Circuit Tester Design

Rhys Adams

Contents

Decomposition:

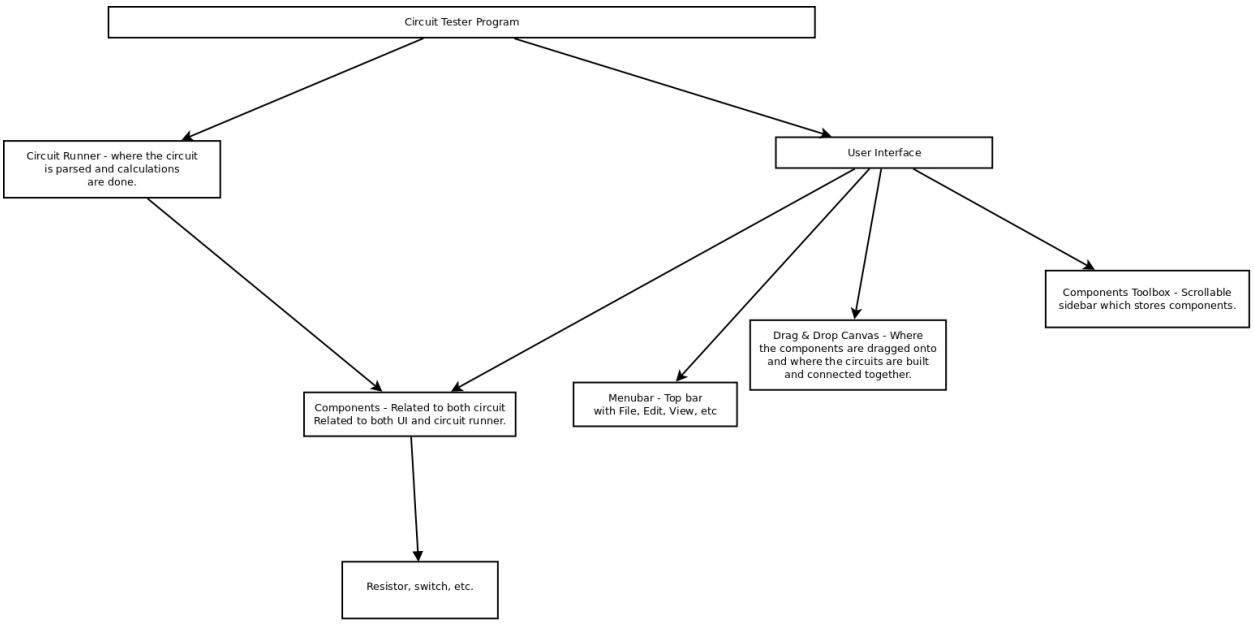

Figure 1: Decomposition

Justifications:

Circuit runner

because the back end logic is totally separate to the front end display, so these need to be separated.

This is also where the linear algebra is done, so I would like to keep the library imports in one class only.

User interface

This will be purely for logic related to the GUI. This will be the only area where I will be importing the QT libraries, as again I wish to keep libraries separate.

Menubar

This will be the part that deals with the bar at the top of the window. The appearance is slightly different depending on the operating system used, as Windows has the menubar in the window, with a file and a item menu, whereas MacOS has the menubar in the top bar of the desktop, and replaces the file and help menus with a general menu for the application, as seen below:

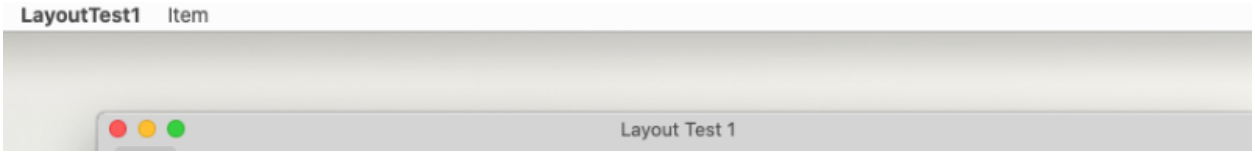

Figure 2: MacOS Menubar

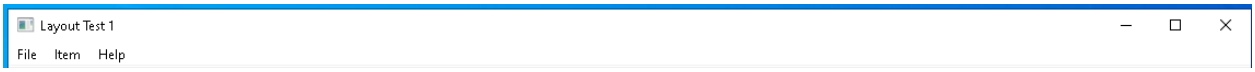

Figure 3: Windows Menubar

Drag and drop canvas

This is the main canvas, so all aspects of the actual circuit go here such as placing down components, connecting them and removing them. Each component is a component class that can draw itself and can also conduct some mathematics.

Components toolbox

This is the scrollable toolbox that contains the components that can be dragged onto the canvas. This is separate to the canvas because this contains only buttons that create the components, whereas the canvas contains the actual component classes.

UI Design

Main Window (Settings Open)

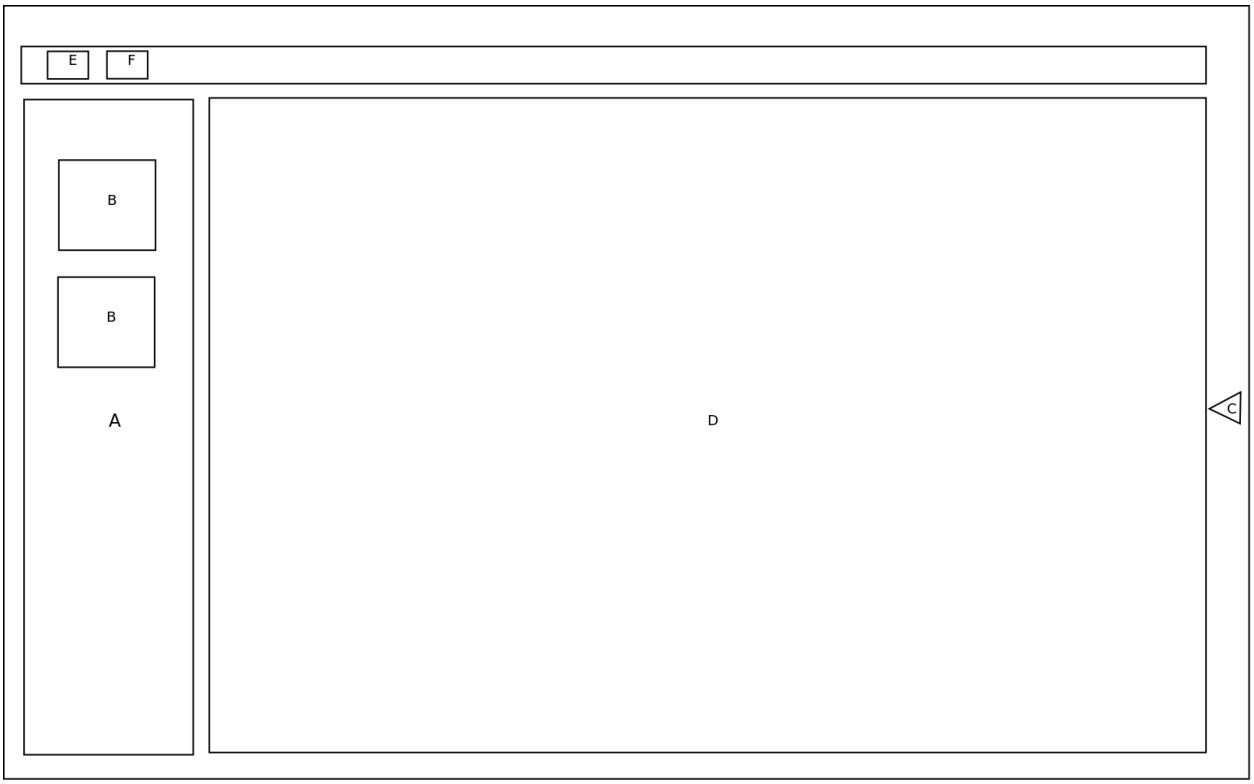

Figure 4: Main Window Wireframe

- A: The toolbox. Contains the components that can be dragged onto the canvas.
- B: Components. These represent the QSceneItems that can be placed onto the canvas. These are buttons that use a smaller version of the image of the component.
- C: Expandable settings menu. This is expanded when the user double clicks on a component that is placed on the canvas. It will contain the different settings that each component will have.
- D: The canvas. This is the main QGraphicsScene that will allow the components to be dragged onto and moved about. It also handles the signals when a line is drawn between two components.
- E: Drag tool. This button is pressed when the user wishes to drag components around the canvas. This is the default action.
- F: Wire tool. This button is pressed when the user wishes to connect components together with a wire.

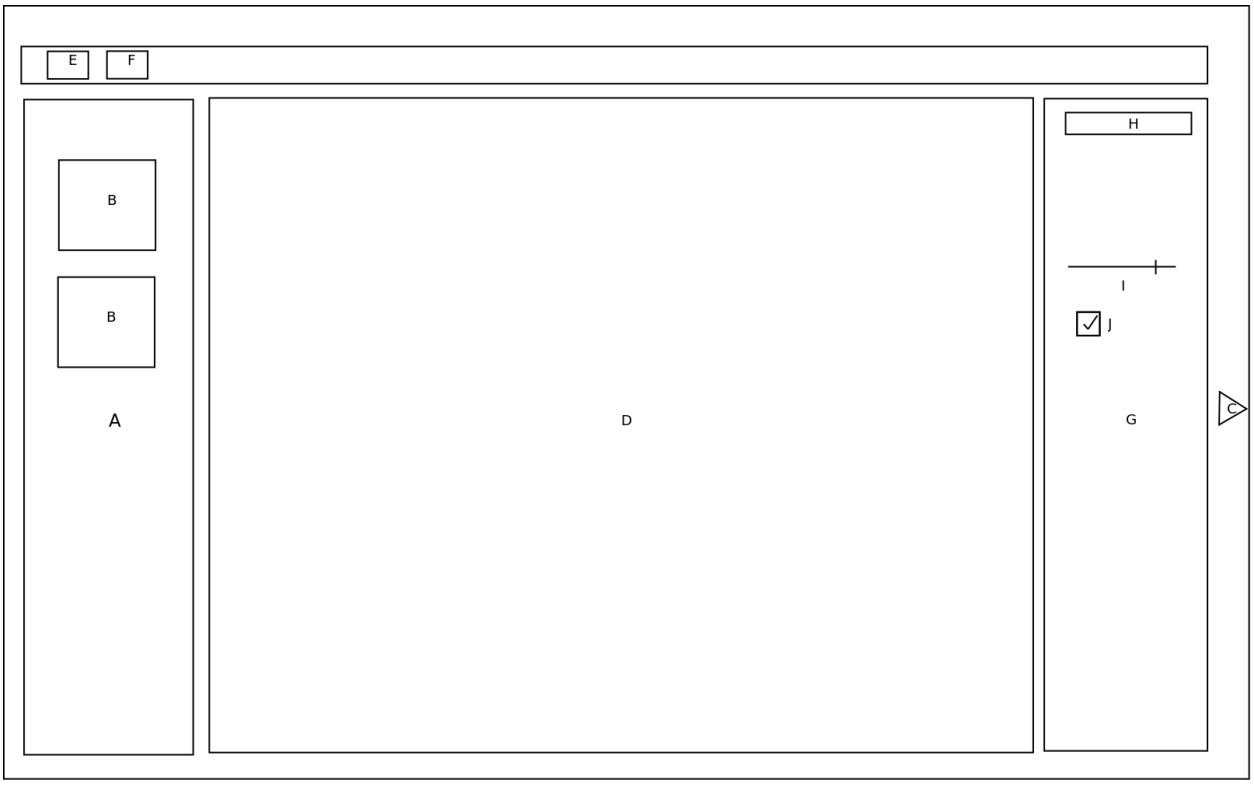

Figure 5: Main Window Wireframe With Settings Open

Main Window (Settings Open)

- G: Settings menu. This is the expanded version of C (which is now facing the opposite direction). This contains the settings that vary for each component.
- H: Title. The name of the currently selected component.
- I: Setting slider. An example of the type of setting that can be set in the menu. A slider could be for battery voltage, or for resistivity of wires.
- J: Settings checkbox. An example of the type of setting that can be set in the menu. A checkbox could be to enable or disable a switch or battery.

Mockups

Main Window

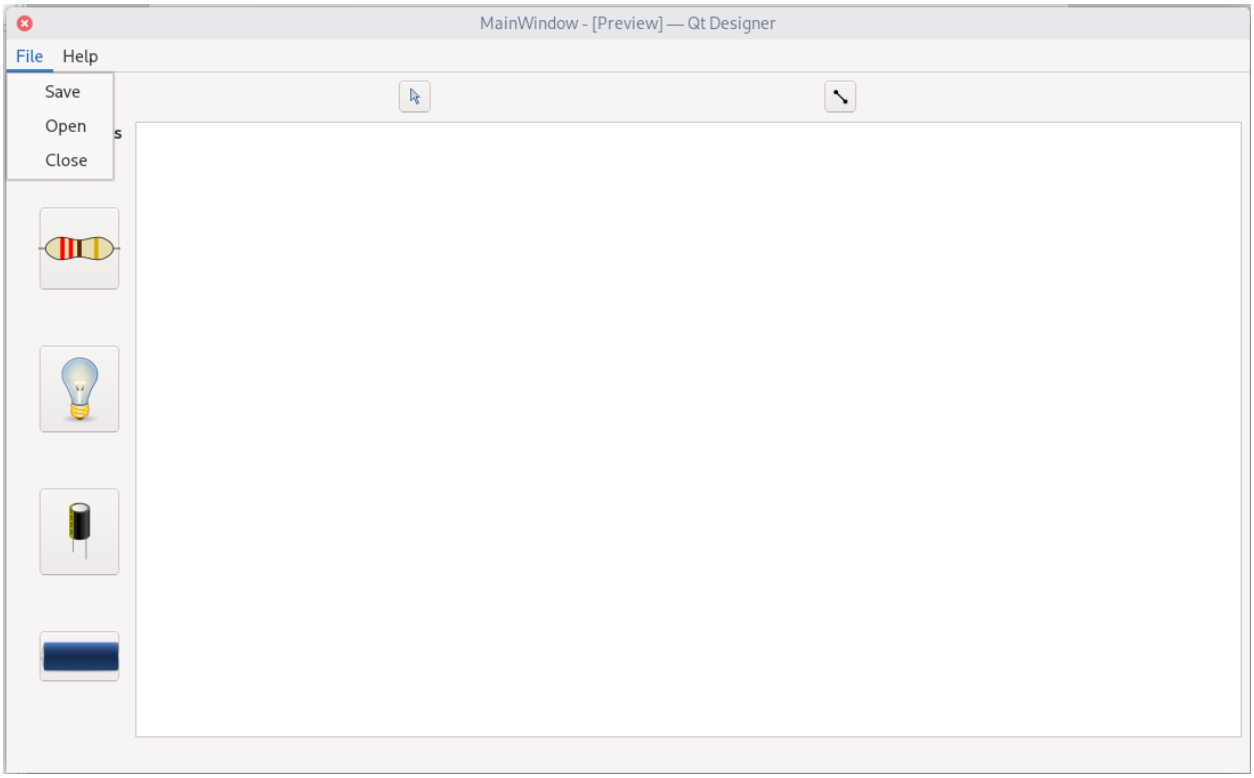

Figure 6: Main Window Mockup

This is what the user will see as they open up the program.

There is a simple yet fully featured scrollable bar at the side, which is designed to hold the components.

The buttons at the top are to toggle the mouse action, with the cursor for letting the user drag the components, and the line to allow the user to connect components together.

Main Window In Use

This is the same as above but with a circuit diagram layed out.

The images on the canvas are just larger versions of the images on the buttons, so are easily recognisable.

Usability Features

My program is designed to be very easy to understand.

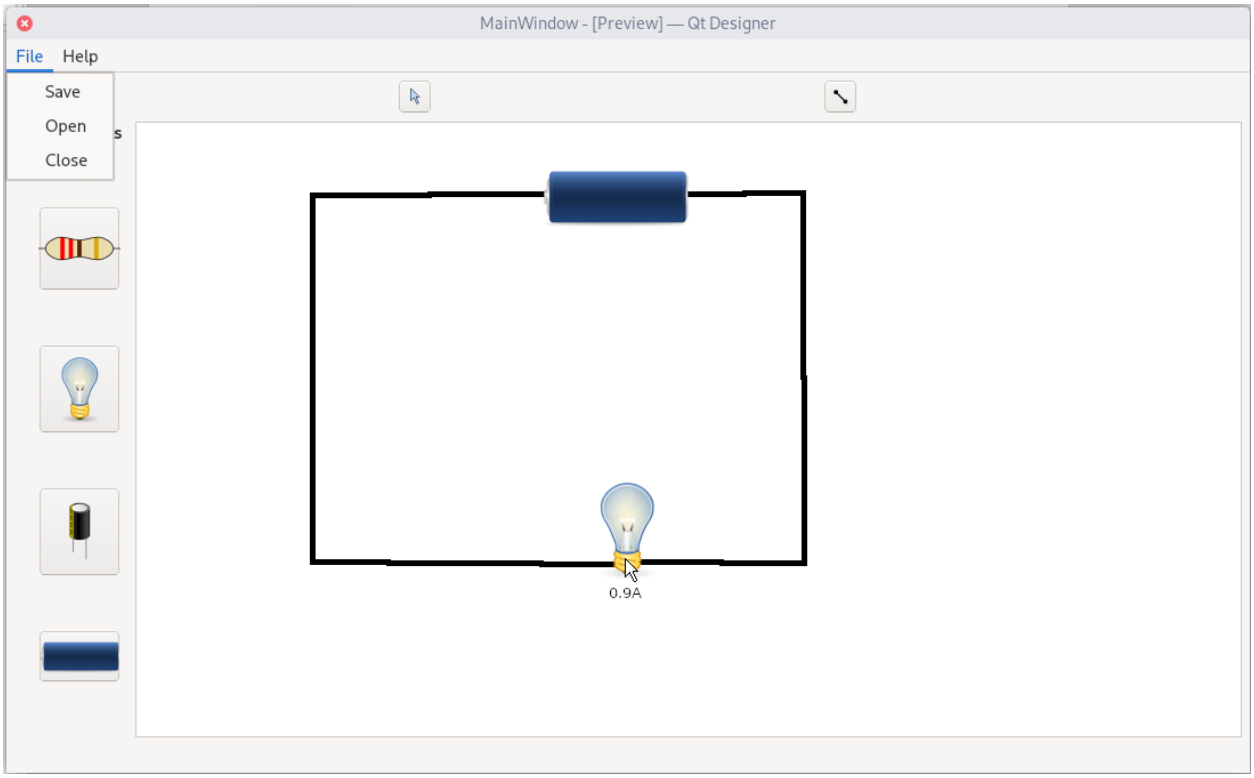

Figure 7: Main Window Mockup With Diagram

The toolbox is designed to show the components as large, colourful pictures, which means that they are very easy to see.

The toolbox buttons are left-clicked once, then the user moves the cursor to the place they wish to put the component. This is where the component is placed.

Algorithms

The most complex part of my program is the circuit simulation code, which uses linear algebra to find the currents and voltages at different parts of the circuit.

I will use modified nodal analysis (MNA) and Kirchhoff's laws to work out currents and voltages at different nodes and vertices of the circuit.

Convert circuits into matrices

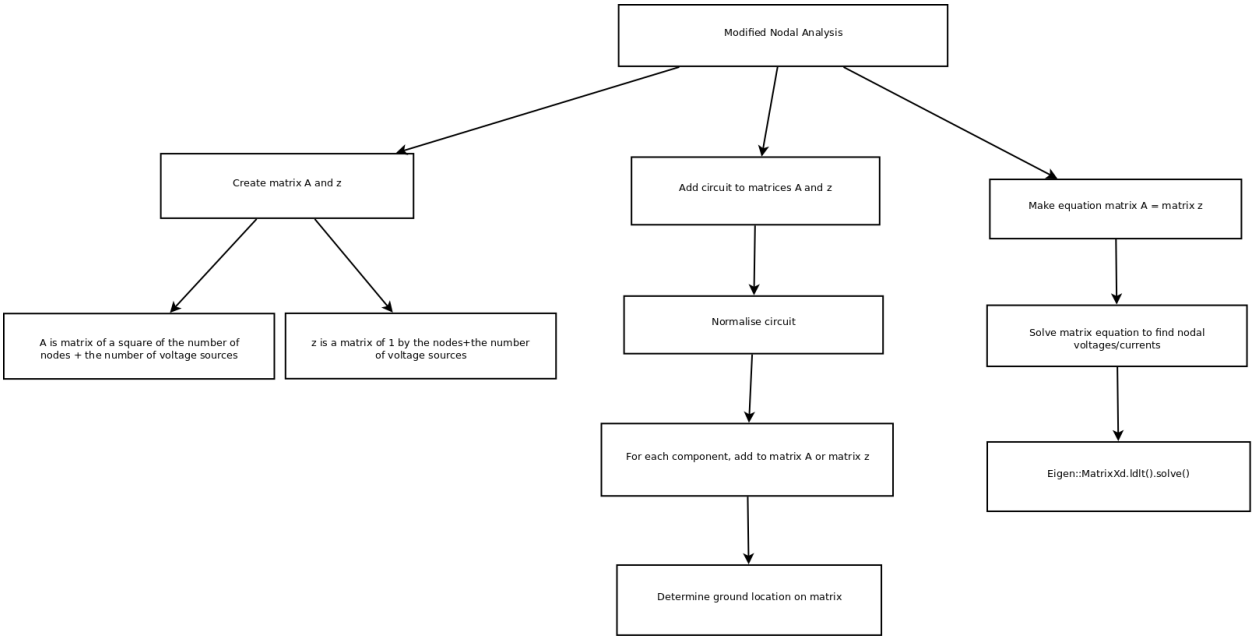

Figure 8: Algorithm Decomposition

Circuit

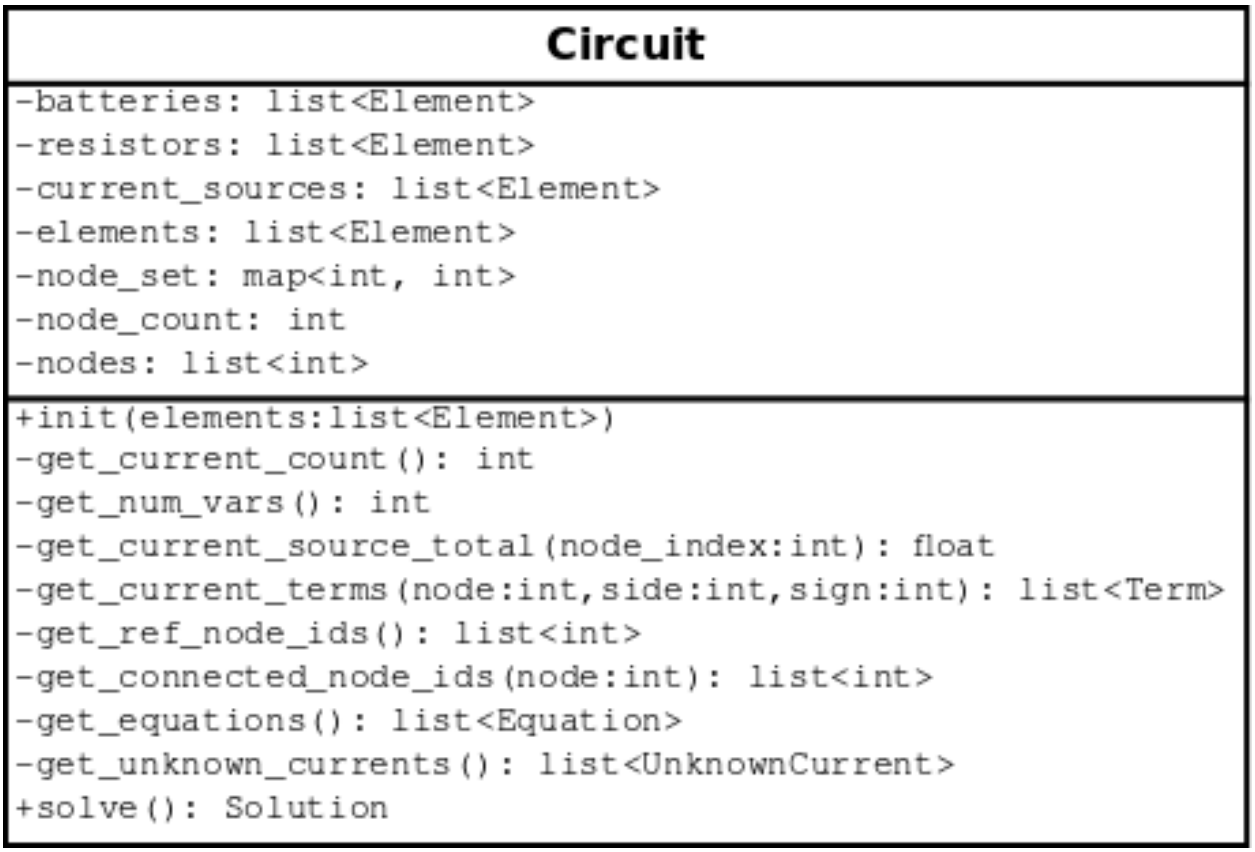

Figure 9: Circuit Class

Initialiser

```
FUNCTION init(circuit_elements)
    FOR e IN circuit_elements DO -- splits the elements array
        IF e IS Battery THEN
            batteries.append(e)
        ELSEIF e IS Resistor THEN
            resistors.append(e)
        ELSEIF e IS CurrentSource THEN
            current_sources.append(e)
        END IF

    END

    {- adds together all 3 arrays. Not using the function parameter
    as this could contain values that are not batteries,
    resistors or current sources -}
```

```
    elements = batteries + resistors + current_sources

    FOR e IN elements DO -- populate the node_set map
        node_set[e.n0] = e.n0
        node_set[e.n1] = e.n1
    END

    node_count = node_set.length() -- count the number of nodes
    nodes = node_set.values() -- set nodes to just the real node ids
END
```

Solver

```
{-
Function that solves the circuit using
linear algebra and matrices.
-}
FUNCTION solve()
    equations = get_equations()
    unknown_currents = get_unknown_currents()
    unknown_voltages = MAP({x => NEW UnknownVoltage(x)}, nodes)

    unknowns = unknown_currents + unknown_voltages

    -- Make two matrices, one which is wide, and the other that is only 1 wide.
    a = Matrix.Zeroes(Float, equations.length(), get_num_vars())
    z = Matrix.Zeroes(Float, equations.length(), 1)

    FOR i IN (0 .. equations.length()) DO
        -- Stamp every equation, passing in the lambda to find the index of each
        -- term of the equation in class attribute unknowns.
        equations[i].stamp(i, a, z, {x => get_index_by_equals(unknowns, x})
    END

    TRY
        -- Hopefully the equation is solved
        x = a.solve(z)
    CATCH EXCEPTION
        -- If there is an error, make an empty output matrix
        -- to stop the whole program from crashing.
        x = Matrix.Zeroes(Float, equations.length(), 1)
    END
```

```
    voltage_map = {}

    FOR v IN unknown_voltages DO
        -- Get the right side of the equation for the voltage at
        -- point voltage node.
        equation_right = x[get_index_by_equals(unknowns, v), 0]
        voltage_map[v.node] = equation_right
    END

    FOR c IN unknown_currents DO
        -- Set the new current for each element.
        c.element.current_solution = x[get_index_by_equals(unknowns, c), 0]
    END

    -- Return a new solution class.
    RETURN NEW Solution(voltage_map, map({x => x.element}, unknown_currents}))
END
```

Miscellaneous functions

```
{-
Gets the number of unknown currents
Current sources are not calculated as they are known
currents.
-}
FUNCTION get_current_count()
    zero_resistors = 0

    {-
    Sum the amount of resistors that have zero resistance.
    these resistors have an unknown current.
    -}
    FOR r IN resistors DO
        IF r.value == 0 THEN
            zero_resistors += 1
        END
    END

    -- Add together the resistors and the number of batteries.
    RETURN zero_resistors + batteries.length()
END FUNCTION
```

```
{-
Used to find out the number of both current and
voltage variables.
-}
FUNCTION get_num_vars()
    RETURN node_count + get_current_count()
END


{-
Finds the total current leaving a node
-}
FUNCTION get_current_source_total(node)
    total = 0

    FOR c IN current_sources DO
        IF c.n0 == node THEN
            {-
            Positive current leaves, but conventional current
            states that outgoing current must be positive.
            -}
            total += c.value
        ELSEIF c.n1 == node THEN
            {-
            Positive current enters, but conventional current
            states that incoming current must be negative.
            -}
            total -= c.value
        END
    END

    RETURN total
END


{-
Gets the terms that enter and leave a node.
Incoming is negative, outgoing is positive
-}
FUNCTION get_current_terms(node, side, sign)
    terms = []

    FOR b IN batteries DO
        n = (side == 0? b.n0 : b.n1) -- Convert int into node.
```

```
        -- If node specified in parameter.
        IF n == node THEN
            -- Batteries have an unknown current
            terms.append(NEW Term(sign, NEW UnknownCurrent(b)))
        END
    END


    FOR r IN resistors DO
        n = side == 0? r.n0 : r.n1 -- Convert int into node.

        -- If correct node and resistor has no resistance.
        IF n == node AND r.value == 0 THEN
            -- If resistance is 0, the resistor has an unknown current.
            terms.append(NEW Term(sign, NEW UnknownCurrent(b)))
        END

        IF n == node AND r.value != 0 THEN
            -- If the resistance is not 0, the resistor has
            -- an unknown voltage.
            terms.append(
                NEW Term(-sign/r.value, NEW UnknownVoltage(r.n1))
            )

            terms.append(
                NEW Term(sign/r.value, NEW UnknownVoltage(r.n0))
            )
        END
    END

    RETURN terms
END



{-
Selects the components that have the reference voltage.
This is the voltage that other components are compared
to.
-}
FUNCTION get_ref_node_ids()
    to_visit = node_set.values()

    ref_node_ids = []
```

```
    WHILE NOT to_visit.empty()
        -- Get the first element in the nodesto be visited.
        ref_node_id = to_visit[0]

        ref_node_ids.append(ref_node_id)

        connected = get_connected_node_ids(ref_node_id)

        -- All nodes connected to this node have had their reference
        -- set to this node, so don't check them.
        FOR c IN connected DO
            -- Delete them from the list
            to_visit.remove(c)
        END
    END

    RETURN ref_node_ids
END



{-
Finds the nodes that are connected to the passed node
-}
FUNCTION get_connected_node_ids(node)
    visited = []
    to_visit = [node]

    WHILE NOT to_visit.empty()
        -- Pop the first value off of to_visit.
        node_to_visit = to_visit.pop(0)

        visited.append(node_to_visit)

        FOR e IN elements DO
            IF e.contains_node_id(node_to_visit) THEN
                -- Get the opposite node, check that too
                opposite = e.get_opposite_node(node_to_visit)

                -- Don't check if has already been checked
                IF NOT visited.contains(opposite) THEN
                    to_visit.append(opposite)
                END
            END
```

```
        END
    END

    -- Remove duplicate values from visited.
    visited = unique(visited)

    RETURN visited
END



{-
Returns a list of equations that must be solved
to solve the overall circuit.
-}
FUNCTION get_equations()
    equations = []

    ref_node_ids = get_ref_node_ids()

    FOR r IN ref_node_ids DO
        -- Reference nodes have a voltage of zero as the voltage
        -- is relative to this.
        equations.append(NEW Equation(0, [
            NEW Term(1, NEW UnknownVoltage(r))
        ]))
    END

    FOR n IN nodes DO

        {-
         One node must be selected to not have its charge conserved.
         This is because having full charge conservation breaks
         the matrix solver.
        -}
        IF NOT ref_node_ids.contains(n) THEN
            incoming = get_current_terms(n, 1, -1)
            outgoing = get_current_terms(n, 0, +1)

            terms = incoming + outgoing

            equations.append(NEW Equation(
                get_current_source_total(n), terms
            ))
```

```
        END
    END

    FOR b IN batteries DO
        -- Make an equation for every battery
        equations.append(NEW Equation(b.value, [
            -- N0 is negative as voltage is lost.
            -- N1 is the positive as voltage is gained.
            NEW Term(-1, NEW UnknownVoltage(b.n0)),
            NEW Term(1, NEW UnknownVoltage(b.n1))
        ]))
    END

    FOR r IN resistors DO
        IF r.value == 0 THEN
            -- If resistance is 0, n0 and n1 are the same, as there
            -- may as well not be a component there
            equations.append(NEW Equation(r.value, [
                NEW Term(1, NEW UnknownVoltage(r.n0)),
                NEW Term(-1, NEW UnknownVoltage(r.n1))
            ]))
        END
    END

    RETURN equations
END



{-
Returns all unknown currents in the
circuit
-}
FUNCTION get_unknown_currents()
    unknowns = []

    FOR b IN batteries DO
        -- Batteries have an unknown current before they
        -- are placed into a circuit.
        unknowns.append(NEW UnknownCurrent(b))
    END

    FOR r IN resistors DO
        -- Zero-resistance resistors have an unknown current.
```

```
        IF r.value == 0 THEN
            unknowns.append(NEW UnknownCurrent(r))
        END
    END

    RETURN unknowns
END
```

Element

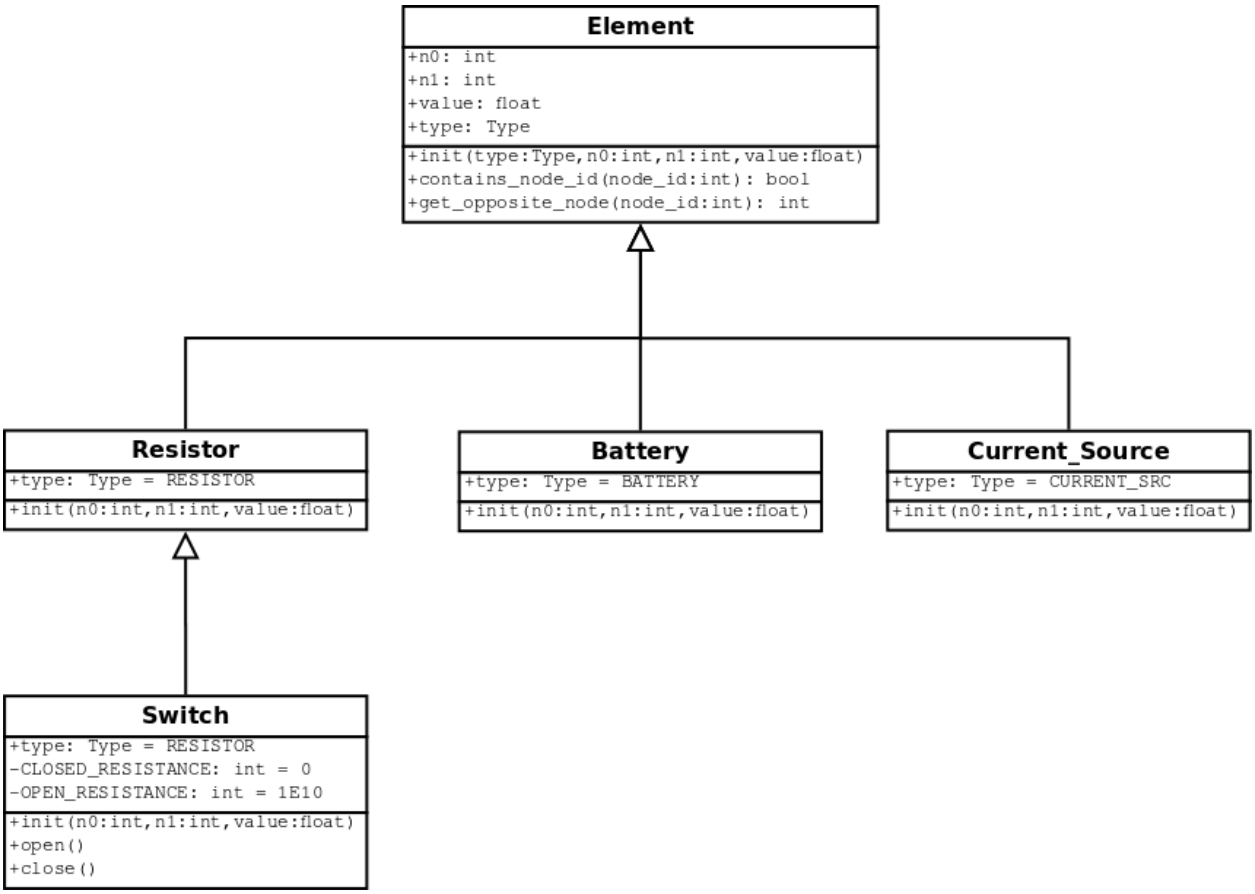

Figure 10: Element Class

Pseudocode for class

```
{-
Returns whether this element is connected
to passed node.
-}
FUNCTION contains_node_id(node)
    RETURN n0 == node OR n1 == node
END



{-
Returns the opposite node to node.
-}
FUNCTION get_opposite_node(node)
```

```
    -- If passed n0, return n1,
    -- and vice versa.
    RETURN node == n0? n1 : n0
END
```

Equation

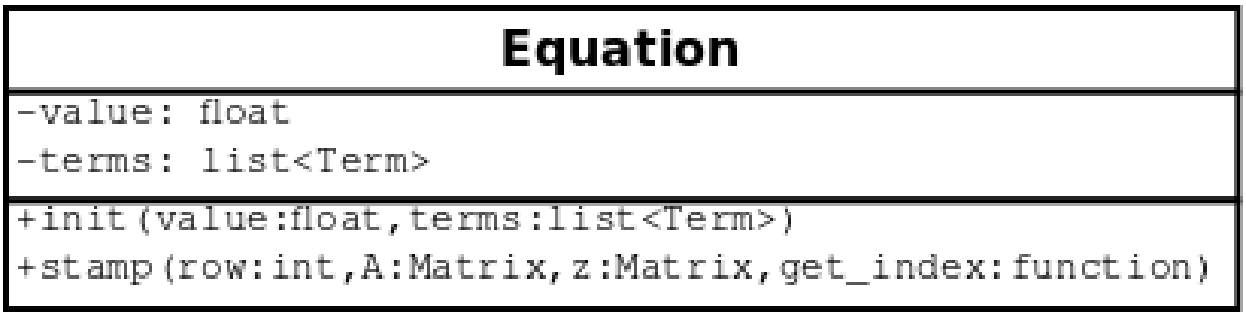

Figure 11: Equation Class

Pseudocode for class

```
{-
Stamps the parts of this component
onto the passed matrice to form the overall
equation.
-}
FUNCTION stamp(row, a, z, get_index)
    z[row, 0] = value

    FOR t IN terms DO
        index = get_index(t.variable)
        a[row, index] = t.coefficient + a[row, index]
    END
END
```

Solution

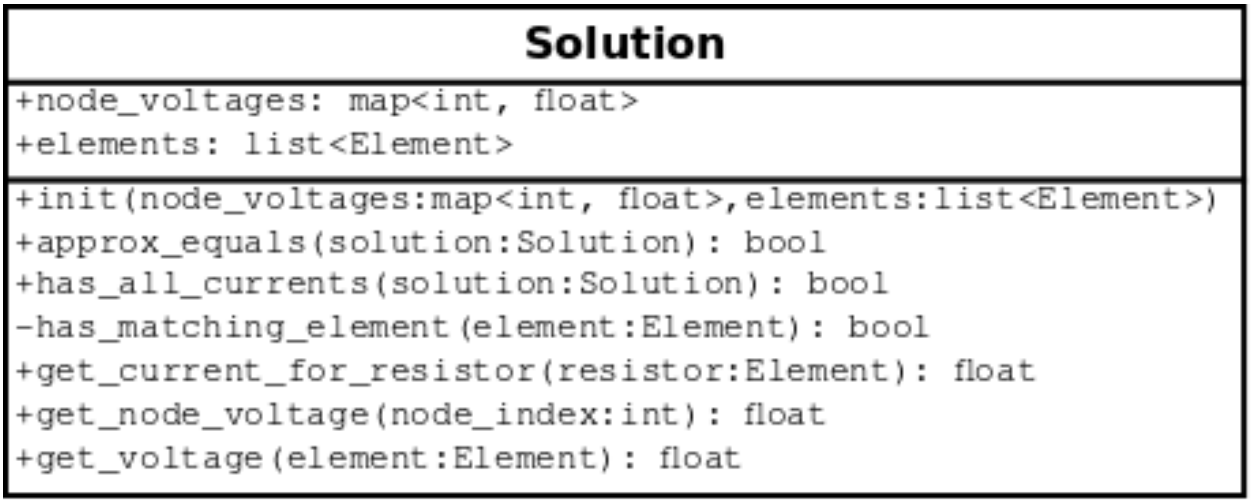

Figure 12: Solution Class

Pseudocode for class

```
{-
Determines if two floats are
approximately equal.
-}
FUNCTION num_approx_equals(a, b)
    RETURN abs(a-b) < 1E-6
END

{-
Determines if two solutions are
close enough to be equal, as floating
point error can cause two identical
solutions to be slightly off.
-}
FUNCTION approx_equals(solution)
    keys = node_voltages.keys()
    other_keys = solution.node_voltages().keys()

    -- Key difference is all the keys in "keys",
    -- without the keys in "other_keys".
    -- This is zero unless there is a big problem.
    key_difference = keys.without(other_keys)

    FOR k IN keys DO
```

```
        -- Check if every key in this equals
        -- the same key in the solution
        IF NOT num_approx_equals(
            solution.get_node_voltage(k),
            get_node_voltage(k)
        ) THEN
            RETURN False
        END
    END

    -- Check if all the currents in this are in solution.
    IF NOT has_all_currents(solution) THEN
        RETURN False
    END

    -- Check if all the currents in solution are in this.
    IF NOT solution.has_all_currents(this) THEN
        RETURN False
    END

    -- All checks passed, must be equal
    RETURN True
END



{-
Finds if all the currents in solution
has an identical current in this.
-}
FUNCTION has_all_currents(solution)
    FOR e IN elements DO
        IF NOT has_matching_element(e) THEN
            RETURN False
        END
    END

    RETURN True
END


{-
Determines if this contains passed
element.
```

```
-}
FUNCTION has_matching_element(element)
    FOR e IN elements DO
        IF e.n0 == element.n0 AND
            e.n1 == element.n1 AND
            num_approx_equals(
                e.current_solution,
                element.current_solution
            )
        THEN
            RETURN True
        END
    END

    RETURN False
END



{-
Returns the current of the passed resistor.
-}
FUNCTION get_current_for_resistor(resistor)
    RETURN -get_voltage(resistor)/resistor.value;
END


{-
Returns the voltage of the passed node.
-}
FUNCTION get_node_voltage(node)
    RETURN node_voltages[node]
END



{-
Returns the voltage of the passed element.
-}
FUNCTION get_voltage(element)
    RETURN node_voltages[element.n1] - node_voltages[element.n0]
END
```

Term

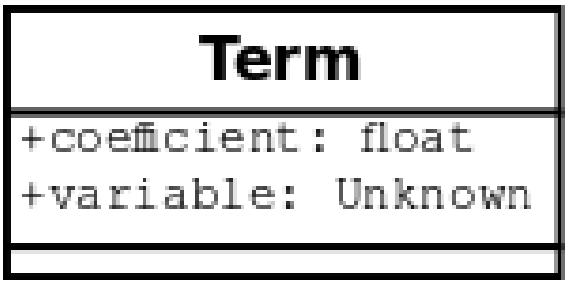

Figure 13: Term Class

This has no pseudocode as it it purely a structure.

In programming languages with a struct type, this is a struct. Otherwise it is a class with no methods.

Unknown

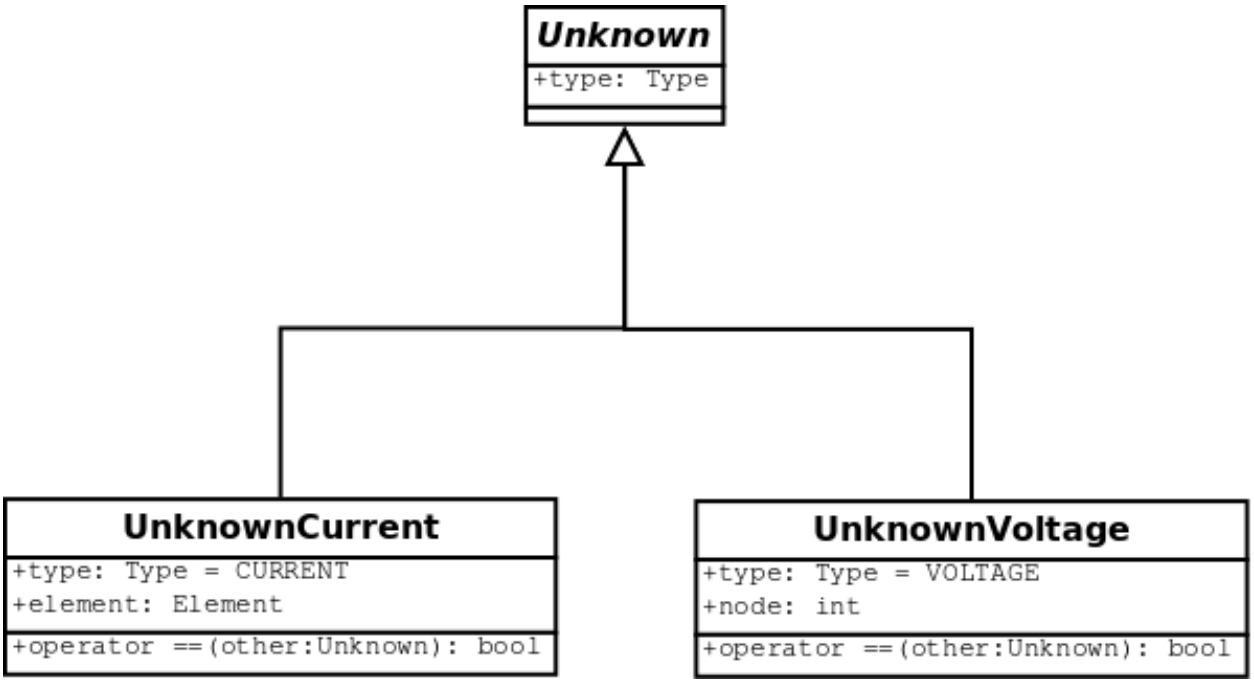

Figure 14: Element Class

These are simply elements for which the voltage or current has not yet been calculated.

Both UnknownVoltage and UnknownCurrent are under the Unknown class, so that they can both be stored in the same arrays as pointers.

The Type attribute allows functions to know if they are an UnknownVoltage or an UnknownCurrent.

Unknown Voltage

```
{-
Equality, on some languages
this can override the == operator,
on others it cannot be overidden.
-}
FUNCTION equals(other)
    -- If not the same type,
    -- they are obviously not equal.
    IF other.type IS NOT type THEN
        RETURN FALSE
    END
```

```
    RETURN node == other.node
END
```

Unknown Current

```
{-
Equality, on some languages
this can override the == operator,
on others it cannot be overidden.
-}
FUNCTION equals(other)
    -- If not the same type,
    -- they are obviously not equal.
    IF other.type IS NOT type THEN
        RETURN FALSE
    END

    RETURN element == other.element
END
```