

De auteur en promotor geven de toelating deze scriptie voor consultatie beschikbaar te stellen en delen ervan te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting uitdrukkelijk de bron te vermelden bij het aanhalen van resultaten uit deze scriptie.

The author and promoter give the permission to use this thesis for consultation and to copy parts of it for personal use. Every other use is subject to the copyright laws, more specifically the source must be extensively specified when using from this thesis.

Gent, Juni 2013

De promotor

De begeleider

De auteur

Prof. dr. ir. T. Schrijvers

Benoit Desouter

Kenneth Saey

Woord vooraf

To do

*Kenneth Saey
Gent 3 Juni 2013*

Inhoudsopgave

1	Inleiding	1
2	Achtergrond en motivatie	2
2.1	Collectible Card Games	2
2.1.1	Spelzones	2
2.1.2	Kaarttypes	3
2.1.3	Vaardigheden	6
2.1.4	DSL en modulariteit	6
2.2	Domein-specifieke talen	7
2.3	Scala	7
2.3.1	Integratie met Java	7
2.3.2	Scala is object-georiënteerd	10
2.3.3	Scala is een functionele programmeertaal	11
2.3.4	Scala is statisch getypeerd	11
2.3.5	Scala is uitbreidbaar	11
3	Ontwikkeling van een DSL	14
3.1	Kaarten	14
3.2	Creatures	16
3.2.1	Aanmaak van Creatures	16
3.2.2	Het probleem met implicits	17
3.3	Vaardigheden	18
3.4	Vaardigheden met parameters	19
4	Evaluatie	21
4.1	Creatures en vaardigheden	21
4.2	Vaardigheden met parameters	21
4.3	Vaardigheden samenstellen	22

5 Gerelateerd werk	23
5.1 SandScape	23
5.2 Forge	23
6 Conclusie	25
Bibliografie	26

Hoofdstuk 1

Inleiding

Collectible Card Games (CCGs) of *Trading Card Games* (TCGs) zoals *Magic: The Gathering* en *Shadow Era* zijn snel veranderende spellen waaraan meerdere keren per jaar nieuwe kaarten en spelregels toegevoegd worden. Sinds het einde van de jaren '90 bestaan er ook computerspellen gebaseerd op CCGs. Nieuwe versies van die computerspellen worden echter minder snel released dan hun fysieke tegenhangers. Eén van de redenen hiervoor is dat voor elke nieuwe kaart of spelregel nieuwe broncode geschreven moet worden.

In een ideale wereld waar computers natuurlijke taal volledig begrijpen zouden nieuwe kaarten door de auteurs zelf kunnen toegevoegd worden, enkel gebruik makend van de instructies die al op de kaarten geprint staat. Een oplossing die het midden houdt tussen het schrijven van broncode en het interpreteren van instructies op de kaarten zelf is een domein-specifieke taal (DST). Een domein-specifieke taal is een programmeertaal die speciaal ontwikkeld werd op problemen binnen een goed gedefinieerd probleemgebied op te lossen. In deze masterproef wordt uitgelegd hoe een DST kan helpen om de snelle veranderingen in een collectible card game te kunnen volgen in de digitale versie ervan.

De inhoud van deze masterproef ziet er als volgt uit: TODO

Hoofdstuk 2

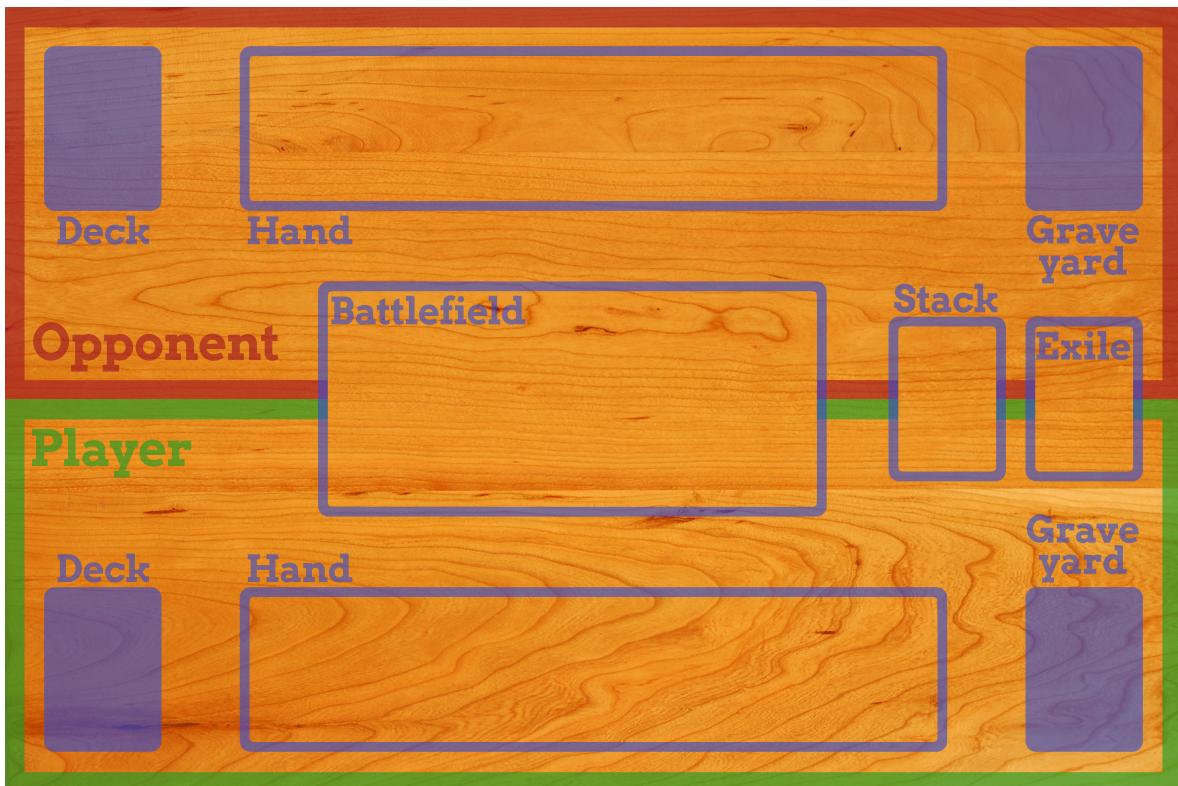
Achtergrond en motivatie

2.1 Collectible Card Games

Een *collectible card game* is een type kaartspel, meestal gespeeld met twee personen, met als doel om de levenspunten van de tegenstander op nul te krijgen. Dit gebeurt door het spelen van verschillende kaarten. Het spelbord (meestal gewoon een tafel) is opgedeeld in zones. Elke zone kan een of meerdere kaarten bevatten, en de acties die met een kaart ondernomen kunnen worden zijn afhankelijk van de spelzone waarin de kaart zich bevindt. De globale spelregels beschrijven op welke manier kaarten van spelzone kunnen veranderen. Er zijn steeds verschillende types van kaarten aanwezig, vaak met hun eigen specifieke eigenschappen en vaardigheden, die een invloed uitoefenen op alle onderdelen van het spel. CCGs zijn een niet te onderschatten uitdaging vanuit het standpunt van een software ontwikkelaar, aangezien ze veel verschillende vormen van modulariteit bevatten en een overvloed aan regels die de basisregels van het spel grondig kunnen beïnvloeden.

2.1.1 Spelzones

Een eerste niveau van modulariteit binnen CCGs zijn de verschillende zones op het virtuele spelbord. De beschikbare zones van één welbepaald spel veranderen bijna nooit, maar verschillende CCGs beschikken wel over verschillende spelzones. Bovendien hebben spelzones hun eigen effect op de spelregels, met betrekking tot de acties die uitgevoerd kunnen worden met kaarten die zich in de zone bevinden. Een van de oudste en meest populaire CCGs is *Magic: The Gathering* dat in 1993 door Richard Garfield (Garfield (1993)) ontworpen werd. Voorbeelden zullen aan de hand van dit spel gegeven worden. *Magic: The Gathering* bevat zes verschillende spelzones, zoals te zien op Figuur 2.1. Elke speler heeft zijn eigen deck of *library*. Deze bevat aan het begin van het spel alle kaarten waarmee een speler het spel wil spelen. Vaak staat er een limiet op het aantal kaarten dat een deck mag bevatten. Tijdens het spel zullen er verschillende momenten zijn waarop een speler een of meerdere kaarten van zijn deck mag trekken om op die manier in het spel te brengen.



Figuur 2.1: Magic: The Gathering - Spelzones

Elke speler heeft ook een *handzone*. Dit is geen echte zone op het spelbord, maar dit zijn de kaarten die de speler letterlijk in zijn hand houdt, afgeschermd van de tegenstander. Aan het begin van het spel trekt een speler een welbepaald aantal kaarten van zijn deck en plaatst die in zijn hand. Gedurende het spel wordt de hand aangevuld met de kaarten die van zijn deck getrokken worden.

De *battlefield*-zone is een gemeenschappelijke zone waar zich de actie van het spel bevindt. Hier worden kaarten tegen elkaar uitgespeeld in de hoop de tegenstander te verslaan. Kaarten waarvan alle acties opgebruikt zijn of die om een of andere reden *vernietigd* werden komen op de aflegstapel terecht, die in *Magic: The Gathering* de *graveyard* genoemd wordt.

Verder bevat *Magic: The Gathering* nog twee andere spelzones, die minder frequent voorkomen in andere CCGs. De *stack* bevat alle *spells* en *abilities*, twee speciale types kaarten. Verder is er ook nog een *exile*-zone, waar kaarten terecht komen die uit het spel verwijderd moeten worden.

2.1.2 Kaarttypes

Verschillende types en subtypes van kaarten zorgen voor een tweede niveau van modulariteit. *Magic: The Gathering* bijvoorbeeld bevat onder andere *land*-kaarten, *creature*-kaarten

(wezens), *sorceries*, *instants*, *enchantments*, *artifacts* en hun subtype *equipment* en *planeswalkers*. Voorbeelden van deze kaarten zijn te zien in Figuur 2.2. Landkaarten zijn onderverdeeld in verschillende subtypes waarvan de vijf voornaamste *bergen*, *bosSEN*, *eilandEN*, *moerassen* en *vlaktes* zijn. De *creature*-kaarten kunnen volgens de basisspelregels enkel van de *handzone* naar de *slagveldzone* (*battlefield*) verplaatst worden, indien de *slagveldzone* van die speler de correcte types en aantallen van landkaarten bevat zoals aangegeven op de *creature*-kaart. In dit geval zijn de landkaarten dus een vereiste voor het spelen van de *creature*-kaarten.



Figuur 2.2: Magic: The Gathering - Kaart types

Het spel *Shadow Era Studios* (2011) bevat een gelijkaardig type van *creature*-kaarten (Figuur 2.3), maar de vereiste om deze te kunnen spelen is hierbij niet een specifieke set van landkaarten, maar wel een voldoende grote voorraad *grondstoffen* die vergroot kan worden door andere kaarten op te offeren.



Figuur 2.3: Shadow Era - Creature

Andere kaarten kunnen dan weer effect hebben op de eigenschappen van kaarten. Zo heeft een *equipment*-kaart een invloed op de *creature*-kaart waaraan ze gehecht is. Figuur 2.4 toont de *equipment*-kaart *Whispersilk Cloak* en de *Creature*-kaart *Craw Wurm*. Zolang de *equipment*-kaart aan de *creature*-kaart gehecht is, krijgt de *creature*-kaart de *abilities* *unblockable* en *shroud*.

Figuur 2.4: Het effect van *equipment* op *creatures*

Het is dus belangrijk bij de implementatie van een CCG dat veranderingen aan eigenschappen en spelregels zonder ingewikkelde broncode geïmplementeerd kunnen worden.

2.1.3 Vaardigheden

Een derde niveau van modulariteit is dat vele kaarten eigenschappen hebben die zich gedragen als *vaardigheden* (*abilities*). Vaardigheden veranderen de *gameplay* minder dramatisch: het effect van een vaardigheid is meestal in tijd beperkt, namelijk zolang de kaart meedoet aan het actieve spel, en de invloed ervan beperkt zich meestal tot de kaarten die rechtstreeks met de kaart in kwestie in contact komen. Een goed voorbeeld van een dergelijke vaardigheid is *Flying*. *Drifting Shade* (Figuur 2.5), een wezen met de vaardigheid *Flying* kan tijdens een aanvalsactie enkel gebloktoberd worden door wezens die ook de vaardigheid *Flying* bezitten.



Figuur 2.5: Flying vaardigheid: Drifting Shade

De beperkte invloed van eigenschappen die zich gedragen als vaardigheden is echter geen vaststaand feit, wat opnieuw een belangrijk punt is om in acht te nemen tijdens de ontwikkeling van een computerversie van een CCG.

2.1.4 DSL en modulariteit

Het doel van deze masterproef is om een domein-specifieke taal te ontwikkelen die eenvoudig genoeg is zodat ze gebruikt kan worden door mensen met weinig programmeerervaring (zoals de auteurs van CCGs) maar toch krachtig genoeg om alle niveaus van modulariteit aan te passen. Hier kunnen opeenvolgende computerversies van CCGs niet alleen sneller uitgebracht worden, maar is het ook mogelijk om verschillende CCGs te ontwikkelen vanaf eenzelfde codebasis.

2.2 Domein-specifieke talen

Een domein-specifieke taal (DSL) is een programmeertaal die speciaal ontworpen werd op problemen binnen een goed gedefinieerd probleemdomein op te lossen. Ze verschilt van een *general-purpose language* (GPL) in het feit dat een GPL in staat moet zijn om problemen in alle domeinen op te lossen (uiteraard met variabele efficiëntie). Domein-specifieke talen vallen onder te verdelen in drie categorieën.

De meest algemeen bekende categorie van domein-specifieke talen zijn de alleenstaande DSLs. Zoals de naam aangeeft beschikken alleenstaande DSLs over hun eigen compiler of interpreter om de code uit te voeren. Voorbeelden van alleenstaande DSLs zijn HTML, voor het definiëren van de structuur van webpagina's, en SQL, voor het schrijven van query's voor SQL-databases.

Een tweede type van DSLs zijn DSLs die ingebed zijn in applicaties. Deze domein-specifieke talen kunnen niet worden gebruikt buiten hun gastheertoepassing en zijn vaak sterk verbonden met het doel van de gastheertoepassing. De formules die in Microsoft Excel gebruikt kunnen worden in de cellen van een rekenblad zijn een voorbeeld van een DSL die ingebed is in een applicatie.

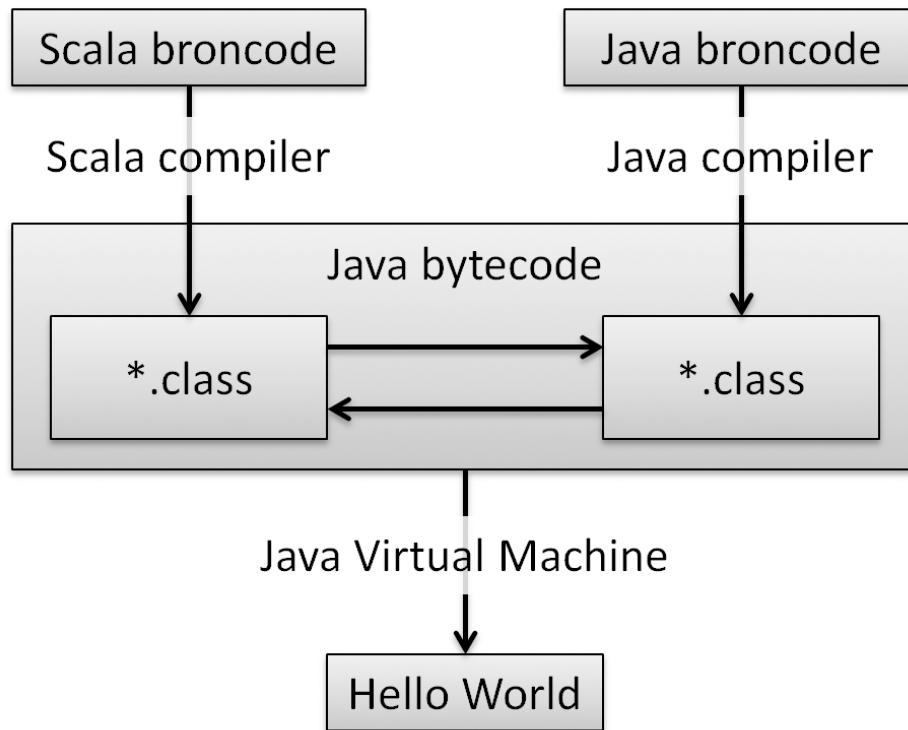
Het derde en laatste type van DSLs zijn de ingebedde domein-specifieke talen (*Embedded Domain Specific Languages* of EDSLs). Deze zijn ingebed in een programmeertaal en zijn vaak het gevolg van een slim gebruik van methodenamen en *syntactic sugar* aangeleverd door de programmeertaal. Indien de gastheertaal over voldoende *syntactic sugar* beschikt wordt het mogelijk om een EDSL sterk op natuurlijke taal te doen lijken. Dit is een zeer handig kenmerk indien nieuwe code snel moet worden toegevoegd door anderen dan de auteur van de originele broncode.

2.3 Scala

Scala is een *general purpose language* die ontworpen is om veelvoorkomende *programming patterns* uit te drukken op een bondige, elegante en type-veilige manier (Odersky (2010)).

2.3.1 Integratie met Java

Scala broncode wordt door de Scala-compiler gecompileerd naar Java bytecode die volledige compatibel is met bytecode die door een Java-compiler vanuit Java broncode gegenereerd wordt (zie Figuur 2.6). Dit betekent dat Scala, net zoals Java, uitgevoerd wordt door de Java virtuele machine (JVM). Meer nog, doordat Scala en Java compatibel zijn op bytecode niveau is het mogelijk om bestaande Java bibliotheken aan te spreken vanuit Scala en omgekeerd, bestaande Scala bibliotheken kunnen in Java programma's gebruikt worden.

**Figuur 2.6:** Scala-Java integratie

Ondanks het feit dat Scala en Java bytecode-compatibel zijn, is dit niet het geval voor de broncode. Scala-code lijkt op Java-code, maar is meestal een stuk bondiger. Constructie van een lege klasse *Person* is nog heel gelijkaardig (zie Codefragmenten 2.1 en 2.2).

```

1 public class Person {
2 }
```

Code Listing 2.1: Java klasse

```

1 class Person {
2 }
```

Code Listing 2.2: Scala klasse

Het toevoegen van een publiek veld *name* met bijhorende constructor is echter veel eenvoudiger in Scala (Codefragment 2.4) van in Java (Codefragment 2.3), aangezien de constructor meteen op de klassenaam volgt.

```

1 public class Person {
2     public String name;
3     public Person(String name) {
```

```

4     this.name = name;
5   }
6 }
7 Person p = new Person("name"); // Constructor
8 p.name; // Getter
9 p.name = "new name"; // Setter

```

Code Listing 2.3: Java klasse met publiek veld

```

1 class Person(var name: String){
2 }
3 val p: Person = new Person("name") // Constructor
4 p.name // Getter
5 p.name = "new name" // Setter

```

Code Listing 2.4: Scala klasse met publiek veld

Indien we het veld *name* privaat maken Scala toe om *getters* en *setters* te schrijven die ervoor zorgen dat de toegang van buitenaf tot een privaat veld uniform wordt met de toegang tot een publiek veld (Codefragment 2.6). In Java is dit niet mogelijk en verschilt toegang tot een privaat veld van toegang tot een publiek veld. Vergelijk hiertoe Codefragmenten 2.3 en 2.5.

```

1 public class Person {
2   private String name;
3   public Person(String name) {
4     this.name = name;
5   }
6   public String name() { // Getter
7     return name;
8   }
9   public void name(String name) { // Setter
10    this.name = name;
11  }
12 }
13 p.name(); // Getter
14 p.name("new name"); // Setter

```

Code Listing 2.5: Java klasse met privaat veld

```

1 class Person(name: String) {
2   private var _name: String = name
3   public def name: String = _name // Getter
4   public def name_=(name: String) = _name = name // Setter

```

```

5 }
6 p.name // Getter
7 p.name = "new name" // Setter

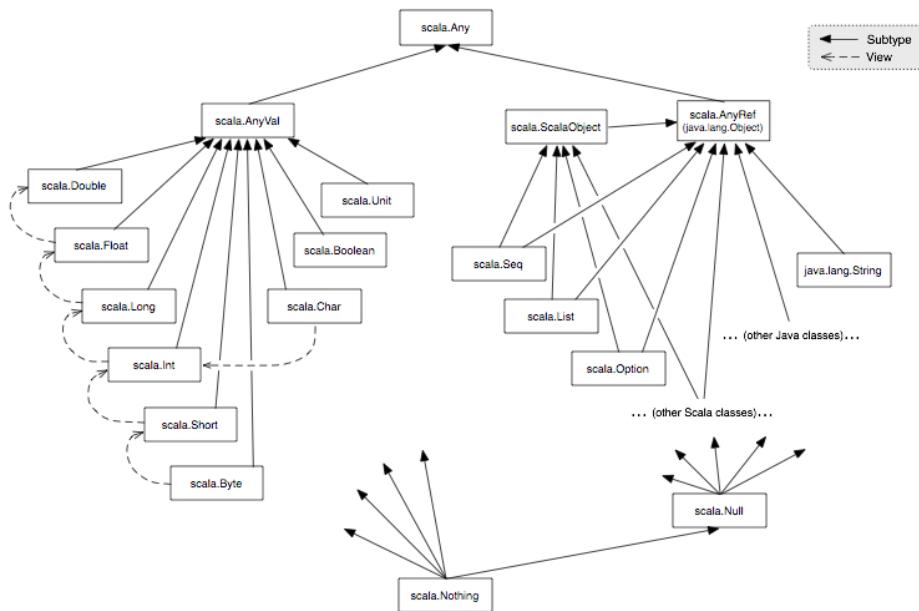
```

Code Listing 2.6: Scala klasse met privaat veld

Merk ook op dat in Scala het gebruik van een puntkomma niet vereist is. Gelet op codeerstandaarden hebben we in Java 25 regels code geschreven en in Scala slechts 14, ongeveer de helft. Scala is dus inderdaad een stuk bondiger, zonder in te boeten aan leesbaarheid. Door de uniforme toegang tot publieke en private velden van een klasse kunnen we zelfs zeggen dat Scala eenduidiger is dan Java.

2.3.2 Scala is object-georiënteerd

Tijdens kennismakingen met Java wordt er vaak gezegd: *In Java, everything is an object*. Dit refereert naar de object-georiënteerdheid van Java, maar is eigenlijk een ontrecht statement, aangezien primitieve types (*int*, *double*, ...) en functies in Java geen objecten zijn. Scala lost dit echter op door één superklasse *scala.Any* te voorzien. *scala.Any* heeft twee subklassen: *scala.AnyVal*, een superklasse voor de voorgedefinieerde klassen die primitieve types voorstellen, zoals *scala.Int* en *scala.Double* en *scala.AnyRef*, een superklasse voor alle andere klassen. Een overzicht van de klasstructuur van Scala is te zien in Figuur 2.7.

**Figuur 2.7:** Scala klasse-hiërarchie

Nog belangrijker dan het voorzien van klassen voor de primitieve types uit Java is het feit dat

in Scala ook functies objecten zijn. Functies zijn objecten van de klasse *Function*. Dit leidt tot de volgende sectie.

2.3.3 Scala is een functionele programmeertaal

Naast een puur object-georiënteerde invalshoek bezit Scala ook verschillende eigenschappen die ze tot een functionele programmeertaal maakt. Scala biedt ondersteuning voor anonieme functies. Codefragment 2.7 toont een anonieme functie die de derde macht van een getal berekent. Het type van deze functie is *Function1[Int, Int]*, een functie met één argument van het type *Int* dat een object van het type *Int* als resultaat heeft.

```
1 i => i * i * i
```

Code Listing 2.7: Anonieme functie

Aangezien functies objecten zijn kunnen ze eenvoudig aan andere functies meegegeven worden als argument. Zo biedt Scala een natuurlijke ondersteuning aan voor *hogere-orde functies*. Codefragment 2.8 geeft een voorbeeld van een functie *max* die het maximum van een reeks *values* bepaald aan de hand van een functie *comparator* die twee elementen met elkaar vergelijkt. Door andere functies als eerste argument mee te geven kan de gebruiker zelf bepalen volgens welk criterium het maximum bepaald moet worden.

```
1 def max(comparator: (Int, Int) => Int,
2         values: ArrayBuffer[Int]): Int = { // Implementatie
3 }
```

Code Listing 2.8: Hogere-orde functie

Verder kunnen functies in Scala genest worden en biedt de programmeertaal ondersteuning voor *currying*, het definiëren van meerdere parameterlijsten bij een methode. Het aanroepen van een dergelijke functie met minder parameterlijsten levert opnieuw een functie op, met de overige parameterlijsten als argumenten.

2.3.4 Scala is statisch getypeerd

Scala is, net zoals Java, een statisch getypeerde taal. Dit betekent dat het controleren van types al plaats vindt tijdens het compileren en niet pas tijdens het uitvoeren. Voordeel hiervan is dat er tijdens het uitvoeren van het programma geen *type-checks* meer moeten gebeuren.

2.3.5 Scala is uitbreidbaar

De Scala programmeertaal is uitzonderlijk geschikt voor het ontwerpen van domein-specifieke talen. Scala biedt hiervoor verschillende mechanismen. Een eerste mechanisme is dat elke methode zonder parameters als postfix operator gebruikt kan worden en elke methode met

slechts een parameter als infix operator. Binnen het thema van CCGs zouden we een *Creature*-klasse kunnen bedenken zoals in Codefragment 2.9 met twee methodes. Een *attacks*-methode met als argument het aan te vallen *Creature* en een *dies*-methode die opgeroepen hoort te worden wanneer een wezen dood gaat.

```

1 class Creature {
2     def attacks(other: Creature) {
3         // Implementation
4     }
5     def dies = {
6         // Implementation
7     }
8 }
```

Code Listing 2.9: Infix en postfix operatoren

Codefragment 2.10 toont hoe de infix (regel 3) en postfix operator (regel 5) gebruikt kunnen worden samen met hun equivalentie, meer traditioneel object-georiënteerde versies (regels 4 en 6). Door van dit mechanisme gebruik te maken hebben we een DSL voor *Creatures* gebouwd met twee operatoren.

```

1 val creature: Creature = new Creature
2 val enemy: Creature = new Creature
3 creature attacks enemy // Infix operator
4 creature.attacks(enemy) // OO-equivalent
5 enemy dies // Postfix operator
6 enemy.dies() // OO-equivalent
```

Code Listing 2.10: Gebruik van infix en postfix operatoren

Verder legt Scala veel minder beperkingen op aan methodenamen, waardoor methodes die als infix of postfix gebruikt worden niet te onderscheiden vallen van echte operatoren. Zo kunnen we een methode *+* definiëren voor een klasse *Fraction* (zie Codefragment 2.11) die een breuk voorstelt en de *+* verder gebruiken alsof de instanties van de *Fraction*-klasse echte getallen zijn en geen objecten.

```

1 class Fraction(numerator: Int, denominator: Int) {
2     def +(other: Fraction): Fraction = {
3         new Fraction(numerator * other.denominator
4             + denominator * other.numerator,
5             denominator * other.denominator
6         )
7     }
8 }
```

```

9 val quarter: Fraction = new Fraction(1, 4)
10 val third: Fraction = new Fraction(1, 3)
11 quarter + third // = Fraction(7, 12)

```

Code Listing 2.11: Vrije methodenamen

Een derde mechanisme dat van pas komt bij de constructie van DSLs is de automatische constructie van closures (Odersky (2008)). Een closure is de combinatie van de referentie naar een functie en haar eigen lokale variabelenomgeving. Indien een methode parameters bevat die zelf parameterloze functies voorstellen, dan kunnen de namen van deze methodes zelf gebruikt worden als argumenten van functies. Dit heet *call-by-name*-evaluatie. Codefragment 2.12 geeft aan wat hiervan het effect is. Er wordt een methode *my_while* gedefinieerd die twee parameters heeft, beide van het type *parameterloze-functie-type*. Op regel 8 van het codefragment wordt de methode aangeroepen met twee parameters. Deze parameters worden echter niet meteen geëvalueerd, maar ze worden geëncapsuleerd in functie-objecten die doorgegeven worden aan de body van de *my_while*-methode en pas geëvalueerd worden op de plaatsen in die body waar de naam van de parameters voorkomen. Dit is op regels 2 en 3 van het codefragment. Aan de hand van dit mechanisme is het mogelijk om zelf controlestructuren te bouwen voor een DSL.

```

1 def my_while(cond: => Boolean)(body: => Unit): Unit = {
2   if(cond) {
3     body
4     my_while(cond)(body)
5   }
6 }
7 var i = 10
8 my_while(i > 0) {
9   println(i)
10  i -= 1
11 }

```

Code Listing 2.12: Automatische constructie van closures

Al deze mechanismen in Scala zullen ons helpen bij het bouwen van onze eigen DSL voor collectible card games.

Hoofdstuk 3

Ontwikkeling van een DSL

Het ontwikkelen van een domein-specifieke taal voor *collectible card games* is een proces dat best ondernomen wordt in incrementele stappen gericht op een specifiek type van modulariteit.

3.1 Kaarten

De basis van een CCG zijn uiteraard de kaarten, dus het is een logische eerste stap om kaarten te modelleren. Gemeenschappelijk aan alle kaarten is dat ze een naam hebben. De klassedefinitie voor een kaart ziet er bijgevolg uit als in Codefragment 3.1.

```
1 class Card {  
2     var _name: String = ""  
3 }
```

Code Listing 3.1: Card klasse

Aangezien kaarten eenvoudig toegevoegd moeten kunnen worden, is het belangrijk om een stuk DSL te voorzien voor het aanmaken van een kaart met een specifieke naam. Aangezien we willen aanleunen bij natuurlijke (engelse) taal kunnen we gebruik maken van het werkwoord *to call* om een methode te definiëren zoals in Codefragment 3.2.

```
1 class Card {  
2     var _name: String = ""  
3     def called(name: String): this.type = {  
4         _name = name  
5         this  
6     }  
7 }
```

Code Listing 3.2: Called-methode

Met deze zeven regels code en de kracht van Scala hebben we zonet een eerste stuk DSL geïmplementeerd om kaarten met een specifieke naam aan te maken. In onze DSL kunnen we de aanmaak van een nieuwe kaart met de naam *Black Lotus* nu schrijven zoals in Codefragment 3.3.

```
1 new Card called "Black Lotus"
```

Code Listing 3.3: DSL kaart creatie

Hierbij wordt gesteund op de *syntactic sugar* die aangeboden wordt door Scala. Zonder *syntactic sugar* zou die regel code geschreven moeten worden zoals in Codefragment 3.4. Aangezien de *called*-methode een methode is met één argument laat Scala ons toe om alle interpunctie (zowel de punt als de ronde haakjes) weg te laten.

```
1 (new Card).called("Black Lotus")
```

Code Listing 3.4: Kaart creatie zonder syntactic sugar

Belangrijk om op te merken is dat de *called*-methode zichzelf (*this*) terug geeft. Dit is nodig om van het principe van *method chaining* gebruik te kunnen maken. Dit betekent dat we het aanmaken van een kaart met een specifieke naam op één regel kunnen schrijven, zonder gebruik te maken van een tussentijdse variabele om de kaart zonder naam op te slaan. In een DSL komt dit eigenlijk neer op de constructie van zinnen. Zonder het principe van *method chaining* zou de creatie van die kaart er uitzien zoals in Codefragment 3.5.

```
1 val card = new Card
2 card called "Black Lotus"
```

Code Listing 3.5: Kaart creatie zonder method chaining

Scala bevat nog een handige feature, namelijk impliciete conversies, die ons toelaat om de aanmaak van kaarten nog compacter te schrijven. Impliciete conversies laten ons toe om methodes te definiëren die de één type omzetten in een ander. Als Scala tijdens het compileren van code een incorrect type tegen komt, dan zal hij eerst op zoek gaan naar een toepasbare impliciete methode en de conversie toepassen vooralleer een typefout te genereren. Aangezien een eenvoudige kaart enkel door een naam (van het type *String*) gedefinieerd wordt kunnen we een impliciete methode voorzien die een instantie van het type *String* omzet in een instantie van het type *Card*. De code om dit te bekomen is terug te vinden op regel 1 tot en met 3 van Codefragment 3.6.

```
1 object CardImplicits {
2     implicit def String2Card(name: String): Card = {
3         new Card called name
4     }
5 }
6 class Card {
```

```

7  var _name: String = ""
8  def called(name: String): this.type = {
9    _name = name
10   this
11 }
12 }
```

Code Listing 3.6: Impliciete String naar Card conversie

Hiermee wordt de aanmaak van de kaart *Black Lotus* gereduceerd tot Codefragment 3.7

```
1 "Black Lotus"
```

Code Listing 3.7: Kaart creatie met implicits

3.2 Creatures

3.2.1 Aanmaak van Creatures

Kaarten met enkel een naam maken natuurlijk geen interessant spel. Er is ook nood aan kaarten die het spel doen vorderen. In CCGs is de meest voorkomende manier om het spel te laten vooruitgaan een aanval doen met wezens (*creatures*). *Creatures* zijn een specifiek type van kaarten en elk CCG bezit dit concept, hoewel de naam ervan niet altijd dezelfde is. In *PokéMon*, een CCG voor jongeren, heteren de wezens, toepasselijk, PokéMon.

De belangrijkste eigenschappen van een *Creature* zijn de concepten van aanvalskracht en levenspunten. Aanvalskracht geeft aan hoeveel schade een *creature* kan doen, terwijl levenspunten aangeven hoeveel schade een *creature* kan oplopen vooral leer het dood gaat (en dus meestal onbruikbaar wordt). Aan de hand van deze definitie van een *creature* kunnen we een subklasse van *Card* schrijven met twee extra variabelen om een *creature* in code te modelleren. Dit is zichtbaar in Codefragment 3.8.

```

1 class Creature extends Card {
2   var _damage: Int = 0
3   var _health: Int = 0
4 }
```

Code Listing 3.8: Creature klasse

Om onze DSL uit te breiden zodat we volwaardige *creatures* kunnen aanmaken voegen we aan die klasse nog twee methodes toe, gelijkaardig aan de *called*-methode van de klasse *Card*. Zie Codefragment 3.9 voor de twee methodes en Codefragment 3.10 voor de aanmaak van een *Creature* met onze DSL.

```
1 class Creature extends Card {
```

```

2   var _damage: Int = 0
3   var _health: Int = 0
4   def with_damage(damage: Int): this.type = {
5     _damage = damage
6     this
7   }
8   def with_health(health: Int): this.type = {
9     _health = health
10    this
11  }
12 }
```

Code Listing 3.9: with_damage- en with_health-methodes

```
1 new Creature called "Stoneforge Mystic" with_damage 1 with_health 2
```

Code Listing 3.10: DSL creature creatie

Ook hier kunnen we gebruik maken van *implicits* (zie Codefragment 3.11) om de aanmaakt van een *creature* binnen onze DSL te reduceren tot Codefragment 3.12.

```

1 object CreatureImplicits {
2   implicit def String2Creature(name: String): Creature = {
3     new Creature called name
4   }
5 }
```

Code Listing 3.11: Implicite String naar Creature conversie

```
1 "Stoneforge Mystic" with_damage 1 with_health 2
```

Code Listing 3.12: Creature creatie met implicits

3.2.2 Het probleem met implicits

Er duikt nu echter een probleem op met de code in Codefragment 3.7. De compiler heeft nu de keuze uit twee verschillende impliciete methodes. De *String* kan zowel naar een instantie van de klasse *Card* als van de klasse *Creature* geconverteerd worden. Aangezien de compiler geen willekeurige keuze kan maken zal dat stuk code niet meer compileren. Dat we geen kaarten van de klasse *Card* kunnen aanmaken is op zich geen ramp, in een spel zullen er namelijk enkel subklassen gebruikt worden. Het is echter niet ondenkbaar dat er ook subklassen van de klasse *Creature* toegevoegd zullen worden waardoor, volgens het zelfde principe, geen instanties van de klasse *Creature* meer aangemaakt zouden kunnen worden met implicits. Door de inconsistentie in onze DSL over waar wel en waar geen impliciete conversie gebruikt

kunnen worden is het een beter idee om voor dit doeleinde het gebruik van implicits compleet te laten vallen.

3.3 Vaardigheden

En spel wordt interessanter naarmate het meer mogelijkheden heeft en indien de standaard spelregels met verschillende variaties doorbroken kunnen worden. Bij CCGs wordt het standaard gedrag van *creatures* aangepast door de toevoeging van vaardigheden (*abilities*). Vaardigheden hebben een invloed op de acties, zoals aanvallen en verdedigen, die een *creature* kan ondernemen. Vaardigheden kunnen tijdens het spel toegevoegd of verwijderd worden van *creatures* dus is het best om deze volledig los van de *Creature* klasse te implementeren. Om de vaardigheid daarna aan een *creature* toe te voegen maken we gebruik van het *decorator* ontwerppatroon, zie Codefragment 3.13.

```

1 class AbilityCreature(var creature: Creature) extends Creature {
2   def called(name: String): this.type = creature.called(name)
3   def with_damage(damage: Int): this.type = {
4     creature.with_damage(damage)
5   }
6   def with_health(health: Int): this.type = {
7     creature.with_health(health)
8   }
9 }
```

Code Listing 3.13: AbilityCreature klasse

Een *creature* met de vaardigheid *Flying* kan nu gemodelleerd worden zoals in Codefragment 3.14.

```

1 class FlyingCreature(val parent: Creature)
2   extends AbilityCreature(parent) {
3 }
```

Code Listing 3.14: FlyingCreature klasse

Het toevoegen van een vaardigheid aan een *creature* willen we natuurlijk ook voorzien in onze DSL. Hiervoor zullen we een extra methode in de *Creature*-klasse moeten schrijven die als werkwoord in onze DSL gebruikt kan worden. De methode, die we *has_ability* zullen noemen en die opgeroepen dient te worden op een bestaande instantie van de klasse *Creature* zal een parameter moeten hebben die aangeeft over welke vaardigheid het gaat. Verder moet de methode als resultaat een instantie van de juiste vaardigheidsklasse terug geven die rond het originele *creature* gewikkeld zit. Gelukkige schiet de functionele kant van Scala ons hier te helpen. Functies zijn in Scala ook instanties van een klasse, namelijk van de klasse *FunctionX*,

waarbij X het aantal parameters voorstelt. Als we nu een functie schrijven die een *Creature*-instantie als parameter heeft en een instantie van de juiste vaardigheidsklasse terug heeft, dan hebben we een geschikte parameter gevonden voor de methode *has_ability*. Een voorbeeld van dergelijke functie voor de vaardigheid *Flying* vinden we terug in Codefragment 3.15. De implementatie van de methode *has_ability* is terug te vinden op regels 12 tot en met 14 van Codefragment 3.16.

```

1 val Flying: Function1[Creature, FlyingCreature] = {
2   c => new FlyingCreature(c)
3 }
```

Code Listing 3.15: Flying vaardigheidsfunctie

```

1 class Creature extends Card {
2   var _damage: Int = 0
3   var _health: Int = 0
4   def with_damage(damage: Int): this.type = {
5     _damage = damage
6     this
7   }
8   def with_health(health: Int): this.type = {
9     _health = health
10    this
11  }
12   def has_ability(function: Function1[Creature, AbilityCreature]): AbilityCreature = {
13     AbilityCreature = {
14       function(this)
15     }
16 }
```

Code Listing 3.16: has_ability-methode

Aangezien we de vaardigheidsfunctie een eenvoudige naam meegegeven hebben, namelijk *Flying*, kunnen we in onze DSL een vaardigheid toevoegen aan een *creature* zoals in Codefragment 3.17.

```

1 new Creature called "Devouring Swarm" has_ability Flying
```

Code Listing 3.17: Vaardigheid toevoegen in de DSL

3.4 Vaardigheden met parameters

Naast de eenvoudige vaardigheden zoals *Flying* zijn er ook vaardigheden die zelf een of meerdere parameters bezitten. Een voorbeeld hiervan is de vaardigheid *Absorb X*. Een *creature*

met deze vaardigheid kan X schade absorberen vooralleer zijn levenspunten verminderd worden. De implementatie van bijhorende vaardigheidsklasse verschilt enkel van de standaard vaardigheidsklasse in dat ze een extra parameter meekrijgt (zie Codefragment 3.18).

```

1 class AbsorbCreature(val parent: Creature, x: Int)
2   extends AbilityCreature(parent) {
3 }
```

Code Listing 3.18: AbsorbCreature klasse

Ook de definitie van de bijhorende vaardigheidsfunctie kent een gelijkaardige uitbreiding, zoals te zien in Codefragment 3.19.

```

1 val Absorb: Function2[Int, Creature, AbsorbCreature] = {
2   i => c => new AbsorbCreature(c, i)
3 }
```

Code Listing 3.19: Absorb vaardigheidsfunctie

In onze DSL kan de *Absorb*-vaardigheid gebruikt worden zoals in Codefragment 3.20.

```

1 new Creature "Lymph Sliver" has_ability Absorb(1)
```

Code Listing 3.20: Vaardigheid met parameter toevoegen in DSL

Opgemerkt moet worden dat de *Absorb*-vaardigheid de waarde van X meekrijgt tussen haakjes. Hier zijn twee dingen aan de hand. Ten eerste zien we hier een voorbeeld van *currying*, een principe waarbij een functie slechts op een deel van de parameters wordt toegepast en een functie teruggeeft die de resterende parameters (in dit geval een *Creature*-instantie) als parameters neemt. Het resultaat hiervan is dat de *has_ability*-methode inderdaad een instantie van de klasse *Function1[Creature, AbilityCreature]* meekrijgt. De tweede opmerking is dat de haakjes rond de parameter niet weggelaten kunnen worden. De reden hiervoor is dan *FunctionX*-objecten een *apply*-methode gebruiken om hun functionaliteit toe te passen en dat Scala geen *syntactic sugar* voorziet voor het weglaten van ronde haakjes bij de *apply*-methode.

Hoofdstuk 4

Evaluatie

4.1 Creatures en vaardigheden

Tijdens de ontwikkeling van de codebasis en DSL voor *creatures* met vaardigheden werden vijf verschillende vaardigheden geïmplementeerd (zie Tabel 4.1). Uit een lijst van 46 andere vaardigheden die gebruikt worden in *Magic: The Gathering* Talk (2011) waren er zes die enkel betrekking hadden op *creatures* en vaardigheden, de reeds geïmplementeerde features. Van die zes kon er slechts één niet onmiddellijk geïmplementeerd worden omdat de vaardigheid een extra argument nodig had.

Tabel 4.1: Initiële vaardigheden

Vaardigheid	Beschrijving
Flying	Creatures met Flying kunnen enkel geblokkeerd worden door creatures met de vaardigheid Flying of Reach.
Reach	Creatures met Reach kunnen creatures met Flying blokkeren.
Shadow	Creatures met deze vaardigheid kunnen enkel creatures met Shadow blokkeren en enkel door heb geblokkeerd worden.
Trample	Creatures met Trample doen alle schade die na het blokkeren overschiet rechtstreeks aan de verdedigende speler.
Unblockable	Creatures met Unblockable kunnen niet geblokkeerd worden.

4.2 Vaardigheden met parameters

TODO (wat hier in de extended abstract stond is niet meer geldig).

4.3 Vaardigheden samenstellen

Het idee van het combineren van vaardigheden in een nieuwe vaardigheidsklasse kan met de huidige codebasis moeilijk gedaan worden. Het is eenvoudig om een functie (de compositie-operator) te schrijven die twee vaardigheden als argument neemt en de respectievelijke constructors na elkaar toepast, maar eens dit gedaan is bestaat er geen eenvoudige manier om te weten te komen of het huidige *creature* opgebouwd werd met de compositie-operator of eenvoudigweg door toevoeging van twee vaardigheden. Dit is echter een belangrijk verschil wanneer we vereisen dat vaardigheden van een *creature* weggehaald kunnen worden.

Hoofdstuk 5

Gerelateerd werk

5.1 SandScape

SandScape Knitter (2011) is de online, browsergebaseerde omgeving voor WTactics, “Een volledige vrij aanpasbaar kaartspel met grote strategische diepgang en prachtige looks” Snowdrop (2011). SandScape is een computerspel dat in een browser gespeeld wordt en zo goed als alle *collectible card games* aan kan. Dit wordt bereikt door geen spelregels op de leggen. De spelers kunnen zelf een kaartset importeren en spelen op een virtuele tafel. De rest van het spel is aan de spelers zelf. Zij moeten zichzelf spelregels opleggen en zorgen dat ze na geleefd worden. Door deze aanpak kunnen inderdaad praktisch alle *collectible card games* gespeeld worden, maar er ontbreekt uiteraard een grote vorm van automatisering. Zo zullen spelers bijvoorbeeld zelf hun levenspunten moeten aanpassen na elke succesvolle aanval van de tegenstander.

Dit is het compleet tegenovergestelde van een speciaal gebouwde computerversie van een CCG. Speciaal gebouwde computerversies kunnen elk aspect van het spel dat niet om gebruikersinteractie vraagt automatiseren, maar laten niet toe dat spelers hun eigen regels definiëren. Onze DSL bevindt zich ergens in het midden van beide opties. Door gebruik te maken van de DSL kunnen auteurs nieuwe CCGs maken, met een eigen set regels en kaarten, terwijl ze nog steeds kunnen profiteren van zo veel mogelijk automatisering.

5.2 Forge

Forge H. (2009) is een Java gebaseerde implementatie van *Magic: The Gathering*. De broncode is niet publiek beschikbaar, maar het spel kan wel aangepast worden door de spelers. Spelers kunnen hun eigen kaarten toevoegen door gebruik te maken van de Forge API Friarsol (2010), een scripting taal voor het definiëren van kaarten die geparst wordt door de Forge Engine. Een belangrijk deel van de API is de *Ability Factory*, een uitgebreide verzameling variabelen zoals *Cost*, *Target*, *Conditions* en vele anderen om vaardigheden en *spells*

te definiëren.

Aangezien deze scriptingtaal specifiek ontwikkeld werd voor het aanmaken van *Magic: The Gathering* kaarten is dit eigenlijk ook een vorm van een domein specifieke taal. De scriptingtaal laat wel enkel toe om kaarten aan te maken, waardoor ze minder krachtig is dan onze DSL, maar door het feit dat een scriptingtaal geparst wordt in plaats van gecompileerd is ze waarschijnlijk wel eenvoudiger om onder de knie te krijgen voor niet-programmeurs en spelers.

Hoofdstuk 6

Conclusie

TODO

Bibliografie

- Friarsol (2010). Forge api. http://www.slightlymagic.net/wiki/Forge_API. Accessed: 05/03/2013.
- R. Garfield (1993). Magic: The gathering. Introduced by: Wizards of the Coast.
- C. H. (2009). Forge. <http://www.slightlymagic.net/wiki/Forge>. Accessed: 05/03/2013.
- Knitter (2011). Sandscape. <http://sourceforge.net/projects/sandscape/>. Accessed: 05/03/2013.
- M. Odersky (2008). A tour of scala: Automatic type-dependent closure construction. <http://www.scala-lang.org/node/138>. Accessed: 26/04/2013.
- M. Odersky (2010). The scala programming language. <http://www.scala-lang.org/node/25>. Accessed: 26/04/2013.
- Snowdrop (2011). Wtactics. <http://wtactics.org/the-game/>. Accessed: 05/03/2013.
- W. G. Studios (2011). Shadow era. <http://www.shadowera.com/content.php?140-About8>. Accessed: 27/04/2013.
- Talk (2011). http://mtg.wikia.com/wiki/Keyword_Abilities. Accessed: 15/02/2013.