

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
3.1.1 De basisklasse	14
3.1.2 Toevoeging van een DSL sleutelwoord	15
3.1.3 Een eerste sleutelwoord	16
3.1.4 Nog een stap verder	16
3.2 Creatures	17
3.2.1 Aanmaak van Creatures	17
3.2.2 Implicite methodes	19
3.2.3 Het probleem met implicits	19
3.3 Vaardigheden	20
3.3.1 Vaardigheden als subklasse	20
3.3.2 Problemen met vaardigheden als subklassen	21
3.3.3 Vaardigheden als decorators	21

3.3.4	Vaardigheden in de DSL	24
3.4	Vaardigheden met parameters	25
3.5	Een nieuw type kaarten: Landen	26
3.5.1	De landkaarten	26
3.5.2	Landkaarten als vereisten voor andere kaarten	27
3.6	Uitvoerbare kaarten: Sorceries	31
3.6.1	Doelwitten	32
3.6.2	Samengestelde doelwitten	33
3.6.3	Acties	35
3.6.4	Acties en doelwitten toekennen aan kaarten	36
3.7	Overzicht van de DSL	37
4	Evaluatie	40
4.1	Creatures en vaardigheden	40
4.2	Vaardigheden met parameters	40
4.3	Vaardigheden samenstellen	41
5	Gerelateerd werk	42
5.1	SandScape	42
5.2	Forge	42
6	Conclusie	44
Bibliografie		45

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

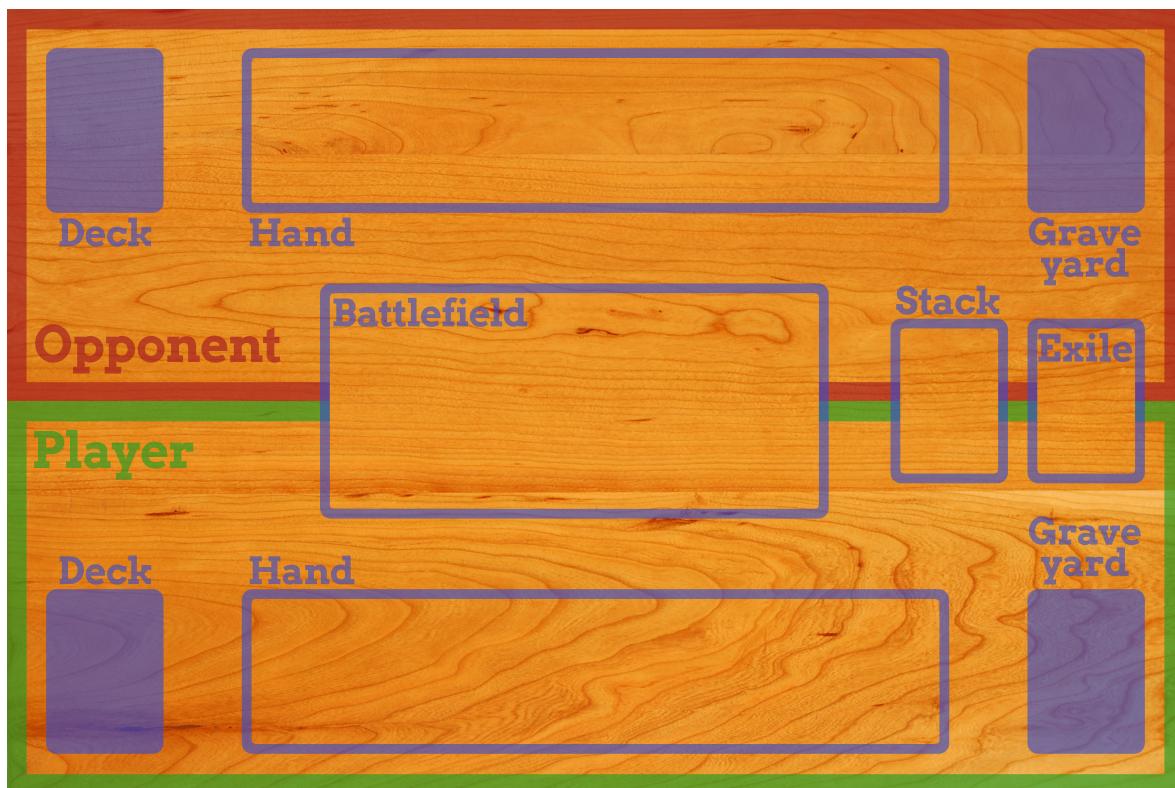
Achtergrond en motivatie

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.

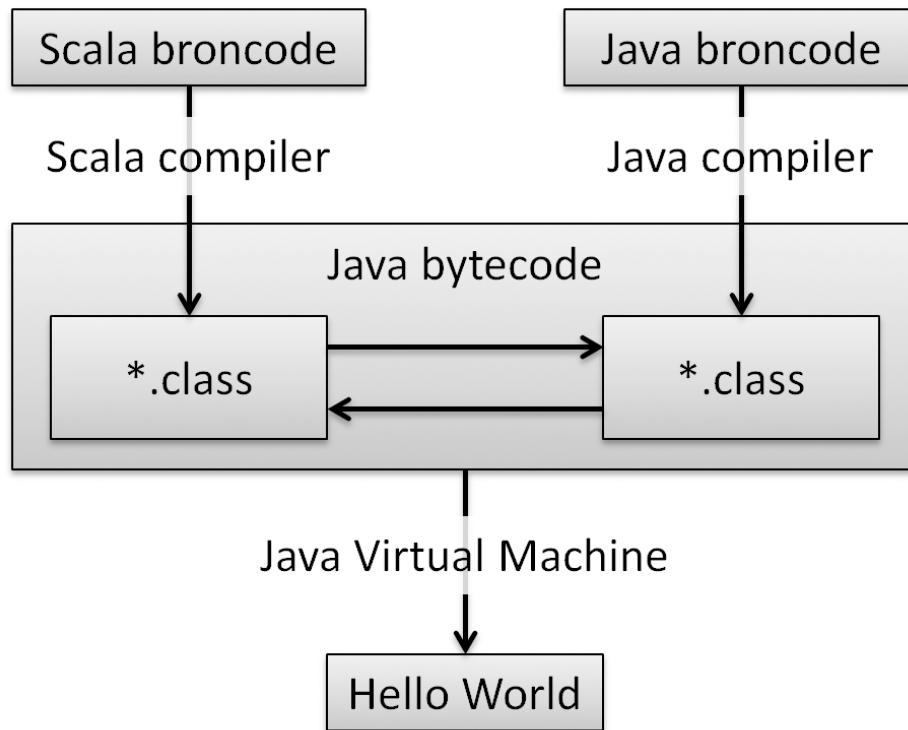
Het derde en laatste type van DSLs zijn de ingebedde domein-specifieke talen (*Embedded Domain Specific Languages* of EDSLs). Deze zijn ingebed in een programmeertaal en zijn vaak het gevolg van een slim gebruik van methodenamen en *syntactic sugar* aangeleverd door de programmeertaal. Indien de gastheertaal 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 }
```

Codefragment 2.1: Java klasse

```

1 class Person {
2 }
```

Codefragment 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

```

Codefragment 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

```

Codefragment 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

```

Codefragment 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

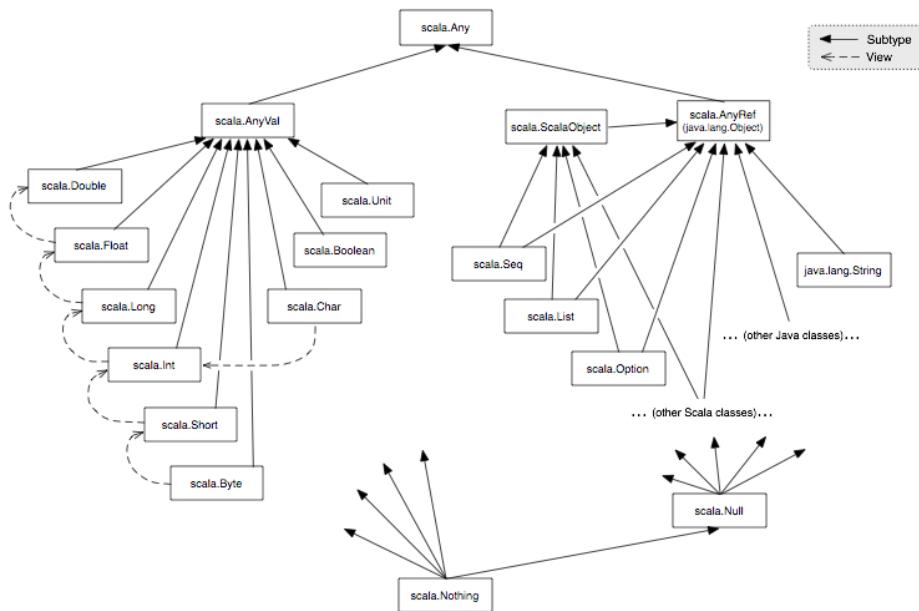
```

Codefragment 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
```

Codefragment 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 }
```

Codefragment 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 }
```

Codefragment 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
```

Codefragment 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)

```

Codefragment 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 }

```

Codefragment 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 verschillende stappen. In elke stap wordt de DSL uitgebreid met syntax die het mogelijk maakt om nieuwe functionaliteit te beschrijven, enkel gebruik makend van de DSL.

Het aspect van CCGs dat het meest onderhevig is aan verandering zijn de kaarten. Bij de meerderheid van CCGs is het zo dat er frequent uitbreidingen komen in de vorm van nieuwe sets kaarten. Deze nieuwe kaarten voegen niet zozeer nieuwe functionaliteit toe aan het spel, maar bestaan wel uit al dan niet unieke combinaties van reeds bestaande eigenschappen. Het doel van deze DSL is dan ook om nieuwe kaarten op een zo eenvoudig mogelijke manier te kunnen construeren en toe te voegen aan het spel.

In de secties die volgen wordt de DSL stap voor stap uitgebreid, tot we uiteindelijk in staat zijn om verschillende types van kaarten met een hele reeks eigenschappen te genereren aan de hand van eenvoudige DSL-constructies. Beginnen doen we met het basisconcept van een kaart: een object met een naam. Vervolgens splitsen we kaarten op in twee verschillende types: *Landen* en *Creatures*. Daarna verleggen we de focus tijdelijk naar *Creatures* door onze DSL uit te breiden met vaardigheden om vervolgens terug te keren naar een nieuw type van kaarten, namelijk *Sorceries*, met bijhorende acties (*Actions*) en doelwitten (*targets*).

3.1 Kaarten

3.1.1 De basisklasse

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. Een minimale klassedefinitie voor een kaart met een naam ziet er in Scala uit zoals in Codefragment 3.1.

```
1 class Card {
```

```

1
2   var _name: String = ""
3 }
```

Codefragment 3.1: Card-klasse

De toevoeging van kaarten is de meest frequente verandering aan een CCG, dus moet het in onze DSL heel eenvoudig zijn om het aanmaken van een nieuwe kaart uit te drukken. Indien we iemand (in het Engels) zouden vragen om een nieuwe kaart met de naam *Black Lotus* aan te maken, dan zou dit waarschijnlijk klinken als „*Create a new card called Black Lotus*”, waarbij we veronderstellen dat er een nieuw object van de klasse *Card* aangemaakt wordt met een naam-eigenschap die de waarde *Black Lotus* heeft. Indien we uit die zin de eerste twee woorden weg laten verliezen we geen informatie, aangezien het sleutelwoord *new* in objectgeoriënteerde programmeertalen (zoals Scala) duidt op de instantie (en dus creatie) van een nieuw object. In onze DSL willen we dus de zin „*new card called Black Lotus*” als statement voor de creatie van een nieuwe kaart kunnen gebruiken. Lettend op stijlconventies en het gebruik van letterlijke tekst versoepelen we deze wens tot: *new Card called "Black Lotus"*, waarbij de klasse *Card* een hoofdletter mee krijgt en letterlijke tekst tussen aanhalingstekens wordt geplaatst.

3.1.2 Toevoeging van een DSL sleutelwoord

Vertrekkend van de *Card*-klasse uit Codefragment ?? kunnen we een methode met de naam *called* toevoegen, die een naam als argument neemt (zie Codefragment 3.2). Deze methode laat ons toe om de naam-eigenschap van de *Card*-klasse in te vullen.

```

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

Codefragment 3.2: Called-methode

Twee features van Scala zorgen ervoor dat we met de huidige klasse een eerste stuk van onze DSL ontwikkeld hebben: Indien een constructor geen argumenten bevat dan laat Scala toe om de haakjes weg te laten. Daarnaast is de *called*-methode een methode met slechts één argument, waardoor Scala ons toe laat om de methode als infix-operator te gebruiken en bijgevolg alle interpunctie weg te laten. Zonder gebruik te maken van deze twee features zou het aanmaken van een instantie van de klasse *Card* gevolgd door het oproepen van de *called*-methode er uit zien zoals in Codefragment 3.3. Met de twee features kunnen we dit herleiden tot Codefragment 3.4.

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

Codefragment 3.3: Kaart creatie

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

Codefragment 3.4: Kaart creatie in DSL

3.1.3 Een eerste sleutelwoord

Er is echter nog één probleem met Codefragment 3.4. Na de oproep van de *called*-methode krijgen we geen instantie van de *Card*-klasse terug, maar wel een object van het type *Unit*, het Scala-equivalent van het *void*-return type in Java. Indien we de DSL in zijn huidige staat zouden gebruiken, dan zouden we in staat zijn om een nieuwe kaart met een naam te creëren, maar aangezien we nergens de referentie naar die instantie bewaren zou de kaart meteen weer verloren gaan. Om aan dit probleem tegemoet te komen moeten we ervoor zorgen dat de *called*-methode de instantie terug geeft waarop ze opgeroepen werd. Dit kunnen we doen door de definitie van de methode aan te passen zoals in Codefragment 3.5. Door de instantie zelf terug te geven als resultaat van de methode zorgen we ervoor dat ze in *method chaining*, het aaneenschakelen van methodes, kan gebruikt worden.

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

Codefragment 3.5: Called-methode met method chaining

Indien we nu het statement in Codefragment 3.4 opnieuw gebruiken, dan zal het resultaat een instantie van de klasse *Card* zijn met een correcte naam. Op deze manier is het woord **called** het eerste echte sleutelwoord van onze DSL.

3.1.4 Nog een stap verder

Binnen het domein van CCGs is een eenvoudige kaart volledig bepaald door haar naam. De eerste drie woorden uit het statement in Codefragment 3.4 zijn dus strikt genomen overbodig om een nieuwe kaart volledig te definiëren. Scala bevat een *feature*, namelijk impliciete conversies, die we kunnen gebruiken om aan de hand van enkel een string een instantie van de klasse *Card* terug te krijgen. Het is mogelijk om in Scala methoden te schrijven bij een klasse die vergezeld worden van het sleutelwoord *implicit*. Indien Scala tijdens de uitvoering van het

programma een statement tegen komt waarbij de types niet in orde zijn, dan zal Scala eerst proberen om die impliciete methoden toe te passen om zo het correcte type te verkrijgen. Pas indien ook al deze methoden een incorrect type opleveren zal er een typefout opgegooid worden.

Concreet willen we een instantie van de klasse *String* converteren naar een instantie van de klasse *Card*. Hiervoor schrijven we een methode *String2Card* en plaatsten deze in een object genaamd *CardImplicits*, zie Codefragment 3.6. Overal in de code waar we nu willen dat deze impliciete conversie beschikbaar is horen we nu een statische import te doen van het *CardImplicits*-object (Codefragment 3.7).

```

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

Codefragment 3.6: Impliciete String naar Card conversie

```
1 import CardImplicits._
```

Codefragment 3.7: Statisch importeren van impliciete methodes

Nieuwe kaarten kunnen nu eenvoudigweg aangemaakt worden zoals in Codefragment 3.8 .

```
1 val card: Card = "Black Lotus"
```

Codefragment 3.8: Kaart creatie met gebruik van implicits

3.2 Creatures

3.2.1 Aanmaak van Creatures

Kaarten met enkel een naam vormen natuurlijk nog geen interessant spel. Er is ook nood aan kaarten die vooruitgang in het spel brengen. 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* bijvoorbeeld, 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 vooralleer het dood gaat. 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. De implementatie hiervan is te zien in Codefragment 3.9.

```

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

Codefragment 3.9: 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 3.10 voor de twee methodes en Codefragment 3.11 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 }
```

Codefragment 3.10: with_damage- en with_health-methodes

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

Codefragment 3.11: DSL creature creatie

Op deze manier hebben we twee nieuwe sleutelwoorden, **with_damage** en **with_health** toegevoegd aan onze DSL. Om de leesbaarheid van de DSL te bevorderen bestaan de sleutelwoorden uit twee engelstalige woorden, samengevoegd met een underscore. Esthetisch had het mooier geweest om een spatie te kunnen gebruiken tussen te twee woorden, maar dat zou resulteren in een ongeldige methodenaam. Op deze manier hebben we twee nieuwe sleutelwoorden, **with_damage** en **with_health** toegevoegd aan onze DSL. Om de leesbaarheid van de DSL te bevorderen bestaan de sleutelwoorden uit twee engelstalige woorden, samengevoegd met een underscore. Esthetisch had het mooier geweest om een spatie te kunnen gebruiken tussen te twee woorden, maar dat zou resulteren in een ongeldige methodenaam.

Een tweede optie is om wel een spatie te gebruiken en de volledige methodenaam tussen backticks te plaatsen ('*with damage*'), maar om verwarring met letterlijke tekst te vermijden is de eerste optie de beste keuze.

3.2.2 Impliciete methodes

Analoog aan sectie 3.1.4 kunnen we ook voor de klasse *Creature* een impliciete conversie vanuit een String-instantie schrijven (Codefragment 3.12) en gebruiken zoals in Codefragment 3.13 .

```

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

Codefragment 3.12: Impliciete String naar Creature conversie

```

1 import CreatureImplicits._
2 val creature: Creature =
3   "Stoneforge Mystic" with_damage 1 with_health 2
```

Codefragment 3.13: Creature creatie met implicits

3.2.3 Het probleem met implicits

Er duikt nu echter een probleem op indien we instantie van de klasse *Card* en de klasse *Creature* door elkaar willen aanmaken met onze DSL. Zowel het object *CardImplicits* als het object *CreatureImplicits* moeten dan geïmporteerd worden. Indien we een instantie van de klasse *Card* willen aanmaken zoals in Codefragment 3.8 heeft Scala twee mogelijke impliciete conversies om uit te kiezen, aangezien *Creature* een subklasse is van *Card*. Aangezien Scala geen willekeurige keuze kan maken tussen te twee zal dit stuk code niet langer uitvoerbaar zijn.

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. Hierdoor wordt onze DSL inconsequunt op het gebied van impliciete conversies, dus 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 biedt 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 worden aan of verwijderd worden van een *creature*.

3.3.1 Vaardigheden als subklasse

De eerste reflex om vaardigheden te implementeren was als subklassen van de *Creature*-klasse. Op die manier kan een vaardigheidsklasse nog steeds als *creature* gebruikt worden en kan ze de nodige methodes van de *Creature*-klasse overschrijven om de effecten van de vaardigheid te implementeren. Een concreet voorbeeld maakt duidelijk hoe dit werkt.

Aan ons standaard *creature* kunnen we een methode *canBeBlockedBy*, met een tweede *creature* als argument, toevoegen die controleert of het tweede *creature* het eerste kan blokkeren bij een aanval. In de standaard implementatie, zie Codefragment 3.14 , is het antwoord op deze vraag „waar” (*true*).

```

1 class Creature extends Card {
2     // ... voorgaande implementatie
3     def canBeBlockedBy(other: Creature): Boolean = {
4         true
5     }
6 }
```

Codefragment 3.14: Creature met de canBeBlockedBy-methode

De vaardigheid *Flying* zouden we als volgt kunnen definiëren: „Een *creature* met de vaardigheid *Flying* kan enkel geblokkeerd worden door andere *creature* met de vaardigheid *Flying*”. Indien we deze definitie gebruiken om een klasse *FlyingCreature* te maken als subklasse van *Creature*, dan zou de implementatie er uit zien zoals in Codefragment 3.15 .

```

1 class FlyingCreature extends Creature {
2     override def canBeBlockedBy(other: Creature): Boolean = {
3         other.isInstanceOf[FlyingCreature]
4     }
5 }
```

Codefragment 3.15: FlyingCreature als subklasse van Creature

3.3.2 Problemen met vaardigheden als subklassen

Een groot nadeel van deze aanpak is dat subklassen niet dynamisch (tijdens runtime) toegevoegd kunnen worden aan, of verwijderd worden van, bestaande instanties van *creatures*. Om aan een instantie van de klasse *Creature* de vaardigheid *Flying* toe te voegen moet er een nieuwe instantie van de klasse *FlyingCreature* aangemaakt worden met exact dezelfde eigenschappen als de huidige toestand van het *Creature*. Dit is niet praktisch in gebruik maar vormt nog geen onoverbrugbaar probleem.

Een tweede probleem duikt echter op wanneer we meer dan één vaardigheid willen toekennen aan een *creature*. Om meerder vaardigheden te bezitten zou een *Creature* een instantie van meerdere vaardigheidsklassen tegelijk moeten zijn. Net zoals Java ondersteunt Scala echter geen meervoudige overerving aan de hand van het *extends*-sleutelwoord. In tegenstelling tot Java ondersteunt Scala wel een vorm van meervoudige overerving via het gebruik van *traits*. Traits worden op dezelfde manier geïmplementeerd als klassen, met als enige verschil dat ze het sleutelwoord *trait* gebruiken in plaats van *class*. *Traits* kunnen aan klassen toegevoegd worden door middel van het sleutelwoord *with*, analoog aan het sleutelwoord *extends* voor het toevoegen van een superklasse. Meerdere *traits* kunnen aan eenzelfde klasse toegevoegd worden.

In het voorbeeld van de *Flying*-vaardigheid zou een implementatie met *traits* er uitzien zoals in Codefragment 3.16 .

```

1 trait Flying {
2   def canBeBlockedBy(other: Creature): Boolean = {
3     other.isInstanceOf[Flying]
4   }
5 }
6 class FlyingCreature extends Creature with Flying {
7 }
```

Codefragment 3.16: Creature met Flying trait

Als we nu meerder vaardigheden aan een *creature* willen toevoegen kunnen we een subklasse van *Creature* schrijven die meerdere traits gebruikt. Voor elke combinatie van vaardigheden die we willen gebruiken hebben we echter een nieuwe klasse nodig en traits kunnen nog steeds niet dynamisch aan instanties toegevoegd worden, dus hiermee zijn onze twee problemen, namelijk *creatures* met meerder vaardigheden en het dynamisch toevoegen en verwijderen van vaardigheden, nog steeds niet van de baan.

3.3.3 Vaardigheden als decorators

Subklassen en traits voldoen niet aan onze wensen wanneer we vaardigheden willen implementeren. We willen dat vaardigheden dynamisch aan bestaande instanties toegevoegd kunnen

worden en dat we meerdere vaardigheden tegelijkertijd aan één *creature* kunnen koppelen. Beide vereisten kunnen geadresseerd worden door gebruik te maken van het *decorator* ontwerppatroon. Een decorator voor een basisklasse is een klasse die dezelfde interface als de basisklasse aanbiedt en bovendien een referentie naar een instantie van de basisklasse bijhoudt. Een decorator wordt gebruikt om de implementatie van een methode uit de basisklasse dynamisch aan te passen. Een voorbeeld van dit principe wordt getoond in Codefragment 3.17 . Regels 1-4 definiëren een basisklasse met twee methoden. Regels 5-8 definiëren een decorator met dezelfde interface als de basisklasse die de methode *a* niet overschrijft (de oproep wordt gewoon doorgegeven aan de instantie van de basisklasse) en die de methode *b* wel overschrijft met een eigen implementatie. Regels 12-14 tonen tenslotte hoe een decorator „rond” een basisklasse gewikkeld kan worden en vervolgens gebruikt.

```

1 class BasicClass {
2     def a() = println("BasicClass.a")
3     def b() = println("BasicClass.b")
4 }
5 class Decorator(basicClass: BasicClass) extends BasicClass {
6     override def a() = basicClass.a()
7     override def b() = println("Decorator.b")
8 }
9 var base: BasicClass = new BasicClass()
10 base.a() // "BasicClass.a"
11 base.b() // "BasicClass.b"
12 base = new Decorator(base)
13 base.a() // "BasicClass.a"
14 base.b() // "Decorator.b"
```

Codefragment 3.17: Het decorator ontwerppatroon

Om het decorator patroon nu concreet toe te passen voor vaardigheden implementeren we eerst een algemene decorator, *AbilityCreature*, die de complete *creature*-interface van een basisimplementatie voorziet. De implementatie ziet er uit zoals in Codefragment 3.18 . De vier methoden die al aanwezig waren in de *Creature*- en *Card*-klasse geven de controle door aan de gelijknamige methode van de gedecoreerde instantie *creature* van de *Creature*-klasse. Uiteraard zijn er voor de goede werking van het spel nog meer methodes nodig dan hier vermeld, maar deze vier (de drie DSL-sleutelwoorden en de methode *canBeBlockedBy*) geven de algemene vorm van de implementatie weer.

```

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     def canBeBlockedBy(other: Creature): Boolean = {
10        creature.canBeBlockedBy(other)
11    }
12 }
```

Codefragment 3.18: AbilityCreature klasse

De klasse *AbilityCreature* kan nu dienen als superklasse voor specifieke implementatie van vaardigheden. Codefragment 3.19 toont hoe de specifieke implementatie voor de vaardigheid *Flying* er uit ziet. Merk op dat enkel de methode *canBeBlockedBy*, de enige methode waar *Flying* een effect op heeft, overschreven hoeft te worden.

```

1 class FlyingCreature(val parent: Creature)
2     extends AbilityCreature(parent) {
3     override def canBeBlockedBy(other: Creature): Boolean = {
4         other.isInstanceOf[FlyingCreature] &&
5             parent.canbeBlockedBy(other)
6     }
7 }
```

Codefragment 3.19: FlyingCreature klasse

Zowel het vereiste van meerdere vaardigheden voor één *creature* en het dynamisch toevoegen en verwijderen van vaardigheden is nu opgelost, zoals Codefragment 3.20 aantoon. Op regel 1-4 wordt een eenvoudig *Creature* aangemaakt. Op regel 5 wordt de vaardigheid *Flying* toegevoegd, gevolgd door de vaardigheid *Unblockable* op regel 6. Ten slotte worden beide vaardigheden opnieuw verwijderd op regels 7 en 8.

```

1 var c: Creature = new Creature
2             called "Stoneforge Mystic"
3             with_damage 1
4             with_health 2
5 c = new FlyingCreature(c) // Flying toevoegen
6 c = new UnblockableCreature(c) // Unblockable toevoegen
7 c = c.parent // Unblockable wegnemen
8 c = c.parent // Flying wegnemen
```

Codefragment 3.20: Gebruik van vaardigheden

3.3.4 Vaardigheden in de DSL

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 sleutelwoord 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 hulp.

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 3.21. De implementatie van de methode *has_ability* is terug te vinden op regels 12 tot en met 15 van Codefragment 3.22.

```

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

Codefragment 3.21: 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 }
```

Codefragment 3.22: 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 3.23.

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

Codefragment 3.23: Vaardigheid toevoegen in de DSL

3.4 Vaardigheden met parameters

Naast de eenvoudige vaardigheden zoals *Flying* zijn er ook vaardigheden die zelf één 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 3.24).

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

Codefragment 3.24: AbsorbCreature klasse

Ook de definitie van de bijhorende vaardigheidsfunctie kent een gelijkaardige uitbreiding, zoals te zien in Codefragment 3.25.

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

Codefragment 3.25: Absorb vaardigheidsfunctie

In onze DSL kan de *Absorb*-vaardigheid gebruikt worden zoals in Codefragment 3.26.

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

Codefragment 3.26: 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.

3.5 Een nieuw type kaarten: Landen

Naast kaarten die het *Creature*-concept implementeren bevatten CCGs vaak nog andere types van kaarten. Sommige types van kaarten stellen benodigheden voor die beschikbaar moeten zijn om andere kaarten te kunnen spelen. *Magic: The Gathering* bevat een kaarttype *Land*. Er bestaan vijf verschillende types landen (*Forests*, *Islands*, *Mountains*, *Plains* en *Swamps*) zoals te zien in Figuur 3.1. Alle andere types kaarten in *Magic: The Gathering* vereisen dat er één of meerdere landkaarten, al dan niet van specifieke types, op het eigen battlefield aanwezig zijn voordat ze gespeeld kunnen worden. In *PokéMon* krijgt hetzelfde concept de naam *Energy* en het CCG *Shadow Era* voorziet geen nieuw type kaarten, maar gebruikt gewoon een welbepaald aantal willekeurige kaarten als vereiste voor het spelen van andere kaarten.



Figuur 3.1: Magic: The Gathering - Landtypes

Om deze functionaliteit te voorzien, zowel in ons spel als in onze DSL, moet er voldaan zijn aan volgende eigenschappen. Er moet een nieuw type kaart (*Land*) aangemaakt worden dat onderverdeeld is in vijf subtypes, zoals hierboven vermeld. Daarnaast moeten instanties van de landkaarten eenvoudig aan te maken zijn. Ten slotte moet het mogelijk zijn om bij andere types van kaarten het aantal landkaarten dat vereist is om de kaart te spelen, op te geven.

3.5.1 De landkaarten

Landkaarten hebben zelf geen functionaliteit. De enige vereiste die ze stellen is dat ze onderscheiden kunnen worden van andere types kaarten en dat verschillende landtypes onderling van elkaar onderscheiden kunnen worden. In een object-georiënteerde omgeving is het eenvoudig om in te zien dat een subklasse *Land* van de klasse *Card*, die zelf vijf subklassen bezit, aan deze vereisten kan voldoen. De implementaties zien er bijgevolg uit zoals in Codefragment 3.27 .

```
1 class Land extends Card {}
```

```

2
3 class Forest extends Land {}
4 class Island extends Land {}
5 class Mountain extends Land {}
6 class Plains extends Land {}
7 class Swamp extends Land {}

```

Codefragment 3.27: Landkaarten

Het aanmaken van instanties van deze landtypes kan via de constructor (Codefragment 3.28). Het aanroepen van deze constructor is eenvoudig genoeg zodat we deze meteen kunnen opnemen in onze DSL.

```

1 val land: Land = new Forest

```

Codefragment 3.28: Landcreatie

3.5.2 Landkaarten als vereisten voor andere kaarten

Magic: The Gathering gebruikt landkaarten als vereisten voor andere kaarten. De kaart *Bloodlord of Vaasgoth* heeft als vereiste bijvoorbeeld twee *Swamps* en drie landen van een willekeurig type, zoals te zien aan de symbolen in de rechterbovenhoek van Figuur 3.2. Meer algemeen zal de vereiste steeds een aantal (of nul) vaste landkaarten bevatten en een aantal (of nul) willekeurige landkaarten.

**Figuur 3.2:** Magic: The Gathering - Landen als vereiste voor andere kaarten

Het modelleren van vereisten doen we in twee stappen. Ten eerste implementeren we een klasse *Requirement*, de conceptuele voorstelling van een welbepaald aantal kaarten van een specifiek landtype, bijvoorbeeld „2 *Swamps*”. Uit de beschrijving valt af te leiden dat de klasse twee eigenschappen bevat: een aantal (*times*) en een landtype (*land*). Dit leidt tot de implementatie in Codefragment 3.29 .

```

1 class Requirement {
2   var times: Int = 0
3   var land: Land = null
4 }
```

Codefragment 3.29: Requirement-klasse

Om een *Requirement* aan te maken in onze DSL moeten we extra code voorzien. In het engels zouden we een *Requirement* benoemen met een zin van de vorm „2 Swamps”. Aangezien deze zin geen sleutelwoord bevat zullen we die exacte vorm niet in de DSL kunnen krijgen. Als sleutelwoord kiezen we dit keer de vermenigvuldigingsoperator *. Het tussenvoegen van deze operator leidt tot de zin „2 * Swamps”, wat nog steeds vrij natuurlijk overkomt en bijgevolg eenvoudig in gebruik is. De implementatie van de *-operator wordt toegevoegd aan de *Requirement*-klasse in Codefragment 3.30 .

```

1 class Requirement(val times: Int) {
2   var land: Land = null
3   def *(land: Land): Requirement = {
4     this.land = land
5     this
6   }
7 }
```

Codefragment 3.30: Requirement-klasse met *-operator

Merk op dat de methode opnieuw het *this*-object terug geeft, een vereiste voor de aaneenschakeling van methodes in onze DSL. Het argument van de operator is een instantie van (een subklasse van) de klasse *Land*. Indien we echter „*Swamp*” of „*Swamps*” willen schrijven als argument dan kunnen we, net zoals bij de naamgeving van vaardigheden (Codefragment 3.21) van een set waarden gebruik maken, zie Codefragment 3.31

```

1 object LandTypes {
2   val Forest: Forest = new Forest
3   val Forests: Forest = new Forest
4   val Island: Island = new Island
5   val Islands: Island = new Island
6   val Mountain: Mountain = new Mountain
7   val Mountains: Mountain = new Mountain
```

```

8  val Plain: Plains = new Plains
9  val Plains: Plains = new Plains
10 val Swamp: Swamp = new Swamp
11 val Swamps: Swamp = new Swamp
12
13 val Land: Land = new Land
14 val Lands: Land = new Land
15 }
```

Codefragment 3.31: Set namen voor land-instanties

Het gebruik van deze `*`-operator vereist wel dat het linkse term een instantie van de klasse *Requirement* is. Het getal 2 is dit duidelijk niet. We kunnen echter terug keren naar een eerder besproken feature van Scala, namelijk impliciete methoden, om het getal 2 naar een instantie van de klasse *Requirement* te converteren net voor de `*`-operator toegepast wordt. De implementatie van de impliciete conversie is te zien in [Codefragment 3.32](#). Deze conversie vereist een *requirement*-constructor die een getal als parameter neemt. Deze constructor werd stilzwijgend al in [Codefragment 3.30](#) toegevoegd.

```

1 object RequirementImplicits {
2   implicit def Int2Requirement(int: Int) = {
3     new Requirement(int)
4   }
5 }
```

Codefragment 3.32: Impliciete Integer naar Requirement conversie

De vereisten voor het spelen van een kaart bevatten meestal meer dan één type land. Dit kunnen we modelleren door een lijst van instanties van de klasse *Requirement* bij te houden. We doen dit in een speciaal hiervoor ontworpen klasse *Requirements*. We voegen meteen ook een *and*-operator toe om meerdere instanties van de klasse *Requirement* toe te voegen aan de *Requirements* en een object met een impliciete methode om één *Requirement* naar een instantie van de klasse *Requirements* te converteren. De resulterende code is te raadplegen in [Codefragment 3.33](#)

```

1 object RequirementsImplicits {
2   implicit def Requirement2Requirements(requirement: Requirement):
3     Requirements = {
4       new Requirements(requirement)
5     }
6   }
7 class Requirement extends ArrayBuffer[Requirement] {
```

```

8  def this(requirement: Requirement) {
9    this()
10   this += requirement
11 }
12 def and(other: Requirement): Requirements = {
13   this += other
14   this
15 }
16 }
```

Codefragment 3.33: Requirements-klasse

We kunnen in onze DSL nu een volledige set requirements beschrijven met een vrij leesbare syntax, zie Codefragment 3.34 .

```

1 val requirements: Requirements = 3 * Lands and 2 * Swamps
```

Codefragment 3.34: Requirements in de DSL

Het laatste wat we in onze DSL moeten voorzien is een sleutelwoord om de vereisten toe te voegen aan een kaart. De kaart moet natuurlijk een eigenschap hebben waarin de vereisten toegevoegd kunnen worden, daarom breiden we de klasse *Card* uit zoals in Codefragment 3.35 . Er werd ook een methode *requires* toegevoegd, die als sleutelwoord in onze DSL zal dienen.

```

1 class Card {
2   // ... Voorgaande implementatie
3   var _requirements: Requirements = new Requirements()
4   def requires(requirements: Requirements): this.type = {
5     _requirements = requirements
6     this
7   }
8 }
```

Codefragment 3.35: Card-klasse met Requirements

In de DSL kunnen er nu vereisten toegevoegd worden aan kaarten zoals in Codefragment 3.36

```

1 val card: Card = new Creature
2   called "Bloodlord of Vaasgoth"
3   requires 3 * Lands and 2 * Swamps
```

Codefragment 3.36: Gebruik van Requirements in de DSL

3.6 Uitvoerbare kaarten: Sorceries

Tot nu toe hebben we al twee verschillende types van kaarten besproken, landkaarten en *creatures*. De functionaliteit van deze types is duidelijk verschillende van elkaar. Landkaarten bezitten, behalve hun specifieke type, geen eigenschappen en nemen enkel passief aan het spel deel, als vereisten voor andere types van kaarten. *Creatures* daarentegen beschikken wel over een eigen set eigenschappen, zoals aanvalskracht en levenspunten, en kunnen de standaard spelregels beïnvloeden door de vaardigheden die eraan gekoppeld zijn.

Een derde type kaarten met een duidelijk verschillende functionaliteit zijn kaarten die eenmalige acties uitvoeren wanneer ze gespeeld worden. *Magic: The Gathering* bevat dit type van kaarten onder de vorm van *Sorceries* en *Instants*. Beide verschillen in het tijdstip waarop ze gespeeld kunnen worden. *Shadow Era* bevat *Ability*-kaarten met een soortgelijke functionaliteit en in *PokéMon* heten kaarten met dergelijke functionaliteit *Trainers*.

In dit spel zullen we kaarten met dergelijke functionaliteit *Sorceries* noemen, naar *Magic: The Gathering*. We gebruiken *Doom Blade* (Figuur 3.3), een kaart uit *Magic: The Gathering* als voorbeeld om te achterhalen hoe we *Sorceries* kunnen modelleren. Als we de kaart bekijken, dan zien we dat ze bestaat uit twee delen: Een actie, „Destroy”, en een doelwit, „nonblack creature”. Om de ontwikkeling van nieuwe kaarten vlot te laten verlopen vereisen we dat we in onze DSL *Sorceries* kunnen aanmaken en de actie en het doelwit meteen meegeven. Beide zullen dus gemodelleerd moeten worden. We beginnen bij de doelwitten.



Figuur 3.3: Magic: The Gathering - Doom Blade

3.6.1 Doelwitten

Mogelijke doelwitten voor *Sorceries* kunnen sterk van elkaar verschillen. Soms zijn *creatures* het doelwit, maar ook de spelers zelf kunnen het onderwerp van de acties vormen. Om naar doelwitten te kunnen refereren ontwerpen we bijgevolg een abstracte klasse *Target* (Codefragment 3.37) die een referentie naar het eigenlijke doelwit, een instantie van de klasse *Any*, bijhoudt.

```

1 abstract class Target {
2   var _target: Any = null
3   def targetable(target: Any): Boolean
4 }
```

Codefragment 3.37: Target-klasse

Subklassen van *Target* kunnen daarna geïmplementeerd worden om deelverzamelingen van mogelijke doelwitten te definiëren. Dit doen ze door een invulling te voorzien voor de abstracte methode *targetable*, die van een argument teruggeeft of het een geldig doelwit is. De klasse *CreatureTarget* (Codefragment 3.38) accepteert enkel instanties van de klasse *Creature* als geldige doelwitten, terwijl *PlayerTarget* (Codefragment 3.39) hetzelfde doet voor instanties van de klasse *Player*.

```

1 class CreatureTarget extends Target {
2   def targetable(target: Any): Boolean = {
3     target != null && target.isInstanceOf[Creature]
4   }
5 }
```

Codefragment 3.38: CreatureTarget-klasse

```

1 class PlayerTarget extends Target {
2   def targetable(target: Any): Boolean = {
3     target != null && target.isInstanceOf[Player]
4   }
5 }
```

Codefragment 3.39: PlayerTarget-klasse

Door de methode *targetable* een geschikte invulling te geven kunnen alle nodige doelwitten ontworpen worden. Om deze doelwitten beschikbaar te maken in onze DSL via een eenvoudige syntax voorzien we een *Targets*-object (zie Codefragment 3.40) dat namen voorziet voor alle geïmplementeerde doelwitten en statisch geïmporteerd kan worden waar doelwitten nodig zijn.

```

1 object Targets {
```

```

2   val Creature: CreatureTarget = new CreatureTarget
3   val Player: PlayerTarget = new PlayerTarget
4 }
```

Codefragment 3.40: Targets-object

3.6.2 Samengestelde doelwitten

Naast eenvoudige doelwitten zoals *creatures* en spelers kunnen *Sorceries* ook ingewikkeldere doelwitten hebben. Een voorbeeld hiervan is de kaart *Deathmark* uit *Magic: The Gathering*, te zien in Figuur 3.4. De tekst op de kaart, „*Destroy target green or white creature*”, duidt erop dat het doelwit van de *Sorcery* ofwel groene *creatures* ofwel witte *creatures* zijn. We zouden een klasse *GreenOrWhiteCreatureTarget* kunnen ontwerpen om naar dergelijk doelwit te refereren maar dat zou betekenen dat we voor alle mogelijke combinaties van basisdoelwitten zoals „*Creature*”, „*Green*” en „*White*” nieuwe klassen zouden moeten ontwerpen. Het is duidelijk dat deze manier van werken zeer gebruiksvriendelijk is.



Figuur 3.4: Magic: The Gathering - Deathmark

Om dergelijke doelwitten te modelleren zou het beter zijn indien we de basisdoelwitten at runtime kunnen samenstellen tot complexere doelwitten. Dit kan door een klasse *CompoundTarget* te voorzien die een lijst van (al dan niet zelf samengestelde) doelwitten bijhoudt, zie Codefragment 3.41 .

```

1 abstract class CompoundTarget extends Target {
```

```

2   val targets: ArrayBuffer[Target] =
3     new ArrayBuffer[Target]()
4 }
```

Codefragment 3.41: CompoundTarget-kLASSE

Deze klasse bepaald echter nog niet op welke manier doelwitten samengesteld worden. Dit kan op twee manieren. Enerzijds door conjunctie, bijvoorbeeld „green (and) creature”, anderzijds door disjunctie, zoals „green or white”. Deze manieren worden gemodelleerd door de respectieve klassen *ConjunctTarget* (Codefragment 3.42) en *DisjunctTarget* (Codefragment 3.43). Beide klassen bieden een invulling voor de methode *targetable*. De conjunctie gebruikt de methode *forall* om het resultaat voor de conjunctie van de elementen van de doelwitlijst te bepalen. De disjunctie gebruikt de *exists*-methode.

```

1 class ConjunctTarget extends CompoundTarget {
2   def targetable(target: Any): Boolean = {
3     targets .forall(_.targetable(target))
4   }
5 }
```

Codefragment 3.42: ConjunctTarget-kLASSE

```

1 class DisjunctTarget extends CompoundTarget {
2   def targetable(target: Any): Boolean = {
3     targets .exists(_.targetable(target))
4   }
5 }
```

Codefragment 3.43: DisjunctTarget-kLASSE

Aangezien samengestelde doelwitten gebruikt zullen worden bij de aanmaak van *Sorceries* moeten we voorzieningen treffen om dit in onze DSL vlot te laten verlopen. Dit kan opnieuw door toevoeging van impliciete conversies en methoden die als sleutelwoorden gebruikt kunnen worden. De keuze voor sleutelwoorden is logischerwijs *and* voor conjuncties en *or* voor disjuncties. Indien we beide klassen updaten zoals in Codefragment 3.44 en Codefragment 3.45 dan kunnen we samengestelde doelwitten in onze DSL aanmaken zoals in Codefragment 3.46 .

```

1 object ConjunctTargetImplicits {
2   implicit def Target2ConjunctTarget(target: Target):
3     ConjunctTarget = new ConjunctTarget(target: Target)
4   }
5 class ConjunctTarget extends CompoundTarget {
6   def this(target: Target) {
```

```

7   this()  

8     targets += target  

9   }  

10  def targetable(target: Any): Boolean = {  

11    targets forall(_ .targetable(target))  

12  }  

13  def and(target: Target): ConjunctTarget = {  

14    targets += target  

15    this  

16  }  

17 }

```

Codefragment 3.44: ConjunctTarget-klasse met DSL voorzieningen

```

1 object DisjunctTargetImplicits {  

2   implicit def Target2DisjunctTarget(target: Target):  

3     DisjunctTarget = new DisjunctTarget(target: Target)  

4 }  

5 class DisjunctTarget extends CompoundTarget {  

6   def this(target: Target) {  

7     this()  

8     targets += target  

9   }  

10  def targetable(target: Any): Boolean = {  

11    targets exists(_ .targetable(target))  

12  }  

13  def and(target: Target): DisjunctTarget = {  

14    targets += target  

15    this  

16  }  

17 }

```

Codefragment 3.45: DisjunctTarget-klasse met DSL voorzieningen

```

1 val target: Target = Green or White and Creature

```

Codefragment 3.46: Creatie van samengestelde doelwitten in de DSL

3.6.3 Acties

Nu we doelwitten op een eenvoudige manier kunnen beschrijven en aanmaken resten ons nog de acties om de *Sorceries* te vervolledigen. Een algemene actie vereist slechts één zaak,

namelijk dat er een manier aanwezig is om ze uit te voeren. Bijgevolg kunnen we een abstracte klasse *Action* implementeren die een zeer beperkte interface heeft, zie [Codefragment 3.47](#).

```

1 abstract class Action {
2   def execute: Unit
3 }
```

Codefragment 3.47: Action-klasse

Subklassen van deze klasse kunnen specifieke acties implementeren door de methode *execute* een zinvolle invulling te geven. Aangezien de meeste acties echter een doelwit verwachten ontwerpen we een tweede abstracte klasse, *TargetAction* ([Codefragment 3.48](#)) die deze functionaliteit implementeert door een doelwit bij te houden.

```

1 abstract class TargetAction extends Action {
2   var _target: Any = null
3 }
```

Codefragment 3.48: TargetAction-klasse

Een actie zoals *Destroy* kunnen we nu implementeren in een klasse *DestroyAction* die overerft van *TargetAction*, zie [Codefragment 3.49](#). Om in onze DSL op een eenvoudige manier naar de acties te kunnen verwijzen voegen we opnieuw een object toe dat namen geeft aan de acties ([Codefragment 3.50](#)).

```

1 class DestroyAction extends TargetAction {
2   def execute: Unit = {
3     target.asInstanceOf[Creature].destroy
4   }
5 }
```

Codefragment 3.49: DestroyAction-klasse

```

1 object Action {
2   val Destroy: DestroyAction = new DestroyAction
3   val Discard: Int => DiscardAction = {
4     x => new DiscardAction(x)
5   }
6 }
```

Codefragment 3.50: Actions-object

3.6.4 Acties en doelwitten toekennen aan kaarten

De laatste stap die ons nog rest om volwaardige *Sorceries* te kunnen ontwerpen is het toekennen van een actie en doelwitten aan een kaart. Hiervoor implementeren we een nieuwe klasse,

Sorcery, die overerft van de klasse *Card* aangezien het een kaart is. Deze klasse bevat twee eigenschappen: *_action* en *_target* die respectievelijk de actie en de doelwitten bijhoudt. Om *Sorceries* in onze DSL eenvoudig te kunnen aanmaken voorzien we bovendien twee methoden **defined_as** en **target** die gebruikt kunnen worden als sleutelwoorden in de DSL. De implementatie van deze klasse is te zien in Codefragment 3.51 en het aanmaken van een *Sorcery* met behulp van de DSL in Codefragment 3.52.

```

1 class Sorcery extends Card {
2     var _action: Action = null
3     var _target: Target = null
4     def defined_as(action: Action): this.type = {
5         _action = action
6         this
7     }
8     def targets(target: Target): Sorcery = {
9         _target = target
10        this
11    }
12 }
```

Codefragment 3.51: Sorcery-klasse

```

1 val sorcery: Sorcery = new Sorcery called "Doom Blade"
2     defined_as Destroy target NonBlack and Creature
```

Codefragment 3.52: Creatie van een Sorcery in de DSL

Op deze manier is ook het laatste grote type kaarten eenvoudig toe te voegen aan de hand van de DSL.

3.7 Overzicht van de DSL

Tot slot geven we nog een overzicht van de sleutelwoorden die de DSL bevat en hun doel in Tabel 3.1, de beschikbare operatoren voor het aanmaken van *Requirements* en doelwitten in Tabel 3.2, en namen die gedefinieerd werden voor instanties van landen, vaardigheden, acties en doelwitten in Tabel 3.3.

Sleutelwoord	Doel
new Creature	Aanmaak van een kaart van het type <i>Creature</i>
new Sorcery	Aanmaak van een kaart van het type <i>Sorcery</i>
new Forest	Aanmaak van een kaart van het type <i>Forest</i>
new Island	Aanmaak van een kaart van het type <i>Island</i>
new Mountain	Aanmaak van een kaart van het type <i>Mountain</i>
new Plains	Aanmaak van een kaart van het type <i>Plains</i>
new Swamp	Aanmaak van een kaart van het type <i>Swamp</i>
called	Geven van een naam aan een kaart
with_damage	Toekennen van een aanvalskracht aan een kaart van het type <i>Creature</i>
with_health	Toekennen van een aantal levenspunten aan een kaart van het type <i>Creature</i>
has_ability	Toekennen van een vaardigheid aan een kaart van het type <i>Creature</i>
requires	Toekennen van een reeks vereisten voor het spelen van een kaart van het type <i>Creature</i>
defined_as	Toekennen van een actie aan een kaart van het type <i>Sorcery</i>
target	Toekennen van een doelwit aan een kaart van het type <i>Sorcery</i>

Tabel 3.1: Sleutelwoorden in de DSL

<i>Requirements</i>	
Operator	Doel
*	Samenstellen van een aantal een een landtype
and	Samenstellen van verschillende <i>Requirements</i>
Doelwitten	
Operator	Doel
and	Conjunctie van doelwitten
or	Disjunctie van doelwitten

Tabel 3.2: Operatoren in de DSL

Acties	Destroy, Discard(X)
Doelwitten	Creature, Player
Vaardigheden	Absorb(X), Amplify, Battlecry, Bushido(X), Deathtouch, Defender, Exalted, Flanking, Flying, Frenzy(X), Graft(X), Horsemanship, Indestructible, Lifelink, Modular(X), Persist, Poisonous(X), Rampage(X), Reach, Shadow, Trample, Unblockable, Vanishing(X), Wither

Tabel 3.3: Namen in de DSL

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.