FinalDocument

Matthew Bregg, Brandon Duong, Ian Fell, Ervis Shqiponja

December 5, 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, and 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 Plugins

Custom builders can be written, and then loaded in at runtime. See the config section of the Script Specification for details.

3 Turing Complete

To quote Wikipedia,

> Turing complete if it has conditional branching (e.g., "if" and "goto" statements, or a "branch if zero" instruction. See OISC) and the ability to change an arbitrary amount of memory locations (e.g., the ability to maintain an arbitrary number of variables).

```
Wikipedia contributors. "Turing completeness." Wikipedia, The Free
   Encyclopedia. Wikipedia, The Free Encyclopedia, 15 Oct. 2015. Web. 5 Dec. 2015.
```

- Our language has
  - If statements
  - Ability to define variables

4 General Goals, and Overview

4.1 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 lisp-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 imputed in from the user. The images imputed 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 for cards with similar layouts containing different images. 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.

4.2 Scripting Language Specifications

4.2.1 Scripting Language Specifications/Overview

For the scripting language, we have choosen a lisp style language. While far from a complete lisp, it has a syntax similar to scheme, and could easily be extended, without breaking existing scripts.

1. Built in functions
   (a) define

```
(define name value)
(define fooConstant 3)
(define (sqr x) (* x x))
```

Assigns to the name the given value. This version does support defining functions atm. (See `http://www.scheme.com/tspl2d/binding.html`)

- Right now, defines may be nested.
- A variable name can't start with a number, to make it determinable from an int atom.
  - It must start with an alpha character.

(b) Basic number ops Basic number ops, including +,-,/,*,^. Takes in two args, returns a third with the value.

```
(+ 1 2)
(- 1 2)
(* 1 2)
(/ 1 2)
(^ 1 2)
```

(c) concat-string Takes in n string, or numbers, returns one string of all those concattenated

```
(concat-string "foo" 3 "bar")
(concat-string "Foo" "bar")
```

(d) cons Takes two arguments, returns a pair holding the two arguments as one object

```
(cons a b)
(cons 1 2)
```

(See `http://download.plt-scheme.org/doc/4.2.4/html/guide/Pairs__Lists__and_Scheme_Syntax.html`)

(e) card Takes a card-size, name, and two layouts, one for the front, one for the back, and then the shape of the card.

```
(card card-size "name" frontLayout backLayout shape)
```

When a card is rendered, it will pair the layout with the wholeCard position-scaled (0 0 100 100)

(f) If statements The language has basic support for if statements. The language takes in three s expressions, the first is evaluated, if it is "#f", then the third token is evaluated, and the result returned. If it is anything else, then the second token is evaluated, and the result returned.

```
(define x...
(define y...
(if "#f" 3 4)
(if (== x y) (layout "foo") (layout "bar"))
```

i. Logic Operators The language supports two logic operators, `=, and !`. These return "#f", or "#t".

```
(!= "foo" "bar")
```

ii. NOTE Using "#f" and "#t" rather than #t #f, small variation from normal lisp.

(g) render Takes in a single card, a list of cards, or a pair of cards, and 0-n families. Renders them.

```
(render card family)
(render cards family)
(render (cons carda cardb) family)
(render some-cards family0 ... familyn-1)
```

Note: The family is an optional argument, leaving it empty is the same as calling

```
(render cards (family))
```

Which runs render with an empty family.

(h) list Takes in a n arguments, and returns a list of them.

```
(list N0 ... Nn-1)
(list 1 2 3 4 5 6 7)
```

(i) position-scaled Takes in a x-offset%, y-offset%, and a scale-width% and scale-height%, and returns a position-scaled object.

- The two scale arguments are optional, default to 100.

```
(position-scaled x-offset% y-offset% scale-width% scale-height%)
(position-scaled 0 0 50 50)
(define wholeCard (position-scaled 0 0 100 100))
```

(j) leaf-layout Returns a layout. Takes in a decal, or a string. In the event a string is given, the decal will be looked up in the family. This layout can then be used with the above layout function.

```
(layout image)
(layout foobarImage)
(layout "foo")
(layout (color-decal "white"))
```

i. Leaf-Layout options A Leaf-Layout can be given a third argument, to determine some extra behavior. Takes an extra parameter, either a W, or an H, A, or O.

- If W, width will be at most maximimum width of an image.
- If H, height will be at most, maximum height of given image
- If A, the original aspect ratio will be maintained.
- If O, original size will be mantained.
- IF S, stretch to fit.
- All the options aside S, which does need to, will add transparent padding to return a size render desires.
- The default is "S", so calling with the "S" argument is the same as not having a third argument

(k) layout Creates a Layout object. A layout contains 0-n tuples of layouts position-scaleds, and shapes. Takes 0-n tuples of layouts position-scaleds and shapes as arguments.

```
(Layout
  (list layout0 position-scaled0 shape0)
  (list layout1 position-scaled1 shape1)
  ...
  (list layoutn-1 position-scaledn-1 shapen-1))
```

```
(Layout
  (list layoutFoo position-scaledFoo rectanglebar)
  (list (layout foobarImage) wholeCard rectanglefoo)
  (list (layout "foo") (circle 3.14))
)
```

(l) family Creates a map of strings to decals, a family. Takes in a name, and N pairs.

- Requires a family name.
  - The family name is added to the card name when a card is rendered, to avoid name collisions when rendering the same card with multiple families.
  - If a multiple families given, append the names of all the families.

```
(family name pair0 ... pairn-1)
(family "fooFamily" (cons "foo" fooImage) (cons "bar" barImage))
```

(m) eval-file Takes in n filepaths, evals each file in given order

```
(eval-file "filename.filename")
(eval-file "foo.script")
(eval-file "foo.script" "bar.script")
```

Evals foo.script. Returns null.

(n) Decals

- Image Decal

```
(image "filepath.[jpg|png|etc]")
(image "foo.jpg")
```

- Color Decal
  - A decal takes in a color

    ```
    (color-decal "color")
    (color-decal "white")
    ```
- String decal

A string from a given font.

```
(string "StringText" "Font" "Color" Size)
(string "Hello World!" "Arial.font" "Red" 12)
(string "1" "Arial.font" "Red" 12)
```

- Mask Decal
  - Takes a decal, foo, and has the non transparent portions of foo replaced with corresponding portions of bar. What portion of bar maps to what portion of foo is determined by the position scaled.

    ```
    (define foo-decal (image "foo.png"))
    (define bar-decal (image "bar.png"))
    (mask-decal foo-decal bar-decal (position-scaled 0 0 100 100))
    ```
  - Example of effect, mask would leave transparent back ground.

- Inverted Mask Decal
  - Takes a decal, foo, and a decal bar, and has the non transparent parts of bar removed from foo, leaving a bar shaped hole in foo.
  - The position is used to determine where bar should be cut from foo.

```
(define foo-decal (image "foo.png"))
(define bar-decal (image "bar.png"))
(inverted-mask-decal foo-decal bar-decal (position-scaled 0 0 100 100))
```

  - Example: A circle hollowed out
- Rotate Decal
  - Takes in a decal, and a number, and returns a rotated version of that decal

```
(define foo-decal (image "foobar.png"))
(rotate-decal foo-decal 90)
;;Returns a decal rotates 90 degrees.
```

- Corner Rounding Decal
  - Takes a decal, and rounds the corners.

```
(define foo-decal (image "foobar.png"))
(corner-rounder foo-decal)
;;Returns a decal with its corners rounded.
```

- Crop Decal
  - Takes in a decal, and a position-scaled, crops the decal to the area the position-scaled defines.

```
(define foo-decal (image "foobar.png"))
(crop-decal foo-decal (position-scaled 50 50 50 50))
;;Returns a decal cropped to the middle.
```

(o) Shapes

- Rectangle

```
(rectangle width height)
(rectangle 100 200)
```

- Triangle

```
(triangle lengthA lengthB lengthC)
(triangle  100 200 300)
```

- AnyShape
  - Connect point0 -> point1, and then pointn-1 -> point0 to make a shape

```
(any-shape point0x point0y point1x point1y ... pointn-1x pointn-1y)
(any-shape  100 100 200 200 300 300)
```

- Circle

```
(circle radius)
(Circle 100)
```

(p) Position-Scaleds A position-scaled that can be used in the script

```
(position-scaled x-offset% y-offset% scale-width% scale-height%)
(position-scaled 0 0 100 100)
```

(q) Size A size is used by a card to determine how many pixels it will be.

```
(size width height)
```

4.2.2 Config file

- Allows one to set various options
- Current options are
  - script-file
    - Specify the script to run
    - No default, can be overridden by terminal args
  - output-format
    - Specify what format to output in
    - Defaults to png
  - output-file
    - Specify where to output the result to
    - Defaults to ./
  - logfile
    - Specify where to log to
    - Defaults to .cardlog
  - load-builder
    - Takes in a name, and the path to a builder java file.
    - Loads said builder into script evaluator
      `(set-option "load-builder" "name" "path")`
- Each option is enter in this format

```
(set-option "option-name" values)
```

-So for example

```
(set-option "output-dir" "./")
```

1. Standard Library The program will ship with numerous built in functions. These functions will have std- prepended to their name, and will provide various functionalities that preimplemented in the language, to save the user the hassle of implementing them. All these functions could be done normally, in the scripting language.

   This will be implemented in a std-functions.script, which will the driver will run through the evaluator when the program first runs.

   (a) add-border

      i. Parameters
         - input-layout : The layout to add a border to
         - color : The color to make the border
         - size : how thick to make the border

      ii. Return value
         - A layout with the border added

      iii. Example implementation

```
(define (std-add-border input-layout color size)
  (layout
   (list input-layout
         (position-scaled size size
                          (- 100 size) (- 100 size)))
   (list
    (leaf-layout (color-decal color)) whole-card)
   )
  )
```

   (b) get-colored-shape-in-a-layout

      i. Parameters
         - shape : The shape to put in the layout
         - color : the color to make it.

      ii. Return value
         - A layout with a leaf, which is paired with the shape, holding the color, over the whole layout.

      iii. Example implementaiton

```
(define (std-get-colored-shape-in-a-layout shape color)
  (layout
   (list
    (leaf-layout
     (color-decal color)) whole-layout shape)
   )
  )
```

(c) get-hollow-decal

i. parameters

- decal : The decal to be hollowed
- size : A number, determines how much to hollow

ii. Return value

- A decal that has been hollowed

iii. Example implementation

```
(define (std-get-hollow-decal decal size)
  (inverted-mask-decal decal decal
                       (position-scaled size size
                                        (- 100 size) (- 100 size)))
  )
```

(d) Some constants

- std-whole-layout : (position-scaled 0 0 100 100)
- std-pi : Value of pi

(e) Many more things!

4.3 Examples

4.3.1 52 cards

- Config

```
(set-option "script-name" "52CardDeckScript.script")
```

- Script

```
;;-------------------------------------------------------
;;size(in pixels) of the card
;;-------------------------------------------------------
(define cardSize (size 70 100))


;;-------------------------------------------------------
;;back image of the card
;;-------------------------------------------------------
(define cardBackImage (image "cardBack.png"))

;;-------------------------------------------------------
;;suit images
;;-------------------------------------------------------
(define spadeImage (image "spadeImage.png"))
(define heartImage (image "heartImage.png"))
(define diamondImage (image "diamondImage.png"))
(define clubImage (image "clubImage.png"))

;;-------------------------------------------------------
```

```
;;all face card images
;;-----------------------------------------------------
(define jackHeartImage (image "jackHeart.png"))
(define queenHeartImage (image "queenHeart.png"))
(define kingHeartImage (image "kingHeart.png"))
(define jackDiamondImage (image "jackDiamond.png"))
(define queenDiamondImage (image ("queenDiamond.png"))
(define kingDiamondImage (image ("kingDiamond.png"))
(define jackSpadeImage (image ("jackSpade.png"))
(define queenSpadeImage (image ("queenSpade.png"))
(define kingSpadeImage (image ("kingSpade.png"))
(define jackClubImage (image ("jackClub.png"))
(define queenClubImage (image ("queenClub.png"))
(define kingClubImage (image ("kingClub.png"))

;;-----------------------------------------------------
;;shapes
;;-----------------------------------------------------
(define rectangleCard (rectangle 70 100))
(corner-rounder rectangleCard)

(define rectangle (rectangle 70 100))

;;-----------------------------------------------------
;;relative positions and card size
;;-----------------------------------------------------
(define wholeCard (position-scaled 0 0 100 100))
(define numberTopLeft (position-scaled 5 5 5 5))
(define numberBotRight (position-scaled 85 85 5 5))
(define suitTopLeft (position-scaled 5 10 5 5))
(define suitBotRight (position-scaled 85 90 5 5))
(define centerFace (position-scaled 10 5 75 90))

;;-----------------------------------------------------
;;family for all 4 suits
;;-----------------------------------------------------
(define HeartFamily (family (cons "suit" heartImage)
                            (cons "jackImage" jackHeartImage)
                            (cons "queenImage" queenHeartImage)
                            (cons "kingImage" kingHeartImage)))
(define DiamondFamily (family (cons "suit" diamondImage )
                              (cons "jackImage" jackDiamondImage)
                              (cons "queenImage" queenDiamondImage)
                              (cons "kingImage" kingDiamondImage)))
(define SpadeFamily (family (cons "suit" spadeImage)
                            (cons "jackImage" jackSpadeImage)
                            (cons "queenImage" queenSpadeImage)
                            (cons "kingImage" kingSpadeImage)))
```

```
(define ClubFamily (family (cons "suit" clubImage)
                           (cons "jackImage" jackClubImage)
                           (cons "queenImage" queenClubImage)
                           (cons "kingImage" kingClubImage)))

;;-----------------------------------------------------
;;base layouts for all the cards
;;-----------------------------------------------------
(define (cardBackLayout)
  (layout
  (list ((layout cardBackImage) wholeCard rectangleCard))

(define (backgroundLayout value color)
  (layout
  (list ((layout whiteRectangle) wholeCard rectangle))
  (list ((layout "suit") suitTopLeft rectangle))
  (list ((layout "suit") suitBotRight rectangle))
  (list ((layout (string value "arial" color 12)) numberTopLeft rectangle))
  (list ((layout (string value "arial" color 12)) numberBotRight rectangle))))

(define (aceLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 40 40 20 20) rectangle))))

(define (twoLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 45 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 75 10 10) rectangle))))

(define (threeLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 45 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 75 10 10) rectangle))))

(define (fourLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))))

(define (fiveLayout value color)
  (layout
```

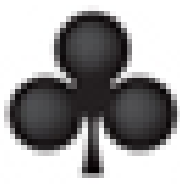
```
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 45 10 10) rectangle))))


(define (sixLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))))

(define (sevenLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 65 10 10) rectangle))))

(define (eightLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 45 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 65 10 10) rectangle))))

(define (nineLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 35 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 35 10 10) rectangle))
```

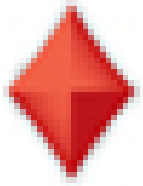
```
  (list ((layout "suit") (position-scaled 25 55 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 55 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 45 10 10) rectangle))))

(define (tenLayout value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "suit") (position-scaled 25 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 15 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 35 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 35 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 55 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 55 10 10) rectangle))
  (list ((layout "suit") (position-scaled 25 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 65 75 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 25 10 10) rectangle))
  (list ((layout "suit") (position-scaled 45 65 10 10) rectangle))))

(define (jackLayout  value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "jackImage") centerFace  rectangle))))

(define (queenLayout  value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "queenImage") centerFace rectangle))))

(define (kingLayout  value color)
  (layout
  (list ((backGroundLayout value color) wholeCard rectangle))
  (list ((layout "kingImage") centerFace rectangle))))

;;-------------------------------------------------------
;;cards
;;-------------------------------------------------------
(define blackAceCard
  (card cardSize "blackAce" (aceLayout "A" #000000) cardBackLayout))
(define blackTwoCard
  (card cardSize "blackTwo" (twoLayout "2" #000000) cardBackLayout))
(define blackThreeCard
  (card cardSize "blackThree" (threeLayout "3" #000000) cardBackLayout))
(define blackFourCard
  (card cardSize "blackFour" (fourLayout "4" #000000) cardBackLayout))
(define blackFiveCard
  (card cardSize "blackFive" (fiveLayout "5" #000000) cardBackLayout))
```

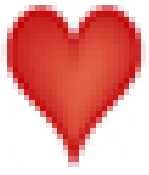
```
(define blackSixCard
  (card cardSize "blackSix" (sixLayout "6" #000000) cardBackLayout))
(define blackSevenCard
  (card cardSize "blackSeven" (sevenLayout "7" #000000) cardBackLayout))
(define blackEightCard
  (card cardSize "blackEight" (eightLayout "8" #000000) cardBackLayout))
(define blackNineCard
  (card cardSize "blackNine" (nineLayout "9" #000000) cardBackLayout))
(define blackTenCard
  (card cardSize "blackTen" (tenLayout "10" #000000) cardBackLayout))
(define blackJackCard
  (card cardSize "blackJack" (jackLayout "J" #000000) cardBackLayout))
(define blackQueenCard
  (card cardSize "blackQueen" (queenLayout "Q" #000000) cardBackLayout))
(define blackKingCard
  (card cardSize "blackKing" (kingLayout "K" #000000) cardBackLayout))

(define redAceCard
  (card cardSize "redAce" (aceLayout "A" #DD311D) cardBackLayout))
(define redTwoCard
  (card cardSize "redTwo" (twoLayout "2" #DD311D) cardBackLayout))
(define redThreeCard
  (card cardSize "redThree" (threeLayout "3" #DD311D) cardBackLayout))
(define redFourCard
  (card cardSize "redFour" (fourLayout "4" #DD311D) cardBackLayout))
(define redFiveCard
  (card cardSize "redFive" (fiveLayout "5" #DD311D) cardBackLayout))
(define redSixCard
  (card cardSize "redSix" (sixLayout "6" #DD311D) cardBackLayout))
(define redSevenCard
  (card cardSize "redSeven" (sevenLayout "7" #DD311D) cardBackLayout))
(define redEightCard
  (card cardSize "redEight" (eightLayout "8" #DD311D) cardBackLayout))
(define redNineCard
  (card cardSize "redNine" (nineLayout "9" #DD311D) cardBackLayout))
(define redTenCard
  (card cardSize "redTen" (tenLayout "10" #DD311D) cardBackLayout))
(define redJackCard
  (card cardSize "redJack" (jackLayout "J" #DD311D) cardBackLayout))
(define redQueenCard
  (card cardSize "redQueen" (queenLayout "Q" #DD311D) cardBackLayout))
(define redKingCard
  (card cardSize "redKing" (kingLayout "K" #DD311D) cardBackLayout))

;;-----------------------------------------------------
;;makes the subDeck of 13 cards, for each color
;;-----------------------------------------------------
(define redSubDeck (list redAceCard redTwoCard redThreeCard redFourCard
```

```
                          redFiveCard redSixCard redSevenCard
                           redEightCard redNineCard redTenCard redJackCard
                           redQueenCard redKingCard ))
(define blackSubDeck (list blackAceCard blackTwoCard blackThreeCard blackFourCard
                            blackFiveCard blackSixCard blackSevenCard
                           blackEightCard blackNineCard blackTenCard blackJackCard
                           blackQueenCard blackKingCard ))

;;-------------------------------------------------------
;;renders the subdeck 4 times for each suit
;;-------------------------------------------------------
(render redSuitSubDeck heartFamily)
(render redSuitSubDeck diamondFamily)
(render blackSuitSubDeck clubFamily)
(render blackSuitSubDeck spadeFamily)
```

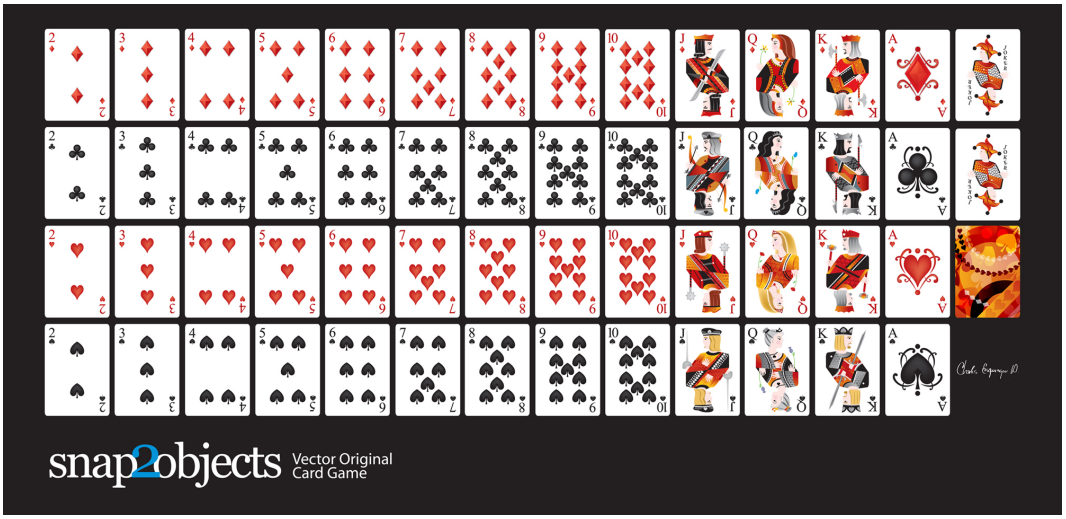
1. Data files







2. Mock-Ups



4.3.2 Monopoly

- Config

```
(define blackRectangle (rectangle 0x000000))

(define railroad_image "./images/railroad.gif")
(define electric_company_image "./images/electric_company.png")
(define water_works_image "./images/water_works.png")

;; define the low level layouts
(define (BLACK_BOARDER)
  (layout
   (cons (layout blackRectangle) (position-scaled 5 5 85 20))
   ;; create a black rectangle
   (cons (layout blackRectangle) (position-scaled 4.5 4.5 85 20))
   ;; creating an arbitrary colored rectangle
   )
  )

(define (HEADER color roadName)
  (layout
   (cons (layout blackRectangle) (position-scaled 10 7 80 13))
   ;; create a black rectangle
   (cons (layout (rectangle color)) (position-scaled 10.5 7.5 79 12))
   ;; creating an arbitrary colored rectangle
   (cons (layout (string "TITLE DEED" "Arial" "0x000000" 10) "W")
         (position-scaled 30 15 10 10))
   ;; create a string TITLE DEED
   (cons (layout (string roadName "Arial" "0x000000" 20) "W")
         (position-scaled 25 20 20 20))
   ;; create a string AVE
   )
```

```
  )

(define (BASE_RENT rent)
  (layout
   (cons (layout (string "RENT" "Arial" "0x000000" 12) "W")
         (position-scaled 40 40 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 45 40 100 100))
   )
  )

(define (WITH_HOUSES rent number)
  (layout
   (cons (layout (string (+ "WITH " number " House(s)") "Arial" "0x000000" 12) "W")
         (position-scaled 20 (+ 45 (* number 5)) 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 70 (+ 45 (* number 5)) 100 100))
   )
  )

(define (WITH_HOTEL rent)
  (layout
   (cons (layout (string "WITH HOTEL" "Arial" "0x000000" 12) "W")
         (position-scaled 40 70 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 45 70 100 100))
   )
  )

(define (MORTGAGE m)
  (layout
   (cons (layout (string "MORTGAGE VALUE" "Arial" "0x000000" 12) "W")
         (position-scaled 40 75 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 45 75 100 100))
   )
  )

(define (HOUSE_COST c)
  (layout
   (cons (layout (string "HOUSES COST" "Arial" "0x000000" 12) "W")
         (position-scaled 40 80 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 45 80 100 100))
   (cons (layout (string "each" "Arial" "0x000000" 12) "W")
         (position-scaled 50 80 100 100))
   )
  )
```

```
(define (HOTEL_COST c)
  (layout
   (cons (layout (string "HOTELS," "Arial" "0x000000" 12) "W")
         (position-scaled 40 85 100 100))
   (cons (layout (string (+ "$" rent ".") "Arial" "0x000000" 12) "W")
         (position-scaled 45 85 100 100))
   (cons (layout (string "plus 4 houses" "Arial" "0x000000" 12) "W")
         (position-scaled 50 85 100 100))
   )
  )
;;Below string is broken in two parts to fit in page,
;;in real example, keep on one line
(define mortage_string_x "If a player owns ALL the Lots of any Color-Group,
 the rent is Doubled on Unimproved Lots in that group.")
(define (FINE_PRINT c)
  (layout
   (cons (layout
          (string mortage_string_x "Arial" "0x000000" 12) "W")
         (position-scaled 40 90 5 5))
   )
  )

(define (ICON c)
  (layout
   (cons (layout (image c))
         (position-scaled 40 25 100 100))
   )
  )

(define (NAME n)
  (layout
   (cons (layout (string n "Arial" "0x000000" 12) "W")
         (position-scaled 10 40 100 100))
   )
  )

(define (RAILROAD_RENTS key value)
  (layout
   (cons (layout (string key "Arial" "0x000000" 12) "W")
         (position-scaled 10 55 100 100))
   (cons (layout (string value "Arial" "0x000000" 12) "W")
         (position-scaled 70 55 100 100))
   )
  )

(define (MORTGAGE_VALUE value)
  (layout
```

```
   (cons (layout (string "Mortgage Value" "Arial" "0x000000" 12) "W")
         (position-scaled 10 85 100 100))
   (cons (layout (string value "Arial" "0x000000" 12) "W")
         (position-scaled 70 85 100 100))
   )
  )

(define (UTILITY_RENT startY str)
  (layout
   (cons (layout (string str "Arial" "0x000000" 12) "W")
         (position-scaled 20 startX 100 100))
   )
  )

(define (B_NAME name)
  (layout
   (cons (layout (string name "Arial" "0x000000" 12) "W")
         (position-scaled 25 20 100 100))
   )
  )

(define (B_MORTGAGED value)
  (layout
   (cons (layout (string (+ "MORTGAGED for $ " value ) "Arial" "0x000000" 12) "W")
         (position-scaled 25 20 100 100))
   )
  )

(define B_FINE_PRINT
  (layout
   (cons (layout (string "Card must be turned this side up if property is mortgaged"
                         "Arial" "0x000000" 12) "W")
         (position-scaled 0 0 100 100))
   )
  )

;; define the high level functions
(define (ROAD
         color
         roadName
         rent
         rent1House
         rent2Houses
         rent3Houses
         rent4Houses
         rentHotel
         mortgageValue
         houseCost
```

```
        hotelCost)
  (card (size 5 5) "ROAD_CARD"
    (layout
      (cons (BLACK_BOARDER) (position-scaled 5 5 85 20))
      (cons (HEADER color roadName) (position-scaled 10 7 80 13))
      (cons (BASE_RENT rent) (position-scaled 40 40 10 10)
      (cons (WITH_HOUSES rent1House 1) (position-scaled 20 (+ 45 (* number 5)) 10 10))
      (cons (WITH_HOUSES rent2Houses 2) (position-scaled 20 (+ 45 (* number 5)) 10 10))
      (cons (WITH_HOUSES rent3Houses 3) (position-scaled 20 (+ 45 (* number 5)) 10 10))
      (cons (WITH_HOUSES rent4Houses 4) (position-scaled 20 (+ 45 (* number 5)) 10 10))
      (cons (WITH_HOTEL rentHotel) (position-scaled 40 70 10 10))
      (cons (MORTGAGE mortgageValue) (position-scaled 40 75 10 10)
      (cons (HOUSE_COST houseCost) (position-scaled 40 80 10 10))
      (cons (HOTEL_COST hotelCost) (position-scaled 40 85 10 10))
      (cons (FINE_PRINT) (position-scaled 5 80 10 10))
      )
      (BACK_OF_CARD roadName mortgageValue)
    )
  )
(define (RAILROAD name iconpath)
  (card (size 5 5) "RAILROAD_CARD"
    (layout
      (cons (ICON iconpath) (position-scaled 40 25 10 10))
      (cons (NAME name) (position-scaled 10 40 25 25))
      (cons (RAILROAD_RENTS "Rent" "$25") (position-scaled 10 55 12 12))
      (cons (RAILROAD_RENTS "If 2 R.R.'s are owned" "50") (position-scaled 10 65 12 12))
      (cons (RAILROAD_RENTS "If 3 ''     ''  ''" "100") (position-scaled 10 75 12 12))
      (cons (RAILROAD_RENTS "If 4 ''     ''  ''" "200") (position-scaled 10 85 12 12))
      (cons (MORTGAGE_VALUE 100) (position-scaled 10 85 12 12))
      )
     )
    (BACK_OF_CARD name 100)
    )
  )

;;Below string is broken in two parts to fit in page,
;;in real example, keep on one line
(define utility_60 "If both ''Utilities'' are owned rent
 is 10 times amount shown on dice.")
(define (UTILITY name iconpath)
  (card (size 5 5) "UTILITY"
    (layout
      (cons (ICON iconpath) (position-scaled 40 25 10 10))
      (cons (NAME name) (position-scaled 10 40 25 25))
      (cons (UTILITY_RENT 40
                "If one ''Utility'' is owned rent is 4 times amount shown on dice.")
                (position-scaled 20 startX 12 12))
      (cons (UTILITY_RENT 60
```

```
               utility_60) )
      (cons (MORTGAGE_VALUE 75 ) (position-scaled 10 85 12 12))
      )
    (BACK_OF_CARD name 75)
    )
  )

(define (BACK_OF_CARD name value)
  (layout
    (cons (B_NAME name) (position-scaled 25 20 14 14))
    (cons (B_MORTGAGED value) (position-scaled 25 20 14 14))
    (cons (B_FINE_PRINT) (position-scaled 5 80 10 10))
    )
  )

;; render the road cards
(render (ROAD "#452A77"
        "MEDITERRANEAN AVE." "2" "10" "30" "90" "160" "250" "30" "50" "50"))
(render (ROAD "#452A77"
        "BALTIC AVE." "4" "20" "60" "180" "320" "450" "30" "50" "50"))

(render (ROAD "#ACC0D9"
        "ORIENTAL AVE." "6" "30" "90" "270" "400" "550" "50" "50" "50"))
(render (ROAD "#ACC0D9"
        "VERMONT AVE." "6" "30" "90" "270" "400" "550" "50" "50" "50"))
(render (ROAD "#ACC0D9"
        "CONNECTICUT AVE." "8" "40" "100" "300" "450" "600" "60" "50" "50"))

(render (ROAD "#D5307C"
        "ST. CHAERLES PLACE" "10" "50" "150" "450" "625" "750" "70" "100" "100"))
(render (ROAD "#D5307C"
        "STATES AVE." "10" "50" "150" "450" "625" "750" "70" "100" "100"))
(render (ROAD "#D5307C"
        "VIRGINIA AVE." "12" "60" "180" "500" "700" "900" "80" "100" "100"))

(render (ROAD "#CF6519"
        "ST. JAMES PLACE" "14" "70" "200" "550" "750" "950" "90" "100" "100"))
(render (ROAD "#CF6519"
        "TENNESSEE AVE." "14" "70" "200" "550" "750" "950" "90" "100" "100"))
(render (ROAD "#CF6519"
        "NEW YORK AVE." "16" "80" "220" "600" "800" "1000" "100" "100" "100"))

(render (ROAD "#DD311D"
        "KENTUCKY AVE." "18" "90" "250" "700" "875" "1050" "110" "150" "150"))
(render (ROAD "#DD311D"
        "INDIANA AVE." "18" "90" "250" "700" "875" "1050" "110" "150" "150"))
(render (ROAD "#DD311D"
        "ILLINOIS AVE." "20" "100" "300" "750" "925" "1100" "120" "150" "150"))
```

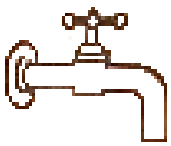
```
(render (ROAD "#FEF037"
        "ATLANTIC AVE." "22" "110" "330" "800" "975" "1150" "130" "150" "150"))
(render (ROAD "#FEF037"
        "VENTNOR AVE." "22" "110" "330" "800" "975" "1150" "130" "150" "150"))
(render (ROAD "#FEF037"
        "MARVIN GARDENS" "24" "120" "360" "850" "1025" "1200" "140" "150" "150"))

(render (ROAD "#39803A"
        "PACIFIC AVE." "26" "130" "390" "900" "1100" "1275" "150" "200" "200"))
(render (ROAD "#39803A"
        "NORTH CAROLINA AVE." "26" "130" "390" "900" "1100" "1275" "150" "200" "200"))
(render (ROAD "#39803A"
        "PENNSYLVANIA" "28" "150" "450" "1000" "1200" "1400" "160" "200" "200"))

(render (ROAD "#253E97"
        "PARK PLACE" "35" "175" "500" "1100" "1300" "1500" "175" "200" "200"))
(render (ROAD "#253E97"
        "BOARDWALK" "50" "200" "600" "1400" "1700" "2000" "200" "200" "200"))

;; render the railroad cards
(render (RAILROAD "READING RAILROAD"      railroad_image))
(render (RAILROAD "PENNSYLVANIA RAILROAD" railroad_image))
(render (RAILROAD "B. & O. RAILROAD"      railroad_image))
(render (RAILROAD "SHORT LINE"            railroad_image))

;;  render utility cards
(render (UTILITY "ELECTRIC COMPANY" electric_company_image))
(render (UTILITY "WATER WORKS"      c_water_works_image))
```

- Script

```
(set-option "monopoly" "./monopoly.script")
```

1. Data files





2. Mock-Ups (Mockups made, created and generated by Nandeck)

TITLE DEED

MEDITERRANEAN AVE.

RENT $2

With 1 house	$ 10 .
With 2 houses	30 .
With 3 houses	90 .
With 4 houses	160 .

With Hotel $ 250 .

Mortgage Value $ 30 .
Houses Cost $ 50 . each
Hotels, $ 50 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

BALTIC AVE.

RENT $4

With 1 house	$ 20 .
With 2 houses	60 .
With 3 houses	180 .
With 4 houses	320 .

With Hotel $ 450 .

Mortgage Value $ 30 .
Houses Cost $ 50 . each
Hotels, $ 50 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

ORIENTAL AVE.

RENT $6

With 1 house	$ 30 .
With 2 houses	90 .
With 3 houses	270 .
With 4 houses	400 .

With Hotel $ 550 .

Mortgage Value $ 50 .
Houses Cost $ 50 . each
Hotels, $ 50 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

VERMONT AVE.

RENT $6

With 1 house	$ 30 .
With 2 houses	90 .
With 3 houses	270 .
With 4 houses	400 .

With Hotel $ 550 .

Mortgage Value $ 50 .
Houses Cost $ 50 . each
Hotels, $ 50 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

CONNECTICUT AVE.

RENT $8

With 1 house	$ 40 .
With 2 houses	100 .
With 3 houses	300 .
With 4 houses	450 .

With Hotel $ 600 .

Mortgage Value $ 60 .
Houses Cost $ 50 . each
Hotels, $ 50 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

ST. CHARLES PLACE

RENT $10

With 1 house	$ 50 .
With 2 houses	150 .
With 3 houses	450 .
With 4 houses	625 .

With Hotel $ 750 .

Mortgage Value $ 70 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

STATES AVE.

RENT $10

With 1 house	$ 50 .
With 2 houses	150 .
With 3 houses	450 .
With 4 houses	625 .

With Hotel $ 750 .

Mortgage Value $ 70 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

VIRGINIA AVE.

RENT $12

With 1 house	$ 60 .
With 2 houses	180 .
With 3 houses	500 .
With 4 houses	700 .

With Hotel $ 900 .

Mortgage Value $ 80 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

ST. JAMES PLACE

RENT $14

With 1 house	$ 70 .
With 2 houses	200 .
With 3 houses	550 .
With 4 houses	750 .

With Hotel $ 950 .

Mortgage Value $ 90 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

TENNESSEE AVE.

RENT $14

With 1 house	$ 70 .
With 2 houses	200 .
With 3 houses	550 .
With 4 houses	750 .

With Hotel $ 950 .

Mortgage Value $ 90 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

NEW YORK AVE.

RENT $16

With 1 house	$ 80 .
With 2 houses	220 .
With 3 houses	600 .
With 4 houses	800 .

With Hotel $1000 .

Mortgage Value $ 100 .
Houses Cost $ 100 . each
Hotels, $ 100 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

KENTUCKY AVE.

RENT $18

With 1 house	$ 90 .
With 2 houses	250 .
With 3 houses	700 .
With 4 houses	875 .

With Hotel $1050 .

Mortgage Value $ 110 .
Houses Cost $ 150 . each
Hotels, $ 150 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

INDIANA AVE.

RENT $18

With 1 house	$ 90 .
With 2 houses	250 .
With 3 houses	700 .
With 4 houses	875 .

With Hotel $1050 .

Mortgage Value $ 110 .
Houses Cost $ 150 . each
Hotels, $ 150 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

ILLINOIS AVE.

RENT $20

With 1 house	$ 100 .
With 2 houses	300 .
With 3 houses	750 .
With 4 houses	925 .

With Hotel $1100 .

Mortgage Value $ 120 .
Houses Cost $ 150 . each
Hotels, $ 150 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

TITLE DEED

ATLANTIC AVE.

RENT $22

With 1 house	$ 110 .
With 2 houses	330 .
With 3 houses	800 .
With 4 houses	975 .

With Hotel $1150 .

Mortgage Value $ 130 .
Houses Cost $ 150 . each
Hotels, $ 150 . plus 4 houses

If a player owns ALL the lots of any Color-Group, the rent is DOUBLED on UNIMPROVED lots in that group.

5 Card Representation System

5.1 Overview

The Card object is constructed by five attributes: two layouts for front and back, a name, a width and a height, and a shape. The layout is of object ComponentLayout, which could be composed of either a list of LayoutTuples or a Decal. A LayoutTuple is a POD with a Layout, PositionScaled, and a Shape. 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 color object.

5.2 ScriptObject Class

- **Purpose:**

It is an abstract class which will be declaring the interface for all the concrete object to use. The subclasses will be overriding its method, `acceptBuilder ()`, in order to create a builder to build objects.

- **Interaction:**

This class does not inherit from any other classes, however it serve as a base class for all other components, which will be inheriting from this.

- **Design Patterns it belongs:**

This class is only part of the visiter pattern explicitly, but lots of design patterns will be used in its subclasses and by overriding its method.

5.3 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.

5.3.1 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 call render on the decal held, and pass in the appropriate information.

- **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.

5.3.2 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. A child of it is a LayoutTuple.

- **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. ArrayComponentLayout Class

   - **Purpose:**

   This is a concrete class that will be able to hold an array of children of types LayoutTuple. The array will be filled with zero or many references to the LayoutTuple objects.

   - **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.

5.3.3 LayoutTuple Class

- **Purpose:**

It is a concrete class which will allow our system to create structure where a specific Component-Layout is associated with a PositionScaled. This is necessary because every layout must be placed in a specific relatively scaled position.

- **Interaction:**

This class does not inherit from any other classes, however, Composite layouts 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.

- **Contains**

It is a POD, holding a Layout, a PositionScaled, and a Shape.

5.4 Decal Class

Decal provides an interface for something graphical. Separate graphical objects from the rest of the system,

- Inherits from ScriptObject, and thus, can accept a visitor.
- Implements this Visiter method, visiter only ever need to know if something is a decal, not what type, to the Decal Type is the only Decal that implements acceptBuilder.

5.4.1 DecalWithFM

- **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, the former of which may be overridden, the latter of which is final.

- **Interaction:**

This class will inherit from the Decal class and other subclasses will be implementing the rest of the functionalities defined by this abstract class.

- **Design Patterns it belongs:**
  - Has a factory method, to retrieve the correct visitor for rendering the held shape.
- **Rendering**
  - To render, a decal makes a visitor, has that visitor visit the shape passed in to render, renders that shape with the image/color/text.
  - This process is the same for 3 decals, so they just have to override the FM method, render can be left alone

1. Text Class

   - **Purpose:**

   A concrete class which conforms to the interface set by the Decal class. It will user an option to put text objects into a card. It 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.

2. ImageDecal

   - **Purpose:**

   A concrete class which offers the users an option 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. Implements the FM.

3. Color Class

   - **Purpose:**

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

   - **Interaction:**

   It will be inheriting from the Decal abstract class

   - **Design Patterns it belongs:**

   Implements the FM.

4. Decal Rendering Visiters Each Visiter knows how to render either a text, image, or color decal for each shape. The visiter is made by said decal via an FM, and then is accepted by the shape to do the rendering. Paramatized with anything the visitor needs for rendering.

5.4.2 DynamicDecal

- **Purpose:**

It is a concrete class which offers the ability to look up a certain decal by name. It has one attribute of type string which will hold the value of the key.

- **Interaction:**

It will conform to the interface set by the Decal class and override some of the methods defined in there. Render will look up the decal in family, and if it exists, then delegate the render call to it.

- **Design Patterns it belongs:**

It can be seen as the RealSubject in the Proxy design pattern.

5.5 Decal Decorator

The Decal decorators implement the decorator pattern for decals. They provide various functionality, such as rounding corners, masks, rotation, or cropping. Override render to call render on each decal(s), and then do some transformaton to the returned image(s).

5.5.1 Pattern

Decorator

5.5.2 Purpose

Provide basic manipulation utilities, things such as masks, and rotation, and cropping can be added, and chained, and the thing using it has no idea it is dealing with a decorator, not a true decal.

5.6 Shapes

5.6.1 Shape

Provides a basic interface for a shape that can be visited. Subclasses will override with the accept method as appropriate, and will contain attributes to represent their shape.

5.6.2 SubClasses

1. Rectangle Class
    - Width
    - Height
2. Circle Class
    - Radius
3. Triangle Class
    - Sides length a,b,c
4. AnyShape Class
    - List<Point>

5.6.3 ShapeVisiter

Interface for something capable of visiting a Shape. Can be implemented for various things like rendering shapes, positioning things inside shapes...

1. Rendering shape visitor : Can visit a shape, and then returns a rendered image of it. A version for image, text, and color
2. Abstract Placer : Can position sub images inside an image, relative to a shape. Used by the CompositeLayout to position it's children inside it's shape.

5.7 Card Class

- **Purpose:**

This is a concrete class which will be implementing the interface create by ScriptObject class. The objects create by this class will have all the necessary components that a card needs. In addition it will implement the `render ()` method in order to render itself, passing it's size, the families it's receives, and it's shape to each layout.

- **Interaction:**

It will be inheriting from the ScriptObject concrete class and it will hold two layouts of type ComponentLayout, size attributes, a name, and a shape.

- **Design Patterns it belongs:**

No design patterns used in it.

5.8 PositionScaled Class

- **Purpose:**

It is a concrete class which will be implementing the interface defined by the ScriptObject class. It will offer the user to scale position where a component will be placed. The user needs will need to specify the change in percentage for the width, height, x and y position.

- **Interaction:**

It will implement the interface set from the ScriptObject class and override some of behaviours defined there.

- **Design Patterns it belongs:**

No design patterns used in it.

5.9 Size Class

- **Purpose:**

It is a concrete class used by the user to set the size for various components that will be used in the card. It has two attributes a width and a height.

- **Interaction:**

It will conform to the interface set by the ScriptObject class.

- **Design Patterns it belongs:**

No design patterns used in it.

5.10 Family Class

- **Purpose:**

It is a concrete class that will offer users the option to associate a certain Decal object with a string. This could be useful because if a certain decal needs to appear in many cards (or layouts) we can refer to it by the name and we don't need to create it multiple times. In this case, the families name will be added to the card name to prevent filename conflicts.

- **Interaction:**

This class will be implementing the interface set by the ScriptObject class.

- **Design Patterns it belongs:**

No design patterns used in it.

5.11 ConsPair Class

- **Purpose:**

It is an abstract class that will enable the user to associate two different object with each other. Bothe these object will be of type ScriptObject. Script object is an abstract class implemented by many other concrete classes.

- **Interaction:**

It will be implementing the interface set by the ScriptObject class and override the functionalities defined in there.

- **Design Patterns it belongs:**

No design patterns used in it.

5.12 List Class

- **Purpose:**

It is a concrete class that will enable the user to create a list of different objects. The elements in this list will be of type ScriptObject.

- **Interaction:**

It will conform to the interface set by the ScriptObject class and override some of the functionalities defined there.

- **Design Patterns it belongs:**

No design patterns used in it.

5.13 ErrorScriptObject Class

- **Purpose:**

It is a concrete class whose purpose is to create an error or display an error message if a certain token is undefined or unfound in the script language.

- **Interaction:**

It will implement and conform to the interface set by the parent class ScriptObject.

- **Design Patterns it belongs:**

No design patterns used in it.

5.14 NullScriptObject Class

- **Purpose:**

It is a concrete class whose purpose is to to denot that there is nothing to be created or nothing is being returned.

- **Interaction:**

It will implement and conform to the interface set by the parent class ScriptObject.

- **Design Patterns it belongs:**

No design patterns used in it.

5.15 UndefinedFunction Class

- **Purpose:**

It is a concrete class whose purpose is to throw an error when a call to an undefined function is made. The functions should be defined in the scripting language and then later on called with respective parameters. When a define builder receives it as a first argument, it will then define it!

- **Interaction:**

It will implement and conform to the interface set by the ScriptObject class.

- **Design Patterns it belongs:**

No design patterns used in it.

5.16 RenderedCard Class

- **Purpose:**

It is a concrete class, whose object will be created as the final result of calling render methods in every component. When everything is rendered, this object will be created.

- **Interaction:**

It will not be inheriting form any other class or being used as subclass for other classes. However, it will be create by the ScrinptEvaluater's concrete classes.

- **Design Patterns it belongs:**

No design patterns used in it.

5.17 Design Patterns used in Card Representation System

The design pattern that we thought would be fit to use in the Card Representation System are **Composite**, **Iterator**, **Proxy**, **Addapter**, **Factory Method**, **Visiter** and **Decorator**.

5.17.1 Factory Method

1. Why we used? Each non decorating, and non dynamic decal has a specific visiter which it must make. The super is the client.
2. What we gained? Customizes the render method with which object should be created, enabling the superclass to implement it. The render method calls this, similar to how a Template method calls methods, but since this is the only step customized in render, I don't consider render a template method.
3. Consequences The decal decorators, and dynamic decals, don't have a visitor to return, and thus, have to return null. This is bad.

5.17.2 Visiter

1. The Shape Visitor used by Composite Layout
   (a) Why we used? A composite layout has some shape, but doesn't know what shape, nor should it!. It also shouldn't know how to position things inside a shape. Instead, the visiter is accepted by the shape, and thus knows what shape it has, and can then properly place items.
   (b) What we gained? The composite layout need not know how to position things inside a shape, but still can.
   (c) Consequences Visiter can get unwieldy, and a bit confusing. If we add a new shape, all visiters need be updated.
2. The Shape Visitor used by Decal
   (a) Why we used? A Decal knows what it has that represents some visual thing, an image, text, or shape, but not how to render it in a circle, rectangle, or other shape, or even what shape it has. That is left to the visiter to determine, and do the rendering.
   (b) What we gained? Decal don't care about what shape they are told to render in, just make their visiter, and have the visiter visit and render in that shape.
   (c) Consequences A visiter for each DecalWithFM implementor. If we add a new shape, all visiters need be updated.

5.17.3 Decorator

1. Why we used? We found that we wanted to do certain actions to decals, that would result in new decals be added. Cropping an image, masking an image, rotating an image, etc. Rather than making these as new decals, which would cause a class explosion, and be limited only to the pairs we made, we decided that it makes sense to crop/rotate/mask any decal, and made these actions decal decorators.
2. What we gained? We can do basic image manipulation by chaining decorator decals, and then using those decorators like normal decals.

3. Conseqences Potentially inefficient.

5.17.4 Composite

1. Why we used? Our team decided to uses the Composite pattern because it offers an easy way to represent hierarchies and also it offers an easy way to treat individual objects and composite objects the same way. Our system allows for a layout component to have layouts or simpler objects such as decal.

2. What we gained?

   - Easy way to represent hierarchies of objects.
   - Easy way to nest composed and simple objects, inside one another.
   - We can treat individual and composed objects the same way.

3. Consequences

   - It makes it easier to represent layouts and decals.
   - It makes it easier to access the elements nested inside one another.
   - It would be easier to add different types of decal, which are not defined right now.
   - The design looks general.

5.17.5 Iterator

1. Why we used? We decide to use this in order to allow an easy and simple way to access all the elements that will be used to represent a card object.

2. What we gained?

   - A way to access the components without exposing its internal representation.
   - We could support different traversal algorithms, but we only need one.
   - It provides a uniform interface even for different structures.

3. Consequences

   - No bad consequences on using the iterator patterns.

5.17.6 Proxy

1. Why we used? Proxy pattern is used very lightly and only just one part of it. It is being used in the DynamicDecal class. The Dynamic Decal is the RealSubject, which will carry (execute) the request forwarded to it. The request would be to receive a decal given a string.

2. What we gained?

   - We have the ability to indirectly reference to the Dycamin Decal class and retrieve something from it.

3. Consequences

   - No consequences as a result of using this pattern.

5.17.7 Adapter

1. Why we used? The adapter pattern is used for our Image class. Our image class will not be able to directly conform to the Image interface provided by the Java libraries. Therefore, we need to create a JavaImageAdapter class to adapt the interface we want with the interface we have.
2. What we gained?
   - Ability to use an Image object, which behaves similar to the one used in Java libraries.
3. Consequences
   - No bad consequences by using this design pattern.

6 Driver System

6.1 Driver Class

- **Purpose:**

The Driver will be the first thing the program will call and it will invoke all the other functions. In essence the Driver is our main.

- **Interaction:**

The Driver will first call the Options class and parse through the config file. Afterwards the Driver will make the scriptEvaluator which will keep track of the script directory and make the appropriate Lexor. The evaluator will then use all the tokens outputted from the Lexor, throw it through the Builder Factory and construct the appropriate objects. In the end the evaluator returns the rendered cards to the Driver, who sends it to the Output class for it to be written and saved.

Will also load any builders requested to be loaded, and place them in the ScriptEvaluators environment.

- **Design Patterns it belongs:**

No design patterns used in it.

6.2 Logger Class

- **Purpose:**

The purpose of the Logger is to keep track of any errors found when parsing through the files and constructing the objects. The Logger will exit out of the program and output the list of errors when checked.

- **Interaction:**

The errors will be added to the Logger when errors are found when contructing objects in the different builders. When the driver checks to see if there are any errors, the Logger will exit the program and output the appropriate errors.

- **Design Patterns it belongs:**

The Logger class used the singleton design pattern. We don't need multiple instances and its helpful because we are able to encapsulate the sole instance of the Logger class and have strict control on adding and checking errors.

6.3 Options Class

- **Purpose:**

Is to keep track of different variables of the deck found in the config file, such as input/output/logger directories or card output types(png, jpg...).

- **Interaction:**

Is created in the driver after parsing the config file. Its also used by the logger and output class both use the options class for knowing where to write to.

- **Design Patterns it belongs:**

No design patterns used in it.

6.4 Output Class

- **Purpose:**

Where all the cards in the deck are written to.

- **Interaction:**

The Driver will create the ouput class after evaluating the script, based on the options class. This is where all the cards will be written to after they are all created.

- **Design Patterns it belongs:**

No design patterns used in it.

7 ScriptEvaluator system

This includes all the Builders, and the Lexer. Whenever I say Builders, I mean subclasses of ScriptObjectBuilder.

7.1 Basic Number manipulation

Basic number manipulation is provided by the NumberOperationBuilder, which is paramaterized with a command. A builder for each command should be mapped with the appropriate symbol in the environment, with the rest of the builders.

7.2 Builder

7.2.1 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, followed by 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. In the ScriptEvaluatorImp

```
ScriptObject doParse(Token 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.

The ScriptEvaluator serves as the Director. The implementor, CardLispScriptEvaluator, could potentially be replaced with a different one, allowing the same builders to be used with a different language.

Most, however, do not actually need it. One could simply give a null ScriptObjectEvaluator to those that do not need it. One could also make a constructor that automatically does this, to avoid the programmer having to worry or possibly just split the ScriptObjectBuilder. One can keep the existing base, but add another subclass, and give that one the ScriptEvaluator. Then, only the Builders that need it would have it. If the dependency on the ScriptEvaluator was removed for some builders, it would help for standalone use, but for parsing a script, we wouldn't know when to make a Script object. We would then need a builder that requires a ScriptEvaluator reference. Transparency would be lost either way, and it doesn't make sense to try to build ScriptObjects outside the script.

1. To Summarize 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 that can be used to add new language features.

7.2.2 What we gained?

- Ability to easily change how a certain object is constructed, by just replacing the builder.
- The same process can be used to construct all ScriptObjectBuilders.
- Can add new products by adding new builders.

7.2.3 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 adds some coupling.
  - Evaluator has a larger interface than it should to allow this coupling with the builders.

7.2.4 Where used

In the ScriptObjectBuilder, and subclasses

7.3 Visitor

7.3.1 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.

7.3.2 What we gained?

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

7.3.3 Consequences

- Must modify the ScriptObjectBuilder class for each new ScriptObject made
  - However, since there 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, this is not a big problem.
- Visitor has a lot of methods, potentially a lot to inherit.

7.3.4 Where used

In the ScriptObjectBuilder, and subclasses.

7.4 Command

7.4.1 Where used?

Number Operation Builder

7.4.2 Why we used?

Basic number manipulation can be done with some opertation being applied to two numbers, resulting in a third. Some operations could be division, adition, etc. To make a whole new builder for each one would be a class explosion, so instead, we paramaterize the builder with the correct number operation.

7.4.3 Consequences

- Only one builder is needed to handle any number operation of the form a <some operation> b = c.
- Extra indirection than a hardcoded method, slightly slower, could potentially be slightly harder to follow, won't know immediately what builder does.
- The Builder is paramatized with some number command to run upon creating a number value.

7.5 State (Builders)

7.5.1 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.

7.5.2 What we gained?

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

7.5.3 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.

7.6 FactoryMethod

7.6.1 Usage A

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.

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.

7.6.2 Usage B

1. Where Used
   In the ShapeDecalBuilder, to get the correct state to go to, after getting a color. Since all shapes start with needing to get a color, but then go to some state specific state, depending on what specific subclass of ShapeBuilder it is, this is used to tie those paralell hierarchies together, and get the correct state.

2. Why?
   Two paralell hierarchies, the ShapeBuilder subclasses, and their initial states. Since these are designed to work together, the factory method works perfectly here.

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

7.7.1 Why we used?

The Scripting language contains two types of Atom literals. These are numbers (doubles), and strings. We want to use Java's 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 the same issue arises. 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.

7.7.2 What we gained?

Double, and String can now be used with their Adapters as if they were any other ScriptObject subtype.

7.7.3 Consequences

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

7.8 Protection Proxy (String/Double wrappers)

7.8.1 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.)

7.8.2 What we gained?

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

7.8.3 Consequences

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

7.9 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 be to make a data interface, which all the Builders would be dependendent 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.

7.10 Composite for ScriptObject and Token

7.10.1 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

7.10.2 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.

- 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!

7.11 Prototype (Builders)

7.11.1 Why we used?

Some of the Builders are paramterized 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. 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 to store the FunctionBody and arguments it is given and stored with, it allows flexibility for future builders. For example, a number operation builder might have one concrete builder class, that takes in the operation to do, +,-,/,*,etc, and then store that builder paramaterized with each operation as a prototype.

7.11.2 Implementation note

Most of the builders 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.

7.11.3 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.

7.11.4 Consequences

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

7.12 Abstract Factory with Prototypes

7.12.1 Why we used?

7.12.2 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 lessen hard-coding Builder types in tokens.
- Provide a central repository of the builder prototype.

7.12.3 Consequences

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

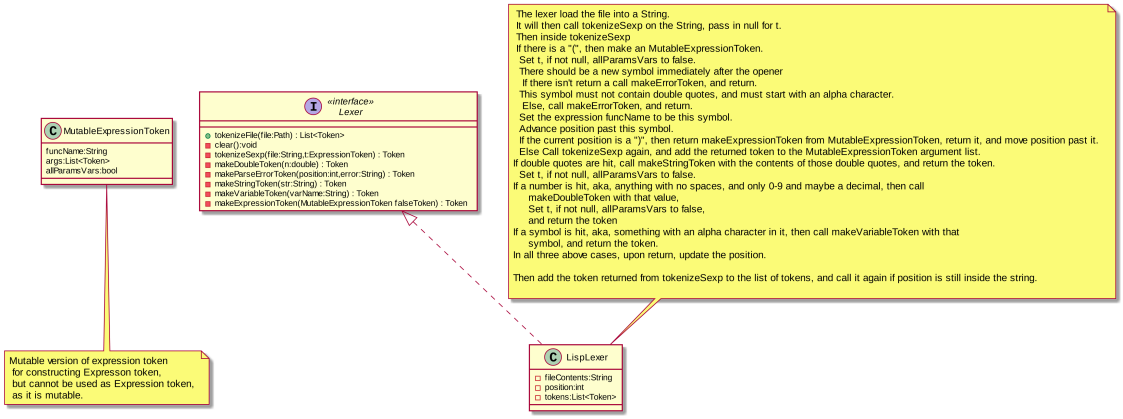
7.13 Lexer

7.13.1 Purpose

To take an input file, and return a list of tokens.

To add support for additional scripting language, provided that the language can be represented with the existing tokens. To do this, one must make a new implementor of the interface, and a corresponding ScriptEvaluator implementor.

7.13.2 UML



7.13.3 Token

- Tokens are immutable after creation.

1. Types
   (a) VariableToken
      - Leaf token, holds a variable name.
      - Returns a VariableBuilderVisitor, with either an UndefinedVariable object if variable is not found, or the variable if it is found in the environment variables.

   (b) AtomToken
      - Returns a ConstantBuilderVisitor 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.
   - Returns 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 composed only of variables, then return UndefinedFunctionBuilder with the given parameter names and function name.
         · If the result of this Builder is an UndefinedFunction script object, and is given to a DefineBuilder as the first argument, that DefineBuilder will then define the UndefinedFunction in the environment variables, so next time the function name is looked up, it will be found in the environment!
       * Else, return an UndefinedFunctionBuilder with an ErrorScriptObject.

2. Purpose
   - To represent a the language in objects, rather than plain text.
   - 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.

7.14 ScriptEvaluator

7.14.1 Environment Subsystem

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

2. EnvrionmentList
   - The Environments 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.

7.14.2 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 subclass Builders need to care about said new ScriptObject. Unlike with the usual Visitor pattern, not all the Builder Visitors need to be updated.

- They encapsulate building a script object.
- See the Builder section earlier for more info.

7.14.3 RenderedCard

A simple "Plain Old Data" class, holds the rendered images of the card along with its name.

7.15 Interactions

The ScriptEvaluator implementor is what will iterate 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 evaluate 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.

7.15.1 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.

7.16 Note on Lexer + ScriptEvaluator interfaces

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