Steven Anderson
Erik Christiansen
Qianwen Ding
Sabrina Gutierrez
Matthew Kroeze
Killian McCoy
Alexander Robau
CIS4930
12/7/2015
Design Patterns SDCG Design Documentation
Uwe

I affirm that the work submitted is my own and that the Honor Code was neither bent nor broken.

Steven Anderson
Erik Christiansen
Qianwen Ding
Sabrina Gutierrez
Matthew Kroeze
Killian McCoy
Alexander Robau
CIS4930
12/7/2015
Design Patterns SDCG Design Documentation
Uwe

Learning Experiences

Steven Anderson

Over the course of the second iteration of this project, I further improved my understanding of the design patterns we learned in this class. Through the discussion of the subteam designs, I learned how to better use the patterns we were using and because of this made a better overall design. It was difficult to find time for everyone to meet up, as large groups have that issue. We all had a general agreement as to what the design should look like and how the patterns should be used, so I would say that is the easiest part of the project. The only thing I would consider changing is to give slightly more time for the super teams to meet, since there was such little time where everyone was free.

Erik Christiansen

The second iteration of this project gave us a chance to consider our original design in light of new ideas presented by the other subteam. This enabled us to achieve a fresh perspective on the task at hand regarding how different design patterns could be used to solve the challenges present in designing such a system. The most challenging part of the assignment was ensuring everyone was on the same page in terms of our understanding of each subteam's design so we could move forward in integrating what we thought were the best elements of each. The easiest part of this iteration was delegating responsibility across the 7 of us. I think we all worked well together.

I think that the learning objective of this assignment was to further aid in our understanding of the design patterns by considering different perspectives regarding how they could be used in the design of a complex system. I think we accomplished this educational objective quite well during our thorough discussions and integration of our designs into a cohesive unit.

Qianwen Ding

During this part of the project, we had the chance to see different designs for solving the same problem and to compare the designs of the two subteams and pick the best parts out of the two. I think the hardest part of this project is to merge the designs of the two subteams, pick the best part to use from both designs, and make sure everyone agrees on all the aspects of the merged design. That really requires us to gain an understanding of both team's design and have some argument and discussion about why one option may be better over the other and what improvement can we make. I think the east part of this project is to create the visual representation and deliverable document once we all have a common understanding and agreement of our final design. I think the educational objective of this project is for us to gain a deeper understanding of the design patterns and their practical uses in designs by letting us see other subteam's design. That way, we can compare and contrast the two designs, learn how we can improve our design. And eventually come up with a better design. I think we achieved the educational objective.

Sabrina Gutierrez

During this next stage of the final project I was able to comprehend where many design patterns should be inserted in order for code management and extensibility and it was interesting to compare and contrast the subgroups' designs. The easiest part of this stage was interacting with the assigned group since we all respected each other and all worked well with each other. The most difficult part of this stage was definitely merging the ideas and going through the other subgroup's design to see the holes that we could fill in order to make a complete design. I believe the educational objectives were to learn where design patterns should be implemented and how to manage ideas and objectively choose which is the best design.

Matthew Kroeze

The easiest part of the project was creating the visuals for our presentation. It was easy to split up the design among our group (compartmentalize the design by subsystem) such that the creation of the visuals was parallelized among the 7 of us. The hardest part of the project was combining both subgroup's systems. Getting everyone in the supergroup to a common understanding with respect to the operation of both systems was extremely time-consuming.

I believe that the learning objective for this assignment was to finalize our understanding of the application of design patterns to a "real-world" system. I believe that we succeeded in achieving this goal by a) correctly applying a sizable number of design patterns and b) avoiding the pitfall of attempting to "cram in" design patterns that are not appropriate for our system.

Killian McCoy

During this part of the term project, I learned how patterns that we considered but ultimately did not fit in our sub-group's design could actually work together in the larger, more modular system of our super-group. Our final UML proved to me that we could realistically use approximately a third of the design patterns documented by the Gang of Four, and justify their added complexities because of the flexibilities they provided for our overall system. Working with a larger group was not too challenging because we all communicated well and were considerate of each other's ideas as well as were not hesitant to correct small mistakes or misunderstandings. The easiest part of the project was creating the supporting documents for the final UML, and the hardest part of the project was combining and adjusting the best features of our sub-team UMLs. I believe the learning objective was to reflect our team's comprehensive understanding of the taught design patterns through our system design, and I believe that we completely accomplished this.

Alexander Robau

The hardest part about the project for me was figuring out how to balance simplicity with features when it came to inventing and defining a scripting language. It is very important to have a scripting language that is simple and easy to learn, but similarly it was important to make the scripting language powerful. I believe that we have done that with the templating function. The easiest part was writing the final documents and slide show. Once an idea is well defined, it is easy to explain and write about it. I believe the learning objective was to utilize design patterns in a design for a large and complex program and do so efficiently and modularly. I believe we have met this standard because our system is extremely modular and makes clever use of a multitude of design patterns

General Goals and Functionality Overview

I. Design Principles

Shuffle was created by adhering to the following design principles:

- Provide a relatively small set of simple but powerful commands to the programmer. The programmer can compose these commands in interesting ways to create complex decks.
- Allow programmers to create and share templates for cards to avoid the duplication of features shared by multiple cards.

II. Cards

A *Shuffle* card has two faces (front and back) which can be independently customized. A card has an orientation (note: this is different than the orientation of visual elements covered in *III. Visual Elements of Cards*) which may be portrait or landscape.

A card is bounded by a polygon which defines the shape of the card and is interpreted with respect to the orientation of the card[1]. Programmers must use polygons defined by the *Shuffle* scripting language when defining the bounds of a card. The *Shuffle* language currently defines the following polygons, and can be easily extended to add more shapes:

- **Ellipse:** An ellipse is defined by its major and minor axis lengths. Portrait cards have the major axis lie along the vertical axis, while it lies along the horizontal axis in landscape cards.
- **Rectangle:** A rectangle is defined by two side lengths. Portrait cards have the longer side lie along the vertical axis, while it lies along the horizontal axis in landscape cards.
- **Triangle:** A triangle is defined by its sides. Two types[2] of triangles are allowed as bounds for a card:
    - **Equilateral:** Defined by its side length. One side lies along the horizontal axis for portrait cards, while it lies along the vertical axis for landscape cards.
    - **Isosceles:** Defined by its unique side length and matching side length. The unique side lies along the horizontal axis for portrait cards, while it lies along the vertical axis for landscape cards.

[1] When reasoning about orientations and bounds, it can be helpful to note that the effect on the bounds of setting an orientation to landscape is equivalent to a 90 degree clockwise rotation of the portrait version.
[2] Why not scalene triangles? They can not be positioned intuitively by using portrait and landscape orientations and thus were excluded to maintain the elegant simplicity of the language.

III. Visual Elements of Cards

The two faces of a *Shuffle* card (front and back) can be independently customized. Card designers use three primitive types of visual elements to create their cards:

- **Text:** A string of text. By default (unless otherwise specified) uses Arial font, size 12, black.
- **Image:** A bitmap loaded from an image file. The file must be located in the directory from which *Shuffle* is run.
- **Shape:** A basic shape. A programmer must use a shape keyword, followed by defining the attributes required by that shape keyword (see script file examples for a sample using ELLIPSE and RECTANGLE). Note: this category includes non-polygonal shapes such as two-dimensional paths.

In addition to the attributes specific to each of these elements, the programmer may specify opacity (by default 100%) and orientation (by default 0 degrees of rotation). The programmer *must* specify the position of each element, which uses a coordinate pair to define the location of the center of the element (for more information about positioning on cards, see subsection V.). Elements are layered on the card in the order that they are added by the designer.

IV. Advanced Visual Elements

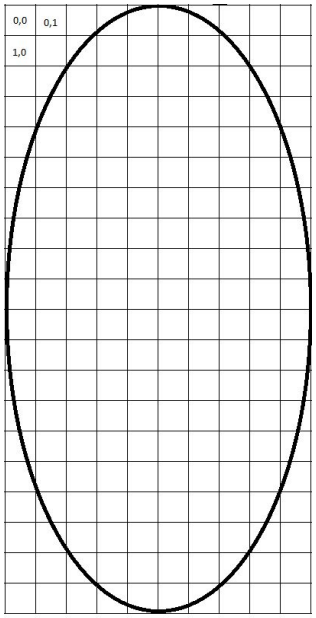
To obtain more sophisticated results with visual elements, designers may use two powerful tools: polygonal containers and borders.

Polygonal containers are used to “crop” their contents. Containers are defined similarly to card face bounds using a polygon supported by the *Shuffle* language. Visual elements are then placed in the polygonal container. When the card is drawn, only the parts of visual elements within the bounds defined by the container will be drawn - the rest will not appear. These containers are particularly useful for defining portraits for images loaded from a file, without cropping the image in an external image editor.

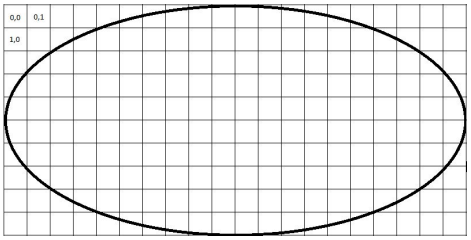
Borders can be applied to any visual elements - even polygonal containers can be given a border. The border appears around the edges of the visual element and the card designer can choose to set its color (by default black) and thickness (by default 5 pixels).

V. Positioning on a Card

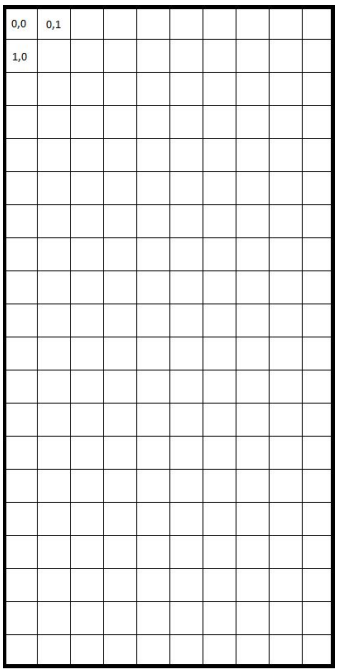
Visual elements and polygonal containers are positioned using a coordinate pair to define the location of their center. The coordinate pair system is best learned through careful study of the following examples:



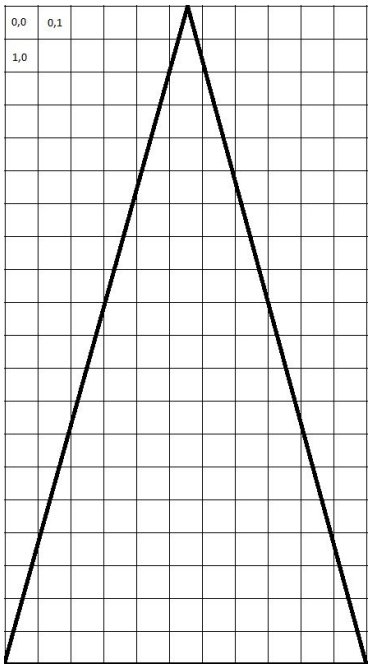
An ellipse-bounded card face in portrait orientation



The same ellipse-bounded card, now in landscape orientation



A rectangle-bounded card in portrait orientation



A scalene triangle-bounded card in portrait orientation

VI. Configuration

Shuffle is configured by modifying the file *shuffle.cfg* in the same folder as the executable. The default contents of the file are as follows:

```
lang=”Shuffle”
output=”png”
compress=
outputdir=
```

An explanation of the options:

- **Language:** sets the scripting language to be used. *ADVANCED USERS ONLY. DO NOT CHANGE THIS IF YOU INTEND TO USE THE PROVIDED SHUFFLE SCRIPTING LANGUAGE OR YOUR SCRIPT WILL NOT WORK.*
- **Output:** sets the image format to be used for the output. Uses .png by default, but can be set to other formats such as .jpg
- **Compress:** sets the compression format. By default, does not compress output. If set, compresses the output to the specified format (ZIP, 7z, etc.) and saves the archive (named “yourDeckName.format”) to the output directory.
- **Output Directory:** The full path of the directory to which output should be saved. By default, the output directory is the directory where the executable is run from.

VII. Script Files

Script files are placed in the same directory as the executable. When the executable is run, all scripts in its directory will be executed.

Just as with the coordinate system, the best way to learn about writing a basic *Shuffle* script file is to examine one:

```
# this is a comment
# all statements are closed by a semicolon
deck "Hello Cards"; # required - name of the deck
description "My very first deck"; # optional - deck description

############################
# Now let's add some cards  #
############################
[4]   card "Hello!" # 4 copies of the card named "Hello!"
      orient LANDSCAPE # optional - default is portrait
      bound RECTANGLE sides=(200,100) color=0x0000;
                                          # required - card bounds
      ##### note that the above was all one statement #####
      FRONT{
            ELLIPSE axis=(25,25) color=0xFFFF pos=(100,50);
            TEXT content="Hello!" pos=(100,50);
      }
      BACK{
            RECTANGLE size=(200,100) orient=90 color=0xFFFF;
      }

[1]   card "Goodbye!" # 1 copy of the card named "Goodbye!"
      bound ELLIPSE axis=(200,100) color=0x0000;
      FRONT{
            ELLIPSE axis=(10,10) color=0xAAAA pos=(20,20);
            TEXT  content="Goodbye" font="Comic Sans" size=100
                  pos=(100,50);
      }
      BACK{
            TEXT content="GOODBYE!" size=200 pos=(100,50);
      }
```

VIII. Templates

Sometimes, we might want to define elements that are common to multiple cards. In that case, we use templates as follows

```
deck "Hello Cards V2";
description "My second deck";

template "Core"
bound RECTANGLE sides=(200,200) color=0xFFFF;
FRONT{
      TEXT content="Property of Shuffle" pos=(0,0) size=8
}
BACK{
      IMAGE file="shuffle_logo.png" pos=(100,100);
}

[4]   card "Hello!"
      uses "Core";
      centerString content="Hello!";
      FRONT{
            ELLIPSE axis=(25,25) color=0xFFFF pos=(100,50);
      }

[1]   card "Goodbye!"
      uses "Core";
      centerString content = "MWAHAHAHA";
      FRONT{
            ELLIPSE axis=(10,10) color=0xAAAA pos=(20,20);
      }
      BACK{
            TEXT content="GOODBYE!" size=200 pos=(100,50);
      }
```

IX. Advanced Scripting

This section's example will demonstrate how to use polygonal containers, chain templates, create bordered elements, override previous element attributes and include external template files[3].

```
template "Core"
bound RECTANGLE sides=(200,200) color=0xFFFF;
FRONT{
      TEXT AS stuff1 content="Property of Shuffle" pos=(0,0)
      size=8;
}
BACK{
      IMAGE file="shuffle_logo.png" pos=(100,100);
}
```

File 1: core.tpl

[3] These external template files must be located in the same directory as the executable and script file.

```
deck “Hello Cards V3”;
description “I’m getting good at this!”;

#####################################################
#     The following statement will be replaced by   #
#     the contents of the core.tpl file before the  #
#     script is executed by the card generator.     #
#     Make sure the any inserted .tpl files are     #
#     in the same directory as the script!     #
#####################################################
utilizes “core.tpl”;

template “Extended Core”
uses “Core”;
BACK{
      BORDER thick=10 color=0xFFFF
             TEXT content=”Don’t steal this!” pos=(30,30);
}

[4]   card “Hello!”
      uses “Core”;
      FRONT{
            ELLIPSE axis=(25,25) color=0xFFFF pos=(100,50);
            TEXT content=”Hello!” pos=(100,50);
      }

[1]   card “Goodbye!”
      uses “Extended Core”;
      FRONT{
            POLYCONTAINER RECTANGLE sides=(50,50){
                  ELLIPSE axis=(10,10) color=0xAAAA pos=(20,20);
                  TEXT  content=”Goodbye” font=”Comic Sans”
                        size=100 pos=(100,50);
            }
      }
      BACK{
            myText content = “I changed!”;
      }
```

File 2: hello_deck_v3.shf

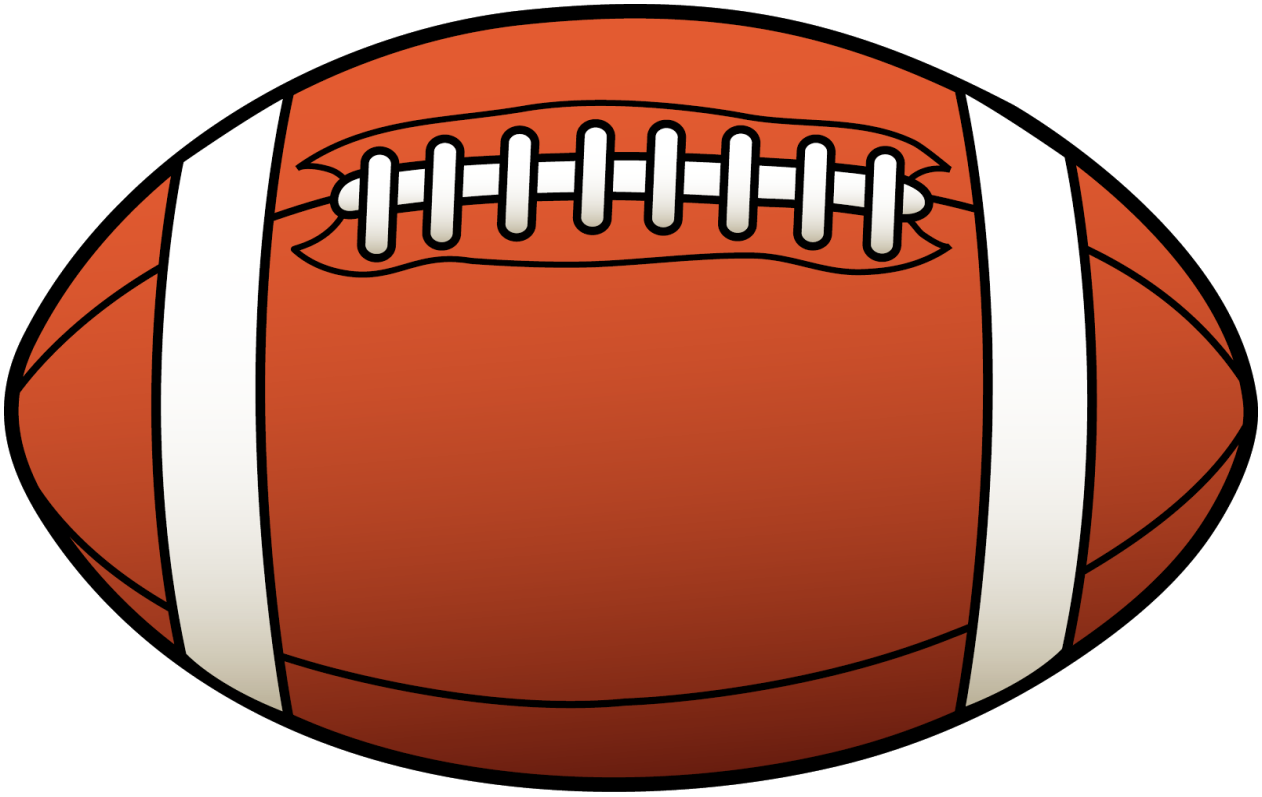
Sample Usage

I. Gator Gameday

A. Config

```
lang=”Shuffle”
output=”png”
compress=”ZIP”
outputdir=
```

B. Data Files



football.png

head_coach.png

mascot.png

C. Script

```
deck “Gator Gameday”;
description “Play with all your favorite Florida Gators!”;

template “Core”
orient PORTRAIT
bound ELLIPSE axis=(4500,2300);
FRONT{
      IMAGE file=”football.png” pos=(2250,1150) orient=90;
}
BACK{
      IMAGE file=”football.png” location=(2250,1150);
      TEXT content=”Gator\n Gameday” size=64 color=0xFF4A00
            font=”Meiryo UI Italic” pos=(2250,1150) type=BOLD;
      TEXT content=”Do not copy without authorization” size=8
                                                pos=(2250, 2100);
}

[1]   card “Head Coach”
      uses “Core”;
      FRONT{
            TEXT content=”Head Coach” font=”Meiryo UI”
                  size=24 pos=(2250,200);
            BORDER thick=20
                  POLYCONTAINER RECTANGLE sides=(500,500)
                                          pos=(2250,1150){
                        IMAGE file=”head_coach.png”
                                    pos=(2250,1150);
                        TEXT content=”Let’s win this!” size=10
                              font=”Meiryo UI” color=0xFFFFFF
                              pos=(2250,1350);
            }
            BORDER thick=10
                  TEXT content=”Control an opponent of your
                        choice\n during their turn.” size=14
                        font=”Calibri”;
      }

[1]   card “Mascot”
      uses “Core”;
      FRONT{
            BORDER thick=10
                  POLYCONTAINER RECTANGLE size=(2000,500)
                           color=0x000000 pos=(2250,1150){
                        IMAGE file=”mascot.png” pos=(2250,1150);
                        TEXT content=”Go Gators!” size=48
                                          pos=(2250, 2000);

            }
```

```
TEXT content="Mascot" font="Meiryo UI" size=24
                                    pos=(2250,200);
TEXT content="A mascot cheers the team to victory"
      font="Meiryo UI" size=16 type=ITALICS
      color=0xFFFFFF pos=(2250,200);
```

D. Output (Mock-Ups)

II. Doritos Crunch Challenge

A. Config

```
lang=”Shuffle”
output=”jpg”
compress=
outputdir=”C:\Users\Dorito_Fan\Cardgame”
```

B. Data Files

dorito.png

logo.png

C. Script

```
deck “Doritos Crunch Challenge”;
description “FOR THE BOLD”;

template “Core”
orient PORTRAIT
bound EQUILATERAL TRIANGLE side=500;
FRONT{
      IMAGE file=”dorito.png” pos=(250,250);
      ELLIPSE axis=(50,50) color=0xFF0000 pos=(240,370);
}
BACK{
      IMAGE file=”dorito.png” pos=(250,250);
      IMAGE file=”logo.png” pos=(250,175);
      TEXT  content=”FOR THE BOLD” pos=(250,250)
                  color=0xFFFFFF size=32
                  font=”Gil Sans Ultra Bold Condensed” opacity=75;
}

[10]  card “Crunch”
      uses “Core”
      FRONT{
            TEXT  content=”CRUNCH” orient=40 color=0xFFFFFF
                  font=”Gil Sans Ultra Bold Condensed” opacity=75
                  size=66 pos=(300,200);
            TEXT  content=”3” color=0xFFFFFF opacity=75
                  font=”Gil Sans Ultra Bold Condensed”
                  size=96 pos=(240,370);
            TEXT  content=”A light snack”
                  color=0xFFFFFF font=”Gil Sans Ultra Bold
                  Condensed” pos=(280,400)
      }

[10]  card “CHOMP”
      uses “CORE”
      FRONT{
            TEXT  content=”CHOMP” orient=40 color=0xFFFFFF
                  font=”Gil Sans Ultra Bold Condensed” opacity=75
                  size=66 pos=(300,200);
            TEXT  content=”7” color=0xFFFFFF opacity=75
                  font=”Gil Sans Ultra Bold Condensed”
                  size=96 pos=(240,370);
            TEXT  content=”Satisfies hunger\nfor 7 hours”
                  color=0xFFFFFF font=”Gil Sans Ultra Bold
                  Condensed” pos=(280,400);
      }
```

D. Output (Mock-Ups)

Front

Back

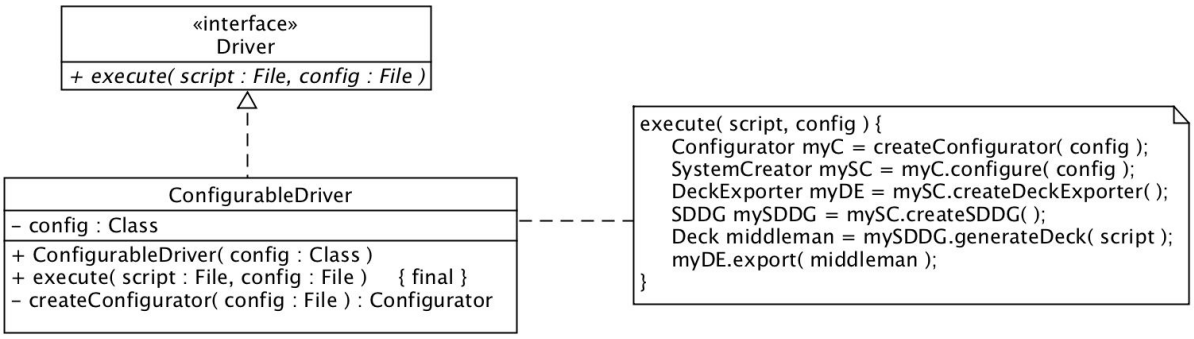
Front

Back

Subsystem 1- Driver

This system is the entry point for the program. It takes in the script file and config file, and creates the necessary objects needed to drive the deck generation and export process. This system ties all other systems together.

UML



Collaborators

This works with the Configurator, Deck Exporter, System Creator, Script Driven Deck Generator, and Deck subsystems.

Design Patterns

None.

Original Design

This subsystem is from the Caverna subteam design and was chosen as it provided more extensibility and a more systematic way to connect all the separate subsystems together to drive the entire SDCG program.

Class: Driver

The Driver class defines an interface with a single method that takes in the script and configuration file needed to generate the deck.

UML

Responsibilities
Define the interface for running the program.

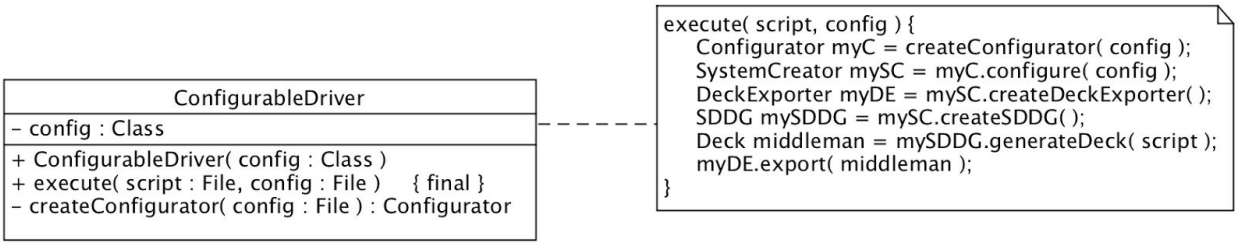
Collaborators
This is implemented by the abstract class ConfigurableDriver class.

Design Patterns
This class did not use any design patterns.

Class: ConfigurableDriver

The ConfigurableDriver class provides a concrete implementation for initiating the processes needed for setting up the configuration according to the config file, generate the deck, and export the deck.

UML



Responsibilities

Initiate the configuration, deck generation, and deck export processes by interacting with other subsystems.

Collaborators

This class implements the interface Driver. It also works with Configurator, SystemCreator, DeckExporter, ScriptDrivenDeckGenerator, and Deck to initiate the necessary processes.

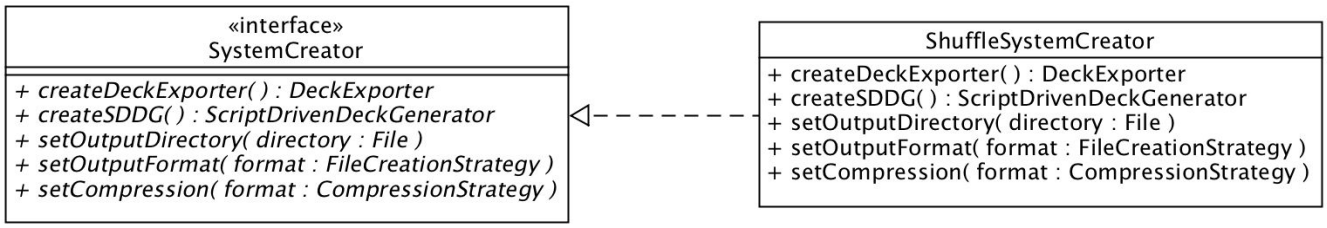
Design Patterns

None.

Subsystem 2 - SystemCreator

This system creates the systems used for creating and exporting the deck with the proper configurations that the user defined in the config file.

UML



Collaborators

This system is configured by the Configurator system. It is also created by the Configurator system. It works with DeckExporter, ScriptDrivenDeckGenerator, FileCreationStrategy, and CompressionStrategy systems. It is used by the Driver system.

Design Patterns

Factory Method - Used to define an interface for creating DeckExporter and ScriptDrivenDeckGenerator, but lets subclasses decide which class to instantiate.

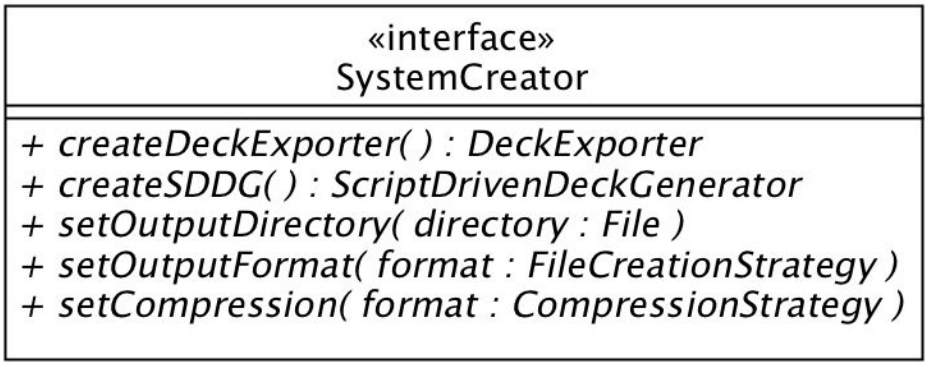
Original Design

This subsystem is from the Caverna subteam design and was chosen as it provided more extensibility and modularity in configuring and driving the program.

Class: SystemCreator

The SystemCreator class defines an interface with methods that return objects of type DeckExporter and ScriptDrivenDeckGenerator. It also contains setter methods to configure itself.

UML



Responsibilities
Defines the interface for creating DeckExporter and ScriptDrivenDeckGenerator

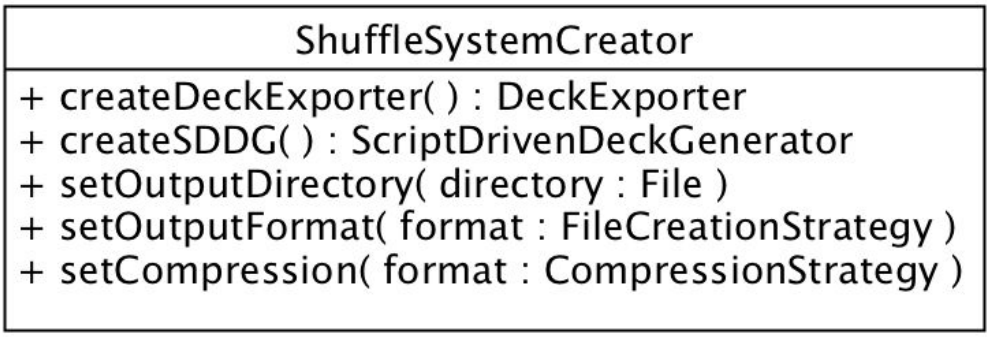
Collaborators
This class is implemented by ShuffleSystemCreator.

Design Patterns
Factory Method - Used to define an interface for creating DeckExporter and ScriptDrivenDeckGenerator, but lets subclasses decide which class to instantiate.

Class: ShuffleSystemCreator

The ShuffleSystemCreator class contains the concrete implementation for methods that return DeckExporter and ScriptDrivenDeckGenerator with the user defined output file directory, output file format, and compression option.

UML



Responsibilities
Returns instances of DeckExporter and ScriptDrivenDeckGenerator with the proper configuration.

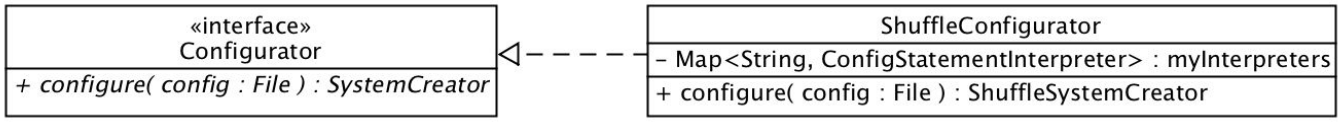
Collaborators
This class is configured by ShuffleConfigurator. Instances of this class can be created by ShuffleConfigurator. This class returns DeckExporter and ScriptDrivenDeckGenerator objects.

Design Patterns
Factory Method - Used to define an interface for creating DeckExporter and ScriptDrivenDeckGenerator, but lets subclasses to decide which class to instantiate.

Subsystem 3 - Configurator

This is the system used by our Driver system to get the SystemCreator which the driver uses to get the deck creation and deck exporter objects. This system's main purpose is to return a properly configured SystemCreator for our Driver system to use.

UML



Collaborators

The Configurator system works with the SystemCreator system to create a properly configured object of type SystemCreator. The Driver system uses this system to interact with SystemCreator. This system also works with the ConfigStatementInterpreter system to interpret the lines in the config file.

Design Patterns

There is no design pattern used for this system.

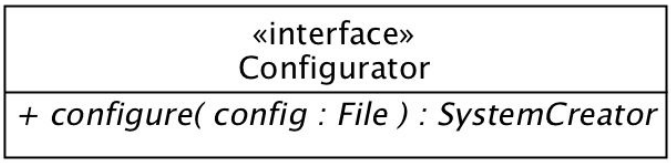
Original Design

This subsystem is from the Caverna subteam design and was chosen as it provided more extensibility and modularity in configuring and driving the program. It can be easily extended to support new scripting language and config file.

Class: Configurator

Provides an abstraction common to all objects that can take a config file to configure the overarching system. Allows our design to be easily extended to support new configurators that use a different configuration file format than our existing one.

UML



Responsibilities
Defines a common type for objects that take a configuration file and use it to configure a card-generation system.

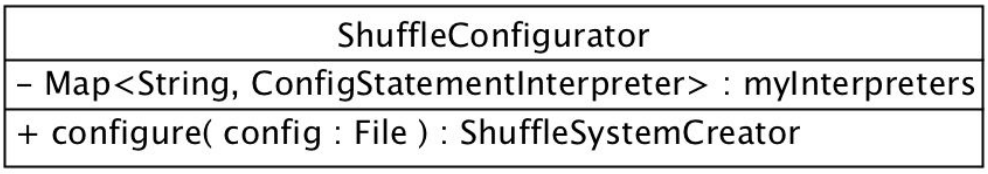
Collaborators
This class is implemented by ShuffleConfigurator.

Design Patterns
None

Class: ShuffleConfigurator

Provides a concrete implementation of the Configurator abstraction. The Map contains the mapping of the keyword in the config file to the corresponding Interpreter who knows how to handle that specific line.

UML



Responsibilities
Provides an implementation of the Configurator type. Returns an instance of SystemCreator configured to return the correct objects.

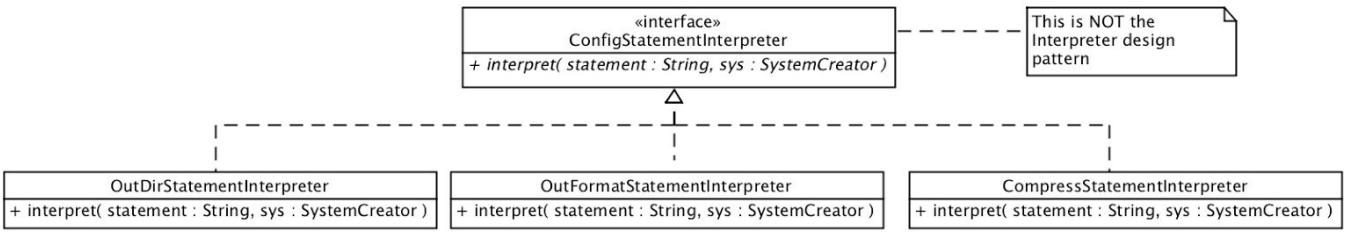
Collaborators
This class creates (and configures) a ShuffleSystemCreator. This class works with ConfigStatementInterpreter to correctly interpret the config file and configures the ShuffleSystemCreator accordingly.

Design Patterns
None

Subsystem 4 - ConfigStatementInterpreter

This system knows how to interpret each line in the config file and configures SystemCreator accordingly.

UML



Collaborators

This system works with the SystemCreator system to configure a SystemCreator object. This system provides the config file interpreting service to the Configurator system that enables the Configurator system to return a properly configured SystemCreator.

Design Patterns

None. (Note: this system does not use the GoF Interpreter design pattern)

Original Design

This subsystem is from the Argricola subteam design because Argricola has a really good and extensible design for interpreting the lines in the script and config file. This Interpreter design and Caverna's Configurator design works well together.

Class: ConfigStatementInterpreter

This class provides an interface that all other concrete Interpreter classes need to implement.

UML

Responsibilities
Define an interface that all concrete Interpreter classes need to implement.

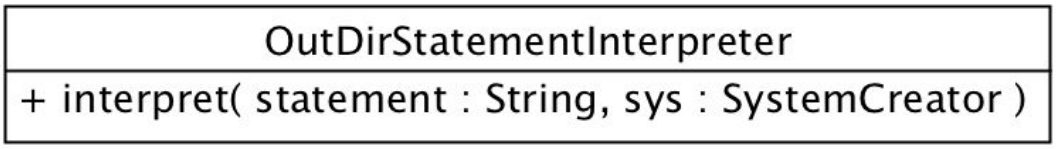
Collaborators
This class works with SystemCreator to configure it. It is also used by the Configurator.

Design Patterns
None

Class: OutDirStatementInterpreter

This class provides a concrete implementation of the ConfigStatementInterpreter abstraction. It takes in the output directory statement from the config file and a SystemCreator object as parameters in order to configure the SystemCreator object with the correct output directory.

UML



Responsibilities
Provides an implementation of the ConfigStatementInterpreter type. Configures the SystemCreator with the correct output directory.

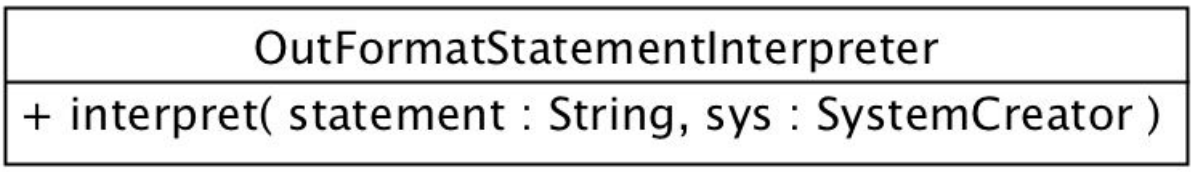
Collaborators
This class works with SystemCreator to properly configure it. It also implements the ConfigStatementInterpreter interface.

Design Patterns
None

Class: OutFormatStatementInterpreter

This class provides a concrete implementation of the ConfigStatementInterpreter abstraction. It takes in the output format statement from the config file and a SystemCreator object as parameters in order to configure the SystemCreator object with the correct output format.

UML



Responsibilities
Provides an implementation of the ConfigStatementInterpreter type. Configures the SystemCreator with the correct output format.

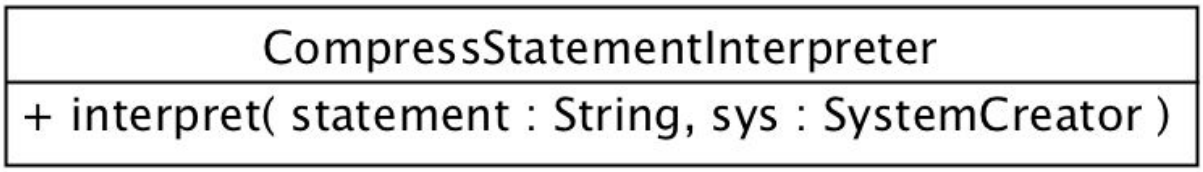
Collaborators
This class works with SystemCreator to properly configure it. It also implements the ConfigStatementInterpreter interface.

Design Patterns
None

Class: CompressStatementInterpreter

This class provides a concrete implementation of the ConfigStatementInterpreter abstraction. It takes in the compression statement from the config file and a SystemCreator object as parameters in order to configure the SystemCreator object with the correct compression option.

UML



Responsibilities
Provides an implementation of the ConfigStatementInterpreter type. Configures the SystemCreator with the correct compression option.

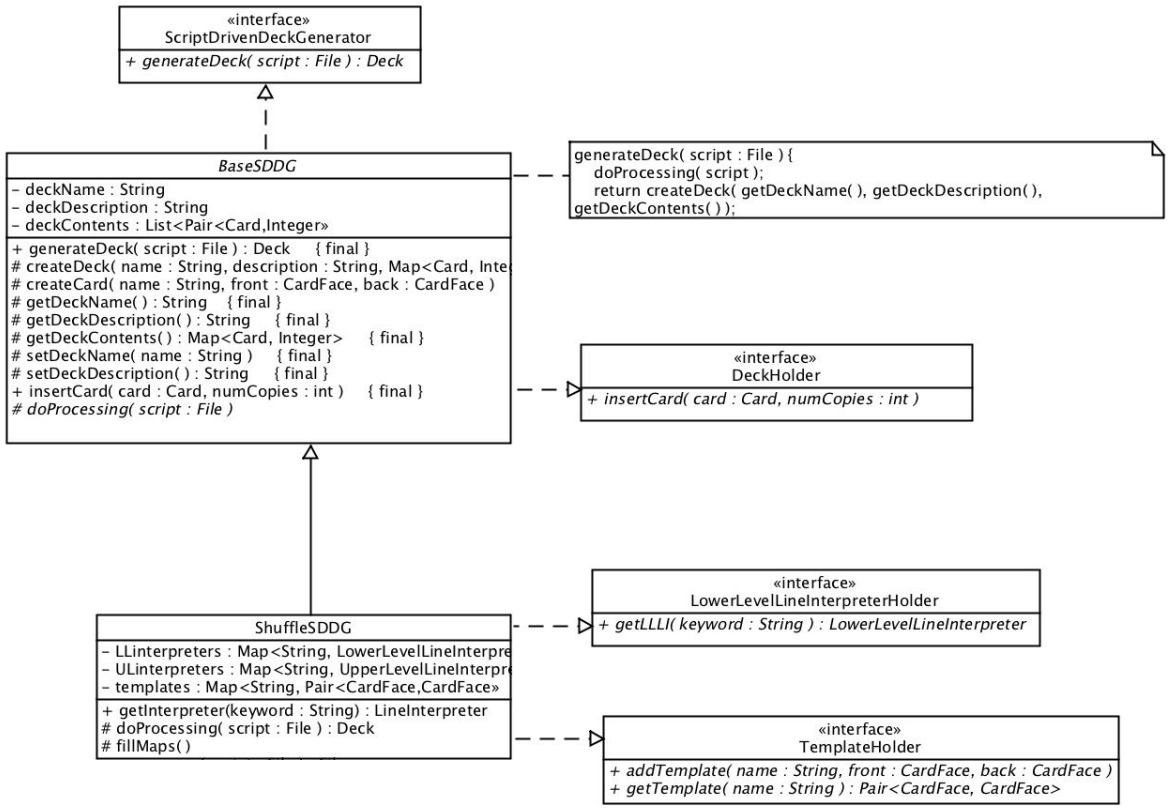
Collaborators
This class works with SystemCreator to properly configure it. It also implements the ConfigStatementInterpreter interface.

Design Patterns
None

Subsystem 5 - ScriptDrivenDeckGenerator

This system uses an input script to generate the deck of cards.

UML



Collaborators

The SDDG system works with VisualElement and Deck systems in its creation of card decks. It also works with the SystemCreator system and an Interpreter hierarchy and mapping system to properly produce cards and apply templating systems.

Design Patterns

Interpreter: An Interpreter hierarchy is used to interpret data that follows keywords and instantiate the proper objects for card construction.
Adapter: A couple simple adapters (LowerLevelLineInterpreterHolder, TemplateHolder, DeckHolder) allow Interpreters to work with and retrieve other interpreters so that nested keywords (templates, borders, etc) can be properly constructed

Class: ScriptDrivenDeckGenerator

Provides an abstraction common to all objects that can take a script file to output an internal Deck representation. Allows our design to be easily extended to support new scripting languages other than *Shuffle* as well as new internal Deck representations other than the *Shuffle* deck.

UML

Responsibilities
Defines a type for objects which can take a script file and output the internal Deck representation.

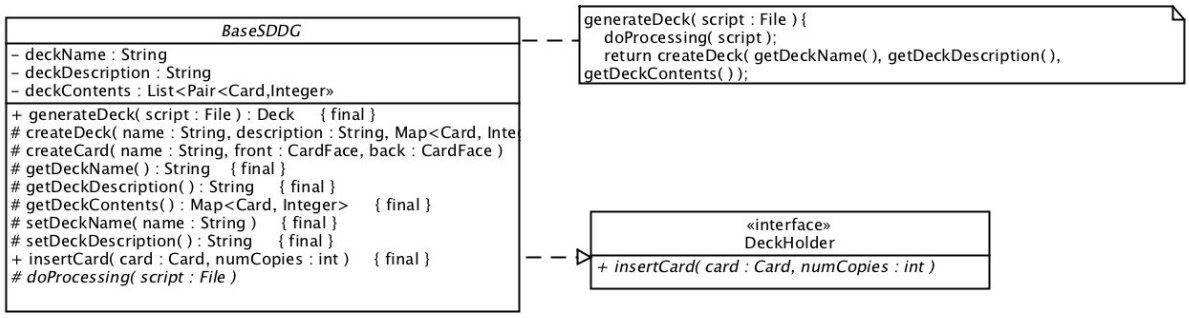
Collaborators
This class returns an implementation of the Deck type. It is also implemented by BaseSDDG.

Design Patterns
None

Class: BaseSDDG

Mechanism for code reuse with respect to commonalities that most SDDGs *should* have (but not that all *must* have, thus the need for the SDDG abstraction) in both procedure and state.

UML



Responsibilities

Factors out attributes common to any “basic” deck.
Implements the ScriptDrivenDeckGenerator interface.
Implements the DeckHolder interface.
Provides services to subclasses for use in creating decks (e.g. createCard( ))
Maintains the invariant that deck generation consists of doing processing, then creating a deck with the current state of the object as arguments.

Collaborators

Implements the ScriptDrivenDeckGenerator interface.
Implements the DeckHolder interface.
Extended by ShuffleSDDG
Creates ShuffleCards (default behavior)
Creates ShuffleDecks (default behavior)

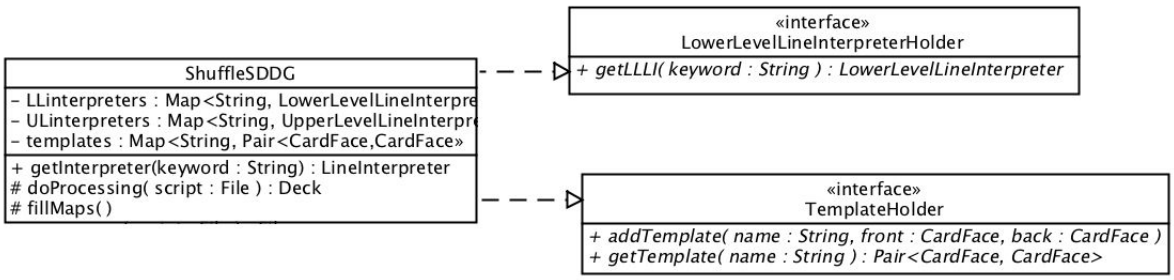
Design Patterns

Factory Method - for creating Cards and Decks. Provides default behavior (see Collaborators).
Template Method - generateDeck() is the Template Method. Template Method pattern is used to define the skeleton of an algorithm for generating deck.

Class: ShuffleSDDG

A concrete extension of the BaseSDDG that can properly interpret *Shuffle* scripts.

UML



Responsibilities

Create decks using Shuffle scripting language script files. This processing involves iterating through the script and getting the correct interpreters for various keywords and then returning utilizing these Interpreters to instantiate the correct structural deck objects. The shuffle language we have invented reads files utilizing an Interpreter hierarchy, thus additionally, there is a fillMaps() function which is called at construction of the ShuffleSDDG that fills the maps with every type of interpreter. These Interpreters are mapped to the keywords which they are designed to interpret the data following. When a keyword is read in, these maps are utilized to return the correct interpreter and then the line data following the keyword is passed into the interpret function. This way, ShuffleSDDG is not strongly coupled to the reading of the language, and there is no “god object” created that is completely responsible for knowing how to read the entirety of the script.

Collaborators

This class extends the BaseSDDG class.

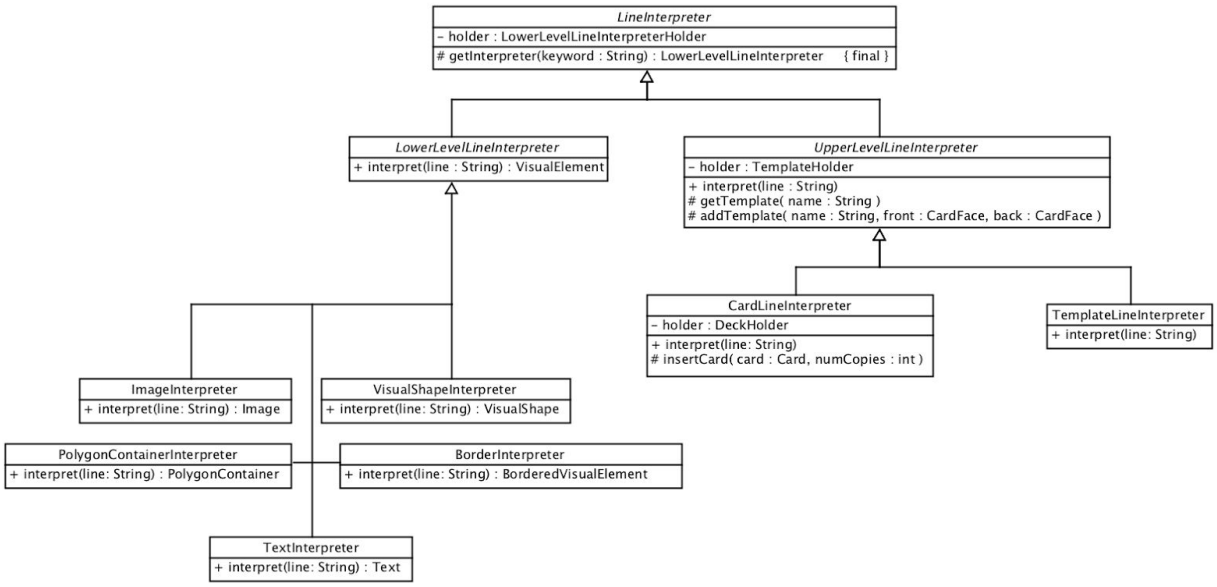
Design Patterns

Template Method - overrides the primitive op doProcessing( ) to do parsing specific to a Shuffle language script file.

Hierarchy: Interpreter

Provides a modular system to instantiate the correct VisualElements based on keywords and data that follows said keywords.

UML



Responsibilities

Allow for the creation of visual elements supported by the Shuffle language.

Collaborators

This hierarchy creates instances of the VisualElement hierarchy. It is also used by ShuffleSDDG to create VisualElements.

Design Patterns

A variation of the Interpreter pattern is utilized here.

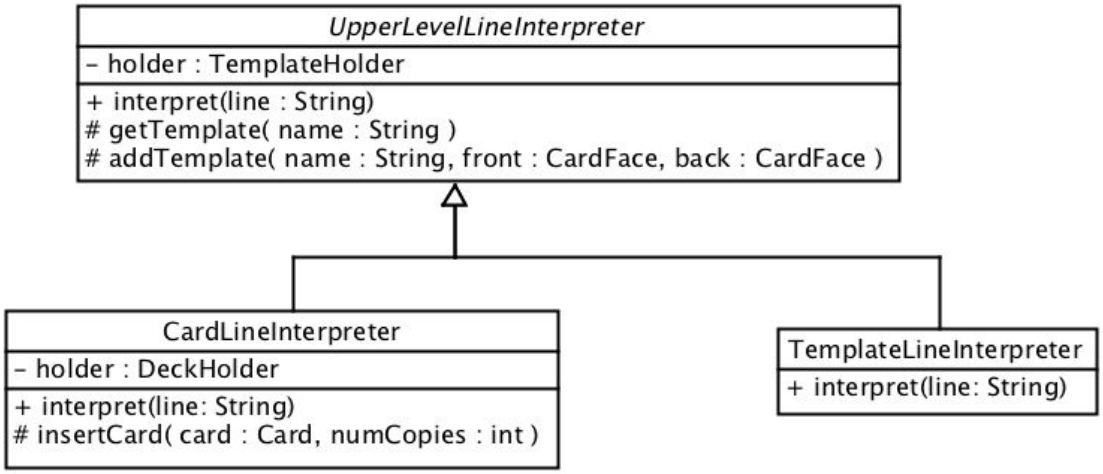
Overview

Lower level interpreters create base objects that do not require the use of templates and are not generally used by the SDDG. Templates are more complex because their implementation is utilized by multiple cards and can have nested templates Thus they share a hierarchy called UpperLevelLineInterpreter which allows for the addition of templates (which are stored as mappings to their given names) to the given card or template when necessary.

Class: UpperLevelInterpreter

Abstract class that allows the creation of cards and templates that have the option of utilizing other templates.

UML



Responsibilities
Allow for the creation of cards and templates for multiple cards to utilize.

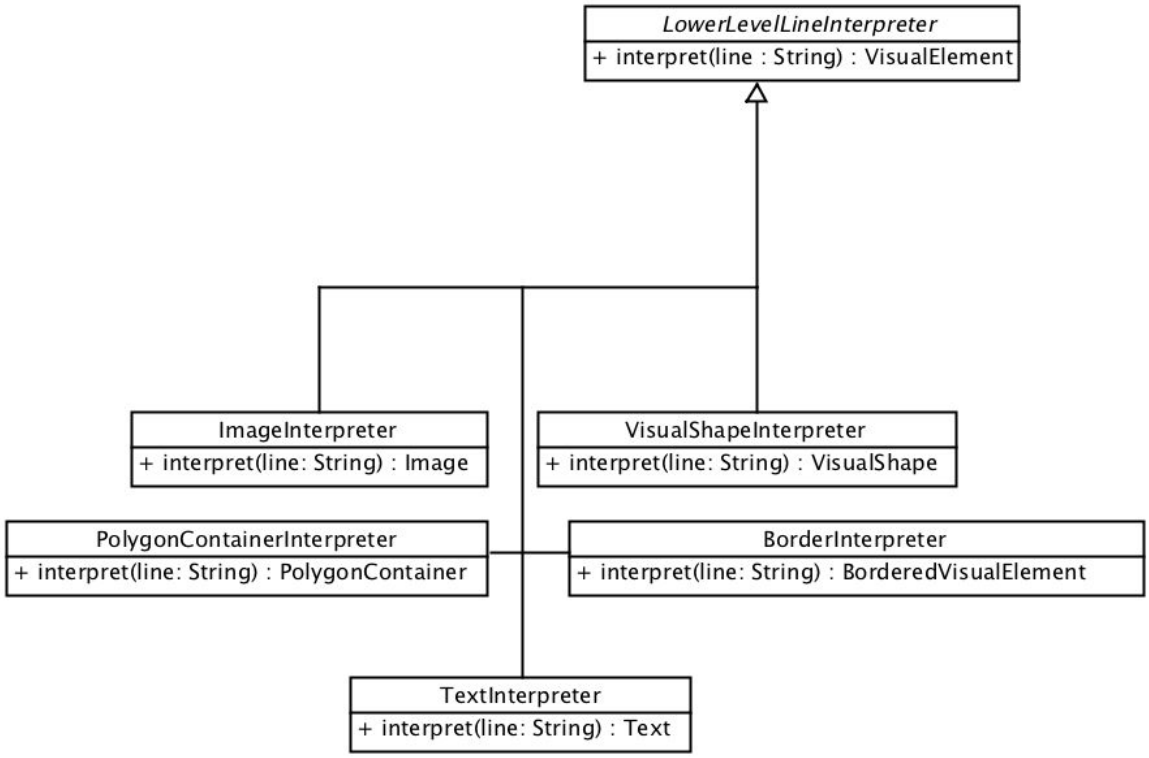
Collaborators
Inherits from LineInterpreter, the abstract class that utilizes an adaptation of SDDG to allow access to LowerLevelLineInterpreters.
Creates CardFace objects that are then used to construct Card objects.

Overview
Unlike LowerLevelLineInterpreter, the ShuffleSDDG works directly with the UpperLevelLineInterpreters, and this is why they are known as “upper level”. The UpperLevelLineInterpreters themselves are responsible for using LowerLevelLineInterpreters (which are retrieved from a mapping of lower level keywords to LowerLevelLineInterpreters known by the abstract class LineInterpreter) to make the objects they need to create cards and Templates. The top level keywords of the Shuffle scripting language are Card and Template. When there keywords are read, the entirety of the script following up until the next top level keyword are passed to an instance of either CardLineInterpreter or TemplateLineInterpreter. The ShuffleSDDG gets the correct class by utilizing a mapping of keywords to UpperLevelLineInterpreters that it holds.

Class LowerLevelLineInterpreter

Abstract class that allows the creation of the more basic VisualElements which do not utilize templates.

UML



Responsibilities
Create VisualElements that can be added to Cards or Templates.

Collaborators
Inherits from LineInterpreter, the abstract class that utilizes an adaptation of SDDG to allow access to LowerLevelLineInterpreters.
Creates VisualElements that can be added to Cards or Templates.
Utilized by UpperLevelLineInterpreters to create objects for Templates and Cards that they construct.

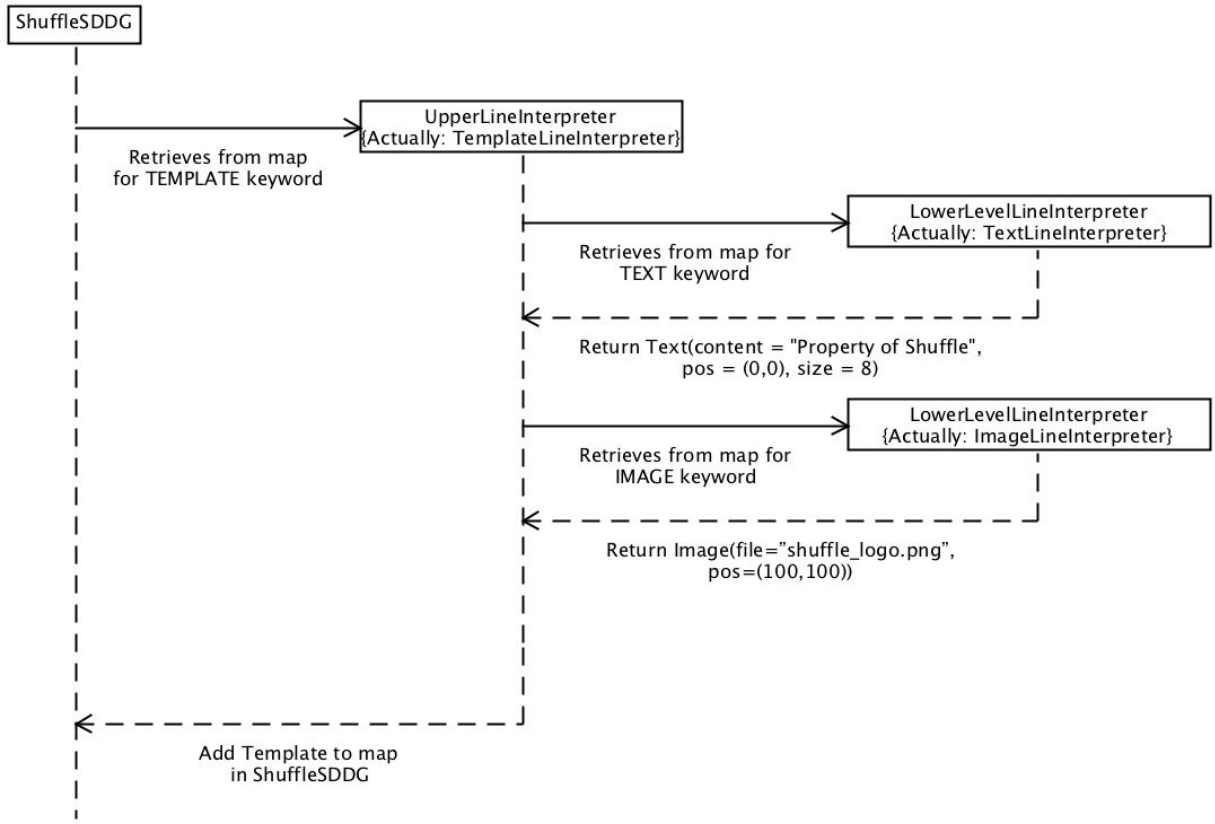
Overview

LowerLevelLineInterpreters form the backbone of the system. In reality, these interpreters create all of the visual objects that actually compose a card. UpperLevelLineInterpreters are simply responsible for grouping them in composites that can help with the construction of cards that have shared characteristics and the Cards themselves. Some LowerLevelLineInterpreters (in particular, the BorderInterpreter and PolygonContainerInterpreter) utilize other LowerLevelInterpreters because these types will have embedded keywords in their scripting. When they encounter a nested keyword, they will utilize the Keyword -> LowerLevelLineInterpreter mapping in the abstract LineInterpreter class to get other LowerLevelLineInterpreters and pass the data following the nested keyword to the correct interpreter.

Interpretation Overview

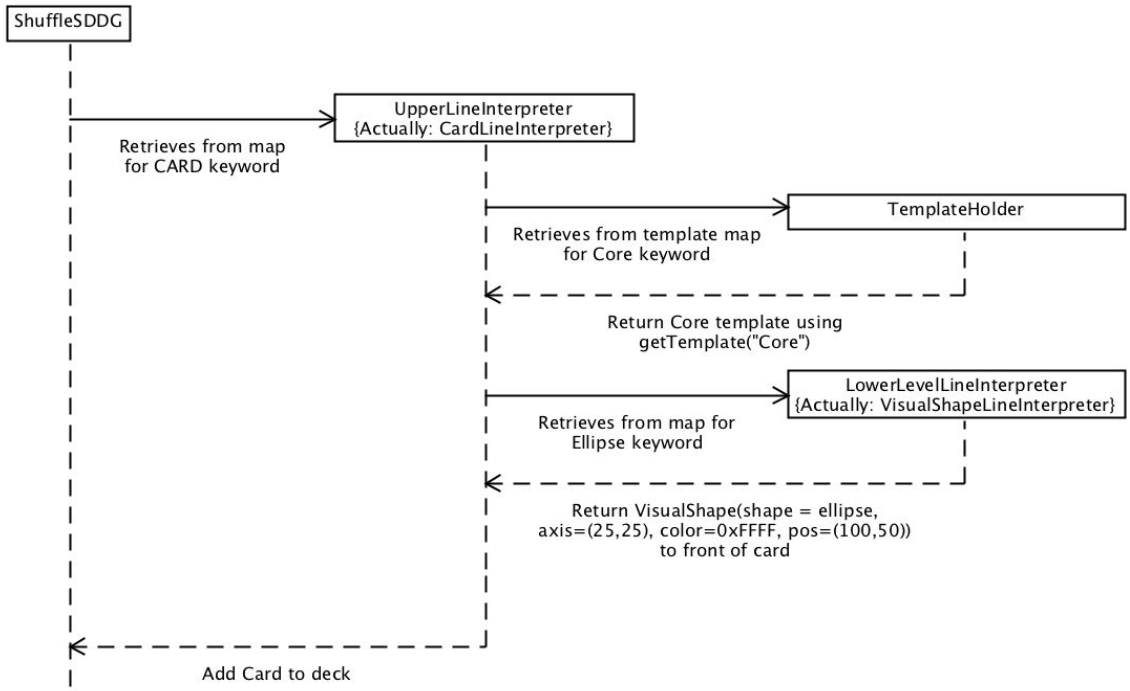
The following is an overview of how a script is interpreted.

```
template "Core"
bound RECTANGLE sides=(200,200) color=0xFFFF;
      FRONT{
            TEXT content="Property of Shuffle"
                 pos=(0,0) size=8
      }
      BACK{
            IMAGE file="shuffle_logo.png"
                  pos=(100,100);
      }
```



Interpretation Overview

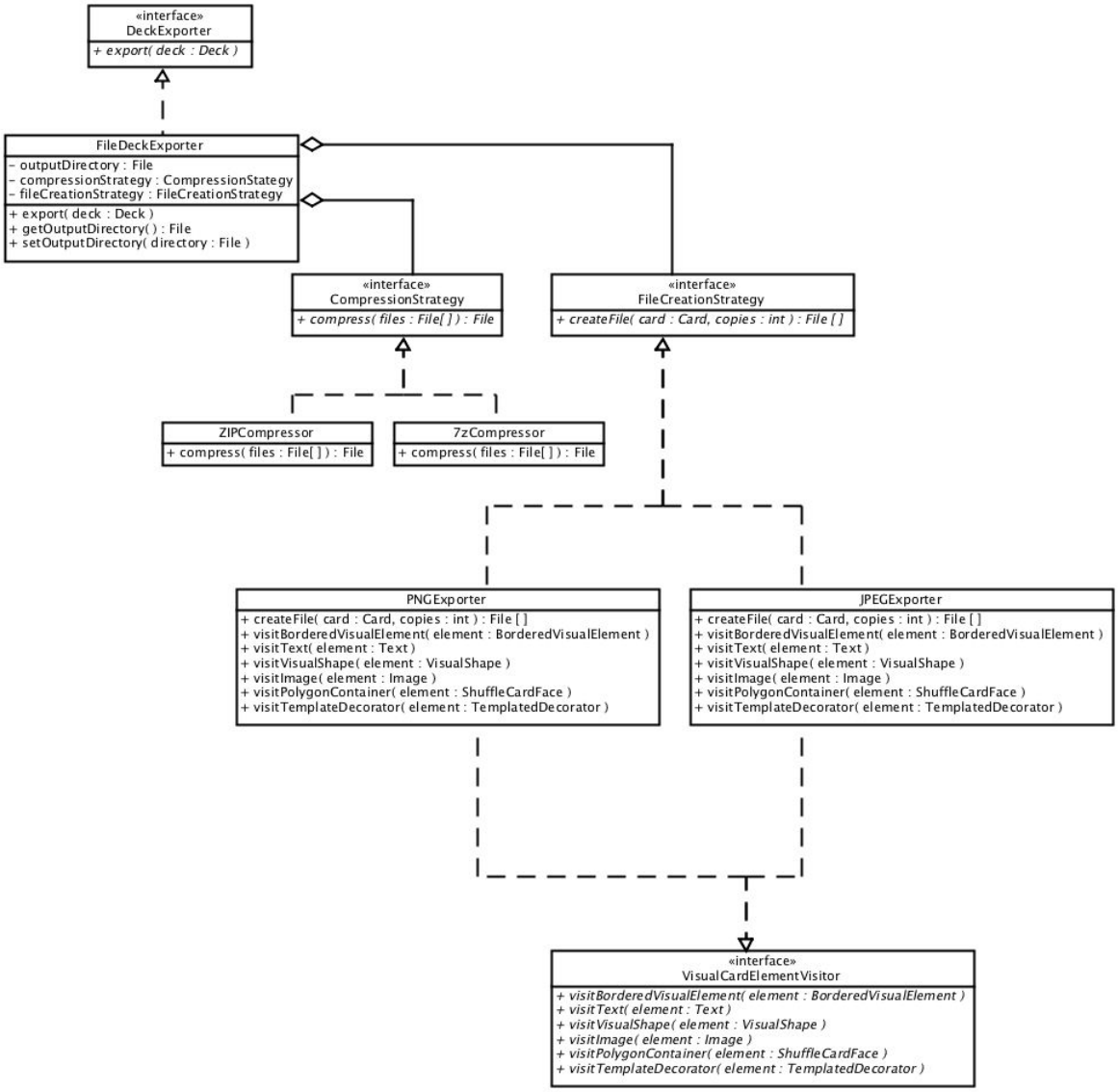
```
[4]   card "Hello!"
      uses "Core";
      FRONT{
            ELLIPSE axis=(25,25) color=0xFFFF
                    pos=(100,50);
            }
```



Subsystem 6 - DeckExporter

This system outputs the Deck to appropriately named, rasterized bitmaps (one per card) with user defined output directory, output file format, and compression option. It does so by drawing out the VisualElements on each Card in the Deck.

UML



Collaborators

Instance of this system can be created by the SystemCreator system. This system also works with the VisualElement and Iterator systems to draw out each Card in the Deck.

Design Patterns

Visitor - Used to draw out the VisualElements on the Card

Strategy - Encapsulates different Compression and Image export algorithms. Used to selects the compression option and the image format of user’s choice.

Class: DeckExporter

The DeckExporter class defines an interface with one method that takes in a Deck and export the Deck to image files with user selected output image file format, output directory, and compression option.

UML

Responsibilities
Define an interface for exporting the Deck.

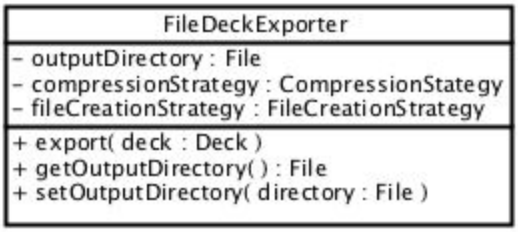
Collaborators
This class is implemented by the FileDeckExporter class.

Design Patterns
There is no design pattern used for this class.

Class: FileDeckExporter

A concrete realization of the DeckExporter abstraction for use with the current system.

UML



Responsibilities

Export a deck using its compression and exporting strategies to the set output directory

Collaborators

This class uses a CompressionStrategy to compress the output and a FileCreationStrategy to determine the file format to output.

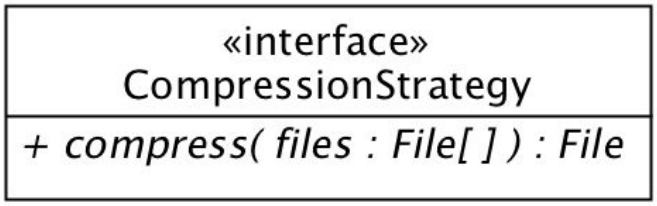
Design Patterns

Strategy - it uses strategy objects for compression and file export.

Class: CompressionStrategy

Provides an abstraction common to all compression strategies used by the system's file exporter.

UML



Responsibilities
Provides a common interface for all compression strategies, which takes an array of Files to compress and return the archive File.

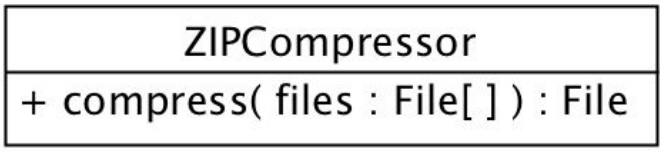
Collaborators
This class is used by FileDeckExporter to compress its output.

Design Patterns
Strategy - Part of a strategy used by FileDeckExporter to compress its output

Class: ZIPCompressor

A realization of the CompressionStrategy abstraction that allows for output to be compressed in a ZIP archive.

UML



Responsibilities
Compress the given files to ZIP archive and return the archive path.

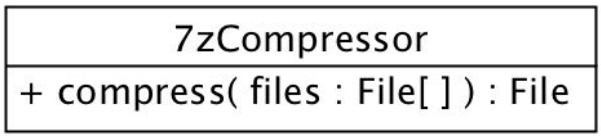
Collaborators
This class can be used by FileDeckExporter to compress its output.

Design Patterns
Strategy - Part of a strategy used by FileDeckExporter to compress its output.

Class: 7zCompressor

A realization of the CompressionStrategy abstraction that allows for output to be compressed in a 7z archive.

UML



Responsibilities
Compress the given files to 7z archive and return the archive path.

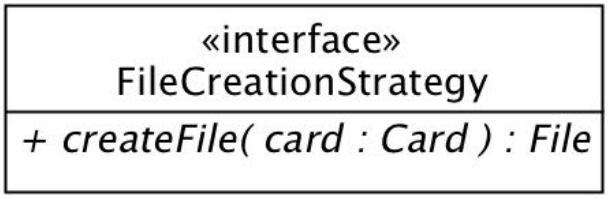
Collaborators
This class can be used by FileDeckExporter to compress its output.

Design Patterns
Strategy - Part of a strategy used by FileDeckExporter to compress its output.

Class: FileCreationStrategy

Provides an abstraction common to all file creation strategies used by the system's file exporter (i.e. objects that take an internal Card representation and export it to some File).

UML



Responsibilities
FileCreationStrategy defines the interface that manages the strategies responsible for creating different file exports.

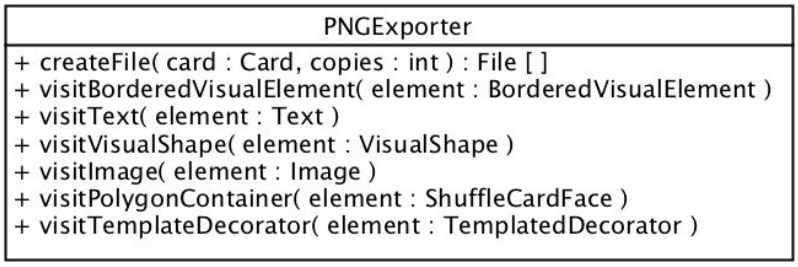
Collaborators
FileCreationStrategy is extended by the specific file creation strategies, such as PNGExporter or JPEGExporter.

Design Patterns
Strategy - FileCreationStrategy utilizes the Strategy pattern in its determination which specific export strategy to use.

Class: PNGExporter

A realization of the FileCreationStrategy abstraction that allows for output to be saved to a PNG format as well as the VisualCardElementVisitor abstraction to VisualElement creation.

UML



Responsibilities

PNGExporter saves the created cards in PNG format.

Collaborators

PNGExporter subclasses FileCreationStrategy and VisualCardElementVisitor.

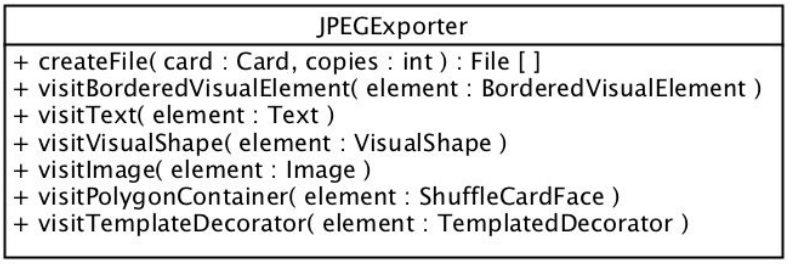
Design Patterns

Strategy - PNGExporter is a specific strategy used to export created cards. It is a subclass of FileCreationStrategy.
Visitor - PNGExporter utilizes the Visitor pattern in its visitation of the cards' visual elements.

Class: JPEGExporter

A realization of the FileCreationStrategy abstraction that allows for output to be saved to a JPEG format as well as the VisualCardElementVisitor abstraction to VisualElement creation.

UML



Responsibilities

JPEGExporter saves created cards in JPEG format.

Collaborators

JPEGExporter subclasses FileCreationStrategy and VisualCardElementVisitor.

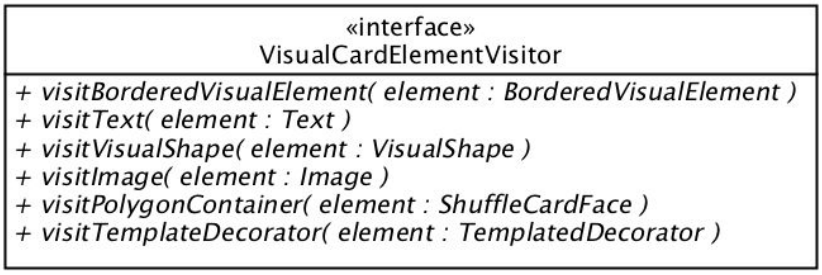
Design Patterns

Strategy - JPEGExporter is a specific strategy used to export created cards. It is a subclass of FileCreationStrategy.
Visitor - JPEGExporter utilizes the Visitor pattern in its visitation of the cards' visual elements.

Class: VisualCardElementVisitor

This class defines an interface with methods that perform drawing operations on each VisualElements.

UML



Responsibilities
Provides an interface for PNGExporter and JPEGExporter to implement.

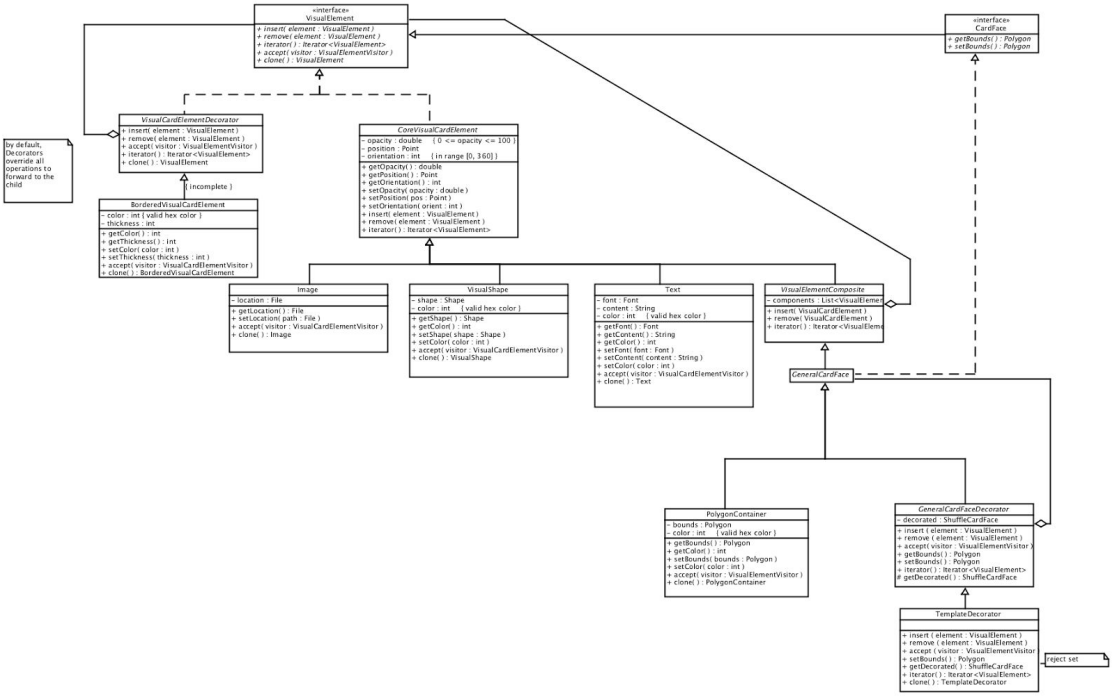
Collaborators
This works with the VisualElement subsystem to draw out the elements such as Text and Image. It is implemented by the JPEGExporter and PNGExporter classes.

Design Patterns
Visitor - VisualCardElementVistor utilizes the Visitor pattern to define what operations to perform on visual elements.

Subsystem 7 - Visual Element

This subsystem defines the individual elements that are placed on the Card Face, including how the elements are traversed, drawn, and composed.

UML



Collaborators

This subsystem works with the Deck Subsystem, as the aggregate of these elements make up the Card Face object, which is composed by the Card object, which in turn is composed by the Deck object. This subsystem also works with the Deck Exporter subsystem by having the elements be drawn with the Visitor pattern.

Design Patterns

Decorator - The VisualCardElementDecorator decorates CoreVisualCardElements
Composite - VisualElementComposites work and the CardFace are aggregates of VisualElements in a part-whole hierarchy.
Iterator - The Iterator is used to traverse the CardFace elements
Visitor - The Visitor pattern is used to draw the individual elements.

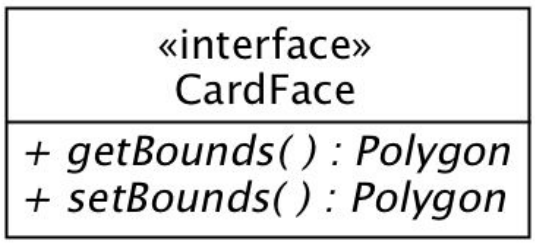
Original Design

This subsystem is from the Caverna subteam design and was chosen as it provided a more robust and extensible model of the Type of objects that exist on a Card. For the core elements that exist on a card, it also provided a simpler interpretation, being a Composite focused structure instead of a Decorator focused structure. This subsystem also added clone functionality so that other scripts would have access to it if they desired.

Class: CardFace

A CardFace is the front and back faces of a Card. These have set bounds and are, in effect, aggregates of VisualElements which will be displayed when the card is drawn.

UML



Responsibilities
To define the interface that a CardFace must implement.

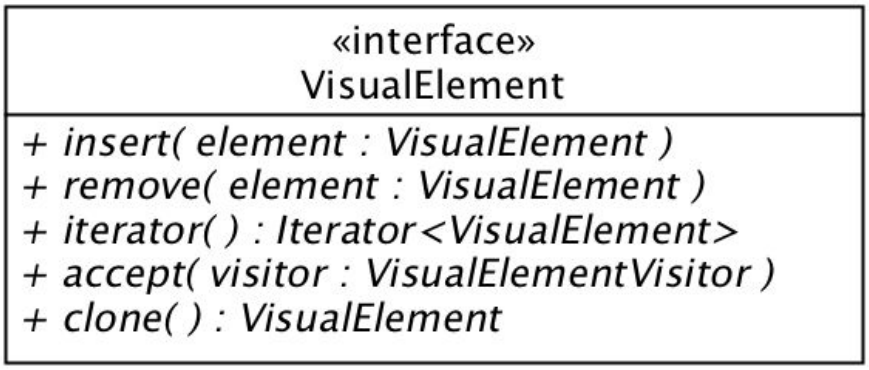
Collaborators
This is composed by ShuffleCard as the front and back of the Card, it is implemented by GeneralCardFace, and it is a sub interface of the VisualElement interface.

Design Patterns
No design patterns are used with this class.

Class: VisualElement

Defines an interface for Visual Elements that will be placed onto a CardFace. These are the elements that, when drawn, will be the visible elements that aggregate into a card's appearance.

UML



Responsibilities
Define an interface that all VisualElements on a card will have.

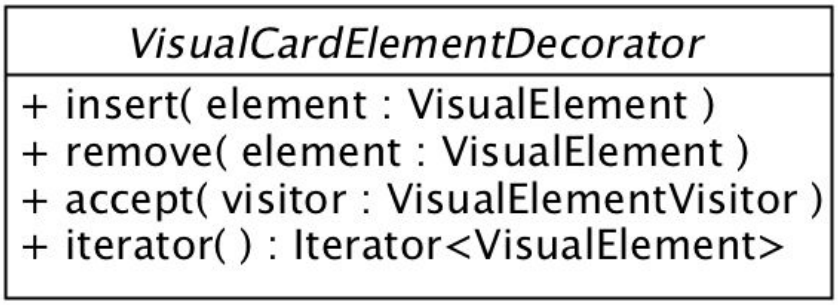
Collaborators
This is implemented by the VisualCardElementDecorator and CoreVisualCardElement, composed by VisualElementComposite and VisualCardElementDecorator, subclassed by CardFace, implements the Iterator interface, and works with the VisualElementVisitor to draw itself.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.

Class: VisualCardElementDecorator

The VisualCardElementDecorator functions as a way to extend functionality in how an element is drawn. It uses the Visitor pattern to draw itself and is part of the implementation of the Decorator pattern.

UML



Responsibilities
Decorate VisualElements in order to add extra aspects to how the elements are displayed.

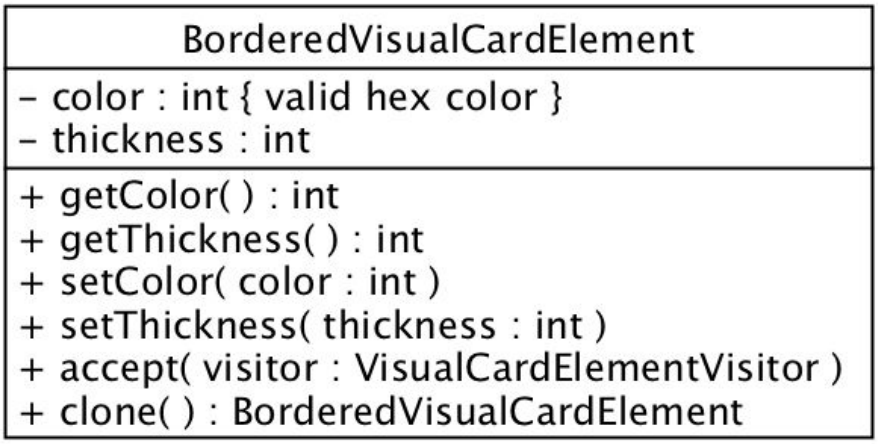
Collaborators
Implements the VisualElement interface and is composed of a VisualElement. It is subclassed by BorderedVisualCardElement. It also interacts with the VisualElementVisitor to draw itself.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator- This is the abstract decorator of the Decorator pattern.

Class: BorderedVisualCardElement

Adds a border to an element when the element is drawn, with a defined thickness and color. It uses the Visitor pattern to draw itself.

UML



Responsibilities
Decorate VisualElements in order to add a border to the element when it is drawn.

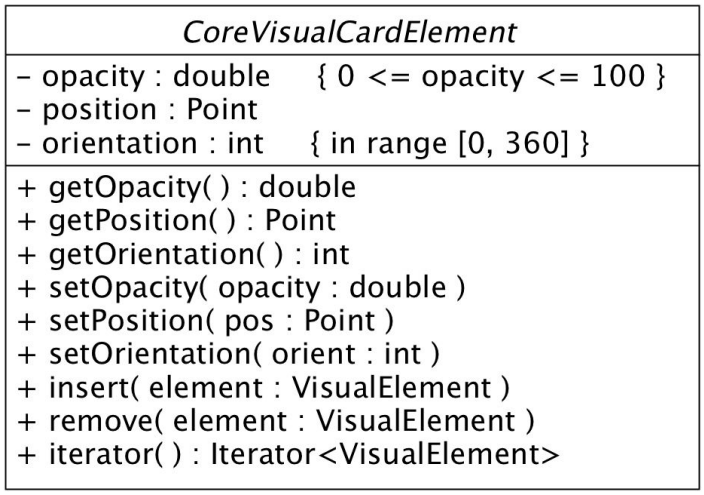
Collaborators
This subclasses the VisualCardElementDecorator class.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator - This is an example of a concrete decorator in the Decorator pattern

Class: CoreVisualCardElement

This is the abstract representation of all CoreVisuaCardlElements that will be displayed on a CardFace. This can be decorated or be part of a composite, but it's main goal is to define the interface that will be implemented by its subclasses.

UML



Responsibilities
Establish the interface that all objects on a Card Face will implement. Such features include how large the element is, its absolute position on the card, its opacity, and how it is oriented.

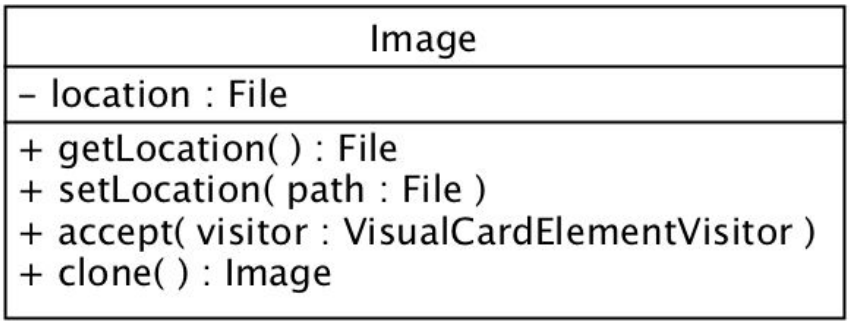
Collaborators
Decorated by VisualCardElementDecorators, subclassed by Image, Text, VisualShape, and VisualElementComposite, accepts a VisualElementVisitor, can return an Iterator, and implements VisualElement.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator - This object can be Decorated by a VisualCardElementDecorator.

Class: Image

One of the concrete representations of the CoreVisualCardElement.

UML



Responsibilities
Holds an image that may be drawn on the card.

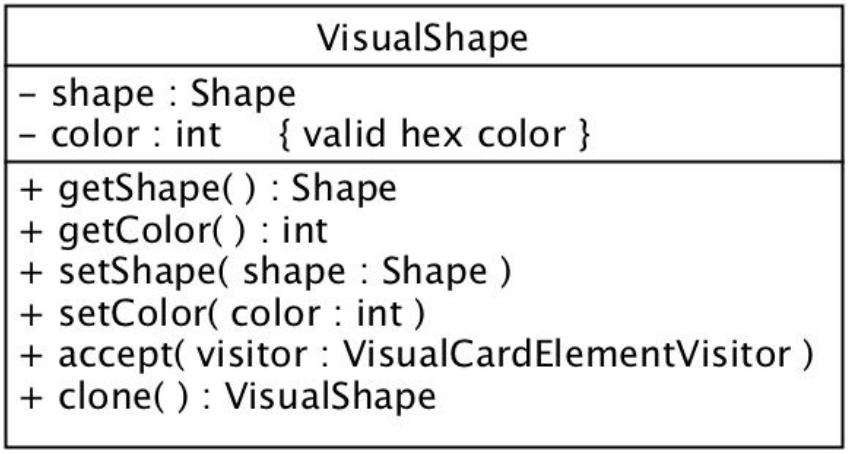
Collaborators
Decorated by VisualCardElementDecorators, subclasses by CoreVisualCardElement, accepts a VisualElementVisitor, and can return an Iterator.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator - This object can be Decorated by a VisualCardElementDecorator.

Class: VisualShape

One of the concrete representations of the CoreVisualCardElement.

UML



Responsibilities
Defines a shape and the color of the shape that may be drawn on the card.

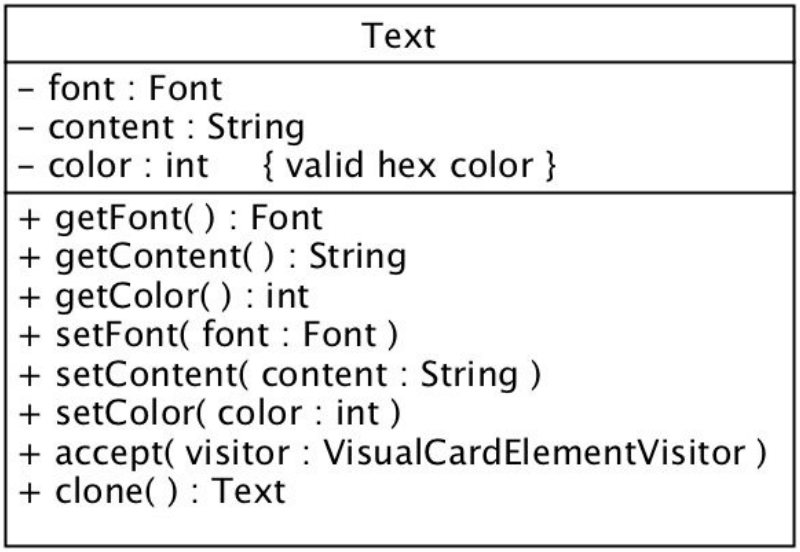
Collaborators
Decorated by VisualCardElementDecorators, subclasses by CoreVisualCardElement, accepts a VisualElementVisitor, and can return an Iterator.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator - This object can be Decorated by a VisualCardElementDecorator.

Class: Text

One of the concrete representations of the CoreVisualCardElement.

UML



Responsibilities
Maintains a String representation of some text that will be displayed on the card, including the text's font and color.

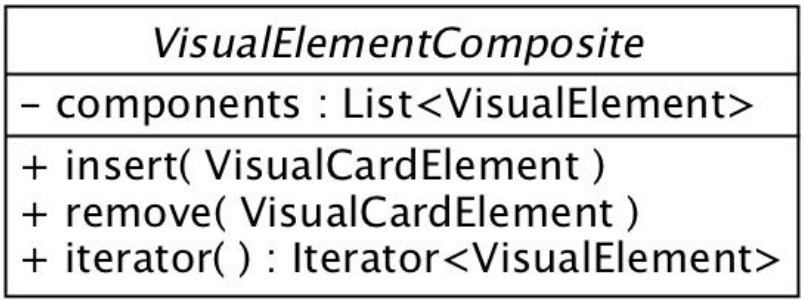
Collaborators
Decorated by VisualCardElementDecorators, subclasses by CoreVisualCardElement, accepts a VisualElementVisitor, and can return an Iterator.

Design Patterns
Visitor - This uses the Visitor Pattern (defined in another subsystem) in order to draw the elements
Iterator - Returns an Iterator which is used when iterating through all of the elements on the CardFace in order to draw all parts of the card.
Decorator - This object can be Decorated by a VisualCardElementDecorator.

Class: VisualElementComposite

The VisualElementComposite class functions as a way to compose VisualElements such that individual VisualElement and composition of VisualElements can be treated uniformly.

UML



Responsibilities
Provide a way to compose VisualElements

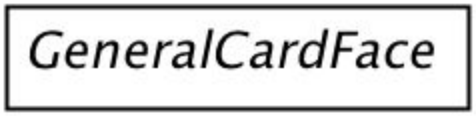
Collaborators
This class is subclassed by ShuffleCardFace and is composed of VisualElement. It also subclasses CoreVisualCardElement, can return Iterator, and can be decorated by VisualCardElementDecorator.

Design Patterns
Composite - Used to compose VisualElement such that individual VisualElement and composition of VisualElements are treated uniformly.
Iterator - Used to iterate through all the VisualElements in the composite.
Decorator - This class can be decorated by VisualCardElementDecorator

Class: GeneralCardFace

The GeneralCardFace class provides an interface for PolygonContainer and ShuffleCardFaceDecorator to implement so that they can be treated as CardFaces while still inherits the attribute and methods of VisualElementComposite.

UML



Responsibilities
Provides an interface that PolygonContainer and ShuffleCardFaceDecorator can implement so that the VisualElementComposite can be treated as GeneralCardFace.

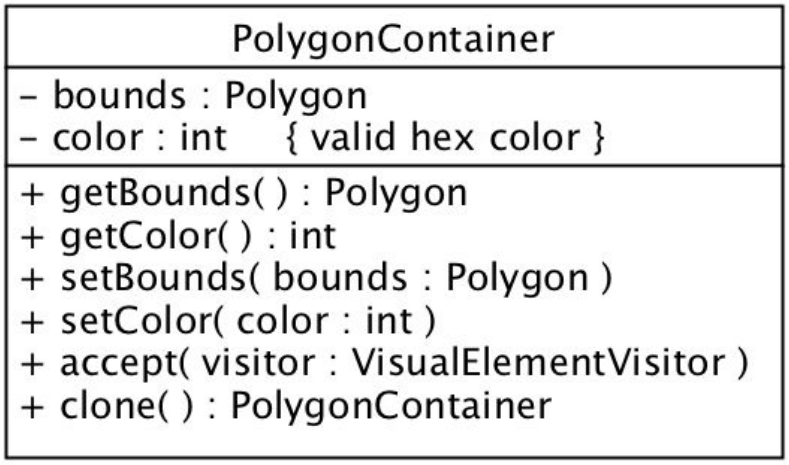
Collaborators
This class extends the CardFace class and subclasses VisualElementComposite class. It is also subclassed by PolygonContainer and ShuffleCardFaceDecorator

Design Patterns
There is no design pattern used for this class.

Class: PolygonContainer

The PolygonContainer class defines the bound for the CardFace. Everything outside the defined bound will not be rendered when drawing the card. It can contain a list of VisualElements within its defined bound and be treated as ShuffleCardFace. When template is used for card, this class serves as the template for the card which will be rendered first, and additional card elements specific to each card will be rendered on top of it. When no template is used for the card, this class will serve as the card that will be rendered.

UML



Responsibilities
Defines the bound for each card. Serves as template for the card when template is used, and serves as the card itself when no template is used.

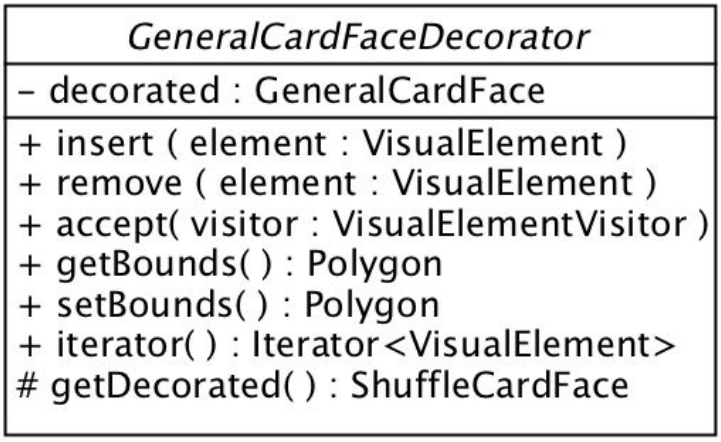
Collaborators
This class subclasses ShuffleCardFace, works with VisualElementVisitor, and can contain VisualElement.

Design Patterns
Visitor - Used to draw the VisualElemements.
Composite - Used to compose VisualElement such that individual VisualElement and composition of VisualElements are treated uniformly.
Decorator - This class can be decorated by ShuffleCardFaceDecorator

Class: GeneralCardFaceDecorator

The GeneralCardFaceDecorator class functions as a way to extend functionality in how a card face is composed. It maintains a reference to a ShuffleCardFace object named *decorated* and defines an interface that conforms to the GeneralCardFace's interface. It forwards request to *decorated* and then add additional responsibilities to that object after forwarding the request.

UML



Responsibilities
Decorates GeneralCardFace in order to add additional responsibilities dynamically.

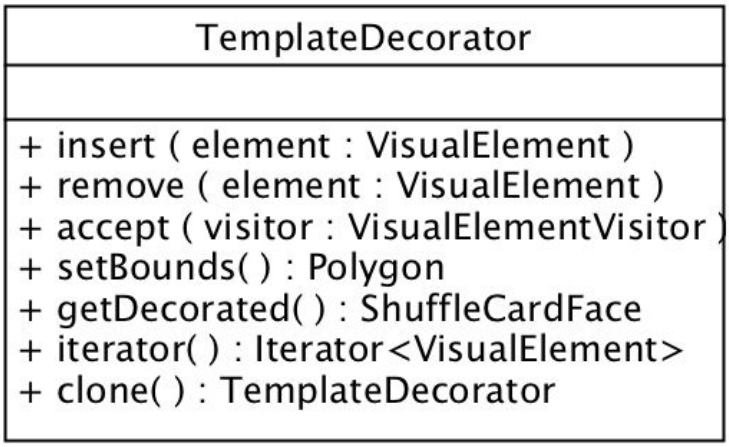
Collaborators
This class implements GeneralCardFace and is composed of ShuffleCardFace. It is subclassed by TemplateDecorator. It also uses VisualElementVisitor to draw itself and Iterator to iterate through its elements.

Design Patterns
Visitor - Used to draw the VisualElemements.
Composite - Used to compose VisualElement such that individual VisualElement and composition of VisualElements are treated uniformly.
Decorator - Used to decorate GeneralCardFace
Iterator - Used to iterate through the VisualElements in the composite

Class: TemplateDecorator

The TemplateDecorator class adds additional responsibility to ShuffleCardFace dynamically by adding card specific VisualElements onto the template when template is used for the card.

UML



Responsibilities
Decorate ShuffleCardFace in order to add card specific VisualElement to the template.

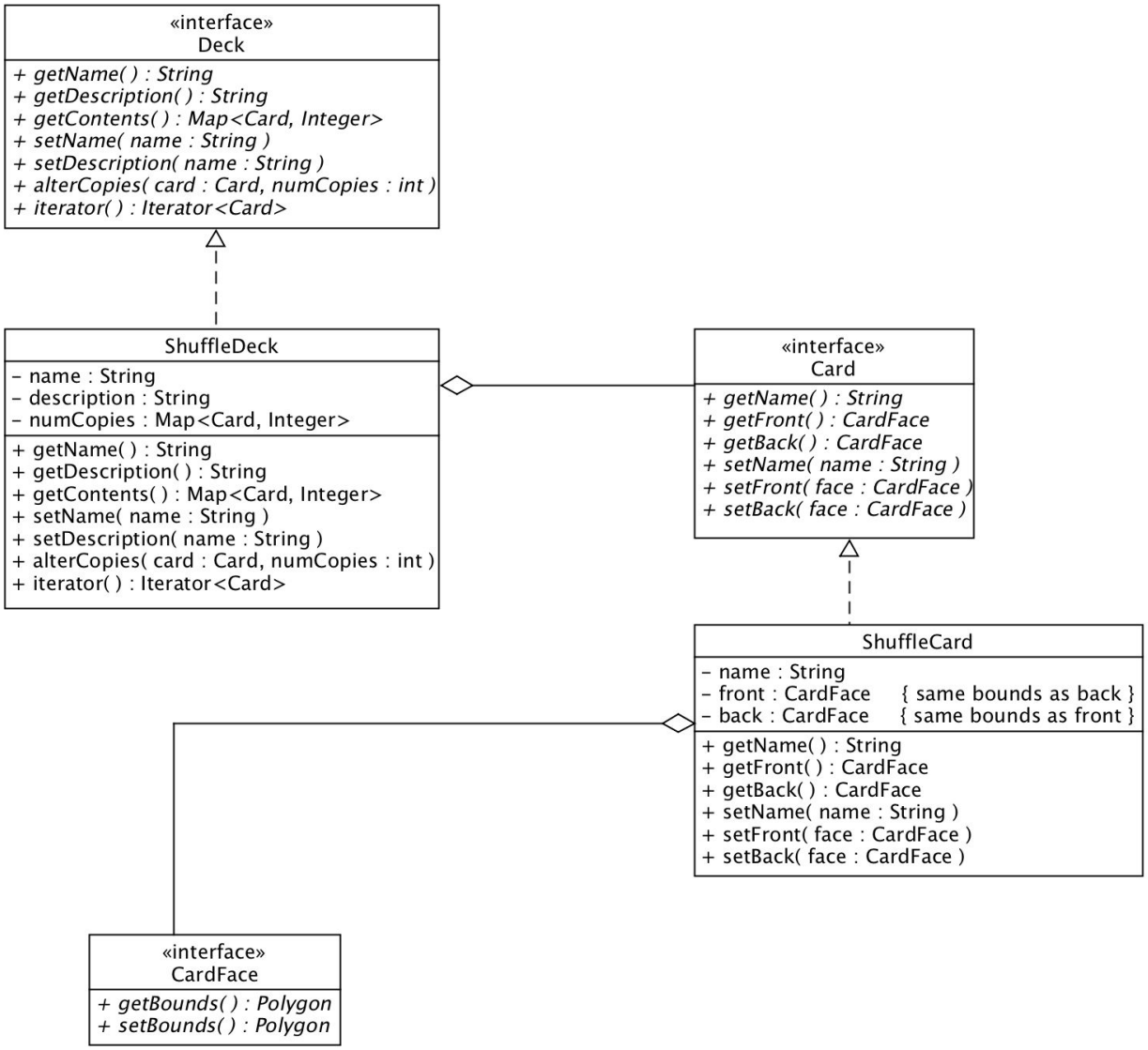
Collaborators
This class subclasses the ShuffleCardFaceDecorator class. It also uses VisualElementVisitor to draw itself and Iterator to iterate through its elements.

Design Patterns
Visitor - Used to draw the VisualElemements.
Composite - Used to compose VisualElement such that individual VisualElement and composition of VisualElements are treated uniformly.
Decorator - Used to decorate ShuffleCardFace
Iterator - Used to iterate through the VisualElements in the composite

Subsystem 8 - Deck

The Deck subsystem manages the set of cards generated by the script.

UML



Collaborators

It works with the Visual Element Subsystem by holding the composite CardFace structures in Cards, the Deck Exporter system by being exported by it, and the SDDG subsystem which generates the Deck.

Design Patterns

It uses the Iterator pattern to iterate through the list of cards.

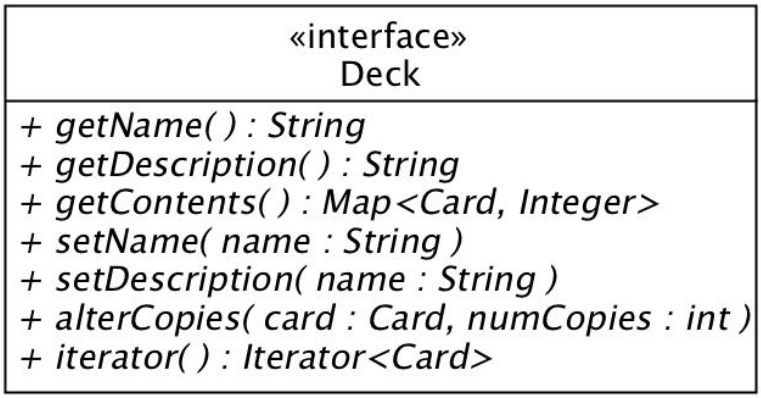
Original Design

This subsystem is from the Caverna subteams design. It was chosen because we felt that this subsystem provided a more robust and reusable interpretation of a “deck of cards”.

Class: Deck

Defines the interface that all Deck types must follow. All Decks must have a name, description, and a map of cards to the number of that card.

UML



Responsibilities
Deck defines the generic interface for creating decks of cards. It must store a map of Cards to the number of that Card, be able to iterate through the list of Cards, and alter the number of copies of a Card.

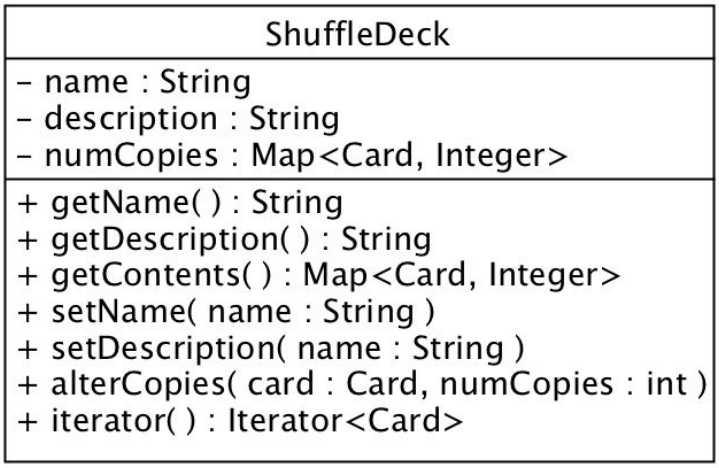
Collaborators
Decks are composed of Cards, are generated by BaseSDDG, and are exported by DeckExporter.

Design Patterns
Iterator - It uses the Iterator pattern to iterate through the cards in the deck.

Class: ShuffleDeck

This class implements the Deck interface and defines one concrete implementation.

UML



Responsibilities
ShuffleDeck defines Deck specifications for the Shuffle system. It creates decks composed of Cards.

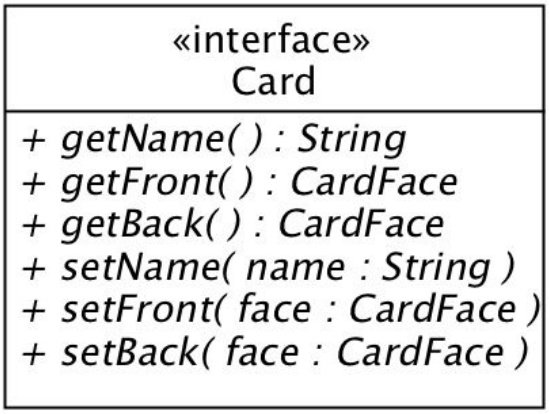
Collaborators
ShuffleDeck is composed of Cards and implements the Deck interface.

Design Patterns
Iterator - It uses the Iterator pattern to iterate through the cards in the deck.

Class: Card

Defines the interface that all Card types must follow. All Cards must have a name, front face, and back face.

UML



Responsibilities
Card defines the generic card interface. It is composed of CardFaces.

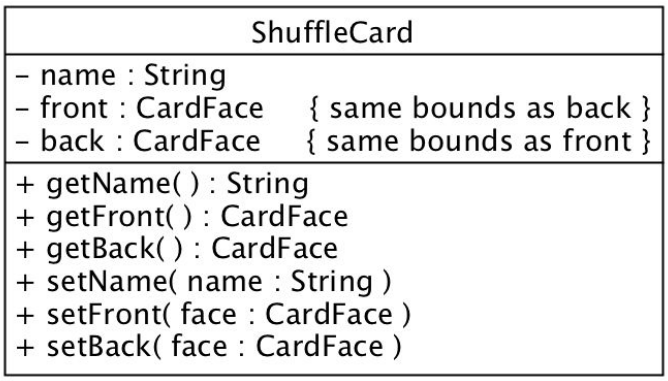
Collaborators
Cards are composed of CardFaces and are composed by Decks.

Design Patterns
No design patterns are used by this class.

Class: ShuffleCard

This class implements the Card interface and defines one concrete implementation of that interface.

UML



Responsibilities

ShuffleCard defines Card specifications for the Shuffle system. It creates individual cards as compositions of card faces.

Collaborators

ShuffleCard implements the Card interface and is composed of two CardFaces.

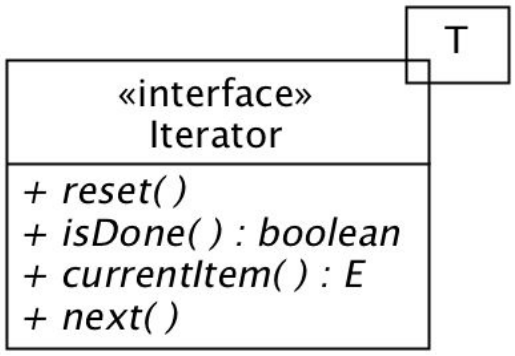
Design Patterns

No design patterns are used by this class.

Class: Iterator

Used to iterate through any objects that have lists of elements. To preserve the clarity of the document, it was not directly attached to any other UML subsystems. If a class has in Iterator in it, it implements this interface.

UML



Responsibilities
The iterator interface is responsible for defining the means by which the elements of deck structure are traversed.

Collaborators
Iterator works with the Deck and VisualElement subsystems to iterate through cards and visual elements on cards.

Design Patterns
Iterator - It uses the Iterator pattern to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.