Gent, Juni 2013

De promotor	De begeleider	De auteur
Prof. dr. ir. T. Schrijvers	Benoit Desouter	Kenneth Saey

Woord vooraf

Todo

Kenneth Saey
Gent 3 Juni 2013

Inhoudsopgave

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 gereleased 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 programmeer taal die speciaal ontwikkeld werd op problemen binnen een goed gedefinieerd probleemdomein 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.

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.

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

Het spel *Shadow Era* bevat een gelijkaardig type van *creature*-kaarten, maar de vereiste om deze te kunnen spelen is hierbij niet een specifieke set van landkaarten, maar wel een voldoende grote voorraad *gronDSLoffen* die vergroot kan worden door andere kaarten op te offeren.
Andere kaarten kunnen dan weer effect hebben op de eigenschappen van kaarten, of op de algemene spelregels zelf. Het is dus belangrijk bij de implementatie van een CCG dat veranderingen aan eigenschappen en spelregels zonder ingewikkelde broncode geïmplementeerd kunnen worden.

Een derde niveau van modulariteit is dat vele kaarten eigenschappen hebben die zich gedragen als *vaardigheden*. 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*: Een wezen met de vaardigheid *Flying* kan tijdens een aanvalsactie enkel afgeblokt worden door wezens die ook de vaardigheid *Flying* bezitten. 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.

Het doel van deze masterproef is om een domein-specifieke taal te ontwikkelen 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 bovenop de zelfde 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 een 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 programmeertaal gebouwd bovenop de Java virtuele machine. Scala is object-georiënteerd maar bevat daarnaast paradigmas uit functionele programmeertalen. Bovendien bevat Scala een aantal features die de taal bijzonder geschikt maakt voor het ontwikkelen van DSLs die sterk lijken op natuurlijke taal.
De eerste is dat Scala geen beperkingen oplegt aan methodenamen. Een methode met de naam + is niet ongewoon en zorgt er onder andere voor dat (wiskundige) operatoren gedefinieerd kunnen worden voor zelfgeschreven klasses.
Daarnaast bevat Scala veel *syntactic sugar* voor het weglaten van interpunctie. De broncode

```
10.+(2)
```

roept bijvoorbeeld de methode + op het object *10* op met argument *2*, maar dankzij *syntactic sugar* is dit volledig equivalent met het statement

```
10 + 2
```

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, heten 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 (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 {
```

```
  var _damage: Int = 0
  var _healt: Int = 0
  def with_damage(damage: Int): this.type = {
    _damage = damage
    this
  }
  def with_health(health: Int): this.type = {
    _health = health
    this
  }
}
```

Code Listing 3.9: with_damage- en with_health-methodes

```
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.

```
object CreatureImplicits {
  implicit def String2Creature(name: String): Creature = {
    new Creature called name
  }
}
```

Code Listing 3.11: Impliciete String naar Creature conversie

```
"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 isntantie 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 subklasses gebruikt worden. Het is echter niet ondenkbaar dat er ook subklasses 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 lost 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.

```
class AbilityCreature(var creature: Creature) extends Creature {
  def called(name: String): this.type = creature.called(name)
  def with_damage(damage: Int): this.type = {
    creature.with_damage(damage)
  }
  def with_health(health: Int): this.type = {
    creature.with_health(health)
  }
}
```

Code Listing 3.13: AbilityCreature klasse

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

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

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 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 3.15. 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 _healt: 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]):
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: Vaardgiheid 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 geblokeerd 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, browsergebasseerde 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 *collectibel 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 nageleefd 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 gebasseerde 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.

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.

Snowdrop (2011). Wtactics. http://wtactics.org/the-game/. Accessed: 05/03/2013.

Talk (2011). http://mtg.wikia.com/wiki/Keyword_Abilities. Accessed: 15/02/2013.