CENG 491

DIGIMOD
FINAL DESIGN REPORT

GROUP NAME: **MANAS YAZILIM**

GROUP MEMBERS: **Gökhan ÇAPAR**

Hüseyin B. AYYILDIZ

A. Serhat DEMİR

Seltchouk AHMED

1. INTRODUCTION 4
  1.1 Problem Definition and Project Scope 4
    1.1.1_Problem Definition 4
    1.1.2_Goals and Objectives 4
    1.1.3_Statement of Scope 5
2. DESIGN CONSTRAINTS 6
  2.1 Constraints 6
  2.2 Limitations 6
3. FUNCTIONAL MODEL AND DESCRIPTION 7
  3.1 Data Flow Diagrams 7
    3.1.1_Level 0 DFD 7
    3.1.2_Level 1 DFD 8
    3.1.3_ PSPECS – The Process Specifications 8
4. ARCHITECTURAL DESIGN 9
  4.1 System Decomposition 9
    4.1.1_GUI Module 10
    4.1.2_Canvas Module 10
      4.1.2.1_JGraph Overview 11
        Keyboard Bindings 11
        Customization 11
        Customization a Graphs Display 11
        Selection 12
    4.1.3_Simulator Module 12
      4.1.3.1_JHDL Overview 13
      4.1.3.2_JHDL Primitives 13
        Cells and Wires 13
        Logic Gates 14
        Clock 14
      4.1.3.3_Other Building Blocks 16
        Adders and Subtractors 16
        Multiplexers and Buffers 16
        Modules 16
      4.1.3.4_Library 17
      4.1.3.5_The Simulator 17
      4.1.3.6_Validation 17
    4.1.4_File System Module 18
      4.1.4.1_ DIGIMOD File Format 18
      4.1.4.2_ DIGLOG File Format 21
    4.1.5_Scripting Module 24
      4.1.5.1_ Boolean Functions 24
  4.2 Transition from Canvas to JHDL 26
5. UNIFIED MODELLING LANGUAGE DESIGN 28
  5.1 Program Use Case 28
  5.2 Scripting Use Case 31
  5.3 Class Diagrams 34
    5.3.1_Canvas Class Diagram 34
      5.3.1.1_Canvas Class Diagram Description 34
      5.3.1.2_GWire Class Diagram 35
      5.3.1.3_Gate Class Diagram 36

5.3.1.4_ManyInputGate Class Diagram 37
5.3.1.5_CELL Class Diagram 38
5.3.1.6_Markers Class Diagram 39
5.3.1.7_Clock, Switch, LED Class Diagrams 40
5.3.1.8_FlipFlop Class Diagrams 41
5.3.1.9_Canvas Event Control Class Diagram 42
5.3.1.10_Undo&Redo Class Diagram 43
5.3.2_GUI Class Diagram 44
5.3.2.1_ JGraph Class Diagram 45
5.3.2.2_ GUI Class Diagrams 46
5.3.2.3_GUI Event Control Class Diagram 47
5.3.3_File I/O Class Diagram 48
5.4 Drawing Activity Diagram 48
6. HELP DESIGN 49
7. SCHEDULING 50
7.1 Gantt Chart 50
8. TESTING 50
8.1 White Box Testing 50
8.2. Black Box Testing 51
9. CONCLUSION 51

1. INTRODUCTION

1.1 Problem Definition and Project Scope

1.1.1_Problem Definition

Resulting product of our seven month development process is a digital logic simulator which we named DIGIMOD. DIGIMOD let users to draw and plot logic circuits, open or save files, store current circuit into library, import and export LGF files, zoom in and zoom out, use its own script...etc. With its user friendly GUI the purpose of DIGIMOD is to fill in digital logic simulator family with its fascinating features which are all free.

Detailed aspects, objectives, goals, functionality of DIGIMOD are stated in the following sections.

1.1.2_Goals and Objectives

We aim to build full functional digital logic simulator which will make developing and working in logic circuits much easy then ever. We will handle this with our script which brings several fascinating incomes. Overall features of DIGIMOD's script can be found in following sections.

We want DIGIMOD to have user friendly GUI. Most of free digital logic simulators are bad designed in visual way, and the ones who have fascinating GUI's are need to be paid.

Our goal is to submit a free, graphically well designed digital logic simulator to users especially for academic purposes.

Another goal for DIGIMOD is that it will be a totally safe and stable product. As we stated above we hope DIGIMOD will also be used for academic purposes it is important for the product to be stable and safe. A failure during the logic circuit development process would cause failure of the user and DIGIMOD would be an unreliable product for users which actually we don't want to face.

We believe that this system will increase the number of people using this kind of tools, because as well as providing a practical way of developing circuits, our system contains a multifunctional script which eases life.

1.1.3_Statement of Scope

DIGIMOD is a powerful multi platform program that can be used while working on digital logic circuits.

A user will be able to,

- develop circuits with this tool.
- save current file.
- open an existing file.
- send gates and circuit items in to library.
- retrieve gates and circuit items from library.
- edit circuits by cut/copy/paste/move features.
- use zoom in and zoom out on circuit panel in order to improve visual needs.
- print desired circuits.
- import/export LFG files.
- derive appropriate circuit with a logic function using script.
- derive output of a circuit by supplying input using script.

This information is given in order to state the scope of our project and product not to explain how it is handled. We will go into details in further parts of this report.

2. DESIGN CONSTRAINTS

2.1 Constraints

DIGIMOD should be carefully designed in order to work efficiently and satisfy requirements stated before. On the other hand there are some design constraints which we have to consider while designing our program.

Since DIGIMOD will be platform independent we have several restrictions for choosing development tools. We may encounter several problems because of this constraint. We will handle this problem using JAVA tools.

DIGIMOD will use LGF files during its operations which forces us to dissolve LGF file format. Since it is worthwhile to use DIGLOG's library this constraint becomes very important.

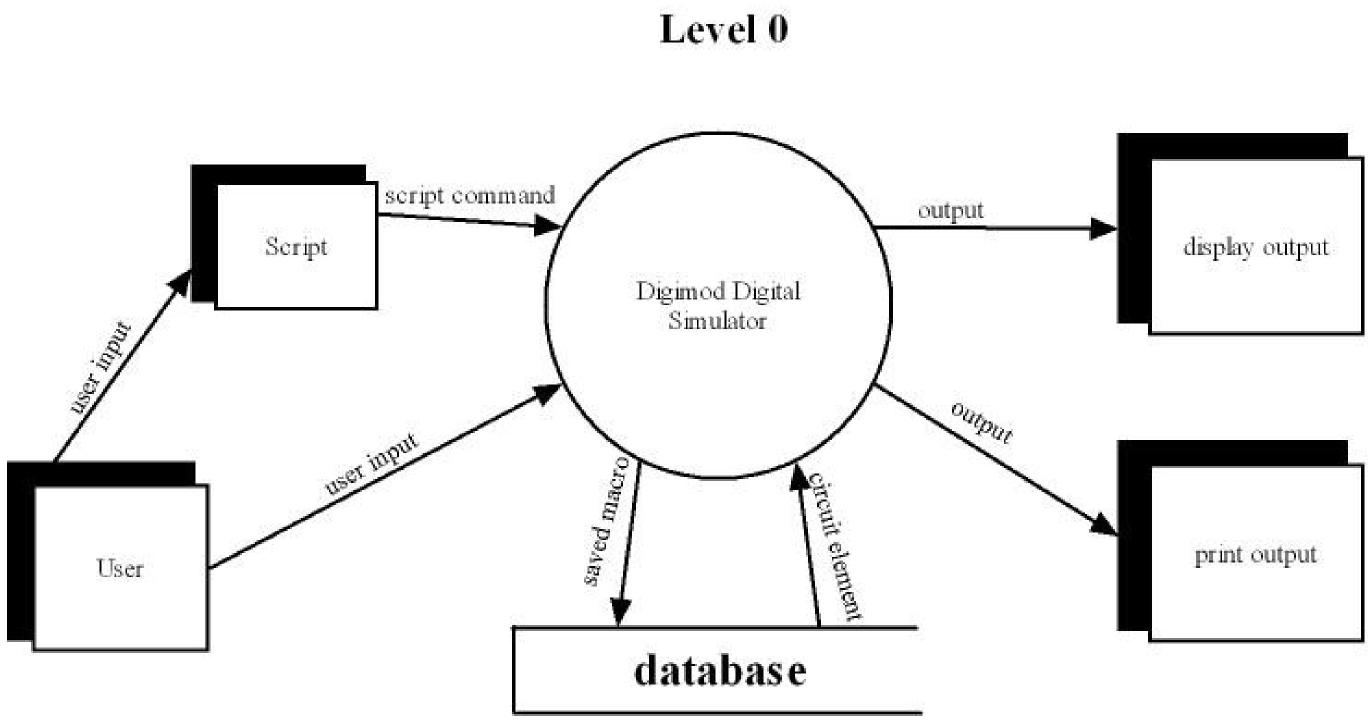
2.2 Limitations

- Time limitation. Since we have six months left we have to prepare an adequate schedule. We also have to comply with this schedule in order to success.
- Hardware limitation. In order to give enough sources to development tools and DIGIMOD, we will use at least 1000 MHz CPU, 256 Mb RAM, 40 GB hard drive.
- Portability. As stated before DIGIMOD will be platform independent program. It can be used in every operating system safely.
- Programming language. Our final decision of programming language is JAVA. At the time of our analysis part we were thinking to use .NET as our development platform. But in order not to separate our development tools we recede from using .NET since we will have to use a few tools for scripts, simulators and some graphic needs. We will handle all this within JAVA platform. But there were other choice using C++ but we give up that idea too with same reason. We are planning to integrate simulator in to our platform and other facilities like graphic libraries and script languages will be handled all in to same interface

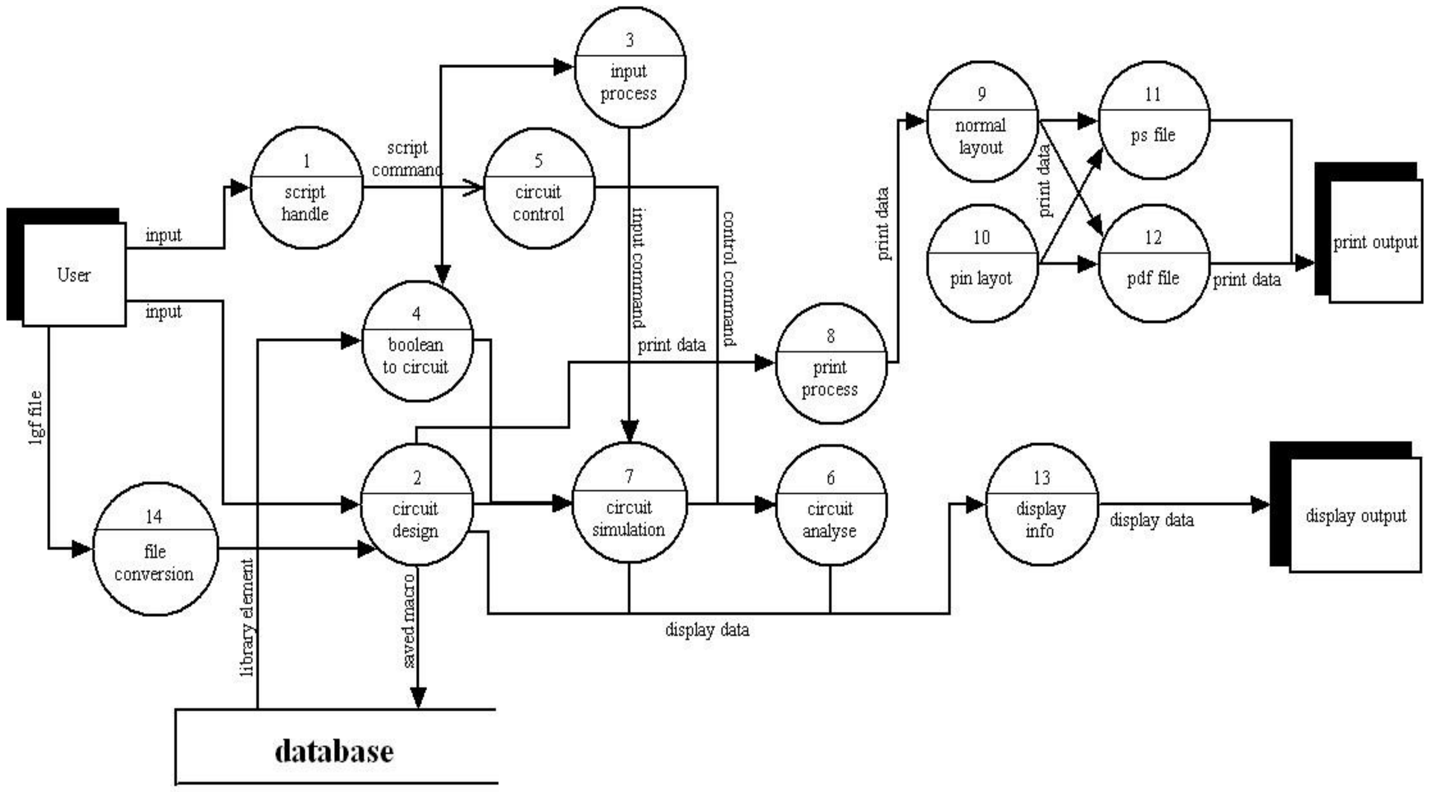
3. FUNCTIONAL MODEL AND DESCRIPTION

3.1 Data Flow Diagrams

3.1.1_Level 0 DFD



3.1.2_Level 1 DFD



3.1.3_ PSPECS – The Process Specifications

PSPEC: circuit design

Since our program is a circuit simulator. It will also allow users to design their circuits by hand, using the built in elements in the library. After a new circuit is designed, it can be added as a new element to the library so that it will avoid complexity of bigger designs and it will be available for future use.

PSPEC: input process

This is an important feature for our program. Program will accept inputs with the help of the script and users will access outputs without running the program. This will save a lot of time during processing of large number of files.

PSPEC: circuit control

During simulation and analysis of a circuit, users may want to change inputs or state of the circuit. This will be also handled in our program so that users can modify and see circuit behavior while the program is running.

PSPEC: Boolean to circuit

Our program will be able to read Boolean functions and turn them into logic circuits.

PSPEC: circuit simulation

After designing or importing a circuit, users may simulate their designs to see if it works correctly.

PSPEC: circuit analysis

This feature will help users to find their errors and also see the behavior of the circuit. We will monitor what is going on during simulation using graphs.

PSPEC: print process

We will have two main types of print support. First is directly getting printout from the printer and the second one is printing to a file. Users can have both PS and PDF file formats available for printouts.

PSPEC: file conversion

Our program will support also diglog file format. It will import LGF files and process on their circuits.

4. ARCHITECTURAL DESIGN

4.1 System Decomposition

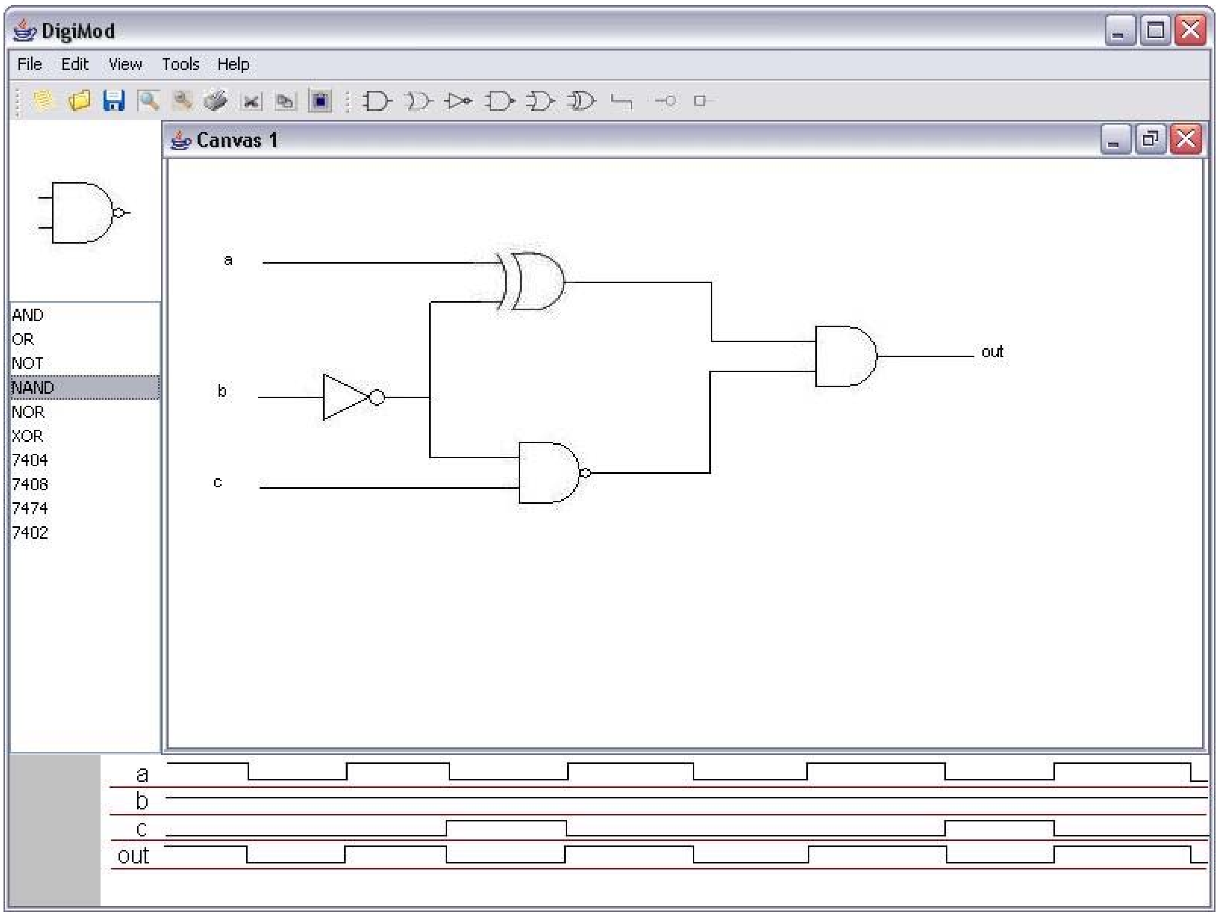
We have divided our system into 5 modules which are

- GUI
- Canvas
- Simulator
- File System
- Scripting

4.1.1_GUI Module

GUI module is built using SWING library. Our Graphical User Interface is simple but functional and self-explanatory. When user executes the program an empty screen arises. At the center of the screen Canvas widow appears. At the top there is menu bar and under menu bar lies standard tool bar. User can do basic file editing operations and insert primitive circuit elements to the circuit using this bar. At the left hand side there is a component list which shows selected library elements names. When a library element is chosen in the list, its graphical representation will appear right above the list. Double clicking to name of the element makes it appear on the canvas.

Below you can see an example view of the GUI.



4.1.2_Canvas Module

Canvas Module is one of the most important modules of our system because user directly interacts with this module to design circuits. Our design allows working with multiple canvases on the fly. We are using JGraph library for drawing all the circuit elements on the canvas. Detailed information about design of this module is available in Canvas Class Diagrams section.

4.1.2.1_JGraph Overview

A JGraph object doesn't actually contain data; it simply provides a view of the data. Like any non-trivial Swing component, the graph gets data by querying its data model. JGraph displays its data by drawing individual elements. Each element displayed by the graph contains exactly one item of data, which is called a cell. A cell may either be a vertex or an edge. Vertices may have neighbors or not, and edges may have source and target vertices or not, depending on whether they are connected. Class diagram of JGraph is available in Class Diagrams section.

Keyboard Bindings

JGraph defines the following set of keyboard bindings:

- Alt-Click forces marquee selection if over a cell.
- Shift- or Ctrl-Select extends or toggles the selection.
- Shift-Drag constrains the offset to one direction.
- Ctrl-Drag clones the selection.
- Doubleclick/F2 starts editing a cell.

You can change the number of clicks that triggers editing using setEditClickCount().

Customization

There are a number of additional methods that customize JGraph. For example, setMinimumMove() defines the minimum amount of pixels before a move operation is initiated. setSnapSize() defines the maximum distance for a cell to be selected. setFloatEnabled() enables/disables port floating.

With setDisconnectOnMove() you can indicate if the selected subgraph should be disconnected from the unselected rest when a move operation is initiated. setDragEnabled() enables/disables the use of Drag And Drop, and setDropEnabled() sets if the graph accepts Drops from external sources.

Customization a Graphs Display

JGraph performs some look-and-feel specific painting. We can customize this painting in a limited way. For example, we can modify the grid using setGridColor() and setGridSize(), and we can change the handle colors using setHandleColor() and setLockedHandleColor() methods.

If we want finer control over the rendering, we can subclass one of the default renderers, and extend its paint()-method. A renderer is a Component-extension that paints a cell based on its attributes. Thus, neither the JGraph nor its look-and-feel-specific implementation actually contain the code that paints the cell. Instead, the graph uses the cell renderers painting code.

Selection

Apart from the single-cell and marquee-selection, JGraph's selection model also allows to "step-into" groups, and select children. This feature can be disabled using the setAllowsChildSelection() method of the selection model instance.

Since we are interested in knowing when the selection changes, we have implemented the GraphSelectionListener interface and add the instance using the method addGraphSelectionListener. valueChanged will be invoked when the selection changes, that is if the user clicks twice on the same vertex valueChanged will only be invoked once.

4.1.3_Simulator Module

This module is the heart of the system. Simulation part of DIGIMOD is built on JHDL. JHDL is a set of Field Programmable Gate Array (FPGA) CAD tools that allows the user to design the structure and layout of a circuit, debug the circuit in simulation and interface for bit-stream synthesis, and so forth. JHDL is the best choice for us to use, since it is free, open-source and easy to configure and customize. In this section we will explain how we built our program on JHDL.

JHDL allows us to write an optional behavioral model for any structural cell. This capability is provided for a number of purposes:

- All library primitive elements are behaviorally modeled. If you want to add new primitives to JHDL you will create behavioral models for them.
- Once a complex module such as a multiplier is debugged, a behavioral model of it can be used in subsequent simulations of designs using it. This will make the simulations run much faster.

4.1.3.1_JHDL Overview

Simply put, JHDL is a structurally based Hardware Description Language (HDL) implemented with JAVA. It provides object classes used to build up circuit structure. These include:

- static cells, such as boolean gates, registers, etc.,
- parameterizable modules, such as multipliers, counters, etc., and
- generic platform-independent APIs

for creating wires and basic circuit elements.

Available appendages to the JHDL circuit model include a set of tools for debugging, simulating, testing and interfacing to the circuit, both as it exists in simulation ("in software",) and while the program is executing on an FPGA ("in hardware.") These appendages interact with the JHDL circuit model through three well defined APIs:

- Circuit Structure and Circuit State APIs allow for the creation of netlisters and other specialized viewers.
- Simulator APIs allow tools to control execution of the simulator (for both simulation and hardware execution) as well as receive key feedback from the simulator.

JHDL is an exploratory attempt to identify the key features and functionality of good FPGA tools. The design of an FPGA system has three major arenas:

- The structure or organization of the circuits.
- The layout of the circuits.
- The interface of the FPGA circuits with the host application software.

4.1.3.2_JHDL Primitives

Cells and Wires

In our design, each circuit file is enclosed in a "cell." In order to interface with other circuit elements the cell's interface will be described when objects created. The interface lists the input and output ports into and out from the cell. Types of cell ports are in or out. An additional cell interface parameter declares the cell parameters (variables). Once an instance of the cell is created then parameters to cell interface ports (in/out) will be connected.

After overhead has been set up, user can create the circuit's logic by combining logic elements and tying them together with Wires.

Like everything in the program wires are also JAVA objects. There are two kinds of wires. The first will be used for connecting distinct circuit cells. Second used inside a cell to make necessary connections between gates.

Logic Gates

There are 6 primitive logic gates. Namely;

- AND
- NOT
- OR
- XOR
- NAND
- NOR

Other gates and cells are combinations of them.

Structurally instantiating every single wire and component in a design would be cumbersome, it was determined that a package of commonly used components would be extremely useful. To make creating multi-bit elements easy, different sized wires will be available. In addition to that, most of the methods will take variable-width arguments, based on the widths of the wires that are passed as parameters. The methods that will represent gates will be overloaded accordingly. These methods will return wire objects. If required, returned wires can be named and used separately.

All Boolean gates will be capable to accept between 2 and 9 input wires. Boolean gate methods with up to four inputs will accept arbitrary-width wires; methods with five or more inputs only accept 1-bit wires.

Clock

Clocking can be handled either implicitly or explicitly. That is, if a single global clock is adequate for a design, the user need never explicitly declare clocking structures and circuits with synchronous elements (registers/flipflops/etc) are automatically connected to a globally synchronous clock.

On the other hand, it is possible for the user to define his own clocking. This can be for a single clock design or for a design with multiple clocks.

Every clock driver in the system has a clock schedule that represents pattern of bits which will be repeated on the clock wires. A clock schedule is applied during across the length of a cycle. Each entry in the clock schedule is a step in the cycle. So for the default clock, the schedule is "01" meaning that in the first phase of a cycle's simulation the default clock is low and for the second phase the default clock is high.

Step and cycle commands are used to tell the simulator to execute the simulation. A step executes a single element of the clock schedule. Cycle executes to the end of the current cycle.

There are three basic elements to clocking.

- Synchronous elements
- Clock drivers
- Clock wires

Synchronous elements require a clock to produce their outputs. This would include registers, memories, and any simulation models that use clocks. Synchronous elements can be rising-edge triggered, falling-edge triggered or both. When a rising or falling edge is seen on the clock input of a synchronous element they will be "clocked". That is, the *clock()* method of the element will be called.

A clock driver is a simulation abstraction which was introduced to help simulate these new features. A clock driver is simply a pattern generator which generates the clock signal applied to synchronous elements. The clock driver generates a signal based on its schedule. The schedule is used to determine when synchronous elements should be clocked. Since clock drivers are a simulation abstraction and are not netlist-able, they should exist only in test benches, like stimulators. A clock wire is simply a wire which is connected to a clock driver and to the clock input of synchronous elements. These typically connect the clock driver directly to synchronous elements, but there may be other gates in between. If there are, it is called a gated clock circuit. Gated clocks slow down the simulation tremendously because the simulator doesn't know a priori when they are going to cause the clocking of synchronous elements. Thus, it cannot pre-schedule the evaluation of gated clock synchronous elements. We will not use gated clocks unless they are absolutely necessary in our design.

Default clocking mode should be sufficient for designs which contain only a single global clock. To use it, we will simply use the implicit clock versions of Logic method calls and primitive constructors. When doing so, JHDL automatically creates a global clock wire and wire it up to all synchronous cells in the design. In addition, it creates a simulation model of a clock driver which provides the stimulus needed to simulate our circuits.

However, there will be times when a design must contain more than one clock. A typical example would be an FPGA board where the memory subsystem is clocked at a different rate than the FPGA fabric. In this case, the user cannot use default clocking but must define the various clocks in the system, their simulation schedules, and explicitly state which synchronous elements are wired to which clocks.

4.1.3.3_Other Building Blocks

Adders and Subtractors

We will provide methods to instantiate basic adders and subtractors in the target architecture. These methods will optionally accept carry-in and carry-out signals if desired. Each of these methods returns the wire connected to the "sum" port of the adder / subtractor.

Multiplexers and Buffers

There will be 2:1, 4:1, or 8:1 multiplexers and a wire connection buffer element will be available in the program library.

Modules

A module is a pre-built, parameterizable JHDL design component, i.e. a multiplier or counter. They are technology independent circuit elements, by adding more predefined circuit building blocks. Some of the modules that will be present are followings;

- Accumulator: clearable accumulator which supports both addition and subtraction
- Array Multiplier: parameters allow specification of combinational or pipelined operation, signed or unsigned operands.
- Booth Multiplier: sequential booth multiplier, retires 2 bits at a time.
- Integer Divider: parameters allow specification of combinational or pipelined operation, signed or unsigned operands.
- Adders and subtractors: basic adders and subtractors.
- Comparators: greater than, less than, and the like
- Cordic: a family of CORDIC units (one circular and one linear) with parameters to allow the selection of mode (vector, rotation, unified), word size, etc.
- Delay line: provides parameterizable number of cycles of delay. Implemented using on-chip RAM's where possible.
- Down counter and Up Counter: clearable and loadable.

- Floating-point multipliers and adders: signed/unsigned, user defined mantissa and exponent sizes

4.1.3.4_Library

DIGIMOD will have a large library that is provided by JHDL. In addition to that user defined cells can be saved to the library. It is easily handled using cell interface structures. When a user wants to define a new library object, he will be asked to specify the interface. In other words, the number of inputs and outputs of the macro will be defined. After that, using symbol editor, a new symbol created to represent visualization of the macro in drawings. The module generator libraries distributed with JHDL provide parameterized, complex, hand-placed building blocks which users can leverage to simplify the design of JHDL-based circuits.

4.1.3.5_The Simulator

We will use JHDL’s circuit simulator directly in our program. The three basic simulator commands are reset, step, and cycle. When program is being used to simulate a circuit, the first call made to the simulator is always reset. DIGIMOD makes the reset call automatically when you cycle or step for the first time.

You can do a reset at any time during a simulation to reset everything back to its initial state, preparatory to restarting the simulation.

The step command is used to run the circuit to the next changing clock edge or if there are multiple clocks the circuit will be run until one of the clocks change.

The cycle command is similar to the step command, except that it runs the circuit to the end of the current clock cycle. For example, the default clock has a schedule of "01" and a cycle command will set the clock low and propagate the circuit and then set the clock high and propagate the circuit. In the special case that the user has called step prior to a cycle command; a subsequent cycle command will simply finish out the current cycle.

Errors may occur during the simulation of a circuit. Some of the most common simulation errors are putting multiple values on the same wire during a clock step or not putting a value on a wire during a clock step (floating wire). In both these cases a simulation exception is thrown.

4.1.3.6_Validation

In addition to providing a mechanism for the entry of designs, JHDL provides a full framework for the visualization, simulation, and verification of the resulting circuits. JHDL provides an open, API-based framework to allow for the creation of a wide variety of circuit visualization tools. Thus, all information required is provided for us to create our own simulation and verification environment.

4.1.4_File System Module

All file operations are made by this module. We have decided to keep our files in xml format. In addition program will support Diglog files with .lgf extension. Printing is also handled by this module.

4.1.4.1_ DIGIMOD File Format

We will store files in XML format. DTD of our drawing files as follows;

```
<!ELEMENT CELL_MACRO (DRAW)>
<!ATTLIST CELL_MACRO
  id            CDATA    #REQUIRED
  name          CDATA    #REQUIRED
  description   CDATA    #IMPLIED
  width         CDATA    #REQUIRED
  height        CDATA    #REQUIRED
>

<!ELEMENT DRAW (INPUT_MARKER, OUTPUT_MARKER, CELL, AND,
MANY_AND, OR, MANY_OR, NAND, MANY_NAND, NOR, MANY_NOR, NOT, XOR,
SWITCH, CLOCK, WIRE)>
<!ATTLIST DRAW
   grid         CDATA   #REQUIRED
>

<!ENTITY % position
 "x             CDATA       #REQUIRED
  y             CDATA       #REQUIRED"
>

<!ENTITY % boolean
  "true | false"
>

<!ENTITY % wire_state
  "HIGH | LOW | NC"
>

<!ELEMENT CELL EMPTY>
<!ATTLIST   CELL
  id            CDATA          #REQUIRED
  %position;
>
```

```
<!ELEMENT  INPUT_MARKER EMPTY>
<!ATTLIST   INPUT_MARKER
  name           CDATA           #REQUIRED
  %position;
>

<!ELEMENT  OUTPUT_MARKER EMPTY>
<!ATTLIST   OUTPUT_MARKER
  name           CDATA           #REQUIRED
  %position;
>

<!ELEMENT  AND EMPTY>
<!ATTLIST   AND
  %position;
>

<!ELEMENT MANY_AND EMPTY>
<!ATTLIST MANY_AND
  %position;
  input_count   CDATA          "3 | 4 | 5 | 6 | 7 | 8 | 9"
>

<!ELEMENT  OR EMPTY>
<!ATTLIST   OR
  %position;
>

<!ELEMENT MANY_OR EMPTY>
<!ATTLIST MANY_OR
  %position;
  input_count  CDATA          "3 | 4 | 5 | 6 | 7 | 8 | 9"
>

<!ELEMENT NAND EMPTY>
<!ATTLIST NAND
  %position;
>

<!ELEMENT MANY_NAND EMPTY>
<!ATTLIST MANY_NAND
  %position;
  input_count  CDATA          "3 | 4 | 5 | 6 | 7 | 8 | 9"
>

<!ELEMENT NOR EMPTY>
<!ATTLIST NOR
  %position;
>
```

```
<!ELEMENT MANY_NOR EMPTY>
<!ATTLIST MANY_NOR
  %position;
  input_count  CDATA          "3 | 4 | 5 | 6 | 7 | 8 | 9"
>

<!ELEMENT NOT EMPTY>
<!ATTLIST NOT
  %position;
>

<!ELEMENT XOR EMPTY>
<!ATTLIST XOR
  %position;
>

<!ELEMENT SWITCH EMPTY>
<!ATTLIST SWITCH
  %position;
  state         (%boolean;)       #REQUIRED
>

<!ELEMENT CLOCK EMPTY>
<!ATTLIST CLOCK
  %position;
  counter         CDATA            #REQUIRED
>

<!ELEMENT POINT EMPTY>
<!ATTLIST POINT
  %position;
>

<!ELEMENT WIRE (POINT+)>
<!ATTLIST WIRE
  state         (%wire_state;)    #REQUIRED
>
```

Above Document Type Definition keeps all required information about a drawing. It keeps coordinate information, input & output specifications of circuit elements. All connections are made via wires. Since wires are straight lines, starting and end points of wire and state of wire information will be kept in XML. Connection information gathered from coordinates of wires and circuit elements.

4.1.4.2_ DIGLOG File Format

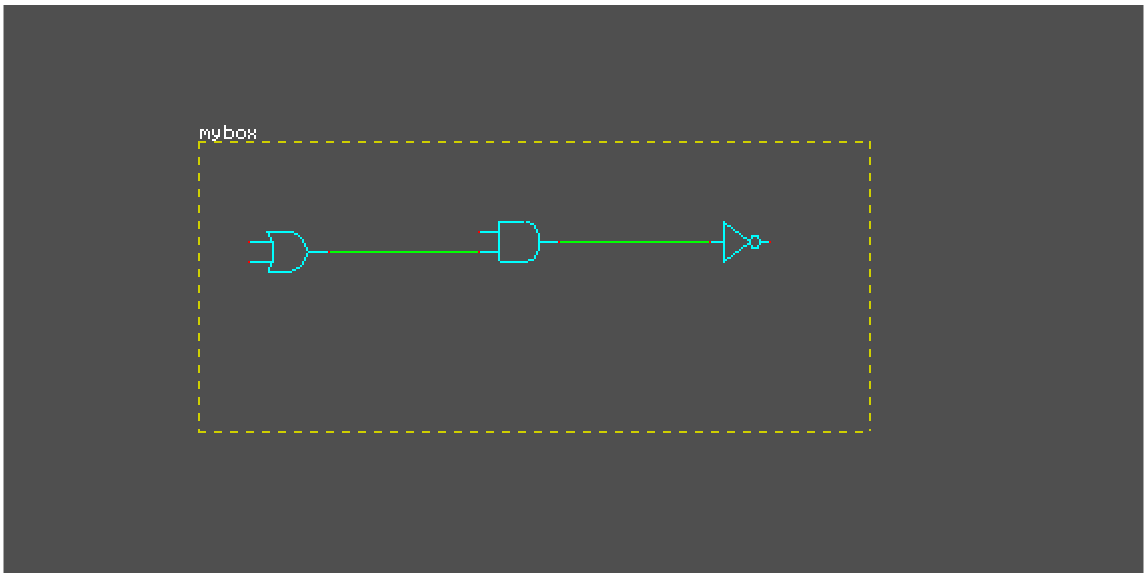
As we have declared before, DIGIMOD will be capable of importing and exporting LGF files. Let’s consider importing first. One can import a LGF file by three ways: first he or she can use main window menus File-Open-and browse anyfile.LGF as he/she desires. Second way is to use shortcuts on the main program. When user press ctrl+o file dialog box is opened and he or she can choose his or her LGF file. Final method is using script program. User can type “load <filename.LGF> “ on script and importing process is started.

Transferring data from current local LGF file to DIGIMOD’s main interface will be managed in a tricky way. Since we could not find any information about the structure and convention of LGF file system, we examined several LGF files from the simplest one to most complex one. Roughly, our aim is to transfer and render the codes in the file to graphics on the main program. Since we will render the codes as previously stated, our importLGF function should have a parameter which determines the mode of the function. If this parameter –named mode – is zero function will clear main circuit design interface and then loads the file. This is actually what we mean by import LGF file. But this is not enough because interacting with library is an essential of our program and when an object is loaded to main page user’s current circuit should not vanish. Fortunately our second mode arises in order to handle this problem. When user wants to load an object from library, this is actually a kind of LGF importing, importLGF function is called with parameter mode which is assigned to one that tells to program that load file into current page without any loss of data. Process starts after mode is given zero or one according to user request.

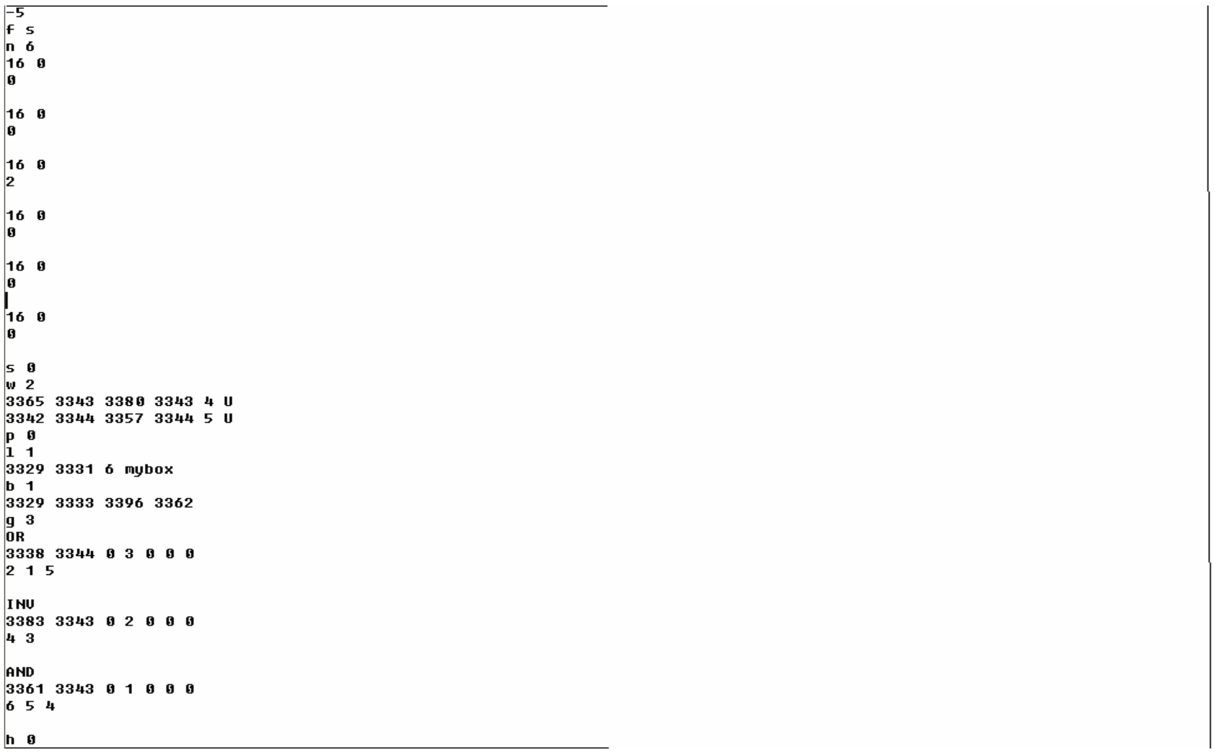
When a user uses script program to load a file, program looks for the file, if the file could not be found in the current directory an error message is given to user. If the file is found our function is called with appropriate parameters. Using graphical user interface to open a file is a certain process so there is no possibility not to find the file because it is chosen by user from current directory.

When process is invoked, importLGF function searches desired file. If it can not find stops the process and raises an error, otherwise continues to work. Our function will read data from the file sequentially. In order to make things clear giving a circuit and its LGF format will be helpful.

A digital logic circuit:



LGF file format description of this circuit.



```
-5
f s
n 6
16 0
0

16 0
0

16 0
2

16 0
0

16 0
0

16 0
0

s 0
w 2
3365 3343 3380 3343 4 U
3342 3344 3357 3344 5 U
p 0
l 1
3329 3331 6 mybox
b 1
3329 3333 3396 3362
g 3
OR
3338 3344 0 3 0 0 0
2 1 5

INV
3383 3343 0 2 0 0 0
4 3

AND
3361 3343 0 1 0 0 0
6 5 4

h 0
```

LGF file format has a standard that it start with -5, so first controlling the first character will be an appropriate and fast way to classify this file as a LGF file definitely. Then function will read other characters sequentially. As another standard LGF files have “f s” letters after -5, so function will try to find these characters in order to continue to read. Now it is time for our function to control mode argument. If mode is zero, current page will be erased so user will be informed about this process and after user’s approval process will continue. “#” is used in LGF file in order to comment the code. So function will skip this character and those who comes after that as a comment sentence.

In file “n” is the node number of LGF file. It is the sum of input and output parts of the gates. For example if there is only one gate, let’s say an “and gate”, n will be three because and gate has three peaks. If there is an AND gate and a NOT gate which is not connected, then n will be five. But there is a trick arises when the gates are connected. It is actually meant free-not connected- peaks by n. So if the and gate and not gate is connected via a line then n will decrease to four. “s” defines signals in file, actually it is not commonly used. “l” defines labels. This is actually a bit tricky. If label is found then our job is to connect labels as if there are no labels. Actually labels are used for readability. If l is 3 there are three lines for these three labels where labels are defined with coordinates and gate numbers. “w” gives the number of wires in circuit. After w there are coordinates of start-end peaks of wires with names which peak is connected with which peak. If w is 8 then eight lines are used for defining wires according to their connections. “b” defines boxes in DIGLOG program. If b is 1 it means there is only one box in circuit and after b 1 line, a single line comes to define this box’s coordinates. “g” defines gates. It gives the total number of gates in the circuit. This is another tricky part. If g is 3 then there are three gate definitions in nine lines, three for one gate. First line of definition is the name of gate. If it is a not gate then it writes NOT, if it is and or gate it writes OR...etc. After this, line coordinates of the gates arises and finally unique peak numbers comes to define exact peak of gates. Another interesting feature is that if two peaks are connected then this line is modified and they are both named with the name of smaller peak code. “h” defines the history part. If there are objects like clock, stall...etc in the circuit h gives their number. For example if there are three clocks then three lines are reserved to define these clocks.

Exporting of LGF files is a reverse process of importing. There are two ways to export the circuit in to a LGF file: first user can use main window menus File-Save As-and choose directory with browse window and then name the file anyfile.LGF as he/she desired. Second method is using script program. User can type “save <filename.LGF> “ on script and exporting process is started.

Exporting is handled exportLGF function which does opposite of importLGF function. According to above information it will write the circuit in to LGF files.

4.1.5_Scripting Module

JHDL provides us a library for using a command line, namely CVI (Command Line Interface). We will use it for scripting. In the background behavioral properties of our design will be the same.

We will supply an interpreter to support scripting. Except from drawing and analyzing circuits, every feature of the program can be used via interpreter. One of the useful usages is; it supports giving inputs to the circuit and getting outputs without opening main program. Interpreter works independently from the main program but it will also be able to work while the main program is running.

In our interpreter, first a user loads a file. This loaded file can be an existing circuit file, a diglog file or a text file containing a Boolean function. After loading a file, user may give inputs and get output without opening the main program. He/she may want to convert the loaded file into a diglog file or convert a loaded diglog file into our circuit file format.

Another function that our interpreter supports is printing. Again without opening the main program user can get the printout of the loaded file either directly from the printer or print the circuit into a file. Printed file types are postscript file and portable document file formats.

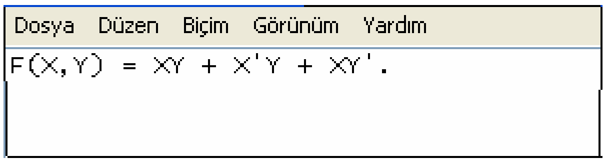
When our interpreter is run with the program, user will be able to send inputs through interpreter during circuit simulation. This feature will be very helpful during simulation running and analyzing process.

JHDL provides us a library for using a command line, namely CVI (Command Line Interface). We will use it for scripting. In the background behavioral properties of our design will be the same.

4.1.5.1_ Boolean Functions

One interesting feature of DIGIMOD is, it can convert a text file, which holds a boolean function, to a logic circuit. User can use this facility in two ways. First he/she can use file-open dialog box and choose a text file. Second is via script where user writes “load myFile.txt” and function is loaded to system. After loading process our program will create logic circuit and put it onto main interface.

This file is like:



```
F(X,Y) = XY + X'Y + XY'.
```

Syntax of the function is not very strict but it has some necessities. Function will start with F and user will declare arguments in parenthesis separated with commas. After “=”, function begins. If two parameters has no mark between each other than it means logical AND, “+” means logical OR, “ ' ” means logical NOT. “*” means logical NAND, “%” means logical NOR and “#” means logical XOR. These are very primitive logic operators and they are possible to increase with the implementation process. File can contain 1 and 0 in it. Function ends with “.” operator.

When load process is finished successfully, convertFun() function which takes a file as argument will be invoked. This function reads the file and draws the circuit. Program first determines logical gates and gives appropriate names to gates and their peaks. Variables are assigned to these peaks and likewise LGF algorithm described in import/export LGF section. 1 and 0 are automatically converted to +5V and GRND in logic circuit.

Parenthesis “(” and “)” are another important part of boolean function. It assigns priority to operations. Operators have priorities and as predictable parenthesis have the first priority in our system as in logic design. All these operators and variables are handled in a stack.

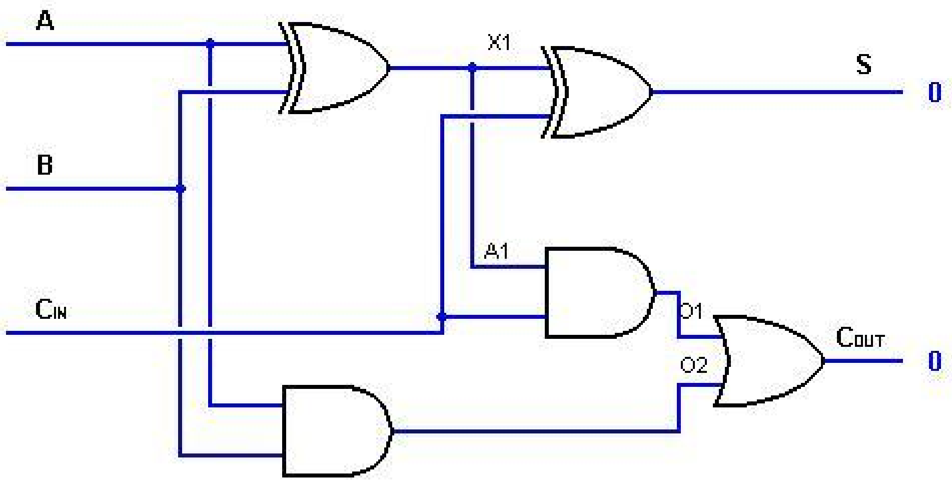
If the text file is lack of syntax necessities then converting operation is stopped by convertFun() function and an error message is raised to user. Error messages are different from each other except “Can not draw file” message. Error messages will be designed with guiding principle which helps user to correct their errors in file. For example if there is an unclosed parenthesis, which is discovered by stack operations, “Unclosed parentheses” error message will raise.

On the other hand there is a problem for this section. We assume boolean functions will be as simple as it can be. Otherwise circuit will become too complex to be “practical”. We decided to use a few quick operations like A1 = A, A+1 = A ...etc. More complex algorithms may be used in DIGIMOD but leaving this complex simplification process to user will be more applicable.

4.2 Transition from Canvas to JHDL

On the canvas as described in canvas section there is no data only graphical representation. However, in our case all the information required for creating data model can be extracted from layout of circuit elements and wire connections. All circuit elements takes wires as inputs and again give wires as outputs. Another convention is power to the circuit is supplied via switches or clocks. So our algorithm starts with inspecting power supplies in the circuit. If a gate, wire or any element does not connected to a power supply, there is noting to simulate about that component.

After inspecting power sources we need to detect which components are connected to them. At last we start from outputs and while going towards inputs we construct JHDL model. Every circuit element has a JHDL function on its inputs. When we reach an element according to its type we call its JHDL function. This function has arguments of type wire. If an input wire to an element comes from another element, recursively its parent element's function will be called to return the state of that wire which will be used as an argument in its child element's function. This process goes until we reach a power supply. For example, consider following drawing which is a full adder circuit.



There are two outputs in the circuit. Namely; *Sum* and *Cout*. Let's simulate our algorithm on them.

To determine *S* we start from that wire. Than we reach an XOR gate. To specify its output, relevant JHDL function will be called with 2 arguments which are X1 and Cin as follows;

```
xor_o(Cin, x1, s);
```

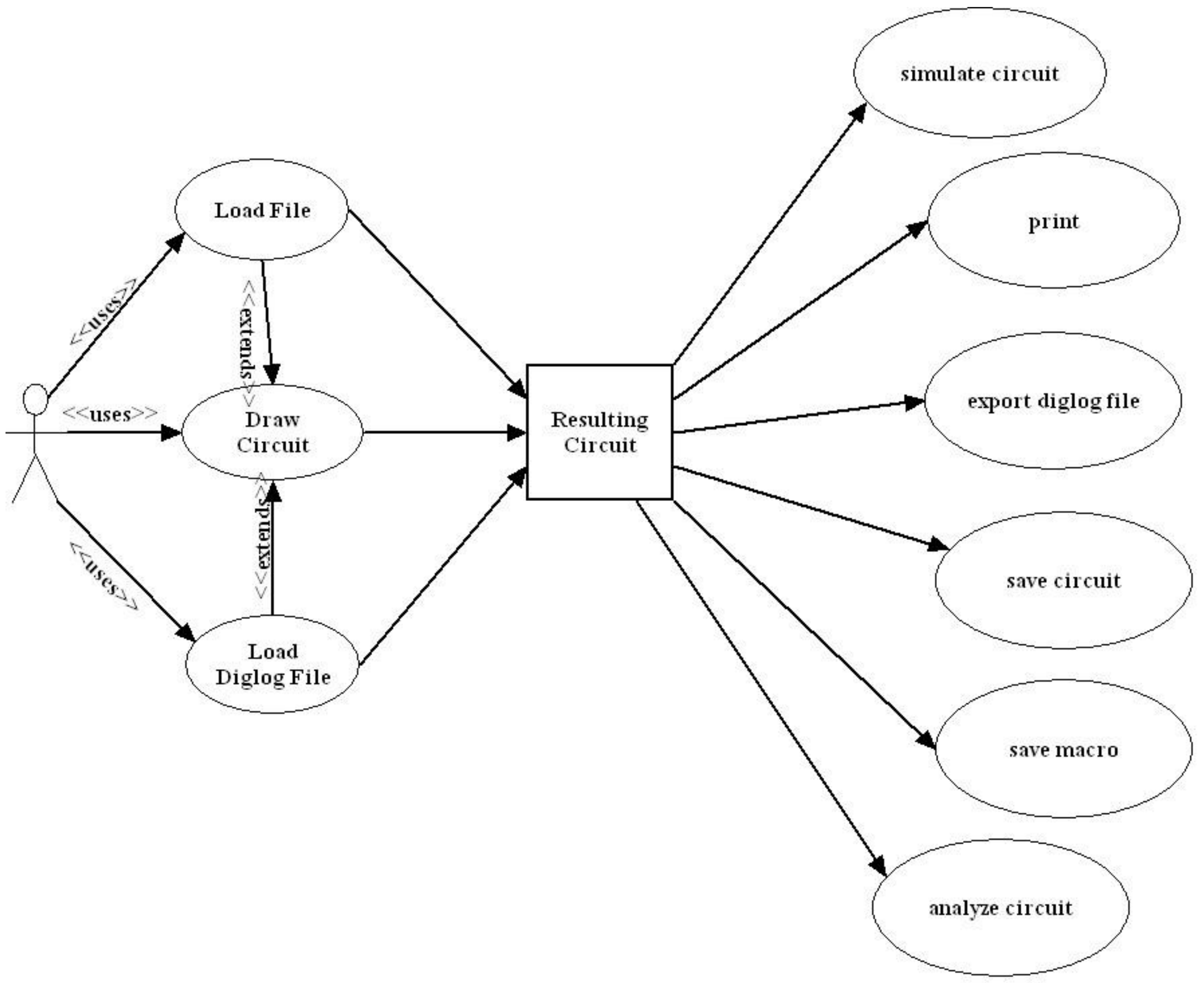
This function constructs a new 2 input XOR gate. Cin is an input to the circuit, however, we need to know about x1. To determine we go to the other XOR gate whose function is "*xor_o(a, b, x1);*" replacing *x1* with its equivalent we reach s output as follows;

xor_o(Cin, xor(a,b),s);

that function returns a wire object named "*s*". Other output can be determined similarly.

5. UNIFIED MODELLING LANGUAGE DESIGN

5.1 Program Use Case



Program Use Case Diagram

Load File

Objective	*To allow user to load a previously saved circuit file.*
Precondition	-
Main Flow	1. *The user interacts with the main window of the program.* 2. *User loads a file by using load button.* 3. *After pressing "open" button standard browse window appears.* 4. *User finds desired file and presses "open" button to finish operation.*

Draw Circuit

Objective	*To allow user to draw circuit.*
Precondition	-
Main Flow	1. *User can draw a circuit using component library, basic circuit elements or previously saved macros.* 2. *User chooses which circuit element to be used then presses related button and finally places it wherever he likes.*

Load Diglog File

Objective	*To allow user to load a previously saved .LGF file.*
Precondition	-
Main Flow	1. *The user interacts with the main window of the program.* 2. *User loads a .LGF file by using load button.* 3. *After pressing "open" button standard browse window appears.* 4. *User finds desired file and presses" open" button to finish operation.*

Simulate Circuit

Objective	*To allow user to control circuit by simulation*
Precondition	*A circuit file must be opened or drawn.*
Main Flow	1. *The user interacts with the main window of the program.* 2. *User presses "simulate" button. So logic high and logic low wires and virtual leds are shown on screen in different colors.*

Print

Objective	*To allow user to get a hard copy or printable soft copy of the current circuit.*
Precondition	*A circuit must be opened or drawn.*
Main Flow	1. *The user interacts with the main window of the program.* 2. *User presses "print" button.* 3. *After pressing print button a selection window appears.* 4. *User can select directly printing, printing to a .PS file or .PDF file.*

Export Diglog File

Objective	*To allow user to create a .LGF file from current circuit.*
Precondition	*A circuit must be opened or drawn.*
Main Flow	*1. The user interacts with the main window of the program.* *2. User presses “save as” button.* *3. After pressing button a window appears and asks the name and location of new .LGF file.*

Save Circuit

Objective	*To allow user to save current circuit in a file.*
Precondition	*A circuit must be opened or drawn.*
Main Flow	*1. The user interacts with the main window of the program.* *2. User presses “save” button.* *3. After pressing “save” button a window appears and asks the name and location of new file.*

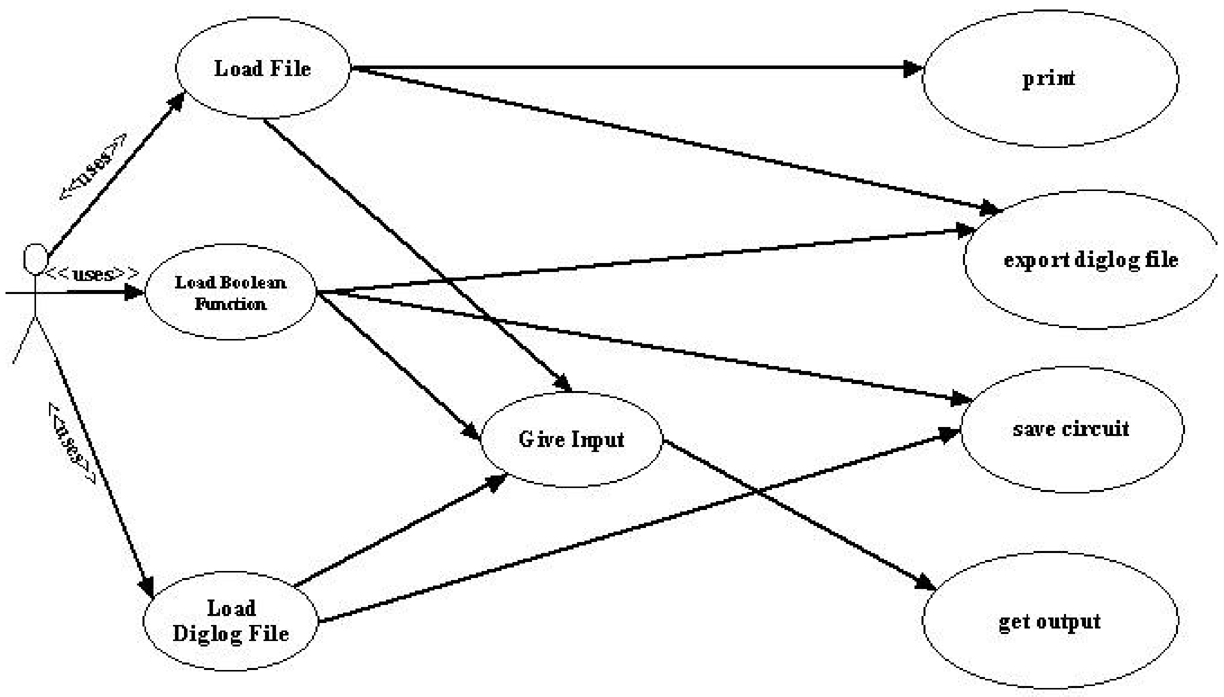
Save Macro

Objective	*To allow user to create macro from current circuit.*
Precondition	*A circuit must be opened or drawn.*
Main Flow	*1. The user interacts with the main window of the program.* *2. User presses “Save As Macro” button.* *3. After pressing the button a window appears and asks to specify input and output names and the name of the macro.*

Analyze Circuit

Objective	*To allow user to analyze a circuit.*
Precondition	*A circuit file must be opened or drawn.*
Main Flow	*1. The user interacts with the main window of the program.* *2. User presses “analyze” button.* *3. After pressing button a window appears and user drags and drops which elements that he wants to analyze.* *4. If user wants to analyze a wire, he can put an output led element and use it.*

5.2 Scripting Use Case



Scripting Use Case Diagram

Load file

.

Objective	*To allow user to load a previously saved circuit file.*
Precondition	*File that will be loaded must be in the program directory. Otherwise full path of the file must be entered.*
Main Flow	5. *The user interacts with the interpreter* 6. *User loads a file by typing <load fileName>* 7. *User is informed about success of the load operation.*

Load .LGF file

Objective	*To allow user to load a previously saved .LGF file.*
Precondition	*File that will be loaded must be in the program directory. Otherwise full path of the file must be entered.*
Main Flow	1. *The user interacts with the interpreter* 2. *User loads a .LGF file by typing <load fileName>* 3. *User informed about success of the load operation.*

Load Boolean Function

Objective	*To allow user to load a previously saved Boolean function from a text file.*
Precondition	*File that will be loaded must be in the program directory. Otherwise full path of the file must be entered.*
Main Flow	1. *The user interacts with the interpreter* 2. *User loads a file by typing <load fileName>* 3. *User informed about success of the load operation.*

Give input

Objective	*To allow user give inputs to loaded circuit.*
Precondition	*A circuit file must be loaded successfully.*
Main Flow	1. *The user interacts with the interpreter* 2. *User enters inputs according to loaded circuit.* 3. *If given inputs does not match user is warned.* 4. *After entering proper inputs, according outputs are printed to the screen.*

Printing

Objective	*To allow user printing out a loaded circuit file.*
Precondition	*A circuit file must be loaded successfully.*
Main Flow	1. *The user interacts with the interpreter* 2. *User prints a file by typing <print>* 3. *If user wants to print to file (.ps) or (.pdf), he uses <print –f> or <print –p> respectively and file name to be saved should be entered afterwards.* 4. *If user does not specify a file name file automatically saved with then name "out.ps" or "out.pdf".* 5. *User informed about success of the printing operation.*

Export Diglog File

Objective	*To allow user to create a .LGF file from current circuit.*
Precondition	*A circuit file must be loaded successfully.*
Main Flow	1. *The user interacts with the interpreter* 2. *User prints a file by typing <export fileName>* 3. *User informed about success of the exporting operation.*

Get Output

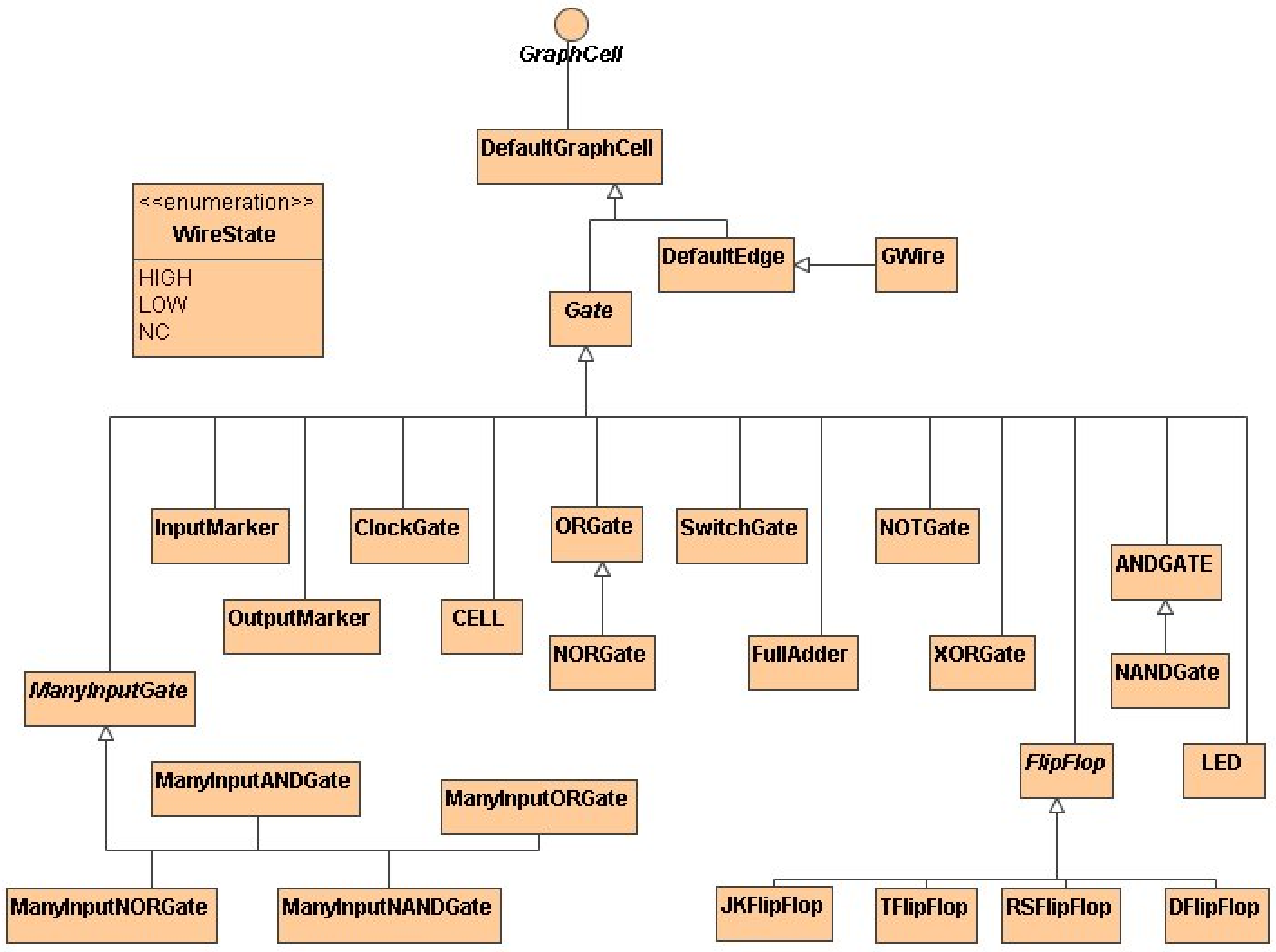
Objective	*To allow user to get output from current circuit.*
Precondition	*A circuit file must be loaded successfully.*
Main Flow	*1. User can get output of the current circuit as described in Give Input use case.*

Save Circuit

Objective	*To allow user to create circuit file from current circuit.*
Precondition	*A circuit file must be loaded or generated from Boolean function file successfully.*
Main Flow	*1. The user interacts with the interpreter* *2. User saves a file by typing <save fileName>* *3. User informed about success of the saving operation.*

5.3 Class Diagrams

5.3.1_Canvas Class Diagram



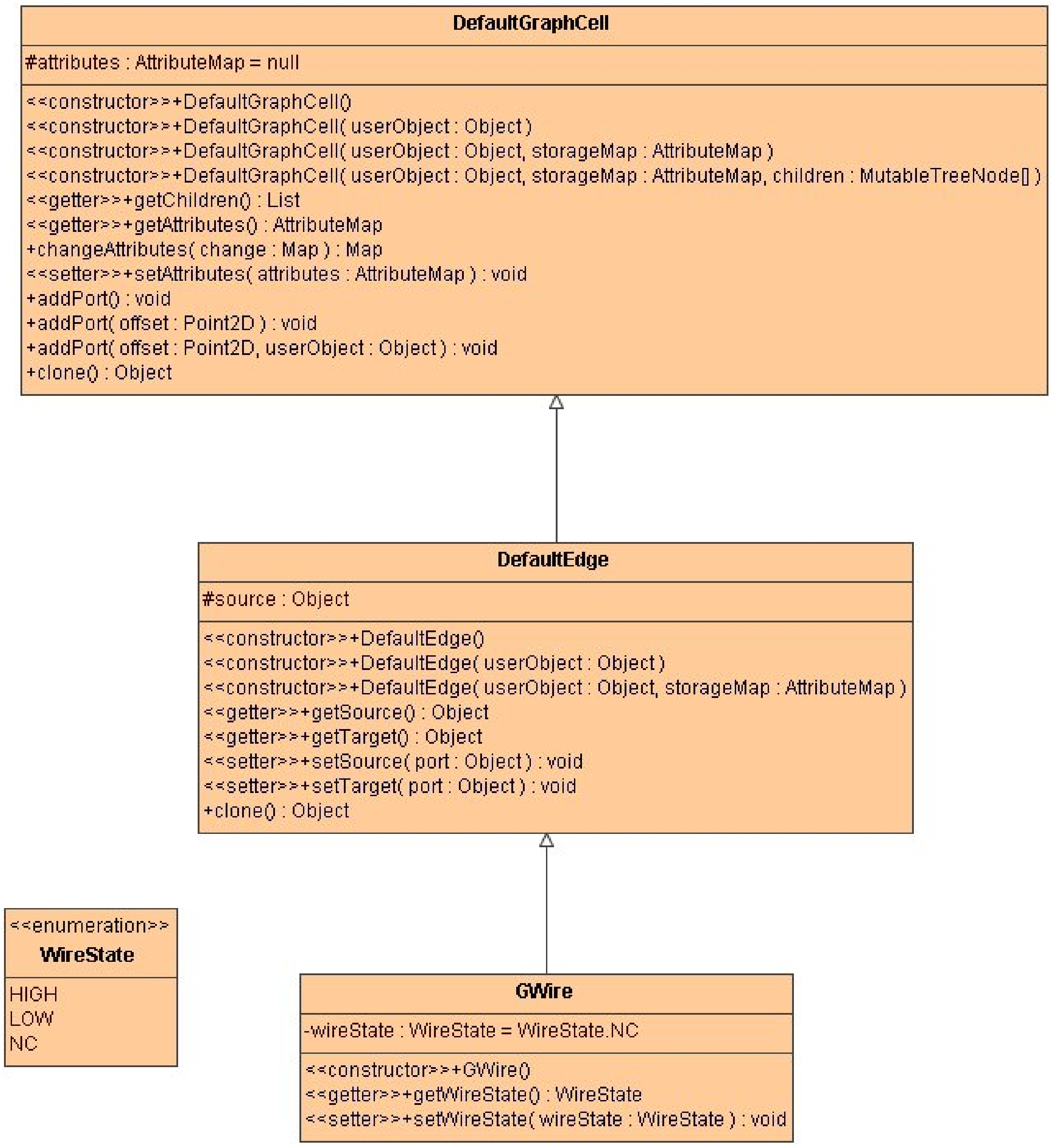
5.3.1.1_Canvas Class Diagram Description

We are using an open-source graphic library for JAVA named JGraph for graphical representation of our system. DefaultGraphCell and DefaultEdge classes come from JGraph. In our design we divide our graphical object in to two main types, namely gates and wires. Since there is a class named wire in JHDL we named our graphical wire class as GWire to avoid confliction.

GWire is inherited from DefaultEdge class of JGraph because graphically they are nothing but lines. And similarly all other components are represented as rectangular. So they are inherited from DefaultGraphCell class which defines the requirements for objects that appear as GraphCells. This is the base interface for all GraphCells. Of course clocks, LED's or switches are not logically gates. However graphically they are kept in the same way, for that reason they are all inherited from Gate class.

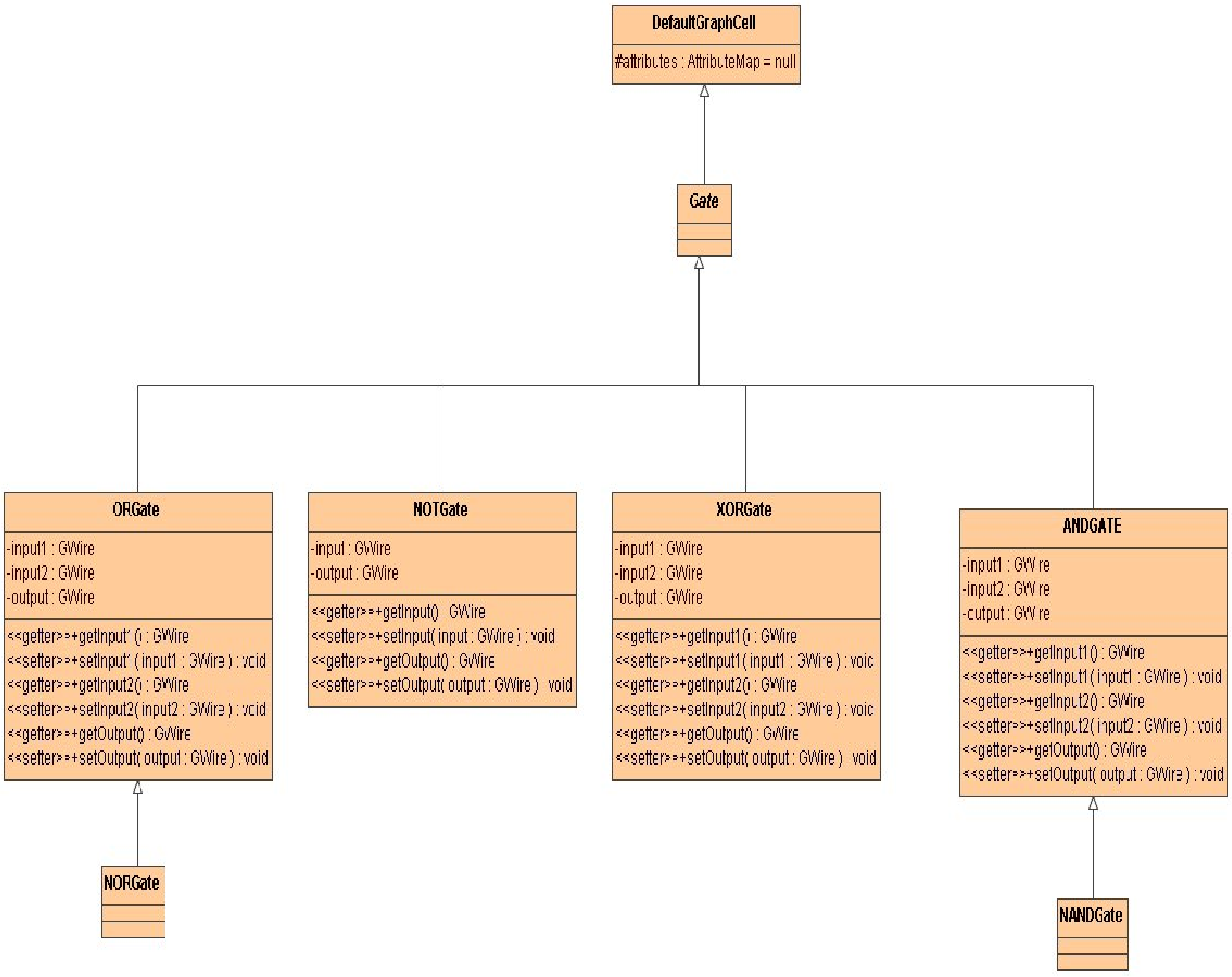
There are only class names in the above diagram because we do not want to make complicated to read it. Each class will be analyzed in the following sections individually.

5.3.1.2_GWire Class Diagram


DefaultGraphCell
#attributes : AttributeMap = null
<<constructor>>+DefaultGraphCell()
<<constructor>>+DefaultGraphCell(userObject : Object)
<<constructor>>+DefaultGraphCell(userObject : Object, storageMap : AttributeMap)
<<constructor>>+DefaultGraphCell(userObject : Object, storageMap : AttributeMap, children : MutableTreeNode[])
<<getter>>+getChildren() : List
<<getter>>+getAttributes() : AttributeMap
+changeAttributes(change : Map) : Map
<<setter>>+setAttributes(attributes : AttributeMap) : void
+addPort() : void
+addPort(offset : Point2D) : void
+addPort(offset : Point2D, userObject : Object) : void
+clone() : Object
DefaultEdge
#source : Object
<<constructor>>+DefaultEdge()
<<constructor>>+DefaultEdge(userObject : Object)
<<constructor>>+DefaultEdge(userObject : Object, storageMap : AttributeMap)
<<getter>>+getSource() : Object
<<getter>>+getTarget() : Object
<<setter>>+setSource(port : Object) : void
<<setter>>+setTarget(port : Object) : void
+clone() : Object
<<enumeration>>
WireState
HIGH
LOW
NC
GWire
-wireState : WireState = WireState.NC
<<constructor>>+GWire()
<<getter>>+getWireState() : WireState
<<setter>>+setWireState(wireState : WireState) : void

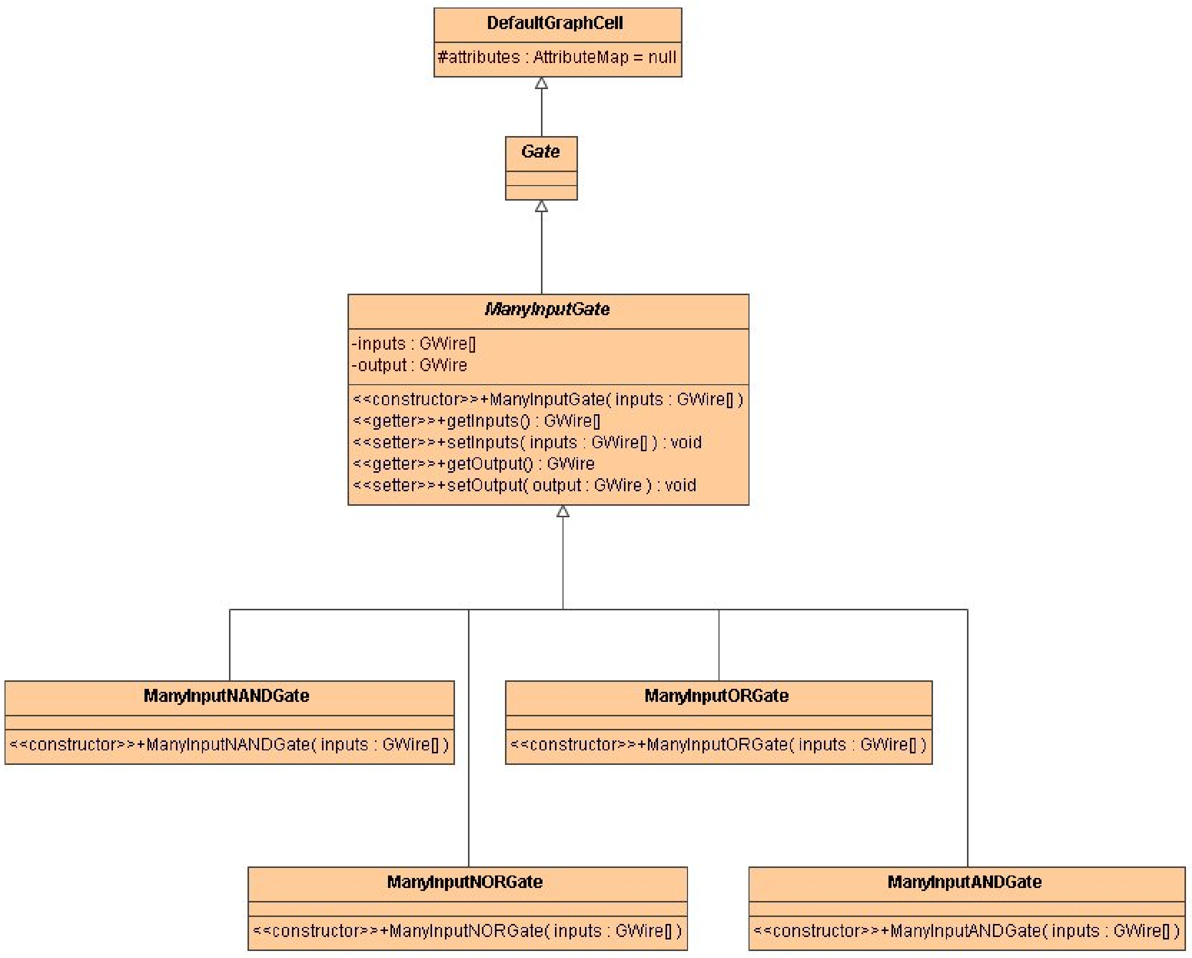
Gwire class is for the graphical representation of a wire. Objects of this class form the inputs and outputs for all gates. It is inherited from DefaultEdge class which is basically wire class of JGraph. It has a wireState member of type WireState that holds the state of the wire (high, low, not connected).

5.3.1.3_Gate Class Diagram


DefaultGraphCell
#attributes : AttributeMap = null
Gate
ORGate
-input1 : GWire
-input2 : GWire
-output : GWire
<<getter>>+getInput1() : GWire
<<setter>>+setInput1(input1 : GWire) : void
<<getter>>+getInput2() : GWire
<<setter>>+setInput2(input2 : GWire) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
NOTGate
-input : GWire
-output : GWire
<<getter>>+getInput() : GWire
<<setter>>+setInput(input : GWire) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
XORGate
-input1 : GWire
-input2 : GWire
-output : GWire
<<getter>>+getInput1() : GWire
<<setter>>+setInput1(input1 : GWire) : void
<<getter>>+getInput2() : GWire
<<setter>>+setInput2(input2 : GWire) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
ANDGATE
-input1 : GWire
-input2 : GWire
-output : GWire
<<getter>>+getInput1() : GWire
<<setter>>+setInput1(input1 : GWire) : void
<<getter>>+getInput2() : GWire
<<setter>>+setInput2(input2 : GWire) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
NORGate
NANDGate

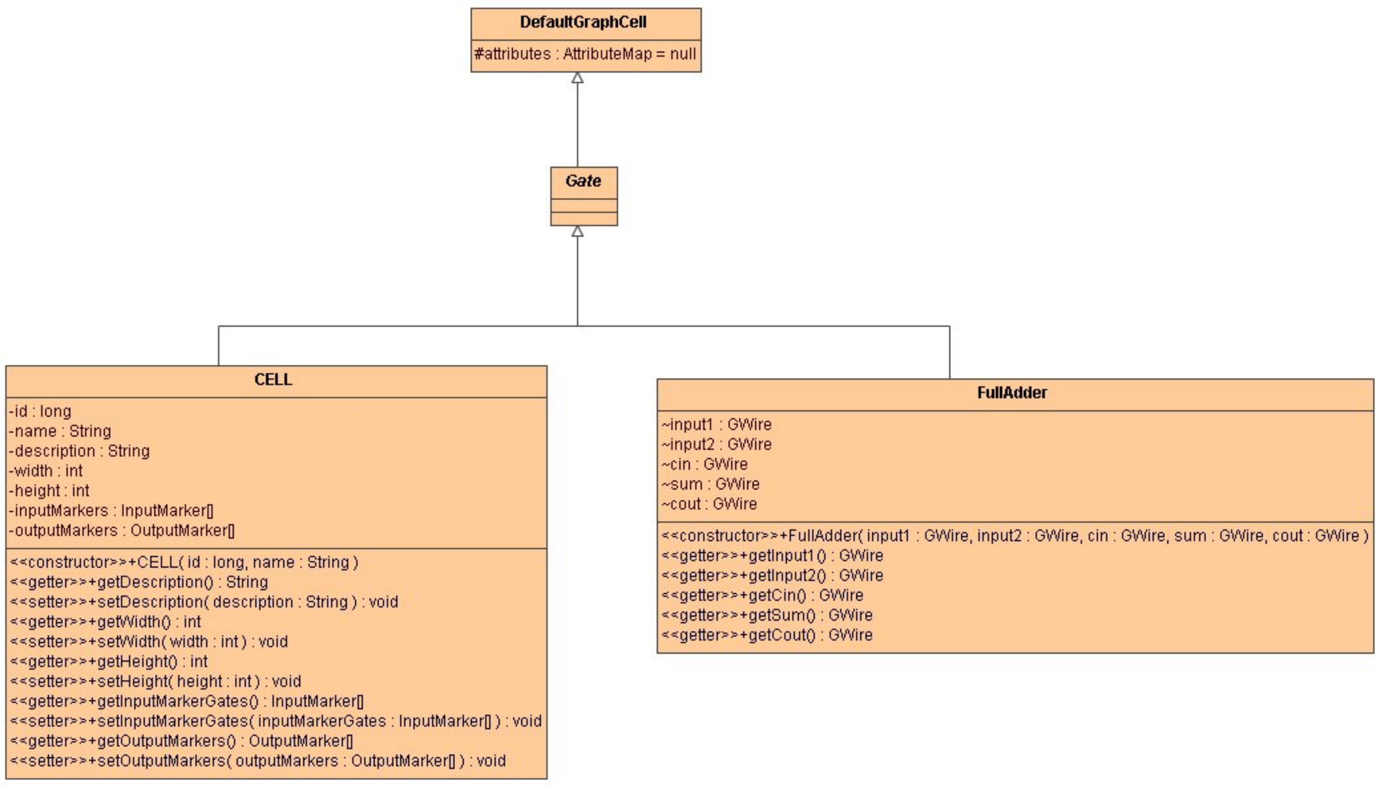
We implement all primitive gates with specifications of their interfaces which are input and outputs. They are default representations of one-input NOT and two-input OR, AND, XOR, NOR, NAND gates. Since they are widely used in FPGA’s they are implemented explicitly. Multi-input logic gates, library elements and cells are represented separately with different specifications.

5.3.1.4_ManyInputGate Class Diagram

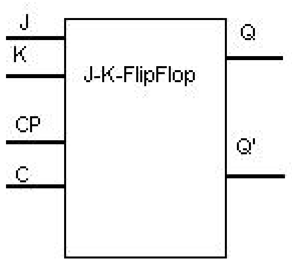

DefaultGraphCell
#attributes : AttributeMap = null
Gate
ManyInputGate
-inputs : GWire[]
-output : GWire
<<constructor>>+ManyInputGate(inputs : GWire[])
<<getter>>+getInputs() : GWire[]
<<setter>>+setInputs(inputs : GWire[]) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
ManyInputNANDGate
<<constructor>>+ManyInputNANDGate(inputs : GWire[])
ManyInputORGate
<<constructor>>+ManyInputORGate(inputs : GWire[])
ManyInputNORGate
<<constructor>>+ManyInputNORGate(inputs : GWire[])
ManyInputANDGate
<<constructor>>+ManyInputANDGate(inputs : GWire[])

ManyInputGate is the base class for gates with three or more inputs like 3-input “and” gate or 4-input “or” gate. It has a member called “inputs” which is type of GWire list and it holds the list of inputs that comes to the gate. Class has get and set input/output functions like other gate classes.

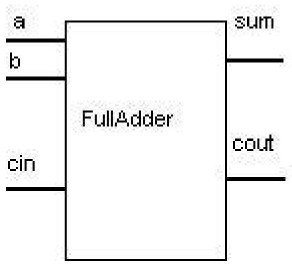
5.3.1.5_CELL Class Diagram


DefaultGraphCell
#attributes : AttributeMap = null
Gate
CELL
-id : long
-name : String
-description : String
-width : int
-height : int
-inputMarkers : InputMarker[]
-outputMarkers : OutputMarker[]
<<constructor>>+CELL(id : long, name : String)
<<getter>>+getDescription() : String
<<setter>>+setDescription(description : String) : void
<<getter>>+getWidth() : int
<<setter>>+setWidth(width : int) : void
<<getter>>+getHeight() : int
<<setter>>+setHeight(height : int) : void
<<getter>>+getInputMarkerGates() : InputMarker[]
<<setter>>+setInputMarkerGates(inputMarkerGates : InputMarker[]) : void
<<getter>>+getOutputMarkers() : OutputMarker[]
<<setter>>+setOutputMarkers(outputMarkers : OutputMarker[]) : void
FullAdder
~input1 : GWire
~input2 : GWire
~cin : GWire
~sum : GWire
~cout : GWire
<<constructor>>+FullAdder(input1 : GWire, input2 : GWire, cin : GWire, sum : GWire, cout : GWire)
<<getter>>+getInput1() : GWire
<<getter>>+getInput2() : GWire
<<getter>>+getCin() : GWire
<<getter>>+getSum() : GWire
<<getter>>+getCout() : GWire

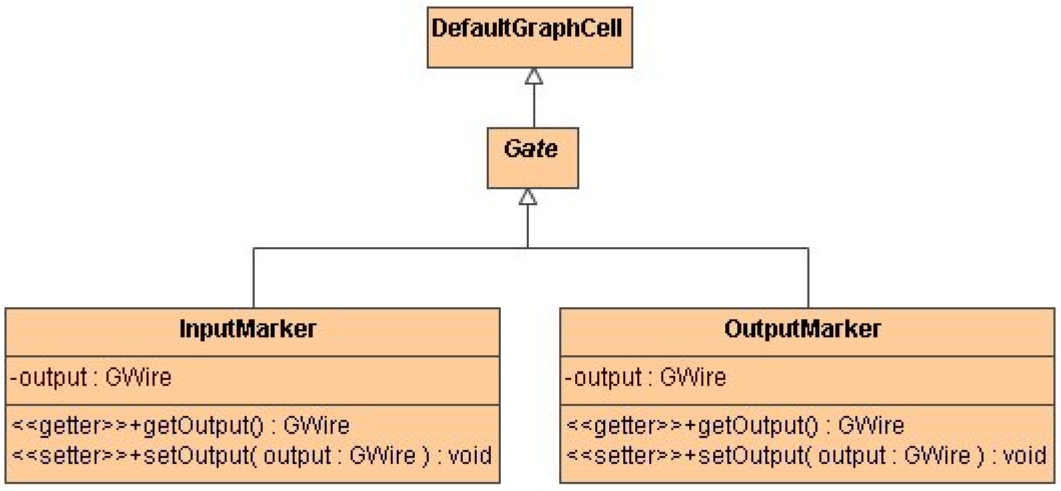
User defined macros are kept as cells. Cells are different from normal files. If user wants to draw a Cell, he/she asked to specify its name, description, input markers and output markers. According to number of inputs and outputs its symbol will be determined. Although FlipFlops are implemented explicitly, user may want to create a new cell such as JKFlipFlop with direct clear; we must define 4 inputs with names (“j”, “k”, “CP” for clock and “C” for clear), and 2 outputs with their names (“q” and “q`”). These are used as interfaces of the cell and wire objects will be connected to these pins. A primitive representation of the JKFlipFlop cell will look like below image.



In addition to user defined macros, our program will supply a library. Library components will be represented similar to user defined macros. However their symbols and input & output attributes will be defined previously. To give an example we put FullAdder class into class diagram. Although we haven’t decided on their symbols, its graphical representation will look like below image.

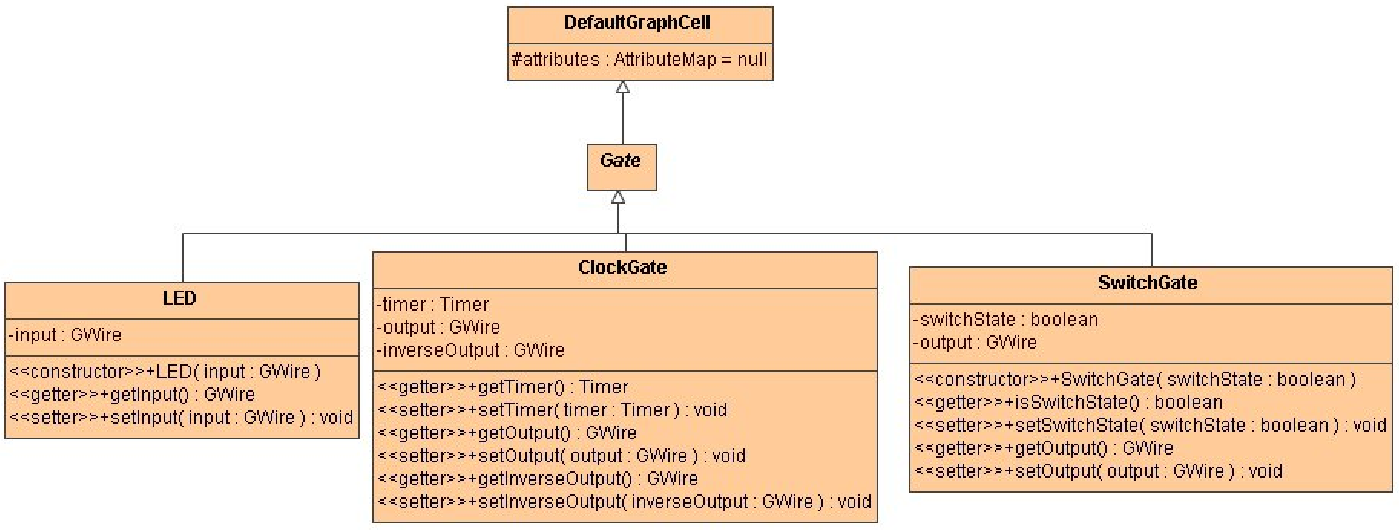


5.3.1.6_Markers Class Diagram


DefaultGraphCell
Gate
InputMarker
-output : GWire
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
OutputMarker
-output : GWire
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void

InputMarker and OutputMarker objects are used to describe a cells interface as described in CELL Class.

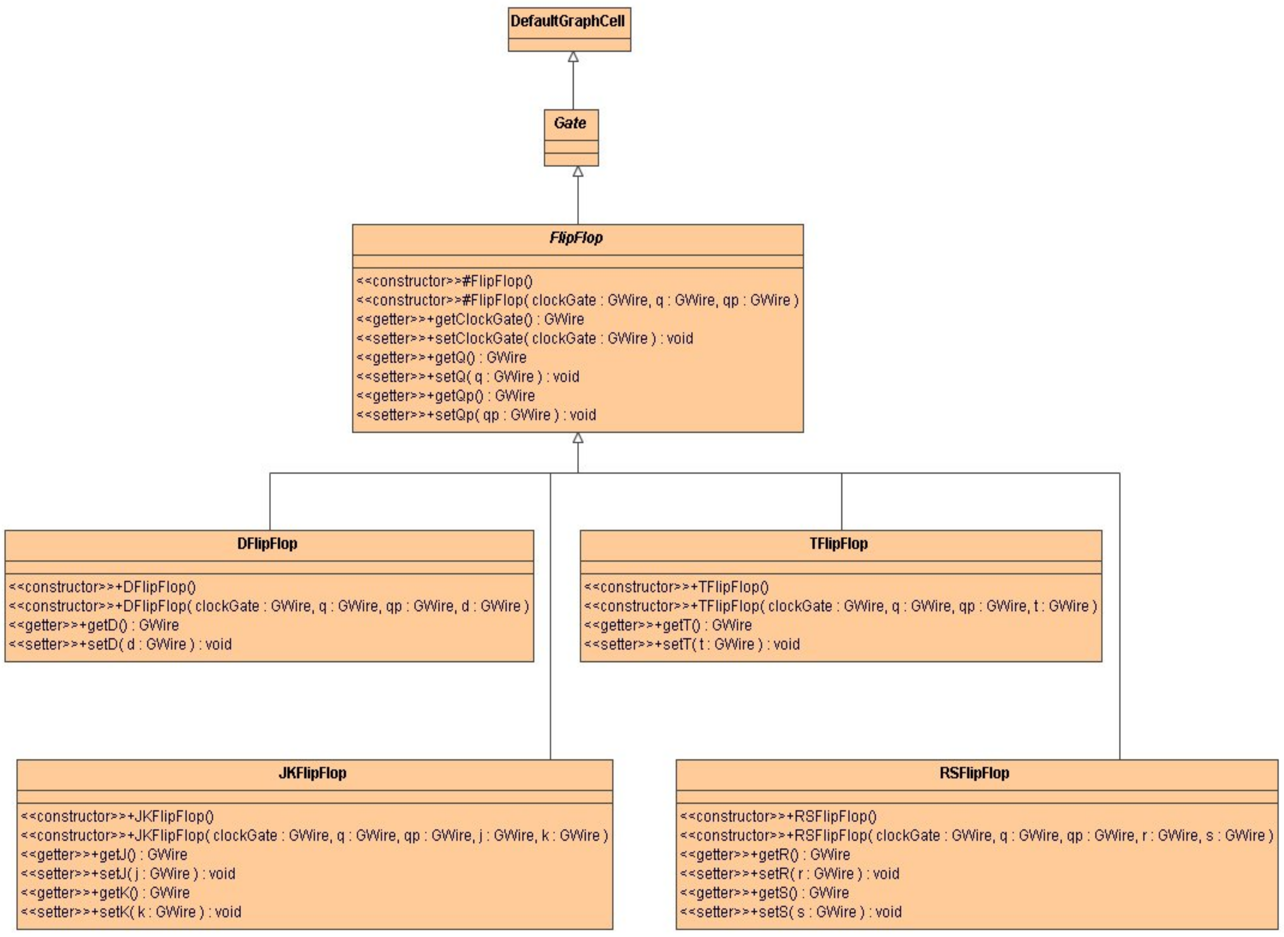
5.3.1.7_Clock, Switch, LED Class Diagrams



DefaultGraphCell
#attributes : AttributeMap = null
Gate
LED
-input : GWire
<<constructor>>+LED(input : GWire)
<<getter>>+getInput() : GWire
<<setter>>+setInput(input : GWire) : void
ClockGate
-timer : Timer
-output : GWire
-inverseOutput : GWire
<<getter>>+getTimer() : Timer
<<setter>>+setTimer(timer : Timer) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void
<<getter>>+getInverseOutput() : GWire
<<setter>>+setInverseOutput(inverseOutput : GWire) : void
SwitchGate
-switchState : boolean
-output : GWire
<<constructor>>+SwitchGate(switchState : boolean)
<<getter>>+isSwitchState() : boolean
<<setter>>+setSwitchState(switchState : boolean) : void
<<getter>>+getOutput() : GWire
<<setter>>+setOutput(output : GWire) : void

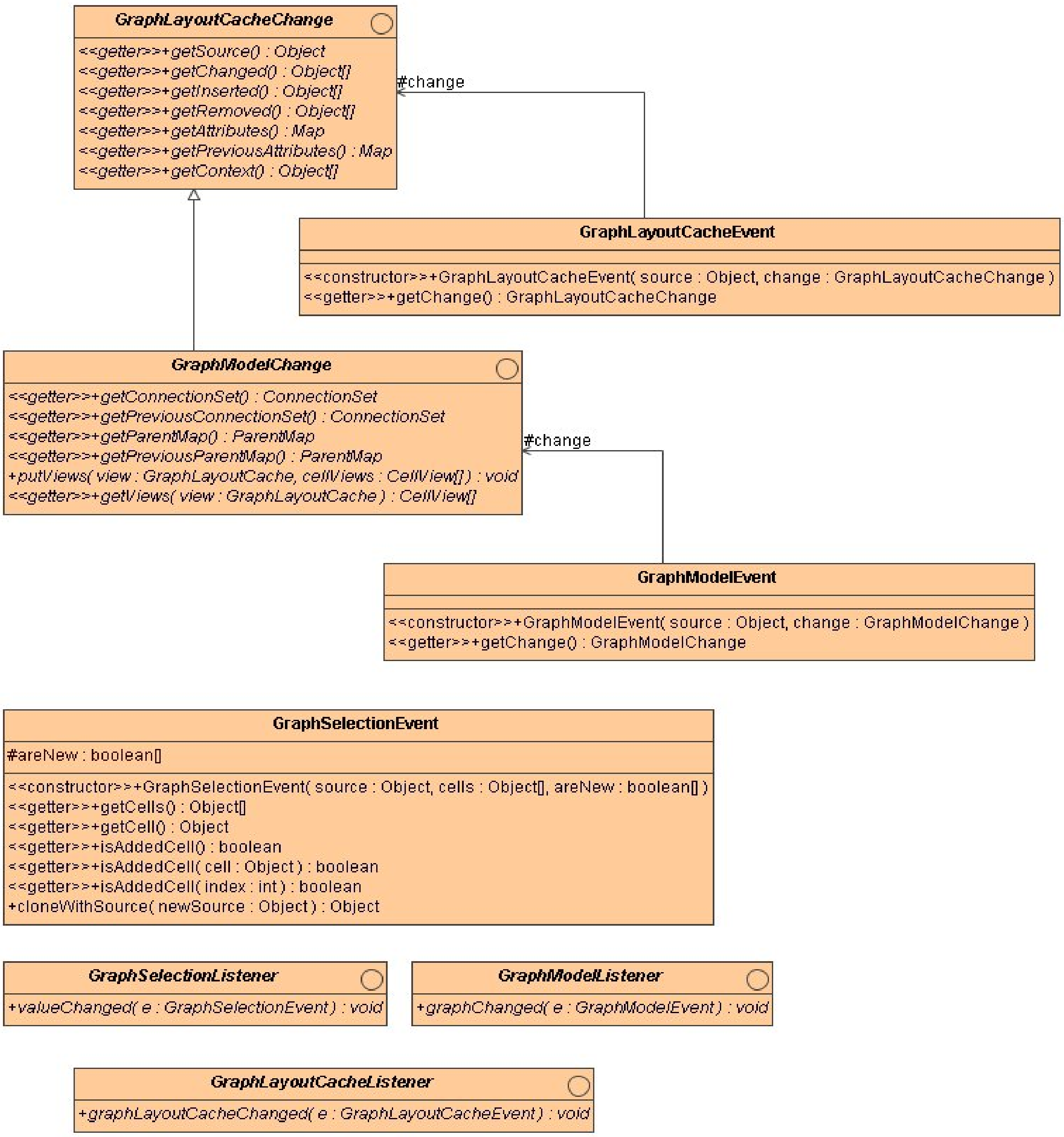
These three components are also inherited from Gate class, because their graphical representations will be done similarly. Switch will supply inputs to the circuit and LED's are used for showing outputs of the circuit. Clock is a specialized version of the Switch. Such that it produces an oscillating input signal and its frequency is determined by timer.

5.3.1.8_FlipFlop Class Diagrams


DefaultGraphCell
Gate
FlipFlop
<<constructor>>#FlipFlop()
<<constructor>>#FlipFlop(clockGate : GWire, q : GWire, qp : GWire)
<<getter>>+getClockGate() : GWire
<<setter>>+setClockGate(clockGate : GWire) : void
<<getter>>+getQ() : GWire
<<setter>>+setQ(q : GWire) : void
<<getter>>+getQp() : GWire
<<setter>>+setQp(qp : GWire) : void
DFlipFlop
<<constructor>>+DFlipFlop()
<<constructor>>+DFlipFlop(clockGate : GWire, q : GWire, qp : GWire, d : GWire)
<<getter>>+getD() : GWire
<<setter>>+setD(d : GWire) : void
TFlipFlop
<<constructor>>+TFlipFlop()
<<constructor>>+TFlipFlop(clockGate : GWire, q : GWire, qp : GWire, t : GWire)
<<getter>>+getT() : GWire
<<setter>>+setT(t : GWire) : void
JKFlipFlop
<<constructor>>+JKFlipFlop()
<<constructor>>+JKFlipFlop(clockGate : GWire, q : GWire, qp : GWire, j : GWire, k : GWire)
<<getter>>+getJ() : GWire
<<setter>>+setJ(j : GWire) : void
<<getter>>+getK() : GWire
<<setter>>+setK(k : GWire) : void
RSFlipFlop
<<constructor>>+RSFlipFlop()
<<constructor>>+RSFlipFlop(clockGate : GWire, q : GWire, qp : GWire, r : GWire, s : GWire)
<<getter>>+getR() : GWire
<<setter>>+setR(r : GWire) : void
<<getter>>+getS() : GWire
<<setter>>+setS(s : GWire) : void

Four FlipFlops are kept individually in different classes their input output properties specified in their constructors.

5.3.1.9_Canvas Event Control Class Diagram


GraphLayoutCacheChange
<<getter>>+getSource() : Object
<<getter>>+getChanged() : Object[]
<<getter>>+getInserted() : Object[]
<<getter>>+getRemoved() : Object[]
<<getter>>+getAttributes() : Map
<<getter>>+getPreviousAttributes() : Map
<<getter>>+getContext() : Object[]
#change
GraphLayoutCacheEvent
<<constructor>>+GraphLayoutCacheEvent(source : Object, change : GraphLayoutCacheChange)
<<getter>>+getChange() : GraphLayoutCacheChange
GraphModelChange
<<getter>>+getConnectionSet() : ConnectionSet
<<getter>>+getPreviousConnectionSet() : ConnectionSet
<<getter>>+getParentMap() : ParentMap
<<getter>>+getPreviousParentMap() : ParentMap
+putViews(view : GraphLayoutCache, cellViews : CellView[]) : void
<<getter>>+getViews(view : GraphLayoutCache) : CellView[]
#change
GraphModelEvent
<<constructor>>+GraphModelEvent(source : Object, change : GraphModelChange)
<<getter>>+getChange() : GraphModelChange
GraphSelectionEvent
#areNew : boolean[]
<<constructor>>+GraphSelectionEvent(source : Object, cells : Object[], areNew : boolean[])
<<getter>>+getCells() : Object[]
<<getter>>+getCell() : Object
<<getter>>+isAddedCell() : boolean
<<getter>>+isAddedCell(cell : Object) : boolean
<<getter>>+isAddedCell(index : int) : boolean
+cloneWithSource(newSource : Object) : Object
GraphSelectionListener
+valueChanged(e : GraphSelectionEvent) : void
GraphModelListener
+graphChanged(e : GraphModelEvent) : void
GraphLayoutCacheListener
+graphLayoutCacheChanged(e : GraphLayoutCacheEvent) : void

These Classes used for controlling and sensing modifications that made on the canvas.

GraphLayoutCacheChange class defines the interface for objects that may be used to represent a change to the graph layout cache.

GraphLayoutCacheEvent class encapsulates information describing changes to a graph layout cache, and is used to notify graph layout cache listeners of the change. Note that graph layout cache events do not repeat information in graph model events if there is no view specific information. The idea of this event is to provide information on what has changed in the graph layout cache only.

GraphModelChange class defines the interface for objects that may be included into a GraphModelEvent to describe a model change.

GraphModelEvent encapsulates information describing changes to a graph model, and is used to notify graph model listeners of the change.

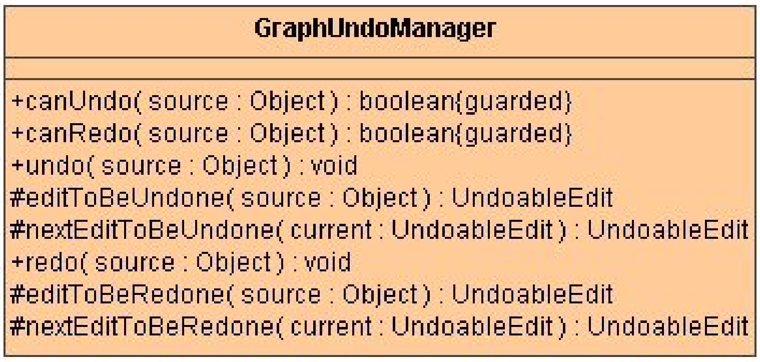
GraphSelectionEvent is an event that characterizes a change in the current selection. The change is based on any number of cells. GraphSelectionListeners will generally query the source of the event for the new selected status of each potentially changed cell.

GraphSelectionListener is the listener that's notified when the selection in a GraphSelectionModel changes.

GraphModelListener defines the interface for an object that listens to changes in a GraphModel.

GraphLayoutCacheListener defines the interface for an object that listens to changes in a GraphModel.

5.3.1.10_Undo&Redo Class Diagram



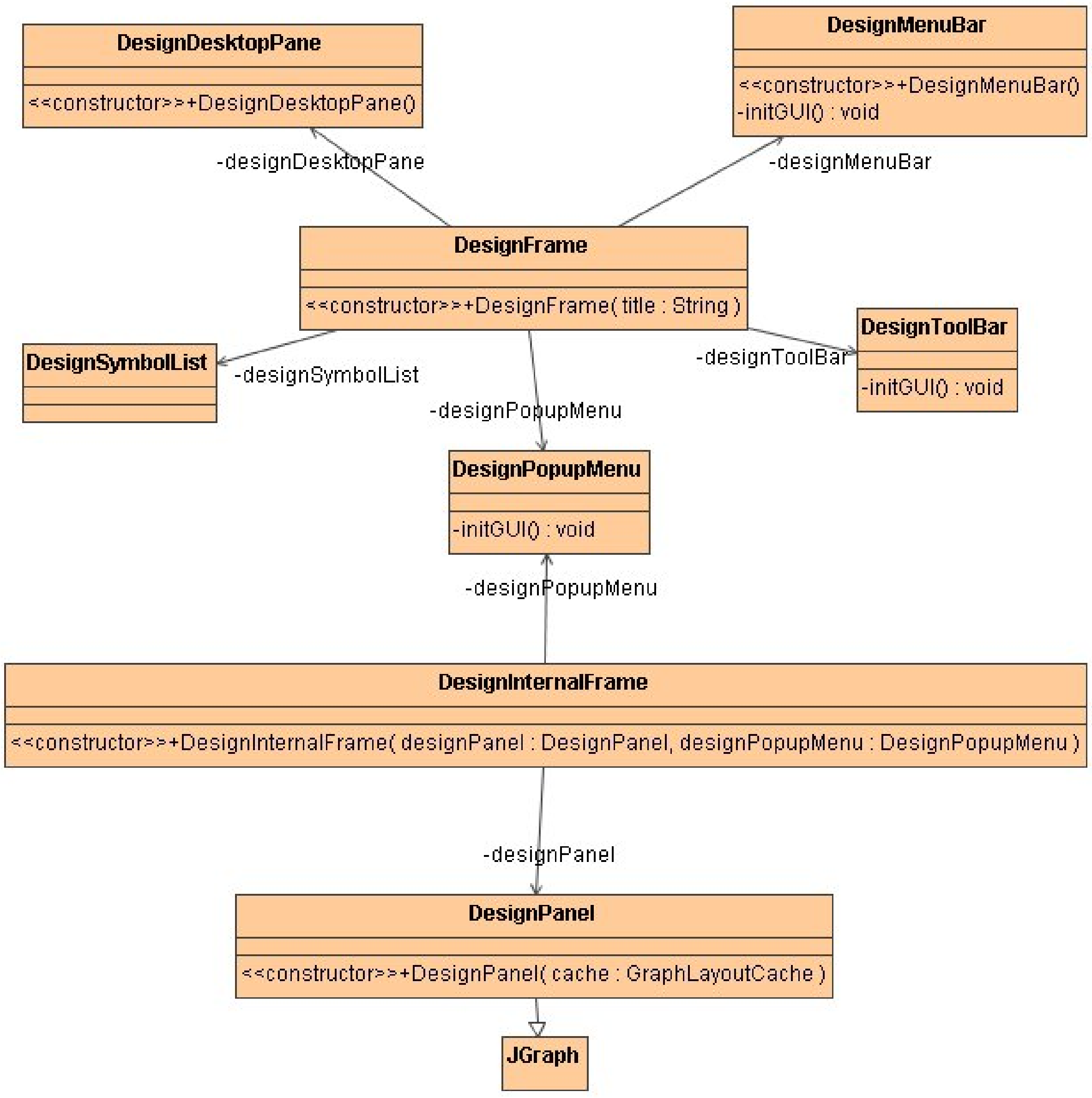
```
GraphUndoManager

+canUndo( source : Object ) : boolean{guarded}
+canRedo( source : Object ) : boolean{guarded}
+undo( source : Object ) : void
#editToBeUndone( source : Object ) : UndoableEdit
#nextEditToBeUndone( current : UndoableEdit ) : UndoableEdit
+redo( source : Object ) : void
#editToBeRedone( source : Object ) : UndoableEdit
#nextEditToBeRedone( current : UndoableEdit ) : UndoableEdit
```

This class is an undo manager that may be shared among multiple GraphLayoutCache's. This class is extended from Javax.Swing.Undo.UndoManager Class. We will use that class for managing undo & redo operations in or system.

To enable Undo-Support, a GraphUndoManager has been added using addGraphSelectionListener. The GraphUndoManager is an extension of Swing's GraphUndoManager that maintains a command history in the context of multiple views. In this setup, a cell may have a set of attributes in each view attached to the model.

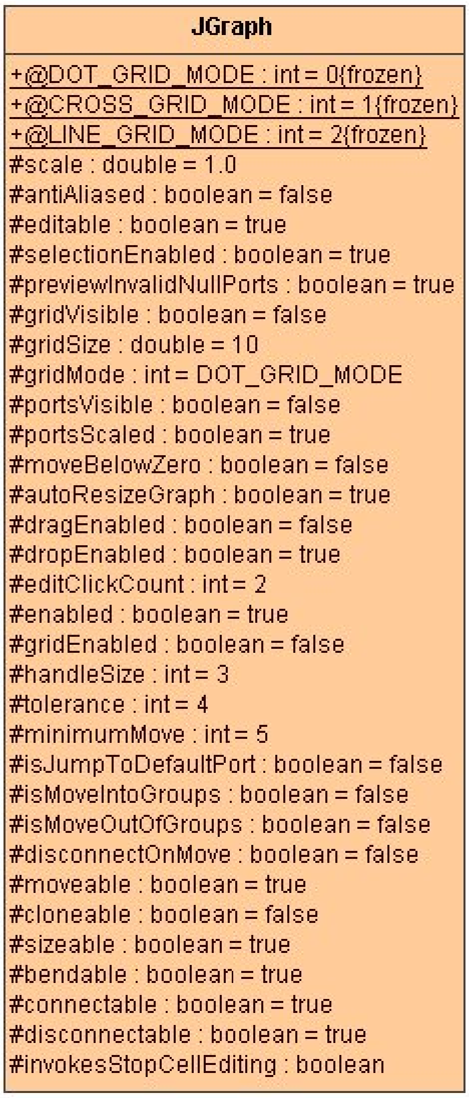
For example, consider a position that is stored separately in each view. If a node is inserted, the change will be visible in all attached views, resulting in a new node that pops-up at the initial position. If the node is subsequently moved, say, in view1, this does not constitute a change in view2. If view2 does an "undo", the move and the insertion must be undone, whereas an "undo" in view1 will only undo the previous move operation.

5.3.2_GUI Class Diagram

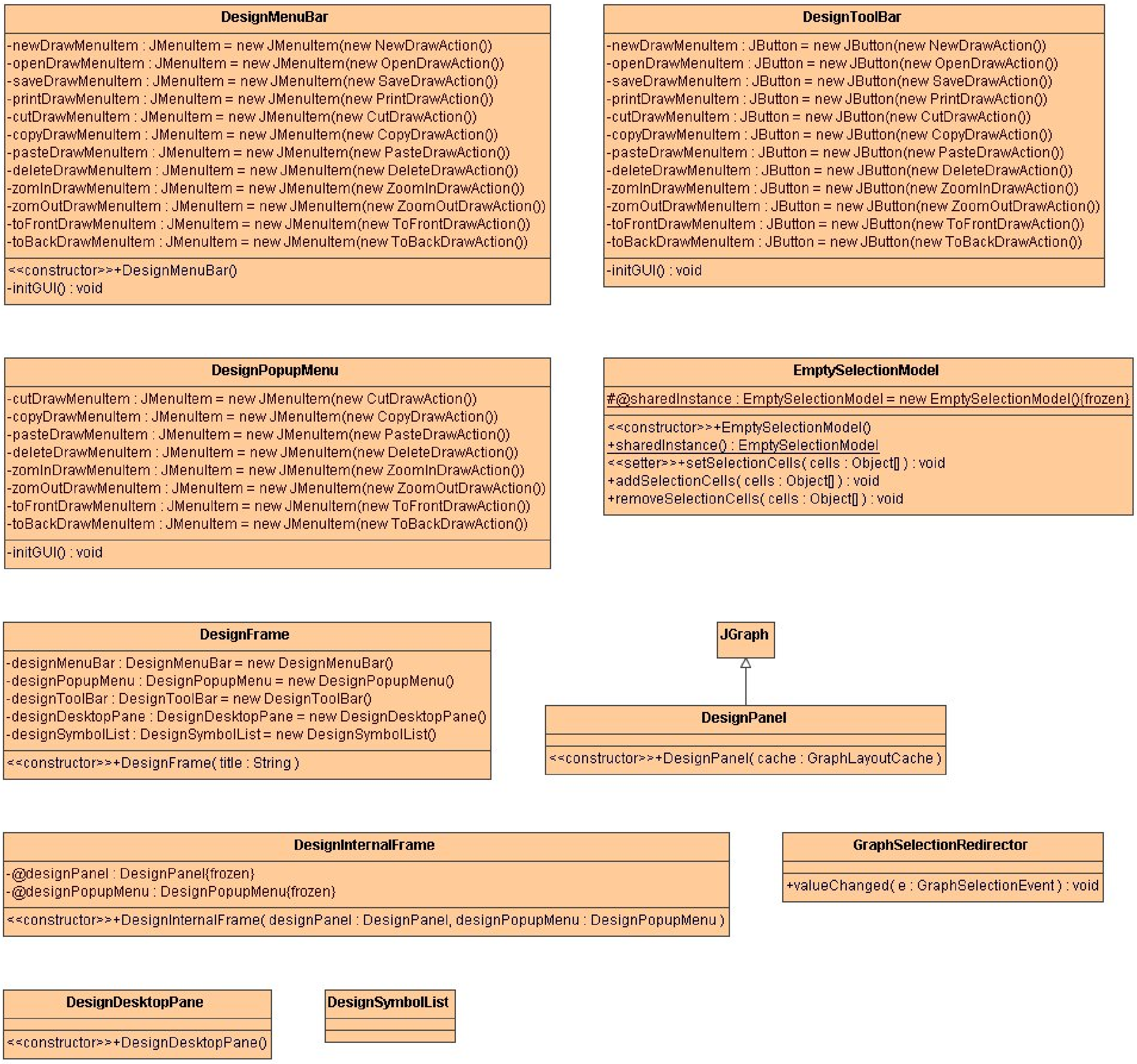


These Classes used for constructing Graphical User Interface of our program. Let’s have a look to their members.

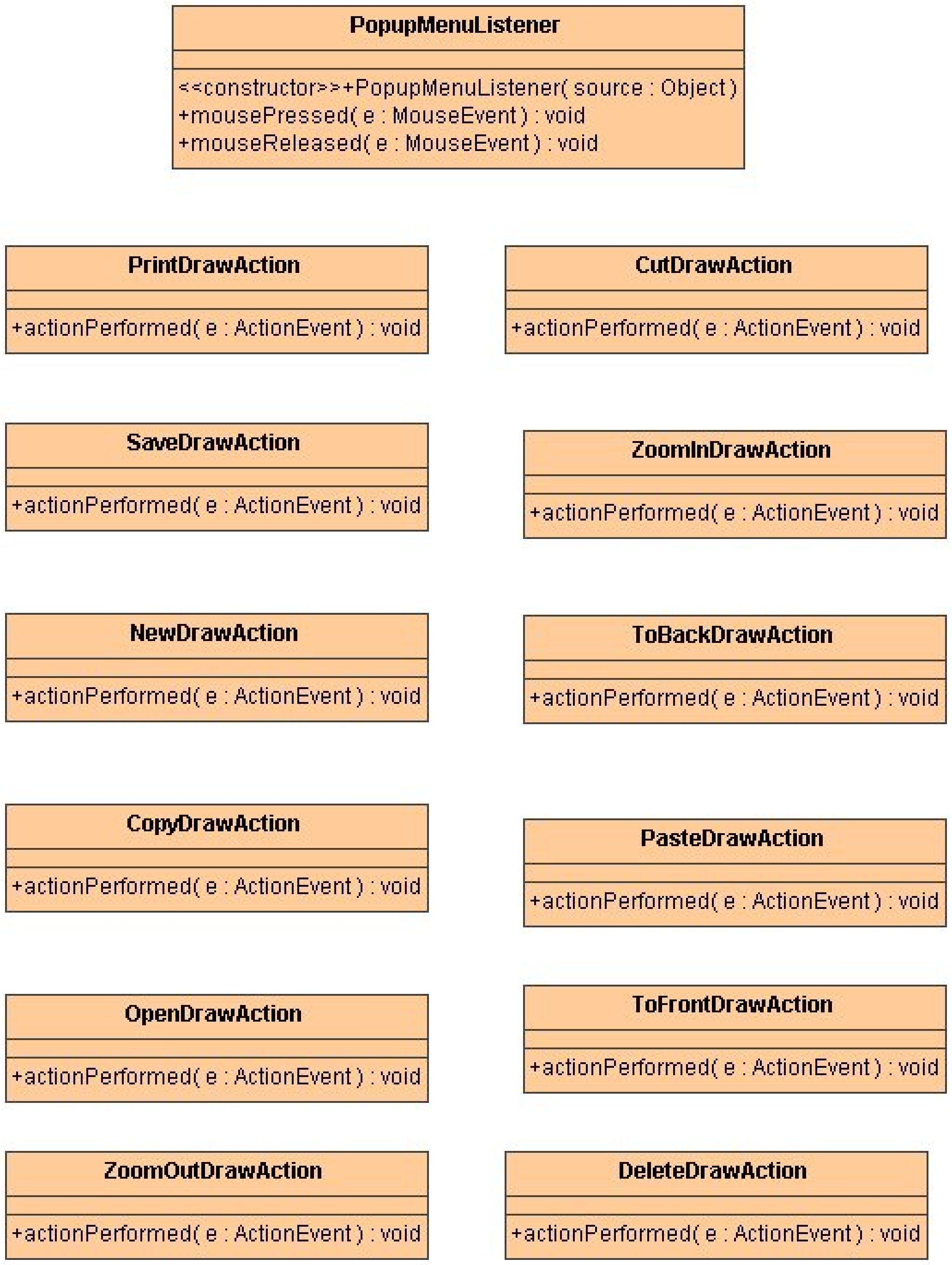
5.3.2.1_ JGraph Class Diagram


JGraph
+@DOT_GRID_MODE : int = 0{frozen}
+@CROSS_GRID_MODE : int = 1{frozen}
+@LINE_GRID_MODE : int = 2{frozen}
#scale : double = 1.0
#antiAliased : boolean = false
#editable : boolean = true
#selectionEnabled : boolean = true
#previewInvalidNullPorts : boolean = true
#gridVisible : boolean = false
#gridSize : double = 10
#gridMode : int = DOT_GRID_MODE
#portsVisible : boolean = false
#portsScaled : boolean = true
#moveBelowZero : boolean = false
#autoResizeGraph : boolean = true
#dragEnabled : boolean = false
#dropEnabled : boolean = true
#editClickCount : int = 2
#enabled : boolean = true
#gridEnabled : boolean = false
#handleSize : int = 3
#tolerance : int = 4
#minimumMove : int = 5
#isJumpToDefaultPort : boolean = false
#isMoveIntoGroups : boolean = false
#isMoveOutOfGroups : boolean = false
#disconnectOnMove : boolean = false
#moveable : boolean = true
#cloneable : boolean = false
#sizeable : boolean = true
#bendable : boolean = true
#connectable : boolean = true
#disconnectable : boolean = true
#invokesStopCellEditing : boolean

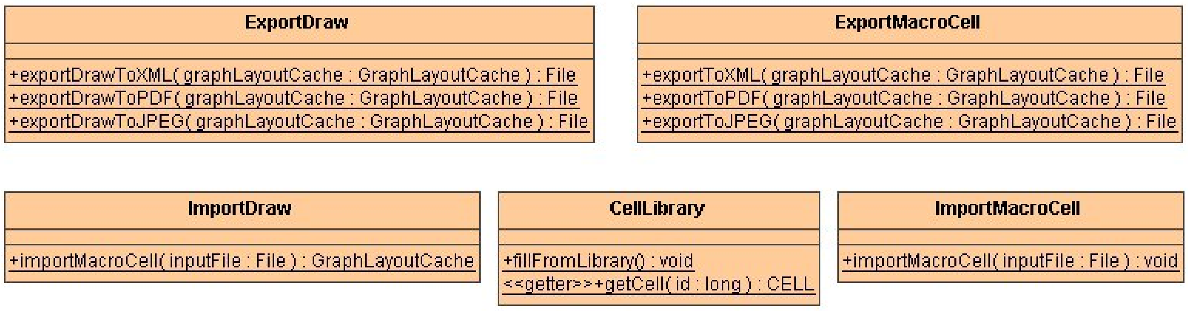
5.3.2.2_ GUI Class Diagrams


```
DesignMenuBar
-newDrawMenuItem : JMenuItem = new JMenuItem(new NewDrawAction())
-openDrawMenuItem : JMenuItem = new JMenuItem(new OpenDrawAction())
-saveDrawMenuItem : JMenuItem = new JMenuItem(new SaveDrawAction())
-printDrawMenuItem : JMenuItem = new JMenuItem(new PrintDrawAction())
-cutDrawMenuItem : JMenuItem = new JMenuItem(new CutDrawAction())
-copyDrawMenuItem : JMenuItem = new JMenuItem(new CopyDrawAction())
-pasteDrawMenuItem : JMenuItem = new JMenuItem(new PasteDrawAction())
-deleteDrawMenuItem : JMenuItem = new JMenuItem(new DeleteDrawAction())
-zomInDrawMenuItem : JMenuItem = new JMenuItem(new ZoomInDrawAction())
-zomOutDrawMenuItem : JMenuItem = new JMenuItem(new ZoomOutDrawAction())
-toFrontDrawMenuItem : JMenuItem = new JMenuItem(new ToFrontDrawAction())
-toBackDrawMenuItem : JMenuItem = new JMenuItem(new ToBackDrawAction())
<<constructor>>+DesignMenuBar()
-initGUI() : void

DesignToolBar
-newDrawMenuItem : JButton = new JButton(new NewDrawAction())
-openDrawMenuItem : JButton = new JButton(new OpenDrawAction())
-saveDrawMenuItem : JButton = new JButton(new SaveDrawAction())
-printDrawMenuItem : JButton = new JButton(new PrintDrawAction())
-cutDrawMenuItem : JButton = new JButton(new CutDrawAction())
-copyDrawMenuItem : JButton = new JButton(new CopyDrawAction())
-pasteDrawMenuItem : JButton = new JButton(new PasteDrawAction())
-deleteDrawMenuItem : JButton = new JButton(new DeleteDrawAction())
-zomInDrawMenuItem : JButton = new JButton(new ZoomInDrawAction())
-zomOutDrawMenuItem : JButton = new JButton(new ZoomOutDrawAction())
-toFrontDrawMenuItem : JButton = new JButton(new ToFrontDrawAction())
-toBackDrawMenuItem : JButton = new JButton(new ToBackDrawAction())
-initGUI() : void

DesignPopupMenu
-cutDrawMenuItem : JMenuItem = new JMenuItem(new CutDrawAction())
-copyDrawMenuItem : JMenuItem = new JMenuItem(new CopyDrawAction())
-pasteDrawMenuItem : JMenuItem = new JMenuItem(new PasteDrawAction())
-deleteDrawMenuItem : JMenuItem = new JMenuItem(new DeleteDrawAction())
-zomInDrawMenuItem : JMenuItem = new JMenuItem(new ZoomInDrawAction())
-zomOutDrawMenuItem : JMenuItem = new JMenuItem(new ZoomOutDrawAction())
-toFrontDrawMenuItem : JMenuItem = new JMenuItem(new ToFrontDrawAction())
-toBackDrawMenuItem : JMenuItem = new JMenuItem(new ToBackDrawAction())
-initGUI() : void

EmptySelectionModel
#@sharedInstance : EmptySelectionModel = new EmptySelectionModel(){frozen}
<<constructor>>+EmptySelectionModel()
+sharedInstance() : EmptySelectionModel
<<setter>>+setSelectionCells( cells : Object[] ) : void
+addSelectionCells( cells : Object[] ) : void
+removeSelectionCells( cells : Object[] ) : void

DesignFrame
-designMenuBar : DesignMenuBar = new DesignMenuBar()
-designPopupMenu : DesignPopupMenu = new DesignPopupMenu()
-designToolBar : DesignToolBar = new DesignToolBar()
-designDesktopPane : DesignDesktopPane = new DesignDesktopPane()
-designSymbolList : DesignSymbolList = new DesignSymbolList()
<<constructor>>+DesignFrame( title : String )

JGraph

DesignPanel
<<constructor>>+DesignPanel( cache : GraphLayoutCache )

DesignInternalFrame
-@designPanel : DesignPanel{frozen}
-@designPopupMenu : DesignPopupMenu{frozen}
<<constructor>>+DesignInternalFrame( designPanel : DesignPanel, designPopupMenu : DesignPopupMenu )

GraphSelectionRedirector
+valueChanged( e : GraphSelectionEvent ) : void

DesignDesktopPane
<<constructor>>+DesignDesktopPane()

DesignSymbolList
```

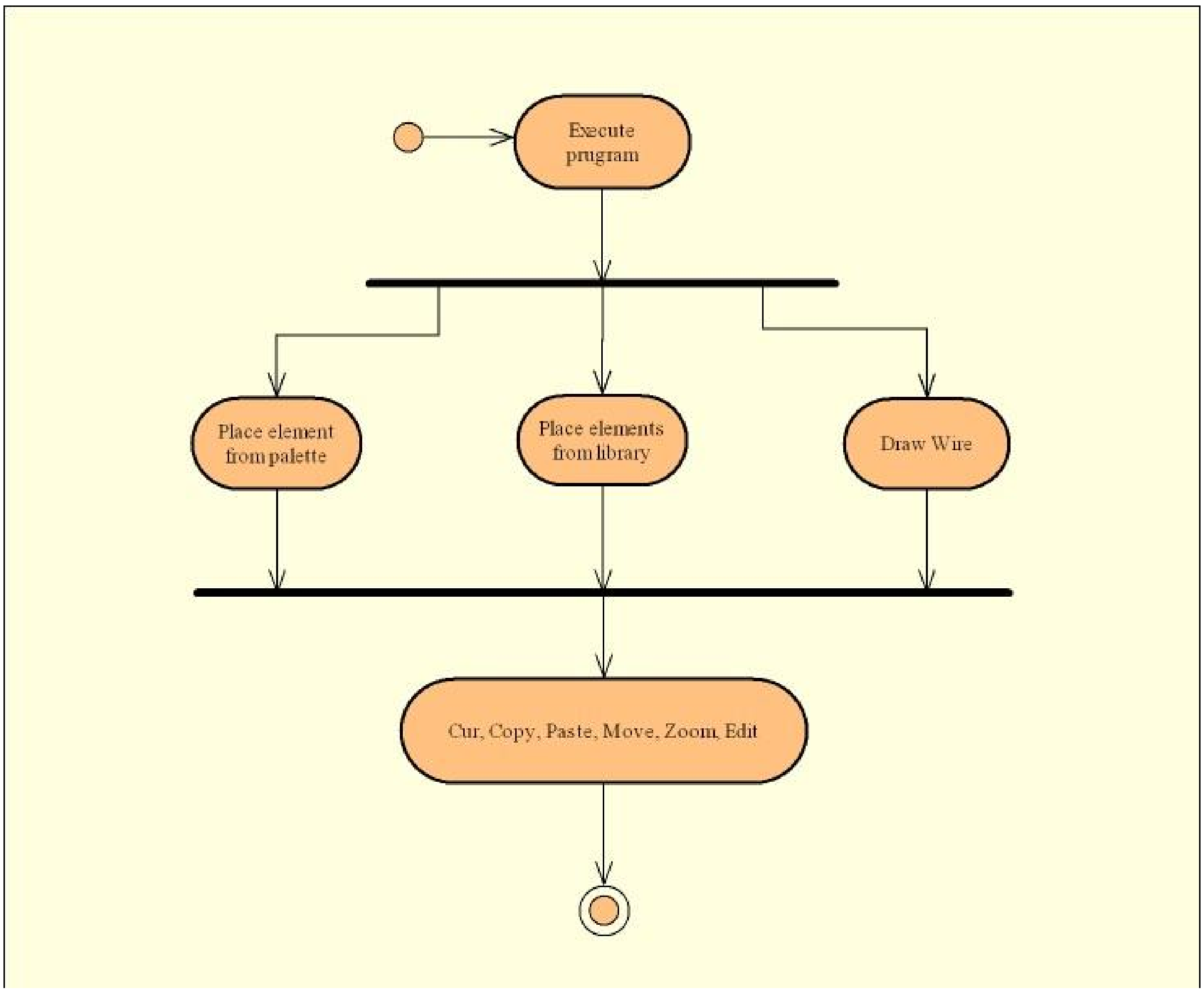
5.3.2.3_GUI Event Control Class Diagram


PopupMenuListener
<<constructor>>+PopupMenuListener(source : Object)
+mousePressed(e : MouseEvent) : void
+mouseReleased(e : MouseEvent) : void
PrintDrawAction
+actionPerformed(e : ActionEvent) : void
CutDrawAction
+actionPerformed(e : ActionEvent) : void
SaveDrawAction
+actionPerformed(e : ActionEvent) : void
ZoomInDrawAction
+actionPerformed(e : ActionEvent) : void
NewDrawAction
+actionPerformed(e : ActionEvent) : void
ToBackDrawAction
+actionPerformed(e : ActionEvent) : void
CopyDrawAction
+actionPerformed(e : ActionEvent) : void
PasteDrawAction
+actionPerformed(e : ActionEvent) : void
OpenDrawAction
+actionPerformed(e : ActionEvent) : void
ToFrontDrawAction
+actionPerformed(e : ActionEvent) : void
ZoomOutDrawAction
+actionPerformed(e : ActionEvent) : void
DeleteDrawAction
+actionPerformed(e : ActionEvent) : void

5.3.3_File I/O Class Diagram


ExportDraw
+exportDrawToXML(graphLayoutCache : GraphLayoutCache) : File
+exportDrawToPDF(graphLayoutCache : GraphLayoutCache) : File
+exportDrawToJPEG(graphLayoutCache : GraphLayoutCache) : File
ExportMacroCell
+exportToXML(graphLayoutCache : GraphLayoutCache) : File
+exportToPDF(graphLayoutCache : GraphLayoutCache) : File
+exportToJPEG(graphLayoutCache : GraphLayoutCache) : File
ImportDraw
+importMacroCell(inputFile : File) : GraphLayoutCache
CellLibrary
+fillFromLibrary() : void
<<getter>>+getCell(id : long) : CELL
ImportMacroCell
+importMacroCell(inputFile : File) : void

5.4 Drawing Activity Diagram


Execute prugram
Place element from palette
Place elements from library
Draw Wire
Cur, Copy, Paste, Move, Zoom, Edit

6. HELP DESIGN

Help menu is very helpful for lots of software products as it is in DIGIMOD. Since we will create a new program for the digital logic world, users who are used to draw on current simulators may not become easily accustom to our product. Also for new users who are also new to logic methodology will probably need a help part. Besides, our script functionality should be explained briefly in help menu in order to make this feature easily adaptable. On these grounds we can conclude that help menu is an essential part of our project.

We have considered a number of design issues to help the user to handle the system easily which are explained below:

- Help menu must be available at all times during system interaction. A user may need the help menu at the beginning or in the middle of an application. For all this situations, whenever it is needed one can read help file. If help menu is clicked during any application, help page is opened on the same level and after user finish with help page, he/she will click “back” button to return to the current application without loosing any data.

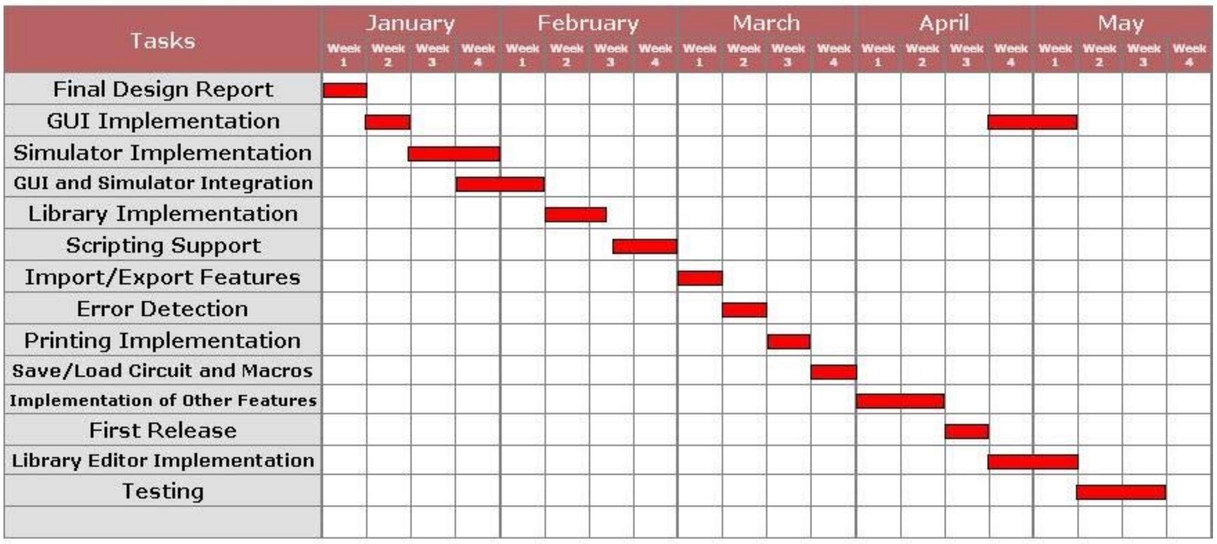
- As partially explained above, help document will be available from help menu at the top of product, also there may be recently and frequently used buttons where one can reach help menu too. Finally there will be a keyboard shortcut in order not to be totally dependent to graphical user interface and to make a fast access.

- All functionality and script features will be explained briefly in help menu. We want our help section to be a helpful one. It will comply with all necessary formalities.

- We will structure help section as one can reach what he/she seeks as fastest as possible. In order to provide this functionality we will create layered architecture of information to ensure increasing detail as user goes further.

7. SCHEDULING

7.1 Gantt Chart



Tasks	January				February				March				April				May			
	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4
Final Design Report	🟥																			
GUI Implementation		🟥														🟥	🟥			
Simulator Implementation			🟥	🟥																
GUI and Simulator Integration				🟥	🟥															
Library Implementation						🟥	🟥													
Scripting Support							🟥	🟥												
Import/Export Features									🟥											
Error Detection										🟥										
Printing Implementation											🟥									
Save/Load Circuit and Macros												🟥								
Implementation of Other Features													🟥	🟥						
First Release															🟥					
Library Editor Implementation																🟥	🟥			
Testing																		🟥	🟥	

8. TESTING

We need to test the software to validate it. So, before making the product ready for users we will be doing necessary white box, black box testing both after implementation of modules or sub modules and after combining them to test whether they are integrated properly. During the testing we will be concerned about the inputs and their expected outputs. The general strategy of our testing will be applying white box testing when possible and black box testing if it is not possible to apply white box testing or the test cases for a particular structure has an impractical amount. So, white box and black box testing will be applied within the regions explained below:

8.1 White Box Testing

We will use white-box testing on the functions of modules just after implementing them so that the details are still in mind and faults can be corrected easily. While applying this method inputs will be given to the functions and outputs will be examined if they are same with the expected outputs.

8.2. Black Box Testing

We will apply black box testing on the modules as a whole and inspect whether they do their correct behavior by inspecting the outputs for given inputs and by inspecting the files to see whether proper changes have been done. This will be done on modules by providing dumb modules that only get input and give output so that we can see the behavior of the tested module as if it is really a part of the entire. Also after combining modules, we will do integrating testing which will again be typical black box testing as explained above.

9. CONCLUSION

Aim of this final design report is to form a way to create methods of our design which will show the way of our implementation. Data given in this report such as use cases, class diagrams, Gantt chart will be our guide to implement DIGIMOD properly in time. This report is also beneficial for letting us to create a milestone for our project and preparing a pre-design report before final design report that leads perfection in design. As a result this report is an essential part of our project.