FinalDocument

DP Team

November 29, 2015

Contents

1 Overall System, How all these sub systems interact

The driver system gets everything going, and runs things at the highest level perspective. It's what reads in the initial options, sets the ScriptEvaluator, makes the ScriptEvaluator.

The driver then will let options get the correct file, and then get the lexer from the ScriptEvaluator, tokenize the file, eval the tokens. Then, it will take the RenderedCard list, give each card to output, with options. Output will then write the card to disk. The program is then done, exit, printing any information left in logger.

Note that during each step, the Driver checks logger for errors, and decides if it should abort.

This describes the highest level of how the systems interact. For detail of how the systems interact internally, see each system.

2 General Goals, and Overview

2.1 What features the system has

The goal of our project is to design a functional Script-driven card generator (SDCG) system that can output highly customizable cards. Using a lispe-esque inspired script language, the user can easily construct a multitude of unique card decks. The cards will be built from scratch using graphics, images, and text inputted in from the user. The images inputted are easily scalable and can be stretched or specified to maintain its original width, height, or even its aspect ratio. Our design allows for not only an easily customizable front layout, but also one for the back as well. Since most decks will have common patterns found throughout all the cards, we encourage users to make use of the layout component, which will enable the user to reuse and recycle similar base designs when making their cards. Another unique functionality of our design is the family component. With the use of families a user will be able to map user-made strings with specific images. This is useful when decks are divided into subdecks containing common images in each subdeck, because the added functionality this aspect gives is that cards with similar layouts but containing different images will be able to render the layouts along with the specified family. When a user renders a card, it asks for the family as well, and will render all the images that are mapped with any of the keys it finds. This will render the appropriate image based on the key the user built the card or layout with. Lastly, our script language allows

the construction of functions to help aid in process of creating your deck. Functions allow the user to create layouts with generic variable arguments that can be passed in when creating or rendering objects.

2.2 Example scripts

2.3 Example config file

2.4 Examples

2.4.1 52 cards

`Examples/52_deck_example/deckConfig.txt` CONFIG HAS TO BE REDONE, CONFIG IS NOW SCRIPT `Examples/52_deck_example/deckScript.txt`

2.4.2 Monopoly

`Examples/monopoly_example/monopoly.script` CONFIG HAS TO BE REDONE, CONFIG IS NOW SCRIPT `Examples/monopoly_example/monopoly.config`

3 Card Representation System (Ervis)

3.1 Overview

The Card object is constructed by five attributes: two Layouts for front and back, a name, a width and a height. The Layout is of object ComponentLayout, which could be composed of other layouts or Decals. A Decal is the base object that can be put on a Card object. A Decal can be made of either an Image, a Text, a DynamicDecal or a Shape object.

3.2 ScriptObject Class

Purpose: Interaction: Design Patterns it belongs:

3.3 ComponentLayout Class

Purpose: It is an abstract class which declares an interface for the objects that can belong to a layout such as a Decal or another layout. It will implement the default behaviours for some of the functionalities which will be the same in all the class that will inherit from this. It will declare an interface for accessing the child components.

Interaction: This abstract class will be inheriting from ScriptObject class which is another abstract class. In addition it will override the method necessary to build the layouts called acceptBuilder (). Two subclasses will inherit from this the LeafLayout class and CompositeLayout class.

Design Patterns it belongs: This class is designed using Composite design pattern and the iterator design patterns is included in it as well.

3.3.1 LeafLayout Class

Purpose: It is an abstract class that will represent the leaf object in a composition. It has no children and the leaf will be of type object Decal. Moreover, this class has character attribute to denote the size that the decal will have in relative to the layout. The options are specified in the ScriptingLanguageSpecification.org. It will be able to offer default behavior on how to render a decal. The render () method will be overrriden by the concrete subclasses fo the Decal class.

Interaction: This class will inherit from the ComponentLayout abstract class. In addition from this class we can navigate to the abstract class Decal.

Design Patterns it belongs: This class is designed using Composite design pattern.

3.3.2 CompositeLayout Class

Purpose: It is an abstract class that will declare an interface for components that will have children. In addition it will provide default implementation to some of the methods necessary for adding child layout components and for rendering layouts into the card or another layout. It will also provide a default implementation for the method iterator (), which will create an iterator to traverse all the composed layouts.

Interaction: This class will inherit from the ComponentLayout abstract class and it will serve as a base for the other two classes ArrayComponentLayout and SingleComponentLayout.

Design Patterns it belongs: This class is designed using Composite design pattern and the iterator design patterns is included in it as well to offer a way to access the composed layouts.

1. SingleComponentLayout Class

   Purpose: This is a concrete class that will be able to hold only a single child of type ComponentLayout. It will have zero or one thing as an attribute, a LayoutPositionPair, which will be discussed below.

Interaction: This class will inherit form the CompositeLayout abstract class. It will inherit the default implementation for the render method ().

Design Patterns it belongs: This class is designed using Composite design pattern and the iterator design patterns is included in it as well to offer a way to access the composed layouts.

2. ArrayComponentLayout Class

   Purpose: This is a concrete class that will be able to hold an array of children of types ComponentLayout. The array will be filled with zero or many references to the LayoutPositionPair obejcts.

   Interaction: This class will inherit form the CompositeLayout abstract class. It will inherit the default implementation for the render method (), addLayout (), removeLayout and iterator () to create an iterator.

   Design Patterns it belongs: This class is designed using Composite design pattern and the iterator design patterns is included in it as well to offer a way to access the composed layouts.

3.3.3 LayoutPositionPair Class

Purpose: It is a concrete class which will allow our system to create structure where a specific ComponentLayout is associated with a PositionScaled. This is necessary because every layout most be placed in a specific relatively scaled position.

Interaction: This class does not inherit from any other classes, however, the SingleComponentLayout and the ArrayComponentLayout will maintain zero or more references to the objects created by this class. **Design Patterns it belongs:** It is not part of any of the design patterns.

3.4 Decal Class

Purpose: This Abstract class will provide a common interface for various types of decals. I will also be implementing a default behaviors for the render () method and acceptBuilder () method, which will be overridden by the subclasses.

Interaction: This class will inherit from the ScriptObject class and four other subclasses will be implementing the rest of the functionalities defined by this abstract class.

Design Patterns it belongs: No design patterns were used in this part of the system.

3.4.1 Text Class

Purpose: A concrete class which comforts to the interface set by the Decal class. It will offer the user to put text objects into a card. I has three attributes a Color, a size and a font. This class will know how to render itself and how to load a text from a path given in the configuration file.

Interaction: It will inherit from the abstract class Decal.

Design Patterns it belongs: No design patters used in it.

3.4.2 ImageDecal

Purpose: A concrete class which offers the users to put an image into a card. This class will know how to render itself and how to load an image from a path given in the configuration file.

Interaction: It will inherit from the abstract class Decal.

Design Patterns it belongs: No design patterns used in it.

3.4.3 DynamicDecal

Purpose:

Interaction:

Design Patterns it belongs:

3.4.4 Shape Class

Purpose: This abstract class will provide a common interface for various shapes which conforming to the interface set by the Decal abstract class.

Interaction: It will be inheriting from the Decal abstract class and four other classes or more will be implementing its functionalities.

Design Patterns it belongs: No design patterns used in it.

1. Rectangle Class

   Purpose: It will be implementing the interface set by the Shape class and it will offer the user the ability to draw a rectangle on a card. It will also be implementing the render () method in order to render itself on a card. I has three attributes, two dimension and a Color type attribute.

Interaction: I will be implementing the interface set by the Shape class and also inheriting from it.

Design Patterns it belongs: No design patterns used in it.

2. Circle Class

   Purpose: It will be implementing the interface set by the Shape class and will offer users the ability to draw a circle with a specific radius and specific color on the card. It will override the render () method inherited from the Shape abstract class.

   Interaction: I will be implementing the interface set by the Shape class and also inheriting from it.

   Design Patterns it belongs: No design patterns used in it.

3. Triangle Class

   Purpose: It will be implementing the interfaces set by the Shape class and will offer the users the ability to draw a triangle with specific sides and specific color on the card. It will override the render () method.

   Interaction: I will be implementing the interface set by the Shape class and also inheriting from it.

   Design Patterns it belongs: No design patterns used in it.

4. AnyShape Class

   Purpose: It will be implementing the interface set by the Shape class and will offer the user to build any type of shape by just giving a set of points. The user is responsible for giving the correct amount of point and computing where those points should be.

   Interaction: I will be implementing the interface set by the Shape class and also inheriting from it.

   Design Patterns it belongs: No design patterns used in it.

3.5 Card Class

Purpose: Interaction: Design Patterns it belongs:

3.6 PositionScaled Class

3.7 Size Class

3.8 Family Class

3.9 ConsPair Class

3.10 List Class Class

3.11 ErrorScriptObject Class

3.12 NullScriptObject Class

3.13 DoubleScriptWrapper Class

3.14 StringScriptWrapper Class

3.15 UndefinedFunction Class

3.16 RenderedCard Class

3.17 Design Patterns used in Card Representation System

3.17.1 Why we used?

3.17.2 What we gained?

3.17.3 Consequences

4 Driver System (Includes Logger, output, options)

4.1 Options Reading system

4.1.1 for each class

1. Purpose
2. What patterns it partakes in

4.1.2 Design Patterns used

1. Why we used?
2. What we gained?
3. Consequences

4.2 for each class

4.2.1 Purpose

4.2.2 What patterns it partakes in

4.3 Design Patterns used

4.3.1 Why we used?

4.3.2 What we gained?

4.3.3 Consequences

4.4 Logging system

4.4.1 for each class

1. Purpose
2. What patterns it partakes in

4.4.2 Design Patterns used

1. Why we used?
2. What we gained?
3. Consequences

5 ScriptEvaluator system (Matt) (This includes all the builders, and the Lexer

When ever I say Builders, I mean subclasses of ScriptObjectBuilder

5.1 Builder

5.1.1 Why we used?

We have a ScriptObject which needs to be constructed, but the script objects vary quite a bit, and are all constructed differently. We can however, use a same general process, of first determining what to make, and then the arguments given. Thus, we can use a builder to separate the actual construction and representation from the construction process. The builder itself knows what to do from the parameters given, and the tokens return the correct

builder. The ScriptEvaluator then can run the same process for each builder to receive the script object result.

This process will look something like this.

```
ScriptObject doParse(Tokens token) {
    Builder builder = token.getBuilder(this);

    for (Token arg : token.getArgumentTokens()) {
        builder.addToken(this);
    }
    ScriptObject obj = builder.getResult();

    return obj;
}
```

Builder will define addToken something like this.

```
void addToken(Token token) {
    ScriptObject obj = eval.doParse(token);
    obj.accept(this);

}
//This will be overriden by some builders!
```

The Builder itself is only dependent on the ScriptEvaluator, which contains the minimium operations needed for the language.

This serves as the Director, and the implementor, CardLispScriptEvaluator, could potentially be replaced with a different one, allow the same builders to be used with a different language.

Most however, do not actually need it. One could simple give a null ScriptObjectEvaluator to those that do not need it. Or, one could make a constructor that automatically does this, to avoid the programmer having to worry. Or, split the ScriptObjectBuilder. Keep the existing base, but add another subclass, and give that one the eval. Then, only the Builders that need it would have it. If it wasn't late Thursday, I would do this, but the benefits are minor, if any. Since ScriptObject often need the environment for look ups, it some ScriptObjects could be made, but some couldn't. Thus, even if the dependency on the ScriptEvaluator was removed for some builders, we wouldn't know when something that does require it might be made. Transparency would be loss either way, and it doesn't make sense to try to build ScriptObjects outside the script.

1. To Summarize Thus, the builders can be used with a variety of languages, and some could be used anywhere, although doing so would cost some transparency. They effectively isolate building objects from the rest of the scripting language, and allow a uniform process to create them all. They enable easily changing the construction process for a new object, and adding new builders can be used to add new language features.

5.1.2 What we gained?

- Ability to easily change how a certain thing is constructed, just replace the builder
- The same process to construct all ScriptBuilders
- Can add new products by putting in new builders

5.1.3 Consequences

- Lots of builder classes, complicated design.
- Builder might be overkill for some simple objects constructed.
- Builder has access to script evaluator, which is needed for construction, but is some coupling.
  - Evaluator has a big larger interface than it should to allow this coupling with the builders.

5.1.4 Where used

In the ScriptObjectBuilder, and subclasses

5.2 Visitor

5.2.1 Why we used?

Needed to perform various operations across the various forms of ScriptObjects, both for rendering, and to construct ScriptObjects that contain ScriptObjects. Avoids need to cast when retrieving a ScriptObject from the environment, the ScriptObject tells the visiter what is being added.

5.2.2 What we gained?

- Ability to avoid casting when adding parameters, and retrieving variables from the environment.

5.2.3 Consequences

- Must modify the ScriptObjectBuilder class for each new ScriptObject made
  - However, since their is a default for adding, that is, to forward to addScriptObject for an unexpected/unneeded type, only the concrete builders that need to deal with this new type need to be modified, so in practice, not a big problem
- Visiter has lots of methods, potentially lots to inherit.

5.2.4 Where used

In the ScriptObjectBuilder, and subclasses.

5.3 State (Builders)

5.3.1 Why we used?

Most of the ScruptObjectBuilder concrete subclassess change what they do depending on what arguments are given in. Generally they need to choose how to create the thing they are supposed to build, based on arguments are given in. There are some cases where a builder might choose between a few different, but similar, concrete class based on the arguments. And some other of the ScriptObjects are only valid if certain arguments are given, and until then, the context doesn't know if valid arguments were given!

This results in a context that needs to mantain its state, and change what it does based on what arguments have been given in. The state pattern is an ideal fit for this. Also, most builders have an error state they will go to if an invalid sequence of arguments occurs.

5.3.2 What we gained?

- Builders isolate behavior in state.
  - No need for conditional logic to check what should be done, states handle this.

5.3.3 Consequences

- Many of the context (Builders) have to provide numerous extra operations to support the tight coupling between the two, some of which potentially violate state.
  - However, since the Builders will almost always be treated as their super class, ScriptObjectBuilder, which has a much tighter interface, this is a non-issue.
  - Additionally, since the states are to be implemented as inner classes, these operations need not be part of the public interface, again making it a non-issue.

5.4 FactoryMethod

5.4.1 Where used

In the ScriptEvalutor interface, getLexer. Returns a implementor of the Lexer interface.

While right now there is only one Lexer implementor, if another language were added, this would change.

5.4.2 Why?

If another language is added, then we will want to ensure we are using the correct scripting lexer for it. This ensures that with the parallel type hierarchy, the correct lexer and ScriptEvaluator are used. Since it is just a pair, an abstract factory is overkill, a single method will do.

5.5 Adapter (String/Double wrappers are tailored object adapters)

5.5.1 Why we used?

The Scripting language contains two types of Atom literals. These are numbers (doubles), and strings. We want to use Javas built in String and double type, but those can't be aggregated with the rest of the ScriptObjects. String could potentially be stored as common type Object, but then we'd lose the ScriptObject specific stuff. double could be boxed in Double, and then stored as object, but same issue. The solution, is to make tailored object adapters, one for each type. They each have just one operation to adapt, which is to

get the value. This lets the double and String be used with the rest of the ScriptObjects in the system.

5.5.2 What we gained?

double, and String can now be used with their Adapters as if they were any other ScriptObject sub type.

5.5.3 Consequences

- Inefficiency of an extra object, and an extra reference to follow.

5.6 Protection Proxy (String/Double wrappers)

5.6.1 Why we used?

These are constant values, they shouldn't be changed. (If set was added, then this would change, and we would need to add a set method to the proxies. This would still be good, as it would ensure the objects can only be changed one way.)

5.6.2 What we gained?

- String/Double ScriptObjects cannot be changed, and if that changes, it will be through one easily monitorable point.

5.6.3 Consequences

- Inefficiency of an extra object, and an extra reference to follow.

5.7 ScriptEvaluator and the Facade Patter

The ScriptEvaluator was originally going to just be a Facade. The ScriptObjectBuilder subclasses would be fine to use without it, and could be used separately on tokens. However, as the Environment got more complicated, and a current working directory path was needed, the ScriptObject became coupled with the Builders. A possible redesign would to make a data interface, which would be all the Builders dependended on, which could then enable the ScriptEvaluator to just be a Facade. However, I do not think that much would be gained from this, and while it is a fairly easy change to make, it is probably not worth the effort. A more worthwhile Facade could

be to make something that takes in a file path, runs the Lexer on it, then the ScriptEvaluator, but this would still be a fairly minor thing.

This being a fairly minor thing is the main reason I believe this not being a Facade is not a problem, the things it is doing are fairly simple. While it is interacting with a complex subsystem, the interactions are fairly simple.

5.8 Composite for ScriptObject and Token?

5.8.1 ScriptObject

ScriptObject and Token both feature recursive composition. However, for the ScriptObject, this is limited to just a few special cases, and the ScriptObject has no child management operations. Additionally, it's intent is not to represent part-whole hierarchies, or to let clients treat individuals/collections uniformly. It's intention is to provide a common type, with some common functionality for all objects that exist in the scripting language. Then, code can interact with these objects, and only know that it is some object from the scripting language, but not care exactly what it is.

- Not Composite pattern

5.8.2 Token

The Token features an ExpressionToken, which can have other tokens as arguments, and these can be further expression tokens. The rest of the Tokens are leaves. The Token type also contains basic child management, in the form of getting the list of arguments as tokens. Leaf tokens return an empty list. The intent is to enable an expression to be treated the same, regardless of whether it is a simple literal, a simple expression, or a bunch of sub expressions.

- Thus, this is an example of composite pattern

1. Why we used? Used to enable expression tokens to be made up of subexpressions, and for any piece of an expression to be treated the same when iterating through it, regardless of if it is a Variable, Expression, or Atom literal.

2. What we gained?

    - ScriptEvaluator is simple, it does a simple iteration through the tokens.

- When designing, was able to fairly easily split the original AtomToken into two subclasses, which fit better. Flexibility in adding Token types.
- Tokens are similar to existing textual structure of language, easy to parse into tokens.

3. Consequences
   - The getArguments() is unneeded for most tokens, which are leaves
     - But at least well defined, it's just empty!

5.9 Prototype (Builders)

5.9.1 Why we used?

Some of the Builders are paramtized and configured. (The FunctionBuilder is the main one). Additionally, need a way to get a new instance of the correct builders. One option is to store class objects, or a giant conditional statement for each builder. But the latter hard codes them, and makes it hard to add dynamically (needed for FunctionBuilder), and both don't allow builders that have been configured to be stored.

While the FunctionBuilder is the main one that needs this, to store the FunctionBody and arguments it is given, and then stored with, it allows flexibility for future builders. For example, a number operation builder might have one builder concrete class, that takes in the operation to do, +,-,/,*,etc, and then store that builder paramatized with each operation as a prototype.

5.9.2 Implementation note

For most of the builders, they are easy to clone. They are stored with their freshly constructed state, and don't have much to share. They can share the initial state, but upon changing state, the clone will get it's own. The only condition is that any change to the clone shouldn't effect the original. Lists should be cloned, but items don't need to be deep copied. ScriptObjects can be shared, as they are not changed after being constructed. (If the builder has the object it is constructing, and thus changing, then it should either set a new one, or deep copy it on clone.) Since Tokens are not changed, the FunctionBuilder can share these.

5.9.3 What we gained?

- Can store Builders in the factory easily, and retrieve them via cloning.

- Can change a builder to change an operation, and then store it under a new name, essentially adding a builder to the system.

5.9.4 Consequences

- Clone adds some complications.
  - Need to be careful of what can be shared, what must be deep copied.

5.10 Abstract Factory with Prototypes

5.10.1 Why we used?

5.10.2 What we gained?

- Enabled tokens to easily retrieve the Builder they need.
- To be able to store the Builders created for defined functions, and retrieve them as if they were the predefined Builders
- To lesson hard coding Builder types in tokens.
- Provide a central repository of the builder prototype.

5.10.3 Consequences

- Memory consequence, Builder prototypes use memory in the map.
- String comparisons can be more expensive time wise than hard coded class instances.

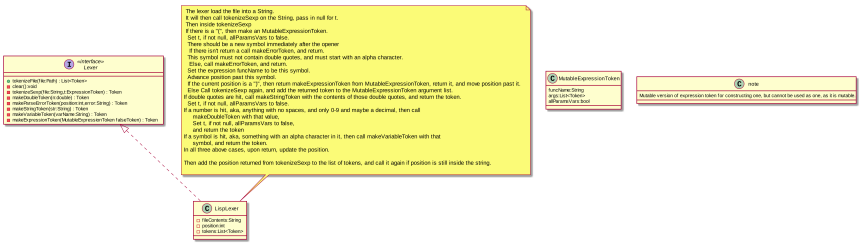
5.11 Lexer

5.11.1 Purpose

To take an input file, and return a list of tokens from it. A fairly simple interface, see the UML for decals.

To add support for additional scripting language, provided that the language can be represented with the existing token, just make a new implementor of the interface, and a corresponding ScriptEvaluator implementor!

5.11.2 UML



5.11.3 Token

- Tokens are immutable after creation.

1. Types
   (a) VariableToken
      - Leaf token, holds a var name.
      - Returns a VariableBuilderVisiter, with either an UndefinedVariable object if variable is not found, or the variable after looking it up in the env.

   (b) AtomToken
      - Returns a ConstantBuilderVisiter with the value wrapped in an appropriate ScriptObject.

      i. StringAtomToken
         - Leaf token, holds a string.

      ii. DoubleAtomToken
         - Leaf token, holds a double.

   (c) ParseErrorToken
      - Leaf token, represents an error that occured during parsing.
      - Return a ConstantBuilderVisiter, with an ErrorScriptObject as the value.

   (d) ExpressionToken
      - The composite of the layouts.
        – Holds other tokens, they are the arguments given to the expression.
        – Holds the name of the function invoked.
      - Looks for its builder in the environment.

- If found, return it!
- If none found, then make an UndefinedFunctionBuilder
  * If ExpressionToken is only variables, then return UndefinedFunctionBuilder with the given parameter names, and func name.
    · If the result of this Builder, an UndefinedFunction script object is given to a DefineBuilder as the first argument, that define builder will then define it in the env, so next time the funcname is found, it will be found in the environment!)
  * Else, return an Undefined Function Builder with an ErrorScriptObject.

2. Purpose To represent a the language in objects, rather than plain text. To destringify it. That logic can be put in one place, the lexer. Each token then knows what it is, and knows what builder to get. This separates the text representation of the scripting language, from the objects it creates.

5.12 ScriptEvaluator

5.12.1 Environment Subsystem

1. Environment An environment frame. Holds a map of strings to defined variables, and a BuilderFactory.

2. EnvrionmentList The Environment for the language. Contains a list of Environments, and operations to check from the most recently defined to the original, global env if a variable, or builder is defined. Can also manage and remove environments. Calling a function will add a frame to this, exiting a function removes said frame.

3. BuilderFactory Holds the builder prototypes in a map. Can add, and retrieve them from the map.

5.12.2 Builder subsystem

The meat of this system. ScriptObjectBuilder has numerous concrete builders. Generally, one for each ScriptObject subclass.

- See the UML for a complete list.

The goal of these builders is to know how to construct a ScriptObject. Adding a new object just requires adding a new builder for it, and then adding a new method. Only the appropriate sub Builders need to care about said new ScriptObject, so unlike with the usual visiter pattern, not all the Builder Visiters need to be updated.

They encapsulate building a script object.

See the Builder section earlier for more info.

5.12.3 RenderedCard

A simple POD class, holds the rendered images of the card, plus it's name.

5.13 Interactions

The ScriptEvaluator implementor is what will go through the tokens, and run the constructor process on the builder. The driver gets a Lexer from said implementor, and then uses that lexer to make Tokens. The driver then uses the ScriptEvaluator to eval those tokens. The tokens know what builder to make, and may do some small configuration to it. Then the builders get directed, as said above. When all the tokens are finished, the driver will retrieve the rendered cards.

5.13.1 CardRepresentation system and Decal interactions

- The ScriptEvaluator interacts with the CardRepresentation, and Decals fairly heavily.
  - The builders create decals, and place them in leaf-layouts.
  - The builders create, and assemble layouts.
  - The builders create cards, and give them layouts.
  - The builders create families.
  - Render will call .render on a card, with the given families.
    - After doing this, the result will be stored
  - All objects defined are stored in the environment of the ScriptEvaluator

5.14 Note on Lexer + ScriptEvaluator interfaces

Parallel hierarchy! For each language supported, there will be a ScriptEvaluator + a Lexer implementor pair for it!