Similitude

ENGI-980B

Project

Supervisor : Prof. Theodore S. Norvell

Advait Trivedi

MUN # : 201892202

Abstract

Similitude is a system that can be used by students as well as instructors to create, check or simulate digital circuits and also to check if a given circuit works correctly or not.

It is a multi-platform digital logic simulation system, although, predominantly web based. The tool allows the user to create digital logic systems that encompass various logic gates. The user can then create complex systems using logic gates and store them for later use. Moreover the system allows abstraction of full circuits to be used as a component in another circuits. The system allows users to login and maintain their sessions, ensuring privacy and discrete working.

The project has been partly completed by previous students and my work on this project will be to improve the front end structuring and make it well structured, adhering to the concepts of Object Oriented Programming, along with various Software design principles, to ensure that the code is easily extensible, reusable and adaptable to a better extent. Along with that, I have also worked on the creation for a model for simulation of signals/waveforms in the system, and also added work on the view and controller portions for simulating circuits and maintenance of simulation tests.

Contents

Chapter 1

Introduction

Carrying on from the 980A section of the Project, wherein a simple prototype was designed, keeping in mind what the requirements, or goals were, at that point. Similitude was going to be a web-based (works in the browser) system, which enables students, academicians and professors to design and operate upon digital logic circuits the likes of which are currently design using physical integrated circuits and VHDL. This would save the respective user time, and provide logistics for creation of circuits, thus, removing the physical need of using actual ICs at all.

Given that the system was primarily built in a hap-hazard manner, consistent work has been put into making a well designed system, which will be further talked about in detail in the upcoming chapters. There was a directed effort to drive the system structure and work to the given requirements, and to make it more scalable, robust and easy to work with for any developer in the future.

A major part of the work was put into the view and controller layers which would be supported by the existent model. They follow a *partly event driven architecture*, which further enables a developer to setup the system as a reactive-agent, as in the system can now react to user-performed event, using it's *internal state*. The model was re-worked to a fair extent too. The addition of the model manipulation sub-layer is one such construct that came up during work. Along with that there has been work on the future aspects of the system, namely simulation and adding a *value system* to the model, where in one can use the system to create logical signals, i.e. digital wave forms etc. to use in the system, with processes like simulation in the future. Further more features incorporated along with revamping the older structures in the back-end. The attribute syste was now added to allow editing logic gates/components for their properties like input ports etc. The normalisation process was also worked upon which ensures that a circuit is never left in an *abnormal* form, i.e. no extra links, all like ends close to ports are connected to ports etc.

Chapter 2

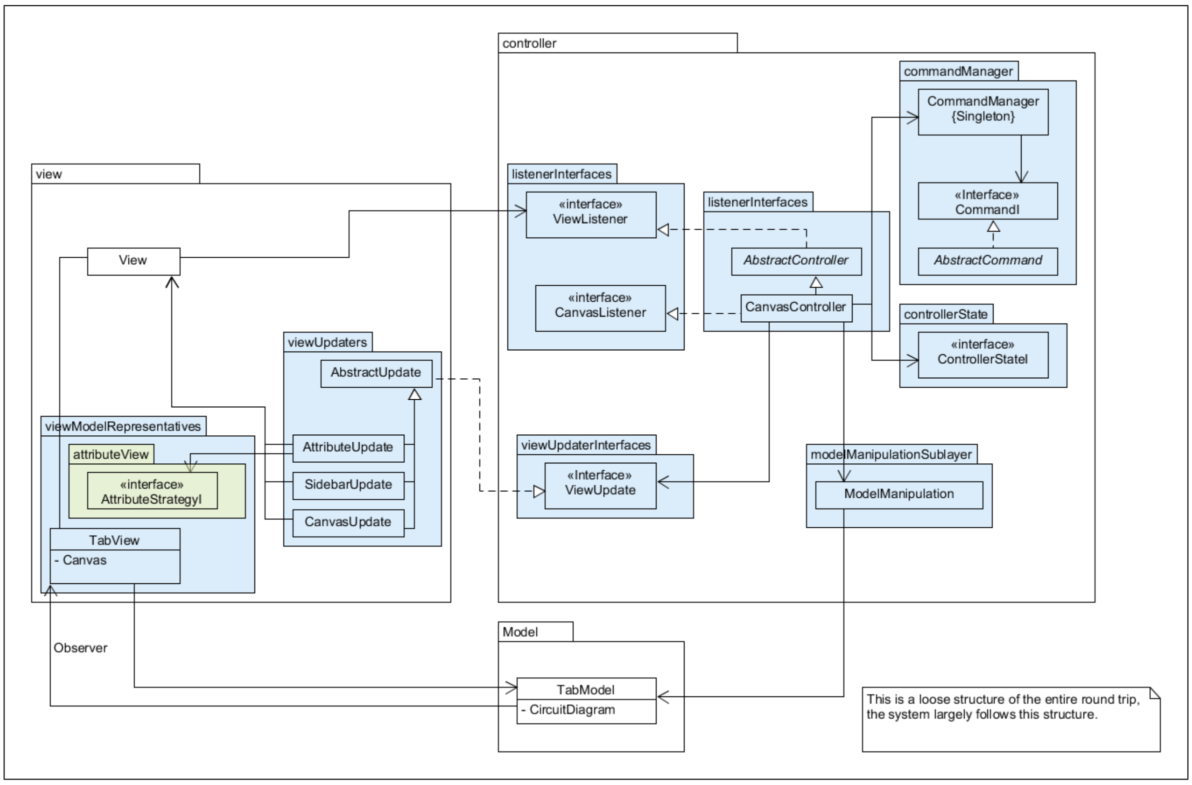
High level structure of the project

As mentioned earlier the project was majorly worked upon in the view and controller layers. The View and the controller as they fit in the Model-View-Controller (MVC) pattern [1], operate in a circle, i.e. the view sends user performed actions to the controller, packed as events, and the controller decides to perform changes on the model, through the model manipulation layer, by checking it's own internal state. This means that the controller action will vary as per it's internal state according to the performed user action. Once an action is deemed by the controller to be fit to change the model, it passes control to the model manipulation layer, which creates a command out of the action parameters and essentially executes the command, thus changing the model, properly. This is then through the observer pattern returned back to the view. Thus the view is then updated. There are also linkages through a ViewUpdate interface, just in case one wants to perform specific actions, through the controller, after an update.

This change comes after modification of the earlier version produced in the prototype of the project. It builds up on top of it, due to changes in requirements and additional features over time.

2.1 Model Manipulation Sub-layer

One of the bigger points about the inclusion of the model manipulation sub-layer was to accumulate all the changes to the model that the controller wishes to perform on the model into one place. Thus it becomes a single gateway for the controller to use to edit the model. It is mostly accessed by the Controller states majorly to perform state defined actions on the model, in the event that a change is even warranted for the incoming event and the present state. This kind of accumulation of responsibility simplifies the developers job and now any other developer working on the system does not have to check the entire code base to refer to changes made by the controller to the model, and instead look at this one class instead.



The high level representation of the full structure.

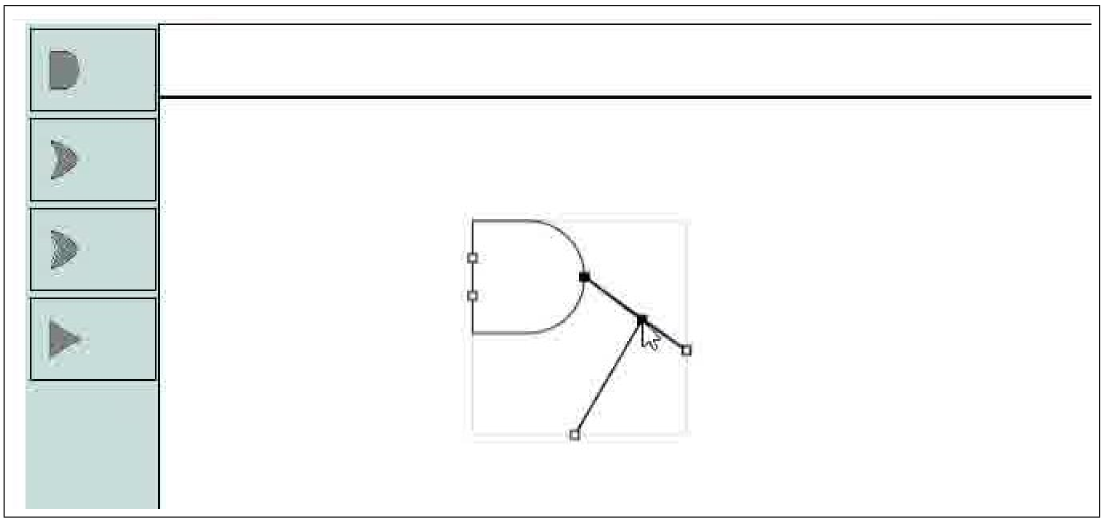
2.2 The Normalisation process

The normalisation process is essentially a path defined for any circuit that shall be designed. It is used to tidy up the circuit after each change to the circuit, so that it matches the way it looks. A circuit might be “denormalised” between an interaction, but as soon as it ends, the normalisation process is triggered and the circuit is normalised again. A normal diagram has the following properties.

- Any two distinct connections that are close to each other, both contain a port.
- Any two disctinct connections that are close to each other (and therefor both contain ports) must be connected by a link.
- No connection is close to a link unless it includes an endpoint of that link.
- No link has two endpoints that are directly connected to each other.
- No two links connect the same pair of connections.

It resides as an implemented algorithm in the model manipulation sub-layer.

Being a function/algorithm, there is no direct visuall aid to provide for it, apart from the fact that, it can be referred to directly in the code, which is pretty much self-explanatory.


An example of link resolution through normalisation, where in links are split through the process automatically.

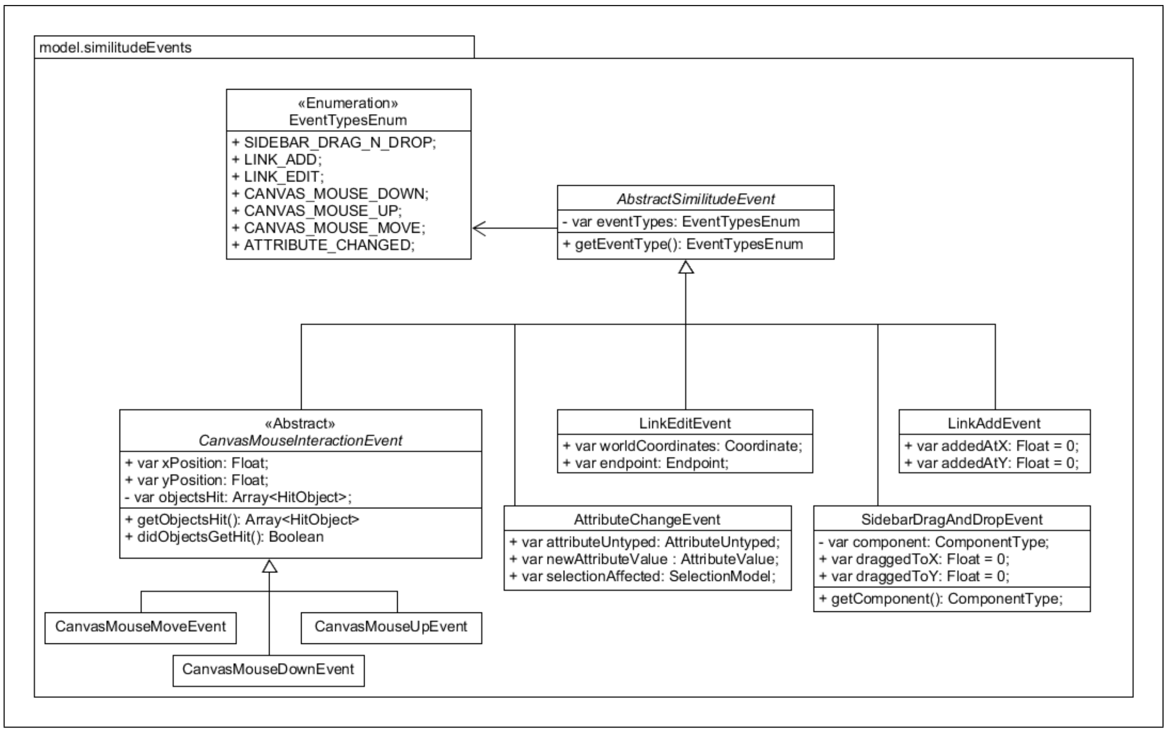
Chapter 3

Handling user actions through events and commands

3.1 The partial event driven architecture

The partly event driven architecture, was picked up to allow abstraction of user performed events. The user when performs actions are to be recorded and processed by the system. Using an event structure allows this kind of recording and processing, in a systematic manner. All events inherit from the “AbstractSimilitudeEvent” class, meaning that they all belong to this type. Each event has it’s own set of fields, which are necessary in different cases to record different types of events. These events are then in different portions of the code created, used and fired to the controller, which, again accepts any inheriting child of the super class “AbstractSimilitudeEvent”.

The reason why one would call this a “partial/partly” event driven, is because in real event driven systems, everything that happens to the system, i.e. any interaction with the system is treated as an event. There are designated roles for every entity that interacts with the system, and components of the system, internally also interact using events. The designated roles usually go from being event producers to consumers. Events use an event bus to travel through the system. None of this applies to Similitude. The only portion of a true event driven model, adopted in this system, is that of packing a certain relevant set of actions as events and using them in the controller. One can note that there is no event bus at all, and not everything that interacts with or happens to the system is treated as an event.


model.similitudeEvents
«Enumeration»
EventTypesEnum
+ SIDEBAR_DRAG_N_DROP;
+ LINK_ADD;
+ LINK_EDIT;
+ CANVAS_MOUSE_DOWN;
+ CANVAS_MOUSE_UP;
+ CANVAS_MOUSE_MOVE;
+ ATTRIBUTE_CHANGED;
AbstractSimilitudeEvent
- var eventTypes: EventTypesEnum
+ getEventType(): EventTypesEnum
«Abstract»
CanvasMouseInteractionEvent
+ var xPosition: Float;
+ var yPosition: Float;
- var objectsHit: Array<HitObject>;
+ getObjectsHit(): Array<HitObject>
+ didObjectsGetHit(): Boolean
LinkEditEvent
+ var worldCoordinates: Coordinate;
+ var endpoint: Endpoint;
LinkAddEvent
+ var addedAtX: Float = 0;
+ var addedAtY: Float = 0;
AttributeChangeEvent
+ var attributeUntyped: AttributeUntyped;
+ var newAttributeValue : AttributeValue;
+ var selectionAffected: SelectionModel;
SidebarDragAndDropEvent
- var component: ComponentType;
+ var draggedToX: Float = 0;
+ var draggedToY: Float = 0;
+ getComponent(): ComponentType;
CanvasMouseMoveEvent
CanvasMouseUpEvent
CanvasMouseDownEvent
The Events structure.

3.2 The command pattern

The command pattern [1] is a software design pattern that defines an interface for atomic and modular actions to be executed in a class relevant to it. It allows this atomic action to be undone or redone. This is achieved by defining methods to undo or redo the action itself in a class specific to the action, which implements the interface for the command. A command interface usually has 3 methods (but, not limited to always. One can choose to expand it) namely,

- undo()
- redo()
- execute()

A command also has command UIDs in Similitude that allow the command manager to differentiate between different commands, or in a converse manner, mark multiple commands with the same UID and treat them as a batch command, where in they can all be undone or redone by the commmand manager in one go.

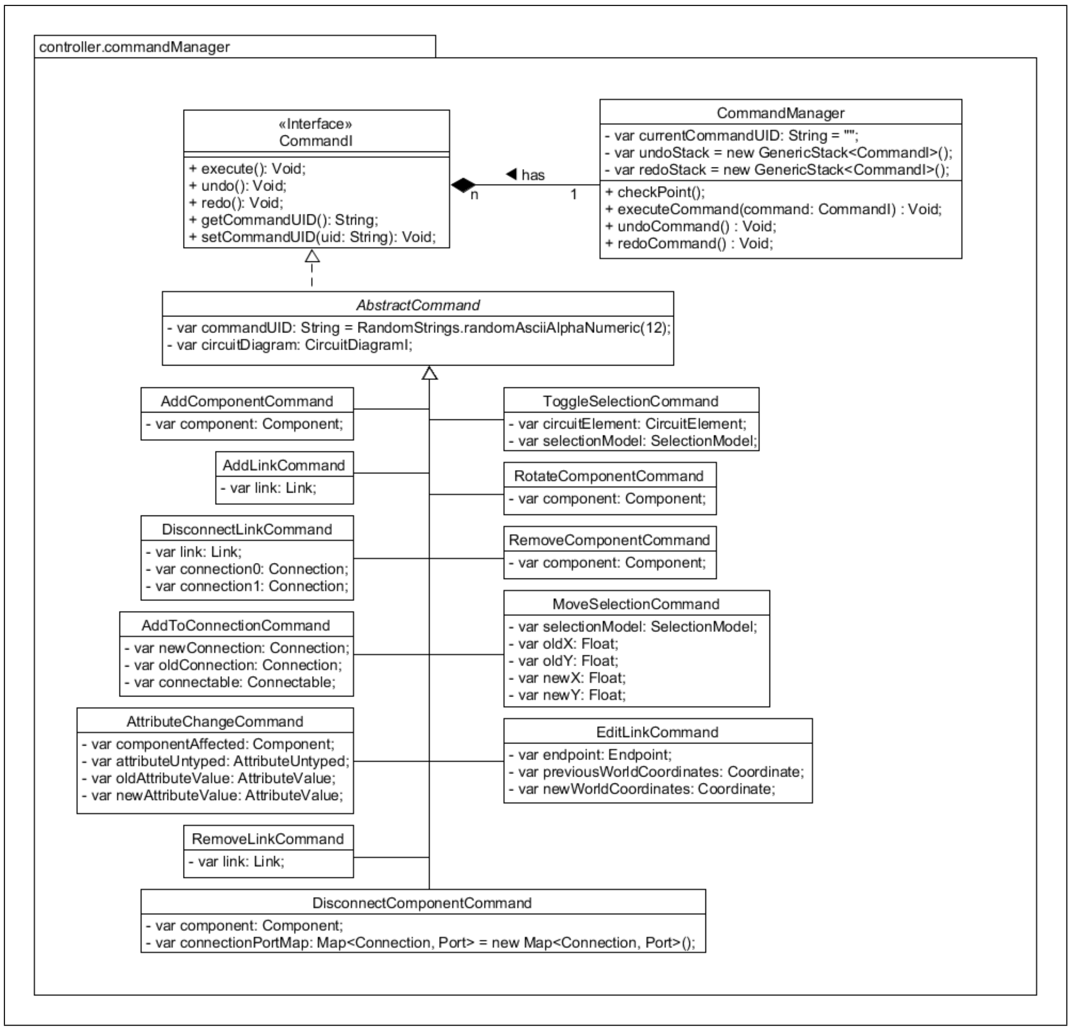
This class is then used by an internal singleton object, i.e the “CommandManager”. The command manager is responsible for housing an handling different command that are created on an incoming event by the model manipulation sublayer inside the controller. The process goes according to the following,

- A command is created through the model manipulation layer, if the controller decides to edit the model.

- this command is then given to the command manage through the

  ```
  execute(command: CommandI)
  ```

  function in the command manager, which in turn calls the command's execute() method to perform the change on the model.
- this executed command is then pushed on to the undo stack.
- if the user chooses to undo the action, then this command is on the top of the undo stack. It is popped by the command manager and then the undo() function in the command is called.
- Now, this undone command is pushed onto the redo stack.
- In the off chance that the user wishes to redo the action, the command is popped off the redo stack and the redo() function in it is called.

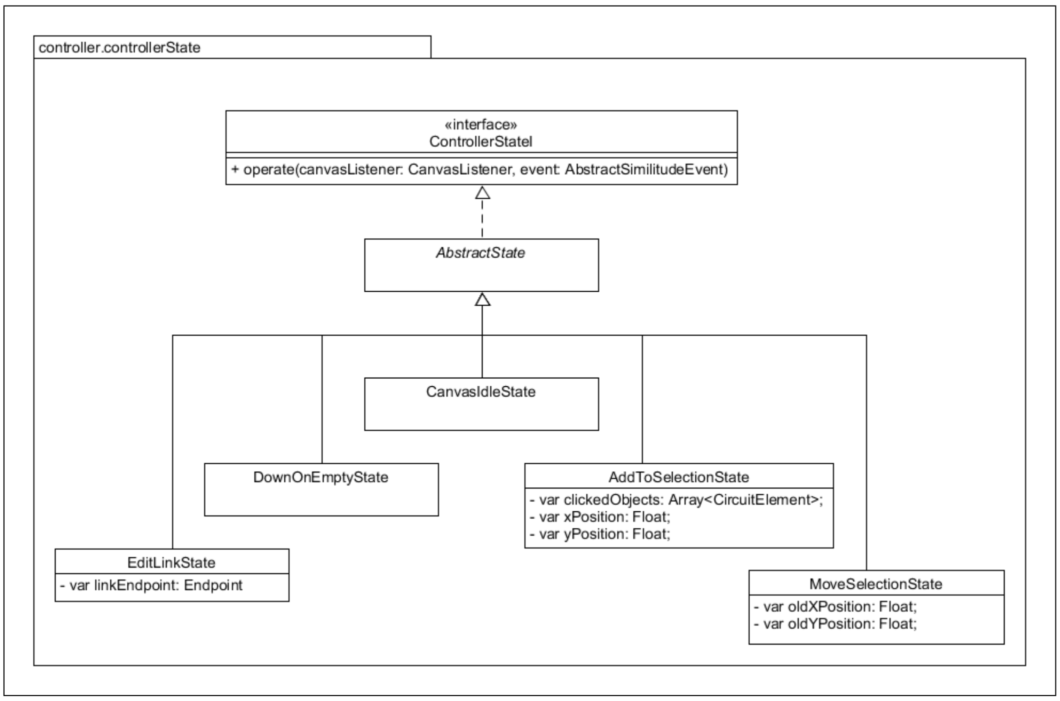

controller.commandManager
«Interface»
CommandI
+ execute(): Void;
+ undo(): Void;
+ redo(): Void;
+ getCommandUID(): String;
+ setCommandUID(uid: String): Void;
has
n
1
CommandManager
- var currentCommandUID: String = "";
- var undoStack = new GenericStack<CommandI>();
- var redoStack = new GenericStack<CommandI>();
+ checkPoint();
+ executeCommand(command: CommandI) : Void;
+ undoCommand() : Void;
+ redoCommand() : Void;
AbstractCommand
- var commandUID: String = RandomStrings.randomAsciiAlphaNumeric(12);
- var circuitDiagram: CircuitDiagramI;
AddComponentCommand
- var component: Component;
ToggleSelectionCommand
- var circuitElement: CircuitElement;
- var selectionModel: SelectionModel;
AddLinkCommand
- var link: Link;
RotateComponentCommand
- var component: Component;
DisconnectLinkCommand
- var link: Link;
- var connection0: Connection;
- var connection1: Connection;
RemoveComponentCommand
- var component: Component;
MoveSelectionCommand
- var selectionModel: SelectionModel;
- var oldX: Float;
- var oldY: Float;
- var newX: Float;
- var newY: Float;
AddToConnectionCommand
- var newConnection: Connection;
- var oldConnection: Connection;
- var connectable: Connectable;
AttributeChangeCommand
- var componentAffected: Component;
- var attributeUntyped: AttributeUntyped;
- var oldAttributeValue: AttributeValue;
- var newAttributeValue: AttributeValue;
EditLinkCommand
- var endpoint: Endpoint;
- var previousWorldCoordinates: Coordinate;
- var newWorldCoordinates: Coordinate;
RemoveLinkCommand
- var link: Link;
DisconnectComponentCommand
- var component: Component;
- var connectionPortMap: Map<Connection, Port> = new Map<Connection, Port>();

The Command pattern structure.

Chapter 4

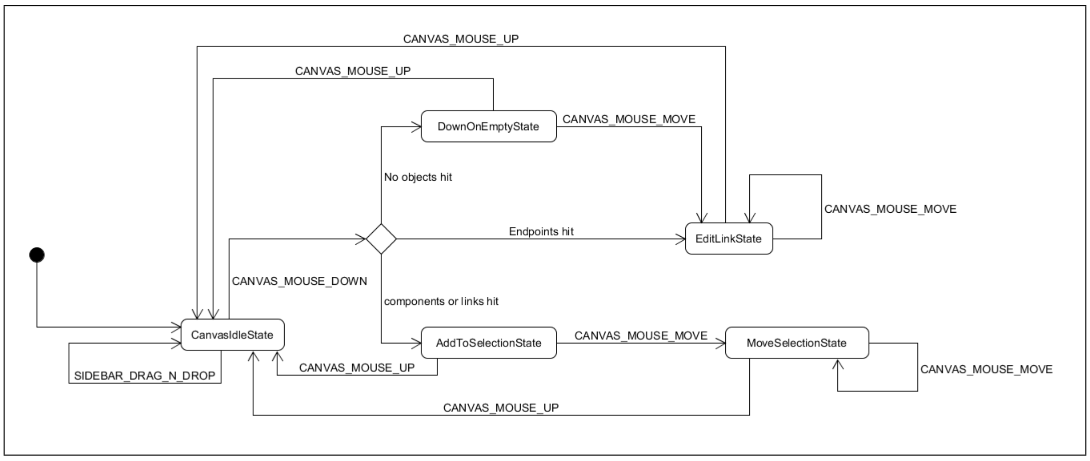
A reactive system

The reactive system something that was imagined over the idea of allowing the system to interact with the pre-defined events in a different fashion depending upon the situation in which the event is received. This means that the system will maintain an internal state which will govern these actions, based on the incoming event. The system essentially, receives an event, checks the state, creates and executes necessary commands, and then updates it's state accordingly.


controller.controllerState
«interface»
ControllerStateI
+ operate(canvasListener: CanvasListener, event: AbstractSimilitudeEvent)
AbstractState
CanvasIdleState
DownOnEmptyState
AddToSelectionState
- var clickedObjects: Array<CircuitElement>;
- var xPosition: Float;
- var yPosition: Float;
EditLinkState
- var linkEndpoint: Endpoint
MoveSelectionState
- var oldXPosition: Float;
- var oldYPosition: Float;

The Controller state pattern structure.

This also points a flow of states that are predefined and used in the system. Each state was manually crafted to fit the flow as imagined while creating the system. Such a state pattern reduces developer load of handling each use case in a unique manner, and allows for easy encoding of states using an event-state combination. Thus ensuring that there is one discrete state for each desired effect on the model/system, according to various events that lead to the state or away from it.



The transition between predefined states.

As mentioned before a state pattern was employed here, which took the "AbstractSimilitudeEvent" as an input to operate upon itself. A state pattern, in Software design allows the developer to encapsulate state specific functionality in a class designated for each state, thus properly adhering to the Single Responsibility Principle, which categorically states that each class has to hold functionality only and only relevant to itself and this is not to be extended, repeated or replicated in any other class. Thus the developer now does not have to redefine individual state logic, and uses state objects instead, as a whole to call actions in it with required parameters to operate and change itself as an when needed [1].

In Similitude the state interface simply has an operate() function that allows the state to directly call the function each time it is called upon for an event.

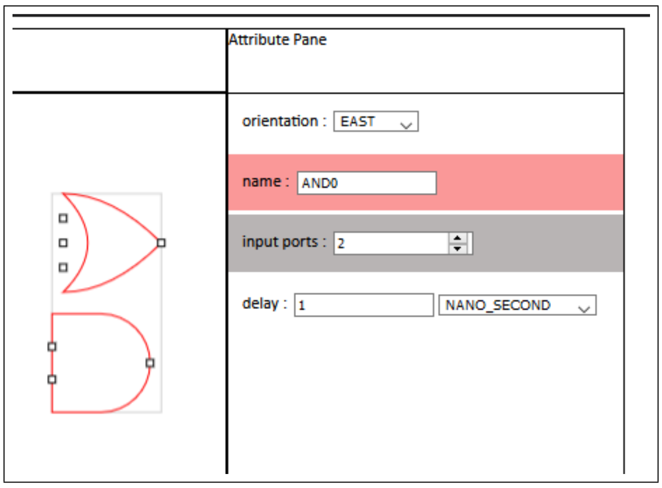
Chapter 5

The attribute system

The attribute system was a feature that was added to enhance logic gate/component editability. It allows the user to select and edit multiple logic components. It is used with a visual aid in the attribute pane. The attribute model hosts, four basic kinds of attribute types, namely,

- String, which was used to name a component, each component has a unique name, which can't be the same as any other component.
- Time, this attribute type has been used to denote time periods, which in the current system is used to allow the denotion of delays, which are a part of natural ICs as defects.
- Orientation, used to denote the orientation of the attribute.
- Integer, used to denote numbered values in attributes. Currently, the number of input ports on a component.

Each attribute type is linked to an attribute Value, these are then mapped together in their respective component. These values can be retrieved and compared using their corresponding singleton "AttributeType" objects.



Attribute Pane

orientation : EAST

name : AND0

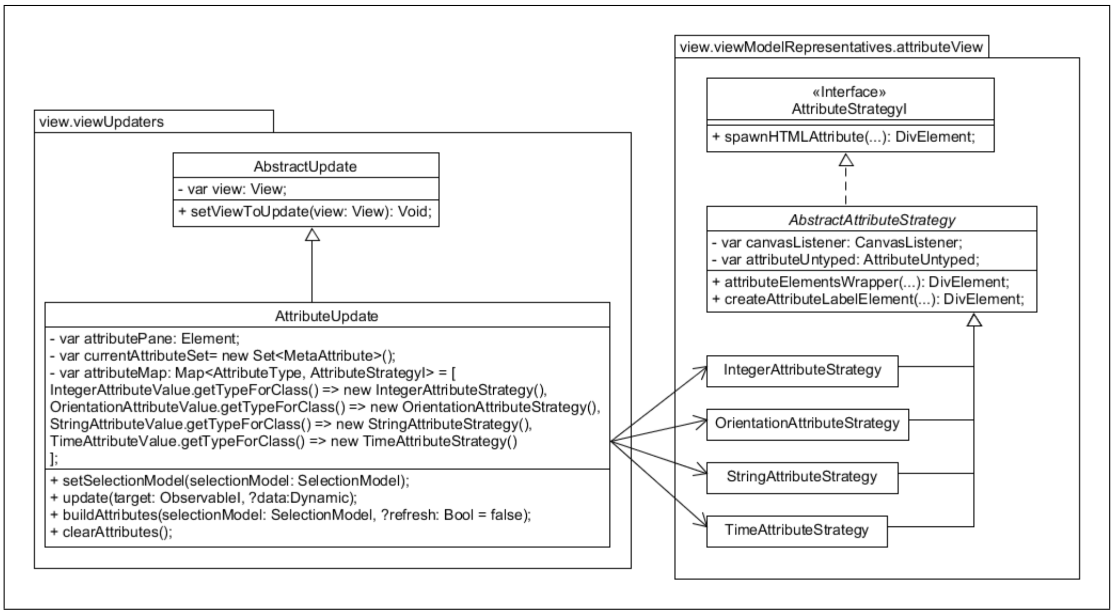
input ports : 2

delay : 1 NANO_SECOND

The attribute pane showing feedback for wrong values, as well as different values.

As you can see in the figure above, that red marked attributes, denote wrong/invalid values, i.e. they are actually wrong, and cannot be true for the selected set of components. In the above example this is denoted by the name attribute which can't be the same for 2 or more components. The grey colour denotes that the attributes of all the selected components for that attribute type are different, but can remain that way.

The View layer holds a strategy pattern [1] to house the population of the UI of these attributes. Each strategy essentially holds a different algorithm to display the attribute in the UI with controls to edit it, thus, enabling the developer to seemlessly add more strategies in the future, in a modular fashion and not worry about disturbing or breaking the rest of the system.


view.viewUpdaters
AbstractUpdate
- var view: View;
+ setViewToUpdate(view: View): Void;
AttributeUpdate
- var attributePane: Element;
- var currentAttributeSet= new Set<MetaAttribute>();
- var attributeMap: Map<AttributeType, AttributeStrategyI> = [
IntegerAttributeValue.getTypeForClass() => new IntegerAttributeStrategy(),
OrientationAttributeValue.getTypeForClass() => new OrientationAttributeStrategy(),
StringAttributeValue.getTypeForClass() => new StringAttributeStrategy(),
TimeAttributeValue.getTypeForClass() => new TimeAttributeStrategy()
];
+ setSelectionModel(selectionModel: SelectionModel);
+ update(target: ObservableI, ?data:Dynamic);
+ buildAttributes(selectionModel: SelectionModel, ?refresh: Bool = false);
+ clearAttributes();
view.viewModelRepresentatives.attributeView
«Interface»
AttributeStrategyI
+ spawnHTMLAttribute(...): DivElement;
AbstractAttributeStrategy
- var canvasListener: CanvasListener;
- var attributeUntyped: AttributeUntyped;
+ attributeElementsWrapper(...): DivElement;
+ createAttributeLabelElement(...): DivElement;
IntegerAttributeStrategy
OrientationAttributeStrategy
StringAttributeStrategy
TimeAttributeStrategy

The strategy pattern used for the creation of the UI for the attributes.

Chapter 6

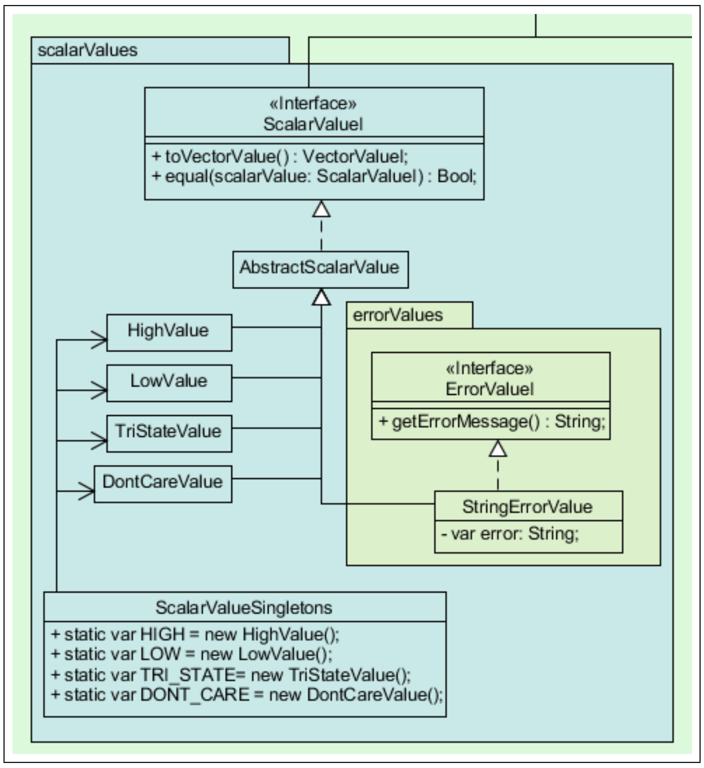
The Value system

The value system enhsrines a newly created structure to support digital logic values in the system. It was planned to house multiple inputs to logic gates, as well as signal streams or complete waves of data.

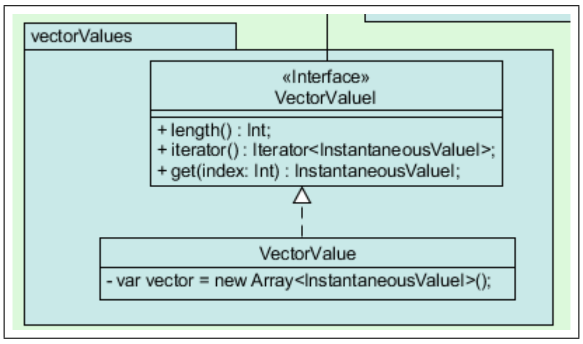
A conceptual classification of how values are seen in the system, and how a developer should envision them, is,

- Complete signals
- Instantaneous values
  - Vector values
  - Scalar values
    - High
    - Low
    - Don't Care
    - Tri-State (Z)
    - Error

It is divided in 2 types, namely the complete signal (a.k.a SignalValue) and the instantaneous values, which denote the value at a given instance of time. Signals are made up of a stream of instantaneous values mapped as a function of time. Instantaneous values can themselves be classified into two other types, namely vector values and scalar value. Vector values denote multi input lines, i.e. multiple scalar values in one unit of time. Scalar values are singular values, of which four, namely, High, Low, Tri-state and don't care values are singletons. Error values are charted to be created each time to be used and then destroyed later on. The reason being that error values have been planned to be of multiple types themselves, denoting different kinds of errors. This renders them unusable as singletons, and also carrying only a loose interface, which cannot always be concretely defined.


scalarValues
«Interface»
ScalarValueI
+ toVectorValue() : VectorValueI;
+ equal(scalarValue: ScalarValueI) : Bool;
AbstractScalarValue
HighValue
LowValue
TriStateValue
DontCareValue
errorValues
«Interface»
ErrorValueI
+ getErrorMessage() : String;
StringErrorValue
- var error: String;
ScalarValueSingletons
+ static var HIGH = new HighValue();
+ static var LOW = new LowValue();
+ static var TRI_STATE= new TriStateValue();
+ static var DONT_CARE = new DontCareValue();

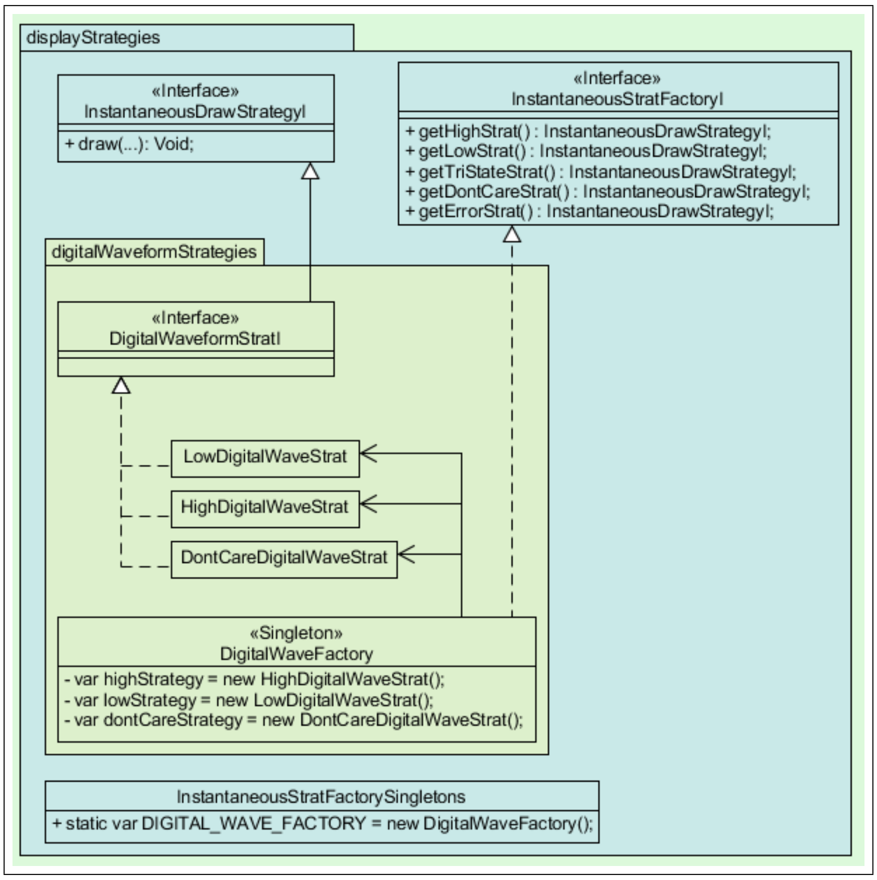
The structure for scalar values.


vectorValues
«Interface»
VectorValueI
+ length() : Int;
+ iterator() : Iterator<InstantaneousValueI>;
+ get(index: Int) : InstantaneousValueI;
VectorValue
- var vector = new Array<InstantaneousValueI>();

The structure for vector values.

The model itself, houses, the UI designs through a strategy pattern [1] for each signal that will in the future allow for change, or addition of new strategy to exist. It uses strategy pattern on two different levels where in each signal is going to draw itself using a drawing adapter. The first level defines an over-arching strategy, say digital waveforms or numbered waveforms, which can be called directly using their singleton objects. In the system they are called "StratFactories". These strat factories have on the deeper second level defined algorithms to draw individual respective values, according to their overarching parent. This kind of two level modularization allows the developer to add as many over-arching designs, in a completely isolated manner from each other.

The "SignalDrawingAdapter" employs the adapter pattern [1], and lies in the view, it supports decoupling of the view and the model, and enables, using different value drawing systems to attach themselves to the adapter if something like that is decided in the future.


displayStrategies
«Interface»
InstantaneousDrawStrategyI
+ draw(...): Void;
«Interface»
InstantaneousStratFactoryI
+ getHighStrat( ) : InstantaneousDrawStrategyI;
+ getLowStrat( ) : InstantaneousDrawStrategyI;
+ getTriStateStrat( ) : InstantaneousDrawStrategyI;
+ getDontCareStrat( ) : InstantaneousDrawStrategyI;
+ getErrorStrat( ) : InstantaneousDrawStrategyI;
digitalWaveformStrategies
«Interface»
DigitalWaveformStratI
LowDigitalWaveStrat
HighDigitalWaveStrat
DontCareDigitalWaveStrat
«Singleton»
DigitalWaveFactory
- var highStrategy = new HighDigitalWaveStrat();
- var lowStrategy = new LowDigitalWaveStrat();
- var dontCareStrategy = new DontCareDigitalWaveStrat();
InstantaneousStratFactorySingletons
+ static var DIGITAL_WAVE_FACTORY = new DigitalWaveFactory();

The strategy pattern used for the creation of the UI elements for values.

There were quite a bit of planning hurdles that came along which were resolved after some meticulous thought. The two-layered strategy pattern for the UI designs was something that might still need a bit of tuning, given that there is already an implemented digital wave form strategy in there, but, there are different strategies like A number waveform which might be challenging to implement given the currently defined interfaces.

Chapter 7

Comparison and research into other digital logic simulators

A lot of the UI aspects one would see in similitude are pretty similar to other digital logic simulators out there. The UIs that came up in preliminary research were for circuitverse.org and logic.ly.

The ability to drag and drop components onto the canvas, connecting them by dragging links are something common to a lot of digital logic simulators. What similitude might have as an improvement is it's ability to normalise the circuit, which allows for the afore mentioned uses cases in the chapter called "The normalisation process". The uses cases that similitude's normalisation covers are not as extensively covered in a lot of other tools. The ability to also create multiple links even without a connection to a component is something that is uniquely seen in similitude. This might be because there is a lot of extra weight in creating and maintaining a loose link in a circuit and trying to normalise it further. Most editors would drop this in the favour of a functional UI, and avoid the extra load to deal with this altogether. Other than that the value system will be quite a good feature, when fully developed with a better planned interface in the future. This would be an added plus in the system.

The downsides of the current UI, though are that they are built to support the clean up that was performed in the entire project and to allow confirmation for the same. This means that it is not a dynamic UI that changes with screen sizes. It is much more rudimentary, and might call for some work, until it looks a bit more polished.

Chapter 8

Conclusion

This project was a learning experience, it allows one to see how different software design patterns interact with each other in a large system, where one has all the freedom to mix and match a lot of ingenious design ideas. Along with that this was also an experience in working in a co-ordinated fashion in 2 people teams with a proper goal based approach.

As mentioned earlier, the value system can be charted for improvement or expansion in future considerations. A lot of the newly adapted parts of the system are scalable and should allow any developer to expand upon them with ease, and with no rigidity or fragility in the system. The UI can also be subject to a complete overhaul if need be, or significant changes in the future.

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.