine Methode. Diese rufen sich größtenteils gegenseitig auf. Im Event für das Würfeln wird beispielsweise nach Abschluss das Event für die Ankunft auf einem Feld aufgerufen.

Aufbau einer Events Klasse

Die Eventsklasse ist abstrakt. Sie besteht zum einen aus allen einstellbaren Regeln als Attribute und zum anderen aus allen möglichen Events als abstrakte Methoden. Außerdem verfügt die Klasse über einige nur für Unterklassen zugängliche Attribute für das Spielgeschehen, wie unter anderem das letzte Würfelergebnis, eine Liste aller Spielernamen und den Index des aktuellen Spielers und der Methode player(), über welche der aktuelle Spieler anhand des Namens aus der Liste vom Server erhalten wird.

Vorteil der Vererbung

Wir haben uns für eine abstrakte Klasse entschieden, anstatt die Events direkt zu implementieren, um uns die Möglichkeit offen zu halten, verschiedene Spielabläufe zu implementieren. Das Spiel besteht letzten Endes aus zwei Events – MegaEditionEvents und MegaEdition2PlayerEvents. Dabei überschreibt letztere lediglich die Methode des ersteren, welche die minimale Anzahl von Spielern festlegt. Denn Monopoly ist eigentlich für drei Spieler gedacht, kann aber in unserem Fall auch zu zweit gespielt werden, da wir der Einfachheit halber Auktionen und Handel weggelassen haben.

4.2 Implementierung

Spielablauf

Der Spielablauf ist aufgrund der schon erwähnten Komplexität des Spiels immer unterschiedlich. Im Wesentlichen beginnt ein Spielzug mit dem Aufruf von onDiceRole oder onBusDrive. Diese werden über das ButtonC2SPacket aufgerufen. Durch den übergebenen Spielernamen wird sichergestellt, dass der Spieler, der das Packet gesendet hat, auch wirklich dran ist. Nach Würfeln wird onArrivedAtField aufgerufen. Diese Methode führt dann je nach Art des Feldes onArrivedAtPurchasable, onArrivedAtEventField usw. aus. Um den Zug zu beenden, wird über das ButtonC2SPacket onTryNextRound ausgeführt und wenn mayDoNextRound true zurückgibt, wird letztlich onNextRound ausgeführt. Natürlich gibt es noch viele andere Events, die jedoch seltener aufgerufen werden.

Einstellungen

Über den SettingsScreen können alle Einstellungen angezeigt, geändert und wieder gespeichert werden. Dazu wird der Modus, also eine Events.Factory, und das vorherige Event übergeben. Eine Events.Factory ist ein Konstruktor einer beliebigen Eventsklasse. Ist das vorherige Event nicht null, werden die Regeln auf die Werte dieses Events gesetzt. Anschließend können die Regeln beliebig verändert werden. Zuletzt wird anhand der Events.Factory und der ausgewählten Regeln eine neue Instanz einer Eventsklasse erstellt und auf den Server angewendet. Einstellungen können daher nur vom Host bearbeite werden.