HOCHSCHULE TRIER
Trier University of Applied Sciences
Informatik - Computer Science

Entwurf eines Frameworks für digitale Sammelkartenspiele und Referenzimplementierung eines Clients

Designing a framework for digital trading card games and implementing a reference client application

Patrick Kranz

Projektstudium

Betreuer: Prof. Dr. Rainer Oechsle

Trier, 2.5.2016

Kurzfassung

Es gibt eine große Anzahl von Onlinediensten, die Sammelkartenspiele anbieten. In der Regel ist all diesen gemein, dass sie einen zentralen Dienst bereitstellen. Bei diesen Diensten ist hinterlegt, welchem Nutzer welche Karten gehören. Das führt insbesondere dazu, dass während einer temporären Störung des Dienstes oder nach seiner Einstellung durch den Betreiber die Spieler keine Karten mehr tauschen oder Partien spielen können. Zudem wird eine Internetverbindung benötigt, um zu überprüfen, ob ein potentieller Tauschpartner tatsächlich im Besitz der angegebenen Karten ist.

Auch das Spielen mit den erworbenen Karten selbst wird in der Regel über einen zentralen Dienst abgewickelt. Dabei werden die Regeln, wer mit wem spielt von dem jeweiligen Betreiber festgelegt.

Ziel dieser Arbeit ist es, ein System zu entwickeln, das erlaubt ohne Verbindung zu einem zentralen Server, Karten mit anderen Nutzern sicher zu tauschen und Spiele zu spielen. Dies soll den Nutzer unabhängiger von einem Betreiber machen und somit seine Rechte stärken.

In dem von uns vorgeschlagenem System erhält der Nutzer die Kontrolle darüber wann er mit wem welche Karten tauscht. Auch nachdem ein Betreiber, der dieses System nutzt, seinen Dienst einstellt, können die Nutzer die von diesem Betreiber erhaltenen Karten weiter einsetzen und handeln.

Zudem erlaubt unser System das gegenseitige Tauschen von Karten, welche von verschiedenen Betreibern erworben wurden. Somit sind die Nutzer nicht mehr fest auf einen Dienst angewiesen, sondern können Karten einer Vielzahl von Betreibern nutzen.

Das Spiel selbst wird direkt zwischen den beteiligten Spielern ausgetragen, ohne einen Mittelsmann als vertrauenswürdige dritte Instanz zu benötigen.

Das von uns implementierte System ist als Server-Client-System entwickelt worden, bei dem die Clients auch ohne Verbindung zu einem Server weiterhin nutzbar bleiben sollen. Um dies zu erreichen, nutzen wir asynchrone kryptographische Verfahren, sowohl beim Handeln als auch beim Spielen.

Inhaltsverzeichnis

1 Einleitung und Problemstellung

Sammelkartenspiele sind Kartenspiele, bei denen jeder Spieler seine eigenen Karten besitzt und mit einer Teilmenge aller existierenden Karten spielt. Zum Vergleich spielt man bei Mau-Mau immer mit den gleichen Karten. Im Normalfall werden bei Sammelkartenspielen nicht bestimmte Karten erworben, sondern zufällig zusammengestellte Kartenpackungen (sogenannte Booster). Abhängig davon welchen Spielstiel ein Spieler verfolgt, benötigt er unterschiedliche Karten. Des Weiteren haben bestimmte Kombinationen von Karten zusätzliche Synergieeffekte. Da Karten im Normalfall nicht gezielt erworben werden können, handeln die Spieler diese untereinander. Bei einem solchen Handel können beide Spieler profitieren, da Beispielsweise die Karten den Spielstiel des Tauschpartners besser unterstützen als den eigenen.

Bei einem Sammelkartenspiel kann man verschiedene Tätigkeiten unterscheiden:

- Das Erwerben der Karten,
- das Tauschen von Karten mit anderen Spielern,
- das Zusammenstellen von Karten zu Decks und
- das Spielen mit anderen Spielern.

Sowie in oben genannten Punkten der Besitz einer Karte von Belang ist, so nimmt er auch in dieser Arbeit einen hohen Stellenwert ein. Dabei ist es egal, ob mit einer Karte gespielt werden soll oder diese gegen eine andere gehandelt wird. All dies ist nur möglich, insofern man der Besitzer einer Karte ist. Eine solche Karte, die der Spieler nicht besitzt, kann er weder tauschen noch zum Spielen nutzen. Somit ist ein essentieller Bestandteil eines Sammelkartenspieles der Wert einer Karte, die durch ihre begrenzte Verfügbarkeit entsteht.

In der realen Welt wird dies über den physischen Besitz einer Karte sichergestellt. Eine Vervielfältigung ist mit haushaltsüblichen Mitteln nicht ohne weiteres möglich, da bei einer solchen Qualitätseinbußen zu tragen kommen.

In der virtuellen Welt hingegen sind verlustfreie Kopien essentieller Bestandteil, ohne dessen die meisten Systeme nicht funktionieren würden. Um Daten zu betrachten, benötigt man eine Kopie dieser Daten. Aus diesem Grund reicht der Besitz der Daten, die eine Karte darstellen, alleine nicht aus.

Viele Systeme setzen auf einen zentralen Server, der als vertrauenswürdige dritte Instanz jederzeit gefragt werden kann, welcher Spieler welche Karte besitzt. Jeder Erwerb und Tausch von Karten wird über diesen abgewickelt. Ein solcher Dienst erzeugt die Karten und führt Buch über die Besitzverhältnisse dieser. Nach einem Abschalten dieses Dienstes kann nicht mehr nachvollzogen werden, welchem Nutzer welche Karten gehören. Zudem kann der Dienst Nutzern nachträglich Karten wieder entziehen, ohne dass der Nutzer dies beeinflussen kann.

Eine weitere Variante stellen Systeme dar, die den Wert einzelner Karten ignorieren und jedem Spieler beliebige Karten zur Verfügung stellen. Da jeder Spieler mit jeder Karte spielen darf, gibt es keinen Grund weitere Sicherungsmaßnahmen zu treffen.

Ziel dieser Arbeit ist es, ein System zu konzipieren und prototypisch zu implementieren, welches nur bedingt von einem zentralen Service abhängig ist. Dabei versuchen wir folgende Ziele zu erreichen:

- Z_1: In dem System existieren Teilnehmer, welche neue Karten erschaffen und verteilen können.
- Z_2: Ein Teilnehmer des Systems kann bekannte Besitzverhältnisse zwischen Karten und Teilnehmern verifizieren.
- Z_3: Eine Karte kann von jedem Teilnehmer des Systems auf unerlaubte Veränderung überprüft werden, um ihre Integrität festzustellen.
- Z_4: Zwei Teilnehmer können Karten beliebig handeln.
- Z_5: Zwei Teilnehmer können miteinander spielen, ohne die Möglichkeit zu besitzen zu betrügen.
- Z_6: Es wird keine Kommunikation zu unbeteiligten Teilnehmern des Systems benötigt.

Fokus dieser Arbeit liegt hierbei auf der Implementierung des Clients. Die Implementierung des Servers wird in der Arbeit „Entwurf eines Frameworks für digitale Sammelkartenspiele und Referenzimplementierung eines Servers“ von Tobias Krumholz beschrieben.

2 Notationen und mathematische Grundlagen

An dieser Stelle werden in der Arbeit verwendete Notationen und mathematische Grundlagen erläutert.

2.1 Notationen

Einige Textabschnitte werden besonders hervorgehoben, die folgende Bedeutung haben:

> **Beispiel**
>
> Beispiele sollen Sachverhalte anhand eines Beispiels verdeutlichen.

Beispiel x

> **Erläuterung**
>
> Erläuterungen gehen auf einen bestimmten Sachverhalt genauer ein.

Erläuterung x

> **Pseudocode**
>
> Pseudocode beschreibt einen Algorithmus.

Pseudocode x

Unterhalb der jeweiligen Box befindet sich die Nummerierung. Diese wird im Text verwendet, um auf den entsprechenden Kasten zu verweisen.

Zitate und Quellen sind in runden Klammern angegeben. Entsprechende Einträge finden sich im Literaturverzeichnis.

2.2 Mathematische Grundlagen, Begriffe und Definitionen

An einigen Stellen der Arbeit werden verschiedene Mengen und Operationen genutzt, welche im Folgenden definiert werden.

2.2.1 Grundlegende Mengen

Die Menge der Primzahlen ist definiert mit $\mathbb{P}$. Zudem definieren wir noch die Mengen $\mathbb{Z}_n$ und $\mathbb{Z}_n^*$ für ein $n \in \mathbb{N}$.

$$\mathbb{Z}_n \stackrel{\text{def}}{=} \{0,1, \dots, n-1\}$$

$$\mathbb{Z}_n^* \stackrel{\text{def}}{=} \{a \in \mathbb{Z}_n | ggt(a,n) = 1\}$$

Wobei $ggt(a,n)$ den größten gemeinsamen Teiler zwischen a und n berechnet.

2.2.2 Kongruenz

Die Aussage a ist zu n kongruent Modulo m wird wie folgt notiert:

$$a \equiv_m n$$

Dies bedeutet, dass die Divisionen $\frac{a}{m}$ und $\frac{n}{m}$ denselben Rest besitzen:

$$a \equiv_m n \Leftrightarrow a \bmod m = n \bmod m$$

2.2.3 Quadratischer Rest

Die Mengen $\mathbb{QR}_m$ der Quadratischen Reste und $\mathbb{NQR}_m$ der Quadratischen Nichtreste sind wie folgt definiert:

$$\mathbb{QR}_m \stackrel{\text{def}}{=} \{i \in \mathbb{Z}_m^* \mid \exists a \in \mathbb{Z}_m \; a^2 \equiv_m i \}$$

$$\mathbb{NQR}_m \stackrel{\text{def}}{=} \{i \in \mathbb{Z}_m^* \mid \nexists a \in \mathbb{Z}_m \; a^2 \equiv_m i \}$$

Die Mengen $\mathbb{QR}_m$ und $\mathbb{NQR}_M$ sind disjunkt und ihre Vereinigung bildet die Menge $\mathbb{Z}_m^*$. Ob eine Zahl bezüglich eines Modulo m ein Quadratischer Rest oder ein Quadratischer Nichtrest ist, ist allgemein nicht effizient entscheidbar.

Ob ein $a \in \mathbb{Z}_m^*$ teil von $\mathbb{QR}_m$ oder $\mathbb{NQR}_m$ ist, lässt sich hingegen für alle $p \in \mathbb{P}$ mittels folgender Formeln feststellen:

$$a^{\frac{p-1}{2}} = 1 \Leftrightarrow a \in \mathbb{QR}_p$$

$$a^{\frac{p-1}{2}} = -1 \bmod p \Leftrightarrow a \in \mathbb{NQR}_p$$

(Bundschuh 1992)

Falls $m = p_1 * \dots * p_i$ mit $i \geq 2$ und $p_1, \dots, p_i \in \mathbb{P}$ dann gilt:

$$a \in \mathbb{QR}_m \Leftrightarrow \forall p \in \{p_1, \dots, p_i\} \; a \in \mathbb{QR}_p$$

$$a \in \mathbb{NQR}_m \Leftrightarrow \exists p \in \{p_1, \dots, p_i\} \; a \in \mathbb{NQR}_p$$

Das Lengendre-Symbol ist definiert für $p \in \mathbb{P}$ und $a \in \mathbb{Z}_p^*$

$$\left(\frac{a}{p}\right) \stackrel{\text{def}}{=} \begin{cases} +1, & a \in \mathbb{QR}_p \\ -1, & a \in \mathbb{NQR}_p \end{cases}$$

Das Jacobi-Legendre-Symbol (kurz Jacobi-Symbol) ist definiert für $n \in \mathbb{N}$ und $a \in \mathbb{Z}_n^*$:

$$\left(\frac{a}{n}\right) \stackrel{\text{def}}{=} \begin{cases} \left(\frac{a}{n}\right), & n \in \mathbb{P} \\ \left(\frac{a}{p}\right) * \left(\frac{a}{m}\right), & p \in \mathbb{P}\ n = p * m \end{cases}$$

Das Jacobi-Symbol ist Multiplikativ sowohl im Nenner als auch im Zähler.

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right) * \left(\frac{b}{n}\right)$$

$$\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right) * \left(\frac{a}{n}\right)$$

Die Menge $\mathbb{J}_m^1$ ist definiert als

$$\mathbb{J}_m^1 \stackrel{\text{def}}{=} \left\{a \in \mathbb{Z}_m | \left(\frac{a}{m}\right) = 1\right\}$$

Falls $m \in \mathbb{N}$, $a, b \in \mathbb{J}_m^1$ und $c = a * b$, dann ist auch $c \in \mathbb{J}_m^1$.

Die Menge $\mathbb{PQR}_m$ der Pseudoquadrate, die alle Zahlen, für die das Jacobi-Legendre-Symbol $+1$ zurückliefert, sich jedoch nicht in $\mathbb{QR}_m$ befinden, enthält, ist definiert als

$$\mathbb{PQR}_m = \{a \in \mathbb{Z}_m \mid a \in \mathbb{J}_m^1 \wedge a \in \mathbb{NQR}_m\}$$

Die Vereinigung der Mengen $\mathbb{QR}_m$ und $\mathbb{PQR}_m$

$$\mathbb{QR}_m \cup \mathbb{PQR}_m = \mathbb{J}_m^1$$

Falls $m \in \mathbb{N}$, $a \in \mathbb{Z}_m^*$, dann ist $a^2 \in \mathbb{QR}_m$. Sei $b = a^2$, dann ist $b \in \mathbb{QR} \Leftrightarrow \exists c \in \mathbb{Z}_m^*$ für das gilt $c^2 \equiv_m b$. Sei $c = a$.

Allgemeiner kann man für $m \in \mathbb{N}$, $x, y \in \mathbb{J}_m^1$ und $z \in \mathbb{QR}_m$ für die gilt $y = x * z$ schreiben,

$$y \in \mathbb{PQR}_m \Leftrightarrow x \in \mathbb{PQR}_m$$

(Goldwasser und Micali 1982, 370)

3 Konzepte

Bei der Entwicklung des Systems wird grundlegend zwischen zwei Teilen unterschieden:

Ein Bereich beschäftigt sich mit der Kommunikation zwischen zwei Clients, um das Spielen zwischen diesen zu ermöglichen. Dabei soll ein faires Spiel gewährleistet werden, welches auch bei Manipulation der Software keinem der Spieler Vorteile geben darf.

Der andere Bereich befasst sich mit dem Erwerb, Besitz und Handel von Karten. Dies betrifft sowohl Client zu Client als auch Client zu Server-Verbindungen. Das von uns entwickelte Protokoll unterscheidet prinzipiell nicht zwischen Server und Client. Alle Teilnehmer des Netzwerkes sind gleichberechtigt. Dennoch unterscheiden sich Möglichkeiten, welche die Implementierungen besitzen. Beispielsweise besitzt der Client keine Logik zum Erzeugen von Karten, obwohl ihm das Protokoll dies erlauben würde. Im folgenden Abschnitt wird dieser Umstand genauer erläutert.

3.1 Das Handelssystem

Eines der zu erreichenden Ziele ist, dass zwei Personen bzw. zwei Clients Karten, die diese besitzen, miteinander tauschen können, ohne dass sie Unterstützung von anderen Personen erhalten.

Es gibt bereits viele Systeme, die ein Handeln von digitalen Gütern erlauben. Im Kontext von Sammelkartenspielen sind das zum Beispiel „Hearthstone: Heroes of Warcraft“ oder „Pokémon TCG Online“. Diese nutzen in der Regel zentrale Server, über die gehandelt wird. So können alle Clients zweifelsfrei und auf einfachem Wege erfahren, wer Besitzer einer Karte ist.

Dies hat jedoch den Nachteil, dass ein Versagen des Zentralservers dazu führt, dass niemand mehr weiß, wer welche Karte besitzt. Das bedeutet, dass der Wert und der Kartenbesitz der Spieler, zumindest für die Zeit des Ausfalls, nichtig sind. Sollte der Zustand nicht behoben werden können, kommt dies einem vollständigen Verlust gleich.

Eine weitere, weit verbreitete Variante sind digitale dezentrale Währungssysteme bzw. Kryptowährungen. Allen voran ist Bitcoin (Nakamoto 2009) zu erwähnen, aber es existieren auch weitere wie Nxt oder Litecoin. Diese speichern alle Transaktionen in einem dezentralen Netzwerk. Im Allgemeinen wird dies Blockchain genannt. Der Name wird von den sogenannten Blöcken hergeleitet, welche die einzelnen Transaktionen bündeln, die untereinander verknüpft eine Kette bilden.

Wird eine Transaktion durchgeführt, sendet der Client eine Nachricht über das Netzwerk. Andere Clients, in der Regel Miner genannt, bündeln daraufhin mehrere Transaktionen zu einem Block zusammen. Zusätzlich zu den Transaktionen enthält der Block noch einen

Verweis auf den letzten Block der Blockchain und einen frei wählbaren Wert. Dieser Wert muss so gewählt sein, dass bei der Anwendung einer Hashfunktion auf den gesamten Block das Ergebnis bestimmte Eigenschaften erfüllt (z.B. die ersten 5 Bits des Ergebnisses müssen 0 sein). Ist diese Berechnung gelungen, kann der Miner sein Ergebnis veröffentlichen und der Block wird von allen Teilnehmern an ihre lokale Kopie der Blockchain gehängt. Damit sind alle Transaktionen dieses Blockes abgeschlossen. Die Sicherheit der Blockchain basiert unter anderem auf dem benötigten Aufwand, einen Block zu erzeugen.

Je nach Kryptowährung kann sich dieses Vorgehen unterscheiden. Häufig sind solche Systeme dezentral organisiert und die Blockchain oder vergleichbare Konstrukte sind die virtuelle zentrale Anlaufstelle um festzustellen, welcher Teilnehmer welchen Wert besitzt.

Kryptowährung

„Durch kryptographisch abgesicherte Protokolle und dezentrale Datenhaltung ermöglichen Kryptowährungen bargeldlosen digitalen Zahlungsverkehr ohne Zentralinstanzen wie etwa Banken. An die Stelle eines bedruckten Stück Papiers (Geldschein) oder eines geprägten Stück Metalls (Münze) zur Repräsentation des Tauschwerts tritt der Besitz eines kryptologischen Schlüssels zu einem ebenfalls kryptologisch signierten Guthaben in einer gemeinschaftlichen Buchführung. In der Regel wird dabei eine vorher festgelegte Anzahl an Währungseinheiten durch das gesamte System gemeinschaftlich erzeugt, wobei die Rate vorher festgelegt und veröffentlicht bzw. durch den kryptographischen Modus der Erzeugung limitiert ist.“

(Kryptowährung - Wikipedia 2016)

Erläuterung 1

Sowohl das serverbasierte Konzept, als auch die dezentrale Variante welche Kryptowährungen in der Regel umsetzten haben den Nachteil, dass der Nutzer mit einer zentralen Stelle kommunizieren muss, um einen Handel abzuschließen. Dies scheint auf den ersten Blick bei den erwähnten Kryptowährungen widersprüchlich zu sein, da diese oft auf einem P2P-Netz aufbauen. Im Idealfall sollen derartige Systeme keine zentrale Instanz besitzen, die anders behandelt wird als alle anderen Teilnehmer. Jedoch bildet die Gesamtheit aller Teilnehmer in einem P2P-Netz diese virtuelle zentrale Instanz. Um eine Transaktion durchführen zu können, muss ein Teilnehmer mit dem Netz in Verbindung stehen.

Diese Arbeit hat sich zum Ziel gesetzt, ein System zu entwickeln, das genau diese Kommunikation zu einem zentralen Server, sei er nun tatsächlich vorhanden oder nur virtuell als Schwarm von Rechnern, zum Handeln nicht benötigt. Eine Motivation dieser Zielsetzung ist, dass es immer noch viele Orte gibt, die keinen oder nur unzureichenden Zugang zum Internet besitzen.

3.1.1 Handelsnetzwerk

Um das gesetzte Ziel zu erreichen, setzen wir auf ein asymmetrisch kryptographisches Verfahren. Jeder Spieler besitzt einen privaten und öffentlichen Schlüssel. Dabei stellt der öffentliche Schlüssel die Identität einer Person in unserem Handelsnetzwerk dar.

Eine Karte besteht aus einer eindeutigen ID, der ID des Erstellers (seinem öffentlichen Schlüssel) und den eigentlichen Daten, welche die Spieleigenschaften ausmachen. Dazu zählen unter anderem ein Name oder das Bild der Karte. Um unberechtigte Änderungen an Karten zu verhindern, gehören zu diesen auch die jeweilige Unterschrift des Erstellers.

Sollen Karten gehandelt werden, werden hierzu Transaktionen abgeschlossen. Dabei stellt eine Transaktion einen Datensatz dar, der die gehandelten Karten (bzw. deren ID) und die handelnden Spieler enthält.

Für jede gehandelte Karte werden neben der Karten-ID und dem Empfangenden bzw. gebenden Spieler noch weitere Werte der Transaktion hinzugefügt. Dabei handelt es sich um den sogenannten Transaktionsindex. Dieser gibt an, wie oft die Karte bereits gehandelt wurde, und einen Hashwert der vorherigen Transaktion. „Vorherige Transaktion“ meint die Transaktion, mit welcher die Karte dem zurzeit noch aktuellen Eigentümer übertragen wurde. Die so neu erstellte Transaktion wird dann von beiden Spielern mittels ihres privaten Schlüssels unterschrieben, womit der Handel abgeschlossen ist.

Bei einem Kartenhandel muss es nicht um einen Tausch oder Handel im eigentlichen Sinne gehen. Auch eine Schenkung, bei der nur ein Spieler Karten erhält, ist möglich.

Jeder Teilnehmer des Netzwerkes kann in der Theorie auch Karten erzeugen. In der Realität wird dies jedoch nicht von den Clients umgesetzt, sondern lediglich von den Servern. Das Erzeugen einer Karte geschieht ebenfalls mittels Transaktion. Hierfür wird eine Transaktion erstellt, bei der die erzeugten Karten einem Spieler übertragen werden. Sowohl der Ersteller der Karte als auch der empfangende Spieler müssen diese Transaktion unterschreiben. Dabei wird der Transaktionsindex auf 0 gesetzt und der vorherige Transaktionshash wird nicht gesetzt. Wichtig ist hierbei, dass die Ersteller-ID der Karte mit dem Geber in der ersten Transaktion übereinstimmt. Dies stellt sicher das nur der Ersteller einer Karte diese erzeugen kann. Somit ist er alleine dafür verantwortlich ob seine erzeugten Karten einen Wert besitzen. Stellt er inflationär neue Karten her, verlieren diese für die Spieler ihren Wert.

Will nun ein Teilnehmer überprüfen, ob einem Spieler eine Karte gehört, lässt er sich von diesem alle nötigen Daten übertragen, die für den Nachweis benötigt werden. Beginnen kann er mit der Transaktion, welche die Karte dem Spieler übertragen hat. Da diese Transaktion von dem vorherigen Besitzer der Karte mit seinem privaten Schlüssel unterschrieben wurde, ist sichergestellt, dass dieser der Transaktion zugestimmt hatte. Der zugehörige zur Überprüfung benötigte öffentliche Schlüssel ist ebenfalls in der Transaktion enthalten. Im nächsten Schritt müssen die Transaktionen, auf der die letzte aufgebaut hat, überprüft werden. Somit wird sichergestellt, dass der vorherige Besitzer die Karte besaß und auch weitergeben durfte. Der Vorgang wird solange wiederholt, bis man zu den erzeugenden Transaktionen gelangt ist. Sind auf dem gesamten Weg keine Unstimmigkeiten aufgefallen, können wir davon ausgehen, dass der Spieler der tatsächliche Besitzer der Karte geween ist.

Da das System keine zentrale Instanz besitzt, bei der die Transaktionen angefragt werden können, muss jeder Spieler für alle Karten, die er besitzt, alle nötigen Transaktionen vorhalten. Dies schließt auch die Transaktionen der Karten ein, die er weggegeben hat, sowie die Transaktionen seiner Tauschpartner. Im Normalfall sind in einer Transaktion mehrere Karten enthalten, so dass ein Graph von Transaktionen entsteht.

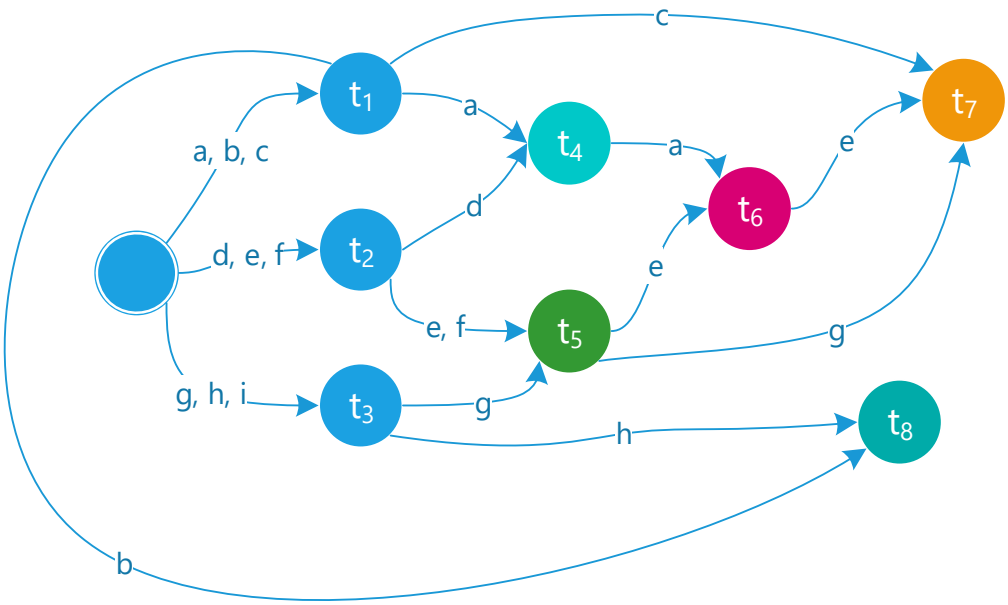


Abbildung 1

Je nachdem was man ausdrücken will, kann man die Zusammenhänge zwischen Transaktionen in verschiedenen Graphen darstellen. Abbildung 1 zeigt einen Transaktionsgraphen, der den Zusammenhang zwischen einzelnen Transaktionen darstellt. Die einzelnen Knoten symbolisieren die jeweilige Transaktion; der nicht beschriftete Startkonten nimmt eine Sonderrolle als nichtexistente Ursprungstransaktion ein. Jede neu erzeugte Karte besitzt diesen als Vorgängertransaktion. Die Kanten stellen die Handelshistorie der Karten dar. Eine Kante von t_i nach t_j mit der Beschriftung a bedeutet, dass die Karte a in der Transaktion t_j vorkommt und dass die Vorgängertransaktion dieser Karte t_i war. Dies ist nur gültig, wenn in t_i die Karte a verarbeitet wurde. Der Übersichtlichkeit halber wurden in Abbildung 1 mehrere Kanten, welche dieselben Ursprungs- und Zielknoten haben, zu einer Kante zusammengefasst und die einzelnen Karten durch Komma voneinander getrennt. Die Transaktionen t_4 bis t_8 wurden unterschiedlich farblich hervorgehoben, damit diese in Abbildung 2 wiedererkannt werden und der Zusammenhang beider Abbildungen deutlich wird. Des einfacheren Verständnisses wegen besitzt dieser Graph weder den Transaktionsindex noch von bzw. zu welchem Nutzer die Karte übertragen wird.

Abbildung 2 zeigt einen Nutzergraphen, der es erlaubt, den Weg nachzuvollziehen, den eine Karte über verschiedene Eigentümer genommen hat. Die Knoten stellen die einzelnen Nutzer dar und die Kanten symbolisieren die Weitergabe der Karten von einem Nutzer zum nächsten. Die Beschriftung enthält neben den Karten auch den dazugehörigen Transaktionsindex. Nicht explizit dargestellt werden die Transaktionen, die Kanten besitzen jedoch dieselbe farbliche Markierung wie die dazugehörigen Transaktionen in Abbildung 1. Somit lassen sich zusammengehörende Kanten einer Transaktion von anderen unterscheiden.

Für eine textuelle Repräsentation kann man eine Transaktion als ein Tupel bestehend aus einer Menge von Unterschriften und Kartenübertragungen darstellen. Eine Kartenübertragung kapselt dabei die benötigten Informationen pro Karte, welche gehandelt werden soll. Seien A und B die öffentlichen Schlüssel der Handelspartner, n die Gesamtzahl der zu handelnden Karten, k_j eine Karte, $index_j$ der Transaktionsindex der Karte k_j, $sender_j$,

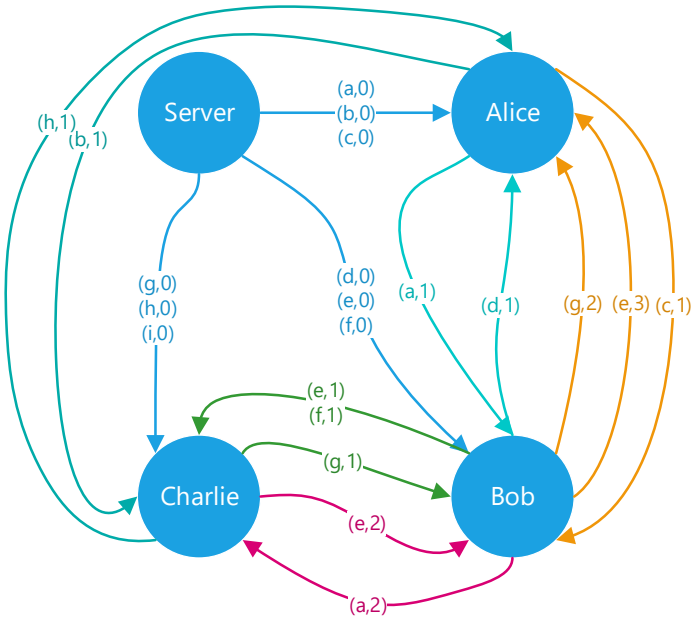


Abbildung 2

$reciever_j \in \{A, B\}$ der Besitzer bzw. der Empfänger der Karte k_j, wobei $sender_j \neq reciver_j$, t_j die Vorgängertransaktion von k_j, mit welcher $sender_j$ Eigentümer selbiger wurde, wobei $j \in \{1, \dots n\}$ und $hash(x)$ eine Funktion, die von einer Transaktion x den Hashwert berechnet, dann ist die Transaktion t wie folgt definiert

$$t = (\{(k_1, index_1, hash(t_1), sender_1, reciver_1), \dots, (k_n, index_n, hash(t_n), sender_n, reciver_n)\}, \{Sig_A, Sig_B\})$$

Dabei sind Sig_A und Sig_B die Signatur zum ersten Teil des Tupels t mittels des privaten Schlüssels zugehörig zum öffentlichen Schlüssel A respektive B.

Beispiel

Zu den Abbildungen Abbildung 1 und Abbildung 2 würde eine textuelle Repräsentation wie folgt aussehen:

$t_1 = (\{(a, 0, \text{hash}(\emptyset), S, A), (b, 0, \text{hash}(\emptyset), S, A), (c, 0, \text{hash}(\emptyset), S, A)\}, \{Sig_S, Sig_A\})$
$t_2 = (\{(d, 0, \text{hash}(\emptyset), S, B), (e, 0, \text{hash}(\emptyset), S, B), (f, 0, \text{hash}(\emptyset), S, B)\}, \{Sig_S, Sig_B\})$
$t_3 = (\{(g, 0, \text{hash}(\emptyset), S, C), (h, 0, \text{hash}(\emptyset), S, C), (i, 0, \text{hash}(\emptyset), S, C)\}, \{Sig_S, Sig_C\})$
$t_4 = (\{(a, 1, \text{hash}(t_1), A, B), (d, 1, \text{hash}(t_2), B, A)\}, \{Sig_B, Sig_A\})$
$t_5 = (\{(g, 1, \text{hash}(t_3), C, B), (e, 1, \text{hash}(t_2), B, C), (f, 1, \text{hash}(t_2), B, C)\}, \{Sig_B, Sig_C\})$
$t_6 = (\{(e, 2, \text{hash}(t_5), C, B), (a, 2, \text{hash}(t_4), B, C)\}, \{Sig_B, Sig_C\})$
$t_7 = (\{(c, 1, \text{hash}(t_1), A, B), (g, 2, \text{hash}(t_5), B, A), (e, 3, \text{hash}(t_6), B, A)\}, \{Sig_B, Sig_A\})$
$t_8 = (\{(b, 1, \text{hash}(t_1), A, C), (h, 1, \text{hash}(t_3), C, A)\}, \{Sig_C, Sig_A\})$

Dabei steht A für Alice, B für Bob, C für Charlie und S für den Server. Die Transaktionen t_4 bis t_8 sind farblich identisch hervorgehoben wie in Abbildung 1 und Abbildung 2.

Beispiel 1

Beispiel

Alice möchte wissen, ob Bob die Karte C_1 besitzt. Charlie hat die Karte C_1 erstellt und Bob übertragen, dabei wurde Transaktion T_1 erstellt. Anschließend hat Bob die Karte C_1 mit Dave gegen die Karte C_2 getauscht, hierbei wurde Transaktion T_2 erstellt.

Bob sendet nun T_1 an Alice, verschweigt ihr aber T_2. T_1 ist offensichtlich eine gültige Transaktion, denn vor dem Tausch von Bob und Dave war sie bereits gültig. Solange Bob Alice nicht sagen will, dass er die Karte C_2 besitzt, muss er ihr die Transaktion T_2 nicht mitteilen.

Für Alice sieht es so aus, als ob Bob noch im Besitzt der Karte C_1 ist, aber die Karte C_2 nicht erhalten hat.

Beispiel 2

Das von uns verfolgte Konzept besitzt bei genauerer Betrachtung eine Schwachstelle. Wir können überprüfen und mit Sicherheit feststellen, ob ein Spieler eine Karte erhalten hat (s.o.). Jedoch können wir nicht überprüfen, ob der Spieler diese Karte bereits weitergab (siehe Beispiel 2). Es existiert keine zentrale Stelle zum Einsehen aller Transaktionen, stattdessen bitten wir in der Regel, den Besitzer uns seine Transaktionen zu übermitteln, um den Besitz überprüfen zu können. Dieser Bitte muss aber nicht wahrheitsgetreu nachgekommen werden. Zwar verhindern die Unterschriften, dass alte Transaktionen manipuliert werden können, aber nicht das Verschweigen selbiger.

Diese Schwachstelle kann durch bösartige Clients ausgenutzt werden, um Karten widerrechtlich zu verdoppeln. Obwohl ihm dies nur beschränkt Vorteile bringen würde (siehe Beispiel 3), muss davon ausgegangen werden, dass dies geschehen wird. Aus diesem Grund muss festgelegt werden, wie vom Netzwerk darauf reagiert wird.

Beispiel

Alice besitzt die Karte C_1, Bob die Karte C_2 und Charlie C_3. Alice tauscht C_1 mit Bob gegen C_2. Anschließend tauscht sie abermals C_1, nun jedoch mit Charlie gegen C_3 (Siehe Beispiel 2). Alice besitzt nun eine Menge an Transaktionen, um ihren Besitz der Karten C2 und C3 nachzuweisen. Da beide Transaktionen jedoch den Tausch von C_1 enthalten, kann sie niemandem beglaubigen, beide Karten zu besitzen. Sie muss sich jedes Mal entscheiden, welche der Karten sie besitzen möchte.

Sollten Bob und Charlie jemals miteinander ihre Transaktionen austauschen, werden sie den Betrug feststellen.

Beispiel 3

Um solche Betrugsfälle möglichst früh zu entdecken, können die Clients sich mehr Transaktionen merken als jene, welche sie benötigen, um anderen den Besitz ihrer Karten nachzuweisen. Im besten Fall merkt sich ein Client jede Transaktion, die er mal zu Gesicht bekam.

Dies verhindert natürlich nicht, dass jemand versucht, zu betrügen, macht es aber durch ein potentielles schnelleres Feststellen unattraktiver und grenzt zudem den Schaden ein, da weniger Nutzer Opfer des Betruges werden.

Wie groß der Schaden ist, der durch einen Betrug entsteht und was für eine Art von Schaden dies ist, hängt letzten Endes damit zusammen, wie mit betrügerischen Transaktionen umgegangen wird. Zunächst sehen wir uns eine durch einen Betrug entstandene, fehlerhafte Transaktion genauer an.

Wenn wir uns die Menge an Transaktionen als Transaktionsgraph betrachten (Abbildung 1, Seite 14), dann besitzt ein Knoten nur eine ausgehende Kante, falls er auch eine eingehende Kante mit derselben Beschriftung besitzt (es ist zu beachten dass in Abbildung 1 mehrere Kanten zu einer Kante zusammengefasst sind und deren einzelnen Beschriftungen mit Komma separiert sind). Wichtig ist, dass für jede eingehende Kante maximal eine ausgehende Kante existieren darf. Genau dies ist der Punkt, an dem die Betrügerin Alice aus Beispiel 3 eine fehlerhafte Transaktion konstruiert um die Karte C_1 ein zweites Mal abzugeben. Dabei gibt sie jedoch nur Teile des Graphen, den sie kennt, Preis, wodurch Charlie den Fehler nicht bemerkt, andernfalls würde er der Transaktion vermutlich nicht zustimmen.

Einen Graphen, der mindestens eine solche fehlerhafte Transaktion besitzt, nennen wir einen fehlerhaften Graphen. Dabei wird ein solcher durchaus von anderen Teilnehmern akzeptiert, da diese Art von Fehler behoben werden kann. Nicht akzeptiert wird hingegen ein Graph, bei dem die Unterschriften fehlerhaft sind oder benötigte Vorgängertransaktionen nicht existieren.

Das Akzeptieren fehlerhafter Graphen ist wichtig. Stellen wir uns einen theoretischen globalen Transaktionsgraphen vor, der alle jemals getätigten Transaktionen enthält. Dieser Graph spiegelt den Zustand wider, in dem sich das Netzwerk befindet. Jeder Nutzer besitzt ein eigenes Modell des Netzwerkes und nimmt dieses somit anders wahr. Basierend auf seinem Wissen über das Netzwerk bzw. die ihm bekannten Transaktionen baut er seinen eigenen Transaktionsgraphen. In Beispiel 3 haben Bob und Charlie in ihrem jeweils eigenen Transaktionsgraphen keine Fehler. Der globale Graph hingegen besitzt einen Fehler, da die Karte C_1 fälschlicherweise doppelt vertauscht wurde. Würde ein solcher Fehler nicht toleriert werden, hätten Charlie und Bob an dieser Stelle das Problem, die Transaktionen des jeweils anderen zu akzeptieren, sofern diese auf dem Tausch mit Alice basieren. Folge wäre eine unwiderrufliche Spaltung des Netzwerkes, womit jeder Betrug dem Netzwerk schweren Schaden zufügen würde.

Sobald einem Client ein Fehler auffällt, muss er behandelt werden. Wir müssen festlegen, wem Karten gehören, die in einer fehlerhaften Transaktion gehandelt wurden.

Die erste Variante wäre eine Bestrafung des Delinquenten, indem die von ihm erworbenen Karten vernichtet werden. Die Opfer hingegen besitzen nun ganz legal dieselbe Karte, was im Grunde einer Verdopplung dieser gleichkommt. Kompliziert wird es, wenn der Betrüger die erworbenen Karten bereits weiter gehandelt hat. In diesem Fall würde er die dadurch erhaltenen Karten verlieren.

Vorteil dieser Variante wäre, dass nur der Betrüger einen Schaden hat. Alle anderen Teilnehmer würden von weiteren Konsequenzen verschont bleiben und das Netzwerk würde nicht weiter gestört werden.

Indirekt entsteht den anderen Teilnehmern des Netzwerkes dennoch ein Schaden. Erstens verliert die verdoppelte Karte an Wert. Jemand könnte mittels wiederholter Verdopplung eine starke Inflation einer Karte herbeiführen. Zweitens wird ein Nutzer lediglich durch seinen öffentlichen Schlüssel identifiziert. Somit könnten mehrere Nutzer in Wahrheit ein Nutzer sein. Im Beispiel 3 könnten Alice, Bob und Charlie derselbe Nutzer sein, welcher lediglich 3 unterschiedliche private Schlüssel besitzt. In diesem Fall hätte dieser Nutzer sein Ziel erreicht und eine Karte verdoppelt. Ab diesem Zeitpunkt könnte er ganz legal mit den verdoppelten Karten spielen.

Dies führt also nicht zu einem sonderlich guten Schutz vor Betrügern. Die Tatsache, dass man auf so einfache Weise Karten verdoppeln kann, motiviert eher zum Betrug. Daher darf ein Konzept, das die Fehler in einem Graphen korrigiert, nicht dazu führen, dass Karten verdoppelt werden. Sofern bekannt ist, wie der Algorithmus vorgeht, kann ein Nutzer dies immer zur Vervielfältigung ausnutzen.

Es wäre wünschenswert, die zweite Transaktion rückgängig zu machen, da die erste in keinster Weise einen Fehler verursacht. Erst die zweite Transaktion führt zu Problemen. Leider besitzen die Transaktionen keine Informationen über den allgemeinen zeitlichen Ablauf. Daher kann nicht festgestellt werden, welche Transaktion zuerst durchgeführt wurde. Auch die Integration eines Zeitstempels in die Daten würde nicht weiterhelfen. Dieser könnte einfach falsch sein. Wollte man einen zuverlässigen Zeitstempel haben, müsste dieser von einer vertrauenswürdigen dritten Instanz erstellt werden. Allerdings widerspricht dies wiederum der Zielsetzung, dass ein Handel zwischen zwei Clients ohne Verbindung zum Internet abgehandelt werden kann.

Die dritte Variante, die später umgesetzt wurde, wickelt die Transaktionen zurück und der betrügerische Nutzer verliert die zuvor verdoppelte Karte. Im Beispiel 3 würden Bob und Charlie die Karte C_2 bzw. C_3 zurückerhalten und Alice die Karte C_1, so als hätte der Tausch nie stattgefunden. Zusätzlich wird die Karte C_1 als ungültig erklärt und kann nicht weiterverwendet werden. Sollten weitere Transaktionen auf diesen Karten aufbauen, so werden auch diese rückgängig gemacht. Dies kann zu einem Schneeballeffekt führen, bei dem viele Transaktionen auf einen Schlag revidiert werden. Würde in Abbildung 1 t_3 rückgängig gemacht werden, so würden ebenfalls die Transaktionen t_5 bis t_8 davon betroffen sein.

Für diesen Fall wird der am Anfang beschriebene Transaktionsindex benötigt. Ohne diesen könnte der Besitzer einer Karte, der die Karte aus einem revidierten Handel zurückerhalten hat, nicht erneut vertauschen. Weil der ursprüngliche und der zweite Tausch nicht voneinander zu unterscheiden wären, könnte man die aktuellere Transaktion nicht herausfinden. Dies ist der Grund das der Transaktionsindex um eins erhöht wird und somit ist die Reihenfolge eindeutig erkennbar ist.

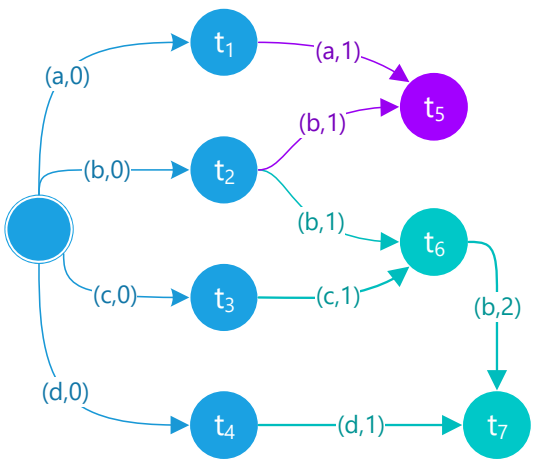


Abbildung 3

Abbildung 3 zeigt einen Transaktionsgraphen, der einen Fehler enthält: von dem Knoten t_2 gehen zwei Kannten mit derselben Beschriftung ab. Alice hat mit t_1 die Karte a erhalten, Bob mit t_2 b, Charlie mit t_3 die Karte c und Dave mittels t_4 die

Karte d. Dies sind die im Diagramm blau dargestellten Transaktionen. Bob tauscht b gegen Charlies c, welcher wiederum b weitergibt an Dave und dafür d bekommt. Somit entstehen die türkisenen Kanten und Knoten. Wiederrechtlich tauscht Bob jedoch b ein weiteres Mal mit Alice. Dies ist möglich, da Bob Alice die türkisenen Transaktionen verschweigt und sie kein Wissen über diese hat. Dies ist die violette Transaktion. Charlie und Dave wiederum wissen nichts von der Existenz dieser Kanten und Knoten.

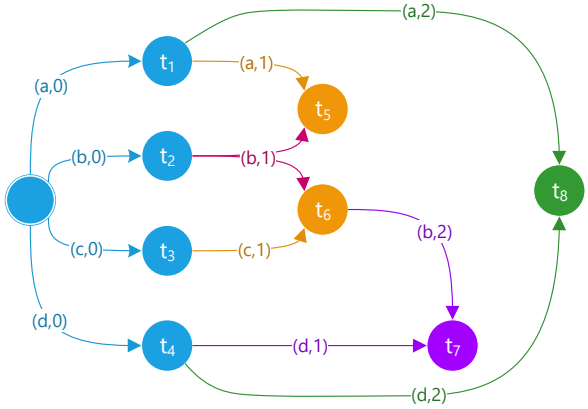


Abbildung 4

Tauschen sich jedoch Alice und Charlie oder Dave aus, fällt ihnen der Fehler auf und sie müssen die fehlerhaften Transaktionen rückgängig machen. Abbildung 4 zeigt, wie Alice und Dave die Transaktionen rückgängig machen und daraufhin ihre zurückerhaltenen Karten a und d miteinander tauschen. Rot markiert die Kanten, die den Fehler verursachen. Orange sind die Elemente, die direkt von dem Fehler betroffen sind und somit ungültig wurden. Aber auch die violette Transaktion muss rückgängig gemacht werden, da sie eine eingehende Kante einer der orangen Transaktionen besitzt. Würden weitere Transaktionen auf t_7 aufbauen, müssten diese ebenfalls revidiert werden. Abbildung 5 verdeutlicht nochmals den Fluss der Karten zwischen den einzelnen Nutzern. Dabei wurden die Kanten identisch eingefärbt wie in Abbildung 4 und stellen demnach keine Gruppierung der Transaktionen dar.

Zu beachten sind vor allem die Transaktionsindizes der grünen Transaktion. Ohne diese Inkrementierung könnte man nicht eindeutig feststellen, ob der Verstoß von Bob, Alice oder Dave ausgelöst wurde. Somit könnte Bob Erin, einem bisher Unbeteiligtem, beweisen, dass Alice die Karte a verdoppelt hatte, indem er diesem die Transaktionen t_5 und t_8 zeigt und seinen Betrug verschweigt. Anschließend könnte Bob Erin die Transaktion t_6 übermitteln, die nun aus Erins Sicht legal wäre, da Bob die Karte b wegen der vermeintlichen Verdopplung von a zurückerhalten hat. Dies führt zu einer Spaltung des Netzes, bei der unterschiedliche Nutzer zu unterschiedlichen Ergebnissen kommen. Je nachdem, in welcher Reihenfolge sie die Transaktionen erhalten haben, stellen sie unterschiedliche Fehlerquellen fest.

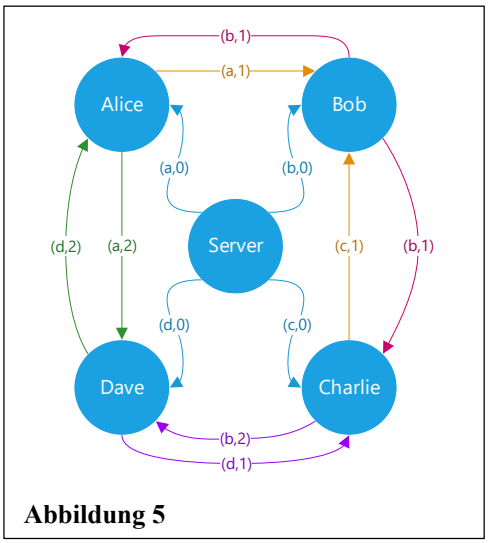


Abbildung 5

Fehlerhafte Transaktionen dürfen aber nicht einfach gelöscht werden. Dies hängt ebenfalls mit dem Transaktionsindex zusammen. Damit von der eingehenden Kante $(a, 0)$ von t_1 die ausgehende Kante $(a, 2)$ gültig ist, muss eine weitere ausgehende Kante $(a, 1)$ von t_1 existieren und diese muss aufgrund eines Fehlers als ungültig erklärt werden. Zu beachten ist, dass dies nicht die Kante sein darf, die den Fehler verursacht. Das heißt, dass

in Abbildung 4 die Karte b nicht nochmal mit einem höheren Transaktionsindex vergeben werden darf.

Das von uns verfolgte Konzept erlaubt, dass zwei Parteien miteinander Handeln ohne sich mit dritten abzustimmen. Dies wird dadurch erkauft, dass Teilnehmer betrügerische Geschäfte in Form von mehrfachem Handeln derselben Karte abschließen können, ohne dass dies sofort auffällt.

Leider kann natürlich auch dieses Verfahren missbraucht werden. Sofern es dem Angreifer nicht um einen eigenen Vorteil geht, sondern er das komplette System sabotieren möchte, kann er versuchen, möglichst viele Karten zu handeln. Nach einer gewissen Zeit, während diese Karten weitergehandelt wurden, erzeugt er eine Kartenverdopplung und kann somit eine große Menge an bereits abgeschlossenen Geschäften rückgängig machen.

3.1.2 Kartensicherheit

Neben der Tatsache, dass der Besitz einer Karte gesichert sein muss, muss auch die Karte selbst vor unerlaubten Manipulationen geschützt werden. Dies ist nötig, da die einzige Informationsquelle der jeweilige Partner ist. Denn in dem von uns gestalteten System soll es möglich sein, mit Personen zu spielen, ohne dass einer der Teilnehmer mit zusätzlichen Quellen in Verbindung treten muss. Sollte ein Spieler eine Karte einsetzen, die dem anderen unbekannt ist, so müssen diesem auch die Werte der Karte mitgeteilt werden. Daher besitzt jeder Spieler eine Kopie der Kartendaten, die er seinem Partner übermitteln kann. Somit ist sichergestellt, dass beide Spieler miteinander spielen und handeln können.

Da es im Interesse des Mitspielers liegt, möglichst gute Karten zu besitzen, müssen diese vor Manipulation geschützt werden. Aus diesem Grund werden alle wichtigen Daten vom jeweiligen Erzeuger signiert. Die Daten enthalten auch den öffentlichen Schlüssel, damit jeder, der diese übermittelt bekommt, überprüfen kann, dass die Integrität nicht verletzt wurde.

Das Einfügen des öffentlichen Schlüssels in die Daten besitzt noch einen zweiten Grund. Da jeder Teilnehmer Karten erstellen kann, und die Daten von Karten allgemein bekannt sind, ist jeder in der Lage, Kopien von beliebigen Karten zu erstellen und zu verteilen. Dies würde jedoch den Wert von eventuell sehr seltenen Karten stark reduzieren. Durch das Einbetten des öffentlichen Schlüssels des Erstellers können solche Kopien vom Original unterschieden werden.

Somit kann ein Spieler frei entscheiden, von welchem Ersteller er Karten akzeptiert.

3.2 Das Spielsystem

Das von uns entwickelte System soll ein Spiel zwischen exakt zwei Spielern ermöglichen. Mehr Teilnehmer an einer Partie werden nicht unterstützt.

Die in diesem Abschnitt erwähnten Karten sind von denen in Abschnitt 3.1 zu unterscheiden. Während es im vorherigen Abschnitt um Besitzverhältnisse ging, beschäftigen wir uns hier mit der Geheimhaltung von Karten während des Spiels. Somit existieren für Karten zwei unterschiedliche Datenstrukturen. Die Erste um den Besitzer einer Karte bestimmen zu können und die Zweite welche für eine Geheimhaltung verdeckter Karten während des Spiels verantwortlich ist. Letzter existieren nur für den Verlauf eines Spieles.

Das Spiel zwischen zwei Clients sollte dabei fair verlaufen. Zu beachten ist, dass ein Client keine Möglichkeit der Überprüfung hat, ob ein anderer Client manipuliert wurde. Um ein faires Spiel zu erreichen, sollten zwei Anforderungen erfüllt sein: Ein Client sollte nicht mehr Informationen besitzen, als sein Nutzer erfahren darf. Jedoch benötigt er gerade so viele Informationen, dass ihm ein versuchter Betrug durch einen anderen Client auffällt. Im folgendem werden zwei naive Implementierungen beschrieben, die jeweils nur eine der beiden Anforderungen genügt.

Gehen wir von einem beliebigen Kartenspiel aus, das pro Spieler aus folgenden Komponenten besteht: dem Nachziehstapel, der Hand und den ausgespielten Karten. Dabei sind die Karten des Nachziehstapels für niemanden einsehbar, die ausgespielten Karten sind für jeden einsehbar und die Karten auf der Hand nur für den jeweiligen Spieler.

In der ersten naiven Variante besitzen beide Clients vollständige Informationen über die Position aller verdeckten und offenen Karten. Wenn ein Spieler eine Karte aus der Hand ausspielt, kann jeder Client überprüfen, ob diese Karte sich auch tatsächlich auf der Hand des Spielers befand. Dies sind jedoch zum Teil Informationen, die der Spieler nicht besitzen darf. Er darf nicht wissen, welche Karte er als nächstes zieht und welche Karten sein Gegenüber besitzt. Ein manipulierter Client könnte diese Informationen an den Nutzer weitergeben und ihm somit einen unfairen Vorteil verschaffen.

Für die zweite naive Variante verwaltet der Client nur Informationen zu Karten, welche dem Spieler gehören. Die Position der Karten des anderen Spielers sind ihm nicht bekannt. Lediglich die Anzahl der Karten auf den verschiedenen Positionen kennt er. Welche Karte dies nun im jeweiligen Fall ist, ist ihm jedoch unbekannt. Damit ist er nicht mehr in der Lage zu wissen, welche Karten der Mitspieler ausspielen kann. In Folge dessen kann er aber auch nicht mehr überprüfen, ob ein Mitspieler die Karte, die er gerade spielt, auch auf der Hand hatte. Ein manipulierter Client kann jederzeit eigene Karten, die nicht offen liegen, austauschen. Damit kann ein Spieler das Glückselement zum großen Teil ausschalten. Für ihn ist lediglich die Anzahl der Karten auf seiner Hand interessant. Welche dies sind, ist unbedeutend, da er jede beliebige Karte, auch aus seinem Nachzugsstapel, nutzen kann. Auch dies stellt gegenüber einem fairen Client einen großen Vorteil dar.

In diesem Projekt versuchen wir, ein System zu entwickeln, welches keine der beiden oben beschriebenen Probleme der naiven Implementierungen besitzt. Dabei gehen wir von folgenden Rahmenbedingungen aus:

- Zu Beginn einer Partie sind alle Karten bekannt und welchem Spieler diese zugeordnet sind.
- Die Spielregeln sind bekannt und für beide Teilnehmer gleich[1].

Bei einem Mau-Mau-Spiel wäre beispielsweise bekannt, dass mit einem standardmäßigen Skatblatt gespielt wird nach einer Variante der Mau-Mau-Regeln.

[1] Die Regeln dürfen den Spielern dennoch unterschiedliche Rollen zuweisen, wie z.B. der Croupier beim Black Jack.

Das implementierte System soll verschiedene Kartenspiele abbilden können. Hierzu legen wir einen Rahmen an Funktionalität fest, der das System zur Verfügung stellt, um die Regeln zu implementieren. Die Regeln selbst werden mittels einer Skriptsprache fixiert.

- Jedes Spiel kann unterschiedliche Bereiche definieren. Jede Karte muss in einem Bereich liegen. Bereiche können beispielsweise Nachziehstapel, Ablagestapel oder die Hand eines Spielers sein. Siehe Beispiel 4.
- Jede Karte in einem Bereich besitzt eine Indexposition und kann somit eindeutig über Bereich und Position benannt werden.
- Karten können offen und verdeckt liegen. Verdeckte Karten sind nur durch ihre Position unterscheidbar, der Wert der Karte ist nicht einsehbar. Offene Karten erlauben es, ihren Wert auszulesen, zum Beispiel Pick Sieben.
- Karten können für einen Spieler verdeckt sein und für einen anderen offen.
- Karten können von einem Bereich in einen anderen geschoben werden.
- Karten in einem Bereich können gemischt werden.

Mau-Mau

Das Mau-Mau-Spiel besitzt einen Nachzugstapel und einen Ablagestapel, der von allen Spielern genutzt wird und einen Bereich für jeden Mitspieler, der als Hand fungiert. Die Anzahl der Bereiche hängt somit von der Spieleranzahl ab und beträgt $Spieleranzahl + 2$ Bereiche.

Beispiel 4

Wir haben zwei verschiedene Möglichkeiten untersucht, dieses Ziel zu erreichen. Die erste Variante nutzt die Eigenschaft von Pseudozufallszahlengeneratoren reproduzierbare Sequenzen von Zahlen zu erzeugen. Somit kann ein Spiel nach dessen Ende auf Unregelmäßigkeiten überprüft werden. Die zweite Variante nutzt das „Mental Card Game“ Protokoll vorgestellt in (Schindelhauer 1998).

3.2.1 Konzept auf Basis eines deterministischen Zufallszahlengenerators

Dieses Konzept basiert auf einem Pseudozufallszahlengenerator. Zu Beginn einer Partie erzeugt jeder Client einen (echten) zufälligen Startwert, merkt sich diesen und nutzt ihn zur Initialisierung seines Pseudozufallszahlengenerators. Die Position aller Karten ist zu Anfangs bekannt. Alle Spielzüge werden aufgezeichnet. Dadurch kann im Nachhinein überprüft werden, ob einer der beiden Spieler betrogen hat. Während des Spieles muss man hingegen dem anderen Spieler vertrauen.

Die meisten Spiele mischen zu Beginn die Kartenstapel verdeckt, da bei einem Kartenspiel oft der Zufall eine wichtige Rolle spielt und daher ein sortierter Kartenstapel zu vorhersehbar ist. Bei diesem Konzept wird der Bereich als gemischt markiert, die Reihenfolge der Karten aber bleibt unverändert.

Eine wichtige Voraussetzung des Pseudozufallszahlengerators ist, dass man von der Sequenz der ersten n generierten Zahlen keinen Startwert errechnen kann, welcher dieselbe Sequenz generiert.

In den meisten Spielen existiert ein gemischter Zugstapel, von dem ein Spieler zieht und die gezogene Karte auf die Hand nimmt. Dies bedeutet, er bewegt die erste (oberste) Karte aus dem Bereich Nachziehstapel in den Bereich Hand. Dabei kennt niemand die Reihenfolge der Karten des Nachziehstapels und außer dem ziehenden Spieler kennt niemand die Karten auf dessen Hand inklusive der gerade gezogenen.

In diesem Konzept wurde der Stapel lediglich als gemischt markiert, die Karten in ihrer Position aber nicht verändert. Anstelle der obersten Karte wird eine zufällige Karte aus dem Stapel gezogen. Anstelle des Mischens, nachdem die Karten in unbekannter Reihenfolge vorliegen, wird beim Ziehen einer Karte eine zufällige gezogen. Somit weiß niemand vor dem Ziehen, welche es sein wird. Der Client, der die Karte zieht, teilt dem anderem Client mit, dass er aus einem Bereich eine Karte entfernt, jedoch nicht von welcher Position.

Bisher hindert niemand einen Client daran, die nächsten Zufallszahlen im vorherein zu berechnen. Somit könnte er herausfinden welche Karte er als nächstes zieht. Um dies zu verhindern, wird eine weitere Zufallszahl des anderen Clients verlangt. Mit beiden Zahlen wird untereinander eine Kontravalenz durchgeführt. Die daraus entstandene Zahl wird mittels Modulo in den Bereich der möglichen Indizes projiziert und gibt den Index der Karte an, die gezogen werden soll.

Sobald eine Karte offen gespielt wird, teilt der Client, der die Karte kennt, dem anderem Client mit, welche Karte gespielt wurde. Beispiel 5 beschreibt diesen Vorgang.

Beispiel

Alice und Bob wollen eine Partie spielen. Es existiert ein Bereich, der als Zugstapel fungiert. Die Karten des Zugstapels sind als gemischt markiert und verdeckt. Die originale Reihenfolge der Karten lautet: Pick 7, Karo 2, Karo Dame.

Alice will nun eine Karte ziehen und anschließend ausspielen. Dazu generiert sie eine neue Zufallszahl. Ebenso generiert Bob eine Zufallszahl und teilt diese Alice mit. Da nur Alice beide Zahlen kennt, weiß sie, welche Karte sie bekommt. Bob weiß nur, dass Alice eine von den drei genannten Karten bekommen wird. Er kann nicht darauf schließen, welche Karte es war.

Alice generiert die Zahl 5, und Bob die Zahl 4. Die Kontravalenz beider Zahlen ist eins. Somit zieht Alice die Karo 2. Anschließend teilt Alice Bob mit, dass sie die Karo 2 ausspielt.

Am Ende der Partie legen Alice und Bob die Startwerte ihrer Pseudozufallszahlengeneratoren offen. Bob kann so nachvollziehen, dass Alice die Karo 2 gezogen hatte und ausspielen konnte.

Hätte Alice betrogen und zu Bob gesagt, dass sie die Karo Dame ausspielt, ohne sie vorher gezogen zu haben, müsste sie ebenfalls ihren Startwert manipulieren. Denn die, zu dem oben beschriebenen Zeitpunkt, generierte Zahl müsste 7 sein. Andernfalls würde Bob erfahren, dass Alice die Karo Dame nicht besaß, als Alice sagte, sie würde diese ausspielen.

Beispiel 5

Beispiel

Alice zieht von einem verdeckten Stapel eine Karte ohne diese Bob zu zeigen. Bob erzeugt eine Zufallszahl und sendet diese an Alice, die ebenfalls eine Zufallszahl erzeugt. Mit diesen beiden Zahlen weiß Alice, welche Karte des Stapels sie zieht. Da Bob die zweite Zahl fehlt, kann er nicht wissen, welche der Karten gezogen wurde.

Würde Bob ebenfalls aus diesem Stapel eine Karte ziehen wollen, ohne diese Alice offen zu legen, müsste Alice ihm eine Zufallszahl nennen und Bob seine eigene generieren. Mit dieser hat er den Index der Zahl, die er ziehen muss. Eine der Karten wurde bereites von Alice gezogen und darf von Bob nicht nochmals gezogen werden. Da Bob nicht weiß, welche Karte er nicht mehr ziehen darf, muss Alice ihm die Karte zuteilen. In diesem Fall weiß Alice, welche Karte Bob erhalten hat.

Beispiel 6

Dieses Verfahren unterliegt einigen Einschränkungen. Jeder Spieler besitzt seine eigenen Karten und kann nur eingeschränkt mit den Karten des anderen Spielers interagieren. Ein Ziehen beider Spieler von demselben verdeckten Stapel ist nicht ohne weiteres möglich, wie Beispiel 6 auf vorheriger Seite verdeutlicht.

Eine weitere Einschränkung dieses Verfahrens ist die Möglichkeit, die Strategie eines Spielers zu analysieren. Es basiert auf der Offenlegung aller Informationen in Form einer globalen Übersicht nach einem Spiel. Man kann so rekonstruieren, welche Karten ein Spieler zu einem beliebigen Zeitpunkt auf der Hand hatte und wie er sich zu diesem Zeitpunkt verhalten hat. Insbesondere bei Spielen, bei denen Bluffen essentieller Bestandteil ist, stellt dieses Verfahren somit ein Problem dar.

3.2.2 Mental Card Game

Das in „A Toolbox for Mental Card Games“ (Schindelhauer 1998) beschriebene Protokoll leidet nicht unter den im vorherigen Abschnitt aufgeführten Nachteilen. Dadurch kann sich bei der Implementierung eines Regelsatzes an Originalkartenspielen orientiert werden, ohne dass bestimmte Aspekte nicht umsetzbar sind.

Dieses Verfahren basiert auf einem Public-, Private-Key-Verfahren, das sich auf den Quadratischen Rest stützt. Denn es existiert kein allgemeines effizientes Verfahren, um festzustellen, ob eine Zahl ein Quadratischer Rest bezüglich eines Modulo ist. (Goldwasser und Micali 1982)

Bei dem von Schindelhauer verwendeten Verfahren besitzt jeder Spieler einen privaten Schlüssel bestehend aus zwei großen Primzahlen $p, q \in \mathbb{P}/\{2\}$, für die gilt $p \neq q$. Aus diesen wird der öffentliche Schlüssel hergeleitet mit $m = p * q$ und $y \in \mathbb{PQR}_m$. Im weiteren Verlauf stellen p_1, q_1, m_1, y_1 die Teile des Schlüssels von Spieler 1 dar und p_2, q_2, m_2, y_2 die Teile des Schlüssels von Spieler 2.

Jeder Karte im Spiel wird eine eindeutige Binärcodierung zugeordnet. Die minimale Anzahl von Bits, die man zur Codierung von N Karten benötigt, ist $\omega = \lceil \log_2 N \rceil$. Damit ist der Typ der Karte $T = b_1 b_2 \dots b_\omega$ mit $b_i \in \{0,1\}$. Im Weiteren verlauf dieser Arbeit hat

jeder Typ einer Karte eine Nummer τ zwischen 0 und $N-1$ zugeordnet, sodass $\tau = \sum_{j=1}^{\omega} 2^{j-1} * b_j$.

Sämtliche Karten werden durch ein Tupel von $\omega * 2$ Zahlen repräsentiert[2]. Diese Tupel sind allen Spielern bekannt. Jeweils zwei Zahlen des Tupels repräsentieren ein Bit aus T. Für alle Zahlen gilt $z_{i,j} \in \mathbb{J}_{m_i}^1$.

$$C = (z_{1,1}, \dots, z_{1,\omega}, z_{2,1}, \dots z_{2,\omega})$$

Das Bit $b_j = b_{1,j} \oplus b_{2,j}$, wobei $b_{i,j}$ in der Zahl $z_{i,j}$ wie folgt kodiert ist:

$$b_{i,j} = qr(z_{i,j}, m_i) = \begin{cases} 0, & z_{i,j} \in \mathbb{QR}_{m_i} \\ 1, & else \end{cases}$$

Um nun aus dem Tupel C den Typ T oder das korrespondierende τ zu berechnen, nutzen wir folgende Formel:

$$\tau = \sum_{j=1}^{\omega} 2^{j-1} * b_{1,j} \oplus b_{2,j}$$

Da die Funktion $qr(z, m)$ nur effizient berechenbar ist, wenn man die Primfaktoren p und q von m kennt (Schindelhauer 1998), ist ein Auslesen des Typs T einer Karte nur mithilfe aller Spieler möglich.

Schindelhauer beschreibt verschiedene Operationen die auf einzelnen Karten oder Stapeln von mehreren Karten ausgeführt werden können. Er unterscheidet zwischen Operationen, die ein Spieler ohne Zutun von anderen unter Kenntnisnahme der öffentlichen Schlüssel ausführen kann, und solchen Operationen, bei denen die aktive Mithilfe anderer Spieler z.B. durch Berechnung von $qr(z_{i,j}, m_i)$ benötigt werden.

Operationen, die ohne Mithilfe anderer Spieler auskommen:

- Erstellen von Karten
- Maskieren von Karten
- Permutation von Karten in einem Stapel

Operationen, welche die Mithilfe anderer Spieler benötigen:

- Aufdecken einer verdeckten Karte (für einen oder mehrere Spieler)
- Mischen eines Kartenstapels

3.2.2.1 Erstellen einer Karte

Um eine offene Karte vom Typ $T = b_1 b_2 \dots b_\omega$ zu erstellen, erzeugt man folgendes Tupel: $C = (y_1^{b_1}, y_1^{b_2}, \dots, y_1^{b_\omega}, 1, 1, \dots, 1)$. Dabei ist zu beachten das $y_1 \in \mathbb{PQR}_{m_1}$ ist und dem Eins-Bit entspricht. Hingegen ist $1 \in \mathbb{QR}_{m_1}$ und $1 \in \mathbb{QR}_{m_2}$, was dem 0 Bit entspricht. Da y_1

[2] Dies stimmt nur bei Spielen mit zwei Spielern. Würde man ein Spiel für mehr Spieler implementieren, wäre das Tupel größer.

Teil des öffentlichen Schlüssels von Spieler 1 ist und die 1 immer in $\mathbb{QR}_m$, für jedes beliebiges m, enthalten ist, ist der Typ der Karte für jeden Spieler ersichtlich.

Beispiel

Sei die Binärkodierung des Typs $T = 010$, dann wird die offene Karte C wie folgt repräsentiert: $C = (1, y_1, 1,1,1,1)$.

Notieren wir anstelle der Zahlen die Mengen, der die Zahlen zugeordnet werden, können wir $C = \left(\mathbb{QR}_{m_1}, \mathbb{PQR}_{m_1}, \mathbb{QR}_{m_1}, \mathbb{QR}_{m_2}, \mathbb{QR}_{m_2}, \mathbb{QR}_{m_2}\right)$ schreiben.

Dies entspricht den Bits $C = (0,1,0,0,0,0)$. Nutzen wir die Kontravalenz Operation auf dem ersten und vierten, zweiten und fünften und dritten und sechsten Bit, erhalten wir wieder unseren Bitvektor 010.

Beispiel 7

3.2.2.2 Maskieren einer Karte

Das Maskieren einer Karte ist eine Operation, die von einer gegebenen Karte C vom Typ T eine Karte C' vom Typ T erstellt, wobei $C \neq C'$. Für diese Operation muss T nicht bekannt sein. Schindelhauer definiert für die Operation des Maskierens von C nach C' den Operator $\twoheadrightarrow$ (Siehe Algorithmus 1).

$C \twoheadrightarrow C'$

1. Für jedes $z_{i,j}$ erzeuge ein zufälliges $r_{i,j} \in \mathbb{Z}_{m_i}^*$ und $\forall j \in \{1, \dots, \omega\}$ erzeuge ein zufälliges $b_j \in \{0,1\}$
2. Erzeuge neue Karte $C' = \left(z'_{1,1}, \dots z'_{2,j}\right)$ mit $z'_{i,j} = z_{i,j} * r_{i,j}^2 * y_i^{b_j} \; mod \; m_i$
3. Beweise anderen Spielern, dass $C \twoheadrightarrow C'$ korrekt ist.

Algorithmus 1

Dass C und C' denselben Typ T besitzen, können wir uns einfach veranschaulichen. Wir wissen, dass $z_{1,j} \in \mathbb{J}_{m_1}^1$ und $z_{2,j} \in \mathbb{J}_{m_2}^1$ ist. Zudem wissen wir, dass $r_{i,j}^2 \in \mathbb{QR}_{m_i}$ ist.

Ist bit_j das j-te Bit des Typs T und $bit_j = 0$, so ist entweder $z_{1,j} \in \mathbb{QR}_{m_1}$ und $z_{2,j} \in \mathbb{QR}_{m_2}$ oder $z_{1,j} \in \mathbb{NQR}_{m_1}$ und $z_{2,j} \in \mathbb{NQR}_{m_2}$. Ist nun unser zufällig gewähltes $b_j = 1$ und $z_{i,j} \in \mathbb{QR}_{m_i}$, dann ist $z'_{i,j} \in \mathbb{NQR}_{m_i}$. Ist hingegen $z_{i,j} \in \mathbb{NQR}_{m_i}$, dann ist $z'_{i,j} \in \mathbb{QR}_{m_i}$. Sollte stattdessen $b_j = 0$ sein, so fließt y_i nicht in $z'_{i,j}$ ein und $z'_{i,j} \in \mathbb{QR}_{m_i} \Leftrightarrow z_{i,j} \in \mathbb{QR}_{m_i}$.

Da b_j sowohl $z'_{1,j}$ als auch $z'_{2,j}$ beeinflusst, ändert sich der Wert von bit_j nicht. Sollte $z_{1,j}$ sich in derselben Menge befinden wie $z_{2,j}$, so befinden sich auch $z'_{1,j}$ und $z'_{2,j}$ in derselben Menge. Waren beide in unterschiedlichen Mengen, so sind auch ihre abgeleiteten Werte in unterschiedlichen Mengen.

An dieser Stelle erweitern wir die Notation von Schindelhauer auf $C \overset{\sigma}{\hookrightarrow} C'$ wobei $\sigma = (r_{1,1}, \ldots r_{1,\omega}, r_{2,1}, \ldots, r_{2,\omega}, b_1, \ldots, b_\omega)$ die Parameter der Maskieren-Operation darstellt.

Ein wichtiger Schritt nach dem Maskieren ist das Verifizieren der anderen Spieler, dass dieses richtig durchgeführt wurde und der Spieler, der die Maskierung durchführte, die Parameter $r_{i,j} \in \mathbb{Z}^*_{m_i}$ und $b_j \in \{0,1\}$ kennt. Die genannten Parameter lassen sich nicht effizient aus C und C' herleiten. Sollte die Verifizierung nicht durchgeführt werden, kann nicht garantiert werden, dass der maskierende Spieler beim späteren Demaskieren der Karte mehr Informationen erhält als ihm zustehen. (Schindelhauer 1998, 13)

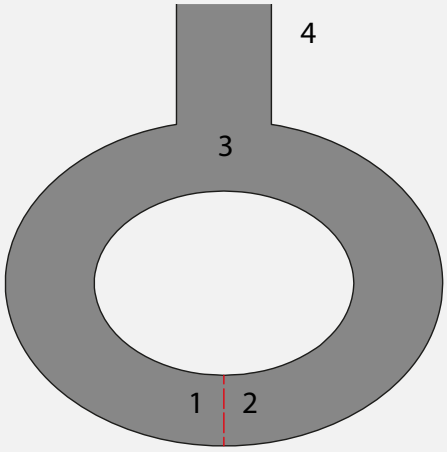
Beispiel

„Peggy möchte Viktor beweisen, dass sie über geheimes Wissen verfügt (eine Tür in einer Höhle öffnen kann), ohne dass sie vor seinen Augen die Tür öffnet (und somit ihm und Dritten zeigt, wie die Tür zu öffnen ist).

Viktor steht bei 4 und sieht Peggy in die Höhle gehen. Er sieht nicht, ob Peggy nach 1 oder 2 geht. Dann geht Viktor zu 3 und verlangt von Peggy, dass sie auf einer von ihm festgelegten Seite aus der Höhle kommt. Je nachdem, ob Peggy an der richtigen Seite steht, muss sie dafür die rote Tür öffnen oder nicht. Kann Peggy die Tür öffnen, kann sie stets auf der von Viktor verlangten Seite hervorkommen. Kann sie die Tür nicht öffnen, erscheint sie in 50 % der Fälle auf der falschen Seite.

Kommt Peggy nach *n* Versuchen jedes Mal auf der von Viktor verlangten Seite aus der Höhle, kann Viktor mit einer Wahrscheinlichkeit von $1 - 2^{-n}$ davon ausgehen, dass Peggy das Geheimnis der Tür kennt, hat aber dennoch kein neues Wissen über die Tür erlangt, z. B. ob diese nur von einer Seite zu öffnen ist.

Beobachtet ein Dritter, was Viktor sieht, so ist er nicht davon überzeugt, dass Peggy das Geheimnis der Tür kennt, da sich Viktor und Peggy abgesprochen haben können, welche Seite Viktor in jedem der Durchgänge verlangen wird.“



(Zero-Knowledge-Beweis - Wikipedia 2015)

Beispiel 8

Gehen wir davon aus, Alice hat eine Karte maskiert, $C \overset{\sigma}{\nrightarrow} C'$. Bob möchte wissen, ob Alice den Parameter σ kennt. Alice will Bob dies beweisen, ohne ihm Informationen über σ mitzuteilen.

Hierzu wird ein kenntnisfreier Beweis (zu eng. Zero Knowledge Proof, siehe Beispiel 8) genutzt. Dabei profitieren wir davon, dass das Maskieren einer Karte eine transitive Operation ist. Alice erzeugt eine Karte C'' für die gilt $C' \overset{\pi}{\nrightarrow} C''$ und sendet diese an Bob. Daraufhin entscheidet sich Bob dafür, dass Alice ihm die geheimen Parameter von $C' \overset{\pi}{\nrightarrow} C''$ oder von $C \overset{\tau}{\nrightarrow} C''$ mitteilt. $C \nrightarrow C''$ ist möglich, weil $\tau = \sigma \circ \pi$ aus σ und π hergeleitet werden kann. Alice kann nur dann in beiden Fällen korrekt antworten, wenn C' aus C erstellt wurde, C'' aus C' und Alice für beide Operationen die geheimen Parameter σ und π kennt. Durch dieses Verfahren erhält Bob keinerlei Informationen über σ für $C \overset{\sigma}{\nrightarrow} C'$. Zudem erhält er eine 50% Chance Alice zu überführen, sollte Sie betrogen haben. Um ein ausreichendes Sicherheitsniveau zu erreichen, wiederholt man diesen Test s-mal. Somit erhöht man die Wahrscheinlichkeit, Alice zu überführen, auf $1 - 2^{-s}$. Alice muss dabei beachten, dass jedes erzeugte C'' sich von den anderen unterscheidet. Andernfalls erhält Bob die Parameter τ von $C \overset{\tau}{\nrightarrow} C''$ und π von $C' \overset{\pi}{\nrightarrow} C''$, um somit den Parameter σ von $C \overset{\sigma}{\nrightarrow} C'$ zu berechnen.

Mit diesem Beweis kann Alice nur Bob nachweisen, dass sie nicht betrogen hat. Eine dritte Partei, zum Beispiel Charlie, welcher Zeuge der Kommunikation ist, hat keine Möglichkeit der Verifikation. Würden Alice und Bob kollaborieren, kann Bob seine Auswahl so treffen, dass Alice nicht überführt wird, obwohl sie betrogen hat.

Das Maskieren einer Karte funktioniert bei einer offenen sowie maskierten (verdeckten) Karte. Das Maskieren einer einzelnen Karte erscheint sinnlos, genauso wie es in einem realen Kartenspiel sinnlos ist, eine einzelne offene Karte zu verdecken. Ohne das sie mit anderen zusammen gemischt wird, kann sich jeder Spieler behalten was der ursprüngliche Wert war. Dennoch ist es eine essentielle Operation, da sie auch beim später betrachteten Mischen der Karten benötigt wird.

3.2.2.3 Kartenstapel permutieren

Einen Kartenstapel zu permutieren, bedeutet jeder Karte des Stapels eine neue Position zu geben und die Karte zu maskieren. Kein Spieler, außer dem Ausführenden der Permutation, weiß, an welcher Position sich eine Karte befindet.

Permutieren wir den Kartenstapel S zu S', dann schreiben wir $S \overset{P_\sigma}{\approx} S'$, wobei P_σ eine Matrix darstellt. Diese Matrix enthält neben den Maskierungsparametern für die einzelnen Karten die Permutationreihenfolge:

$$P_\sigma = \begin{pmatrix} \sigma(1) & r_{1,(1,1)} & \cdots & r_{1,(1,\omega)} & r_{1,(2,1)} & \cdots & r_{1,(2,\omega)} & b_{1,(1)} & \cdots & b_{1,(\omega)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \sigma(d) & r_{d,(1,1)} & \cdots & r_{d,(1,\omega)} & r_{d,(2,1)} & \cdots & r_{d,(2,\omega)} & b_{d,(1)} & \cdots & b_{d,(\omega)} \end{pmatrix}$$

d entspricht der Anzahl der Karten. Die bijektive Funktion $\sigma(x)$ ordnet zu jedem Index $x \in \{1, \dots, d\}$ einen neuen Index $y \in \{1, \dots, d\}$ zu.

Es ist wichtig, allen Mitspielern zu beweisen, dass S' auf legalem Weg entstanden ist. Weil diese Operation transitiv ist, kann man bei der Permutation dasselbe kenntnisfreie Beweisverfahren nutzen, wie schon bei der Maskierung der Karten.

Alice will Bob beweisen, dass $S \overset{P_\sigma}{\approx} S'$ gilt. Bob wählt den Sicherheitsparameter s, die Anzahl von Herausforderungen angibt, denen Alice standhalten muss. Daraufhin generiert Alice S_1'' bis S_s'', für die gilt $S' \overset{P_{\pi_i}}{\approx} S_i''$. Bob wählt für jedes S_i'', ob er den Parameter P_{π_i} oder $P_{\tau_i} = P_{\sigma \circ \pi_i}$ übermittelt bekommen soll. Auf diese Weise kann Bob Alice mit einer Wahrscheinlichkeit von $1 - 2^{-s}$ überführen, falls sie betrügen sollte.

3.2.2.4 Aufdecken einer verdeckten Karte

Um eine Karte einem Spieler offenzulegen, berechnet der andere Spieler, ob die ihm zugeordneten $z_{i,j}$ in $\mathbb{QR}_{m_i}$ oder in $\mathbb{NQR}_{m_i}$ enthalten sind.

> **Beispiel**
>
> Alice und Bob spielen ein Spiel. Alice will eine verdeckte Karte C aufheben. Bob berechnet, in welcher Menge die jeweiligen Zahlen sind, die mit seinem öffentlichen Schlüssel verschlüsselt wurden und übersendet das Ergebnis an Alice. Alice berechnet die zu ihrem öffentlichen Schlüssel gehörenden Werte.
>
> Da Alice nun alle Bits der Karte dekodiert hat, kennt sie den Typ der Karte. Bob hingegen fehlen die von Alice berechneten Informationen. Daher kann er nicht auf den Typ von C schließen.

Beispiel 9

Soll eine Karte allen Spielern offengelegt werden, so publizieren alle Spieler ihre Informationen und jeder kann nun den Typ der Karte mittels Kontravalenz berechnen.

3.2.2.5 Mischen eines Kartenstapels

Um einen Kartenstapel zu mischen, nutzen wir das Verfahren zum Permutieren von Kartenstapeln. Zur Erinnerung, bei der Permutation ändert ein Spieler die Reihenfolge der Karten eines Stapels, sodass kein anderer Spieler die neue Reihenfolge kennt.

Um nun einen Kartenstapel zu mischen, lassen wir diesen von jedem Spieler permutieren. Hiernach besitzt kein Spieler genügend Informationen, um die Position der Karten zu bestimmen. Erst alle Spieler zusammen können die neue Reihenfolge der Karten rekonstruieren. Sollten aber alle Spieler zusammenarbeiten, könnten sie die Informationen auch mit ihren privaten Schlüsseln erlangen.

4 Implementierung

Der Client wurde in C# ursprünglich als Windows 8 App realisiert. Im Verlauf der Arbeit wurde das Projekt auf Windows 10 umgestellt, um die besseren Debug-Möglichkeiten auszunutzen. Es existiert noch eine Buildkonfiguration für Windows 8, die aufgrund des fehlenden Testsystem nicht mehr getestet werden konnte.

4.1 Projektmappenstruktur

Der Quellcode ist in mehreren Projekten aufgeteilt. Diese sind in einer Projektmappe gebündelt. Lässt man Projekte, die später vom Client nicht benötigt werden, außen vor, besteht der Client immer noch aus 27 Projekten. Diese große Anzahl hat zwei Gründe: Erstens führt dies zu einer besseren Kapselung der Daten, da C# einen Zugriffsmodifizerer besitzt, der nur den Zugriff innerhalb eines Projektes erlaubt. Hiervon profitieren vor allem Projekte, welche die Datenbanklogik bereitstellen, da die Kapselung der Datenhaltungsklassen die Nutzung der Datenbank sicherer gestaltet.

Zweitens erlaubt die Aufteilung das Gesamtprojekt vergleichbar einfach auf andere Plattformen zu portieren. Ein Großteil des Quellcodes kann bereits als Desktop-Applikation kompiliert werden. Lediglich eine entsprechende Benutzeroberfläche müsste noch erstellt werden.

Schauen wir uns die Security-Projekte an. Insgesamt existieren vier Projekte: Security, *Security.Interfaces*, *Security.Desktop* und *Security.Store*. Klassen, welche die Windows Store API nutzen, sind in *Security.Store* enthalten. *Security.Desktop* hingegen besitzt Klassen, die gegen die .NET API programmiert wurden. Beispielsweise existiert in beiden Projekten eine Klasse `PublicKey`. *Security.Desktop* kompiliert zu einer DLL, die in normalen .NET-Anwendungen geladen werden kann. *Security.Store* hingegen kann nur von Windows Store Apps verwendet werden. *Security.Interfaces* wiederum kompiliert zu einer PCL-DLL. Diese kann in den meisten Applikationen plattformunabhängig verwendet werden. In dem Projekt sind Interfaces enthalten, die unabhängig von der Zielplattform verwendet werden sollen.

Portable Class Library

Portable Class Library Projekte (kurz PCL-Projekt, z.dt. portable Klassenbibliotheks-Projekte) können von verschiedenen Plattformen verwendet werden. Normalerweise wird eine Assembly nur für eine Plattform kompiliert. Abhängig von der Konfiguration des PCL-Projektes werden verschiedene Plattformen unterstützt. Folgende Liste stellt einen Auszug möglicher Plattformen dar: .Net, Xbox360, Silverlight, Windows Phone, Windows 10 App. Erreicht wird die Portabilität durch die Beschränkung auf den kleinsten gemeinsamen Nenner der Standardschnittstellen der jeweiligen Plattformen.

Erläuterung 2

Das letzte Projekt *Security* stellt eine Besonderheit dar. Es wird nicht zu einer Assembly kompiliert, sondern sein Quellcode wird in die referenzierenden Projekte kopiert. Dies erlaubt, den Code nur einmal zu schreiben, diesen aber über verschiedene Projekte zu kompilieren, ohne den Umweg über eine PCL-DLL zu gehen. Aufgrund starker Einschränkungen im Bereich der verwendbaren APIs, denen PCL-Projekte unterliegen, kann dies Sinn ergeben. In solchen Projekten kann dann über #if-Direktiven gezielt Plattformspezifischer Code ausgeführt werden welcher in PCL-DLLs nicht zur Verfügung steht. Abbildung 6 zeigt einen kleinen Ausschnitt des Klassendiagramms dieser vier Projekte.

Fasst man die einzelnen Projekte thematisch zusammen, so erhält man eine gute Übersicht über den Client. Diese thematische Gruppierung spiegelt sich in den Projektordnern wieder und den Namenspräfixen den sich die Projekte teilen. Die einzelnen Bereiche werden im Folgenden kurz erläutert. In Abbildung 8 und Abbildung 9 wird nochmal eine Übersicht der Zuordnung der einzelnen Projekte in die verschiedenen Bereiche gegeben. Abbildung 7 zeigt die Benutzungsbeziehungen der einzelnen Bereiche an. (Diese Abbildungen stehen auf Seite 34 f.)

4.1.1 Client

Die Projekte, die zu diesem Bereich gehören, beinhalten den Quellcode für die Oberfläche und für die Spiellogik.

4.1.2 Database

In diesen Projekten ist ein SQLite-OR-Mapper implementiert, der alle Informationen über Karten und Server abspeichert, exklusive der Transaktionen.

4.1.3 TransactionMap

Der Transaktionsgraph ist ebenfalls als SQLite-OR-Mapper realisiert. Aufgrund der zusätzlich benötigten Logik zum Validieren und Tauschen von Transaktionen wurde dies von der restlichen Datenbank abgekoppelt.

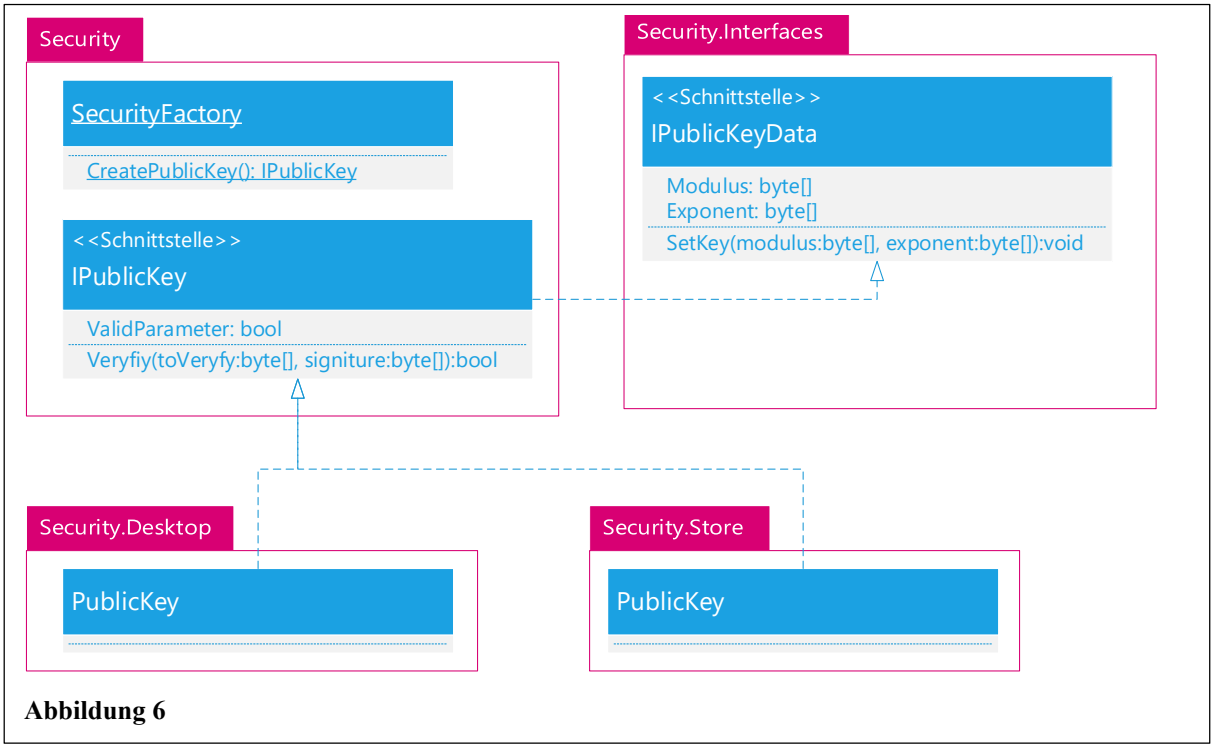


Abbildung 6

4.1.4 Network

Großteile des Netzwerkcodes befinden sich in diesen Projekten inklusive einer abstrakten Socketimplementierung. Dies war nötig, um das Projekt leichter auf andere Plattformen portieren zu können. Bereits vorhanden ist eine Implementierung für Desktop und Windows Store Anwendungen. Dabei verwenden beide zugrundeliegende APIs unterschiedliche Paradigmen. Während die .NET Implementierung ein Polling vorsieht, stellt die Windows Store API ein Push-Model via Events zur Verfügung. Die in diesem Projekt verwendete Abstraktion verwendet ebenfalls das Push-Modell.

Sockets werden über eine Factoryklasse erstellt, die eine Implementierung des Socketinterfaces zurückliefert. Dies erlaubt das Erstellen von Sockets unabhängig von der zugrundeliegenden Implementierung.

4.1.5 Security

Die Security-Projekte stellen Schnittstellen für die Erstellung und Nutzung von Public- und Private-Keys bereit. Zusätzlich erhält man Zugriff auf kryptographisch sichere Hashverfahren.

Wie bereits bei den Sockets sorgt eine Factory-Klasse für eine von der Implementierung unabhängige Verwendung.

Zum Signieren wird das RSA-Verfahren mit einer Schlüssellänge von 1024 verwendet. Als Hash-Algorithmus wird Sha256 eingesetzt. An einigen Stellen der Oberfläche wird ein MD5-Algorithmus verwendet, um Byte-Arrays lesbar darzustellen. Beispielsweise der zur Darstellung von öffentlichen Schlüsseln.

4.1.6 Game.Data

In diesem Projekt sind grundlegende Datentypen definiert, die von allen anderen Projekten verwendet werden sollen. Daher besitzt dieses Projekt auch kaum Abhängigkeiten von anderen Projekten. Andernfalls könnte es beim Erstellen der Projekte zu Problemen mit Zirkelabhängigkeiten kommen.

Da die Klassen wenig Logik enthalten sowie einfach und schnell erweiterbar sein sollten, wurden diese grundlegenden Datentypen über XML spezifiziert. Ein Generator, der vor dem Kompilieren ausgeführt wird, erstellt eine Quellcode-Datei, welche die in XML spezifizierten Klassen enthält.

Die so erstellten Klassen besitzen eingebaute Methoden zur Serialisierung bzw. Deserialisierung zu bzw. von XML. Zusätzlich wird eine XSD-Datei erzeugt, um zu überprüfen, ob eine XML-Datei deserialisiert werden kann.

Es ist ebenfalls festgelegt, wie die Daten signiert werden. Hierzu wurden die Klassen mithilfe einer partiellen Klassendatei um Code erweitert, der aus einem Objekt ein `IEnumerable<byte>` erstellt, das zum Signieren verwendet werden kann.

4.1.7 Logger

Das Logger-Projekt beinhaltet die Logik, die zum Loggen von Informationen und Fehlermeldungen verwendet wird. Dort kann eingestellt werden, welche Daten in eine Datei geschrieben werden, und welche auf der Debugger-Konsole ausgegeben werden.

4.1.8 Misc

Die Misc Projekte stellen eine Sammlung nützlicher Klassen und Methoden da, die in vielen Projekten Verwendung finden. Unter anderem befinden sich hier auch Klassen, welche die Serialisierung von Typen unterstützen.

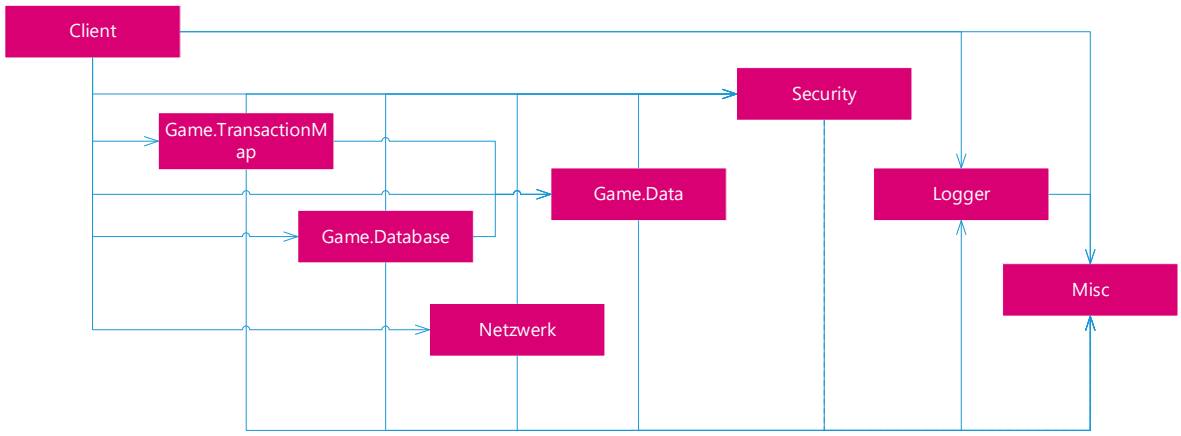


Abbildung 7

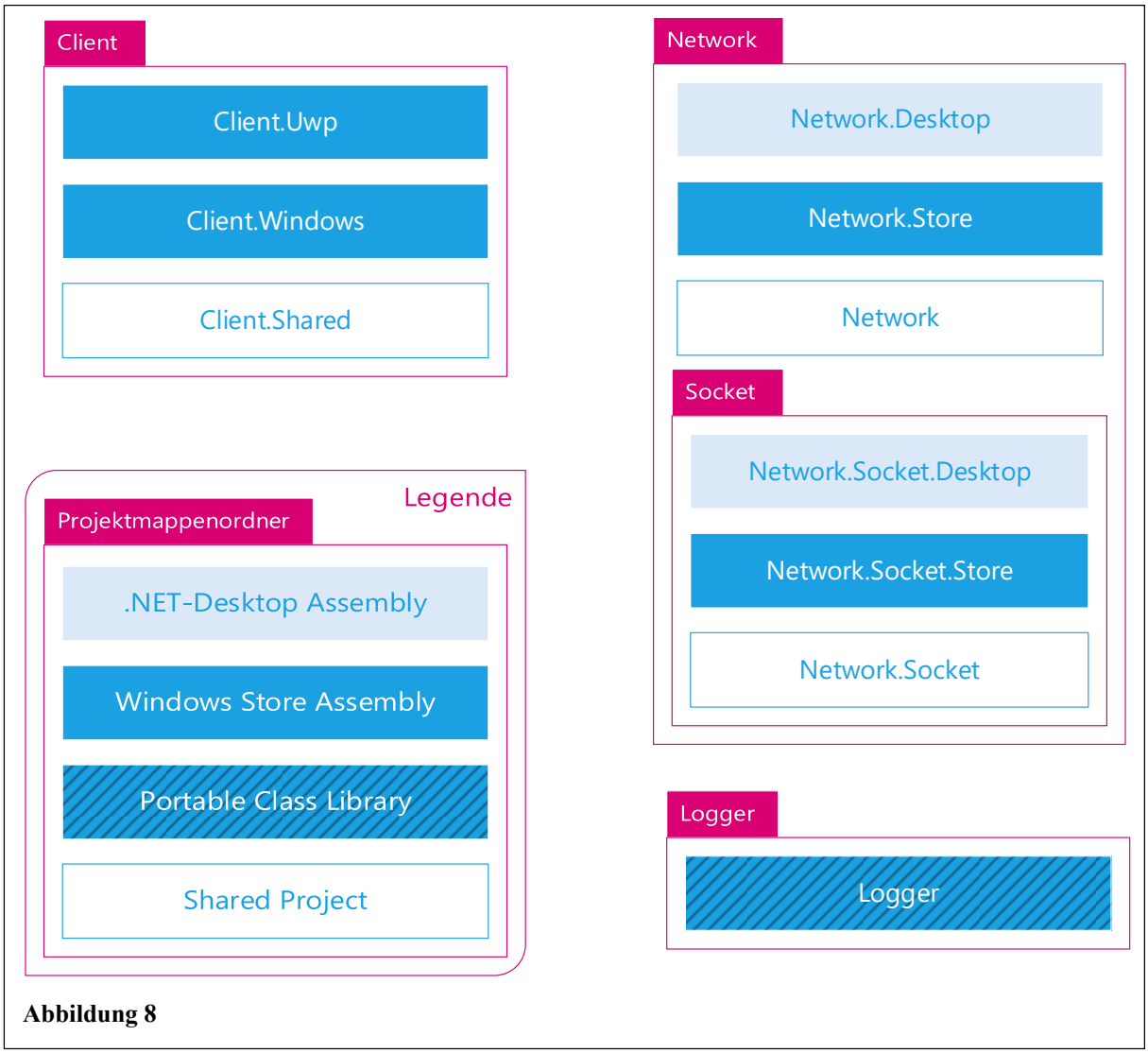


Abbildung 8

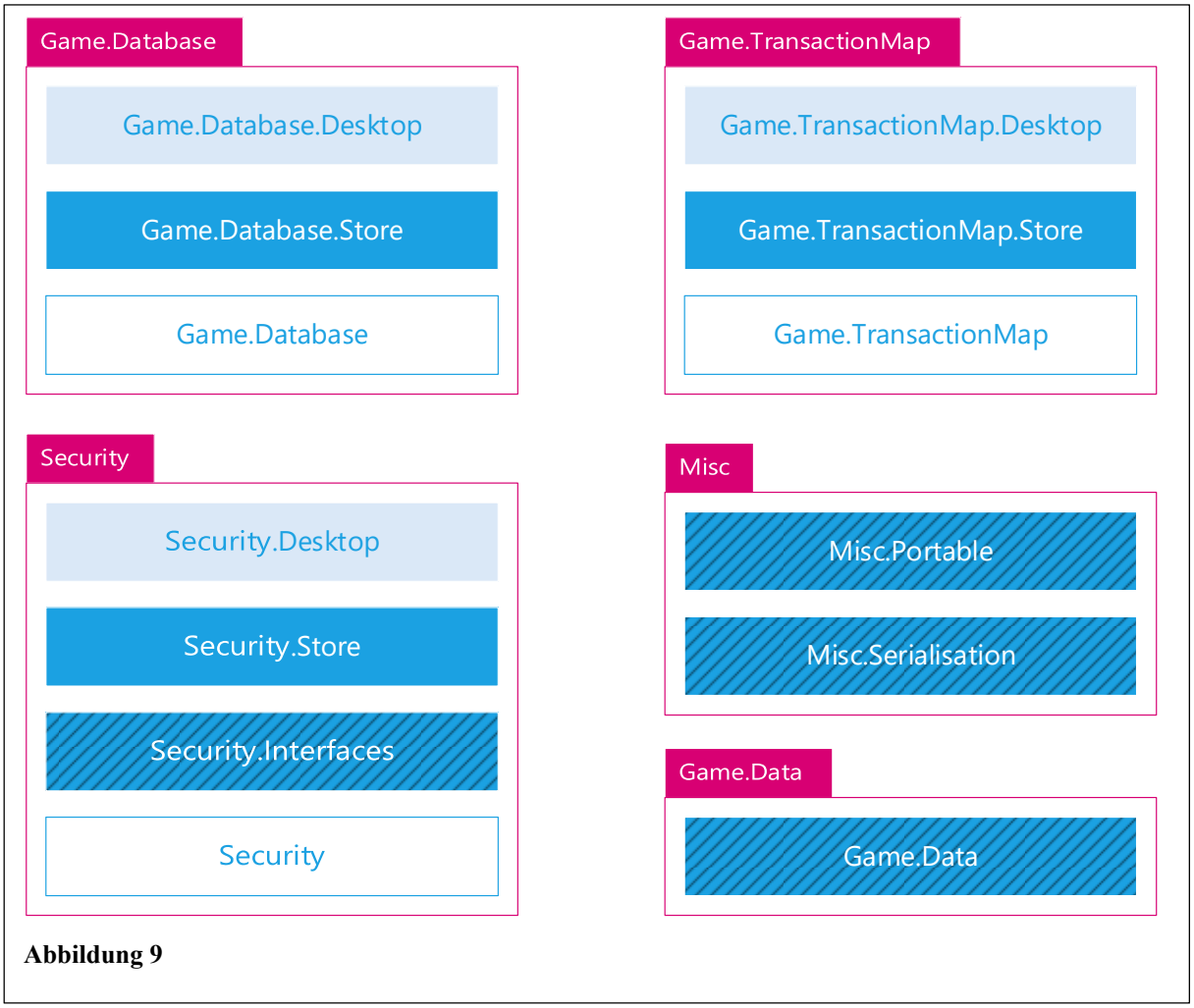

Game.Database
Game.Database.Desktop
Game.Database.Store
Game.Database
Game.TransactionMap
Game.TransactionMap.Desktop
Game.TransactionMap.Store
Game.TransactionMap
Security
Security.Desktop
Security.Store
Security.Interfaces
Security
Misc
Misc.Portable
Misc.Serialisation
Game.Data
Game.Data

Abbildung 9

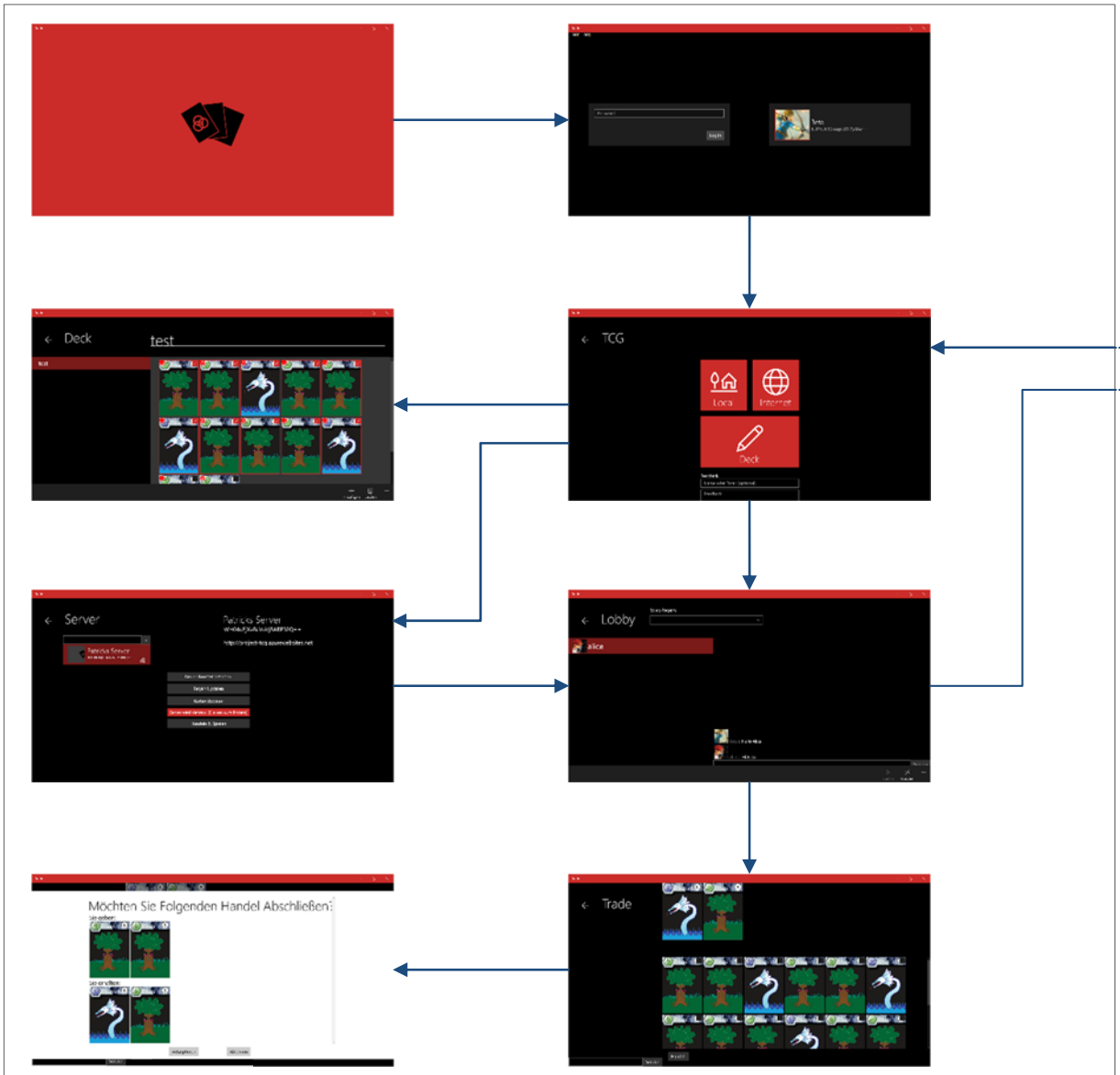


Abbildung 10

4.2 Benutzeroberfläche

Die Benutzeroberfläche wurde in XAML-Dateien definiert, unter Verwendung des Model-View-Viewmodel-Entwurfsmusters (Erläuterung 3). Dies ist der in WPF und Windows Store Applikationen verwendete Standard und bringt sehr gute Tool-Unterstützung durch Visual Studio mit sich.

Es existiert nur eine Oberfläche für Windows Store Applikationen. Obwohl sich der XAML-Code zwischen WPF und Windows Store Applikationen kaum unterscheidet, kann er nicht einfach übernommen werden, ohne einige Änderungen vorzunehmen.

Unter dem Namespace *Pages* befinden sich die XAML-Dateien der einzelnen Seiten, sogenannte Views. Der Namespace *Viewmodel* beinhaltet die verschiedenen Viewmodels zu den Views. Unter *Controls* findet man spezialisierte Steuerelemente, die ebenfalls Views darstellen.

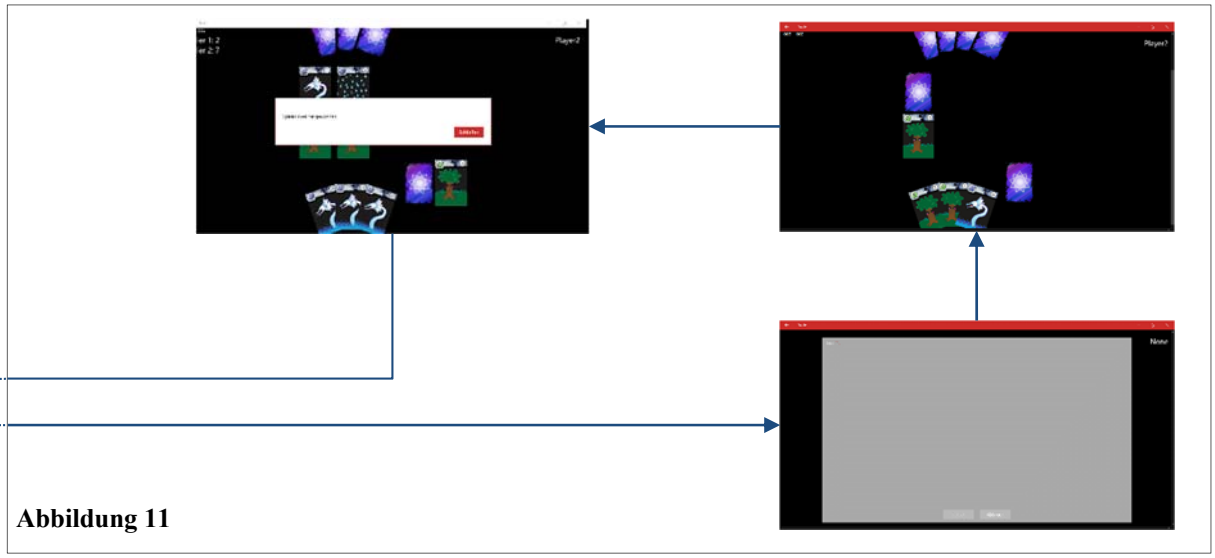


Abbildung 11

4.2.1 Menüführung

Die Steuerung des Clients erfolgt über einzelne Seiten, zwischen denen navigiert werden kann. Zentraler Anlaufpunkt ist das Hauptmenü. Nach dem Applikationsstart wird dieses über die Login-Seite erreicht. Auf dieser ist eine Passworteingabe erforderlich, damit der private Schlüssel, welcher die eigene Identität bestätigt sowie zum Handeln und Spielen benötigt wird, verschlüsselt persistiert werden kann.

Vom Hauptmenü aus erreicht man die Deckverwaltung, in der neue Kartendecks zusammengestellt werden können; die Seiten für die lokale Lobby und die Serververwaltung.

In der Serververwaltung können neue Server eingetragen werden, zu denen der Client Verbindungen herstellen kann. Von diesen Servern kann der Client neue Karten in Form von Boostern[3] erhalten, einer Lobby[4] des jeweiligen Servers beitreten und seine Kartendaten aktualisieren[5]. Zusätzlich kann eingestellt werden ob einem Server vertraut wird und somit die Karten die er erstellt hat akzeptiert werden. Sieht ein Spieler Karten eines nicht vertrauenswürdigen Servers, so werden diese markiert.

Die lokale Lobby, die aus dem Hauptmenü heraus erreicht werden kann, und die Lobby eines Servers sind Treffpunkt für Spieler. Über diese Seite können Spieler miteinander kommunizieren und ein Spiel oder Handel initiieren.

Auf der Handelsseite können mit einem anderen Spieler Karten getauscht werden. Auf der Spielplanseite wird das Spiel zwischen zwei Spielern ausgetragen. Abbildung 10 und Abbildung 11 zeigen die einzelnen Seiten und ihre Beziehungen untereinander. Abbildung 11 zeigt einen Screenshot bei der Deckauswahl vor dem Spiel, während des Spiels und nach dem Spiel.

Sobald das Spiel zu Ende ist, kehrt man zum Hauptmenü zurück.

[3] Kartenpackungen mit zufälligen Karten.
[4] Ein Chatraum in den sich Spieler zu Spielen oder zum Kartentausch verabreden können.
[5] Ein Server der Karten ausgegeben hat kann diese Aktualisieren. So können beispielsweise Rechtschreibfehler behoben werden.

MVVM

MVVM ist eine Variante von MVC. Jedoch wird auf den Controler zugunsten des sogenannten Viewmodels verzichtet. Zur losen Koppelung der View an das Viewmodel wird dabei ein Datenbindungsmechanismus verwendet. Das Viewmodel bereitet die Daten des Models für die View auf, sodass diese im Idealfall keine Logik enthalten muss.

Im Gegensatz zu dem Controler ist dem Viewmodel die View unbekannt. Stattdessen besitzt die View ein Viewmodel, und dieses hat einen Verweis auf das Model.

Die Datenbindung, in der Regel Binder genannt, übernimmt die Informationsübermittlung zwischen Viewmodel und View. Jedem Element der View welches Daten des Viewmodels anzeigen soll, wird an eine entsprechende Eigenschaft des Viewmodles gebunden. Sobald sich diese Eigenschaft des Viewmodels ändert, aktualisiert der Binder die View. Der Binder arbeitet bidirektional, das heißt er kann auch Änderungen in der View zurück in das Viewmodel übertragen. Dieses kann daraufhin auf die Änderungen reagieren und eventuell weitere Eigenschaften abändern oder das Model manipulieren

Erläuterung 3

4.2.2 Das Spielfeld

Genauer gehen wir an dieser Stelle auf die Oberfläche während des Spiels ein. Ein Spiel definiert das Aussehen der Oberfläche während des Spiels, das Spielfeld (siehe hierzu 4.3 Spiellogik). Dabei legt es verschiedene Bereiche fest, in denen Karten liegen können. Jeder Bereich besitzt eine Nummer, die Darstellungsart und die Zuordnung zum Spieler.

Diese Art der Positionierung erscheint zunächst sehr abstrakt, jedoch soll es möglich sein, das Spielfeld auf verschiedenen Bildschirmgrößen nutzbar darzustellen. Daher wurde von einer absoluten Positionierung abgesehen.

Als Referenz-Layout kann man den Spielbereich in Reihen aufteilen, dabei gibt die Nummer die Entfernung von der Spielfeldmitte an. Bereiche, die einem selbst zugeordnet sind, befinden

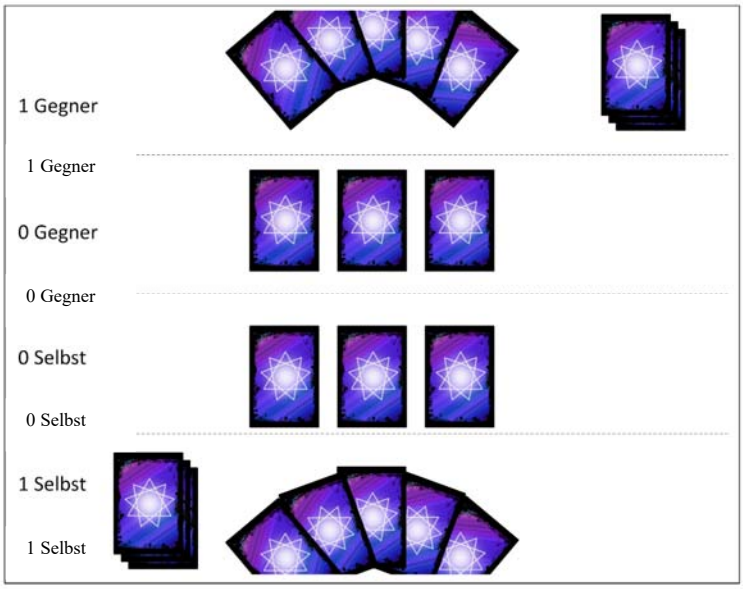


Abbildung 12

sich weiter unten und die Bereiche des Gegners oben. Um den Eindruck zu erwecken, einem anderen Spieler gegenüber zu sitzen, werden Bereiche, welche dieselbe Nummer besitzen, bei einem Selbst von links nach rechts angezeigt, bei seinem Gegenüber von rechts nach links. Abbildung 12 zeigt die Anordnung von sechs Bereichen nach diesem Schema. Dabei sind in Reihe 1 jeweils zwei Bereiche vorhanden. Diese wurden bei beiden Spielern in derselben Reihenfolge definiert, aber für den Gegner spiegelverkehrt angezeigt.

Ebenfalls zu erkennen sind die drei Arten, in der die Karten in einem Bereich angeordnet werden können. Unten links sind sie als Stapel angelegt, daneben als Hand und über diesem als Reihe.

Nach diesem Schema wurde das Layout im Client implementiert. Abbildung 13 zeigt eine mögliche Umsetzung für kleinere Bildschirmgrößen, die aufgrund niedrigerer Priorisierung nicht realisiert wurde. Hierbei sieht man nur einen Ausschnitt des Spielfelds. Zu einer Zeit wird jeweils nur ein Bereich angezeigt. Durch Wischgesten könnte man zwischen den verschiedenen Zeilen wechseln, wohingegen zwischen Bereichen derselben Zeile die unteren Schaltflächen

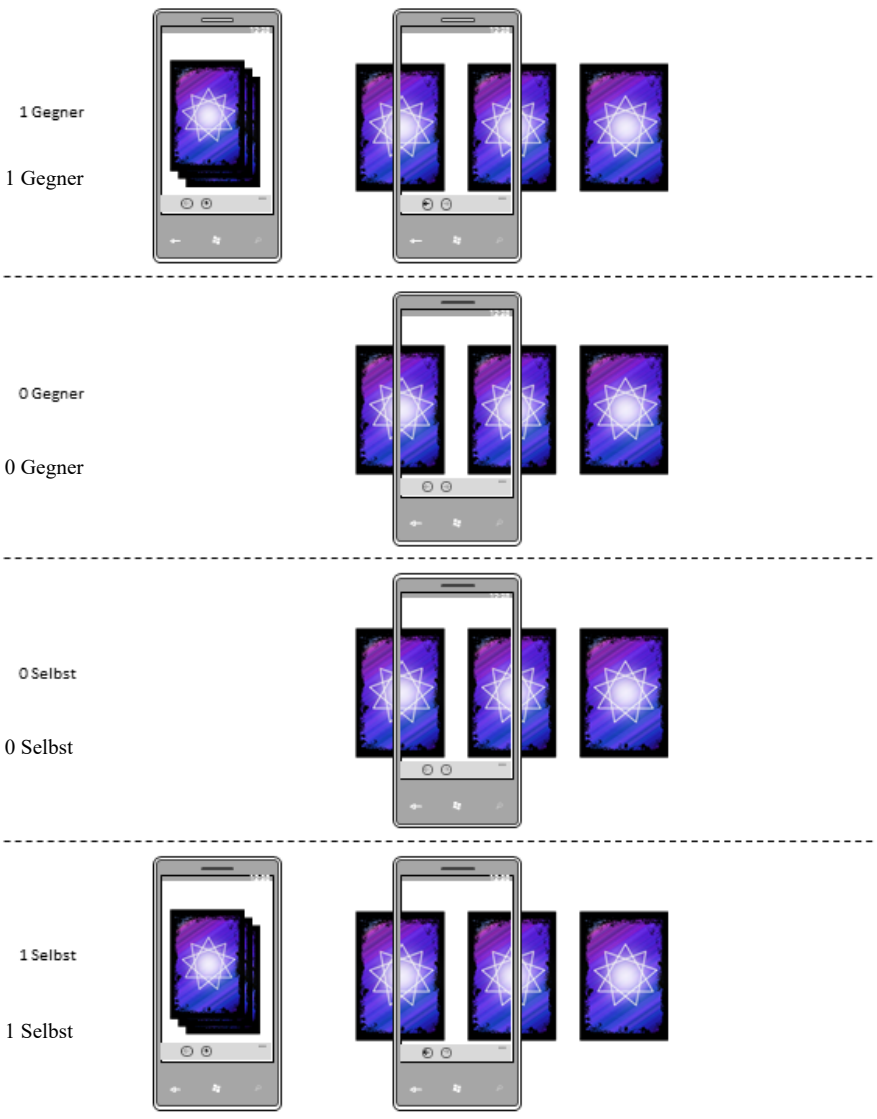


Abbildung 13

verwendet werden würden. Zugunsten der Übersichtlichkeit wurde auf die gekrümmte Darstellung des Handlayoutes verzichtet. Andernfalls könnten Karten am Rand, die eine Starke Neigung besitzen, bei der Darstellung abgeschnitten werden.

Zur Umsetzung wurde wie bei der restlichen Oberfläche auf Standard XAML-Elemente und auf die dazugehörigen Hilfsklassen zurückgegriffen. Da XAML umfangreiche Möglichkeiten bietet, View-Elemente auf einfache Art zu animieren, erweist sich es sich für ein Kartenspiel als ideal.

Die Oberfläche, die beim Spielen verwendet wird, setzt sich aus verschiedenen Klassen zusammen. Als Views existieren ein `GameControl`, welches die einzelnen Elemente wie textuelle Anzeigeelemente, Buttons und das Canvas der Karten organisiert. Die Karten werden durch das `CardControl` dargestellt.

Als Viewmodel existieren das `GameViewmodel`, `RegionViewmodel` und `CardViewmodel`. `Game`- und `RegionViewmodel` werden vom `GameControl` verwendet. Das `CardViewmodel` bereitet die Daten für das `CardControl` auf.

Als Modell wird die Klasse `GameEngine` verwendet (siehe hierzu Abschnitt 4.3 Spiellogik). Diese stellt die Rohdaten zur Verfügung. Beispielsweise wird in dieser gespeichert, welche Karten sich in welcher Region befinden und an welchem Index.

Während das `Card`- und `RegionViewmodel` Daten vorhalten wie zum Beispiel die Position einer Karte in x-, y-, z-Koordinaten oder das anzuzeigende Bild, befindet sich in dem `GameViewmodel` die Logik, die das Layout der Karten übernimmt.

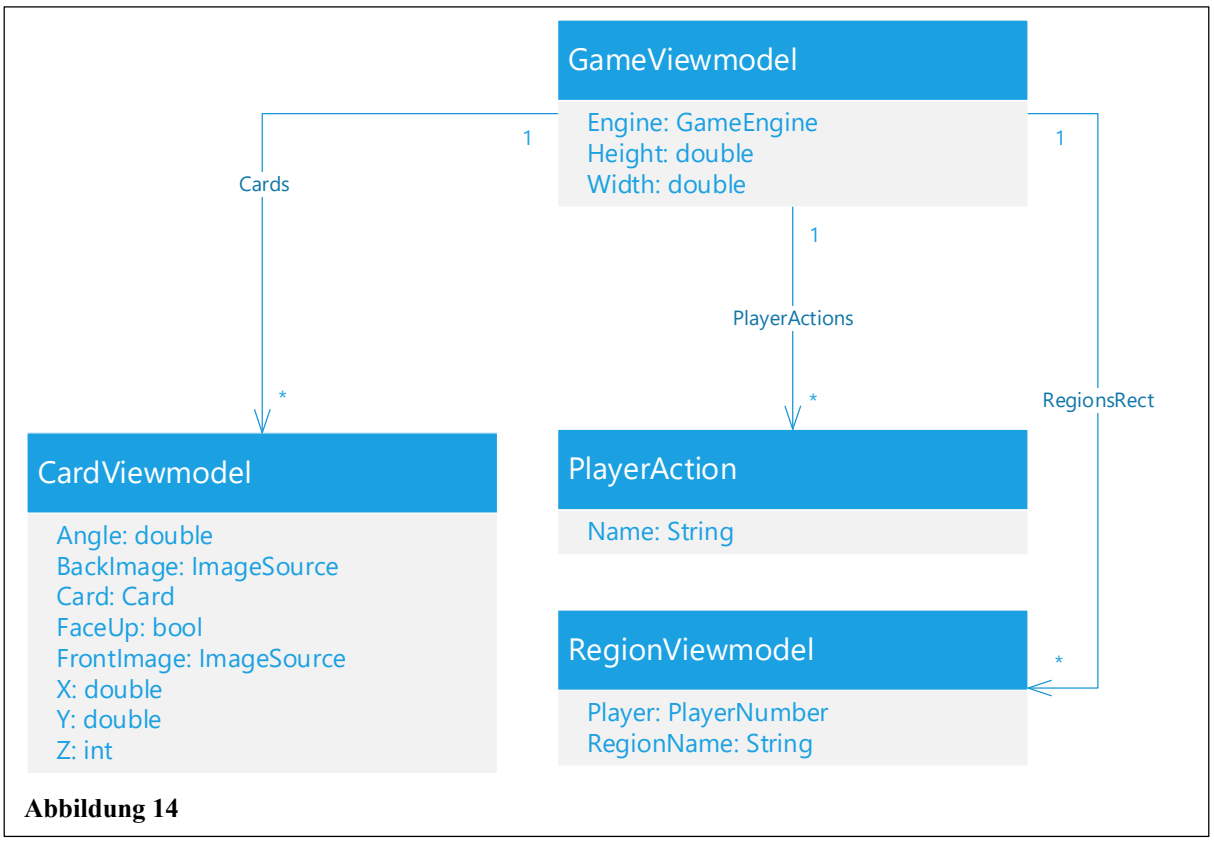


Abbildung 14

An dieser Stelle befindet sich ebenfalls der Code, der die Nutzereingaben mit der `GameEngine` Klasse verknüpft und die nötigen Konvertierungen durchführt. Abbildung 14 zeigt ein reduziertes Klassendiagramm der Viewmodels. Es wird deutlich, dass das `GameViewmodel` die anderen Viewmodel-Klassen verwaltet. Das Model des `GameViewmodels` befindet sich in der Eigenschaft `Engine`, aus dieser werden die Daten des Spieles geladen. Anhand dieser Daten werden die anderen Viewmodells erstellt. In den Height und Width Eigenschaften wird die gewünschte Spielfeldgröße festgehalten. Das `RegionViewmodel` kapselt Größe und Position der einzelnen Bereiche und das `CardViewmodel` den Zustand der einzelnen Karten.

Bei jeder Änderung, sei es die Höhe oder Breite des Spielfeldes oder die Position einer Karte, berechnet `GameViewmodel` das Layout und aktualisiert gegebenenfalls die Position der Karten.

Die im Klassendiagram der Abbildung 14 zu findende Klasse `PlayerAction` stellt Eingaben für den Nutzer dar. Jedes Mal, wenn das Spiel Nutzerinteraktion vorsieht, werden verschiedene Actions erzeugt. Für jede `PlayerAction` bekommt der Nutzer eine Schaltfläche angeboten, um diese Aktion auszulösen. Eine solche `PlayerAction` kann verwendet werden, um dem Spieler zu erlauben passen zu können. Neben den `PlayerActions` existieren noch `Card`- und `RegionActions`. Diese beziehen sich auf Bereiche, in denen Karten liegen bzw. einzelne Karten. Die `Card`- und `RegionActions` können das Ziehen einer Karte vom Talon oder das Ausspielen einer Karte von der Hand veranlassen. Umgesetzt sind sie als `Click`-Ereignisse auf den entsprechenden Views.

Die Animation der Karte, welche die Bewegung der Karte über das Spielfeld veranschaulicht, wird in der Klasse `CardControl` gesteuert. Sobald sich die Koordinaten im Viewmodel ändern, initiiert die View eine Animation der Positions-Eigenschaften der Bilder, welche die Karten repräsentieren.

4.3 Spiellogik

Da es nicht Ziel dieser Arbeit ist, ein explizites Spiel zu implementieren, sollte der Client, in Bezug auf die Spiellogik, einfach anzupassen sein. Im Idealfall ist dafür kein erneutes Kompilieren nötig. Weil Windows Store Apps sehr scharfe Sicherheitsrichtlinien besitzen, können nicht einfach DLLs nachgeladen werden. Somit fällt die Implementierung eines Plug-In-Modells via MEF (Managed Extension Framework) aus. Stattdessen haben wir die Spiellogik in TypeScript implementiert.

Die Gründe hierzu waren pragmatischer Natur. Da die Windows Store Plattform noch nicht lange existierte, war die Auswahl an möglichen Interpreter-Bibliotheken gering. Es gab jedoch schon sehr früh den in C# implementierte JavaScript Interpreter Jint[6]. Um sicherzustellen, dass in dem Spielskript bestimmte Funktionen vorhanden sind, nutzen wir zusätzlich TypeScript. Hierbei handelt es sich um eine Erweiterung von JavaScript, welche dieses um Klassen, Schnittstellen und Typsicherheit erweitert.

4.3.1 Skriptaufbau und Zustandsautomat

Das Spielskript definiert die Spielregeln. Das Minimum, das ein Spielskript erfüllen muss, ist die Existenz einer Klasse, welche die Schnittstelle `GameRules` implementiert. Jenes Interface

[6] https://github.com/sebastienros/jint

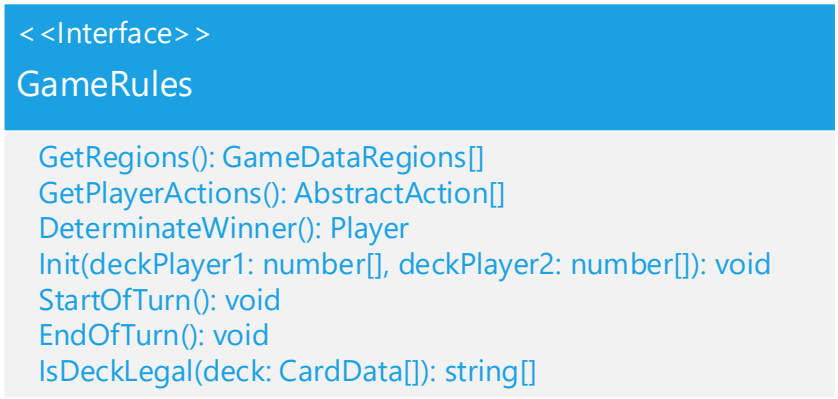


Abbildung 15

ist in der Game.d.ts-Datei definiert sowie einige Hilfsmethoden, die dem Skript erlauben, Einfluss auf die in C# implementierte Spielengine zu nehmen.

In Abbildung 15 ist die Definition der Schnittstelle dargestellt. Die Methode `GetRegions`() liefert Angaben über den gewünschten Aufbau des Spielfeldes (siehe hierzu Seite 38 Abschnitt 4.2.2 Das Spielfeld). `GetPlayerActions`() gibt Aufschluss darüber, welche Möglichkeiten dem Spieler aktuell zur Verfügung stehen. `DeterminateWinner`() liefert den Sieger der aktuellen Partie, sofern dieser bereits feststeht. Andernfalls wird ein Wert, zurückgegeben der symbolisiert, dass noch kein Sieger feststeht.

Zum Start einer Partie werden die Regeln mithilfe der `Init`-Methode initialisiert. Dazu gehören im Normalfall auch das Mischen und Austeilen der Karten. Die Karten der Spieler werden dieser Methode als Nummern übergeben. Die eindeutige Nummerierung der Karten wird von der Spielengine übernommen und dem Skript übermittelt. Die Umwandlung der Karten ist erforderlich, weil das verwendete Verschlüsslungsverfahren jedes Bit der zu verschlüsselnden Daten in zwei großen Zahlen codiert (siehe hierzu Abschnitt 3.2.2 Mental Card Game Seite 24). Für einzelne Zahlen stellt dies kein Problem dar, komplexere Datenstrukturen würden das Verfahren jedoch schnell an seine Grenzen bringen. Die beiden Methoden `StartOfTurn`() und `EndOfTurn`() werden zum Start bzw. zum Ende eines Zuges aufgerufen.

Die Spielengeine führt das Skript aus und verwaltet den aktuellen Zustand des Spiels. Dabei stellt diese Events bereit, um die Oberfläche über Änderungen zu informieren. Zusätzlich steuert sie die Kommunikation mit dem anderen Client. Des Weiteren initialisiert diese Engine die Skriptengine und das Skript selbst. Gesteuert wird die Gameengine durch einen Zustandsautomaten, der in Abbildung 16 dargestellt ist.

Die ersten beiden Zustände bestimmen unter anderem den Startspieler. Zuvor schicken sich beide Clients solange Nachrichten, bis der jeweils andere dies bestätigt. Dies ist garantiert, dass der Partner bereit ist Nachrichten zu verarbeiten, wenn die eigentliche Kommunikation durchgeführt wird. Die Nachrichtenübertragung geschieht zwar zuverlässig, aber der Client muss die empfangenen Nachrichten verarbeiten. Geschieht dies nicht, können Nachrichten verloren gehen (siehe Abschnitt 4.4 Kommunikation Seite 45).

Sind beide Clients bereit, senden sie sich eine zufällig generierte Zahl zu. Der Spieler, der die kleinere gewählt hat, wählt ungerade. Sollten beide Spieler dieselbe Zahl gewählt haben, wird der Vorgang wiederholt. Da keine der beiden Möglichkeiten Vorteile mit sich bringt, muss nicht überprüft werden, ob der andere Client seine Zahl wirklich zufällig gewählt hat. Bevor nun der

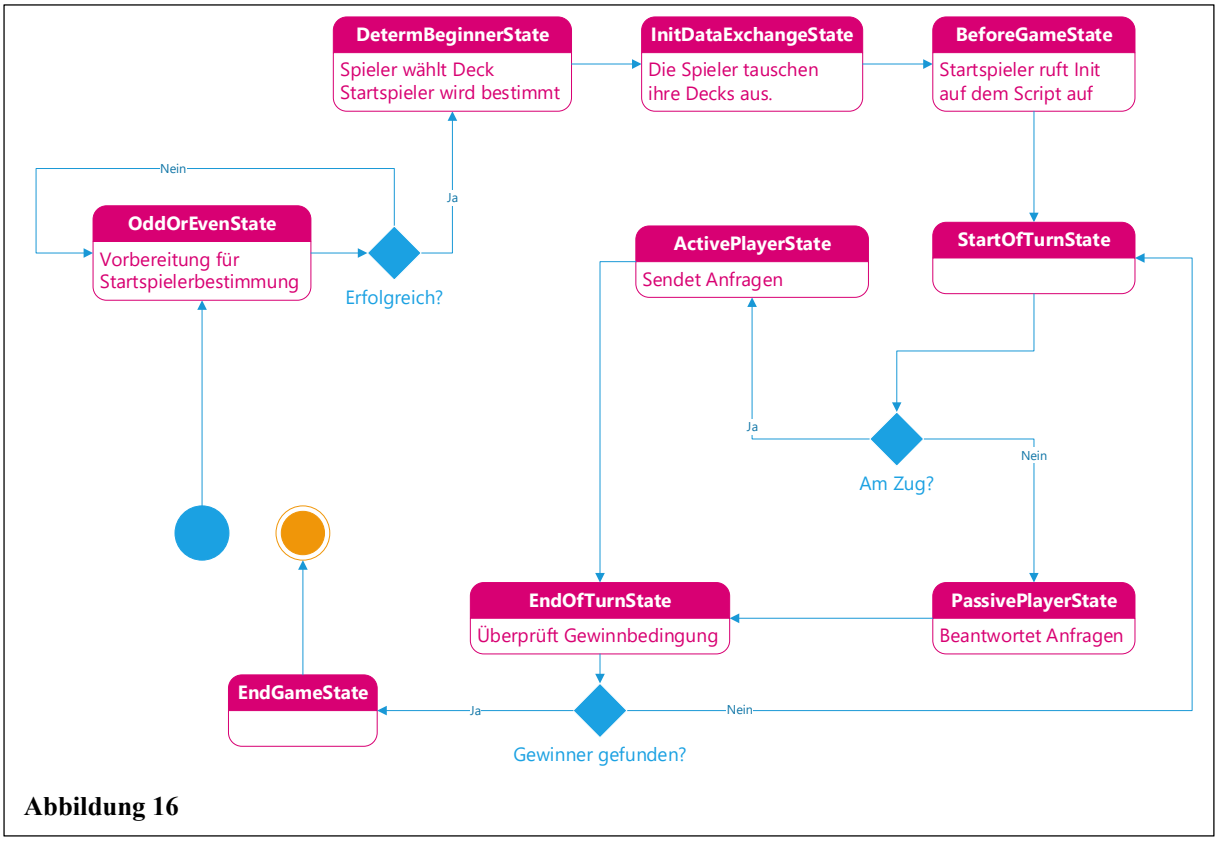


Abbildung 16

Startspieler bestimmt wird, werden die Spieler aufgefordert ein Deck auszuwählen. Diese Entscheidung sollte nicht davon abhängig sein, ob man Startspieler ist.

Die Spielengine löst dazu ein Ereignis aus, das der Benutzeroberfläche signalisiert, den Auswahldialog anzuzeigen. Jedes Deck des Spielers wird daraufhin mit der Skript-Methode `IsDeckValid` überprüft, um ihm eine mögliche Auswahl zu präsentieren.

Nachdem die Spieler ihre Auswahl getroffen haben, wird der Startspieler bestimmt. Es wird eine zufällige Zahl nach demselben im Spiel angewandten Mechanismus ermittelt (siehe Seite 44). Mit Zuordnung der Zahl zu gerade oder ungerade wird der Startspieler (Spieler Eins) bestimmt und der nächste Zustand kann aufgerufen werden.

Der `InitDataExchangeState` dient zum Abgleich der Karten-Daten. Beide Clients halten den Zustand des Spiels vor und manipulieren diesen synchron. Aus diesem Grund muss zu Beginn sichergestellt sein, dass beide denselben Startzustand besitzen. Bei allen folgenden Aktionen gibt es einen passiven und einen aktiven Client. Der Part des Aktiven wird von dem Spieler ausgeführt, der gerade am Zug ist. Nur der aktive Spieler führt Skripte aus, während der passive Spieler die Änderungen mimt, die der aktive Client durchführt. Aus diesem Grund wechselt Spieler Zwei über `BeforeGameState` und `StartOfTurnState` direkt in den `PassivePlayerState`. In diesem wartet er auf Input des ersten Spielers, um die Änderungen bei sich zu übernehmen.

Währenddessen führt der erste Spieler die Initialisierungsmethode des Spielskriptes aus. Anschließend erreicht er den `ActivePlayerState` über `StartOfTurnState`. Letzterer dient der Weiterleitung zum nächsten Zustand abhängig davon, ob der Spieler gerade in der Rolle des aktiven Spielers ist oder nicht. Zu beachten ist, dass in diesem Zustand nicht die `StartOfTurn`-Methode des Skriptes aufgerufen wird. Dies geschieht erst in dem Zustand

`ActivePlayerState`. Der Grund hierfür sind die unterschiedlichen Kontexte, die für die Namen verantwortlich sind. Der `StartOfTurnState` ist der erste Zustand, den der Zustandsautomat in der Runde betritt. Die `StartOfTurn`-Methode wird aus Sicht des Skriptschreibers als erste Methode jeder Runde aufgerufen, allerdings werden die Skriptmethoden, bis auf wenige Ausnahmen, alle von dem `ActivePLayerState` ausgeführt. Nachdem `StartOfTurn`() ausgeführt wurde, ruft die Gameengine `GetPlayerActions`() auf, um dem Spieler in der Oberfläche seine Möglichkeiten aufzuzeigen. Nachdem dieser seine Wahl getroffen hat, wird die entsprechende Aktion ausgeführt. Hierzu besitzt jede vom Skript erstellte Aktion einen Callback, der bestimmt, was beim Auswählen geschieht. Dies wird solange fortgeführt, bis das Skript das Ende der Runde signalisiert.

Bevor eine neue Runde beginnt, wechselt der Zustandsautomat in den `EndOfTurnState`. Hier überprüfen beide Clients lokal, ob es einen Sieger gibt. Sollte dies nicht der Fall sein, beginnt eine neue Runde und der aktive Spieler wechselt. Anderenfalls wird das Spiel durch den `EndGameState` beendet. Wichtig ist, dass die Überprüfung mittels `DeterminateWinner`() ohne Kommunikation mit dem Mitspieler geschieht. Da sich dieser ebenfalls im selben Zustand und nicht im `PassivePlayerState` befindet, erwartet er keine Nachrichten. Ein Fehlverhalten eines der Clients kann im schlimmsten Fall zum Beenden des Spiels führen und den Spieler zurück ins Startmenü versetzen.

Während des Spiels kann das Skript einige grundlegenden Operationen durchführen, die das Spiel steuern. Diese Aktionen sind:

- Karten von einem Bereich in einen anderen bewegen
- Karten mischen
- Verdeckte Karten aufdecken (Für einen Spieler oder alle)
- Nachrichten und Statustexte anzeigen
- Einen Zufallswert erzeugen
- Variablen setzen
- Den Zug beenden

Aus diesen Aktionen kann der Spieldesigner sein Spiel programmieren.

Es gibt einige Besonderheiten bei der Implementierung des Spielskriptes zu beachten. Dies ist durch die Art und Weise, wie die Kommunikation zwischen den Clients gelöst wurde, begründet. Die wechselnden Rollen zwischen aktiven und passivem Spieler tragen hier eine Mitschuld.

Auf das Anlegen von nicht lokalen Variablen sollte verzichtet werden, weil diese nur auf dem lokalen Client vorhanden sind und nicht zu dem Mitspieler übertragen werden. Somit besitzt der andere Client in der nächsten Runde keinen Zugriff auf diese Variablen. Stattdessen gibt es die Hilfsmethoden `Set`- und `GetNumber` sowie `Set`- und `GetString`. Beide erlauben, zu einem Schlüssel vom Typ `String` einen Wert zu speichern bzw. zu lesen. Diese Werte werden zu dem anderen Client überspielt und sind dort in der nächsten Runde verfügbar. Ebenso sollte nicht die Standard-JavaScript-Zufallsfunktion verwendet werden. Stattdessen existiert eine Hilfsfunktion, die den anderen Client bei der Erstellung des Wertes einbezieht. Somit kann kein Client den Zufallswert manipulieren. Das hierzu eingesetzte Verfahren basiert auf einem Protokoll von Manuel Blum (Blum 1981). Dabei wird von dem aktiven Spieler eine zufällige Zahl

generiert und deren Hashwert dem passivem übermittelt. Dieser wiederum übermittelt seinerseits eine ebenfalls zufällige Zahl. Zum Abschluss übermittelt der aktive Spieler seine erstellte Zahl. Die zufällig bestimmte Zahl stellt die Kontravalenz beider Zahlen da.

Aus der Skriptausführung ergibt sich ein Sicherheitsproblem, das aufgrund des zeitlichen Rahmens nicht mehr in dieser Arbeit gelöst werden konnte. Zwar wird das Spielskript kryptographisch vor Veränderungen geschützt, aber es findet während des Spiels keine Überprüfung statt, ob der Gegner das richtige Skript ausführt. Sollte der passive Spieler die Aufforderung erhalten, eine Karte aufzudecken, kommt er dieser nach. Dabei überprüft er nicht, ob dies die Regeln erlauben. Beispielsweise kann ein Spieler zu Beginn seine Karten im Talon aufdecken und weiß somit, welche Karten er als nächstes zieht. Dies würde der andere Spieler in der Oberfläche nicht angezeigt bekommen. Ein klarer Vorteil für den Betrüger.

4.3.2 Mental Card Game

Schindelhauer implementierte das in seinem Paper beschriebene Framework in C++. Für unser Projekt wurde eine C#-Implementierung geschrieben. Die Implementierung realisiert die Teilmenge der von Schindelhauer beschriebenen Funktionen, die für dieses Projekt benötigt werden.

4.4 Kommunikation

In diesem Abschnitt muss man zwischen der Kommunikation zweier Clients und der Kommunikation zwischen Client und Server unterscheiden. Die Kommunikation zum Server läuft über einen Webservice. Dieser stellt eine Schnittstellendefinition als WSDL-Datei zur Verfügung, gegen welche der Client implementiert wurde. Die WSDL-Datei wurde hierzu dem Projekt als sogenannter Dienstverweis hinzugefügt, woraufhin alle nötigen Klassen automatisch generiert wurden.

Zwischen zwei Clients verläuft die Kommunikation komplizierter. Dies liegt unter anderem daran, dass zwei Clients über mehrere Kanäle kommunizieren können. Einerseits können sie sich im selben Netzwerk befinden, andererseits können sie sich auch hinter NAT-Routern verbergen, getrennt durch das Internet. Sofern der Router es aus Sicherheitsgründen nicht unterbindet, stellt die Kommunikation im lokalem Netz kaum Hürden auf. Um eine direkte Verbindung über das Internet erstellen zu können muss etwas getrickst werden.

> **NAT**
>
> NAT steht für Network Address Translation, zu Deutsch Netzwerkadressenübersetzung. In Heimumgebungen wird dieses Verfahren eingesetzt, um mehreren Geräten im lokalen Netzwerk Zugriff zum Internet zu gestatten, obwohl nur eine öffentliche IP-Adresse existiert. Dabei wird jeder internen IP-Adressen-Port-Kombination, die eine Verbindung über das Internet herstellen möchte, ein externer Port zugewiesen. Jedes an diesem Port ankommende Paket wird anschließend zu der entsprechenden internen IP-Adresse an den entsprechenden Port weitergeleitet.
>
> Aus diesem Grund ist es nicht möglich, eine Verbindung zu einem Gerät hinter einem NAT aufzubauen. Die Weiterleitungsregel wird erst dann eingerichtet, wenn ein internes Gerät eine Verbindung aufbaut.

Erläuterung 4

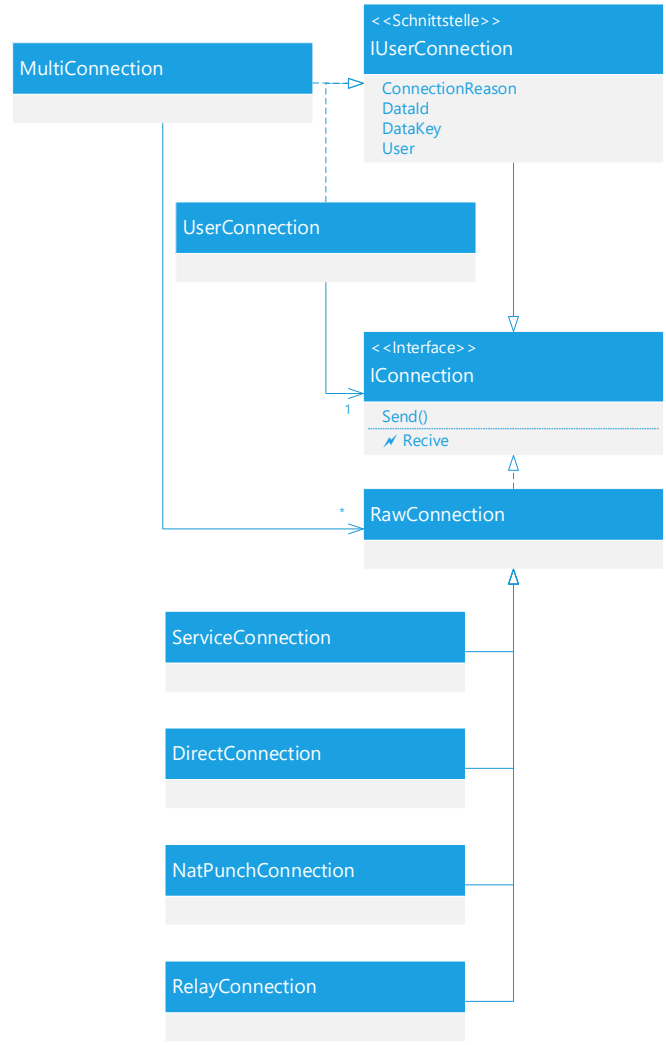


Abbildung 17

Die erste Möglichkeit stellt „NAT hole punching“ dar. Dabei besteht der erste Schritt darin, die öffentliche IP-Adresse herauszufinden. Hierzu kann STUN (Session Traversal Utilities for NAT) verwendet werden, welches in RFC 5389[7] dokumentiert ist. Dabei handelt es sich um ein einfaches UDP-Protokoll, das zur Ermittlung der öffentlichen IP-Adresse genutzt wird. Dieses beschreibt eine Anfrage an einen STUN-Server und dessen Antwort, in der die öffentliche IP-Adresse des Clients enthalten ist.

Um eine Verbindung zwischen zwei Clients aufzubauen, die sich beide hinter einem NAT-Router verbergen, wird ein zusätzlicher Kanal benötigt, über den diese Informationen ausgetauscht werden kann. Anderenfalls ist eine Übermittlung der entsprechend öffentlichen Adresse nicht möglich. Um die Verbindung aufzubauen, initiieren beide Clients eine Verbindung zu einem öffentlichen Server, der ihnen IP-Adresse und Port mitteilt, z.B. der bereits erwähnte STUN-Server. Diese Daten übermitteln die Clients mithilfe des zusätzlichen Kanals an ihren Partner. Anschließend leiten beide ihre Verbindung vom STUN-Sever auf die IP-Adressen-Port-Kombination um, die sie vom jeweils anderen erhalten haben. Da von beiden Clients Pakete zum jeweils anderen geschickt werden, nimmt der Router entsprechende Weiterleitungsregeln auf und die Verbindung ist hergestellt.

Der Nachteil dieser Methode ist, dass sie nicht mit jedem NAT-Router funktioniert. Je nachdem, wie dieser die Übersetzung von internen Adressen zu Ports handhabt, können Pakete mit demselben Ursprungsport und unterschiedlichen Zieladressen unterschiedliche externe Ports zugewiesen bekommen. Zudem gestaltet sich eine Umsetzung mittels TCP schwierig, da eine laufende TCP-Verbindung umgebogen werden muss. Hierfür benötigt man volle Kontrolle über die entsprechenden Sockets. Dieses ist in Windows Store Apps leider nicht gegeben. Daher bleibt nur die Umsetzung per UDP.

Die zweite Möglichkeit ist wesentlich einfacher umzusetzen. Dabei handelt es sich um ein UPnP-Protokoll, über welches man den Router nach seiner externen Adresse fragen kann. Ebenso können Port-Weiterleitungen eingerichtet werden. Für diese Methode muss der Router

[7] https://tools.ietf.org/html/rfc5389

UPnPIGD (Internet Gateway Device Standardized Device Control Protocol) unterstützen. Leider unterstützen gerade von manchem Provider zur Verfügung gestellte und vorgeschriebene Router dieses Protokoll nicht immer.

Die nächste Technik nutzt einen Server als Zwischenstelle. Dabei werden alle Nachrichten an einen Server geschickt, der diese an den Empfänger weiterleitet. Beide Clients bauen dabei eine Verbindung zum selben Server auf, der die Nachrichten übermittelt.

Vollständig umgesetzt wurde nur die letzte Methode, da diese unabhängig vom verwendeten Router funktioniert. Für die anderen beiden Methoden sind bereits grundlegende Funktionen und Klassen vorhanden, jedoch werden diese vom Client nicht genutzt. Abbildung 17 zeigt das zugehörige Klassendiagramm.

Um möglichst unabhängig von der jeweils genutzten Verbindung zu sein, wurden die beiden Interfaces `IConnection` und `IUserConnection` definiert. Beide stellen eine Verbindung zu einem anderen Client dar. Die `IUserConnection` erweitert `IConnection` um Metadaten zur Verbindung wie beispielsweise dem öffentlichen Schlüssel des Kommunikationspartners. Diese Information erlaubt signierte Nachrichten direkt zu überprüfen. `IConnection` wird von `RawConnection` implementiert, dies stellt die Basisklasse für alle weiteren Verbindungen dar. Von dieser leiten vier Klassen ab, die je eine Art von Verbindung darstellt. `ServiceConnection` stellt eine Verbindung über einen Server her. Die Nachrichten werden per Webserviceschnittstelle übertragen. Empfangen werden die Nachrichten durch das Abrufen von Nachrichten auf dem Webserver per Polling. `DirectConnection` ist eine direkte Verbindung, die üblicherweise nur im lokalen Netzwerk genutzt wird.

Die anderen beiden Klassen dienen als Platzhalter und sind nicht funktional implementiert. Dabei soll die `NatPunchConnection` eine Verbindung durch einen NAT-Router erlauben und so eine der oben genannten Methoden nutzen. `RelayConnection` soll eine Alternative zur `ServiceConnection` sein und ein schnelleres Weiterleitungsprotokoll verwenden als das von `ServiceConnection` genutzte.

Des Weiteren existiert die Klasse `UserConnection`, die eine `IConnection` kapselt und diese mit Metadaten anreichert. Darüber hinaus gibt es noch die Klasse `MultiConnection`. Diese kapselt mehrere `RawConnections` und stellt diese von außen als eine einzelne Verbindung dar. Eine neue `RawConnection` kann jederzeit hinzugefügt werden, muss jedoch nicht in der Lage sein, zu senden. Von den verfügbaren Kanälen wird einer ausgewählt, der senden kann. Dies ermöglicht Verbindungen zu wechseln, sobald eine bessere aufgebaut wurde. Dies geschieht für die Klassen, die diese Verbindung verwenden, transparent. Somit entsteht kein weiterer Aufwand bei der Implementierung.

Beispiel

Eine `MultiConnection` wird erzeugt und besitzt eine `ServiceConnection` und eine `NatPunchConnection`. Die `ServiceConnection` ist von Beginn an nutzbar, die `NatPunchConnection` hat noch keine Verbindung. Während der Client bereits die Kommunikation mit seinem Partner über den Server via `ServiceConnection` führt, werden zusätzlich Informationen, wie IP-Adresse, ausgetauscht, um eine direkte Verbindung der Clients zu ermöglichen. Sobald die `NatPunchConnection` eingerichtet ist, wird fortan diese genutzt und der Server kann entlastet werden.

Beispiel 10

4.4.1 ReliableDatagrammSocket

Es gibt eine ganze Menge an Klassen, die sich um die Kommunikation kümmern, die grundlegendsten haben wir bereits kennengelernt. Auf diesen aufbauend existieren weitere Klassen, welche die Kommunikation erleichtern sollen. Die Kommunikation basiert auf der Annahme, dass UDP als Transportprotokoll verwendet wird. Dies führt zu den folgenden Einschränkungen: Die Verbindung ist unzuverlässig und Datagramme können nicht beliebig viele Daten enthalten.

Zum Ausgleich wurde die Klasse `ReliableDatagrammSocket` entworfen. Diese nach dem Decorator-Entwurfsmuster gestaltete Klasse ergänzt einen Datagrammsocket um Zuverlässigkeit und übernimmt die Stückelung der Nutzdaten in mehrere Datagramme. Voraussetzung ist, dass die Gegenstelle ebenfalls diese Klasse verwendet, um die Datagramme wieder zusammenzusetzen.

Zur Umsetzung definiert die Klasse verschiedene Nachrichten.

- `StartComunication`
- `AcknowlageStartComunication`
- `Data`
- `ResendData`
- `EndComunication`
- `AcknowledgeEndComunication`
- `Error`
- `SingelPackage`

Jede Nachricht besitzt einen anderen Header. Jeder Header besitzt folgende drei Felder: Erstens ein Byte, welches als Magic Byte fungiert (Damit wird überprüft, ob dieses Protokoll verwendet wird. Sollte dies nicht stimmen, wird dieses Paket verworfen); Als zweites folgt ein Byte, das den Typ der Nachricht auf einen der oben erwähnten festlegt. Anschließend folgen zwei Bytes, welche die ID der Nachricht beinhalten. Weitere Felder im Header hängen vom Typ der Nachricht ab.

Sollen Daten versendet werden, sendet der Socket eine `StartCommunication`-Nachricht. Dies sendet er in regelmäßigen Abständen, bis er ein `AcknowlageStartCommunication` erhält. Anschließend sendet er die Data-Nachrichten, welche die Daten gestückelt enthalten. Sobald die letzte Data-Nachricht versendet wurde, wird die `EndCommunication`-Nachricht versendet. Bekommt man keine Antwort, wird die `EndCommunication`-Nachricht erneut gesendet. Sollten wir eine `ResendData`-Nachricht erhalten, übermitteln wir die gewünschten Daten erneut. Eine `Error`-Nachricht löst eine Ausnahme aus, die der Aufrufer der Senden-Methode behandeln sollte. Erhält der Socket die `AcknowledgeEndCommunication`-Nachricht, sind die Daten erfolgreich übertragen worden.

Die Methode dieses Sockets ist als asynchrone Methode implementiert, das heißt, sie besitzt als Rückgabewert nicht `void`, sondern Task. Dies erlaubt dem Aufrufer dieser Methode, auf die Beendigung der Aufgabe zu warten, ohne dass der Thread, in dem der Aufruf erfolgt, durch die Netzwerkaufrufe blockiert wird. Da eine Blockierung des GUI-Threads zum Einfrieren der Oberfläche führen würde, ist dies besonders wichtig für alle Methoden, die aus diesem Thread aufgerufen werden sollen.

Die Beendigung der Aufgabe bedeutet in diesem Fall, dass die Daten erfolgreich übermittelt wurden. Dies ist von Bedeutung, da der Socket nicht die Reihenfolge der ankommenden Nachrichten garantiert. Muss eine Reihenfolge der Datagramme eingehalten werden, so muss zwingend vor dem Senden eines zweiten Datagrammes die Übertragung des ersten abgewartet werden. Das erfolgreiche Übermitteln bedeutet nicht, dass die Nachricht verarbeitet wurde. Sollte der Empfänger bereits seinen Socket erstellt haben, aber sich noch nicht an dessen `MessageReceived`-Ereignis angemeldet haben, werden ankommende Nachrichten nicht verarbeitet, sondern verworfen. Dennoch erhält der Sender vom Socket die Bestätigung, dass die Nachricht angekommen ist.

Der weitere Nachrichtentyp `SingelPackage` wird zum Versenden kleiner Datenpakete verwendet. Diese enthalten die komplette Nutzlast und werden wie eine `EndComunication`-Nachricht behandelt. Somit werden diese mit einer `AcknowledgeEndCommunication`-Nachricht bestätigt.

4.4.2 Connectivity-Klassen

Im vorherigen Abschnitt wurde ein Socket beschrieben, dieser lag von der Abstraktionsschicht her unterhalb der Connection-Klassen, welche wir zu Beginn des Abschnittes 4.4 kennengelernt hatten. In diesem Abschnitt werden die Connectivity-Klassen behandelt. Diese nutzen die `IUserConnection`-Schnittstelle und liegen somit eine Abstraktionsschicht über den Klassen, welche diese implementieren.

Als Basisklasse dient `AbstractVerifiableConnectivity`<T>. Diese liefert die Grundfunktionalität der Connectivity-Klassen und wird von anderen Klassen für spezifische Zwecke erweitert. Die Klasse übernimmt folgende Aufgaben: Ausgehenden Nachrichten werden ein von der Connectivity abhängiges Magic-Byte vorangestellt. Zudem stellt die Connectivity ein ihren Konsumenten ein Pull-Modell zur Verfügung um Nachrichten zu konsumieren. Im gegenzug dazu müsste bei der zugrundeliegenden Connection ein Ereignis basiertes Modell genutzt werden.

Das erwähnte Magic-Byte erfüllt den Zweck eines Filters. Somit können verschiedene Connectivity-Objekte dieselbe Connection nutzen, ohne sich gegenseitig zu beeinflussen. Dies wird ausgenutzt, wenn zwei Spieler sich gegenseitig fehlende Karteninformationen während des Spielens oder des Tauschens übermitteln. Über eine Connection werden dazu Nachrichten von zwei verschiedenen Connectivitys versendet. Die Nachrichten welche Karteninformationen wie Bild oder Name enthalten, kommen beim Partner an der dementsprechenden Connectivity an wo sie erwachtet werden. Zur gleichen Zeit versendete Nachrichten über den nächsten Zug des Spielers werden an einer anderen Connectivity bearbeitet. Mithilfe des Magic-Bytes filtern die Connectivitys all jene Nachrichten heraus die nicht für sie bestimmt sind.

Als weitere Aufgabe serialisieren bzw. deserialisieren die Klassen die Nachricht zu einem Objekt vom Typ T. Hierzu müssen in abgeleiteten Klassen entsprechende abstrakte Methoden implementiert werden.

Klassen, die von `AbstractVerifiableConnectivity`<T> ableiten, signieren zusätzlich alle ausgehenden Nachrichten und überprüfen die Signatur aller eingehenden, um sicherzugehen, dass die Verbindung nicht manipuliert wurde.

Folgende Connectivity Klassen existieren:

- `InformationBrokerConnectivity`
  Dient zum Austausch fehlender Kartendaten wie z.B. Bilder.
- `GameConnectivity`
  Überträgt die Nachrichten der Spielengine.
- `MergeConnectivity`
  Überträgt die Daten zum Abgleich zweier Transaktionsgraphen beider Spieler. Dabei werden die unbekannten Transaktionen in den eigenen Graphen aufgenommen.
- `TradeConnectivity`
  Überträgt die Nachrichten während eines Handels.

4.5 Datenhaltungs-Design

In diesem Abschnitt widmen wir uns dem Design der Datenstrukturen, die alle wichtigen Daten vorhalten, die für die Karten und das Spiel benötigt werden.

4.5.1 Kartendaten

Abbildung 18 zeigt die Datenstrukturen, die für die Existenz der Karten verantwortlich sind.

Die zentrale Entität in diesem Diagramm stellt `PublicKey` dar. Dieses ist von Bedeutung, da es die Identität des Erstellers eines Datensatzes angibt (siehe Abschnitt 3.1.2 Kartensicherheit Seite 20).

Viele der Entitäten im Diagramm besitzen eine `Id`. Diese ist ein Universally Unique Identifier (UUID siehe Erläuterung 5, Seite 52). Dabei handelt es sich um eine 16-Byte-Zahl. Diese wird als, in fünf Gruppen unterteilte Hexadezimalzahl ausgeschrieben. Ein Beispiel für eine UUID wäre dff6a80e-4339-44cb-960d-293e4f8e7930. Die Nutzung solcher IDs erlaubt das Erstellen von Datensätzen auf verschiedenen Geräten, ohne dass es zu Namenskonflikten kommt.

Es sollte beachtet werden, dass UUIDs nicht vor bösartigen Teilnehmern geschützt sind. Dieser kann die UUID frei wählen und muss sie nicht zufällig generieren. Somit kann er Konflikte mutwillig herbeiführen. Aus diesem Grund muss bei einer potentiellen Übereinstimmung der Ersteller des Datensatzes verglichen werden. In Abbildung 18 ist diese Beziehung als `Creator` eingetragen.

Das zweite Attribut, das in vielen Entitäten vertreten ist, ist `Signature`. Dieses stellt die Unterschrift des Erstellers des Datensatzes dar. Somit lässt sich die Integrität aller anderen Eigenschaften des Datensatzes sicherstellen. Ein solcher kann von allen Teilnehmern weitergegeben werden, ohne dass diese den Datensatz manipulieren können. Damit ist sichergestellt, dass zwei Spieler den Daten des anderen vertrauen können, ohne eine dritte Instanz fragen zu müssen. Unterschrieben werden stets alle Einträge außer der Unterschrift selbst.

Die Karte selbst wird durch drei Entitäten beschrieben. Dabei stellt `CardInstance` die eigentliche Karte dar, die ein Spieler besitzt. `CardData` kapselt die Daten einer Karte, wie z.B. Name und Werte. `ImageData` stellt ein Bild dar, das im Kontext der Karte die visuelle Repräsentation ist.

Die Spieler besitzen verschiedene `CardInstances`. Diese bestehen aus einer eindeutigen ID-Ersteller-Kombination und der ID eines `CardData`-Satzes.

`CardData` besitzt eine eindeutige ID-Ersteller-Revisions-Kombination, einen Namen, eine Edition, die ID eines Bildes und eine Menge von Schlüssel-Wert-Paaren. Im Diagramm werden diese Paare als die Eigenschaft `Values` dargestellt. Das Attribut `Name` ist selbstbeschreibend, `Edition` erlaubt die thematische Gruppierung verschiedener Karten. Angelehnt ist diese Gruppierung an das gleichnamige Konzept der Auflagen bei reellen Sammelkartenspielen. `Values` beschreibt verschiedene Werte, die eine Karte besitzen kann und welche während des Spiels benötigt werden.

Die Aufteilung der Karte in `CardInstance` und `CardData` liegt in der einfachen Bereitstellung von Eratierungen einer Karte begründet. Würde diese Indirektion nicht existieren, würde einem Spieler ein `CardData`-Satz direkt gehören. Nach der Ausstellung einer solchen Karte kann festgestellt werden, dass die Karte fehlerhaft ist, sei dies ein Rechtschreibfehler oder Werte, die das Spielgleichgewicht stören. Würde die Indirektion nicht existieren, könnte dieser Fehler nicht auf einfache Weise berichtigt werden. Denn wie wir später bei Transaktionen sehen werden, sind diese ebenfalls kryptographich gesichert. Somit würden die Kartendaten in die Signatur der Transaktion einfließen und könnten nicht mehr geändert werden.

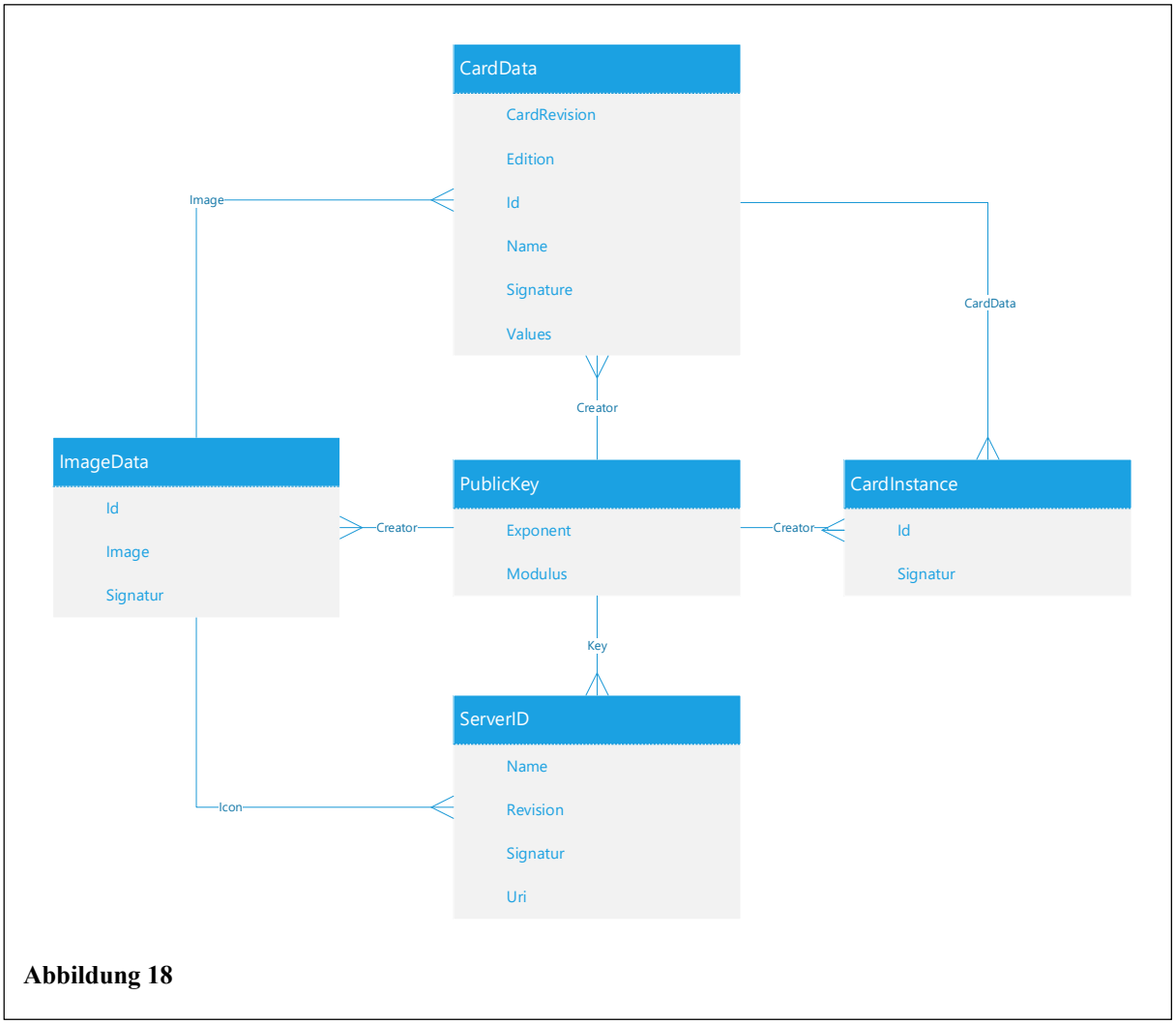


Abbildung 18

Durch die Separierung der Kartendaten von den Instanzen kann jedoch ein neuer Kartendatensatz erstellt werden, welcher den alten ersetzt, ohne dass dabei Transaktionen ungültig werden. An dieser Stelle wird auch das Revisionsattribut genutzt. Diese erlaubt es festzustellen, welcher Datensatz aktueller ist.

Da nicht nur Karten Bilder verwenden, haben wir uns auch an dieser Stelle mit einer Indirektion beholfen und für Bilddaten den `ImageData`-Satz erstellt. Diese besitzen eine eindeutige ID-Ersteller-Kombination und werden mittels Signatur vor Manipulation geschützt.

„All problems in computer science can be solved by another level of indirection“
David Wheeler

„…except for the problem of too many layers of indirection”
Kevlin Henney

Die letzte in Abbildung 18 beschriebene Entität hängt nicht direkt mit den Karten zusammen, aber beinhaltet für den Nutzer wichtige Informationen. Dabei handelt es sich um die von einem Erzeuger selbst ausgestellte Identität. Diese besteht aus einen Namen, einem Icon und der URI, unter der dieser zu finden sein sollte.

Das Netzwerk benötigt keine direkte Verbindung zu einem Server, da Informationen von einem Nutzer zum nächsten propagiert werden. Aufgrund der sporadischen Verbindungen zwischen Nutzern kann dies einige Zeit dauern. Im Falle, dass zu einer Karte der Erzeuger eine Internetadresse angegeben hat und der Client Online ist, kann direkt an der Quelle nach fehlenden oder neueren Datensätzen geschaut werden. Da sich URIs ändern können, wird das gleiche Revisionsmechanismus wie bei Kartendaten genutzt.

UUID

Diese IDs eignen sich hervorragend eindeutige IDs in verteilten Systemen zu erstellen. Sofern zur Erstellung ein gleichverteilter Zufallszahlengenerator verwendet wird.

Auch wenn eine Kollision bei der Erstellung zweier UUIDs nicht ausgeschlossen werden kann, ist dies doch im Normalfall vernachlässigbar. Um dies besser begreifbar zu machen:

Erstellt man über ein Jahr einige 10 Billionen UUIDs, so ist die Wahrscheinlichkeit, dass es zu einer Kollision kommt, ungefähr so hoch wie die Wahrscheinlichkeit in diesem Zeitraum von einem Meteoriten getroffen zu werden.

Anders ausgedrückt, um mit einer Wahrscheinlichkeit von 50% eine Kollision zu erhalten, müssen für die nächsten 100 Jahre jede Sekunde eine Milliarde UUIDs erzeugt werden.

(Universally Unique Identifier - Wikipedia 2016)

Erläuterung 5

4.5.2 Transaktionsdaten

Die Daten der Transaktionen wurden in einer eigenen Datenbank isoliert. Dies soll den Quellcode übersichtlicher machen, da beide nur geringe Überschneidungspunkte haben, jedoch einen relativ umfangreichen Code vorweisen.

Abbildung 19 zeigt die wichtigsten Entitäten dieser Datenbank. Dabei stellt `Transaction` die in Abschnitt 3.1.1 Handelsnetzwerk (Seite 12) vorgestellte Transaktion dar. Als zweite Entität existiert noch `Transfer`. Dieser beinhaltet Informationen zu den einzelnen Karten, die in einer zugehörigen `Transaction` gehandelt werden.

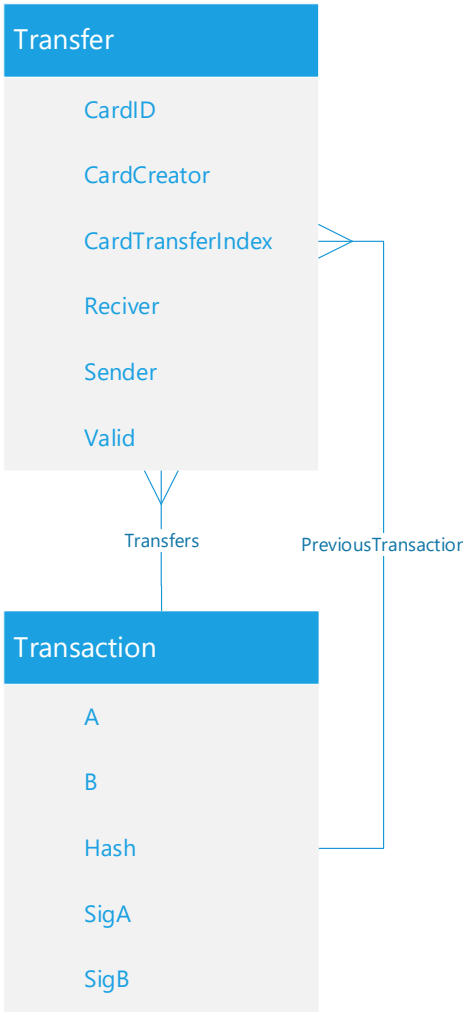


Abbildung 19

Um eine Karte einem `Transfer` zuzuordnen, werden die Eigenschaften `CardID` und `CardCreator` des `Transfer`s genutzt. Diese beiden Eigenschaften identifizieren eindeutig eine einzige Karte. Der `Transfer` speichert aber noch weitere Werte zu jeder Karte ab. So liefert der `CardTransferIndex` die Häufigkeit, mit der die Karte bereits gehandelt wurde. Der Name `Transfer` wurde gewählt, weil diese Entität die Übertragung des Eigentums einer Karte von einem Spieler auf einen anderen symbolisiert. Welche Parteien dies sind, wird in den Eigenschaften `Sender` und `Reciver` festgehalten. Dabei steht der öffentliche Schlüssel des ursprünglichen Eigentümers in `Sender` und der öffentliche Schlüssel des neuen Besitzers der Karte in `Reciver`. Die letzte wichtige Eigenschaft des `Transfer`s stellt einen Verweis auf die letzte Transaktion dar, in welcher die Karte gehandelt wurde. In dieser Transaktion muss der aktuelle `Sender` auch Eigentümer der Karte geworden sein. Es wird noch eine zusätzliche Eigenschaft `Valid` in jedem Transfer gespeichert. Diese gibt an, ob dieser `Transfer` einen Fehler verursacht oder auf anderen Transaktionen aufbaut, welche einen Fehler verursachen. Prinzipiell ist diese Information redundant, da sie aus der Gesamtmenge der Transaktionen berechnet werden kann. Da dies jedoch eine aufwendigere Operation ist, wird das Ergebnis in der Eigenschaft gespeichert.

Eine `Transaction` enthält eine Menge von `Transfer`s. Zusätzlich zu diesen werden noch die Tauschpartner, die an dieser Transaktion beteiligt sind, in den Eigenschaften `A` und `B` festgehalten. Die `Transfers` dürfen keinen `Transfer` enthalten, der in den Eigenschaften `Sender` und `Reciver` andere Spieler gespeichert hat, als in den Eigenschaften `A` und `B` hinterlegt sind. Zur Sicherstellung der kryptographischen Belastbarkeit dienen die Attribute `SigA`, `SigB` und `Hash`. In `SigA` befindet sich die Signatur des Spielers, der in `A` referenziert wird. Sinngemäß gilt dies auch für `SigB`. Die Eigenschaft `Hash` erlaubt es, eine Transaktion zu referenzieren und dabei sicher zu sein, dass diese nicht manipuliert wurde. Sie wird beispielsweise in der Eigenschaft `PreviousTransactionHash` von `Transfer` verwendet. Die drei Eigenschaften `SigA`, `SigB` und `Hash` berechnen sich aus den `Transfers` der Transaktion und der Eigenschaft `A` und `B`. Es werden alle Attribute der Transfers mit Ausnahme von `Valid` zur Berechnung einbezogen. Dies ist nötig, da `Valid` ein berechneter Wert ist, welcher sich beim Hinzufügen von neuen Transaktionen ändern kann. Dies darf jedoch keinen Einfluss auf die Gültigkeit der Signaturen haben.

Ein wichtiger Punkt ist, dass diese Datenstruktur mit anderen ausgetauscht werden soll. Dabei sollen möglichst wenige Daten über das Netz übertragen werden, um den Abgleich zu beschleunigen. Um dies zu erreichen, wird Algorithmus 2 genutzt:

Somit müssen neben den fehlenden Transaktionen zusätzlich die Transaktionen der Mengen $head_B$, $head'_A$ und $head''_A$ übermittelt werden.

Merge-Algorithmus

Bob möchte seinen Graphen seiner Transaktionen um die Informationen aus Alice Graphen erweitern. Alice besitzt den Graphen A und Bob den Graphen B. Die Menge aller Transaktionen, die keine Nachfolgetransaktionen in einem Graphen G besitzen, nennen wir $heads_G$.

1. Bob übermittelt Alice $head_B$. Alice erstellt $head'_B = head_B \,/\, head_A$ und $head'_A = head_A \,/\, head_B$.
2. Alice übermittelt $head'_A$ an Bob und fragt, welche Transaktionen er nicht kennt. Bob übermittelt Alice $head''_A = \{t \in head'_A | \, t \notin B\}$.
3. Alice erzeugt eine Queue q und fügt alle Transaktionen aus $head''_A$ hinzu.
4. Für jedes $t \in q$:
   a. Übermittle t an Bob und finde die Menge v der Vorgängertransaktionen zu t in A
   b. Für alle $vt \in v$, für die gilt $vt \notin head'_B$ füge vt in q ein.
5. Bob besitzt nun $A \cup B$.

Algorithmus 2

4.5.3 Weitere Daten

Es existieren weitere Datensätze, die in der Datenbank der Karten persistiert werden. Dabei handelt es sich um die unterschiedlichen Spielregeln, nach denen die Nutzer spielen können, sowie die Zusammenstellung der Decks.

Ein Regelsatz besitzt eine eindeutige ID-Erzeuger-Revision-Kombination. Zudem enthält diese Entität einen Namen, ein Skript, das die Implementierung der Regeln darstellt, und eine Signatur. Diese Signatur schützt den Datensatz vor unlauteren Manipulationen.

4.5.4 Umsetzung

Zur Realisierung wurde eine eingebettete SQLite[8] Datenbank verwendet. Als OR-Mapper fand sqlite-net[9] Anwendung. Durch die Verwendung dieser Frameworks müssen einige Einschränkungen bei der Umsetzung beachtet werden.

Zum einen werden keine Fremdschlüssel unterstützt, das heißt der OR-Mapper kann nicht automatisiert Objektgraphen serialisieren. Es können dementsprechend keine Eigenschaften eines Objektes serialisiert werden, die nicht von der Datenbank als Datentyp einer Tabellenspalte

[8] http://sqlite.org/
[9] https://github.com/praeclarum/sqlite-net

unterstützt werden. Des Weiteren stehen keine zusammengesetzten Primärschlüssel zur Verfügung.

Da zusammengesetzte Primärschlüssel nicht verwendet werden können, wird stattdessen ein einfacher `int`-Wert für alle Tabellen als Primärschlüssel verwendet. Dieser wird von der Datenbank automatisch vergeben. Um sicherzustellen, dass keine ungültigen Zeilen eingetragen werden, wird das Einfügen von einer statischen Methode übernommen. Diese überprüft mittels Select, ob bereits ein Eintrag mit den entsprechenden Werten vorhanden ist, und verweigert das Einfügen im Zweifelsfall. Der Rückgabewert der Methode gibt darüber Auskunft. Zusätzlich wird in diesen Methoden für Klassen, die eine Revision besitzen, überprüft, ob es nicht bereits neuere Einträge gibt. In diesem Fall schlägt das Einfügen fehl. Außerdem werden ältere, nicht mehr benötigte Revisionen beim Einfügen einer neuen gelöscht.

Als Ersatz für Fremdschlüssel wurden entsprechende Attribute doppelt implementiert, einmal das eigentliche Attribut und eines, das den Namenssuffix FK trägt. Das eigentliche Attribut ist vom Typ `Task`<T>. Dabei wird beim Aufrufen der Eigenschaft ein Select von der Datenbank ausgeführt, um den eigentlichen Typ zu deserialisieren. Dies ist auch der Grund, warum das gewünschte Objekt in einem `Task`<T> gekapselt wird. Eine solche Operation könnte potentiell längerfristig blockieren. Währenddessen ist das zusätzliche Attribut vom Typ `int` und beinhaltet den entsprechenden Fremdschlüssel, mit dem die Abfrage durchgeführt werden kann.

Von außerhalb des Projektes wird die Datenbank der Transaktionen über die statische Klasse Graph manipuliert. Die statische Klasse Database erfüllt denselben Zweck für alle anderen Datenobjekte.

4.6 Erstellung der Binärdaten

Zur Erstellung der Binärdaten ist ein C#-Compiler nötig, welcher C# 6.0 unterstützt und zur Ausführung von Buildscripten sowie zum Auflösen von Abhängigkeiten MSBuild und NuGet. Zusätzlich werden die Standardbibliotheken zum Bau von Windows 10 Apps genauso wie SQLite für die entsprechende Zielplattform benötigt. Alle weiteren Abhängigkeiten werden von NuGet aufgelöst.

Als die Arbeit verfasst wurde, benötigte man zum Erfüllen dieser Voraussetzungen Visual Studio 2015 auf einem Windows 10 System. Zusätzlich muss in der Visual Studio Installation ein Plug-In für SQLite installiert werden.

Nach der Installation des SQLite Plug-Ins müssen die vorhandenen Projekte angepasst werden. Die Versionsnummer der vom Projekt verwendeten SQLite Bibliothek muss mit der Version, die durch das installierte Plug-In bereitgestellt wird, übereinstimmen. Da das Plug-In System von Visual Studio nicht vorsieht ältere Versionen eines Plug-Ins zu installieren, stimmt die Version des Plug-Ins im Normallfall nicht mit der von den Projekten erwarteten Version überein.[10]

Der Benötigte Quellcode kann unter *https://github.com/OTCGS/TCG-Client* gefunden werden.

[10] Anm.d.A .:Würde SQLite NuGet zum Bereitstellen verwenden könnte man sich an dieser Stelle viel Arbeit sparen.

5 Spieldesign

Zum Testen des Clients haben wir ein Beispielspiel implementiert. Dieses baut auf dem Prinzip auf, dass die stärkste Karte gewinnt. Um das Spiel interessanter zu gestalten, fließt zusätzlich noch das asymmetrische Spielprinzip von Schere-Stein-Papier ein. Dabei wird jeder Karte eine Grundstärke und eines von fünf Elementen zugewiesen.

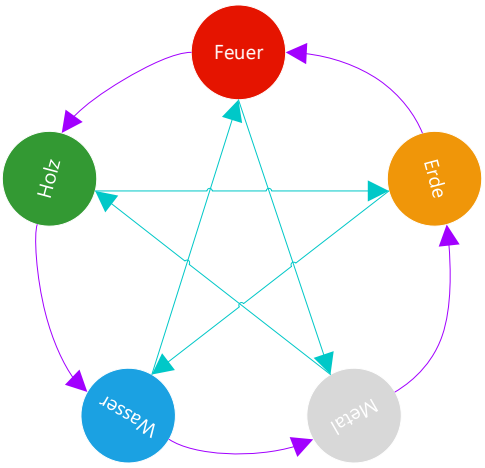


Abbildung 20

Jeder Spieler startet das Spiel mit drei Karten. Ist ein Spieler an der Reihe, zieht er zunächst eine Karte und spielt eine verdeckt aus, womit er seinen Zug beendet. Die Karten eines Spielers werden in eine Reihe gelegt.

Haben beide Spieler jeweils drei Karten ausgespielt, werden die ersten beiden Karten jedes Spielers aufgedeckt. Dabei wird die Stärke der ersten Karten verglichen und der Spieler mit der stärkeren Karte bekommt einen Punkt. Anschließend wird die erste Karte entfernt und die Spieler spielen erneut eine verdeckte Karte aus, sodass wieder drei Karten pro Spieler liegen. Die Auswertung wird wiederholt. Dies geschieht so lange, bis ein Spieler 7 Punkte erhalten hat.

Da die Stärke einer Karte alleine nicht sonderlich interessant wäre, berechnet sich die effektive Stärke aus der Grundstärke der Karte und einem Bonus in Abhängigkeit der um sie herumliegenden Karten. Dabei existieren zwei unterschiedliche Boni, die beide von den Elementen der ausgespielten Karte abhängen. Ein Element kann einen leichten oder starken Vorteil gegen ein anderes Element besitzen.

Sollte eine Karte einer anderen gegenüberliegen, gegen den sie einen starken Vorteil besitzt, so erhält die Karte einen Bonus in Höhe ihrer eigenen Stärke. Sie verdoppelt also ihren Wert.

Abbildung 21

Die andere Möglichkeit einen Bonus zu erhalten, hängt von der benachbarten Karte des Spielers ab. Somit von der Karte, die nach der ersten gespielt wurde. Hierfür müssen zwei Bedingungen erfüllt sein: Die benachbarte Karte muss eine höhere Grundstärke haben als die erste und die benachbarte Karte hat einen Vorteil gegen die gegnerische Karte. Ist es ein leichter Vorteil, so beträgt der Bonus 1½ der Grundstärke der benachbarten Karte. Sollte die benachbarte Karte sogar einen starken Vorteil besitzen, so darf stattdessen der doppelte Wert anstelle des 1½-Fachen hinzuaddiert werden.

Welches Element gegen welches andere einen Vorteil besitzt, kann Abbildung 20 entnommen werden. Diese zeigt die an den Daoismus angelehnten Beziehungen zwischen den Elementen. Die Pfeile geben an, welches Element gegen welches effektiv ist. Zu unterscheiden sind die türkisen von den violetten Pfeilen. Letztere stellen einen stärkeren Vorteil gegenüber dem Element dar.

Abbildung 21 zeigt eine Beispielkarte, welche die Stärke 8 besitzt (obere rechte Ecke). Das Symbol in der linken Ecke repräsentiert das Element Feuer.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein prototypischer Client eines dezentralen Sammelkartenspiels implementiert. Von den gesetzten Zielen konnten alle bis auf eines umgesetzt werden. Das Ziel z_2, den Besitz einer Karte sicher festzustellen, konnte nicht zufriedenstellend erfüllt werden.

Dies erlaubt es jedoch den Spielern, komplett ohne Verbindung zu einem Netzwerk zu spielen und dabei ein gewisses Mindestmaß an Sicherheit zu genießen. Das Spiel selbst wird dabei mittels des Mental Card Game Frameworks kryptographisch abgesichert.

Weiterführend kann der Client noch um weitere Funktionen erweitert werden. Beispielsweise könnten dem Spielskript mehr Funktionen zur Verfügung gestellt werden. Viele Spiele drehen die Karte beispielsweise um 90°, um einen bestimmten Zustand zu repräsentieren. Des Weiteren könnten Karten selbst kurze Skripte definieren, die das Spielskript aufrufen könnten.

Gleichzeitige Züge sind derzeit nicht erlaubt. Dies bedeutet, dass ein Spieler stets auf den anderen warten muss. Gerade für Spiele übers Internet ist Wartezeit ein Frustfaktor. Beispielsweise würde die Möglichkeit gleichzeitig Karten auszuspielen, in dem in dieser Arbeit implementierten Beispielspiel, erlauben, dass beide Spieler ihren Zug gleichzeitig ausführen. Damit würde sich die Dauer einer Runde, von der aktuellen Addition beider Züge, auf das Maximum beider Züge reduzieren. Somit reduziert sich auch die Wartezeit beider Spieler.

Die zurzeit verwendete Skriptengine erleidet Performance-Einbußen bei der Ausführung großer Skripte. Dies trifft unter anderem auf den TypeScript Compiler zu. Die JavaScript-Engine könnte beispielsweise durch andere Open Source Engines ersetzt werden, wie V8 oder Chakra.

Wie bereits erwähnt, konnten nicht alle Ziele vollkommen erfüllt werden. Eine Erhöhung der Sicherheit beim Tauschen wäre daher erstrebenswert. Die aktuell verwendete Variante, dass alle Clients möglichst alle anfallenden Daten zu Transaktionen sammeln, um Delinquenten zu finden, stellt auch einen möglichen Angriffsvektor da. Ein bösartiger Client könnte mutwillig eine Großzahl an Transaktionen erzeugen und somit auf den Clients unnötig viel Speicherplatz belegen.

7 Literaturverzeichnis

Blum, Manuel. „Coin flipping by telephone a protocol for solving impossible problems.“ *SIGACT News*, Januar 1981: 23-27.

Bundschuh, Peter. *Einführung in die Zahlentheorie.* Berlin: Springer, 1992.

Goldwasser, Shafi, und Silvio Micali. „Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information.“ *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing.* San Francisco, California, USA: ACM, 1982. 365-377.

Kryptowährung - Wikipedia. 16. Januar 2016. https://de.wikipedia.org/w/index.php?title=Kryptow%C3%A4hrung&oldid=150275497 (Zugriff am 30. Januar 2016).

Nakamoto, Satoshi. „www.bitcoin.org.“ *www.bitcoin.org.* 24. März 2009. https://bitcoin.org/bitcoin.pdf (Zugriff am 15. Februar 2016).

Schindelhauer, Christian. „A Toolbox for Mental Card Games.“ 1998.

Universally Unique Identifier - Wikipedia. 29. Januar 2016. https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=702228919 (Zugriff am 31. Januar 2016).

Zero-Knowledge-Beweis - Wikipedia. 26. Juli 2015. https://de.wikipedia.org/w/index.php?title=Zero-Knowledge-Beweis&oldid=144415014 (Zugriff am 6. Januar 2016).

Glossar

Deck	Eine Zusammenstellung von Karten.
Talon	Einen Satz Karten, die erst im Verlauf des Spiels genutzt werden. Meistens Kauf- oder Nachziehkarten.
TCG	Trading card game, z. dt. Sammelkartenspiel.
Assembly	Ein Container für .Net-Programmdaten, in der Regel DLL- oder EXE-Dateien.
OR-Mapper	Von Object-relational-mapping, z. dt. Objektrelationale Abbildung. Erlaubt die Persistierung von Objekten in bspw. eine Datenbank.
UPnP	Universal Plug and Play, Protokoll zur herstellerübergreifenden Kommunikation von Geräten.
PCL	Portable Class Library, eine .Net-Bibliothek die in, für verschiedenen Plattformen kompilierten, Assemblys verwendet werden kann.
Magic Byte	Auch allgemein Magic Number, z. dt. Magische Zahl. Ein spezieller Wert mit dem mansche Dateiformate den Datenbeginn kennzeichnen.

Erklärung der Kandidatin / des Kandidaten

☒ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist im Kapitel „Verantwortliche“ zu Beginn der Dokumentation aufgeführt.

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Name der Mitverfasser:.....

Datum

Unterschrift der Kandidatin / des Kandidaten