A Domain Specific Language in Scala for Collectible Card Games

Kenneth Saey

Supervisor(s): Tom Schrijvers, Benoit Desouter

Abstract— **Collectible card games are a type of game with frequently changing game rules and new additions to gameplay. To make changes to the rules, authors have only to update the rulebook. In digital versions of collectible card games however, an update to the game rules is more than just changing english text, it also requires an update of the game code. This is impossible for the average game author. This article describes a programming language for coding game additions in an almost natural language.**

Keywords—**Domain specific language, Scala, collectible card games, runtime gameflow modification**

I. Introduction

Collectible Card Games (CCG), also known as Trading Card Games (TCG), like *Magic: The Gathering*, *Shadow Era* and preteen versions like *PokéMon* are fast evolving games, with new cards and game rules coming out multiple times a year. Since the late 1990s, CCGs have been turned into computer games. New versions of those computer games however, are released on a less regular basis. One of the reasons for this is that for every new card or game rule, new source code has to be written.

In an ideal world where computers are able to completely understand natural language, new cards could be added to the computer game on the fly by the author of the cards, only using the instructions printed on the cards. A solution right in the middle between writting actual source code an parsing and interpreting natural language is a Domain Specific Language (DSL). A DSL is a programming language designed to solve problems within one well-defined set of problems. This article explains how a DSL is be used to counter the fast evolving game mechanics of collectible card games.

Section II-A describes CCGs, their digital versions and why they are in need of an easy way to implement frequent changes. Section II-B explains the concept of DSLs and Section II-C is a short description of the programming language used to embed the DSL in: Scala. Our solution is explained in detail in Section III and evaluated in Section IV against different types of possible changes in game mechanics.

II. Background and Motivation

A. Collectible Card Games

Collectible card games are a type of card game for two or more players. A players cards reside in the players hand or on a gameboard. The gameboard is divided into different zones, all of which have a different effect on the cards they contain. Game rules define how cards change zones. Cards in the game can have a type, properties and abilities, all of which influence parts of the gameplay. CCGs are challenging from a software engineering point of view, since they contain many levels of modularity and many rules that make temporary changes to the general gameplay.

A first level of modularity are the different zones on the (virtual) gameboard in which cards can reside. These typically do not change in between different version of a game, but not every CCG has the same gameboard areas. Gameboard areas have an effect on the actions that can be executed with the cards on them.

Different types of cards provide a second level of modularity. It is not uncommon for new card types to be added to the game. Some card types are prerequisits for playing other cards, while others modify properties, or even gameplay. Especially when adding cards that modify properties and gameplay, it is important not having to change to much source code to implement those cards effects.

A third level of modularity is that many cards have properties called *abilities* (or the like). Abilities change gameplay in a non-intrusive way, meaning that the effect of the ability usually only lasts as long as the card with that ability is in play and that the effect is limited to the card itself. A good example would be the ability *Flying*: Cards with this ability can only be blocked by card which also have that ability.

The goal is to design a DSL which makes it easy enough for people with almost no programming experience to change any and all of these different levels of modularity. This allows not only for one game to be updated more frequently, but also for multiple CCGs to be developed on top of the same code base.

B. Domain Specific Languages

A Domain Specific Language is a programming language, specifically designed for solving problems in a well-defined, closed off, problem domain. It differs from a general-purpose language (GPL) in that a GPL should be able to solve problems in any given domain (of course with variable efficiency). There are three types of DSLs.

The most commonly known type of DSLs are the stand-alone DSLs. As the name says, a stand-alone DSL is a programming language, which has a dedicated compiler or interpreter to execute the code. Examples of stand-alone DSLs are HTML, for defining the structure of web pages and SQL, for writting queries for SQL-databases.

A second type of DSLs are embedded within other applications. Those DSLs cannot be used outside of the host application, and typically solve problems tightly bound to the host application. The functions that Excel provides for use inside spreadsheet cells are an example of an application embedded DSL.

The third and final type of DSLs are the Embedded Domain Specific Languages or EDSLs. They are embedded inside a host language and are usually created by cleverly using method

names and syntactic sugar provided by the host language. When the host language provides enough syntactic sugar, the EDSL will be able to look a lot like a natural language, which is a useful feature when new code has to be added quickly and often to existing source code by someone other than the author of the code.

C. Scala

Scala is a programming language built on top of the Java Virtual Machine. Scala is object-oriented, but at the same time integrates many functional programming paradigms. Scala has two features which make it especially fit for designing a DSL that greatly resembles natural (English) language. The first is that Scala puts no restrictions on method names. A method with the name + is not uncommon, and allows for defining (mathematical) operators on custom classes. Secondly, Scala contains a lot of syntactic sugar for leaving out punctuation. E.g.:

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

calls the method + on the object *10* with argument *2*, but, thanks to syntactic sugar all punctuation can be left out so that

```
10 + 2
```

is completely equivalent.

III. Solution

Designing a domain specific language for collectible card games was done in incremental steps. Each step containing more game functionalities and the accompanying parts of the DSL.

Step 1: Creatures and Abilities

The basis of every collectible card game is the concept of *creatures*. Creatures are a type of card used to directly influence the opponent's health counter. When the opponent's health is reduced to zero, the player wins the game. A creature has a health counter, an attack strength and, optionally, one or more abilities that influence the (game)actions available to that creature. Abilities can be added or removed during gameplay. This is reflected in the code by the use of the *decorator design pattern* in the super class of all abilities: *AbilityCreature*. See code listing 1. All abilities inherit from this class, which means they only have to override methods on which the ability has effect (Code Listing 2). To create the actual DSL for creatures with abilities a number of convenient methods where added tot the Creature class (Code Listing 3). To add abilities to creatures, the method **has_ability** can be used. This method has a method as argument that turns the Creature into an instance of an AbilityCreature. This method will have to be defined for every ability, as seen in Code Listing 4 When putting these four pieces of code together, and using Scalas syntactic sugar, a new creature can be created with the statement in Code Listing 5: A lot is going on here. First of all, syntactic sugar was used to leave out punctuation. With punctuation, Code Listing 5 is equivalent with Code Listing 6: These methods can be chained together, because each of them returns the modified object *this*. The **has_ability** methods uses an other Scala feature, called the apply-method. Each Scala class or companion object can have an apply method. This method is called when an instance of the class, or the companion object is followed by a pair of brackets. Using this feature the statement is translated to Code Listing 7: This is where Code Listing 4 comes into play. The apply method turns the current Creature into a Creature with the ability Flying. Because the companion object Flying extends a one-argument function itself, the braces of the apply method can be left out.

IV. Evalutation

Step 1: Creatures and Abilities

Five abilities where used while designing the code base and DSL for creatures with abilities. These where the abilities in Table I. From a list of 46 other abilities used in the game *Magic: The Gathering* [Talk, 2011], six relate only to creatures and abilities, the already implemented features. Of those six, only one could not directly be implemented in the same way, because the ability needed an argument.

TABLE I
Abilities

Ability	Description
Flying	Creatures with Flying can't be blocked except by other creatures with Flying andor Reach.
Reach	Creatures with Reach can block creatures with Flying.
Shadow	Creatures with this ability can only block or be blocked by other creatures with the Shadow ability.
Trample	Creatures with Trample may deal *excess* damage to the defending player if they are blocked.
Unblockable	Creatures with Unblockable can not be blocked by other creatures

V. Related Work

A. DSLs

Something about domain specific languages in general

B. Open Source Collectible Card Game

How others do it, http://wtactics.org/?, http://librecardgame.sourceforg

VI. Conclusion

To soon for a conclusion

References

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

```scala
class AbilityCreature(var creature: Creature) extends Creature {
  override def damage: Int = creature.damage
  override def health: Int = creature.health
}
```

Code Listing 1
ABILITY CREATURE

```scala
class Flying(val parent: Creature) extends AbilityCreature(parent) {
  override def canBeBlockedBy(creature: Creature): Boolean = {
    if (creature.hasAbility(classOf[Flying])) {
      parent.canBeBlockedBy(creature)
    } else {
      false
    }
  }
}
```

Code Listing 2
FLYING ABILITY

```scala
class Creature {
  var _name: String = ""
  var _health: Int = 0
  var _damage: Int = 0

  def called(name: String): this.type = {
    this._name = name
    this
  }
  def with_health(health: Int): this.type = {
    this._health = health
    this
  }
  def with_damage(damage: Int): this.type = {
    this._damage = damage
    this
  }
  def has_ability(function: Creature => AbilityCreature): AbilityCreature = {
    function(this)
  }
}
```

Code Listing 3
CREATURE CLASS

```
object Flying extends (Creature => Flying) {
  def apply(creature: Creature): Flying = new Flying(creature)
}
```

Code Listing 4
FLYING OBJECT

```
new Creature called "name" with_damage 4 with_health 5 has_ability Flying
```

Code Listing 5
CREATURE CONSTRUCTION

```
(new Creature).called("name").with_damage(4).with_health(5).has_ability(Flying())
```

Code Listing 6
CREATURE CONSTRUCTION WITHOUT SUGAR

```
(new Creature).called("name").with_damage(4).with_health(5).has_ability(Flying.apply())
```

Code Listing 7
CREATURE CONSTRUCTION WITH APPLY