**Circuit Modeller**

**Documentation**

Emil John D. Lopez

EQ1

# Overview

Circuit modelling is one of those tasks that naturally lends itself to object-oriented programming and some of its major features. There are obvious objects such as circuits and components that can naturally be modelled as classes. Furthermore, the large variety of components all possess common characteristics such as a unit of measurement and a fundamental descriptive value (e.g. resistance for resistors). This implies however that some level of polymorphism is required between the components. Although I tried to make this as fully functional as possible, hectic schedule could not allow the development to go any further than this.

# UML Diagram

# Class Descriptions

Below are the descriptions of the designation and role of every class in the entire program.

## Manager

The *Manager* class is mainly in charge of all menus and serves as a connector between all major objects in the program, namely the Circuit, the Graphics, and the CircuitFileParser classes. It mainly does this through composition of pointers to these classes. Through its various methods, it allows these classes to talk to one another without the need for direct connection to one another.

### Some of the Major Methods

\*\*\* This does not list all the methods of this class

1. *mainMenu():* Contains the display for the menu screen and will provide the user with some basic options such as loading a save file and viewing recently opened files.
2. *everythingMenu():* Contains all of the actions the user can do with the circuit modeller
3. *newCircuit()* and *loadCircuit()*: creates and loads the circuit file into the FILE\* variable.

## Components

The *Component* class serves as the base class for all components that can be utilized by the user in the program. This class is necessary because all the components will be loaded into an array, and as such, polymorphism between classes is needed. It possesses some basic attributes of all components such as the main unit of measurement, a model ID (used as a guide for which model is to be printed into the screen), and the actual quantity of measurement.

A noteworthy feature of this class is the need for a model ID. Since all components will be accessed through a generic Component array, this will serve as the marker for which particular component the object is. This was ultimately decided on by the programmer because it is the easiest way to distinguish between the derived classes without compromising the array system already in place.

### Methods

\*\*\*the underscores ( \_ ) may be replaced by either the word Entry or Exit. Entry refers to the components that come before this one and Exit refers to those after.

\*\*\*in this context, the word *connected* literally means the two components are connected in the circuit.

1. *Component*(double property, char unit, int id) [Constructor]: this loads all the properties of the component. This is called upon by all derived classes.
2. *disconnectComponentFrom\_*(Component\*): removes a Component object from a list of connected components.
3. *connectComponentTo\_*(Component\*): adds a Component object to a list of connected components
4. get\_Components(): returns a structure that contains a list of connected components
5. getModelID()
6. getPropertyUnit()
7. getMainProperty()
8. setMainProperty(double)
9. setReferenceIndex(int)
10. getReferenceIndex(int)

### Derived Classes

1. Resistance
2. Capacitance
3. Inductance
4. DCVoltage
5. ACVoltage

## Circuit

The *Circuit* class is the class responsible for storing all the *Component* objects and managing their interactions and connections. It stores the components in two ways, via a *tree of nodes*, with the root component stored by the class and an *array of pointers* for direct access

### Methods

1. *getRootComponent*(): returns the root of the Component tree
2. *getComponentArray*(): returns a pointer to the array of Component pointers
3. *getComponentAt*(int): returns the pointer of the Component at a certain position in the array
4. *addComponent*(): adds a component to the circuit
5. *removeComponentAt*(int) [*unstable*]: removes the component at a particular index

## Graphic

The *Graphic* class is responsible for displaying the graphics for the circuit itself. It mainly utilizes certain functions in the console.h library. The algorithm used for displaying the components was the simplest one conceivable (see details in next section).

### Methods

1. *setColor*(int): set the color of the text (as defined by the console.h graphics library)
2. *revertColor*(): reverts the text color to white
3. *loadCircuitGraphic*(): loads the currently held Circuit object into the console recursively
4. *loadNextComponent*(): a partner method to loadCircuitGraphics that loads the next circuit component onto the screen. Uses recursion to traverse the Component tree
5. *moveBrush*(int, int): Moves the cursor for printing by some displacement in the x and y direction
6. *paint*(char\*): paints a C string unto the current position of the brush and moves the brush at the end of the string
7. *printNodeTrackerOnTopOfBrush*(): Prints the node labels

## CircuitFileParser

The *CircuitFileParser* class is responsible for parsing circuit files to Circuit objects and vice versa. Further details on the file format and parsing algorithm in the next section

### Methods

1. *parse()*: parses the loaded file and returns the parsed Circuit object
2. *updateFile*(Circuit\*): overwrites the file with a parsed version of the passed Circuit object
3. *createComponent*(double, symbol): creates the appropriate Component object using the passed information
4. *symbolToUnit*(char) converts a component symbol (R for resistors, C for capacitors, etc.) to their respective units