VPL Creation Framework

By Dillon McGovern, David Wang, Michael Gordon, Henry Smith, and Dana Graham

# Table of Contents

[**Table of Contents**](#_3yz495hetfb) **2**

[**Introduction**](#_w25t04jivle9) **3**

[**Research**](#_3l03jvewpvi8) **4**

[**Use-Case Diagrams**](#_kv32ybab3vmy) **6**

[End-User (Someone using a developed VPL):](#_nj1wxq8nap0s) 6

[Developer:](#_uhniqcz9qhou) 7

[**System Design**](#_ldgtq5mg0dkc) **8**

[General UML](#_1fz1ck194pit) 8

[The MVC Subsystem](#_3u4wxh7e2ygt) 10

[View](#_ldrnsy2gqhh3) 11

[Sequence Diagram](#_3afox0n1qcxe) 11

[Model](#_e5acqji4shb5) 12

[Controller](#_xznhl81i4coa) 13

[Executor](#_a2047javw7fj) 15

[**Patterns**](#_xtonbyf1gmt) **17**

[Composite Pattern](#_s4ahsxvclo46) 17

[Model-View-Controller Pattern](#_csgonkj38e4e) 17

[Iterator Pattern](#_h246tr36y6tt) 18

[Non-Pattern](#_8w8b2splte16) 19

[**Developer Example**](#_tx3ex6h5qv13) **20**

[Creating ConcreteVisualComponent](#_vl4g1inbtfxa) 20

[Creating ConcreteDataController](#_djlp5itzdmza) 20

[Creating ConcreteOutput](#_blzi40zezstn) 21

[Implementation in Client](#_j7mzs13ybg2d) 21

[**Appendix**](#_pdg0ka3kb8vd) **22**

# Introduction

Visual programming languages are a subset of programming languages where programs are created by manipulating visual elements rather than by textually writing code. Instead of lines of statements, conditionals, and code blocks, VPLs use the arrangement and content within textual and graphical components as the basis of the syntax of the VPL-specific “intermediate” program language. From this “intermediate” language, the graphic component, usually implemented in the form of blocks, are linked together and compiled into a “proper” base language before being executed. Essentially, the only difference between a VPL and a textual programming language is that VPLs abstract raw code statements into modular units of code. This abstraction could be single-line statements such as assignment and basic arithmetic or could be expanded to include multi-line blocks, resulting in templates for more complex structures such as if/else statements, loops, and even function calls.

The goal of this project is to lay the foundations of a framework from which visual programming languages can be created. All VPLs have the same fundamentals: how to create code units, how to connect these units to create a program, and how to convey the output of the program. It is by changing these basic processes unique VPLs are created, and hence the most important aspect of our framework’s design. The end goal of our framework is to allow an end-user developer to create something akin to the Scratch, an educational VPL developed by the MIT Media Lab, albeit to create all kinds of VPLs for all sorts of applications and domains.

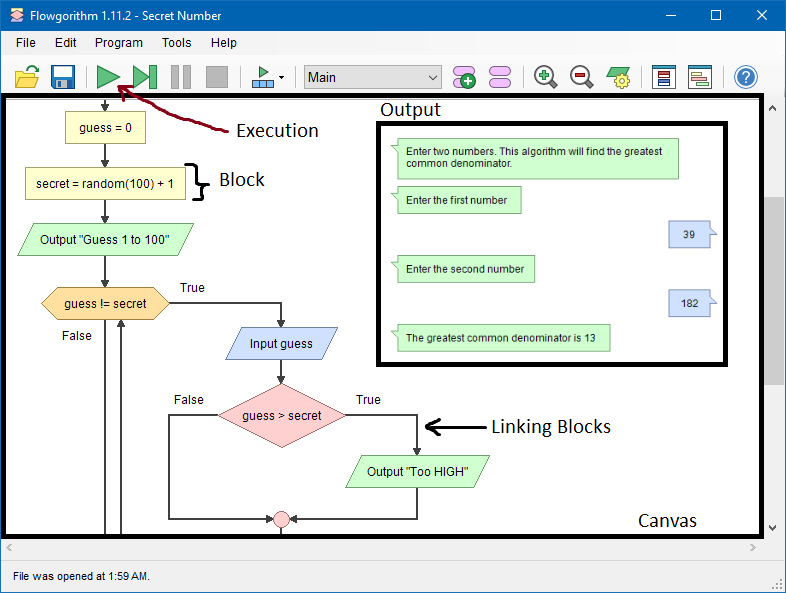
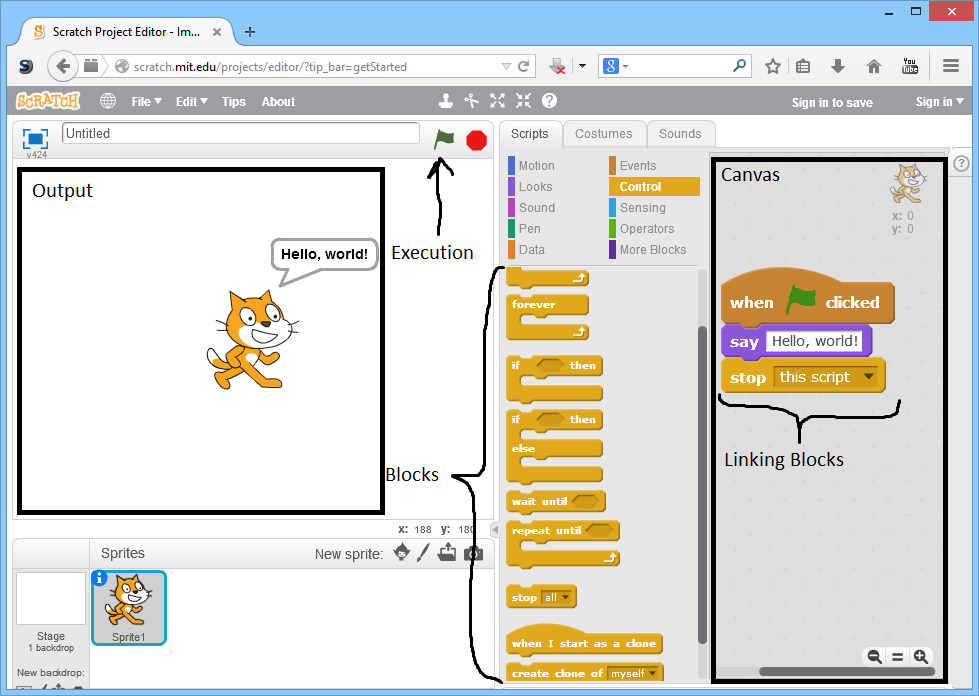
However, since we are developing a framework from which VPLs are built, and not an implementation of a VPL, a degree of abstracted complexity must be maintained in order for the end-user/developer to have enough flexibility. Therefore, the real challenge of creating our framework lies in providing enough predefined functionality to make developing VPLs simple and straightforward, but not so concrete we take away from the developer’s ability and flexibility to create a powerful and unique VPL.

# Research

VPL development frameworks should make it easier for developers to create new visual programming languages. We initially began looking for similar functionality amongst existing VPL’s in order to identify components that we would be able to include in our VPL framework. We identified the following are features that are common to all VPL’s:

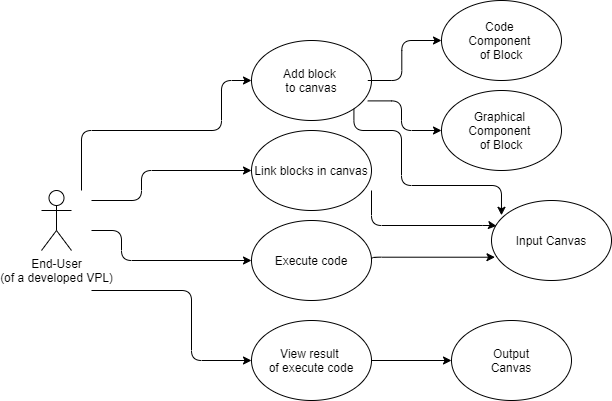
* Visually represent code graphically as some sort of object (we decided to call this object a “block”)
* Create relationships between blocks
* Link code to blocks
* Provide a canvas where the user would be able to drag/drop blocks
* A mechanism to execute blocks
* Displaying the results of executed blocks.

Below are two screenshots of existing VPL’s. The common general components are annotated.



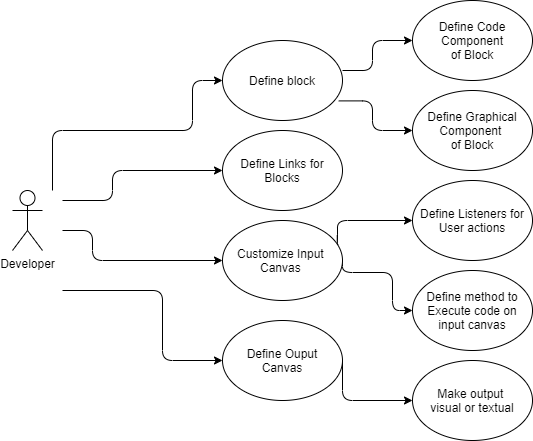
# **Use-Case Diagrams**

## End-User (Someone using a developed VPL):



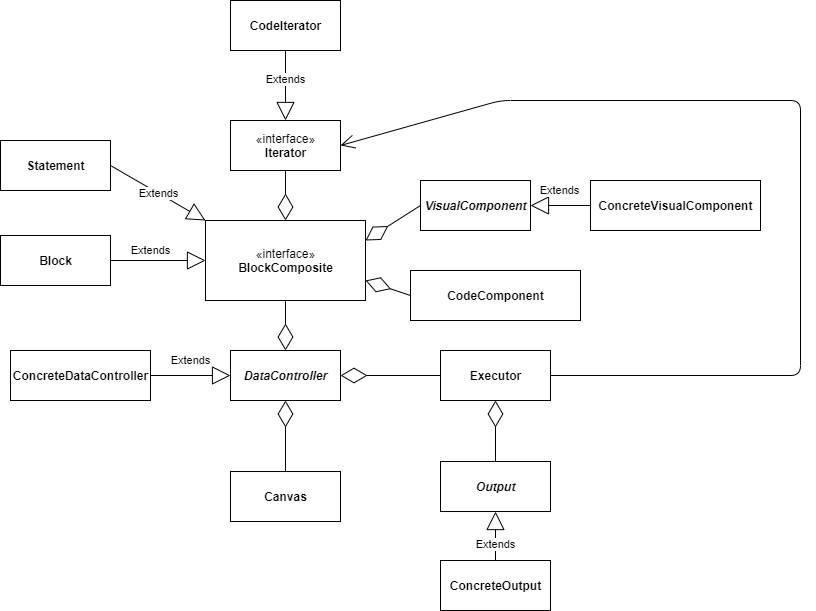
Note: We decided to include this use-case diagram because it helped us come up with the developer use-case diagram, which our system design is based on.

## Developer:



# **System Design**

## General UML



The blocks of the VPL are represented as a BlockComposite, which follows the Composite pattern. The Statement and Block classes implement BlockComposite where Statement represents a statement that would only be a single line of code, such an assignment statement. The Block class represents a block of code that would span multiple lines in a textual programming language and can contain Statement objects as well as other Block objects, similar to how if statements can be nested within other if statements.

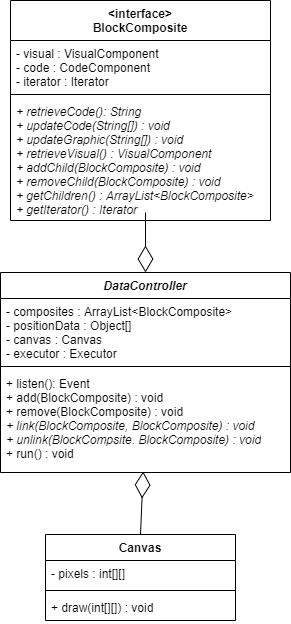
The BlockComposite class contains a VisualComponent and a CodeComponent, as we want to separate the visual aspect of the code block from the actual code portion of the block. The VisualComponent class is abstract as to provide a VPL developer with the flexibility to choose how the code blocks in their VPL will look like, which is done by extending the VisualComponent class into a concrete implementation, the ConcreteVisualComponent.

The DataController class contains the functionality for linking the blocks on the Canvas but is abstracted, as to provide a VPL developer the option to define how exactly blocks on the Canvas will link together. The class also provides methods that listen for specific events that occur on the Canvas, such as clicking or deleting a block. DataController is also responsible for providing the finalized program in the BlockComposite to the Executor.

The Executor uses an Iterator to iterate through and traverse the BlockComposite, creating a file of actual code from the CodeComponent of the blocks, before passing this file off to an external compiler. After compiling and executing, the program output is returned to the Executor, which then hands off the output to be displayed in the Output class.

Our framework abstracts the Output to allow the VPL developer to choose how to display the program output, whether it would be a graphical representation or even something akin to a console output. This customization of the Output class is done by implementing it into a ConcreteOutput class.

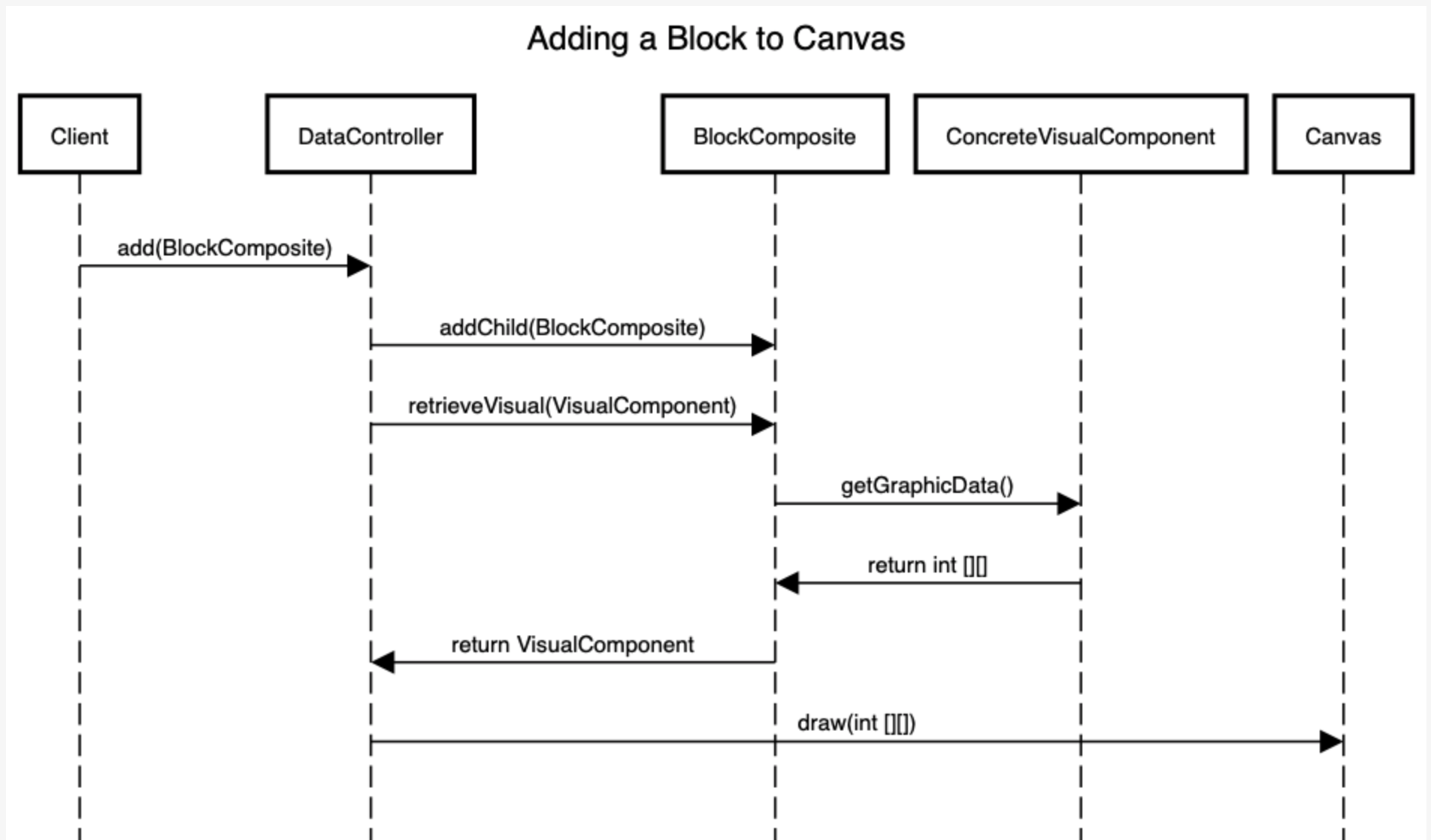
## The MVC Subsystem



The MVC Subsystem is composed of a BlockComposites(Model), a DataController (Controller), and a Canvas (View). The BlockComposites are responsible for storing all text related and numerical values related to a block. The Canvas is responsible for displaying any graphical information related to blocks or block values. The DataController is the means in which the BlockComposites can communicate with the Canvas. The DataController is responsible for updating the BlockComposites and the Canvas as well as telling the Canvas to draw information from blocks.

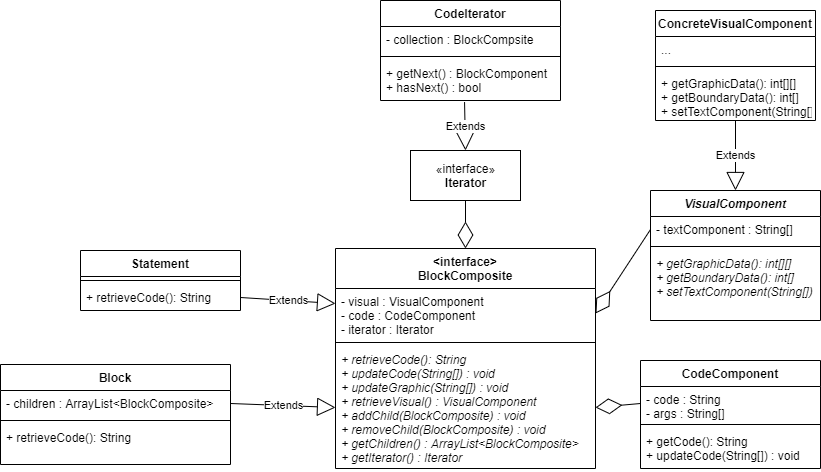
### View

#### Sequence Diagram



The Canvas works in a simple manner. It simply draws whatever the DataController demands. In the case that a client wishes to add a block to the Canvas, the client makes a request to the DataController. From here, the DataController updates the model or BlockComposite structure and receives data on how to draw the block in the form of a 2D integer array. The DataController can then forward this information to the Canvas where the Canvas can draw the block however the Canvas has been implemented to draw items.

### Model



The Model of the MVC pattern is the BlockComposite. The BlockComposite is an interface that contains information about the graphical and code data of each block in the VPL. The concrete classes of the BlockComposite are Statement and Block, with Statement representing a “leaf” of the composite tree, and Block representing the composite.

BlockComponent can be considered to be a Model because it holds all of the data associated with any block created by the developer. The BlockComposite class only holds data about how the block should be visually represented, and what code is linked to that specific block. The client should never interact with the BlockComponent’s data directly (more on this below).

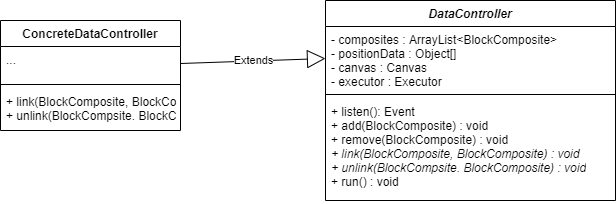
The VisualComponent deals with the graphical data of the BlockComponent. This abstract class contains information about how the block should look when placed on the canvas, along with boundary data to aid in collision detection of the blocks on the canvas. We allow the developer to extend this abstract class in order to create their own definitions of how a block should be portrayed on the canvas (for example, they could opt to use simple shapes to represent blocks, or load in a pre-existing texture).

The CodeComponent class contains information related to what code is linked to each individual block. This data is retrieved in order to execute the program.

Below are descriptions of what the methods associated with the Model do:

* retrieveCode(): Returns data associated with the CodeComponent
* retrieveVisual(): Returns the graphical information of the block
* updateCode() : Updates the CodeComponent with new data from the client
* updateGraphic(): Updates the VisualComponent based on whether or not the user inputs in custom arguments to the block
* addChild/removeChild: Used to remove a certain BlockComposite from the Composite tree
* getIterator(): Returns an iterator that begins with the current BlockComposite

### Controller



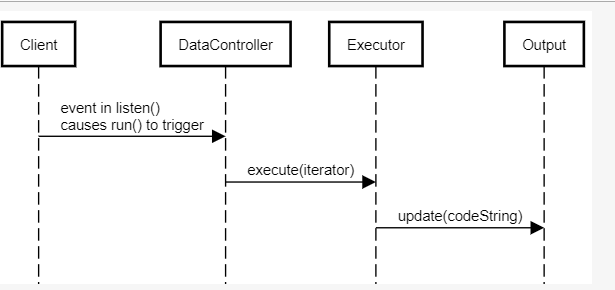
The DataController is an object that coordinates interaction with the Canvas in order to produce updates to the model. The way this is done is by ‘listening’ to the canvas. This is left as an abstract method for the developer to implement in their ConcreteDataController. The function listen() should handle any user interaction with the canvas, including running the output. The developer can make use of pre-implemented add, remove, and run functions to accomplish this. It is advised that the developer implement link() and unlink() to assist the listen function. Conceptually these abstract methods will define the functionality that organizes Blocks and Statements inside the BlockComposite. The field composites contain the array of BlockComposites, this should be manipulated using the add() and remove() functions. The link() and unlink() should also change this structure. Conceptually link() will merge two BlockComposites into one and unlink() will break apart two blockCompsoties from each other into two separate blockComposites. When run() is called, an iterator corresponding to the codeBlock to run is passed to the Executor.

As well as changing the BlockComposite structure, the dataController manages the positions of each block of code by holding that information in the positionData. The listen function will handle any changes to position via drag and drop and update the positionData object. With this data available the canvas draw(int[][]) can be called to update the canvas.

## Executor

# 

This subsystem consists of a few major components: the Executor, Output, and ConcreteOutput. The Executor stores the complete program that results from the Iterator traversing through the block Composite. The Output interface will contain the output that the program will produce. We decided to make this an interface with a ConcreteOutput object that will implement its functions because we wanted to allow flexibility for the developer’s VPL design. The developer can use this interface to decide how the program’s output will be displayed, and will have the ability to implement this themselves. The subsystem is called by the DataController which then uses the Iterator to put together the final program from the BlockComposite. This process is illustrated in the sequence diagram below.



# 

# 

# **Patterns**

For the framework, we decided to implement the Composite, Iterator, as well as the MVC design patterns.

## Composite Pattern

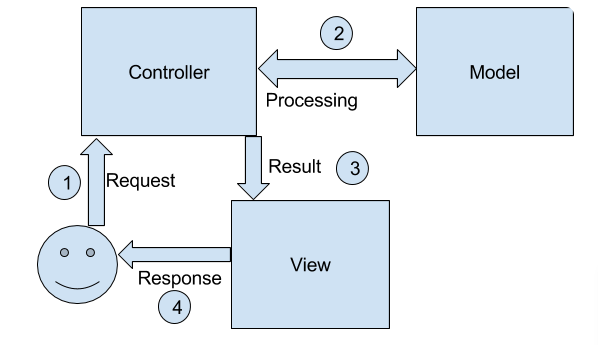
The Composite pattern is used to provide the developers with an uniform standard to organize the structure of the code blocks.

During our research we noticed that existing VPL’s (such as Scratch) and code in general, our team identified that pieces of code are commonly wrapped in other pieces of code.

An example of this nesting would be a print statement wrapped within a while loop. VPLs developed from our framework need a way to store this code in order to display it graphically as well as to execute it. Instead of forcing the developer to create a structure that links the blocks in some sort of list, we used the Composite pattern to provide a common interface for each block while allowing some blocks to hold references to other blocks, representing lines of code located within a loop or an if statement.

## Model-View-Controller Pattern

We designed the interactions between the Canvas, DataController, and BlockComposite as an implementation of the Model-View-Controller pattern. MVC systems are commonly used when dealing with user interfaces, which is one aspect of the framework VPL developers will need to implement. Developers would need to specify the visual aspects of the code blocks as well as the styling of the output canvas.



Referencing the above figure, the Canvas represents the View, the DataController represents the Controller, and the BlockCompsite represents the Model. Users who use a VPL developed from this framework would carry out actions that are interpreted by the DataController, synonymous with the user generating actions or events for an action/event listener. Based on what the user does (for example, dragging and dropping blocks), the DataController will retrieve the graphical data from the BlockComponent’s VisualComponent attribute, and decide how to update the Canvas according to the user’s action. The updated Canvas will then be displayed to the user. We thought this pattern was appropriate because the end user wouldn’t have to interact with the individual blocks at all. All interaction/updating is taken care of by the DataController.

## Iterator Pattern

The Iterator was used within our Executor system to work with the BlockComposite that we had initially established. This design pattern would be used to traverse through the block Composite, and put together each line of code provided by each block. The methods within the Iterator pattern determine the order in which the Iterator will traverse through the Composite. We decided to use the Iterator pattern because we needed a pattern that could traverse our complex data structure while providing the client with a way to collect the final program. While brainstorming about patterns we could use to accomplish this task, we had also considered using the Visitor pattern with our block Composite. However, we decided to ultimately use the Iterator because the Visitor provided functionality that would not be necessary for what we needed to accomplish. The Visitor pattern seemed less useful in this instance because it was more geared towards carrying out different behaviors for each node in our Composite tree. However, since we needed to traverse the tree and execute the same behavior for each node, the Iterator seemed more suited for this task.

## Non-Pattern

A design choice that we included that isn’t considered a “pattern” is allowing the developer to subclass the interfaces that we provide (such as VisualComponent and OutputCanvas). This gives them a huge degree of freedom in developing their VPL, as they aren’t constrained to a specific number of classes to choose from in order to create what they have in mind. However, since these classes all fall under an interface, we are able to take care of mundane tasks such as iterating through the code and updating the canvas without the need for developer intervention (although they do have the ability to overwrite certain processes that would be defined by the framework).

# 

# **Developer Example**

## Creating ConcreteVisualComponent

The developer will subclass the VisualComponent abstract class. For this example, they will use a Square to represent the graphical data. The Square will provide values to the graphicData int[][], coloring the pixels that represent a square black. In this example, there will be one textField that will be stored in the textField member variable. This will be updated through the setTextField method, which will also update the graphical data.

## Creating ConcreteDataController

The developer will subclass the DataController component, giving definition to the link, unlink, and listen methods. The link method will take in two BlockComponents (blockComponent1 and blockComponent2, respectively). This method will call addChild on blockComponent1 and pass in blockComponent2 as an argument. This will add blockComponent2 to the composite structure of blockComponent1. Afterwards, we remove it from the DataController ArrayList. Unlink will do the same thing, but inverted. It will call removeChild on blockComponent1 and pass blockComponent2, and then add blockComponent2 to the DataController ArrayList.

The listen function handles all user interaction with the Canvas. First the listen initializes all of the Block and Statement types with their associated CodeComponent and VisualComponent.These objects will be drawn on a side location in the Canvas. This part of the Canvas will stay static. When a single click and drag occurs on one of these BlockComposites, add will be called with a copy of the BlockComposite. When a double click occurs on a BlockComposite, run() will be called. When a right click on a BlockComposite occurs, remove will be called with that BlockComposite as a parameter. A single click and drag will ‘move’ the BlockComposite, changing the positionData. If a move causes two BlockComposites to collide, link() will be called with the two as parameters. If a BlockComposite is moved out of its current structure, then unlink will be called. After all of these checks, draw() is called on the Canvas to update it.

Pseudo-code example:

CodeComponent whileCode = new CodeComponent(code,1)

Block whileBlock = new Block(whileCode, new ConcreteVisualComponent)

CodeComponent varCode = new CodeComponent(code,1)

Statement varAssign= new Block(varCode, new VarVisualComponent)

.....

## Creating ConcreteOutput

The ConcreteOutput has a textOutput field as a string. When update() is called, this string is printed to an output terminal. ConcreteOutput also has a close function that cleans up the memory and terminates the program.

## Implementation in Client

The client must initialize the Canvas and the Executor. The Canvas and Executor are passed to initialize a ConcreteDataController. Then listen is called on dataController.

Pseudo-code:

Canvas canvas = new Canvas()

Executor executor = new Executor()

ConcreteDataController controller = new ConcreteDataController(canvas,executor)

controller.listen()

# Appendix

