Datenbanksysteme
Hausarbeit
TauDBsi

Dominik Stelter
Jan von Oehsen
Jendrik Grittner

Bremen, 02.03.2017

Inhaltsverzeichnis

1 Informelle Anwendungsbeschreibung (Jan)

In dieser Hausarbeit wird ein Datenbanksystem für das populäre, mobile Spiel „Pokémon Go“ beschrieben, das im Sommer 2016 einen weltweiten Hype ausgelöst hat. Von den Entwicklern her wurde die Datenbank nie veröffentlicht, sodass die wirkliche Implementierung und die zugrunde liegenden Datenstrukturen vollkommen unbekannt sind.

Pokémon Go ist ein geobasiertes Augmented-Reality-Spiel, bei dem man in der echten Welt Pokémon fangen kann und als Trainer per GPS-Erkennung eines Smartphones über den Standort neuer Pokémon aufmerksam gemacht wird.

In dieser Hausarbeit wird nicht die aktuellste Version des Spiels betrachtet. Vor kurzem wurde ein umfassendes Update veröffentlicht, das von uns nicht berücksichtigt wird. Als Grundlage nehmen wir die erste Version, die im Sommer 2016 erschienen ist und bis vor kurzem fast unverändert im Einsatz war.

Außerdem mussten wir uns entschließen, aufgrund der Komplexität des Themas einige Dinge zu vereinfachen, da die Datenbank sonst den zur Verfügung stehenden Rahmen sprengen würde. Dazu zählen vor allem konkrete Werte für einsetzbare Gegenstände (Items), wie z.B. deren Wirkung und Einsatzzeit.

1.1 Pokémon

„Pokémon“ sind tierähnliche Fantasiewesen, die der Spieler fangen, sammeln und gegeneinander kämpfen lassen kann. Sie haben verschiedene Typen wie z.B. Feuer, Wasser oder Pflanze, die im Kampf gegeneinander unterschiedlich effektiv sind. Pokémon können bis zu 2 verschiedene Attacken beherrschen, die auch einen Typ besitzen. Außerdem besitzen Pokémon Wettkampfpunkte, die ihre Kampfkraft angeben. Diese Wettkampfpunkte können durch spezielle Bonbons erhöht werden. Ein anderer Nutzen der Bonbons besteht darin, Pokémon zu entwickeln, sodass sie sich in ein anderen Pokémon verwandeln.

1.2 Spieler

Ein Spieler zeichnet sich durch viele unterschiedliche Dinge aus. Neben dem Namen, Level, Erfahrung oder der Anzahl an Sternenstaub besitzt er auch eine Sammlung von Pokémon, die er gefangen hat. Dazu kommt eine Sammlung an Items, die bestimmte Effekte haben. Außerdem kann er verschiedene Eier besitzen, aus denen Pokémon schlüpfen können und er gehört einem Team an.

1.3 Arenen

Arenen befinden sich an bestimmten Positionen in der realen Welt und gehören jeweils zu einem bestimmten Team. Spieler des Teams können darin ihre Pokémon

zur Verteidigung lagern und Spieler aus anderen Teams können wiederum mit ihren eigenen Pokémon versuchen, eine fremde Arena einzunehmen. Wenn die Einnahme gelingt, dann wechselt die Arena in den Besitz des neuen Teams.

1.4 Items

Items sind einsetzbare Gegenstände mit bestimmten Effekten (die in dieser Hausarbeit allerdings nicht berücksichtigt werden). Sie dienen zum Beispiel dem Fangen oder Heilen von Pokémon. Items können von Spielern gesammelt und eingesetzt werden.

2 Konzeptioneller Entwurf als UML Klassendiagramm und ER-Schema (Jan)

2.1 Grafik zum UML-Klassendiagramm (Dominik)

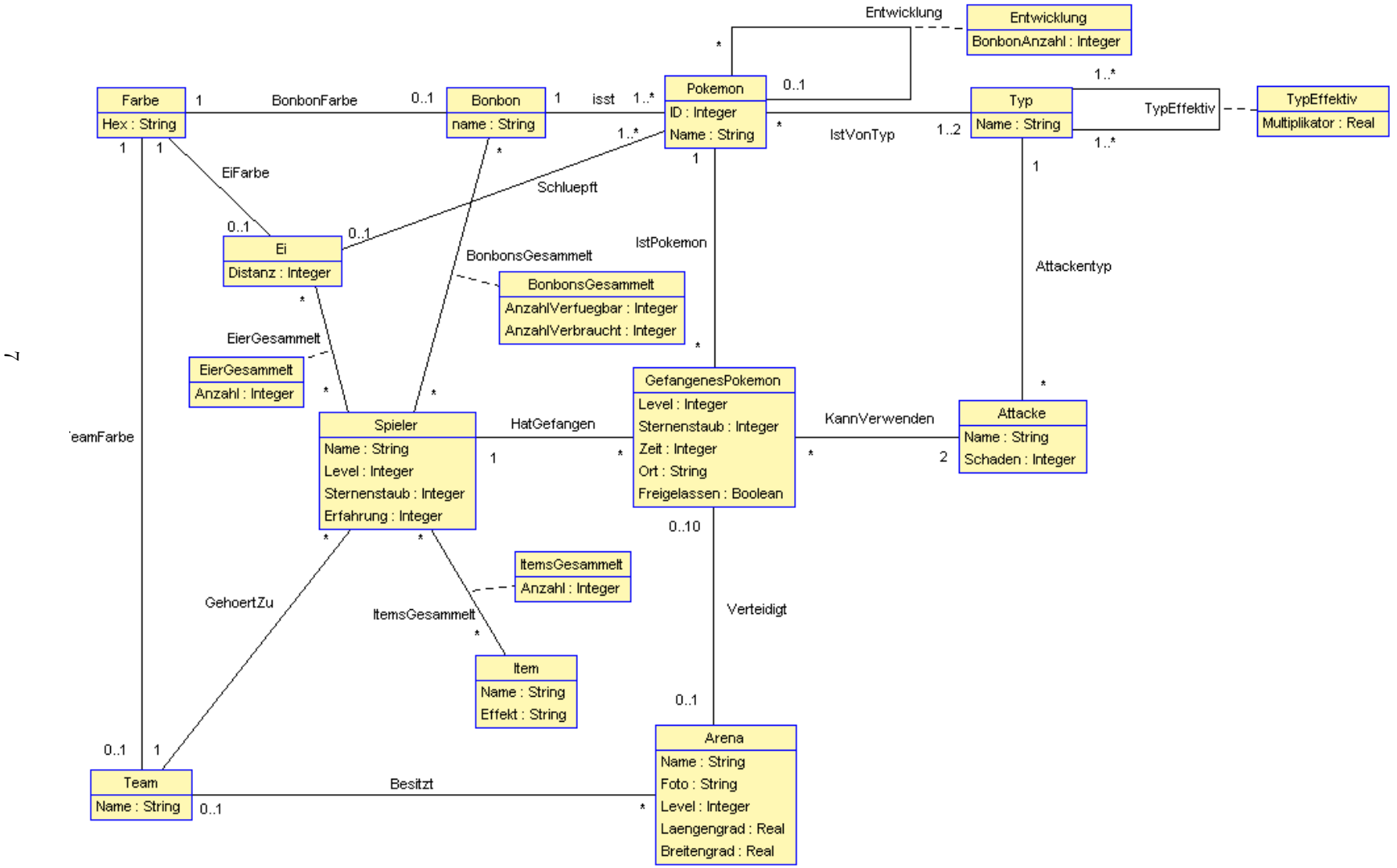

Abbildung 1: UML Klassendiagramm

2.2 Erläuterung zum UML-Diagramm

Das dargestellte UML-Klassendiagramm zeigt den groben Aufbau der Datenbank. Die Klassen entsprechen dabei den Tabellen, die Attribute der Klassen sind Attribute in den Tabellen und die Beziehungen werden über Fremdschlüssel geregelt. Die Datentypen im UML-Diagramm sind auch die Datentypen in der Datenbank.
Da jede der Klassen eine ID als Primary Key besitzt wird diese in den folgenden Erläuterungen nicht jedes mal aufgezählt.

2.2.1 Klassen

Spieler
Die zentrale Klasse trägt den Namen „Spieler“ und enthält Daten vom Spieler und dessen Spielfortschritt. Es gibt Attribute für Name (String), Level (Integer), Sternenstaub (Integer) und Erfahrung (Integer).

Pokemon
Diese Klasse dient dazu, die grundlegenden Daten von Pokémon abzuspeichern. Dafür hat es die Attribute Name (String) und BonBonBedarf (Integer), also wie viele BonBons für eine Entwicklung benötigt werden.

GefangenesPokemon
In dieser Klasse werden die Daten von den individuellen Pokémon abgelegt, die von Spielern gefangen wurden („Vererbung“ der Klasse Pokemon). Deswegen belaufen sich die Attribute dieser Klasse auf Werte, die bei gefangenen Pokemon der gleichen Art unterschiedlich sein können: Level (Integer), Sternenstaub (Integer), Zeit (Integer), Ort (String) und Freigelassen (Boolean).

Arena
Diese Klasse enthält Daten zu Arenen. Die Attribute sind Name (String), Foto (String), Level (Integer), Längengrad (Real) und Breitengrad (Real). Real steht dabei für eine Fließkommazahl, in einer SQL-Datenbank entspricht dies dem Datentyp Float.

Team
Die Klasse Team ist für die (momentan) 3 verschiedenen Teams zuständig. Viele Attribute werden nicht benötigt, deswegen gibt es nur einen Namen (String).

Farbe
Weil Farben für viele unterschiedliche Dinge benötigt werden, haben wir ihnen eine eigene Klasse gegeben, anstatt überall den Farbwert neu einzutragen. Diese Klasse hat einen Farbwert (String) als Attribut, was den Hexadezimalwert der Farbe angibt.

Ei
Die Klasse Ei enthält Daten zu den Eiern, die Spieler besitzen. Als Attribut gibt es nur die Distanz (Integer), die angibt wie weit das Ei bereits ausgebrütet ist.

Item
Die Klasse Item enthält Daten zu Items von Spielern. Sie hat einen Namen (String) und einen Effekt, der auch als String gespeichert wird.

Attacke
Pokémon können unterschiedliche Attacken beherrschen und für solche Attacken ist diese Klasse da. Neben dem Namen (String) gibt es auch einen Schadenswert (Integer) als Attribut.

Typ
Die Klasse Typ ist für die unterschiedlichen Typen, die sowohl Pokémon als auch Attacken haben können, notwendig. Dafür wird nur das Attribut Name (String) benötigt.

2.2.2 Assoziationen

GehoertZu
Eine 1 zu n Beziehung, sodass beliebig viele Spieler zu genau einem Team gehören können.

BonbonsGesammelt
Eine n zu n Beziehung. Gibt an, wie viele Bonbons von welchen Sorten ein Spieler besitzt. Hierfür wird die Assoziationsklasse „BonbonsGesammelt“ benötigt, die es ermöglicht neben der Anzahl an Bonbons die ein Spieler besitzt auch die bereits verbrauchten abzuspeichern.

EierGesammelt
Eine n zu n Beziehung, sodass Spieler beliebig viele Eier besitzen können. Auch hierfür wird wie immer bei n zu n Beziehungen eine Assoziationsklasse benötigt.

HatGefangen
Eine 1 zu n Beziehung, sodass die individuell gefangenen Pokemon auch nur zu genau einem Spieler gehören.

ItemsGesammelt
Eine n zu n Beziehung, die angibt wie viele Items welcher Sorten die Spieler besitzen. Auch hier wird wieder eine Assoziationsklasse benötigt.

TeamFarbe
BonBonFarbe
EiFarbe

Alle Assoziationen die mit Farbe zu tun haben sind 1 zu n Beziehungen. So kann jede Instanz einer angeschlossenen Klasse eine eigene Farbe haben.

Isst
Eine 1 zu n Beziehung. Eine Bonbonart kann genau von einer Evolutionsreihe eines Pokémon gegessen werden. Pokémon können unterschiedlich viele Entwicklungsstufen haben, auch wenn momentan maximal 4 möglich sind haben wir uns aus Gründen der Erweiterbarkeit nicht für 1..4 entschieden.

Besitzt
Diese Beziehung ist eine 1 zu n Beziehung, da jedes Team beliebig viele Arenen besetzen kann und jede Arena von genau einem Team besessen wird.

Verteidigt
In einer Arena können bis zu 10 Pokémon zur Verteidigung gelagert werden und diese Pokémon können nicht in anderen Arenen eingesetzt werden. Deswegen ist dies eine 1 zu 1..10 Beziehung.

IstPokemon
Die Klasse GefangenesPokemon kann quasi als Instanz der Klasse Pokemon gesehen werden. Es liegt eine 1 zu n Beziehung vor, weil es beliebig viele konkrete Pokémon einer Art geben kann.

EntwickeltSichAus
Ein Pokémon kann sich in ein anderes Pokémon entwickeln, muss es aber nicht. Deswegen liegt hier eine 1 zu 0..1 Beziehung vor, an der nur die Klasse Pokémon mit sich selbst teilnimmt.

KannVerwenden
Hier liegt eine 1 zu 2 Beziehung vor, da jedes Pokemon genau 2 Attacken beherrscht.

Attackentyp
Jede Attacke hat genau einen Typ, und Typen können von mehreren Attacken besessen werden. Deswegen ist es eine 1 zu n Beziehung.

IstVonTyp
Diese Beziehung ist eine 2 zu n Beziehung, da jedes Pokemon 2 Typen hat.

IstEffektiv
Typen sind gegen anderen Typen unterschiedlich effektiv. Diese Assoziation gibt dies an, indem je 2 Typen mit der TypEffektiv Tabelle in Beziehung gesetzt werden. Demnach liegt hier eine 2 zu n Beziehung vor.

TypEffektiv
Diese Assoziation dient dazu, um die unterschiedlichen Wechselwirkungen zwi-

schen den Typen darzustellen. Als zentrales Attribut gibt es einen Multiplikator, der als Real abgespeichert wird. Er gibt die Effektivität von einem Typ gegen einen anderen an, z.B. 0.5, 1 oder 2. Die beiden Typen werden mit den Rollen Angreifer und Verteidiger verknüpft.

2.3 Grafik zum ER-Diagramm (Dominik)

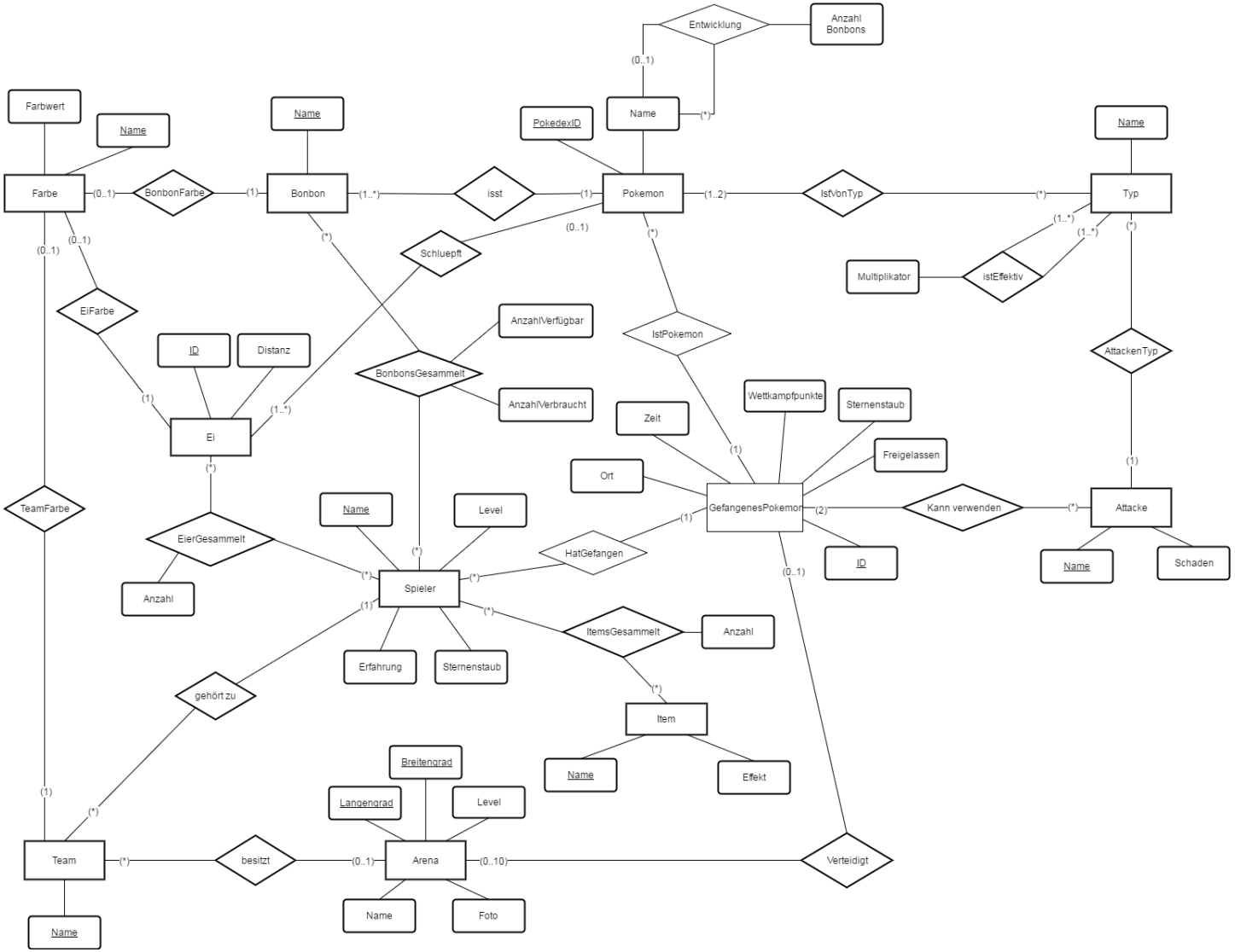

Abbildung 2: ER Diagramm

2.4 Erläuterung zum ER-Diagramm

Das hier gezeigte ER-Diagramm stellt die Entitäten und ihre Beziehungen zueinander da. Entitäten sind in rechteckigen Kästen mit spitzen Ecken gezeichnet, die den Tabellen in der Datenbank entsprechen. Die Attribute der Entitäten sind damit verbunden und in rechteckigen Kästen mit abgerundeten Ecken dargestellt. Diese entsprechen den Attributen der Tabellen. Entitäten können Beziehungen zueinander besitzen. Wenn dies der Fall ist, sind sie über Linien miteinander verbunden, die Beziehung befindet sich dabei in einer Raute auf der Linie. Wenn an solch eine Beziehung direkt Attribute angeschlossen sind, dann werden sie in der Datenbank als eigene Tabelle umgesetzt. An den Beziehungen können sich Multiplizitäten befinden. Diese geben an, mit wie vielen Objekten der angebundenen Entität ein Objekt in Beziehung stehen kann.

An dieser Stelle würde sich eine Aufzählung aller Entitäten inklusive Attributen und Beziehungen zu sehr mit denen des UML-Diagramms überschneiden, sodass darauf verzichtet wird. Deswegen werden nur die relevanten Unterschiede zwischen den beiden Diagrammarten angegeben. Diese sehen wie folgt aus:

Als erstes sollte es offensichtlich sein, dass es viele oberflächliche, grafische Unterschiede gibt. Zum einen sind dies die bereits erwähnten Formen der Kästen, die sich zwischen den beiden Diagrammen unterscheiden. Daneben sind im ER-Diagramm viele Teile in eigene Kästen „ausgelagert“: So sind Attribute nicht direkt in dem Kästchen der Klasse bzw. Entität enthalten, sondern in eigenen Kästen angebunden. Die Beschreibung der Beziehung erhält außerdem eine eigene Raute, an die zugehörige Attribute einfach angebunden werden können. Im Falle von UML werden einfache Beschreibungen der Beziehung schlichtweg an die Linie geschrieben. Wenn Attribute an die Beziehung gehören, dann wird dafür eine eigene Klasse erstellt.

Des Weiteren gibt es eine geänderte Reihenfolge der Multiplizitäten. Dies liegt an der Syntax der beiden Diagrammtypen, in denen diese unterschiedlich definiert ist: Im ER-Diagramm steht die Kardinalität an der gegenüberliegenden Entität. Im UML-Diagramm ist dies genau anders herum.

Außerdem werden im ER-Diagramm keine Datentypen gezeigt. Im UML- Diagramm wird bei jedem Attribut gleich festgelegt, ob es z.B. ein Integer, String oder ein anderer Datentyp ist.

3 Integritätsbedingungen (Dominik)

Die Integritätsbedingungen sind Annahmen, die für die gegebenen Daten und die gegebene Struktur der entworfenen Datenbank aufgestellt werden, um fehlerhafte Zustände zu vermeiden. Zu jeder Zeit sind alle aufgestellten Bedingungen zu erfüllen. Zunächst werden in diesem Kapitel die Integritätsbedingungen informell aufgestellt und anschließend formal mit OCL und relationaler Algebra definiert.

3.1 Informelle Definition und Erläuterung

Die Integrietätsbedingungen werden pro Klasse aufgestellt. Da manche Integritätsbedingungen auch klassenübergreifend sind, werden diese meist der wichtigsten Klasse der Bedingung zugeordnet.

Ei

Da es keine Eier gibt, die sofort nach dem Einsammeln schlüpfen, muss die Distanz zum Schlüpfen eines Eies größer Null sein.

TypEffektiv

Beim Ausführen einer Attacke gegen ein Pokemon wird die Stärke der Attacke durch einen Faktor beeinflusst, der sich aus dem Typ der Attacke und dem Typ des verteidigenden Pokemon ergibt. Möglich ist eine Abschwächung der Stärke der Attacke um die Hälfte, eine Verstärkung um das Doppelte oder keine Beeinflussung. Der angewendete Multiplikator beträgt also entweder 0.5, 1.0 oder 2.0.

GefangenesPokemon

Für jedes gefangene Pokemon müssen mehrere Bedingungen erfüllt sein. Zum einen muss der zum Verstärken des Pokemon benötigte Sternenstaub größer Null sein. Zum anderen ist wichtig, dass der minimale Wert an Wettkampfpunkten von 10 eingehalten wird. Außerdem darf ein Pokemon nur Attacken beherrschen, die entweder vom Typ des Pokemon oder vom Typ Normal sind. Grundsätzlich darf auch kein Pokemon zweimal die gleiche Attacke beherrschen.

Attacke

Es ist nicht möglich, dass Attacken negativen Schaden machen. Ein Schaden von Null ist jedoch möglich, da es die nutzlose Attacke Platscher gibt.

Spieler

Jeder Spieler startet zu Beginn des Spiels auf Level Eins und besitzt keinen Sternenstaub. Eine Begrenzung für den Besitz an Sternenstaub gibt es nicht. Das

Level eines Spielers ist jedoch auf 40 limitiert. Außerdem startet jeder Spieler mit keinen Erfahrungspunkten und kann bis Level 40 20 Millionen Punkte sammeln. Die Ressource Bonbons hängt von der Anzahl der gefangenen Pokemon ab, denn pro Fang erhält der Spieler drei Bonbons. Lässt der Spieler ein Pokemon wieder frei, erhält er einen weiteren Bonbon. Aus diesem Zusammenhang ergibt sich, dass die vom Fangen und Freilassen erhaltenen Bonbons mit der Anzahl der Bonbons im Beutel plus der Anzahl schon in der Vergangenheit verbrauchten Bonbons übereinstimmen müssen.

Arena

Eine Arena hat ein Level von Null bis Zehn. Wichtig ist, dass alle in einer Arena platzierten Pokemon Spielern gehören, die dem gleichen Team angehören. Genau diesem Team gehört dann auch die Arena. Außerdem darf es nicht möglich sein, dass freigelassene Pokemon in einer Arena platziert werden.

Pokemon

Entwickelt sich ein Pokemon, entsteht daraus niemals das gleiche Pokemon wieder. Gibt es für ein Pokemon keine Entwicklung, aus der es entstehen könnte, ist es ein Basispokemon. Da es Bonbons immer nur für Basispokemon gibt, essen die Weiterentwicklungen auch die gleichen Bonbons wie das Basispokemon. Außerdem können aus Eiern nur Basispokemon schlüpfen. Ein Pokemon hat maximal zwei Typen, die aber nicht die selben sein dürfen.

ItemsGesammelt

Von einer Sorte Items besitzt ein Spieler entweder gar keine oder beliebig viele.

BonbonsGesammelt

Von einer Sorte Bonbons besitzt ein Spieler entweder gar keine oder beliebig viele. Auch die Anzahl an verbrauchten Bonbons liegt niemals unter Null.

Entwicklung

Die für eine Entwicklung benötigte Anzahl an Bonbons ist nicht Null und immer positiv.

3.2 Formale Definition mit OCL

Besonders hervorzuheben sind die Invarianten BonbonNumberCheck (Kontext Spieler), AttackTypesCheck (Kontext Gefangenes Pokemon) und ValidPokemonCheck (Kontext Arena), da diese sich jeweils über mehrere Klassen erstrecken.

Alle OCL Ausdrücke, die nur triviale numerische Vergleiche ($<$, $=$, $>$) enthalten werden hier nicht genauer erklärt.

Ei

```
context Ei
-- Eier-Distanz: Grö ß er 0
inv DistanzUeberNull:
    self.Distanz > 0
```

TypEffektiv

```
context TypEffektiv
-- Multiplikator Check
inv MultiplikatorCheck:
    self.Multiplikator = 0 or self.Multiplikator = 0.5 or
        self.Multiplikator = 1 or self.Multiplikator = 2
```

GefangenesPokemon

```
context GefangenesPokemon
-- Sternenstaub > 0
inv SternenstaubCheck:
    self.Sternenstaub > 0
-- Wettkampfpunkte > 10
inv WettkampfpunkteCheck:
    self.Level >= 10
```

Um die Typen der Attacken zu überprüfen, werden die Typen des Pokemon bestimmt und für jede Attacke überprüft, ob der Typ dieser mit einem Typ des Pokemon überstimmt oder vom Typ Normal ist.

```
-- Check if the attacks of a pokemon have the same type
    as the pokemon or the normal type
inv AttackTypesCheck:
    self.attacke->forAll(attacke | self.pokemon.typ->
        includes(attacke.typ) or attacke.typ.Name = '
        Normal')
```

Für jede Attacke des Pokemon wird überprüft, ob diese nur einmal unter den allen Attacken des Pokemon vorkommt.

```
-- Attacks of a pokemon cant be the same
inv NotEqualAttacks:
    self.attacke->forAll(attacke | self.attacke->count(
        attacke) = 1)
```

Attacke

```
context Attacke
-- Schaden >= 0
inv SchadenCheck:
    self.Schaden >= 0
```

Spieler

```
context Spieler
-- Sternenstaub >= 0
inv SternenstaubCheck:
    self.Sternenstaub >= 0
-- Level > 0 und <= 40
inv LevelCheck:
    self.Level > 0 and self.Level <= 40
-- Erfahrung >= 0
inv ErfahrungsCheck:
    self.Erfahrung >= 0 and self.Erfahrung <= 20000000
```

Für jeden Bonbontyp, den der Spieler gesammelt hat, wird zunächst bestimmt, welche der gefangenen Pokemon des Spielers diesen Bonbon essen. Diese Pokemon haben beim Fangen drei und beim Freilassen einen solchen Bonbon gegeben. Die Summe dieser so erhaltenen Bonbons wird pro Bonbontyp mit der Anzahl der gesammelten und verbrauchten Bonbons des Spielers verglichen.

```
-- Check if number of gotten bonbons is equal to the
   number of possible bonbons gotten from caught pokemon
inv BonbonNumberCheck:
    self.bonbonsGesammelt->forAll(gesammelterBonbon |
        let gefangenePokemon = gesammelterBonbon.bonbon.
            pokemon.gefangenesPokemon->excluding(Undefined
            )->select(spieler = self) in
                    gefangenePokemon->size * 3 +
                        gefangenePokemon->select(
                        Freigelassen)->size =
                    gesammelterBonbon.
                        AnzahlVerfuegbar +
                        gesammelterBonbon.
                        AnzahlVerbraucht)
```

Arena

```
context Arena
-- Level Check
inv ArenaLevelCheck:
    self.Level >= 0 and self.Level <= 10
```

Für jedes Pokemon einer Arena wird der Spieler, der dieses besitzt, und dessen Team bestimmt. Daraus ergibt sich eine Menge von Teams, die pro Arena aber nur ein Team beinhalten soll. Das ist genau das Team, dass in der Arena eingetragen ist.

```
-- Valid Pokemon Check
inv ValidPokemonCheck:
```

```
    self.gefangenesPokemon.spieler.team->forAll(team |
        team = self.team)
```

Für jedes in einer Arena platzierte Pokemon wird überprüft, ob dieses nicht schon freigelassen wurde.

```
-- Check if pokemon in arenas are still owned by players
inv NoFreePokemon:
    self.gefangenesPokemon->forAll(Freigelassen = false)
```

Pokemon

```
context Pokemon
```

Für alle möglichen Weiterentwicklungen des Pokemon wird überprüft, dass es nicht das gleiche Pokemon wieder ist.

```
-- Pokemon cant evolve into itself
inv Evolution:
    self.Weiterentwicklung->forAll(entwicklung |
        entwicklung <> self)
```

Für alle möglichen Weiterentwicklungen des Pokemon wird überprüft, ob diese die gleichen Bonbons wie das betrachtete Pokemon essen.

```
-- Evolutions eating same bonbons
inv EvolutionSameBonbons:
    self.Weiterentwicklung->forAll(entwicklung |
        entwicklung.bonbon = self.bonbon)
```

Wenn für ein Pokemon eine Basis angegeben ist, bedeutet dies, dass das Basispokemon sich zu diesem Pokemon entwickelt. In diesem Fall ist das betrachtete Pokemon eine Weiterentwicklung und darf deshalb nicht aus einem Ei schlüpfen.

```
-- Only basis Pokemon can hatch out of eggs
inv OnlyBasisPokemonFromEggs:
    if self.Basis <> Undefined then self.ei = Undefined
        else true endif
```

Für jeden Typ des Pokemon wird untersucht, ob dieser nur einmal in der Liste der Typen des Pokemon vorkommt.

```
-- Types of a pokemon cant be the same
inv NotEqualTypes:
    self.typ->forAll(typ | self.typ->count(typ) = 1)
```

ItemsGesammelt

```
context ItemsGesammelt
-- Number of items >= 0
inv NumberOfItemsCheck:
    self.Anzahl >= 0
```

BonbonsGesammelt

```
context BonbonsGesammelt
-- Number of bonbons >= 0
inv NumberOfBonbonsCheck:
    self.AnzahlVerbraucht >= 0 and self.AnzahlVerfuegbar
       >= 0
```

Entwicklung

```
context Entwicklung
-- Evolution needs bonbons
inv EvolutionBonbonNumberCheck:
    self.BonbonAnzahl > 0
```

3.3 Formale Definition mit relationaler Algebra

Um die Integrietätsbedingungen mit relationaler Algebra zu beschreiben, muss ein relationales Datenbankschema vorhanden sein, da die relationale Algebra auf diesem Schema aufbaut. Deshalb wird im Folgenden schon das relationale Datenbankschema verwendet, das erst in Kapitel 5 ausgearbeitet wird.

Da es sehr viele Primärschlüssel gibt, werden hier nicht alle Einzelfälle aufgeführt, sondern in allgemeiner Form definiert, wie diese in relationaler Algebra angegeben. $\sigma_{R1.a=R2.a \wedge (R1.b_1 \neq R2.b_1 \vee \ldots \vee R1.b_n \neq R2.b_n)}(\delta_{R1}(R) \times \delta_{R2}(R)) = \emptyset$
Dabei ist R die Relation und der Primärschlüssel a. Die restlichen Attribute der Relation R sind b_1 bis b_n.

Möglich ist auch die Benutzung der Projektion, da diese doppelte Elemente nicht selektiert. Sollte die Menge nach der Projektion auf den Primärschlüssel kleiner werden, wurde der Primärschlüssel nicht eindeutig verwendet.
$|\pi_a(R)| = |R|$

Zu beachten ist bei relationaler Algebra, dass keine Null-Referenz vorhanden ist und deshalb Terme meist negiert und insgesamt auf die leere Menge geprüft werden.

Für Übersetzung der aufgestellten Integritätsbedingungen werden im Folgenden Relationen durch die Bildung des Produktes $\times$ verbunden. Bedingungen an einen solchen Verbund werden per Selektion σ vorgenommen. Außerdem werden per Projektion teils Relationen umbenannt, um einen Verbund aus zwei gleichen Relationen zu ermöglichen.

Ei

DistanzUeberNull:
$\sigma_{Distanz \leq 0}(EI) = \emptyset$

TypEffektiv

MultiplikatorCheck:
$\sigma_{Multiplikator \neq 0.5 \wedge Multiplikator \neq 1.0 \wedge Multiplikator \neq 2.0}(TypEffektiv) = \emptyset$

GefangenesPokemon

SternenstaubCheck:
$\sigma_{Sternenstaub \leq 0}(GefangenesPokemon) = \emptyset$

WettkampfpunkteCheck:
$\sigma_{Level < 10}(GefangenesPokemon) = \emptyset$

AttackTypesCheck:
$\sigma_{(Attacke.TypName \neq istVonTyp.TypName \vee Attacke.TypName = 'Normal') \wedge kannVerwenden.AttackeName = Attacke.Name}$
$(Attacke \times \sigma_{GefangenesPokemon.ID = kannVerwenden.GefangenesPokemonID}$
$(kannVerwenden \times \sigma_{GefangenesPokemon.PokedexID = istVonTyp.PokedexID}$
$(GefangenesPokemon \times istVonTyp))) = \emptyset$

NotEqualAttacks:
$|\pi_{GefangenesPokemonID, AttackeName}(kannVerwenden)| = |kannVerwenden|$

Attacke

SchadenCheck:
$\sigma_{Schaden < 0}(Attacke) = \emptyset$

Spieler

SternenstaubCheck:
$\sigma_{Sternenstaub < 0}(Spieler) = \emptyset$

LevelCheck:
Im relationalen Schema nicht mehr vorhanden.

ErfahrungsCheck:
$\sigma_{Erfahrung < 0 \vee Erfahrung > 20000000}(Spieler)$

BonbonNumberCheck:
In folgender Bedingung wird zur Übersichtlichkeit von dem bisherigen Notationsschema abgewichen und es werden mehrere Gleichungen aufgestellt. Dies ergibt sich daraus, dass am Ende zwei Summen berechnet und verglichen werden. Auf beiden Seiten der Gleichung werden die gleichen Selektionen benötigt, sodass diese zunächst in Variablen abgelegt werden, um Wiederholungen zu vermeiden.

$R = \sigma_{Spieler.Name=BonbonsGesammelt.SpielerName}(Spieler \times BonbonsGesammelt)$
$gefangenePokemon = \sigma_{Pokemon.PokedexID=GefangenesPokemon.PokedexID}$
$(GefangenesPokemon \times \sigma_{Bonbon.Name=Pokemon.BonbonName}$
$(Pokemon \times \sigma_{Bonbon.Name=R.BonbonName}(Bonbon \times R)))$
$|\sigma_{Freigelassen=true}(gefangenePokemon)| + |gefangenePokemon| =$
$R.AnzahlVerfuegbar + R.AnzahlVerbraucht$

Arena

ArenaLevelCheck:
$\sigma_{Level<0 \vee Level>10}(Arena) = \emptyset$

ValidPokemonCheck:

$\sigma_{Spieler.TeamName=Team.Name \wedge Team.Name \neq Arena.TeamName}$
$(Team \times \sigma_{GefangenesPokemon.SpielerName=Spieler.Name}$
$(Spieler \times \sigma_{Vertidiger.gefangenesPokemonID=GefangenesPokemon.ID}$
$(GefangenesPokemon \times \sigma_{Arena.OrtID=Verteidiger.OrtID}$
$(Arena \times Verteidiger)))) = \emptyset$

NoFreePokemon:

$\sigma_{Vertidiger.gefangenesPokemonID=GefangenesPokemon.ID \wedge GefangenesPokemon.Freigelassen=true}$
$(GefangenesPokemon \times \sigma_{Arena.OrtID=Verteidiger.OrtID}$
$(Arena \times Verteidiger)) = \emptyset$

Pokemon

Evolution:
$\sigma_{PokedexIDBasis=PokedexIDWeiterentwicklung}(Entwicklung) = \emptyset$

EvolutionSameBonbons:

$\sigma_{Pokemon2.PokedexID=Entwicklung.PokedexIDWeiterentwicklung \wedge Pokemon.BonbonName \neq Pokemon2.BonbonName}$
$(\delta_{Pokemon2}(Pokemon) \times \sigma_{Pokemon.PokedexID=Entwicklung.PokedexIDBasis}$
$(Pokemon \times Entwicklung)) = \emptyset$

OnlyBasisPokemonFromEggs:

$\sigma_{Bonbon.Name=Pokemon2.BonbonName}(Bonbon \times \sigma_{Pokemon2.PokedexID=Entwicklung.PokedexIDWeiterentwicklung}$
$(\delta_{Pokemon2}(Pokemon) \times \sigma_{Pokemon.PokedexID=Entwicklung.PokedexIDBasis}$
$(Pokemon \times Entwicklung))) = \emptyset$

NotEqualTypes:
$|\pi_{PokedexID,TypName}(istVonTyp)| = |istVonTyp|$

ItemsGesammelt

NumberOfItemsCheck:
$\sigma_{ItemAnzahl<0}(ItemsGesammelt) = \emptyset$

BonbonsGesammelt

NumberOfBonbonCheck:
$\sigma_{AnzahlVerfuegbar<0 \vee AnzahlVerbrauch<0}(BonbonsGesammelt) = \emptyset$

Entwicklung

EvolutionBonbonNumberCheck:
$\sigma_{BonbonAnzahl<0}(Entwicklung) = \emptyset$

3.4 Konsistenz, Vollständigkeit und Unabhängigkeit

Betrachtet wird die Gesamtheit der Integritätsbedingungen in Bezug auf ihre Konsistenz, Vollständigkeit und Unabhängigkeit. Es wird also geklärt, ob alle für einen Betrieb der Datenbank nötigen Bedingungen aufgestellt wurden und nicht der Fall eintreten kann, dass sich mehrere Bedingungen gegenseitig ausschließen. In einer solchen Konstellation kann es schnell passieren, dass die Datenbank inkonsistent wird oder es unmöglich wird, neue Daten einzufügen.

Vollständig sind die ausgearbeiteten Integrietätsbedingungen wahrscheinlich nicht, da für diese Ausarbeitung einige Beispiele gewählt wurden, um interessante, komplizierte Beziehungen und deren Einschränkungen aufzuzeigen. Außerdem wurden natürlich auch viele grundsätzliche Bedingungen aufgenommen, aber auch einige vernachlässigt. Dies bedeutet also, dass durch die gegebene Menge an Integritätsbedingungen ein sicherer Betrieb einer Datenbank nicht vollständig gewährleistet ist.
Etwas entschärft wird dieses Problem dadurch, dass in den meisten Anwendungsfällen nicht nur die Datenbank für die Datenkonsistenz sorgt, sondern auch schon durch die Anwendungssoftware manche Eingaben gar nicht möglich sind und viele Fehlerfälle abgefangen werden. Am sichersten ist es natürlich, wenn die datensichernde Instanz jegliches Fehlverhalten ausschließt!

Durch Entitäten übergreifende Integritätsbedingungen werden außerdem Abhängigkeiten geschaffen, die vorgeben, in welcher Reihenfolge Daten aktualisiert werden müssen, damit stets ein zulässiger Zustand gegeben ist.

Zu einer Überschneidung der aufgestellten Integrietätsbedingungen kommt es nicht, da die meisten jeweils nur eine einzige Entität betreffen und die übrigen übergreifenden Bedingungen keine Gemeinsamkeiten aufweisen. Letztere sind immer in sich abgeschlossene „Kreise“, die ganz verschiedene Situationen absichern.

4 Systemzustände vom Modell (Jan)

4.1 Valider Systemzustand

Abbildung 3 zeigt einen beispielhaften korrekten Systemzustand der Datenbank. Die Rollen wurden hierbei größtenteils ausgeblendet, um Übersichtlichkeit zu bewahren. Nur an relevanten Stellen sind diese eingeblendet, dazu zählt z.B. die Effektivität einer Attacke, die sich aus dem Typ des Angreifers(Rolle 1) und Verteidigers(Rolle 2) bestimmen lässt.
Der dargestellte Systemzustand ist ein einfaches Beispiel, in dem alle relevanten Tabellen befüllt sind, aber pro Tabelle nur ein Minimum an Datensätzen vorhanden ist. Dies soll einen Überblick über das gesamte System geben, dabei aber nicht zu überfüllt wirken, sodass schnell verständlich wird wie die Datenbank funktioniert.

In diesem Systemzustand gibt es einen einzigen Spieler mit dem Namen „DoDoSt“. Dieser ist Level 28, gehört zu Team Gelb und besitzt bestimmte Items (99 Tränke, 42 Pokébälle etc.), Eier für verschiedene Distanzen, Bonbons und Pokémon. Es gibt 5 verschiedene Objekte des Typs Pokémon (Würde man korrekt die gesamte Datenbank darstellen, dann müssten davon 150 erzeugt werden). Diese essen bestimmte Bonbons, besitzen Typen und können sich teilweise in ein anderes Pokémon entwickeln. Von diesen Pokémon Objekten erben Objekte des Typs GefangenesPokemon, die vom Spieler gefangene Pokémon darstellen. Im gezeigten Beispiel sind dies ein Bisasam mit dem Level 521 und ein anderes Bisasam mit dem Level 425, das der Spieler bereits wieder frei gelassen hat. Diese beiden gefangenen Pokémon besitzen als Attacken jeweils Rankenhieb und Tackle, die vom Typ Pflanze beziehungsweise Normal sind. Als Typen gibt es in dem Beispiel Feuer, Wasser, Pflanze und Normal. Die Wechselwirkungen / Effektivität zwischen den Typen wird über ein Beziehungsobjekt „TypEffektiv“ geregelt. Für den Fall Feuer und Pflanze bedeutet dies, dass solch ein Objekt für Angreifer=Feuer und Verteidiger=Pflanze mit dem Multiplikator 2 („sehr effektiv“) erzeugt wird, und für den umgekehrten Fall mit dem Multiplikator 0,5 („nicht sehr effektiv“).
Außerdem sehen wir in der Abbildung 3, dass eine Arena existiert. Diese gehört Team Gelb, dem auch der Spieler angehört. Weil er diesem Team angehört, ist es ihm möglich dort ein Pokémon zur Verteidigung einzusetzen, was er mit einem seiner gefangenen Bisasam auch macht.

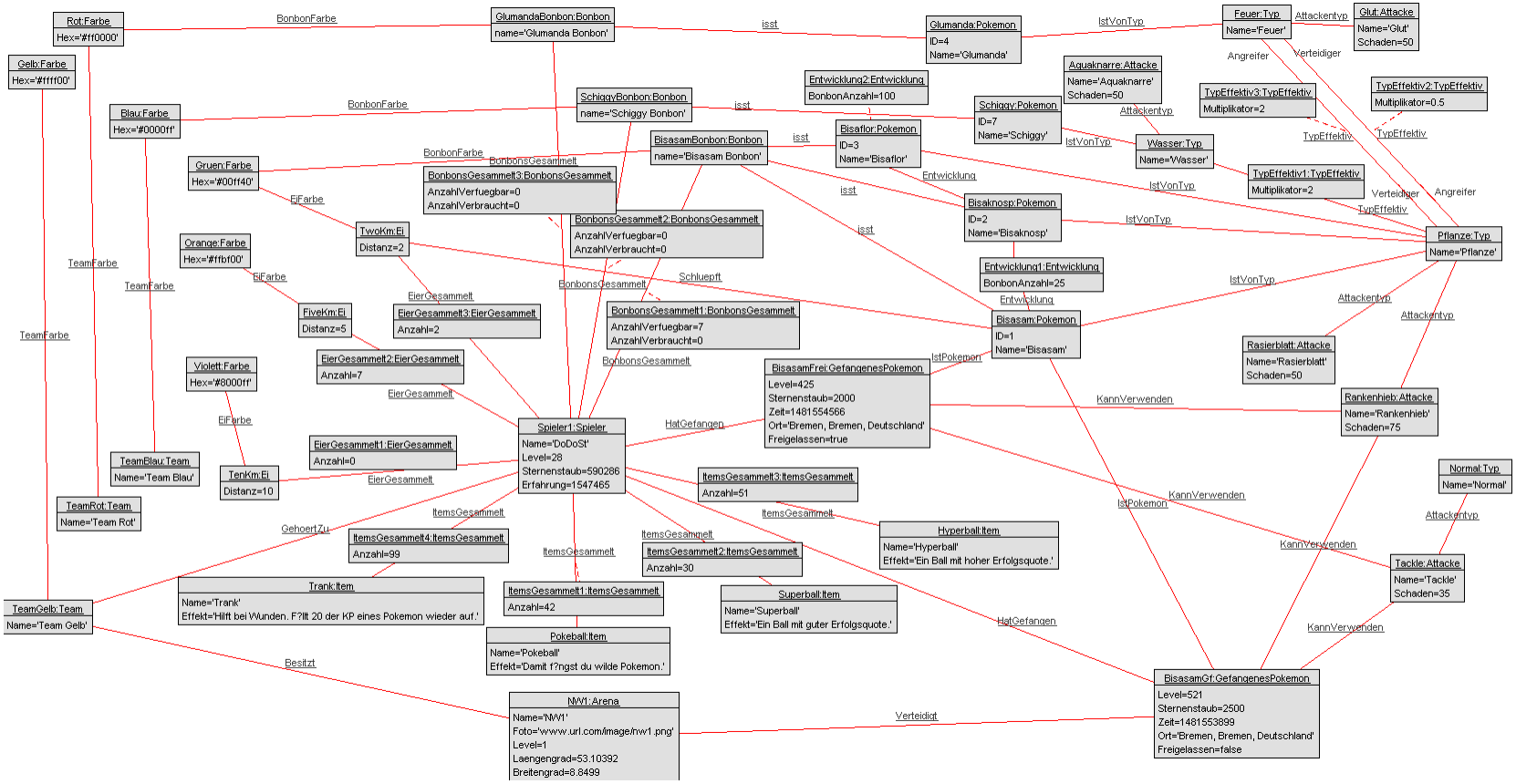

Abbildung 3: Systemzustand

4.2 Invalide Systemzustände

Neben dem in 4.1 gezeigten korrekten Systemzustand folgen nun ein paar inkorrekte Zustände, die jeweils eine bestimmte Invariante verletzten. Da vom Umfang her nicht alle Invarianten überprüft werden können, wird dies an drei Beispielen für die kompliziertesten Invarianten gezeigt. Um Übersichtlichkeit zu bewahren, werden in den Bildern nur die relevanten Objekte gezeigt, in Wirklichkeit müssen natürlich mehr vorhanden sein um alle Invarianten zu erfüllen.

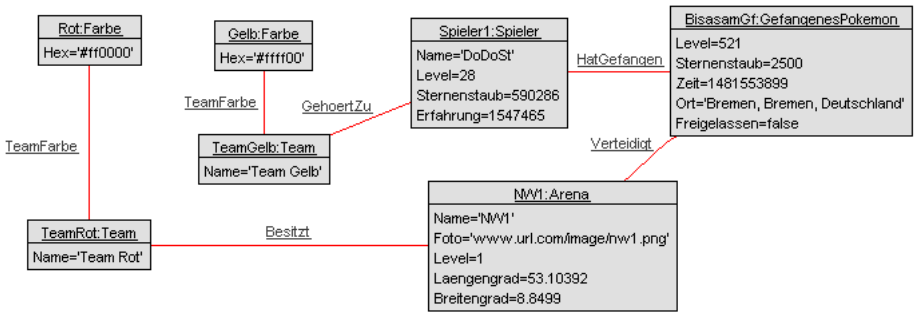

Abbildung 4: Invalider Zustand 1

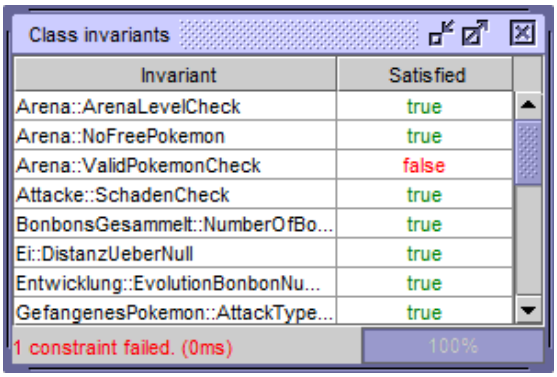

Abbildung 5: ValidPokemonCheck verletzt

In Abbildung 4 und 5 ist ein Beispiel gezeigt, wie ein invalider Systemzustand aussehen kann. Dabei wurde die Invariante „ValidPokemonCheck“ verletzt, die prüft, ob sich in einer Arena nur Pokemon befinden, deren Trainer aus dem gleichen Team stammt, dem auch die Arena gehört. Konkret bedeutet dies im Beispiel, dass die Arena im Besitz von Team Rot ist, aber der Spieler, der ein Pokemon dort zur Verteidigung einsetzt zu Team Gelb gehört. Dadurch wird

die Invariante verletzt, was vom System auch korrekt erkannt wird.

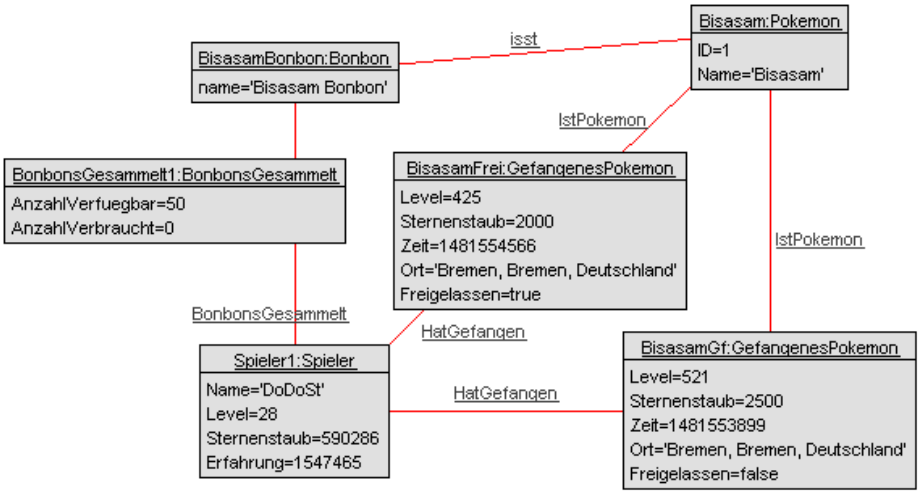

Abbildung 6: Invalider Zustand 2

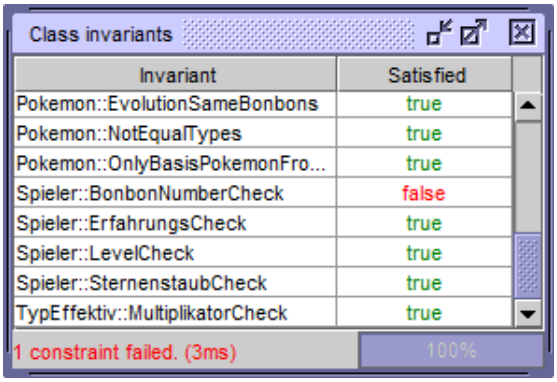

Abbildung 7: BonbonNumberCheck verletzt

Ein weiterer invalider Systemzustand ist in den Abbildungen 6 und 7 gezeigt. Hierbei wird die Invariante BonbonNumberCheck verletzt, die sicherstellen soll, dass ein Spieler eine korrekte Anzahl an Bonbons einer Art besitzt, da sich diese aus den gefangenen und freigelassenen Pokemon berechnen lässt. Korrekt wäre eine Anzahl von 7 verfügbaren Bisasam-Bonbons (2 mal 3 für die gefangenen Bisasam, plus 1 für das freigelassene Bisasam), allerdings beträgt diese im Beispiel 50. Diese Diskrepanz wird beim Invarianten-Check korrekt erkannt, wie

man in Abbildung 7 erkennen kann.

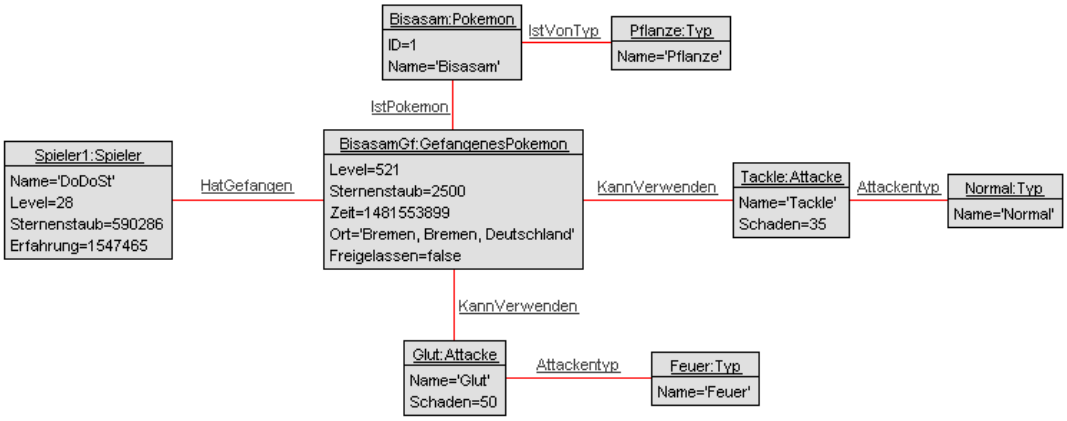

Abbildung 8: Invalider Zustand 3

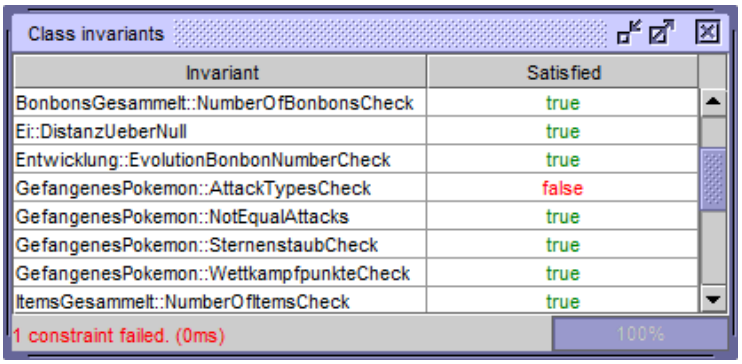

Abbildung 9: AttackTypesCheck verletzt

Der letzte invalide Zustand zeigt, wie die Invariante AttackTypesCheck verletzt wird. Diese soll sicherstellen, dass Pokemon nur Attacken ihres eigenen Typs oder vom Typ Normal beherrschen. Im Beispiel (Abbildung 8) wird dies verletzt, da ein vom Spieler gefangenes Bisasam, das den Typ Pflanze besitzt, die Attacke Glut beherrscht. Diese Attacke ist vom Typ Feuer und sollte deswegen vom Pokemon nicht eingesetzt werden können. Wie man in Abbildung 9 erkennt, wird auch dieses Problem bei der Prüfung der Invarianten erkannt.

5 Realisierung als Relationales DB-Schema (Jendrik)

5.1 Anwendung des Standardverfahrens

Um nun das konzeptionelle DB-Schema (Kapitel 2.3) in ein relationales DB-Schema zu übersetzen, wird das Standardverfahren eingesetzt. Dieses umfasst folgende Schritte:

- **Entitytyp → RS (Relationales Schema)**
  `Übersetzen aller Entitäten (z.B. Pokemon) in eine relationale Form (A(a1, a2, a3))`

- **Beziehungen → RS**
  `Übersetzen aller Relationen wie „gehört zu" in ein relationales Schema. Dabei werden die Schlüsselattribute der Entitäten einer jeden Referenz als Fremdschlüssel übernommen.`

- **Optimierung**
  `Zusammenfassung, Streichen und Umbenennen der Entitäten und deren Attribute`

Außerdem haben wir uns dazu entschieden an dieser Stelle einige Designentscheidungen bezüglich Primärschlüssel, Klassen und Relationen zu erklären und zu begründen.

5.1.1 Übersetzung: Entitytypen

Wir beginnen die Übersetzung des ER-Modells in ein konzeptionelles Datenbank-Schema nach Standardverfahren mit der Übersetzung der Entitäten in ein relationales Datenbank-Schema. Funktionale Beziehungen sind hier bereits integriert, da sie innerhalb des Schemas der Entitäten umgesetzt werden können und somit kein eigenes Relationsschema benötigen.

RS1: `Pokemon(PokedexID, Name, BonbonName, EiID)`
Die `isst` Beziehung wird hier in `Pokemon` integriert, da es sich um eine 1:1..* Beziehung handelt und somit jedes Pokemon nur einen bestimmten `Bonbon` isst. Auf das Hinzufügen eines Attributes „Basis Pokemon“ wurde hier verzichtet, da dies zu Redundanz führen würde (vgl. 2.3 Entwicklung).
Erläuterung: Der Primärschlüssel ist hier der Pokedex Eintrag, da jedem Pokemon per Definition genau eine eindeutige Pokedex-Nummer zugewiesen ist.

RS2: `Bonbon(Name, FarbeName)`
Erläuterung: Da es keine zwei Bonbons mit dem selben Namen geben darf, haben wir uns dazu entschieden diesen als Primärschlüssel zu wählen.

RS3: `Typ(Name)`
Erläuterung: vgl. Bonbon

RS4: `Attacke(Name, Schaden, TypName)`
Um die totale funktionale Beziehung `AttackenTyp` auszudrücken, die angibt, von welchem Typ eine Attacke ist, wird das Fremdschlüssel-Attribut `TypName` genutzt, dass auf einen bestimmten Typ referenziert.
Erläuterung: vgl. Bonbon

RS5: `Farbe(Name, Farbwert)`
Erläuterung: vgl. Bonbon

RS6: `Spieler(Name, Level, Erfahrung, Sternenstaub, TeamName)`
Um jedem Spieler genau ein Team zuzuordnen (total funktional) wird mit einem Fremdschlüssel auf das zugeordnete Team referenziert (`gehört zu`).
Erläuterung: vgl. Bonbon

RS7: `Team(Name, FarbeName)`
Erläuterung: vgl. Bonbon

RS8: `Arena(Breitengrad, Längengrad, Name, Level, Foto, TeamName)`
Die Beziehung `besitzt` wird aufgrund der zugrunde liegenden 1:1..* Kardinalität durch einen Fremdschlüssel als Attribut der Arena realisiert.
Erläuterung: Es sind Arenen mit gleichem Namen, Level etc. erlaubt, falls diese sich an zwei verschiedenen Orten befinden. Aus diesem Grund wurde ein kombinierter Primärschlüssel aus Längen und Breitengrad gewählt.

RS9: `Ei(ID, Distanz, FarbeName)`
Erläuterung: Da ein `Ei` keinen eindeutigen Namen oder ähnliches besitzt wird auf eine eindeutige `ID` zurückgegriffen.

RS10: `Item(Name, Effekt)`
Erläuterung: vgl. Bonbon

RS11: `GefangenesPokemon(ID, SpielerName, PokedexID, Ort, Zeit, Sternenstaub, Wettkampfpunkte, Freigelassen)`
Die Beziehung `HatGefangen` lassen sich aufgrund der 1:1..* Kardinalität über einen Fremdschlüssel als Attribut von `GefangenesPokemon` darstellen. Da GefangenesPokemon eine Spezialisierung von Pokemon mit der Kardinalität von 1:1..* ist, wird diese über einen Fremdschlüssel als Attribut von GefangenesPokemon dargestellt.
Erläuterung: Da diese Klasse die Pokemon repräsentiert, die ein Spieler gefangen hat und ein Spieler mehrere (auch gleiche) Pokemon besitzen kann ist hier eine eigenständige ID als Primärschlüssel notwendig.

5.1.2 Übersetzung: Beziehungen

RS12: `istVonTyp(PokedexID,TypName)`
Die Beziehung `istVonTyp` realisiert die Zuordnung eines Pokemons zu einem oder mehrerer (maximal zwei) Typen.

Erläuterung: Sowohl PokedexID als auch TypName sind Primärschlüssel, um die mehrfache Zuordnung eines Pokemons zu einem Typen zu verhindern.

RS13: `kannVerwenden(GefangenePokemonID,AttackeName)`
Realisiert die Zuordnung der gefangenen Pokemon zu den von diesen erlernten Attacken. Ein Pokemon kann dabei maximal zwei Attacken beherrschen.
Erläuterung: Ein Pokemon kann nur einmal die selbe Attacke erlernen.

RS14: `EierGesammelt(SpielerName, EierID, Eier#)`
Stellt gewissermaßen die gesammelten `Eier` eines Spielers da. Die Relation beinhaltet auch die Anzahl der von `Spieler` gesammelten `Eier`.
Erläuterung: Die Anzahl, die ein Spieler von einem speziellen `Ei` besitzt, soll in dem entsprechenden Attribut festgehalten werden. Aus diesem Grund darf die gleiche Kombination von `Spieler` und `Ei` nicht zwei mal vorkommen.

RS15: `Verteidiger(ArenaLängengrad, ArenaBreitengrad, gefangenePokemonID)`
Stellt die `Verteidigt` Beziehung und somit die von Spielern gefangenen Pokemon dar, die momentan in den verschiedenen Arenen als Verteidigung platziert sind.
Erläuterung: Ein Pokemon darf nur in einer Arena gleichzeitig als Verteidiger eingesetzt werden. Aus diesem Grund wurde Verteidiger als Spezialisierung von `gefangenePokemon` mit einer 1:1 Kardinalität umgesetzt.

RS16: `ItemsGesammelt(ItemName, SpielerName, Item#)`
Diese Relation stellt die Items dar, die bereits vom Spieler gesammelt wurden.
Erläuterung: vgl. EierGesammelt

RS17: `Entwicklung(PokedexIDBasis, PokedexIDWeiterentwicklung, Bonbon#)`
Die Entwicklung von einem Pokemon in ein anderes benötigt eine Anzahl an Bonbons und ist hier als Relation dargestellt. Umbenennung aufgrund der Unterscheidbarkeit der Attribute.
Erläuterung: Identische Entwicklungen sollten vermieden werden. Dies passiert hier durch die Primärschlüssel.

RS18: `TypEffektivität(TypNameAngreifer, TypNameVerteidiger, Multiplikator)`
Beziehung, die die Effektivität der Typen von Pokemon im Kampf gegeneinander mit Hilfe eines Multiplikators angibt. Umbennenung aufgrund von Unterscheidung der zwei Attribute. **Erläuterung:** Die Wahl der Primärschlüssel verhindert das Auftreten von gleichen Kombinationen.

RS19: `BonbonsGesammelt(BonbonName, SpielerName, AnzahlVerfügbar, Anzahlverbraucht)`
Diese Relation stellt die Bonbons dar, die bereits vom Spieler gesammelt/ verbraucht wurden.
Erläuterung: vgl. EierGesammelt

5.2 Besonderheiten der Schema-Übersetzung

Einige Besonderheiten der Schema-Übersetzung, sowie einige Erläuterungen und Erklärungen wurden bereits direkt unter den jeweiligen Übersetzungen preisgegeben. Dennoch gibt es ein paar Eigenheiten, die hier gesammelt für alle oben getätigten Übersetzungen angemerkt werden.
Abseits von den in Kapitel 2.3 definierten Vorgehensweisen werden im folgenden einige Vorgehensweisen erläutert.

Umbenennen von Attributen der Entitäten
Da wir meist Namen oder Bezeichnungen als Primärschlüssel gewählt haben, mussten im letzten Schritt der Schema-Übersetzung viele der Fremdschlüssel umbenannt werden, um eine eindeutige Zuordnung zu gewährleisten (vgl. Kapitel 2.3). Dies passierte immer nach dem selben Schema: [Name der Schemas/-Klasse] + [Name des Attributs].
Sollte eine Umbenennung der Fremdschlüssel Attribute aufgrund von Unterscheidung zweier gleicher Typen nötig sein, wird nach einem anderen Schema umbenannt ([AttributName] + [Bezeichner]). Hierbei wird immer nach Camel Case vorgegangen

Übersetzung: Rekursive Relation
Die Rekursive Relation `Entwicklung` wurde, angelehnt an die n zu m Relation, nach dem selben Prinzip umgesetzt.

Übersetzung: 1:n
Alle total funktionalen 1 zu 1..* Beziehungen wurden als Fremdschlüssel realisiert. Somit wird für eine solche Art von Beziehung kein extra RelationsSchema angelegt. Anwendung fand dies z.B. bei `Pokemon` an der Beziehung `isst`.

Übersetzung: n:m (+ Attribut)
Relationen mit der Kardinalität von n zu m wurden über ein zusätzliches Relationsschema umgesetzt. Mit dem Relationsschema kann die n zu m Relation wie eine 1 zu n Relation behandelt werden. Als Primärschlüssel wurden dabei immer die beiden Fremdschlüssel verwendet, die die m zu n Relation begründen.
Übersetzung: Vererbung/Generalisierung/Spezialisierung
Das Datenbankmodell enthält zwei verschiedene Spezialisierungen von denen eine die 1:n und die andere die 1:1 Kardinalität besitzt. Bei der Übersetzung wurde die 1:1 mit den Fremdschlüsseln als Primärschlüssel und die andere mit dem Fremdschlüssel als normales Fremdschlüssel-Attribut umgesetzt.

5.3 Übersetzung der Integritätsbedingungen (Dominik, Jendrik)

Farbe

```
CREATE TABLE Farbe
(
    Name varchar(255) PRIMARY KEY,
    Farbwert varchar(7) NOT NULL,
    CHECK (LENGTH(Farbwert) = 7)
)
```

Ei

```
CREATE TABLE Ei
(
    Distanz int PRIMARY KEY,
    FarbeName varchar(255) NOT NULL REFERENCES Farbe(Name
        ),
    CHECK (Distanz > 0)
)
```

Bonbon

```
CREATE TABLE Bonbon
(
    Name varchar(255) PRIMARY KEY,
    FarbeName varchar(255) NOT NULL REFERENCES Farbe(Name
        ),
    CHECK (Name ~* '^[a-z]*\$')
)
```

Typ

```
CREATE TABLE Typ
(
    Name varchar(255) PRIMARY KEY,
    CHECK (Name ~* '^[a-z]*\$')
)
```

Attacke

```
CREATE TABLE Attacke
(
    Name varchar(255) PRIMARY KEY,
    Schaden int NOT NULL,
    TypName varchar(255) NOT NULL REFERENCES Typ(Name),
    CHECK (Name ~* '^[a-z]*\$'),
```

```
    CHECK (Schaden >= 0)
)
```

Team

```
CREATE TABLE Team
(
    Name varchar(255) PRIMARY KEY,
    FarbeName varchar(255) NOT NULL REFERENCES Farbe(Name
        ),
    CHECK (Name ~* '^[a-z ]*\$')
)
```

Item

```
CREATE TABLE Item
(
    Name varchar(255) PRIMARY KEY,
    Effekt varchar(255) NOT NULL,
    CHECK (Name ~* '^[a-z]*\$')
)
```

Ort

```
CREATE TABLE Ort
(
    OrtID int PRIMARY KEY,
    Längengrad real NOT NULL,
    Breitengrad real NOT NULL,
    CHECK (Längengrad >= -180 AND Längengrad <= 180),
    CHECK (Breitengrad >= -85 AND Breitengrad <= 85)
)
```

Arena

```
CREATE TABLE Arena
(
    OrtID int PRIMARY KEY REFERENCES Ort(OrtID),
    Name varchar(255) NOT NULL,
    Level int NOT NULL,
    Foto varchar(255),
    TeamName varchar(255) REFERENCES Team(Name),
    CHECK (Name ~* '^[a-z]*\$'),
    CHECK (Level >= 0 AND Level <= 10)
)
```

Pokemon

```
 CREATE TABLE Pokemon
(
    PokedexID int PRIMARY KEY,
    Name varchar(255) NOT NULL,
    BonbonName varchar(255) REFERENCES Bonbon(Name),
    EiDistanz int REFERENCES Ei(Distanz),
    CHECK (Name ~* '^[a-z]*\$'),
    CHECK (PokedexID > 0)
)
```

EntwicklungBonBon

```
 CREATE TABLE EntwicklungBonBon
(
    ID int PRIMARY KEY,
    BonbonAnzahl int NOT NULL,
    CHECK (BonbonAnzahl > 0)
)
```

istVonTyp

```
CREATE FUNCTION MaxTwoAttacks(pPokedexID integer)
RETURNS boolean AS $$
DECLARE
    count integer;
BEGIN
        SELECT count(*) into count FROM istVonTyp WHERE
            PokedexID = pPokedexID;
        RETURN count < 2;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE istVonTyp
(
    PokedexID int REFERENCES Pokemon(PokedexID),
    TypName varchar(255) REFERENCES Typ(Name),
    PRIMARY KEY (PokedexID, TypName),
    CHECK (MaxTwoAttacks(PokedexID))
);
```

TypEffektiv

```
 CREATE TABLE TypEffektiv
(
    TypNameAngreifer varchar(255) REFERENCES Typ(Name),
    TypNameVerteidiger varchar(255) REFERENCES Typ(Name),
    Multiplikator real NOT NULL,
```

```
    PRIMARY KEY (TypNameAngreifer, TypNameVerteidiger),
    CHECK (Multiplikator = 0 OR Multiplikator = 0.5 OR
        Multiplikator = 1 OR Multiplikator = 2)
)
```

Entwicklung

```
CREATE FUNCTION OnlyBasisPokemonFromEggs(
    WeiterentwicklungID integer)
RETURNS integer AS $$
DECLARE basic integer;
BEGIN
    SELECT count(*) into basic FROM Pokemon WHERE
        PokedexID = WeiterentwicklungID AND EiDistanz IS
        NULL;
        RETURN basic;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION EvolutionSameBonbons(WeiterentwicklungID
    integer, BasisID integer)
RETURNS boolean AS $$
DECLARE
        basic integer;
        bonbon varchar(255);
BEGIN
        SELECT BonbonName into bonbon FROM Pokemon WHERE
            PokedexID = WeiterentwicklungID;
        SELECT count(*) into basic FROM Pokemon WHERE
            PokedexID = BasisID AND BonbonName = bonbon;
        RETURN basic = 1;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE Entwicklung
(
    PokedexIDBasis int REFERENCES Pokemon(PokedexID),
    PokedexIDWeiterentwicklung int REFERENCES Pokemon(
        PokedexID),
    EntwicklungBonbonID int REFERENCES EntwicklungBonBon(
        ID),
    PRIMARY KEY (PokedexIDBasis,
        PokedexIDWeiterentwicklung),
    CHECK (PokedexIDBasis != PokedexIDWeiterentwicklung),
    CHECK (OnlyBasisPokemonFromEggs(
        PokedexIDWeiterentwicklung)),
```

```
    CHECK (EvolutionSameBonbons(
        PokedexIDWeiterentwicklung, PokedexIDBasis))
);
```

Spieler

```
CREATE TABLE Spieler
(
    Name varchar(255) PRIMARY KEY,
    Erfahrung int,
    Sternenstaub int,
    TeamName varchar(255) NOT NULL REFERENCES Team(Name),
    CHECK (Erfahrung > 0 AND Erfahrung <= 20000000),
    CHECK (Sternenstaub > 0)
);
```

GefangenesPokemon

```
 CREATE TABLE GefangenesPokemon
(
    ID int PRIMARY KEY,
    SpielerName varchar(255) NOT NULL REFERENCES Spieler(
        Name),
    PokedexID int NOT NULL REFERENCES Pokemon(PokedexID),
    OrtID int NOT NULL REFERENCES Ort(OrtID),
    Zeit date NOT NULL,
    Sternenstaub int,
    Wettkampfpunkte int,
    Freigelassen boolean NOT NULL,
    CHECK (Wettkampfpunkte >= 10),
    CHECK (Sternenstaub > 0)
);
```

Verteidiger

```
CREATE FUNCTION MaximalZehn(ArenaID integer)
RETURNS boolean AS $$
DECLARE
    countDefender integer;
BEGIN
    SELECT count(*) into countDefender FROM Verteidiger
        WHERE OrtID = ArenaID;
         RETURN countDefender < 10;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION ValidPokemonCheck(ArenaID integer,
    GefangenesPokemonID integer)
```

```
RETURNS boolean AS $$
DECLARE
    team varchar(255);
    team2 varchar(255);
BEGIN
    SELECT TeamName into team FROM Arena WHERE OrtID =
        ArenaID;
    SELECT TeamName into team2 FROM Spieler WHERE name IN
         (SELECT SpielerName FROM GefangenesPokemon WHERE
        ID = GefangenesPokemonID);
         RETURN team = team2;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION NoFreePokemon(GefangenesPokemonID integer
    )
RETURNS boolean AS $$
DECLARE
    bool boolean;
BEGIN
    SELECT Freigelassen into bool FROM GefangenesPokemon
        WHERE ID = GefangenesPokemonID;
         RETURN bool = false;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE Verteidiger
(
         OrtID int REFERENCES Arena(OrtID),
         GefangenesPokemonID int REFERENCES
             GefangenesPokemon(ID),
         CHECK (MaximalZehn(OrtID)),
         CHECK (ValidPokemonCheck(OrtID,
             GefangenesPokemonID)),
         CHECK (NoFreePokemon(GefangenesPokemonID))
);
```

kannVerwenden

```
CREATE FUNCTION AttackTypesCheck(GefangenesPokemonID
    integer, AttackeName varchar(255))
RETURNS boolean AS $$
DECLARE
         attackTyp varchar(255);
    count integer;
BEGIN
```

```
        SELECT TypName into attackTyp FROM Attacke WHERE
            Name = AttackeName;
        SELECT count(*) into count FROM IstVonTyp
    WHERE PokedexID = GefangenesPokemonID
    AND TypName = attackTyp;
        RETURN count = 1 OR attackTyp = 'Normal';
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION OnlyTwoAttacks(pGefangenesPokemonID
    integer)
RETURNS boolean AS $$
DECLARE
    count integer;
BEGIN
        SELECT count(*) into count FROM kannVerwenden
            WHERE GefangenesPokemonID =
            pGefangenesPokemonID;
        RETURN count < 2;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE kannVerwenden
(
        GefangenesPokemonID int REFERENCES
            GefangenesPokemon(ID),
        AttackeName varchar(255) REFERENCES Attacke(Name)
            ,
        PRIMARY KEY (GefangenesPokemonID, AttackeName),
    CHECK (AttackTypesCheck(GefangenesPokemonID,
        AttackeName)),
    CHECK (OnlyTwoAttacks(GefangenesPokemonID))
);
```

ItemsGesammelt

```
 CREATE TABLE ItemsGesammelt
(
    ItemName varchar(255) NOT NULL REFERENCES Item(Name),
    SpielerName varchar(255) NOT NULL REFERENCES Spieler(
        Name),
    ItemAnzahl int,
    PRIMARY KEY (ItemName, SpielerName),
    CHECK (ItemAnzahl >= 0)
)
```

BonbonsGesammelt

```
CREATE FUNCTION BonbonNumberCheck(BonbonNameX varchar
    (255), SpielerNameX varchar(255), BonbonVerfügbar int,
     BonbonVerbraucht int)
RETURNS boolean AS $$
DECLARE
     gefangen integer;
     freigelassen integer;
BEGIN
     SELECT COUNT(*) into gefangen FROM GefangenesPokemon
     WHERE SpielerName = SpielerNameX
     AND ID IN (SELECT PokedexID FROM Pokemon WHERE
         BonbonName = BonbonNameX);

     SELECT COUNT(*) into freigelassen FROM
         GefangenesPokemon
     WHERE SpielerName = SpielerNameX
     AND ID IN (SELECT PokedexID FROM Pokemon WHERE
         BonbonName = BonbonNameX)
     AND GefangenesPokemon.Freigelassen = true;

     RETURN BonbonVerfügbar + BonbonVerbraucht = gefangen
         * 3 + freigelassen;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE BonbonsGesammelt
(
     BonbonName varchar(255) NOT NULL REFERENCES Bonbon(
         Name),
     SpielerName varchar(255) NOT NULL REFERENCES Spieler(
         Name),
     BonbonVerfügbar int,
     BonbonVerbraucht int,
     PRIMARY KEY (BonbonName, SpielerName),
     CHECK (BonbonVerfügbar >= 0 AND BonbonVerbraucht >=
         0),
     CHECK (BonbonNumberCheck(BonbonName, SpielerName,
         BonbonVerfügbar, BonbonVerbraucht))
);
```

EierGesammelt

```
 CREATE TABLE EierGesammelt
(
```

```
    EiDistanz int NOT NULL REFERENCES Ei(Distanz),
    SpielerName varchar(255) NOT NULL REFERENCES Spieler(
        Name),
    EierAnzahl int,
    PRIMARY KEY (EiDistanz, SpielerName),
    CHECK (EierAnzahl >= 0)
)
```

5.4 Vermeidung von Redundanz und Anomalien (Jendrik)

Um Redundanzen und Anomalien in einem Relationalen Datenbankschema zu vermeiden, wird bei Datenbanken klassischerweise die Normalisierung angewandt. Unter dieser versteht man das Aufteilen von Attributen, also Spalten der Tabellen, in mehrere Relationen. Bei dieser Aufteilung müssen je nach Grad / Art der Normalisierung andere Normalisierungsregeln angewandt werden, bis letztendlich keine vermeidbaren Redundanzen mehr vorhanden sind.
Eine solche Form führt gleichzeitig auch zum Vermeiden von Anomalien, da durch die Vermeidung von Redundanzen keine Inkonsistenzen bei evtl. unvollständigen Änderungen von redundanten Daten mehr auftreten können.
In den nun folgenden Kapiteln werden die Normalformen des in Kapitel 5.1 erarbeiteten relationalen Datenbankschemas gebildet.

5.4.1 1. Normalform

Die erste Normalform eines Relationalen Datenbankschemas besagt, dass jedes Attribut der Relation(en) einen atomaren Wertebereich haben muss. Dies bedeutet, sie dürfen nicht zusammengesetzt, mengenwertig oder geschachtelt sein. Diese Regel wird an einer Stelle des relationalen Schemas von TauDBsi verletzt, da ein `Ort` aus Längen- und Breitengrad besteht. Der Wert Ort ist somit nicht atomar! Es ergibt sich also folgendes verändertes Schema für die Enitiy `gefangenesPokemon`:

RS:

`GefangenesPokemon(ID, SpielerName, PokedexID, Längengrad, Breitengrad, Zeit, Sternenstaub, Wettkampfpunkte, Freigelassen)`

Hinweis: Auch wenn man denken könnte, dass `Effekt` im Schema `Item` nicht atomar ist, ist dieses atomar, da Effekte sich nicht in Wirkdauer und Effekt aufteilen lassen, wie es in einigen anderen Spielen der Fall ist.

5.4.2 2. Normalform

Die Voraussetzung für eine Relationale Datenbank in zweiter Normalform ist, dass sich diese in der ersten Normalform befindet. Diese Voraussetzung ist bei dem relationalen Schema von TauDBsi offensichtlich erfüllt. Des Weiteren müssen alle nicht Primär-Attribute von jedem Schlüssel-Attribut/Kandidaten voll funktional abhängig sein. Damit sind Relationsschema mit nur einem Primärschlüssel

automatisch in der zweiten Normalform.
Dies ist bei dem relationalen Schema der TauDBsi Datenbank an der folgenden Stelle der Fall:
Das nicht Schlüsselattribut `BonBon#` ist nur von dem Schlüsselattribut `PokedexIDBasis` nicht aber vom Schlüsselattribut `PokedexIDWeiterentwicklung` abhängig, da die Kosten der Weiterentwicklung eines Pokemons lediglich vom Basispokemon abhängig sind. Um diesen Konflikt aufzulösen wird ein Schema `EntwicklungBonBon#` angelegt, welches die Anzahl einer ID zuweist.
RS: `Entwicklung(PokedexIDBasis, PokedexIDWeiterentwicklung, Bonbon#ID)`
RS: `EntwicklungBonBon#(ID, Bonbon#)`

Alle anderen Schema mit mehreren Primärschlüsseln ähneln der folgenden Form:
RS: `ItemsGesammelt(ItemName, SpielerName, Item#)`
In diesen Fällen ist offensichtlich, dass kein Verstoß der Regel vorliegt. Das relationale Schema von TauDBsi befindet sich somit nach den beschriebenen Änderungen in der zweiten Normalform.

5.4.3 3. Normalform

Damit das relationale Schema einer Datenbank der dritten Normalform genügt, muss es sich in der zweiten Normalform befinden. Dies ist nach den in Kapitel 5.4.2 vorgenommenen Änderungen bei dem relationalen Schema von TauDBsi gegeben.
Die zweite Bedingung ist, dass kein nicht Schlüssel-Attribut eine transitive Abhängigkeit zu einem Schlüsselkandidaten haben darf. Diese Bedingung wird bei dem erarbeiteten relationalen Schema an zwei Stellen nicht erfüllt:

Spieler
Das nicht primär-Attribut `Level` des relationalen Schemas `Spieler` ist über das nicht Schlüssel-Attribut `Erfahrung` transitiv von dem Primärattribut `Name` abhängig. Da es sich hierbei jedoch gewissermaßen um eine berechenbare Abhängigkeit handelt, kann das Attribut `Level` einfach weggelassen werden. Das Level des Spielers ist eine unnütze Information die sich aus der Erfahrung eines Spielers errechnen lässt.
RS: `Spieler(Name, Erfahrung, Sternenstaub, TeamName)`
Durch das Entfernen des Attributes wurde die transitive Abhängigkeit und gleichzeitig eine doppelt vorhandene Information eliminiert.

Ei
Bei dem Relationsschema `Ei` ist das nicht Primärattribut `Distanz` von dem Primärattribut `ID` abhängig. Das nicht Primärattribut `farbeName` ist wiederum von dem nicht Primärattribut `Distanz` abhängig. Es besteht also eine unerwünschte transitive Abhängigkeit! Um diese aufzulösen wurde entschieden das Primärattribut `ID` zu entfernen und das nicht Primärattribut `Distanz` zum neuen Primärattribut zu bestimmen, da es auch keine zwei verschiedenen Eier mit selber Distanz geben darf.

RS: `Ei(Distanz, FarbeName)`

5.4.4 Boyce Codd Normalform (BCNF)

Die BCNF setzt voraus, dass das betrachtete Schema sich in dritter Normalform befindet. Zusätzlich müssen alle Determinanten eines Relationsschemas ein Superschlüssel sein. Sollte die Abhängigkeit trivial sein, so muss die Determinante jedoch kein Superschlüssel sein.
Da sich das relationale Schema der TauDBsi Datenbank mit den bisher durchgeführten Änderungen bereits in der dritten Normalform befindet, bleibt nur noch die zweite Bedingung zu prüfen.
Die einzige Determinante, die kein Superschlüssel ist befindet sich in dem Relationsschema `Pokemon`:
RS: `Pokemon(PokedexID, Name, BonbonName, EiID)`
Das Primärattribut `PokedexID` ist offensichtlich von dem nicht Primärattribut `Name` abhängig. Diese Abhängigkeit ist jedoch trivial und somit nicht relevant!

5.4.5 Durch Normalisierung hervorgehende Verbesserungsmöglichkeiten

Durch die Überführung des relationalen Schemas von TauDBsi in die erste Normalform wurde das Attribut `Ort` in `GefangenePokemon` in zwei einzelne Attribute `Längengrad` und `Breitengrad` unterteilt. Es lässt sich so sehr schnell erkennen, dass es nun möglich ist die Attribute `Längengrad` und `Breitengrad` der Schema `GefangenePokemon` und `Arena` auszulagern. Dies ist vor allem sinnvoll, da es oft vorkommen kann, dass Spieler an der selben Stelle Pokemon fangen. So wäre es nicht notwendig immer wieder die selben Längen und Breitengrade zu speichern.
Die Verbesserung, die hier gemacht werden kann würde das folgende Relationsschema zur Folge haben:

RS: `Ort(OrtID, Längengrad, Breitengrad)`

RS: `GefangenesPokemon(ID, SpielerName, PokedexID, OrtID, Zeit, Sternenstaub, Wettkampfpunkte, Freigelassen)`

RS: `Arena(OrtID, Name, Level, Foto, TeamName)`

Somit wird `Arena` zu einer Spezialisierung von Ort. Alle Orte werden in einer separaten Tabelle festgehalten. Somit kann in `GefangenePokemon` einfach auf einen Ort referenziert werden (Fremdschlüssel).

5.4.6 Relationales Schema: Endergebnis

RS1: Pokemon(PokedexID, Name, BonbonName, EiDistanz)

RS2: Bonbon(Name, FarbeName)

RS3: Typ(Name)

RS4: Attacke(Name, Schaden, TypName)

RS5: Farbe(Name, Farbwert)

RS6: Spieler(Name, Erfahrung, Sternenstaub, TeamName)

RS7: Team(Name, FarbeName)

RS8: Arena(OrtID, Name, Level, Foto, TeamName)

RS9: Ei(Distanz, FarbeName)

RS10: Item(Name, Effekt)

RS11: GefangenesPokemon(ID, SpielerName, PokedexID, OrtID, Zeit, Sternenstaub, Wettkampfpunkte, Freigelassen)

RS12: Ort(OrtID, Längengrad, Breitengrad)

RS13: EntwicklungBonBon#(ID, Bonbon#)

RS14: istVonTyp(PokedexID,TypName)

RS15: kannVerwenden(GefangenePokemonID,AttackeName)

RS16: EierGesammelt(SpielerName, EiDistanz, Eier#)

RS17: Verteidiger(OrtID, gefangenePokemonID)

RS18: ItemsGesammelt(ItemName, SpielerName, Item#)

RS19: Entwicklung(PokedexIDBasis, PokedexIDWeiterentwicklung, EntwicklungBonbon#ID)

RS20: TypEffektivität(TypNameAngreifer, TypNameVerteidiger, Multiplikator)

RS21: BonbonsGesammelt(BonbonName, SpielerName, AnzahlVerfügbar, Anzahlverbraucht)

6 Sichten (Jendrik)

Sichten (engl. View) bezeichnen im relationalen Modell eine virtuelle Relation (logische Relation/ virtuelle Tabelle), die über eine Anfrage (z.B. SQL) definiert ist. Diese virtuelle Relation ist im Gegenteil zu einer Basis Relation, die als Teil des konzeptionellen Schemas als Tupel physisch auf der Datenbank vorhanden ist, nicht physisch gespeichert. Virtuelle Relationen werden wie oben bereits angedeutet dynamisch aus einer Datenbank-Anfrage berechnet, die zusammen mit dem Namen und dem Relationsschema bei der Definition der Sicht angegeben wird.
Eine Sicht kann demnach nicht mehr Informationen enthalten, als die Basis-Sicht, da diese als einzige Quelle der Sicht dient. Es können jedoch Relationen/Zusammenhänge gezeigt werden, die zwar in dem ursprünglichen Schema vorhanden, jedoch nicht offensichtlich sind.
Sichten können vom Nutzer genau wie Tabellen genutzt werden, sind jedoch bei der Funktion des Updates eingeschränkt (Problem der änderbaren Sichten). Grund für die Nutzung von Sichten ist die Vereinfachung des Zugriffs auf das Datenbankschema, da bei normalisierten Datenbanken oft eine Verteilung der Daten über Relationen auftritt.

Im Folgenden werden einige Sichten auf das normalisierte Relationsschema unserer TauDBsi Datenbank definiert. Die Wahl der nötigen Sichten ist aus der Repräsentation der Daten in dem mit der Datenbank verbundenen Spiels.

6.1 Alle Arenen/ Spieler eines Teams

Eine Liste aller Arenen/ Spieler eines bestimmten Teams ist für die Übersicht eine sehr wichtige Sache. Dies wird als VIEW umgesetzt, da es nur eine übersichtliche Anzahl an Teams gibt.
Hier am Beispiel von Gelb. Insgesamt gibt es drei Teams im Originalspiel.
Spieler

```
CREATE VIEW SpielerTeamGelb AS
SELECT *
FROM Spieler WHERE TeamName LIKE 'Team Gelb'
```

Arenen

```
CREATE VIEW ArenenTeamGelb AS
SELECT *
FROM Arena WHERE TeamName LIKE 'Team Gelb'
```

6.2 Alle Basis Pokemon

Eine Auflistung aller Basispokemon ist für die Übersicht sehr wichtig.

```
CREATE VIEW BasisPokemon AS
SELECT *
FROM Pokemon
WHERE pokedexID NOT IN (SELECT DISTINCT
    PokedexIDWeiterentwicklung
FROM Entwicklung)
```

6.3 PokemonAusEi

Um auswürfeln zu können, welches Pokemon aus einem Ei schlüpft, ist eine Auflistung aller Pokemon, die aus diesem Ei schlüpfen können, notwendig. Hier am Beispiel von einem 5km Ei. Die Ausgangslage ist ähnlich der bei Team.

```
CREATE VIEW PokemonAusEi AS
SELECT *
FROM BasisPokemon
WHERE Pokemon.EiDistanz = 5
```

6.4 Alle Orte von Arenen

Um Arenen auf der Karte einzeichnen zu können ist eine Liste aller Orte für Arenen sinnvoll.

```
CREATE VIEW ArenaOrte AS
SELECT Ort.Längengrad, Ort.Breitengrad
FROM Arena
INNER JOIN Ort ON Ort.OrtID = Arena.OrtID
```

6.5 ungefangene Pokemon Pokemon

Für statistische zwecke kann diese Angabe durchaus gut verwendet werden.

```
CREATE VIEW ungefangenePokemon AS
SELECT *
FROM Pokemon
WHERE pokedexID NOT IN (SELECT DISTINCT PokedexID
FROM gefangenePokemon)
```

7 Standardanfragen mit SQL und OCL (Jan)

In diesem Abschnitt werden Abfragen in SQL und OCL geschrieben, die bei der Nutzung der Datenbank besonders nützlich sein werden.

7.1 Alle möglichen Attacken eines Pokémon

```
SELECT Attacke.Name, Attacke.Schaden, Attacke.TypName
FROM Attacke, KannVerwenden, GefangenesPokemon, Pokemon
WHERE KannVerwenden.AttackeName=Attacke.Name
AND KannVerwenden.GefangenesPokemonID=GefangenesPokemon.
    ID
AND GefangenesPokemon.PokedexID=Pokemon.PokedexID
AND Pokemon.Name='Bisasam'
```

Mit dieser Abfrage erhält man alle Attacken, die von Pokémon einer bestimmten Art beherrscht werden. Dafür werden die Tabellen Attacke, KannVerwenden, GefangenesPokemon und Pokemon in einer WHERE Klausel über ihre Primär/Fremdschlüssel miteinander verknüpft.
Eine ähnliche Anfrage sieht in OCL so aus:

```
Pokemon.allInstances()->select(Name='Bisasam').
    gefangenesPokemon.attacke
```

In OCL nimmt man sich alle Instanzen der Klasse Pokemon und selektiert davon die, deren Name = Bisasam ist. Von diesen Fällen geht man mit dem Punkt Operator in die Klasse GefangenesPokemon und von dort zu allen Attacken.

7.2 Alle Bonbons eines Spielers

```
SELECT Bonbon.Name, BonbonsGesammelt.BonbonVerfügbar
FROM BonbonsGesammelt
INNER JOIN Bonbon ON Bonbon.Name = BonbonsGesammelt.
    BonbonName
INNER JOIN Spieler ON Spieler.Name = BonbonsGesammelt.
    SpielerName
WHERE Spieler.Name='DoDoSt'
```

Diese Abfrage listet alle Bonbons eines Spielers auf. Dafür wird die Tabelle BonbonsGesammelt mit Bonbon und Spieler verknüpft (INNER JOIN). Es wurde INNER JOIN und keine andere JOIN Art gewählt, da nur die Daten benötigt werden, die in je beiden Tabellen vorhanden sind.
In der WHERE Klausel wird der Name des Spielers, dessen Bonbons man betrachten möchte, geprüft.
Eine ähnliche Anfrage sieht in OCL so aus:

```
Spieler.allInstances()->select(Name='DoDoSt').
    bonbonsGesammelt->collect(AnzahlVerfuegbar)
```

Ähnlich wie bei der ersten Abfrage werden alle Instanzen der Spieler genommen und davon einer mit bestimmtem Namen ausgewählt. Von diesem werden alle gesammelten Bonbons betrachtet und mit collect(AnzahlVerfuegbar) nur die verfügbare Anzahl ausgegeben.

7.3 Alle Arten von Pokémon, die ein Spieler in einem bestimmten Zeitraum gefangen hat

```
SELECT DISTINCT Pokemon.Name
FROM Pokemon
INNER JOIN GefangenesPokemon
ON GefangenesPokemon.PokedexID = Pokemon.PokedexID
WHERE GefangenesPokemon.Zeit BETWEEN TO_DATE
    ('2017-02-23', 'YYYY-MM-DD')
AND TO_DATE('2017-02-26', 'YYYY-MM-DD')
AND GefangenesPokemon.SpielerName = 'DoDoSt'
```

Durch diese Abfrage werden alle unterschiedlichen Arten von Pokémon angezeigt, die ein Spieler in einem bestimmten Zeitraum gefangen hat. Das Schlüsselwort DISTINCT bewirkt dabei, dass doppelte Einträge nur jeweils 1 mal aufgezählt werden. Durch INNER JOIN geschieht eine Verknüpfung mit der GefangenesPokemon Tabelle. In der WHERE Klausel wird geprüft, ob das Datum des Fangs in einem bestimmten Bereich liegt: Damit die Eingabe des Zeitraums nutzerfreundlich als String geschehen kann, muss dieser mit der Funktion TO_DATE in ein Datum geparsed werden (Gilt für PostgreSQL, in MySQL lautet die Funktion STR_TO_DATE und hat eine ähnliche Syntax). Außerdem wird noch auf einen bestimmten Nutzernamen geprüft.
Eine ähnliche Anfrage sieht in OCL so aus:

```
Spieler.allInstances()->select(Name='DoDoSt').
    gefangenesPokemon->select(Zeit>1487808000 and Zeit <
    1488067200)->asSet()
```

Wie im vorherigen Fall beginnt es auch hier damit, einen speziellen Spieler zu selektieren. Von diesem werden die gefangenen Pokemon genommen, die in einem bestimmten Zeitraum gefangen wurden. Dieser Zeitraum wurde mit Timestamps angegeben. Zum Schluss wird der Bag zu einem Set umgewandelt, damit keine doppelten Einträge vorhanden sind.

7.4 Alle Arenen und Spieler, die in diesen Verteidiger abgelegt haben

```
SELECT Arena.Name, Spieler.Name
FROM Arena
```

```
LEFT JOIN Verteidiger ON Verteidiger.OrtID=Arena.OrtID
LEFT JOIN GefangenesPokemon ON GefangenesPokemon.ID=
    Verteidiger.GefangenesPokemonID
LEFT JOIN Spieler ON Spieler.Name=GefangenesPokemon.
    SpielerName
```

Diese Abfrage listet alle Arenen auf, selbst wenn diese nicht mit Verteidigern besetzt sind (Dafür wurde ein LEFT JOIN verwendet) und zeigt außerdem, welche Spieler in der Arena Pokemon zur Verteidigung gelagert haben.
Eine ähnliche Anfrage sieht in OCL so aus:

```
Arena.allInstances()->iterate(
    a : Arena ;
    b : Bag(Bag(OclAny)) = Bag{}
    | b->including(Bag{a.Name})->including(a.
        gefangenesPokemon.spieler->collect(Name))
    )
```

Diese Abfrage listet alle Arenen (in einzelnen Bags) und die Spieler, die in diesen Arenen Pokemon zur Verteidigung eingesetzt haben (auch in einzelnen Bags) auf. Dafür wird über alle Instanzen der Arenen iteriert und dabei die Beutel gefüllt.

7.5 Alle Typen auflisten, gegen die ein Spieler keine sehr effektive Attacke zur Verfügung hat

```
SELECT TypEffektiv.TypNameVerteidiger FROM TypEffektiv
    WHERE TypEffektiv.TypNameAngreifer IN (SELECT Typ.Name
     FROM Typ INNER JOIN Attacke ON Attacke.TypName=Typ.
    Name INNER JOIN KannVerwenden ON KannVerwenden.
    AttackeName=Attacke.Name INNER JOIN GefangenesPokemon
    ON GefangenesPokemon.ID=KannVerwenden.
    GefangenesPokemonID WHERE GefangenesPokemon.
    SpielerName='DoDoSt') GROUP BY TypEffektiv.
    TypNameVerteidiger HAVING MAX(TypEffektiv.
    Multiplikator)<2
```

Mit dieser Abfrage werden alle Typen aufgelistet, gegen die kein Pokemon eines bestimmten Spielers eine sehr effektive Attacke hat. Man kann damit also die „Schwachstelle“ eines Spielers finden. Dafür werden alle Typen von Verteidigern aus der TypEffektiv Tabelle aufgelistet, deren Angreifender-Typ in einer Unterabfrage enthalten ist. Diese Unterabfrage selektiert alle Typen von Attacken, die die Pokemon eines Spielers insgesamt zur Verfügung haben. Diese Verteidigenden-Typen werden daraufhin gruppiert und es wird geprüft, ob kein x2 Multiplikator vorhanden ist, was bedeutet, dass der Spieler keine Möglichkeit hat den Verteidigenden-Typen sehr effektiv zu treffen.
Eine ähnliche Anfrage sieht in OCL so aus:

```
Typ.allInstances()->iterate(
    t : Typ ;
    b : Set(String) = Set{}
    | if t.typEffektiv[Verteidiger].forAll(typ | typ.
        Multiplikator < 2 or Spieler.allInstances()->
        select(Name='DoDoSt').gefangenesPokemon.attacke.
        typ.forAll(typ2|typ2.Name <> typ.Angreifer.Name))
   then b->including(t.Name) else b endif
)
```

Hierbei iterieren wir über alle Typen. Dabei prüfen wir, ob der Typ als Verteidiger bei einer Effektivitätsberechnung einen Multiplikator hat der kleiner als 2 ist, oder falls dies nicht der Fall ist, ob der Spieler diesen angreifenden-Typen gar nicht in seiner Auswahl hat.

Literaturverzeichnis

[1] Gogolla, Martin: Vorlesungsmaterialien zur Vorlesung Datenbanksysteme im Wintersemester 2016/2017, 2016. http://www.db.informatik.uni-bremen.de/teaching/courses/ws2016_dbs, [Online, Abruf 01.03.2017].

[2] Bjorn Christian Schaefer: Anwendungssystem eines Schulungsanbieters. http://www.db.informatik.uni-bremen.de/teaching/courses/ws2016_dbs/Schaefer_2008.pdf, [Online, Abruf 02.03.2017].