TINF21B6

Datenbanken Report

Decksterous (Card Game)

Benedikt Müll
Oliver Saar
Mats Fischer

27.06.2023

Inhaltsverzeichnis

1. Einleitung

1.1. Anwendung

Für unser Software Engineering Projekt haben wir uns für ein webbasiertes Kartenspiel entschieden. Bei diesem Kartenspiel gibt es verschiedene Spielweisen und Regeln, die von Nutzern individuell angepasst werden können, um das Spielverhalten/-erlebnis zu verändern. Auch ein Solo-Abenteuer-Modus ist verfügbar, um gegen KI gesteuerte Computer spielen zu können und Belohnungen, wie zum Beispiel Münzen und Erfahrungspunkte zu erhalten. Zudem gibt es tägliche Aufgaben die Spieler mit weiteren Gegenständen belohnen.

Dabei geht es nicht nur um den Spiel- sondern auch um den Sammelaspekt. Spieler können verschiedene Karten und andere Gegenstände sammeln, die eine gewisse Einzigartigkeit erhalten in dem diese nur in limitierten Zahlen oder zu speziellen Zeitpunkten/Events vorkommen können. Damit erhalten die verschiedenen Gegenstände einen individuellen Wert, der durch die Nützlichkeit von den Spielern selbst festgelegt werden kann. Dazu haben die Spieler die Möglichkeit ihre Gegenstände für einen eigen bestimmten Preis auf dem Community-Markt anzubieten und dort auch Gegenstände von anderen Spielern zu kaufen.

Nutzer können miteinander befreundet sein und Gegenstände auch auf direktem Wege austauschen. Die Inventare der Spieler sind unbeschränkt groß und bestimmte Gegenstände wie zum Beispiel ein Karten-Deck oder ein Karten-Pack können weitere Gegenstände für den Spieler beinhalten.

Neben der Inventarfunktion haben die Nutzer auch die Möglichkeit auf Spielstatistiken zugreifen zu können. Dabei werden ihre aktuellen Leistungen wie die Gewinnrate und mehr dargestellt. Auch ein Ranking aller Spieler ist einsehbar, welches sich täglich aktualisiert, sowie weitere nützliche globale Statistiken.

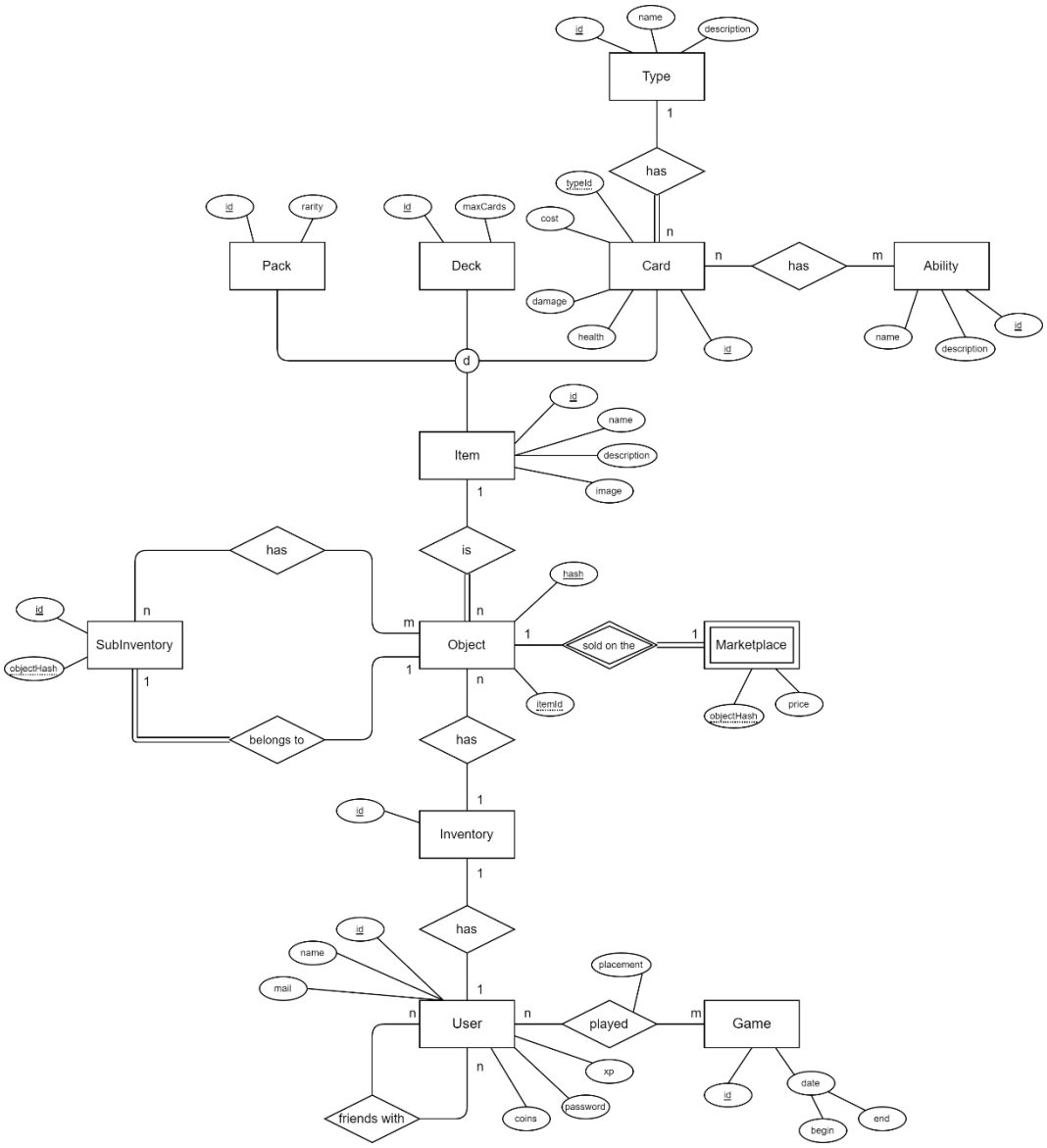
1.2. Umgebung

Für unsere Datenbank haben wir *MariaDB* im Einsatz, mit einem angepassten Benutzer für die Web-Anwendung, die auf einem eigenen Server gehostet wird. Als Anwendung benutzen wir *SQL-Workbench* um SQL-Befehle und weitere Funktionen auszuführen.

2. Model Gestaltung

Mithilfe der *SQL-Workbench* haben wir ein Schema sowie zusätzliche Views zuerst visuell modelliert.

2.1. EER Model Gestaltung



In dem EER-Model wird zuerst eine **User** Entität benötigt, mit den wichtigsten Daten, die dafür notwendig sind, um auch eine Anmeldung bereitzustellen. Dazu nötig sind die Attribute ***name***, ***mail***, ***password***, sowie weitere Attribute für Belohnungen ***xp***, und ***coins***. Da Spieler eventuell ihren Namen ändern wollen möchten, existiert ein ***id** Attribut*, um das Referenzieren auf den Benutzer eindeutig zu machen. Zudem können Benutzer auch mit anderen Benutzern befreundet sein, weshalb eine **n:m** Beziehung hierzu existiert.

Für Spieleinformationen wird die **Game** Entität benötigt, die per ***id*** einzigartig gemacht wird und zwei Datumsstempel ***begin*** und ***end*** besitzt. Über eine weitere **n:m** Beziehung können Spieler den Spielen zugewiesen werden, welche diese gespielt haben, mit einer zusätzlichen ***placement*** Information.

Die nächste Entität **Inventory** besitzt nur eine ***id***, da diese hauptsächlich dazu dient, einem Benutzer Objekte zuweisen zu können. Dabei existiert eine **1:1** Beziehung zu einem Benutzer.
Anmerkung: Wir haben uns als Team dazu entschieden es so zu handhaben, dass ein Benutzer gelöscht werden kann, aber dessen Inventar weiter existiert. Des Weiteren werden auch für die Computer-Bots Inventare benötigt. Da diese aber nicht als Benutzer gelten, müssen Inventare ohne Benutzer auskommen können.

Eine weitere wichtige Entität bildet hier **Item** als Gegenstand und dient als Generalisierung für weitere Entitäten. Diese beinhaltet gemeinsame Attribute wie ***name***, ***description*** und ***image***. Für eine eindeutige Referenzierung existiert hier ebenfalls ein ***id*** Attribut.

Eine Spezialisierung von einem Gegenstand bildet hier **Card**. Attribute dazu sind ***health***, ***damage***, ***cost*** und erneut eine ***id*** zur Eindeutigkeit. Zudem gibt es noch eine weitere Entität **Type** im Zusammenhang zu einer Karte mit ***name***, ***description*** und ***id*** Attributen. Eine Karte gehört somit einem Typ an und bildet eine **1:n** Beziehung. Zuletzt können Karten auch noch Fähigkeiten haben, die mit **Ability** realisiert werden. Diese Entität beinhaltet wieder ***name***, ***description*** und ***id*** als Attribute. Karten können mehrere Fähigkeiten haben, somit bildet dies eine **n:m** Beziehung.

Eine weitere Spezialisierung eines Gegenstandes ist das **Deck**. Dies trägt neben der ***id*** noch ein ***maxCards*** Attribut.

Die letzte Spezialisierung von Gegenstand bildet **Pack** mit einer ***id*** und einer ***rarity*** als Attribute.

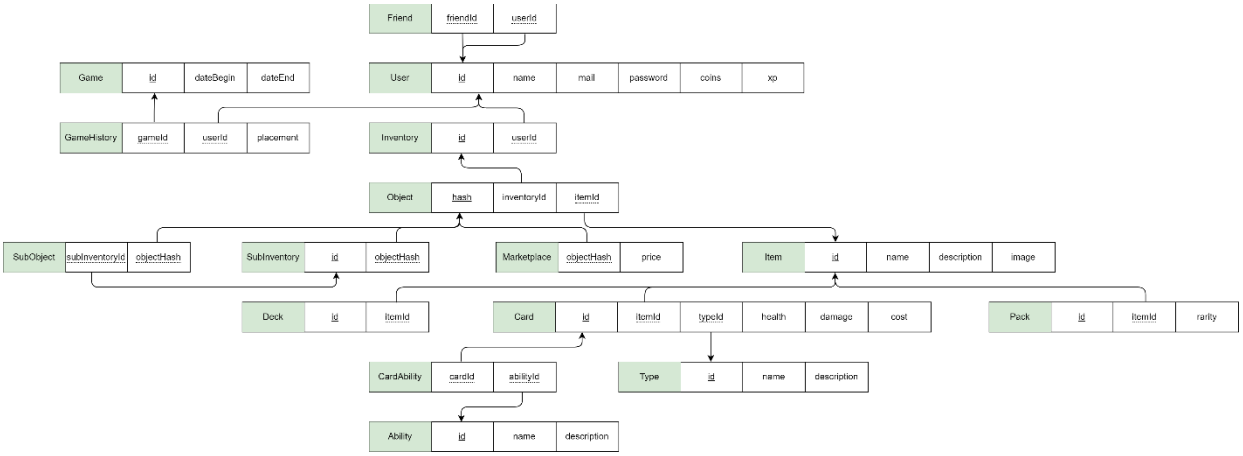
Das **Object** ist eine Instanziierung eines Gegenstandes und beinhaltet somit einen einzigartigen ***hash*** als Attribut. Da ein Gegenstand mehrfach instanziiert werden kann, bildet diese eine **1:n** Beziehung. Zudem kann ein Objekt einem Inventar zugewiesen werden, was eine weitere **1:n** Beziehung darstellt.

Ein Objekt kann zudem verkauft werden, was über die **Marketplace** Entität geschieht. Dabei erhält dies ein ***price*** Attribut und die Referenz ***objectHash*** zum Objekt. Dies bildet somit eine **1:1** Beziehung zu Objekt ab.
Anmerkung: Wir haben uns für den Namen “Marketplace” entschieden, da auch die Option offen gelassen wurde einen “Shop” zu erstellen der Gegenstände direkt anbietet. Somit wäre die Bezeichnung “Sell” oder Ähnliches nicht eindeutig gewesen.

Unter anderem sollen Objekte auch weitere Objekte beinhalten können, wofür die Entität **SubInventory** existiert, mit einem ***id*** Attribut und der Referenz ***objectHash*** zum Objekt. Diese bildet

somit eine **1:1** Beziehung zu Objekt ab. Nun können aber auch beliebig viele Objekte, beliebig vielen Unterinventaren zugewiesen werden wofür eine weitere **n:m** Beziehung abgebildet wird.

2.2. EER-Model zu relationalem Model



Das finale relationale Modell das aus dem EER-Model übersetzt wurde.

Anmerkung: Die Modellbezeichnungen befinden sich hier in den grünen Kästen, da Text über den Attributen Kästen schwer zu lesen war.

2.2.1. Many-To-Many Beziehungen auflösen (n:m)

Bei der Umsetzung zum relationalen Modell wurden zuerst alle **n:m** Beziehungen aufgelöst.

Die Beziehung von Nutzer zu Nutzer wurde einfach als **Friend** bezeichnet und beinhaltet eine ***userId*** sowie eine ***friendId*** zu den jeweiligen Nutzern.
Anzumerken ist hierbei eine wichtige Eigenschaft, die dadurch entsteht:

- Freundesanfragen von Nutzer *X* beinhalten in *userId* dessen ID.
- Freundeseinladungen an Nutzer *X* beinhalten in *friendId* dessen ID.
- Und für überlappende Einträge gilt eine einzige Freundschaft.

Die Beziehung von Nutzern zu Spielen, wurde mit **GameHistory** aufgelöst. Diese beinhaltet neben der ***gameId*** und ***userId*** auch ein ***placement*** des Nutzers.

Bei den Karten und Fähigkeiten wurde eine Auflösung **CardAbility** erstellt mit einer **cardId** und einer **abilityId**.

Und zuletzt wurde die Beziehung zwischen Unterinventaren und Objekten mit **SubObject** aufgelöst. Diese beinhaltet einfach eine ***subInventoryId*** und einen ***objectHash***.

2.2.2. One-To-Many Beziehungen umsetzen (1:n)

Bei der weiteren Umsetzung wurden anschließend alle **1:n** Beziehungen umgesetzt.

Zuerst bekamen die Karten eine ***typeId*** um die Beziehung zum Typ zu implementieren. Karten benötigen nämlich immer einen Typ.

Anschließend wurden alle Spezialisierungen mit einer ***itemId*** versehen, um die Beziehung mit den generalisierten Attributen zu verknüpfen.

Auch das Objekt bekam eine ***itemId*** um auf dessen Instanziierung zu referenzieren. Zuletzt wurde das Objekt auch noch mit einem ***inventoryId*** Attribut ausgestattet, da ein Objekt, ohne eine Inventarzuweisung nicht zu existieren braucht.

2.2.3. One-To-One Beziehungen umsetzen (1:1)

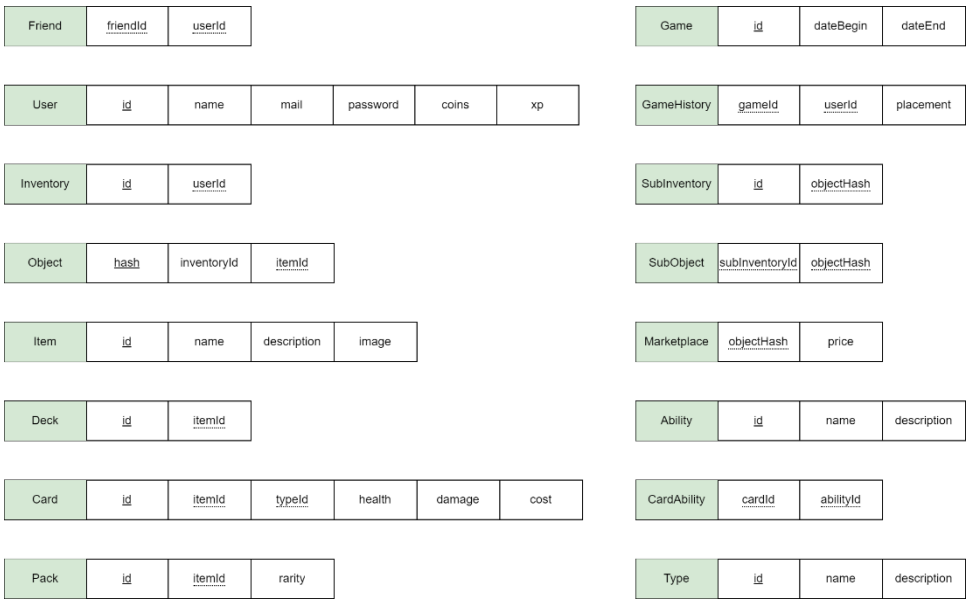
Abschließend wurden alle 1:1 Beziehungen umgesetzt.

Das Inventar bekam eine ***userId*** zur Verknüpfung zu einem Nutzer.

Der Marktplatz, wie vorhin schon erwähnt, besitzt einen ***objectHash*** Attribut zur Referenzierung zu dessen Objekt, genauso wie das Unterinventar mit dem ***objectHash*** Attribut.

2.3. Normalisierung des relationalen Models

2.3.1. First Normal Form (1NF)

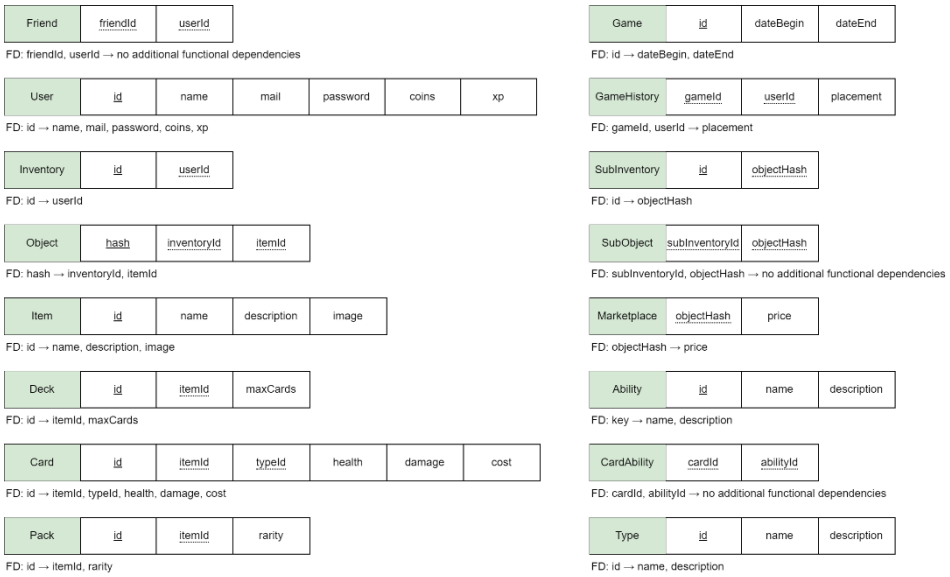


Die erste Normalform besagt, dass jede Zelle einer Tabelle einen einzigen Wert enthalten soll (atomare Werte). Daher sollten keine mehrwertigen Attribute in einer Zelle vorhanden sein.

Außerdem müssen eindeutige Spaltennamen vorliegen, d.h. jede Spalte kann identifiziert werden und es entstehen keine Verwechslungen oder Duplikate.

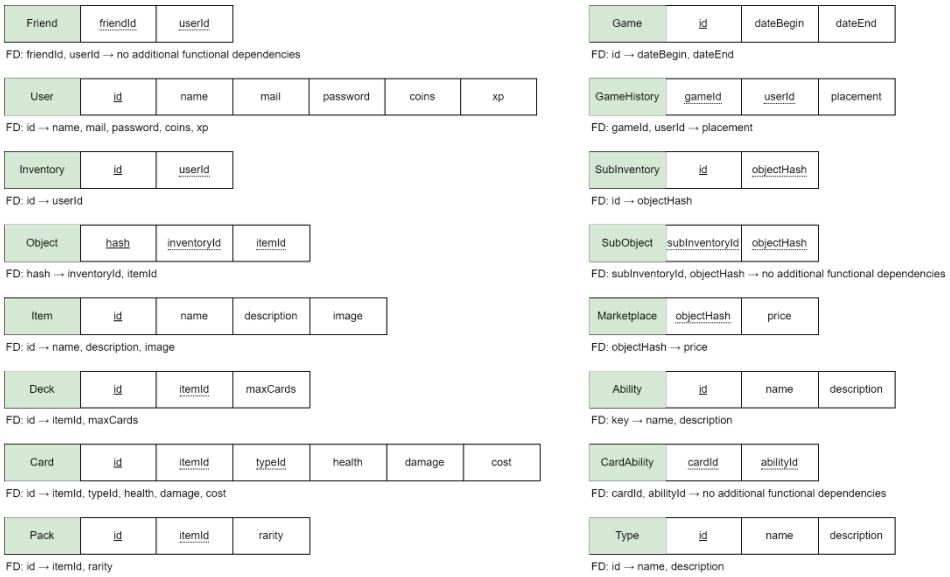
Ein weiterer wichtiger Punkt ist die eindeutige Reihenidentifikation, was bedeutet, dass jede Zeile eindeutig identifiziert werden kann. Dies wird durch die Verwendung eines Primärschlüssels erreicht, welcher ein eindeutiges Attribut für jede Zeile ist.

2.3.2. Second Normal Form (2NF)



Der wichtigste Punkt der zweiten Normalform besagt, dass alle Spalten, die nicht Teil des Primärschlüssels sind, funktional abhängig von der gesamten Kombination des Primärschlüssels sind. Wichtig ist, dass die Abhängigkeit die Gesamtheit des Schlüssels betrifft und nicht nur einen Teil. Alles in Allem trägt die zweite Normalform dazu bei, Datenredundanzen zu vermeiden und Probleme bei der Datenintegrität zu verhindern, indem sichergestellt wird, dass die Tabellendaten strukturiert und sinnvoll organisiert sind.

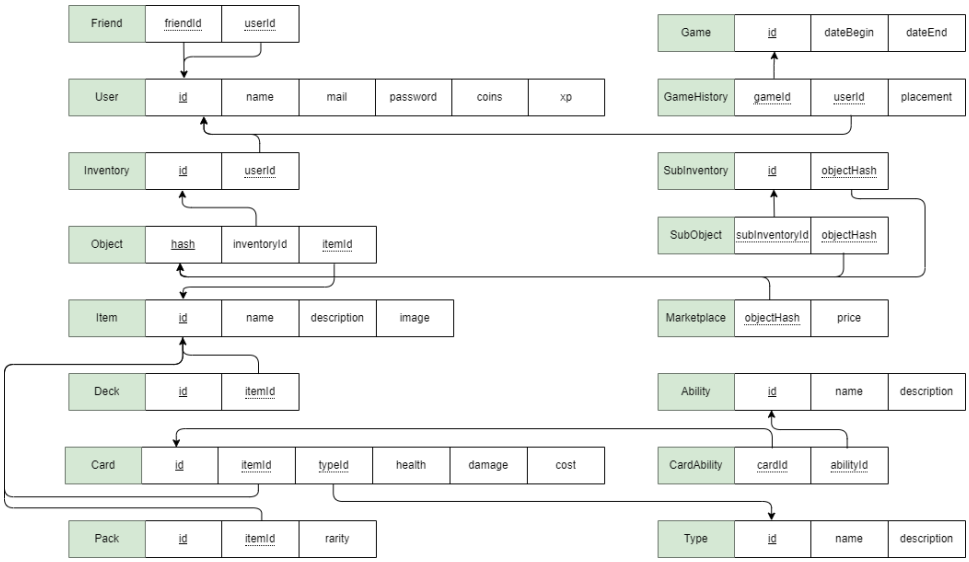
2.3.3. Third Normal Form (3NF)



Die dritte Normalform verfolgt das Ziel, transitive Abhängigkeiten auszulöschen. So eine Abhängigkeit tritt auf, wenn ein Attribut von einem Nicht-Schlüssel-Attribut abhängig ist, anstatt von dem Primary-Key.

Hierfür müssen die funktionalen Abhängigkeiten identifiziert und geprüft werden, ob diese transitiv sind. Falls Transitivität vorliegt, muss diese entfernt werden, indem eine neue Tabelle für diese Beziehung erstellt wird.

2.3.4. Fourth Normal Form (4NF)



Die Voraussetzungen für die vierte Normalform sind, dass natürlich die dritte Normalform vorliegen muss und zusätzlich sollen keine non-trivial multivalued Abhängigkeiten vorhanden sein. Dadurch ist dann die Boyce-Codd Normal Form (BCNF) ebenfalls erfüllt.

3. Constraints

Durch die Beziehung zwischen Object und Marketplace wird beim Kaufen, sowie Löschen eines Objects auf dem Marketplace und der Löschung des generellen Objects, die Beziehung und das Marketplace Object gelöscht. Dieses besteht nur, wenn wirklich etwas auf dem Marketplace angeboten wird.
Bei der Beziehung von SubInventory und Inventory ist die Löschung identisch geregelt. Wird ein Object mit einem SubInventory gelöscht, werden neben der Löschung des SubInventory auch die Beziehungen von SubInventory und Object gelöscht. Wird ein Inventory gelöscht, werden alle Objects die dazu gehören gelöscht.

Wenn ein Item gelöscht wird, müssen die zugehörigen Spezialisierungen gelöscht werden. Falls also eine Karte gelöscht wird, muss der Karteneintrag gelöscht werden und bei einem Pack muss der Packeintrag gelöscht werden.

Bei Löschung eines Nutzers, werden die Entries des Benutzers auch gelöscht. Das eigentliche Objekt bleibt aber bestehen, um trotzdem noch eine Match-Historie zu haben oder seltene Karten nicht permanent zu löschen.

3.1. Domain Constraint

Alle Attribute in unserer Datenbank erfüllen den **Domain Constraint** und haben einen fest definierten **Data Type** und weiteren **Constraints** wie zum Beispiel *NOT NULL* für Bezeichnungen, *UNIQUE* für Benutzernamen und *DEFAULT* für Zahlenwerte, sowie *PRIMARY KEY* und *FOREIGN KEY* für Primär- und Fremdschlüssel.

Wir haben **Length Constraints** in allen Stringattributen, wie zum Beispiel Namen oder auch dem Passwort, welches verschlüsselt als String gespeichert wird.

3.2. Key Constraints

Auflösende Tabellen für **n:m** Beziehungen bilden ihren **Superkey** aus zwei Fremdschlüsseln, die auf die jeweiligen Tabellen referenzieren. Dadurch ist gegeben das kein doppelter Eintrag in der Beziehung existieren kann. Als Beispiel dafür, kann eine Karte mit der ID 1 nur einmal die Fähigkeit mit ID 3 besitzen.

Für alle Einträge haben wir sichergestellt, dass die **Primary Key** Spalten eindeutig und einzigartig sind, womit der **Primary Key Constraint** erfüllt ist.

3.3. Entity Integrity Constraints

Unique Integrity Constraints sind auch bei den meisten Tabellen gegeben, da die einzelnen Werte eindeutig sein müssen und so doppelte Werte in einer Spalte verhindert werden.

Der **Primary Key** darf *NOT NULL* sein und es dürfen keine doppelten Werte vorhanden sein. Dies gilt überall, da in jeder Tabelle ein einzigartiger Primärschlüssel vorhanden ist.

3.4. Referential Integrity Constraints

Stellen sicher, dass in einer Fremdschlüsselbeziehung die referenzierten Werte gültig sind und auf existierende Daten in der verknüpften Tabelle verweisen. Als Beispiel dafür kann man eine Karte nehmen, die ein Fremdschlüssel für einen Typ beinhaltet.

Wenn ein Gegenstand gelöscht werden soll, dann kann eine Spezialisierung wie zum Beispiel eine Karte auf die ***itemId*** referenzieren, weshalb der Gegenstand nicht gelöscht werden kann.

4. Implementierung

Folgend haben wir Beispiele und Auszüge der Implementierung unseres Schemas in der Datenbank.

4.1. Views

In diesem Projekt müssen an verschiedensten Stellen Datenbank-Zugriffe getätigt werden, um die richtige Funktionalität aller technischen Aspekte zu garantieren. Einerseits werden in einigen Fällen an verschiedenen Stellen ähnliche oder gleiche Zugriffe verwendet und andererseits werden bestimmte Anfragen im Kontext anderer Anfragen auf verschiedenste Weise verwendet. Um diesen Sachverhalt effizient anzugehen, wurden zahlreiche Views erstellt, da diese einige Vorteile für diese Situation, wie vereinfachte Datenbankzugriffe, Leistungsverbesserungen und Flexibilität mit sich bringen. Dadurch ist es möglich auf die vordefinierten Views zuzugreifen, ohne komplexe Abfragen erneut zu stellen.

Die in der folgenden Abbildung dargestellte View **UserWins** ermöglicht, dass die Anzahl aller Siege eines Users ausgegeben wird. Dies wird mithilfe eines Joins auf die Spielhistorien-Tabelle **GameHistory** gelöst, welche das Attribut ***placement*** besitzt. Alle Spiele, bei welchen der User nun das “placement = 1” hat, sprich gewonnen hat, werden nun dazu gezählt. Die daraus hervorgehende Tabelle zeigt die ID des Users und die Anzahl seiner Gewinne.

```
DROP TABLE IF EXISTS `card_game_dev`.`UserWins`;
USE `card_game_dev`;
# Get count of wins based on userId
CREATE  OR REPLACE VIEW `UserWins` AS
SELECT Self.id as userId, Other.wins FROM User as Self
LEFT JOIN (SELECT User.id, count(placement) as wins FROM User
        LEFT JOIN GameHistory ON userId = id
        WHERE placement = 1
        GROUP BY User.id) as Other ON Other.id = Self.id;
```

Die View **UserObjects** gibt Informationen über jene Objekte zurück, welche ein bestimmter Benutzer besitzt. Dies ermöglich eine Query, welche die Benutzer-Tabelle **User** mit der Inventar-Tabelle **Inventory**, der Objekt-Tabelle **Object** und der Gegenstands-Tabelle **Item** joint. Diese gelieferten Informationen sind der Hash, der Name und die Beschreibung des jeweiligen Objekts.

```
DROP TABLE IF EXISTS `card_game_dev`.`UserObjects`;
USE `card_game_dev`;
# Get objects belonging to a user over userId
CREATE  OR REPLACE VIEW `UserObjects` AS
SELECT User.id as userId, Object.hash, Item.name, Item.description FROM User
LEFT JOIN Inventory ON Inventory.userId = User.id
LEFT JOIN Object ON Object.inventoryId = Inventory.id
LEFT JOIN Item ON Item.id = Object.itemId;
```

Um alle Items aus dem Marketplace abzurufen, wird die folgende View **MarketplaceItems** verwendet. Auch in dieser View werden der Marktplatz-Tabelle **Marketplace** die Tabellen gejoint, welche die relevanten Informationen für die Ergebnis-Tabelle enthalten. Es muss auf den ***hash***, die ***itemId***, die ***inventoryId*** und die ***userId*** gejoint werden, um die Items zu erhalten. Die Ergebnis-Tabelle enthält nun Informationen über alle auf dem Marktplatz angebotenen Gegenstände.

```
DROP TABLE IF EXISTS `card_game_dev`.`MarketplaceItems`;
USE `card_game_dev`;
# Get all marketplace infos from an object with objectHash
CREATE  OR REPLACE VIEW `MarketplaceItems` AS
SELECT User.id as userId, Object.hash as objectHash, Marketplace.price, Item.id as itemId, Item.name, Item.description FROM Marketplace
LEFT JOIN Object ON Object.hash = Marketplace.objectHash
INNER JOIN Item ON Item.id = Object.itemId
LEFT JOIN Inventory ON Inventory.id = Object.inventoryId
LEFT JOIN User ON User.id = Inventory.userId;
```

Um Statistiken, wie zum Beispiel die Anzahl der heute gespielten Spiele anzuzeigen, wird die View **UserTodayPlays** benötigt. Dazu wird die Tabellen der Nutzer, Spiele und der Spielhistorie gejoint. In dieser resultierenden Tabelle wird nach dem heutigen Datum gefiltert, alle Spiele eines Nutzers gruppiert und somit die Information geliefert, wie viele Spiele ein Spieler heute gespielt hat. Die finale Tabelle enthält verschiedene Informationen über den Nutzer, sowie die tatsächliche Anzahl der heute gespielten Spiele.

```
DROP TABLE IF EXISTS `card_game_dev`.`UserTodayPlays`;
USE `card_game_dev`;
# Get count of todays played games
CREATE  OR REPLACE VIEW `UserTodayPlays` AS
SELECT Self.id as userId, Self.name, Self.mail, Self.password, Self.xp, Self.coins, Other.plays FROM User AS Self
LEFT JOIN (SELECT User.id, count(*) as plays FROM User
        LEFT JOIN GameHistory ON GameHistory.userId = User.id
    LEFT JOIN Game ON Game.id = GameHistory.gameId
        WHERE DATE(Game.beginDate) = CURDATE()
        GROUP BY User.id) as Other ON Other.id = Self.id;
```

FriendAccepted ist eine View, die angibt, welche Freundschaftsanfragen ein Benutzer angenommen hat und somit, mit welchen anderen Benutzern dieser befreundet ist. Durch das Joinen der Nutzer-Tabelle **User** und der Freundes-Tabelle **Friend** werden alle Informationen des betrachteten Nutzers geliefert, sowie erneut Name, Mail, (verschlüsseltes) Password, Erfahrungspunkte und die Menge an Münzen des Benutzers dessen Freundschaftsanfrage angenommen wurde

```
DROP TABLE IF EXISTS `card_game_dev`.`FriendAccepted`;
USE `card_game_dev`;
# Get all accepted friends of userId
CREATE  OR REPLACE VIEW `FriendAccepted` AS
SELECT Self.*, User.name, User.mail, User.password, User.xp, User.coins FROM Friend AS Self
INNER JOIN Friend AS Other ON Other.userId = Self.friendId AND Other.friendId = Self.userId
LEFT JOIN User ON User.id = Self.friendId;
```

Die View **FriendInvites** gibt an, welche anderen Benutzer ein bestimmter Benutzer Freundschaftsanfragen versendet hat. Ähnlich wie in der vorherigen View werden die Nutzer und die Freundes-Tabelle gejoint. Es werden sämtliche Informationen über den versendeten Benutzer geliefert, sowie Name, Mail, (verschlüsseltes) Passwort, Erfahrungspunkte und die Menge an Münzen des Benutzers, welcher die Anfrage erhalten hat.

```
DROP TABLE IF EXISTS `card_game_dev`.`FriendInvites`;
USE `card_game_dev`;
# Get all invitations send by userId
CREATE  OR REPLACE VIEW `FriendInvites` AS
SELECT Self.*, User.name, User.mail, User.password, User.xp, User.coins FROM Friend AS Self
LEFT JOIN Friend AS Other ON Other.userId = Self.friendId AND Other.friendId = Self.userId
LEFT JOIN User ON User.id = Self.friendId
WHERE Other.userId IS NULL;
```

Umgekehrt liefert die View **FriendRequests** Informationen darüber zurück, wer an einen bestimmten Benutzer eine Freundschaftsanfrage gesendet hat. Auch hier werden durch ähnliche Joins, die Eigenschaften des Benutzers, welcher die Anfrage gesendet hat, wie Name, Mail, (verschlüsseltes) Passwort, Erfahrungspunkte und die Menge an Münzen im Inventar zurückgeliefert. Weiterhin wird aufgelistet, an welchen Benutzer diese Freundschaftsanfrage ging.

```
DROP TABLE IF EXISTS `card_game_dev`.`FriendRequests`;
USE `card_game_dev`;
# Get all friend requests for userId
CREATE  OR REPLACE VIEW `FriendRequests` AS
SELECT Self.friendId as userId, Self.userId as friendId, User.name, User.mail, User.password, User.xp, User.coins FROM Friend AS Self
LEFT JOIN Friend AS Other ON Other.userId = Self.friendId AND Other.friendId = Self.userId
LEFT JOIN User ON User.id = Self.userId
WHERE Other.userId IS NULL;
```

4.2. Use Cases und Workflow

Im weiteren Verlauf haben wir ein paar *Use Cases* definiert, die auch in unserer Anwendung Einsatz finden, sowie den Workflow hinter den Abfragen.

Kaufen eines Community Items

```
# Buyed item is (objectHash): 774E9O4C3MKCNSNH
# Buyer is (userId): 6

# Add money to seller
UPDATE User
SET coins = coins + (SELECT price FROM MarketplaceItems
        WHERE objectHash = '774E9O4C3MKCNSNH')
WHERE id = (SELECT userId FROM MarketplaceItems
        WHERE objectHash = '774E9O4C3MKCNSNH');

# Remove money from buyer
UPDATE User
SET coins = coins - (SELECT price FROM MarketplaceItems
        WHERE objectHash = '774E9O4C3MKCNSNH')
WHERE id = 6;

# Update object to buyer inventory
UPDATE Object
SET inventoryId = (SELECT id FROM Inventory
        WHERE userId = 6)
WHERE hash = '774E9O4C3MKCNSNH';

# Remove object from marketplace
DELETE FROM Marketplace
WHERE objectHash = '774E9O4C3MKCNSNH';
```

Ein Anwendungsfall, welcher sehr oft auftreten kann, ist das ein Benutzer einen Gegenstand kauft, der durch einen anderen Benutzer auf dem Marktplatz zur Verfügung gestellt wurde. Die Query für diese Aktion ist jedoch etwas länger als manche Anderen. Zunächst muss dem Verkäufer die entsprechende Menge Münzen gutgeschrieben werden, welche für den Gegenstand verlangt wurden. Danach können dem Käufer Münzen abgezogen und aktualisiert werden, in welchen Inventar sich der Gegenstand befindet. Als letztes wird das Angebot des Gegenstandes vom Marketplace entfernt.

Alle Gegenstände aus einem Sub-Inventar abfragen

```
SELECT Self.hash, SelfItem.name, SelfItem.description, Other.hash AS inventoryHash, OtherItem.name AS inventoryName, OtherItem.description AS inventoryDescription FROM Object AS Self
INNER JOIN Item AS SelfItem ON SelfItem.id = Self.itemId
RIGHT JOIN SubObject ON SubObject.objectHash = Self.hash
INNER JOIN SubInventory ON SubInventory.id = SubObject.subInventoryId
LEFT JOIN Object AS Other ON Other.hash = SubInventory.objectHash
INNER JOIN Item AS OtherItem ON OtherItem.id = Other.itemId
```

Ein weiterer Anwendungsfall ist das Konzept der Sub-Inventare. Da es einige Gegenstände gibt, die selbst weitere Gegenstände enthalten sollen - Decks und Karten-Päckchen beispielsweise - besitzen diese Sub-Inventare. Anwendungsbeispiele für diese Query sind: Den Inhalt eines Decks am Anfang eines Spiels oder beim Öffnen eines Päckchens dessen Inhalt abrufen, welche alle Gegenstände zurückgibt, die sich in einem Sub-Inventar befinden.

5. Abschluss

Alle Grafiken und Skripte können in unserem Software Engineering GitHub Repository gefunden werden.
(https://github.com/i3ene/decksterous/tree/gh-pages)

Das Schema Skript mit den Views und Beispieldaten kann direkt hier gefunden werden.
(https://github.com/i3ene/decksterous/blob/gh-pages/files/database/schema.sql)

5.1. Highlights bei der Gestaltung

Eine wichtige Eigenschaft, die sich durch unsere Modellierung gebildet hat, neben der Handhabung von Freunden, ist die Möglichkeit Unterinventare in weiteren Unterinventaren zu haben.
Diese zyklische Referenz ermöglicht auch komplexere Abbildungen von Daten wie zum Beispiel, dass ein Karten-Pack aus weiteren Karten-Packs bestehen könnte und diese dann schlussendlich Karten beinhalten. Generell lässt sich sowas besser auf eine neuen “Belohnungsgegenstand” sogar besser abbilden: Ein Spieler absolviert beim Spielen eine Aufgabe und bekommt diesen Belohnungsgegenstand, der beim Öffnen weitere Gegenstände beinhaltet. Diese Gegenstände können dann Karten-Packs beinhalten.

5.2. Teilnehmer Beiträge

	Benedikt	*Oliver*	*Mats*
Anwendung	+++	+++	+++
EER-Model	++	+	+
Relational Model	+		
Normalisierung		+	
Constraints	+	++	+
Implementierung		+	++