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 DSL for collectible card games is a process best done in incremental steps, each one covering more game functionalities.

Step 1: Cards

Collectible card game are basically a card game, so cards are a logical first step to model, as shown in Code Listing 1. Cards are identified by a name. Since we want our DSL to resemble natural English language, the verb *to call* is used to name a card, see Code Listing 2. The called-method is an object function with one argument, so scala's syntactic sugar for infix operators lets us use it as seen in Code Listing 3. This is the very first part of a DSL for the definition of a collectible card game.

Step 2: Creatures

Named cards alone do not make a card game. Every card game has a need for some cards to progress the game. With CCGs this is usually accomplished by letting *creatures attack* the opponent. *Creatures* are a type of card. In a programming language this is easily modeled by creating a Creature-subclass of the Card-class. As a part of the example, creatures have some properties, like damage points and health counters (Code Listing 4). Damage points and health counters should be assignable at construction time, but since scala has no syntactic sugar for constructor arguments the easiest way to incorporate those assignations into the DSL is by providing a method (verb) for each of them (Code Listing 5) and call those immediatly after construction (Code Listing 6). By now you may have noticed that every *DSL-verb*-method returns the **this**-object. This allows for method chaining in scala, which is equivalent to *sentence-construction* in the DSL.

Step 3: Abilities

A game is made more interesting by adding modifiers to certain game aspects. In CCGs creatures are modified by the *abilities* they possess. Abilities have an influence on the game actions (like attacking and blocking) available to a creature. Abilities can be added to or removed from creatures during gameplay. Since *seperation of concerns* is a good property to have in any program, the best way to add an ability to a creature is to wrap an ability-class round a creature-class using the Decorator design pattern (Code Listing 7). Using this pattern, a creature with the ability *Flying* is modeled by a class *FlyingCreature*, as in Code Listing 8. To simply add an ability to a creature, another creature-method (DSL-verb) is needed. This time however, we can't simply return the **this**-object, since it needs to be wrapped inside a subclass of *AbilityCreature*. This problem can be solved by passing a function to the DSL-verb which does the actual wrapping. The definition of the DSL-verb can be viewed in Code Listing 9. Now we can add an ability to a creature in our DSL using the syntax in Code Listing 10. To get a readable DSL, we would like for *wrapper_function* in Code Listing 10 to be the name of an ability. Since it still has to be a function we can define a companion object for each new ability, as in Code Listing 11. With syntactic sugar, our DSL now looks like Code Listing 12.

Since the apply method of the ability companion object has basically the same implementation for every ability, it can be abstracted into an ability trait (Code Listing 13) which can then be inherited by the specific ability companion objects (Code Listing 14).

Step 4: Abilities with parameters

When we take a look at CCGs like [Garfield, 1993] we see that abilities themselves can have parameters. To accomodate this feature we need an extension of the ability trait which uses currying to process the extra parameter. How to do this can be seen in Code Listing 15. Code Listing 16 shows how an ability with argument can be used in the DSL.

IV. Evalutation

A. 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. The apply method of the ability trait contains reflection code with hard coded references to the packages in which abilities reside. This code could be removed if, for example, we would pass the name of the ability class to the method, but this would result in more code in the ability class and companion object. Since these last two would be written by the users of our framework the current solution is easiest to use for our users.

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

B. *Abilities with parameters*

Currying was used to combine the ability-parameter and the creature parameter into one apply-method. The same DSL syntax (Code Listing 16) can be achieved without currying by adding an apply-method which temporarily stores the ability-parameter inside the companion object (see Code Listing 17). But since currying is available in scala, using it is the better option.

C. *Ability composition*

The idea of ability composition, an easy way to wrap multiple abilities in one new abilities, is not easy to accomplish with our current code setup. It is easy enough to create a function (the composition operator) which takes two abilities as parameters and constructs the AbilityCreatures one after the other, but once this is done, there is no way to know whether the new creature has been constructed through the use of composition, or just by adding two abilities to it.

V. RELATED WORK

A. *SandScape*

SandScape [Knitter, 2011] is the online, browser based environment for WTactics, "A truly free customizable card game with great strategical depth and beautiful looks" [Snowdrop, 2011]. Basically, SandScape is a computer game played in a browser, which can handle almost any collectible card game. This is accomplished by imposing no game rules at all. Players can import sets of cards and get a virtual table top to place the cards on. The rest of the game is up to the players, they are the ones that enforce the rules and gameplay. This approach does indeed allow for almost every collectible card game to be played, but lacks in automation. For example: players themselves will have to edit their health points after each successful attack.

This is the complete opposite of a custom made computer version of a CCG. Custom made versions are able to automate every aspect of the CCG which does not require user interaction, but do not allow for players to impose their own rules.

Our DSL is somewhere in the middle of both of those options. Using the DSL alone, developers are able to create a new CCG, with its own set of rules and cards and still profit from as much automation as possible.

```scala
class Card {
  var _name: String = ""
}
```

Code Listing 1
CARD CLASS

```scala
class Card {
  var _name: String = ""
  def called(name: String): this.type = {
    _name = name
    this
  }
}
```

Code Listing 2
CARD CALLED

B. Forge

Forge [H., 2009] is a Java based implementation of *Magic: The Gathering*. The source code is not publicly available, but the game can be customised by players. Players are able to add their own cards to the game. This is done through the use of the Forge API [Friarsol, 2010], which is a scripting language parsed by the Forge Engine for defining cards. An important part of the API is the *Ability Factory*, an extensive set of variables like *Cost*, *Target*, *Conditions* and many more to completely define an ability (or spell).

Since this scripting language is specially designed for creating Magic: The Gathering cards, it is actually some form of a domain specific language. Because the scope of the scripting language only covers cards, it is less powerful than our DSL as to expression, but the fact that it is parsed instead of compiled makes it more accessible to non-programmers and players.

VI. Conclusion

To soon for a conclusion

References

[Friarsol, 2010] Friarsol (2010). Forge api. http://www.slightlymagic.net/wiki/Forge_API. Accessed: 05/03/2013.

[Garfield, 1993] Garfield, R. (1993). Magic: The gathering. Introduced by: Wizards of the Coast.

[H., 2009] H., C. (2009). Forge. http://www.slightlymagic.net/wiki/Forge. Accessed: 05/03/2013.

[Knitter, 2011] Knitter (2011). Sandscape. http://sourceforge.net/projects/sandscape/. Accessed: 05/03/2013.

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

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

```
new Card called "Some card name"
```

Code Listing 3
Called DSL

```
class Creature extends Card {
  var _damage: Int = 0
  var _health: Int = 0
}
```

Code Listing 4
Creature class

```
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 5
CREATURE WITH PROPERTY

```
new Creature called "Some creature name" with_damage 3 with_health 4
```

Code Listing 6
CREATURE DSL

```
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 7
ABILITYCREATURE CLASS

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

Code Listing 8
FLYINGCREATURE CLASS

```scala
class Creature extends Card {
  def has_ability(function: Creature => AbilityCreature): AbilityCreature = {
    function(this)
  }
}
```

Code Listing 9
CREATURE HAS ABILITY

```scala
new Creature called "Some creature name" has_ability wrapper_function
```

Code Listing 10
HAS ABILITY DSL

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

Code Listing 11
ABILITY COMPANION OBJECT

```scala
new Creature called "Some creature name" with_damage 3 with_health 4 has_ability Flying
```

Code Listing 12
FLYING DSL

```scala
trait Ability extends (Creature => AbilityCreature) {
  def apply(creature: Creature): AbilityCreature = {
    var objectName = this.getClass.getSimpleName()
    objectName = objectName.substring(0, objectName.size - 1)
    val className = objectName + "Creature"
    val c = Class.forName("masterproef.cards.abilities." + className)
    c.getConstructors()(0).newInstance(creature).asInstanceOf[AbilityCreature]
  }
}
```

Code Listing 13
ABILITY TRAIT

```scala
object Flying extends Ability
```

Code Listing 14

SPECIFIC ABILITY COMPANION OBJECT

```scala
trait Ability1Arg[T] extends Ability {
  def apply(x: T)(creature: Creature): AbilityCreature = {
    var objectName = this.getClass.getSimpleName()
    objectName = objectName.substring(0, objectName.size - 1)
    val className = objectName + "Creature"
    val c = Class.forName("masterproef.cards.abilities." + className)
    c.getConstructors()(0).newInstance(creature, x.asInstanceOf[Object])
      .asInstanceOf[AbilityCreature]
  }
}
```

Code Listing 15

ABILITY WITH ARGUMENT

```scala
object Absorb extends Ability1Arg[Int]
new Creature called "Some creature name" with_ability Absorb(4)
```

Code Listing 16

ABILITY WITH ARGUMENT DSL

```scala
trait Ability1[T] extends Ability {
  var _x: T = null.asInstanceOf[T]
  override def apply(creature: Creature): AbilityCreature = {
    var objectName = this.getClass.getSimpleName()
    objectName = objectName.substring(0, objectName.size - 1)
    val className = objectName + "Creature"
    val c = Class.forName("masterproef.cards.abilities." + className)
    c.getConstructors()(0).newInstance(creature, _x.asInstanceOf[Object])
      .asInstanceOf[AbilityCreature]
  }
  def apply(x: T): this.type = {
    _x = x
    this
  }
}
```

Code Listing 17

ABILITY WITH ARGUMENT WITHOUT CURRYING