A Domain Specific Language in Scala for Collectible Card Games

Kenneth Saey

Supervisor(s): Tom Schrijvers, Benoit Desouter

Abstract—**This article tries to come up with a domain specific language that can easily cope with the frequent changes seen in digital versions of collectible card games. The domain specific language is built in incremental steps, each of wich is able to handle more different sets of changes. Every version of the domain specific language is evaluated to determine which changes can be handled.**

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

I. Introduction

Collectible card games, like *Magic: The Gathering*, *Shadow Era* and pre-teen versions like *PokéMon* are fast evolving games, with new cards and game rules coming out multiple times a year. Since the late 1990s, collectible card games, also know as trading card games, 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, new cards could be added to the computer game on the fly, if the computer game was able to understand the instructions printed on the cards. A solution right in the middle between writting actual source code an parsing natural language is a domain specific language. This article will explain just how a domain specific language could be used to counter the fast evolving game mechanics of collectible card games.

In the next section an explaination is given why computer versions of collectible card games are in need of an easy way to implement new game mechanics, and how a domain specific language can help. Next examples will be used to explain, in detail, the workings of such a domain specific language, embedded within the host language Scala. In the section *Evolution*, the designed domain specific language is evaluated against different types of possible evolutions in game mechanics.

II. Background and Motivation

A. Collectible Card Games

Collectible (or Trading) card games (CCGs) are a type of game, usually between two players, in which one tries to defeat the other by playing cards with different properties and abilities. CCGs are interesting from a programmers 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 areas 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 game play. 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.

All of these levels of modularity are subject to change, so there is a need for a strong code base that can handle many changes with few or no changes to the actual code.

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 genreal-purpose language in that ad 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 DSL 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.

Secondly, some 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 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.

C. Scala

Scala is a programming language built on top of the Java Virtual Machine. Scala is object-oriented, partially because it integrates seemlessly with Java, and integrates at the same time 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 put no restrictions on method names. A method with the name "+" is not uncommon, and allows for class operators. Secondly, Scala contains a lot of syntactic sugar for leaving out language specific 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 uses to directly influence the opponents health-counter. When an opponents health is reduced to zero, the player wins the game. A creature has a health-counter and 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 pattern" in the super class of all abilities: "AbilityCreature". See code listing III.

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

All abilities inherit from this class, which means they only have to override methods on which the ability has effect.

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

To create the actual DSL for creatures with abilities a number of convenient methods where added tot the Creature class.

```
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 => Ability
        function(this)
    }
}
```

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 III

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

When putting these four pieces of code together, and using Scalas syntactic sugar, a new creature can be created with the statement:

```
new Creature called "name" with_damage 4 with_hea
```

A lot is going on here. First of all, syntactic sugar was used to leave out punctuation. With punctuation, the above is equivalent with:

```
(new Creature).called("name").with_damage(4).with
```

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:

```
(new Creature).called("name").with_damage(4).with
```

This is where code listing III 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 “Flying”, “Reach”, “Trample”, “Unblockable” and “Shadow”. From a list of 46 other abilities used in the game “Magic: The Gathering” [**?**], 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.

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.sourceforge.net/dokuwiki/doku.php?

VI. Conclusion

To soon for a conclusion