BlockTacToe

Patrick Budig, Adrian Ilius, Daniel Niemczyk, Patrick Lang
Sommersemester 2018

FH·W·S
Hochschule
für angewandte Wissenschaften
Würzburg-Schweinfurt

Gliederung

1. Einleitung(Daniel N.)

Mit der veröffentlichung des Bitcoin Protokolls Anfang 2009 wurde eine Technologie erschaffen, die es ermöglichen soll global, grenzenlos und ohne Erlaubnis “Geld” zu verschicken. Wenn man jedoch einmal hinter die Fassade von Bitcoin schaut stellt man fest, dass es sich im Grunde nur um eine Plattform des Vertrauens handelt. Eine Plattform welche es ermöglicht Peer-to-Peer Handel zwischen zwei Nutzern zu ermöglichen ohne die Notwendigkeit eines Mittels Mannes und wenn wir hier von Nutzern reden, ist dies nicht nur auf Personen beschränkt. Maschinen in Form von Autos oder Kühlschränken können dazu programmiert werden sich selbst aufzutanken, bzw die Einkäufe zu tätigen und selbstständig zu bezahlen. Bitcoin ist dabei nur die erste Anwendung der sogenannten open public Blockchains. Man kann die Grundbausteine von Bitcoin nehmen und komplett neue, leicht modifizierte eigene open Blockchains erstellen.
Was ist wenn wir Adressen nicht nur die Möglichkeit geben den aktuellen Betrag der jeweiligen Kryptowährung zu speichern, sondern gleichzeitig auch ein eigenes komplexes Programm darzustellen. Smarte Programme die nicht veränderbar sind und in Zukunft genutzt werden könnten, um selbst wichtige Dokumente fälschungssicher auf der Blockchain zu speichern. Transaktionen, welche somit kein Geld verschicken, sondern Daten jeglicher Art, die verschlüsselt und gehasht in solch einem sogenannten Smart Contract gespeichert werden können.

In diesem Projekt wollen Patrick Budig, Adrian Ilius, Daniel Niemczyk und Patrick Lang als Team es versuchen solch einen Smart Contract mit einer geeigneten Schnittstelle zu entwickeln und programmieren.
Wir haben uns für ein Spiel, in dem Fall TicTacToe entschieden, bzw wurde es uns zum Teil auch so angeboten. Um das Spiel “vernünftig” Spielen zu können haben wir uns für ein Programm in Java entschieden, welches mit dem Smart Contract interagieren kann. Hauptgrund dafür ist das alle 4 von uns diese Sprache beherrschen und es mittlerweile auch für Java nützliche Libraries gibt.

Ob es heute bereits Möglich ist Spiele auf einer open Blockchain zu entwickeln ? Sicherlich!
Die Frage die wir uns als Team gestellt haben ist nicht ob das theoretisch möglich wäre, sondern wie effizient und nutzerfreundlich das Ganze bereits heute funktioniert. Aus folgendem Grund haben wir uns folgendes Ziel gesetzt.

Das Ziel dieses Projektes ist es zu überprüfen ob eine open Blockchain in der Lage ist Spielzüge eines Spiels (hier TicTacToe) in einer annehmbaren Zeit und mit hinnehmbaren Kosten zu bearbeiten und den Gewinner zu ermitteln.

2. Ethereum (Daniel N.)

2.1 Was ist Ethereum

Ethereum ist im Grunde die aktuell zweitgrößte Kryptowährung hinter Bitcoin und wurde 2015 angefangen zu entwickeln. Sie kann ebenfalls als “dezentrale virtuelle Währung” eingesetzt werden. Was Ethereum jedoch auszeichnet ist die Möglichkeit eigene dezentrale Apps, sogenannte Dapps oder Smart Contracts zu entwickeln. Ethereum ist somit der Beginn von der Blockchain 2.0 und wird deshalb oft als “Welt-Computer” bezeichnet.
Sowohl Bitcoin als auch Ethereum nutzten die Blockchain als Datenstruktur in Kombination mit dem Proof-of-Work Algorithmus um dezentralen Konsensus zu erzeugen.
In unsere Projekt haben wir jedoch nicht tatsächlich auf Ethereum programmiert, sondern auf einer Test Blockchain namens Ropsten, welche jedoch die selben Eigenschaft beseitzt wie die Orginale.
Ropsten ist eine von mehreren Test Blockchains, welche vor allem von Entwicklern gern gesehen wird. Auf dieser kann unbegrenzt und prinzipiell kostenlos getestet werden. Kostenlos in dem Sinne, das man jederzeit Ether in nahezu unbegrenzter Menge für diese Blockchain erhalten kann. Unterstützt wird das ganze durch eine online Entwicklungsumgebung namens Remix . Für Ethereum haben wir uns entschieden, da dies aktuell mehr oder weniger die einzige Blockchain ist, auf welcher man bereits problemlos Smart Contract erstellen kann. Der Marktanteil von Ethereum beträgt über 95 %. Auch die meisten sogenannten Inital Coin Offerings(ICO) laufen über die Ethereum Blockchain.
Der Grund wieso wir uns überhaupt darauf eingelassen haben auf der Blockchain etwas zu programmieren ist das Interesse am neuen. Diese Technologie steht erst in den Kinderschuhen und hat enormes Potenzial. Diese Technologie kann die Art und Weise wie wir weltweit miteinander Handeln und Werte austauschen genauso radikal verändern wie es das Internet für die Kommunikation auf der ganzen Welt getan hat. Deshalb schadet es nicht, sich selber mal ein Bild von dem ganzen zu machen.

2.2 Was ist Ether?

Ether ist die Währung von Ethereum. Stand heute (23.07.2018) ist ein Ether 380€ wert. Im Umlauf sind aktuell etwas mehr als 100 Millionen Stück. Ether kann verwendet werden, um sich gegenseitig “Geld” zu schicken, dessen Hauptaufgabe liegt jedoch darin als Sprit zu fungieren. Sprit im Sinne von Transaktionsgebühren, um die Miner für alle Rechenoperationen, welche mit dieser Transaktion einhergehen zu entschädigen. Zum einen für gewöhnliche Transaktionen, zum anderen zum Aufrufen und Bearbeiten von Funktionen in einem Smart Contract. Aktuell kostet eine Transaktion ungefähr 0,00005 Ether was ca 2-3 ct entspricht. Ein Ether ist genauso, wie ein Euro aufteilbar in kleinere Einheiten. Die kleinste Einheit bei Ether heißt wei, wobei ein Ether 10^18 wei sind. Eine so kleine Einheit ist notwendig um tausende kleine Rechnungen durchzuführen.
Um die Transaktionsgebühren berechnen zu können müssen wir klären was Gas ist. Gas ist eine Einheit, die den Rechenaufwand für die Ausführung bestimmter Operationen misst. Während Rechnungen, wie Addition, Multiplikation oder ein Vergleich zwischen einzelnen

keine 10 Gas jeweils kosten, kostet das erstellen von permanenten Variablen mehrere 1000 Gas.
Eine Ethereum Transaktion kostet standardmäßig 21.000 Gas. Der Preis einer beliebigen Operation setzt sich aus dem verbrauchten Gas multipliziert mit dem Preis pro Gas zusammen. Diesen Preis kann man prinzipiell frei bestimmen und es gibt keine Regeln. Jedoch besteht das Aufnehmen von Transaktionen in einen Block aus Angebot und Nachfrage. Das bedeutet, wer am meisten zahlt, dessen Transaktion wird am ehesten durch die Miner abgearbeitet. Aktuell pendelt der Gaspreis zwischen 2-3 gwei. Ein (giga)-wei sind 10^9 wei.
Nehmen wir nun eine Transaktion, welche 21.000 Gas benötigt und bezahlen pro Gas 2 gwei kommen wir auf einen Endbetrag von 42.000 gwei, was wiederrum 0.000042 Ether entspricht und unsere ursprünglichen 0.000050 Ether pro Transaktion widerspiegelt. Dieses Grundverständnis wird benötigt, um zu verstehen, warum es so wichtig ist, seinen Smart Contract Gas effizient zu entwickeln. Denn je komplizierter eine Berechnung in einer aufgerufenen Funktion ist, desto höher ist das verbrauchte Gas und desto höher im Endeffekt die Transaktionsgebühren für den User. Mehr dazu im nächsten Kapitel Smart Contracts.

2.3 Smart Contracts

Smart Contracts sind das was Ethereum ausmacht. Die Anwendungsfälle sind sehr weitreichend und können unsere komplette Gesellschaft verändern. Prinzipiell muss man in der Zukunft einfach sehen was durch diese Verträge alles möglich und effizienter sein wird. Schlussendlich wird es nur die Zeit beantworten wann und ob wir die ersten komplett autonom arbeitenden Unternehmen, Organisationen usw auf einem Smart Contract sehen werden und wie so etwas von der Gesellschaft aufgenommen wird.
Geschrieben werden Smart Contract in der extra dafür geschriebenen Programmiersprache Solidity. Diese Sprache ist stark an Java Script angelehnt.
Für das folgende Kapitel sollen einige Eigenschaften von Solidity hier bereits beschrieben werden. Wie bereits beschrieben ist es sehr ratsam, so Ressourcen effizient wie nur möglich zu arbeiten, denn jede Operation kostet im Endeffekt. Ruft man eine Funktion auf, bzw führt eine Transaktion aus, sind die Befehle msg.sender und msg.value automatisch bekannt. Ersteres ist die Adresse, welche die Funktion aufgerufen hat. Über diese Information verfügt man immer, gleichzeitig über deren Wert. Diese beiden Befehle werden in unserem Smart Contrac ebenfalls öfters benutzt. Ebenfalls gibt es in Solidity das kleine Wort require. Ein require findet sich meistens am Anfang einer Funktion wieder und ist eine Art Bedingung die gelten muss, um weiter fortzufahren. Wird diese nicht erfüllt, wird die Transaktion abgebrochen und nicht in der Blockchain hinzugefügt. Zudem sei gesagt, dass Funktionen welche nur einen Wert auslesen, wie z.B die Id eines Spiels mit dem Signalwort view gekennzeichnet sind. Den Rückgabewert dieser Funktionen kann man einfach auslesen und erhält deren Wert “sofort”.
Es sollte vermutlich erwähnt werden das Smart Contracts, nur weil sie dezentral funktionieren, nicht unbedingt dezentral sein müssen. Es lassen sich Smart Contracts programmieren wo gewissen Bereiche, Funktionen nur für bestimmte Personen zugänglich sind. Das heißt es lassen sich gewisse Schlupflöcher implementieren. Diese können nur

erkannt werden, wenn der komplette Smart Contract Open Source ist. In unserem SmartContract wäre es ebenfalls möglich eine Funktion zu implementieren, welche jeglichen Ether Beträge, welche auf unserem Smart Contract gelagert sind und prinzipiell den Spielern “gehören” an unsere Adresse zu überweisen. Der Aufschrei wäre groß, technisch möglich ist sowas jedoch.

3. BlockTacToe (Daniel N.)

3.1 Überblick

Kommen wir nun endlich zum Herzstück unseres Projektes. Unserem eigenen BlockTacToe Smart Contract. BlockTacToe ist die Realisierung eines herkömmlichen Tic Tac Toe Spieles auf der Blockchain, mit allem was dazu gehört. Das bedeutet, dass sich zwei Parteien, bzw Adressen, egal ob Mensch oder Maschine mithilfe dieses Smart Contracts ein gemeinsames Spiel starten können, um gegeneinander um Ether spielen zu können. Was sich anfangs relativ simpel angehört hat, wurde zunehmend immer komplizierter. Denn das Programmieren von dezentralen Smart Contracts erfordert stellenweise komplett verschiedene Denkansätze als das Programmieren von “normalen” Client-Server Modellen.
Im Folgenden sei nochmals erwähnt, dass jegliche Funktionsaufrufe, welche in diesem Kapitel erklärt werden mithilfe von Java und der Schnittstelle Web3j möglich sind. So gesehen agiert die Blockchain als eine Art Server und unsere Java Application als Client, welche Zugriff auf alle public Funktionen und Datenstrukturen hat.
Da man in Solidity keine Objekte aus Klassen instanziieren kann, stellen wir alle unsere Spiele als struct dar. Ein struct ist, gleich wie in C, eine Aneinanderreihung von Attributen. Man könnte auch sagen: ein Objekt, nur ohne Methoden.

```
struct Game{
        uint id;
        uint bet;
        address player1;
        address player2;
        uint turn;
        uint[] gameHistory;
        uint[] board;
        GameState gameState;
        uint timeSinceLastTurn;
    }
```

Dies ist das Grundgerüst eines jeden BlockTacToe Spiels in unserem Smart Contract. Alle neun Attribute werden beim erstellen eines neuen Spiels automatisch gesetzt. Um die Möglichkeit zu haben auf alle aktiven und jemals gespielten Spiele zurückzugreifen, speichern wir alle Spiele in einem dynamisch wachsenden Array namens games. Die Länge des aktuellen Arrays ist entscheidend für die id eines neuen Spiels, denn diese ist

die Länge des Arrays +1. Das bedeutet, dass das Spiel an Position 0 im Array die id 1 besitzt usw. Um auf ein Spiel mit der id n zugreifen zu können, holen wir uns dieses aus games[n-1] heraus.
Der Betrag um den gespielt wird, wird in der Variable bet gespeichert. Wenn sich beide Parteien geeinigt haben um 5 Ether zu spielen, bedeutet dies in unserem Spiel das dies der Spieleinsatz von jedem Spieler ist. Somit wird insgesamt um 10 Ether gespielt. Nachdem ein Spiel zu Ende ist, wird dieser Betrag zurück an die Spieler gehen. Im Falle eines Siegers, bekommt dieser den kompletten Betrag, falls es zu einem Unentschieden kommt, wird der Betrag zwischen beiden Spielern fair aufgeteilt. Um überhaupt zu wissen wer welcher Spieler ist wurden die beiden Variablen player1 und player2 erstellt. In diesen Variablen werden die jeweiligen Adressen der Spieler gespeichert. Der zu beginnende Spieler ist in unserem Fall player1. Damit beide Spieler dieselbe Wahrscheinlichkeit haben das Spiel zu beginnen, wird dies mittels einer Zufallsfunktion ermittelt.
Wenn man sich einmal das Game struct anschaut, wird man schnell feststellen, dass es keinen aktiven Spieler gibt. Wir lassen diesen einfach immer mit Hilfe von turn feststellen. Am Anfang, bevor noch kein Zug getätigt worden ist, ist turn standardmäßig auf 1 gesetzt. Das bedeutet, dass jedes Mal, wenn turn ungerade, sprich 1,3,5,7,9 bzw turn%2 == 1 ist, ist player1 der aktuell aktive Spieler. Andernfalls ist player2 die aktive Addresse.
Wir mussten uns entscheiden, ob wir das Attribut activePlayer anlegen und nach jedem Spielzug ändern, oder ob wir diesen mit Hilfe von turn jedesmal neu ermitteln. Ersteres erfordert mehr Speicherbedarf, zweiteres mehr Rechenleistung. Im Endeffekt macht es keinen wirklich signifikanten Unterschied, dennoch haben wir uns für diese Variante entschieden, einzig und allein deshalb, um unseren Smart Contract so klein wie möglich zu halten.
Um die Spielzüge zu speichern, benutzen wir zwei verschiedene Arrays. Zum einen das board und zum anderen die gameHistory. In der gameHistory stehen die Züge in der Reihenfolge, in der sie aufgetreten sind. Zuerst einmal sollte jedoch geklärt werden, wie wir die Felder auf einem Tic Tac Toe Feld benennen. Wir haben uns für die Variante 1-9 Entschieden, wobei das Feld oben links den Wert 1 erhält, die Mitte den Wert 5 und die 9 repräsentiert die Ecke unten rechts. Das Array [7,1,8] würde dementsprechend bedeuten, dass Spieler 1 seinen ersten Zug auf Feld 7, in der unteren linken Ecke getätigt hat. Spieler 2 hat daraufhin seinen Zug oben links gesetzt und zum Schluss nochmal Spieler 1 in die Mitte der unteren Reihe.
Prinzipiell würde das doch reichen? Das einzige Problem mit dieser Darstellung ist es, herauszufinden, ob das Spiel gewonnen worden ist oder nicht. Bei einem Tic Tac Toe Spiel gibt es 8 Möglichkeiten das Spiel für sich zu entscheiden. Entweder man besetzt eine der drei waagrechten, senkrechten oder eine der beiden Diagonalen. Aus diesem Grund macht es Sinn die Variable board zu erstellen. Board ist ein Array der festen Länge 9 und wird mit lauter Nullen initialisiert. Eine 0 bedeutet, dass das Feld an dieser Stelle leer, bzw noch nicht gesetzt ist. Führt ein Spieler nun einen Zug aus, wird dieser Zug auf genau dieser Stelle im Array durch eine 1 für Spieler 1, oder 2 für Spieler 2 ersetzt. Mit genau dieser Stelle ist jedoch nur die relative Position gemeint. Das Feld 3, ist die 3. Stelle im Array, was jedoch board[2] ist, da ein Array in Solidity ebenfalls mit Index 0 beginnt. Ein Spielfeld [0,1,2,0,1,0,2,1,0] bedeutet, dass Spieler 1 seine Züge auf dem Feld 2,5 und 8 gesetzt hat.

Möchte man nun überprüfen ob ein Spiel gewonnen worden ist, überprüft man alle 8 Gewinnmöglichkeiten. Wenn man alle 8 Linien als Tupel von 3 Zahlen darstellt, wird man feststellen, dass 2,5,8 ebenfalls eine Gewinnmöglichkeit ist, nämlich die 2. senkrecht Linie durch die Mitte. Mit dieser Art der Darstellung lässt sich eine Überprüfung des Gewinners deutlich effizienter und schneller umsetzen. Dennoch haben wir die gameHistory im Game struct gelassen, um maximale Transparenz zu erhalten. Im Gegensatz zu board, sieht man mit gameHistory die Reihenfolge der gesetzten Züge. Prinzipiell wäre es auch möglich bei jeder Sieges-Überprüfung mithilfe der gameHistory ein lokales Board zu erstellen und mithilfe diesem den Sieger zu überprüfen, wir haben uns jedoch dagegen entschieden.
Gibt es keinen Sieger und das Spielfeld ist voll, endet das Spiel mit einem Unentschieden. Gleichzeitig ist eine Überprüfung des Siegers erst ab dem 5. Zug Notwendig, bzw ab dem 3. Zug des ersten Spielers, da man mit nur 2 selber gesetzten Feldern keine Linie mit 3 Feldern voll besetzten kann.

```
enum GameState{
        RUNNING, TIE, PLAYER1WINS, PLAYER2WINS , PLAYER1TIMESUP , PLAYER2TIMESUP
  }
```

Unser Spiel verfügt über 6 verschiedene Spiel Zustände, wovon immer einer in gameState des Spiels gespeichert wird. Wird ein Spiel angefangen ist es im Status RUNNING. Nur in diesem ist es möglich neue Spielzüge zu erstellen. Nachdem ein Spiel vorbei ist, ist es unmöglich es im nachhinein zu manipulieren bzw zu verändern. Nachdem das Spiel durch einen der Spieler gewonnen worden ist, bzw das Spiel unentschieden ausgegangen ist, wird der gameState des Spiels entweder in TIE, PLAYER1WINS oder PLAYER2WINS geändert und eine aftergame Funktion aufgerufen wo der Spieleinsatz automatisch verteilt wird.
Die timeSinceLastTurn Variable gibt die Sekunden seit dem letzten erfolgreichen Zug an. Diese ist notwendig um missbrauch vorzubeugen. Ein Spieler der gerade darauf wartet dass sein Gegner seinen Zug macht, kann die Funktion checkTimeForOneTurnOver aufrufen um zu überprüfen ob der Gegner bereits länger als 10 Minuten für seinen Zug benötigt. Ist dies der Fall, verliert der Spieler automatisch und das gameState wird in PLAYER1/2TIMESUP geändert. Dieser Zustand ist gleichbedeutend mit PLAYER1/2WINS, d.h es gibt einen Sieger und einen Verlierer am Ende des Spiels. 10 Minuten ist in unserem Spiel eine selbst festgelegte Zeitangabe, man könnte diese auch beim erstellen des Spiels selber bestimmen. Ein nicht ziehen des aktiven Spielers bedeutet nicht sofort eine Niederlage. Er kann auch nach 20 Minuten seinen Zug durchführen. Der wartende Spiele hat lediglich die Möglichkeit das Spiel nach 10 Minuten warten zu seinen Gunsten zu entscheiden, falls er nicht noch länger warten möchte. Er kann dies tun, muss es jedoch nicht.

3.2 Wie wird das Geld verwaltet

Zum Ende hin des letzten Unterkapitels, wurde erwähnt dass man ein Zeitlimit pro Zug benötigt um Missbrauch vorzubeugen. Wieso ist das so?
Dies hängt davon ab wie ether in einem bzw unserem Smart Contract verwaltet wird. Unsere ursprüngliche Idee war es, dass zwei Spieler zusammen ein Spiel starten und der Verlierer dem Gewinner das Ether schickt. Bereits schnell mussten wir feststellen dass das, so nicht

funktioniert. Denn nur weil man auf einem Smart Contract gegeneinander spielt bedeutet das nicht, dass der Verlierer dem Gewinner das Ether einfach sendet. Zwingen kann man den Verlierer ebenfalls nicht, bzw eine Funktion schreiben die Ether von der Verlierer Adresse zur Gewinner Adresse sendet. Dies geht nur wenn man den private Key besitzt und wenn man diesen besitzen würde, könnte man auch gleich den kompletten Betrag dieser Wallet verschicken.
Tatsächlich muss der Smart Contract selber, genau wie eine Bank, das Geld verwalten. Man ersetzt eine zentrale Partei, durch sich selbst ausführenden deterministischen Code, der genau das tut was er tun soll, und nicht mehr und nicht weniger. Eine Adresse welche ein Smart Contract ist, kann weiterhin Ether Transaktionen entgegennehmen. Diese Eigenschaft nutzen wir aus und erweitern diese.

```
function deposit()public payable{
        require(msg.value>0);
        balances[msg.sender]+=msg.value;
  }

mapping(address =>uint )public balances;
```

Indem wir Funktionen, in dem Fall unsere deposit Funktion mit dem modifier payable erweitern, erlauben wir dieser Transaktionen mit einem Ether Betrag entgegenzunehmen und gleichzeitig nach belieben zu bearbeiten. Der Smart Contract Adresse ist nun so viel Ether zugewiesen worden, bzw der Ether Betrag wurde um den Betrag in der deposit Funktion erhöht. Wir brauchen jedoch noch irgendeine Zuweisung zu den jeweiligen Adressen, ansonsten würde man nicht wissen wer wie viel gesendet hat. Dies geschieht mithilfe eines mappings. Ein mapping ist vergleichbar mit einem Array. Der große Unterschied zwischen den beiden liegt jedoch in der Tatsache das ein mapping nicht numerisch fortlaufend ist. Das bedeutet dass man einen beliebigen Schlüssel verwenden kann und diesem einen Wert zuweisen kann. In unserem Fall ist der Schlüssel die Adresse und Wert der mitgegebene Ether Betrag. Mithilfe des Befehls balances[adresse] erhalten wir den Betrag, welchen diese Adresse auf diesem Smart Contract verfügt. Diese Funktion darf public sein. Ein verändern des Wertes ist von außen nicht möglich. Man kann lediglich sehen, über wie viel Ether eine Adresse verfügt.
Nachdem man nun mithilfe der deposit Funktion Ether in den Smart Contract eingezahlt hat und dieses im balances mapping den jeweiligen Spieler zugewiesen worden ist, besitzt man die Möglichkeit ein Spiel mit einer anderen Person zu starten. Wie genau die Spiele Initialisierung funktioniert, wird im nächsten Kapitel erklärt. Man prüft bei jedem erstellen eines Spiels ob beide Parteien über genügend Ether verfügen in balances, somit ist sichergestellt das der Spieleinsatz auch definitiv bezahlt werden kann. Ist dies der Fall kann das Spiel erstellt werden und beiden Spielern wird das balance um die Hälfte des bet verringert. Dieses “Geld” befindet sich nun im Spiel selber. Genau aus diesem Grund ist es essentiell nötig einen Timer für die Zugzeit zu erstellen. Ansonsten würde sich das Ether für immer in diesem Spiel befinden. Ein Spieler der einen Zug ausführen muss, obwohl dieser bereits weiß das der Gegner mit seinem darauffolgenden gewinnen kann, hat somit nicht mehr die Möglichkeit einfach keinen Zug zu machen, bzw die Möglichkeit besteht weiterhin, nur kann sein Gegner nach 10 Minuten die Funktion checkTimeForOneTurnOver aufrufen.

Diese überprüft ob der aktuelle Spielerbereits über 10 Minuten für seinen Zug benötigt. Ist dies der Fall, verliert dieser Spieler automatisch.
Nach dem Ende eines Spieles wird die aftergame Funktion aufgerufen, diese beinhaltet das verteilen des bet. _game ist in diesem Fall das Spiel auf welchem gespielt worden ist.

```
if(_game.gameState == GameState.TIE){
        balances[_game.player1]+= _game.bet/2;
        balances[_game.player2]+= _game.bet/2;
                }else if(_game.gameState == GameState.PLAYER1WINS || _game.gameState == GameState.PLAYER2TIMESUP )
        balances[_game.player1]+= _game.bet;
        else
        balances[_game.player2]+= _game.bet;
```

Im Falle eines Unentschieden bekomme beide Spieler ihren ursprünglichen Einsatz zurück.
Im Falle eines Siegers, bekommt dieser das komplette bet auf sein balances gutgeschrieben. Prinzipiell wäre es hier auch Möglich eine kleine Gebühr zu erheben. Man könnte sagen dass immer 1% des Spieleinsatzes an die Adresse des Entwicklers geht, bzw diese Gebühr ebenfalls im balances der Adresse des Entwicklers zu speichern. Da wir jedoch nur auf der Ethereum Test Blockchain spielen und dementsprechend um kein wirkliches Ether spielen, haben wir es nicht implementiert.
Nun haben wir geklärt wie man Ether in den Smart Contract einzahlt und dieser das Ether verwaltet. Doch wie können wir diesen Betrag nun zurück auf unsere Wallet überweisen?

```
function withdraw() public{
      require(balances[msg.sender]>0);
      msg.sender.transfer(balances[msg.sender]);
      balances[msg.sender]=0;
  }
```

Dafür wurde eine withdraw Funktioniert implementiert. Diese prüft zunächst ob der Wert von balances für den Spieler, welcher eine Auszahlung tätigen möchte größer als 0 ist, denn eine Auszahlung von 0 macht keinen Sinn. Nach dieser Überprüfung wird der komplette Betrag von balances an die Adresse gesendet, welche die Funktion aufgerufen hat. Natürlich nur der Betrag welcher sich hinter seinem Schlüssel befindet. Der Wert von balances wird gleichzeitig auf 0 gesetzt, denn der komplette Betrag wurde ausbezahlt. Auch hier gäbe es wieder die Möglichkeit einen Auszahlbetrag nach belieben mitzugeben und nur diesen Betrag auszuzahlen, anstatt alles auf einmal. Anfangs gab es zudem noch das require, dass man sich aktuell nicht in einem aktiven Spiel befinden darf. Nachdem wir jedoch den Betrag um den man Spielen möchte im bet selber speichern, ist dies nicht mehr notwendig. Davor gab es diese Variable nicht und nach einer Niederlage eines Spielers wurde dessen balances um den Spielbetrag verringert und das balances des Gewinners um denselben Betrag erhöht. In diesem Fall könnte der Spieler, der weiß dass er demnächst sicher verlieren wird eine Auszahlung durchführen. Somit wäre sein balances = 0 und der rechtmäßige Gewinner könnte nicht mehr bezahlt werden durch den Verlierer.

3.3 Spielablauf

Kommen wir zum Spielablauf. In diesem Kapitel muss unterschieden werden zwischen dem Ablauf einer Spielerstellung und dem tatsächlichen Spielablauf an sich, nachdem das Spiel erstellt worden ist.
Fangen wir mit der Spielerstellung an, dem wesentlich komplizierten Teil der beiden. Zunächst haben wir für uns festgelegt dass es pro Adresse nur möglich ist in einem aktiven Spiel zu sein. Möchte man ein weiteres starten, muss man das aktuelle beenden. Dafür haben wir ein weiteres mapping erstellt.

```
mapping(address => uint)public  playerGameId;
```

In diesem mapping speichern wir die aktuelle Spiel id einer jeden Adresse um überprüfen zu können ob sich diese bereits in einem aktiven Spiel befindet. Gibt man die Adresse als Schlüssel in das playerGameId mapping erhält man zunächst eine 0 als Ergebnis. Eine 0 bedeutet, dass diese Adresse sich in keinem aktiven Spiel befindet und es dieser gestattet ist ein neues Spiel zu starten. Möchte man nun wissen in welchem Spiel man sich befindet, funktioniert dies nach dem selben Prinzip. Der Spieler gibt seine Adresse ein und erhält die Spiel id in der er sich aktuell befindet. Die gleiche id erhält der Gegner, wenn er dieses mapping aufruft.
Im folgenden haben wir zwei Möglichkeiten erstellt wie man ein Spiel nun tatsächlich erstellen kann. Zum einen mithilfe einer direkten Anfrage an einen anderen Spieler, d.h ein Spieler fragt einen anderen Spieler um wie viel sie spielen möchten. Zum anderen mithilfe einer automatischen Gegnerzuweisung, mehr dazu später.
Um einen anderen Spieler direkt Herauszufordern benötigen wir noch einige zusätzliche Datenstrukturen in dem Fall drei weiter mappings.

```
// player 2 -> player 1 -> Spielbetrag
mapping(address => mapping (address =>uint)) public gameRequest;

//player 1 -> player 2
mapping(address => address) public wantsToPlayWith;

//palyer 2 -> player 1
mapping(address => address) public hasToAcceptTheGame;
```

Das erste mapping dabei ist das wichtigste,denn in diesem sind die beiden Spieler und der Betrag um den gespielt werden soll drinnen gespeichert. Im folgenden gehen wir immer davon aus dass Spieler 1 Spieler 2 eine Spielanfrage gesendet hat um 1 Ether mithilfe folgender Funktion.

```
function initGame(address _opponent, uint _gameValue)public {

    require(balances[msg.sender]>=_gameValue);
    require(playerGameId[msg.sender]==0);
    require(wantsToPlayWith[msg.sender]==address(0));
```

```
    gameRequest[_opponent][msg.sender]=_gameValue;
    wantsToPlayWith[msg.sender]=_opponent;
    hasToAcceptTheGame[_opponent]=msg.sender;

}
```

Kommen wir zur Erklärung dieser Funktion. InitGame erwartet 2 Parameter, den Gegner und den Spielbetrag um welchen gespielt werden soll. Es müssen 3 Bedingungen gelten. Zum einen muss Spieler 1 das nötige Ether bereits auf seiner balance besitzen. Er darf in keinem aktiven Spiel sein und darf keinem anderen Spieler bereits eine Spielanfrage geschickt haben. Soweit so gut. In wantsToPlayWith steht nun drinnen dass Spieler 1 gegen Spieler 2 spielen möchte. Gleichzeitig dient dieses mapping zur Überprüfung in mehreren Funktionen, sowie in dieser, ob Spieler 1 bereits eine Spielanfrage an jemanden gesendet hat. In das hasToAcceptTheGame schreiben wir das ganze nur andersrum. Somit kann man sehen, wer Spieler 2 herausfordert. Dies ist vor allem für unseren Java Front End von relevanz um anzeigen zu können, wer überhaupt eine Spielanfrage gesendet hat. Zum anderen ist vorallem dieses mapping essenziell um an den zweiten Schlüssel in gameRequest zu gelange.
Schauen wir uns mal das gameRequest mapping an. Dieses ist ein mapping in einem mapping, das heißt man benötigt 2 Schlüssel um an den Wert, in dem Fall dem Spielbetrag, zu gelangen. Selbst wenn Spieler 2 weiß dass er nach einem Spiel gefragt worden ist, kann er den Wert aus diesem mapping nicht herauslesen, da ihm der zweite Schlüssel dazu fehlt. Glücklicherweise ist der zweite Schlüssel genau der, den man aus dem hasToAcceptTheGame erhält. Somit erhält man mit gameRequest[Spieler2][hasToAcceptTheGame[Spieler2] den Spielbetrag. Diese Information kann man auslesen und in Java anzeigen lassen, nachdem man eine Spielanfrage erhalten hat. Somit kann angezeigt werden , gegen wen und um wie viel es geht.
Sollte es sich Spieler 1 anders überlegen, hat er jederzeit die Möglichkeit seine Spielanfrage abzubrechen mit hilfe der revokeGameRequest Funktion. Diese kann einfach aufgerufen werden durch Spieler 1 und entfernt jegliche relevanten Informationen zu seiner Spielanfrage in allen drei mappings.
Gleichzeitig hat auch Spieler 2 die Möglichkeit diese Anfrage nicht anzunehmen, wenn er gegen diese bestimmte Person oder diesen Betrag nicht spielen möchte. Auch er muss einfach nur die denyGameRequest Funktion aufrufen und jegliche Informationen zu dieser Spielanfrage werden gelöscht.
Entscheidet sich Spieler 2 jedoch die Anfrage anzunehmen muss er die acceptGameRequest Funktion aufrufen.

```
function acceptGameRequest() public{

    require(hasToAcceptTheGame[msg.sender]!=0);

    address opponent = hasToAcceptTheGame[msg.sender];
    require(balances[msg.sender] >= gameRequest[msg.sender][opponent]);
    require(playerGameId[msg.sender]==0);
    require(wantsToPlayWith[msg.sender]==address(0));
```

```
        //The players balances will be added in the games bet
        balances[msg.sender]-=gameRequest[msg.sender][opponent];
        balances[opponent]-=gameRequest[msg.sender][opponent];

        //reset the mappings
        gameRequest[msg.sender][opponent]=0;
        hasToAcceptTheGame[msg.sender]=0;
        wantsToPlayWith[opponent]=0;

        createGame(msg.sender,opponent, gameRequest[msg.sender][opponent]*2);

    }
```

Auch diese Transaktion schlägt fehl wenn eines der requires nicht erfüllt wird. In diesem Fall muss Spieler 2 logischerweise eine Spielanfrage erhalten haben. Sein balances muss höher sein als der Betrag um den gespielt wird. Er darf sich in keinem aktiven Spiel aufhalten und hat selber keine aktive Anfrage verschickt.
Nachdem dies alles überprüft worden ist und alle Vorraussetzungen erfüllt worden sind, werden beiden Spielern der Spielbetrag von ihren balances abgezogen. Im Anschluss dazu, werden genauso wie in den beiden beschriebenen Funktionen zuvor jegliche Informationen über die Spielanfrage entfernt, da die Anfrage zu einem positiven Ergebnis geführt hat.
Nachdem all das geschafft worden ist, kann das Spiel nun tatsächlich mit der createGame Funktion erstellt werden. Diese erwartet als Parameter die beiden Spieler und das bet. Wer von beiden Spielern anfängt wird “zufällig” ausgelost.

```
games.push(Game( _newGameId,_bet, _p1, _p2,1,new uint[](0),new uint[](9),GameState.RUNNING,now ));
```

Hier wird ein neues Game erstellt, mit allen Variablen und in das array games an letzter Stelle hinzugefügt. Gleichzeitig setzt diese Funktion für beide Spieler die gameId auf die des gestarteten Spieles. Somit befinden sich beide Spieler in einem aktiven Spiel und können kein weiteres starte, solange dieses hier nicht beendet worden ist.

(Patrick B.)
Als Alternative hierzu gibt es die Funktion “randomOpponent”, die es erlaubt eine Spielanfrage an den Contract zu schicken, ohne einen Gegner zu spezifizieren.
Falls der anfragende Spieler noch keine aktive Anfrage hat wird überprüft, ob es in der Warteschlange (das Array “activeRequests”), bereits andere Anfragen gibt und es wird ein Spiel erstellt. Falls keine aktiven Anfragen vorliegen wird der Spieler in die Warteschlange eingefügt und wartet auf einen Gegner. Wenn noch kein Spiel zustande gekommen ist, kann der Spieler seine Anfrage über die Methode “retractRequest” auch wieder zurückziehen. In diesem Modus ist die Höhe des Wetteinsatzes in einer Variablen im Smart Contract festgeschrieben.

(Daniel N.)
Nachdem nun ein Spiel mit einer der beiden Möglichkeiten erstellt worden ist, kann der aktuell aktive Spieler die doTurn Funktion aufrufen.

```
function doTurn(uint _pos) public{
        //address is in a running game
        require(playerGameId[msg.sender]!=0);

        Game storage game =games[playerGameId[msg.sender]-1];
        require(checkAllowedTurns(game, _pos) == true);

require(game.turn%2==1 && msg.sender==game.player1 || game.turn% 2 == 0 &&
msg.sender==game.player2);

        game.gameHistory.push(_pos);
        game.lastTurnMade = now;

        if(game.player1 == msg.sender){
           game.board[_pos-1]=1;
        }else{
           game.board[_pos-1]=2;
        }

        if(game.turn >= 5)
        game.gameState = checkWinner(game);

        if(game.gameState != GameState.RUNNING)
        aftergame(game);

        game.turn ++;
    }
```

Der übergebene Parameter ist das Feld auf welchem man den Zug ausführen möchte. In der checkAllowedTurns Funktion wird überprüft ob dieses Feld gültig ist, bzw ob dieses Feld bereits belegt ist. Zudem muss erneut überprüft werden, ob dieser Spieler sich überhaupt in einem aktiven Spiel befindet, ist dies nicht der Fall wird die Transaktion abgebrochen, ist dies der Fall bezieht sich sein Zug auf das Spiel in dem er aktuell aktiv ist. Gleichzeitig muss überprüft werden ob der Spieler der die Funktion aufruft überhaupt der aktive Spieler ist. Diese Überprüfung machen wir wie bereits am Anfang dieses Kapitels beschrieben mithilfe des turns. Ist turn gerade muss der Funktionsaufrufer player 2 sein, ansonsten player 1.
Der Zug wird nun in die gameHistory und in das board geschrieben. Gleichzeitig wird die Zeit seit dem letzten Zug auf 0 , bzw auf now gesetzt.
Das Spiel wird solange ausgeführt bis einer der beiden Spieler gewonnen hat, bzw das Feld voll ist und es in dem Fall Unentschieden ausgeht.

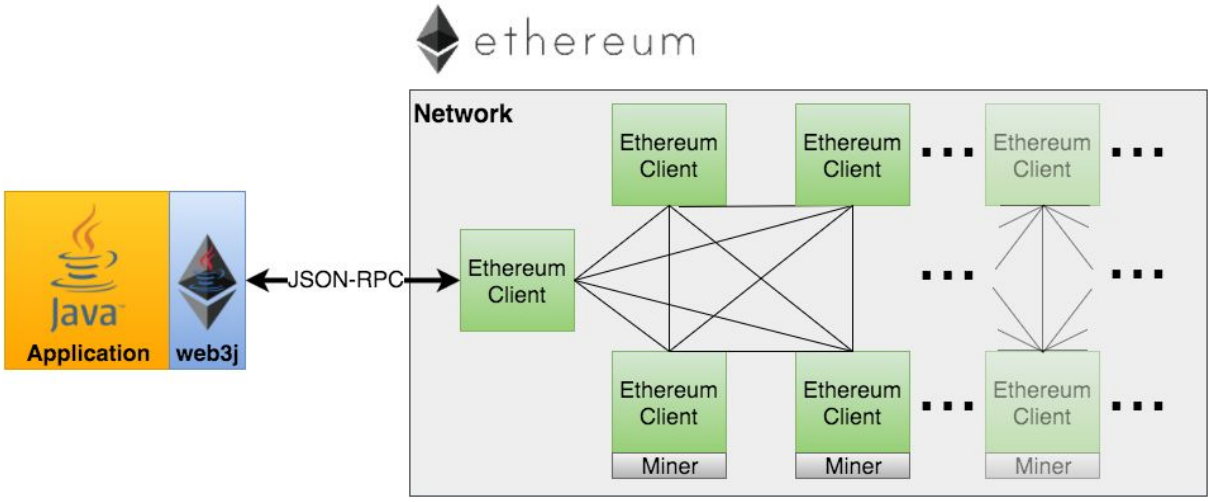
4. Schnittstelle (Patrick Budig)

4.1 Einführung

Wir haben uns dafür entschieden unser Frontend in Java zu implementieren. Das bedeutet wir benötigen eine Java-Library, um die Funktionen des Smart Contracts aus unsere Anwendung heraus aufrufen zu können. Hierfür haben wir die “web3j” - Library verwendet, da es keine wirklichen alternativen Lösungen im Java-Umfeld gibt.

4.2 Web3J

Web3J ist eine Java- und Android-Library, die unter der “Apache License 2.0” steht. Sie ermöglicht leichtgewichtige, modulare, reaktive und typsichere Arbeit mit Smart Contracts und Ethereum Clients. Sie ist die größte und weit verbreitetste Java-Library zur Anbindung an das Ethereum-Netzwerk und erlaubt das Arbeiten auf dem Netzwerk, ohne eigenen Code zur Integration schreiben zu müssen.



[1]

Neben der eigentlichen Java-Library stellt web3j Kommandozeilen-Tools zur Verfügung, um Wallet-Files und Java-Wrapper für die Smart Contracts erstellen zu können.
Wallet-Files ermöglichen einfache Handhabung von Wallets und Credentials im Programmablauf, indem relevante Informationen im json - Format gespeichert werden.
Smart Contract Wrapper bilden die Funktionen des Smart Contracts in einer Java-Klasse ab und erlauben so einfachen und typen-sicheren Zugriff auf diese in der Applikation.
Diese Funktionen können auch in Java direkt aufgerufen werden, statt in der Kommandozeile.

[1] Bildquelle: https://github.com/web3j/web3j

4.3 Anwendung

:

Ein einfacher Contract, wie:

```
pragma solidity ^0.4.0;
contract Minimal{

    uint  c;

    function add(uint a, uint b) public {
        c = a +b;
    }

    function getC() view public returns (uint) {
        return c;
    }

}
```

würde abgebildet werden auf eine Klasse mit folgende Methoden:

- public static RemoteCall<MinimalExample> deploy(...)
- public static MinimalExample load(...)
- public RemoteCall<TransactionReceipt> add(BigInteger a, BigInteger b)
- public RemoteCall<BigInteger> getC()

“Deploy” und “load” um den Smart Contract auf das Netzwerk zu deployen bzw. einen bereits deployten Contract aus dem Netzwerk zu laden und eine Instanz davon zur Verfügung zu stellen, um die Funktionen nutzen zu können.

“Add” und “getC” bilden die entsprechenden Methoden aus dem Contract ab. Das “RemoteCall”-Objekt kann mit Hilfe der Methoden “send()” oder “sendAsync()” ausgeführt werden und führt den tatsächlichen Aufruf an den Contract durch und gibt die entsprechenden Rückgabewerte zurück.

Analog zu diesem sehr einfachen Beispiel bildet die tatsächliche Schnittstelle die in Kapitel 3 beschriebenen Funktionen unseres Smart Contracts ab und stellt sie zur Verfügung.

4.4 Probleme

Zur Verbindung zwischen einer Applikation und dem Ethereum-Netzwerk wird ein Ethereum-Client benötigt. Wie in der Doku von “web3j” empfohlen, haben wir uns dafür entschieden, einen von “Infura” zur Verfügung gestellten Client in der Cloud benutzt.
Leider haben wir zu spät in der Entwicklung festgestellt, dass der “HTTP-Connector” von “Infura” es nicht erlaubt Filter zu verwenden. Das führt dazu, dass es nicht möglich ist während des Programmablaufs auf “Events” aus dem Smart Contract zu “hören”.
Abhilfe hätte hier die Benutzung eines lokalen Clients, wie “Geth” oder “Parity” schaffen können. Auch ein “Websocket” als Alternative zum “Http-Connector” steht seit kurzem von seitens “Infura” zur Verfügung, was die Benutzung von Filtern erlaubt. Dies ist aber noch nicht im Release von “web3J” implementiert.
Allerdings konnten wir das aus zeitlichen Gründen nicht mehr umsetzen.
Das hat zur Folge, dass wir in unserer Applikation sehr viel mit aktiven Nachfragen an den Contract arbeiten mussten und nicht passiv auf Benachrichtigungen warten konnten.

5. Frontend

5.1 Allgemeine Struktur (Adrian Ilius)

Das komplette User Interface ist mit Hilfe von JavaFX entwickelt worden, welches allerdings im nächsten Absatz detaillierter beschrieben werden soll. Das Frontend besteht prinzipiell aus drei verschiedenen Scenes, zwischen denen während des Programmablaufs hin und her gewechselt werden kann. Unter einer Scene versteht man einen “Container”, in dem alle Elemente, die dem Benutzer angezeigt werden sollen, wie zum Beispiel ein Button oder ein Textfeld. Die erste Scene beinhaltet den Willkommensbildschirm, von dem aus der Benutzer sich mit einer vorgespeicherten Wallet oder mit dem Privatekey seiner eigenen Wallet anmelden kann. Nach der Anmeldung wird dem Benutzer die nächste Scene, der Hauptbildschirm, angezeigt. Von hier aus bekommt er verschiedene Informationen, die alle später noch genauer beschrieben werden sollen, doch vor allem kann er von hier ein Spiel starten, was letztendlich dazu führt, dass die dritte Scene aufgerufen wird, die GameScene.
Hier findet dann das eigentliche Spiel statt. Zu jeder dieser Scenes existiert ein Controller, der dafür zuständig ist die logischen Operationen im Hintergrund auszuführen. Zusätzlich zu diesen drei Scenes und ihren Controllern existiert noch eine weitere wichtige Klasse, nämlich UserInfo. Hier werden alle wichtigen benutzerspezifischen Daten gespeichert, die das Programm braucht, um mit dem Smart Contract zu kommunizieren.

5.2 GUI mit JavaFX(Patrick Lang)

Um die GUI zu erstellen wurde JavaFX verwendet. Hierzu musste e(fx)clipse, ein Addon für Eclipse, installiert werden.
Eine GUI in JavaFX funktioniert nach folgendem Prinzip: Es gibt eine Stage in der alle Scenes ablaufen.
Wie die Namen schon vermuten lassen ist hier das Vorgehen eines Theaters abgebildet. Es gibt die Stage bzw. Bühne als Schauplatz der sich nicht verändert und innerhalb dieser Bühne können dann verschiedene Scenes bzw. Szenen nacheinander “gespielt” werden.

Da es etwas umständlich war die gesamte Oberfläche händisch zu erstellen wurde nach einer Alternative gesucht. Nach eingehender Überprüfung wurde der Scene Builder von Gluon ausgewählt. Dieser ist eine kostenlose Software von einem Drittanbieter, die installiert werden muss und es erlaubt eine Szene mit dem Drag and Drop verfahren zu erstellen.
Hierfür muss in Eclipse eine .fxml Datei erstellt werden, die man mit dem Scene Builder öffnet. Im Scene Builder erstellt man eine Szene und füllt diese mit Objekten wie Textfeldern oder Buttons. Die .fxml Datei wird automatisch vom Scene Builder befüllt.
Zur Steuerung der Szene muss in Eclipse für jede Szene eine Controller Klasse erstellt werden. In dieser wird anhand von Methoden geregelt was geschieht wenn ein Event, wie das Drücken eines Buttons, auftritt.

Als erstes wurde versucht alles innerhalb einer Szene stattfinden zu lassen, indem Panels, wie der Login oder die Spieloberfläche, unsichtbar gemacht und deaktiviert wurden. Nach

einer Besprechung wurde festgestellt das dies keine schöne Lösung ist und zu Problemen führen kann. Deshalb wurde die Herangehensweise geändert.
Dies bedeutet, dass nun jedes Panel eine eigene Szene ist. Die erste Szene, die WelcomeScene, wird in der Main-Klasse aufgerufen. Ein Szenenwechsel findet im Controller der aktuellen Szene, durch eine dort Implementierte Methode statt.
Beim Wechsel einer Szene im Controller benötigt man jedoch jedes mal ein ActionEvent um Informationen über die Stage zu bekommen. Da nicht immer ein ActionEvent zur verfügung stand und dies zu Problemen bei den Szenenwechsel führte haben wir die Herangehensweise abermals geändert und die Methoden zum Szenenwechsel in die Klasse BlockTacToeV2.java ausgelagert, welche dann in den Controllern aufgerufen werden können und keinen ActionEvent benötigen.

Startet man das Programm erscheint als erstes die WelcomeScene in welcher man sich mit den Privatkeys unserer Test Wallets oder einem eigenen Privatekey anmelden kann.
Hierfür haben wir uns für eine ChoiceBox entschieden, denn wir müssen uns zum Testen sehr oft anmelden und können so einfach mit zwei Klicks einen unserer Keys wählen.
Die Auswahlmöglichkeiten für die ChoiceBox werden in der initialize Methode im WelcomeSceneController definiert.
Wählt man die Option “mit Private Key” aus, so erscheint ein Textfeld, indem man seinen Private Key eingeben kann.
Hat man den gewünschten Key ausgewählt/eingegeben und drückt auf den Button “Anmelden” gelangt man zur HomeScene.

Die HomeScene ist quasi das Hauptmenü unserer App. Hier kann man eine Anfrage schicken oder überprüfen ob man eine Anfrage bekommen hat. Man kann Anfragen annehmen und seinen Kontostand im Smart Conract bzw. im Wallet erhöhen (einzahlen) oder verringern (abheben).
Damit z.B der Button für “Anfrage senden” weiß welche Methode er ausführen soll kann im SceneBuilder, im Menü “Code” des Buttons, unter “On Action” die im Controller definierte Methode , z.B. sendRequest, ausgewählt werden.
Der aktuelle Kontostand im Smart Contract bzw. im eigenen Wallet wird mit einem Text Objekt angezeigt, welches beim starten des Programms leer ist und dann im HomeSceneController befüllt wird. Damit dies möglich ist muss man zuerst dem Text innerhalb des Scene Builders eine einzigartige fx:id zuweisen, z.B. “EtherBalance” für den Kontostand im Smart Contract. Nun kann man mit der Annotation “@FXML” und “Text EtherBalance” auf diesen Text verweisen und ihn befüllen.
Hat ein Spieler zu wenig Ether im Contract erscheint eine AlertBox, also ein Popup, welches davor warnt. Diese kann nur bestätigt werden.
Hat man ein Spieler gefunden und hat dieser die Anfrage angenommen gelangt man in die GameScene.

Die GameScene ist sozusagen das Spielfeld unseres Tic Tac Toe Spiels und besteht aus Texten die Mitteilungen, wie z.B. aktuellen Spieler, anzeigen, einem Aktualisieren Button und dem Spielfeld. Texte und Button können wie oben bereits erläutert einer Methode zugewiesen werden.

Zur Abbildung des Tic Tac Toe Spiels haben wir uns für die Verwendung von Buttons entschieden, denn ein Tic Tac Toe Spielfeld hat Felder, welche belegt oder nicht belegt sind und ein Button kann gedrückt oder nicht gedrückt sein. Des Weiteren ist ein Button ein guter weg um die im Controller liegenden Methoden durch den Benutzer aufzurufen. Da ein Button auch einen Text haben kann, kann man sehr einfach das Zeichen des aktuellen Spielers dort einfügen und bei belegtem Feld den Button für beide Spieler deaktivieren.
Ist das Spiel vorbei erscheint wieder eine AlertBox, welche anzeigt ob man gewonnen, verloren oder unentschieden gespielt hat. Bestätigt man diese gelangt man zurück in die HomeScene.

Damit sich die verschiedenen Objekte einer Szene beim verkleinern bzw. vergrößern des Fensters richtig verhalten, wurde mit einem GridPane gearbeitet. Dadurch wird die Szene in verschiedene Bereiche eingeteilt, in die man Objekte einfügen kann. Das Grid sorgt dann dafür das die Bereiche und somit die Objekte im Verhältnis zueinander mitwachsen bzw. schrumpfen.

Um das Programm ausführen zu können wird nun noch eine Main-Klasse, nämlich BlockTacToeAppV2.java, benötigt. Diese ruft die Stage, mit der Szene, also dem .fxml, und dem dazugehörigen Controller auf.

5.3 class UserInfo (Adrian Ilius)

Wie schon erwähnt werden hier alle benutzerspezifischen Daten gespeichert, die über die verschiedenen Scenes hinweg benötigt werden, um mit dem Smart Contract zu kommunizieren. Die Klasse besteht ausschließlich aus einer Reihe von Attributen und ihren jeweiligen Gettern und Settern. Die Klasse beinhaltet folgende Attribute:

private Credentials **cred**;
private BlockTacToe **ticGame**;
private BigInteger **gameID**;
private String **startingPlayer**;
private int **playerintvalue**;
private int **turn**;
private BigInteger **walletBalance**;
private BigInteger **contractBalance**;

Im Folgenden soll nun auf jedes Attribut kurz eingegangen werden, um zu veranschaulichen wozu es verwendet wird:

private Credentials **cred**:

In diesem Attribut wird die Wallet des Spielers gespeichert. Der Datentyp Credentials wird von der web3j-Library zur Verfügung gestellt und beinhaltet alle Information bezüglich der Wallet. Der primäre Nutzen dieses Attributs liegt in der von Credentials angebotenen

Methode .getAddress(), mit der die eigene Wallet-Adresse zurückgegeben werden kann, was für viele Methoden des Smart Contracts wichtig ist.

private BlockTacToe **ticGame**:

Hier ist die eigentliche Referenz auf den Smart Contract zu finden, die über web3j hergestellt wird. Mit Hilfe dieses Attributs können alle Methoden des Contracts ausgeführt werden. (z.B. ticGame.getTurn(BigInteger gameID).send();)

private BigInteger **gameID**:

Das Attribut gameID wird für nahezu alle Methoden des Smart Contracts benötigt, denn es referenziert im Contract auf das eigene Spiel. Der Datentyp ist deswegen auch als BigInteger gewählt, denn obwohl auch ein int hinreichend für die Funktionalität wäre, braucht der Contract diesen Wert als BigInteger.

private String **startingPlayer**:

startingPlayer beinhaltet die Wallet-Adresse des Spielers, der das Spiel beginnt. Es ist insofern wichtig, da durch das Mitzählen der Züge und der Information wer anfängt immer festgestellt werden kann welcher Spieler gerade den nächsten Zug machen muss.

private int **playerintvalue**:

Hierbei handelt es sich lediglich um eine Hilfsvariable, um den Spielablauf einfacher zu implementieren. Wenn der Client Spieler 1 ist beinhaltet sie den Wert “1”, ansonsten den Wert “2”.

private int **turn**:

In diesem Attribut wird die Nummer des Zuges abgespeichert. Da es maximal 9 Züge (= 9 Felder) sind, kann turn Werte von 1 bis 9 annehmen.

private BigInteger **walletBalance**:

In walletBalance wird der aktuelle Wert der Wallet (in Ether) abgelegt, mit der sich der Benutzer angemeldet hat.

private BigInteger **contractBalance**:

Im Endeffekt gilt hier das gleiche, das auch bei walletBalance gilt, nur dass der Wert, den der Spieler in den Contract überwiesen hat gemeint ist.

5.4 Controller

Jeder Controller deklariert für sich einen GasLogger, die Klasse UserInfo und die Klasse BlockTacToeAppV2. Der GasLogger dient, wie der Name schon sagt dazu das Gas, das für die Transaktionen benötigt wird zu loggen. Die Klasse UserInfo wird benötigt, um über die verschiedenen Controller hinweg auf die Attribute der Klasse zuzugreifen und die Klasse BlockTacToeAppV2 ist lediglich die Stage für die verschiedenen Scenes und wird benötigt, um zwischen den verschiedenen Scenes zu wechseln.

5.4.1 WelcomeSceneController

Hier werden lediglich drei Methoden implementiert. Zum einen die Methode initialize(), die aufgerufen wird, sobald die Stage die WelcomeScene öffnet und zum anderen die Methode login(), die ausgeführt wird, wenn der Benutzer auf den Button “Anmelden” drückt. Die dritte Methode heißt quit() und dient nur dazu das Programm zu beenden, wenn man auf den Button “Beenden” drückt.
Die Scene besitzt eine ChoiceBox, in der man eine von mehreren vordefinierten Wallets auswählen kann oder sich mit einem Privatekey mit seiner eigenen Wallet anmelden kann. Da nur für das Anmelden mit eigener Wallet eine TextBox nötig ist, wird in der initialize()-Methode ein Listener für die ChoiceBox implementiert, der dafür zuständig ist die TextBox nur einzublenden, wenn man sich “mit Privatekey anmelden” will. Nachdem man nun eine der Optionen gewählt hat und den “Anmelden” Button drückt wird die login() Methode ausgeführt. Danach schreibt die Methode die Credentials der ausgewählten Wallet in das entsprechende Attribut in UserInfo und lädt außerdem den Smart Contract mit dieser Adresse. Wenn das alles ausgeführt wurde wechselt nun die Scene und dem Benutzer wird die HomeScene dargestellt.

5.4.2 HomeSceneController

In der HomeScene wird zusätzlich noch eine Hilfsklasse AlertBox deklariert, die dazu dient die häufige Benutzung von Popups zur Anzeige von Informationen (v.a. “Transaktion in Bearbeitung”, wenn eine Methode des Smart Contracts aufgerufen wird, da dieser Aufruf durchaus mehrere Sekunden bis in den Minutenbereich dauern kann) zu erleichtern. Außerdem wird eine Variable state = 0 deklariert, die für weitere Methoden in diesem Controller noch wichtig sein wird. In diesem Controller werden eine Vielzahl an Methoden, vor allem für die verschiedenen ButtonActions, implementiert.

Aber auch hier wird als erstes initialize() ausgeführt, die wiederum die Methode checkRequest() aufruft. Hier werden die Anzeigen für WalletBalance und ContractBalance aktualisiert. Des weiteren überprüft diese Methode mit der hasActiveRequest(eigene Adresse) Methode des Smart Contracts, ob eine Spielanfrage für den angemeldeten Benutzer/die angemeldete Wallet vorliegt. Der Rückgabewert ist ein boolean und hat den Wert true, falls eine Anfrage vorliegt. Hierfür gibt es 3 Stati: “Offene Anfrage, warte auf Gegner” (state = 1) , “Match gefunden” (state = 3) und “Keine offene Anfrage” (state =2). Wenn der Rückgabewert true ist wird state = 1 gesetzt und die entsprechende Nachricht angezeigt. Falls der Rückgabewert false ist wird anschließend die Methode playerGameId() des Contracts aufgerufen. Diese ist nämlich genau dann ungleich 0, wenn der Benutzer bereits einem Spiel zugewiesen ist. Wenn das der Fall ist setzt die Methode state = 3, zeigt die entsprechende Nachricht an und speichert die erhaltene GameID in UserInfo ab. Wenn weder hasActiveRequest() = true noch playerGameId != 0 wird state mit der entsprechenden Nachricht auf 2 gesetzt. Es gilt anzumerken, dass die Methode checkRequest() auch manuell durch den Button “Prüfen” aufgerufen werden kann.
Die Methode depositToContract() wird beim drücken des Buttons “Einzahlen” ausgeführt. Es öffnet sich ein neues Dialogfenster, in dem der Benutzer aufgefordert wird eine Betrag (in Ether) einzugeben. Wichtig hierbei ist, dass der Dezimalwert mit einem Punkt getrennt wird (und nicht wie in Deutschland üblich mit einem Komma), falls das Format allerdings falsch sein sollte wird der Benutzer darauf hingewiesen und kann den Vorgang wiederholen. Außerdem wird methodenintern der eingegebene Wert in die Einheit Wei umgerechnet, da die Methode deposit() des Smart Contracts die übergebene Zahl als solche interpretiert. Zusätzlich wird überprüft, ob der Restwert der eigenen Wallet noch groß genug ist, um für die Transaktionen zu zahlen. Falls die WalletBalance niedrig ist, wird der Benutzer darauf hingewiesen, kann aber, wenn er möchte trotzdem mit der Überweisung fortfahren. Letztendlich wird dann die Methode deposit(Betrag in wei) des Contracts in einem eigenen Thread ausgeführt. Hier gilt es anzumerken, dass wir die deposit() Methode in einem eigenen Thread implementiert haben, weil uns aufgefallen ist, dass bei Methoden des Smart Contracts, die eine Transaktion erfordern die App häufig abgestürzt ist, da die Transaktionen einige Sekunden dauern. Die Auslagerung in ein eigenes Thread hat das Problem behoben.
Die Methode updateContractBalanceDisplay(double balance) zeigt den aktualisierten ContractBalance mit 15 Nachkommastellen auf dem HomeScreen an. Selbes gilt für updateWalletBalanceDisplay für die WalletBalance.
Die Methoden getContractBalance() und getWalletBalance() lesen die aktuellen Kontostände aus Contract bzw. Wallet aus und und speichern sie in dem jeweiligen Attribut in UserInfo.
Beim Drücken des “Send” Buttons wird die Methode sendRequest() aufgerufen. Als erstes wird überprüft, ob die ContractBalance ausreichend ist, um ein Spiel zu starten. Wenn das der Fall ist wird der in checkRequest() definierte state ausgelesen. Bei 0 oder 2 wird die Methode randomOpponent() des Smart Contracts, wieder in einem eigenen Thread, aufgerufen und somit eine Spielanfrage an einen Gegner gesendet. Falls state weder 0 noch 2 ist wird der Benutzer darauf hingewiesen, dass von ihm bereits eine Anfrage gesendet wurde bzw. ein Spiel schon läuft. Zum Schluss wird von sendRequest() nochmal checkRequest() aufgerufen, um die Spielanfragen zu aktualisieren.
Beim Drücken des Buttons “Abheben” wird im Controller die Methode withdraw() aufgerufen, die wiederum die Methode withdraw() des Contracts in einem eigenen Thread aufruft. Hier

wird der aktuelle ContractBalance zurück an die Wallet überwiesen. Anschließend aktualisiert die Methode die neuen Balances mit updateContractBalanceDisplay() und updateWalletBalanceDisplay().
Mit dem Button “Logout” wird die Methode logout() aufgerufen, die die Scene wieder zur WelcomeScene wechselt.
Der Button “Zurückziehen” löst die Methode retract() auf. Diese überprüft wiederum state. Wenn state = 1, wird in einem eigenen Thread die Methode des Contracts retractRequest() ausgeführt, ansonsten wird der Spieler benachrichtigt, dass entweder das Spiel “bereits gestartet” ist oder dass “kein aktives Spiel” vorhanden ist, das er ablehnen kann.
Wenn der Button “Annehmen” gedrückt wird, führt der Controller die Methode accept() aus. Diese überprüft den state und wechselt nur im Fall state = 3 die Scene zur GameScene, an sonsten wird der Benutzer nur darüber benachrichtigt, dass noch kein Spiel gefunden wurde.

5.4.3 GameSceneController

Auch in diesem Controller wird als erstes die initialize() Methode aufgerufen. Diese initialisiert das Attribut startingPlayer der Klasse UserInfo mit dem Wert, den sie von der Contract Methode getStartingPlayer() bekommt. Das gleiche wird anschließend mit dem Attribut turn und der Contract Methode getTurn() durchgeführt. Nun wird der Text opponentText mit dem Rückgabewert der Methode getOpponent() des Contracts versehen. Anschließend wird im initialize() die Methode setGameText() ausgeführt, die in Abhängigkeit von startingPlayer und turn den Inhalt des gameText entweder auf “Du bist am Zug!” oder “Gegner ist am Zug!” setzt und es werden, falls der Client den ersten Zug machen muss die Spielfeldbuttons mit enableButtons() aktiviert. Jetzt wird überprüft, ob der Client den ersten Zug machen muss oder der Gegner und dementsprechend wird das entsprechende Symbol (X oder O) in das Feld symbolText geschrieben und anschließend mit Hilfe der disableButtons() Methode alle Spielfeldbuttons deaktiviert. Am Ender der initialize() Methode wird die Methode refresh() aufgerufen, die etwas später beschrieben wird.
Wenn der lokale Spieler an der Reihe ist, kann er nun einen der 9 Spielfeldbuttons drücken. Dabei ruft die Methode, die dem Button zugewiesen ist die Methode buttonAction(Feldnummer(1-9)) auf. Als Erstes deaktiviert diese alle Buttons mit der disableButton() Methode. Danach führt sie in einem eigenen Thread die Methode doTurn(Feldnummer) des Contracts aus. Auch hier muss der Wert erst in einen BigInteger umgewandelt werden, damit der Smart Contract diesen interpretieren kann.
Um festzustellen, ob ein Spieler gewonnen hat muss der Controller den Smart Contract nach dem GameState fragen. Dies tut er mit der getGameState() Methode, die die Methode getGameState() des Smart Contracts ausführt. Der Rückgabewert ist ein int und kann folgende Werte annehmen: 0: Spiel läuft; 1: Unentschieden; 2: Spieler 1 hat gewonnen; 3: Spieler 2 hat gewonnen; -1: Abfrage fehlgeschlagen.
In der Methode checkWinCondition() nutzt der Controller den Rückgabewert der Methode getGameState(), um eine entsprechende Popup Nachricht zu generieren. Allerdings nur, wenn GameState != 0. Danach wechselt die Scene in die HomeScene.
Die Methode setButtonText(buttonid, symbolnr) setzt das übergeben Symbol in das übergeben Feld und deaktiviert es anschließend.

Beim Drücken des Buttons “Aktualisieren” wird die Methode refresh() ausgeführt, die auch am Ende von initialize() ausgeführt wurde. Diese Methode holt sich mit der Methode des Contracts getBoard() das aktuelle Spielfeld (wenn Gegner gezogen hat ist das noch nicht im Client sichtbar) und aktualisiert alle Felder, die in dem übergebenen board nicht 0 sind, damit der letzte gegnerische Zug angezeigt werden kann. Anschließend wird das Attribut turn in UserInfo mit dem Rückgabewert der Methode getTurn() des Contracts überschrieben, damit der Client die aktuelle Zugnummer übernimmt (wichtig, da viele Zustände, z.B. disableButton() davon abhängig sind welchen Wert turn hat). Nun werden innerhalb von refresh() die Methoden checkWinCondition() und setGameText() aufgerufen und der zugText mit der aktuellen Zugnummer aktualisiert.

6. Zeit- und Kostenanalyse (Patrick Budig)

6.1 Einführung

Wie in Kapitel 2 bereits erwähnt kostet jede Funktion, die auf dem Smart Contract ausgeführt wird sogenanntes Gas, was mit Ether bezahlt werden muss.
Ausnahmen hierbei sind Funktionen die keine Änderung am Zustand des Contracts zur Folge haben, sondern nur Werte auslesen. Diese kosten kein Gas.
Im Folgenden soll dargestellt werden wie hoch diese Kosten in unserem Contract für das Spielen sind.

6.2 Zeitanalyse

Unser Smart Contract läuft auf dem “Ropsten Testnetz”. Das ist eine Version des Ethereum-Netzwerks auf dem mit kostenlosen-zu-erhaltendem Ether Code getestet werden kann.
Hier gibts es natürlich weniger “Miner” als auf dem Hauptnetz, entsprechend aber auch weniger Transaktionen. Die Zeit, die eine Transaktion in Anspruch nimmt, schwankte in unseren Test stark, je nach Auslastung des Netzes; von 20 Sekunden bis hin zu mehreren Minuten.
Im realen Ethereum-Netzwerk sollte mit durchschnittlichem Gas-Preis[2] eine Transaktion in etwa 30 Sekunden bestätigt werden können.
Jeder Zug in unserer Anwendung benötigt eine Transaktion und damit die gleiche Zeit wie eine Transaktion.

6.3 Kostenanalyse

In folgender Tabelle sind die durchschnittlichen Kosten für die einzelnen Funktionen des Smart Contracts aufgeführt.
Hierzu zu bemerken ist, dass jeder Spieler die Kosten für seine Transaktionen selbst trägt.
Es wird ein Gas-Preis von 4 Gwei (4 x 10^-9 Ether) und ein Ether-Preis von 406,09 € angenommen[3].

[2] 4 Gwei pro Gas (Stand 24.07.2018, Quelle www.ethgasstation.info)
[3] Stand: 24.07.2018

Funktion	Kosten in Gas	Kosten in Gwei	Kosten in Euro
Geld einzahlen	42000	168000	0,0682 €
Geld abheben	23000	92000	0,0374 €
Game Request an Spieler senden	85000	340000	0,1380 €
Game Request annehmen	300000	120000000	0,4873 €
Zufälligen Game Request senden	168000	672000	0,2729 €
Zufälligen Game Request zurückziehen	23000	92000	0,0374 €
Zug senden	100000	400000	0,1624 €
3 Züge senden	300000	12000000	0,4873 €
5 Züge senden	500000	20000000	0,8121 €
9 Züge senden	900000	36000000	1,4619 €

6.4 Fazit

In einem eher kurzweiligen, schnellen Spiel, wie “Tic Tac Toe” ist die Wartezeit von mindestens 30 Sekunden zwischen Zügen relativ hoch.
Auch die Kosten von fast einem Euro pro Spiel pro Spieler machen diese Anwendung aus unserer Sicht eher unattraktiv.

Wie bei allen Anwendungen im Bereich “Cryptocurrency” muss natürlich auch in diesem Fall der stark schwankende Kurs dieser “Währungen” berücksichtigt werden, wenn man die Nutzbarkeit feststellen möchte.

Weiterführend könnte dieser Stelle eine genauere Analyse der Aufteilung der Kosten auf die einzelnen Operationen innerhalb eines Zuges durchgeführt werden, um Optimierungen durchführen zu können., um gegebenenfalls die Kosten senken zu können.

Dies würde mutmaßlich aber auf Kosten von Funktionalität gehen, wie beispielsweise den Wegfall des Spielverlaufs oder der Möglichkeit einen Timeout zu haben und auch dann wahrscheinlich keine wesentliche Kostenreduzierung ermöglichen.

Wie in einem vorherigen Kapitel bereits erwähnt, sind vor allem Änderungen am Zustand eines Contracts, das ist das Anlegen / Schreiben einer Variablen, gas-intensive Operationen. Das bedeutet, dass Spiele mit komplexer Logik, die aber nicht nennenswert mehr Daten benötigen , um den Spiel-Zustand darzustellen (beispielsweise Mühle), für diese Anwendung eher geeignet sein könnten.

7. Lessons learned (Daniel)

Was haben wir vier mithilfe diesem Projekt gelernt? Zum einen besteht ein Projekt tatsächlich aus über der Hälfte der Zeit mit organisieren, besprechen, planen und co. Was wir definitiv gelernt haben ist, das Zeitmanagement schon etwas wichtiges ist und man nicht immer alles auf den letzten Drücker machen kann/sollte. Auch war unsere Kommunikation untereinander stellenweise zu gering und es wurde sich wochenlang nicht ausgetauscht. Im Endeffekt sind wir jedoch alle froh dass es am Ende doch noch alles geklappt hat und wir unser Projekt erfolgreich abgeben können.