



Simplifying a ton of physics, one step at a time.

Made by: Prannaya Gupta (M20307, Class of 2023)

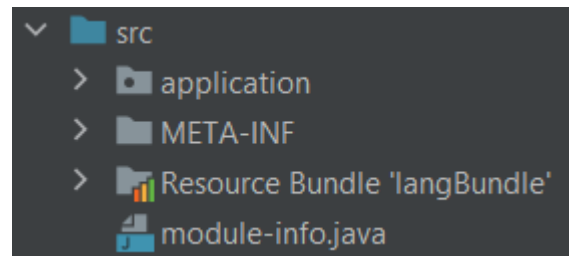
Phyton is a Direct Current (DC) Circuit Simulator made to provide a hands-on learning opportunity to students, especially during a period like the current (as of written) Covid-19 situation, where students cannot visit laboratories to indulge in practical sessions using electrical circuits. A main problem that students face without any practical session is the inability to visualize complex circuit diagrams realistically, and this is where phyton comes in. Phyton eases this issue by forming the circuit entirely using circuit symbols. These symbols allow students to comparatively understand how these circuits can be formed.

These circuits also provide an interface to teachers where they can make complex circuit diagrams to set as questions, while giving the answer to many possible questions at the side. Although the scope of the simulator is currently very small, I still believe that this application can be helpful for students, especially those with a more kinaesthetic or visual learning style. The application allows formation of series circuits, where one can vary the electromotive force (EMF) of a cell or resistance of a resistor, for instance. The application also sports a simple file management system, where one can save their circuits either as EXML or XML files. One can also screenshot the view and save it as a .png, .jpg, .gif or .bmp.

This program is quite large (37 MB) and should be used on a laptop/desktop, since the user interface is specially designed for these devices. Powered by Java, this application was made using the JavaFX GUI Framework, and boasts a simple bright mode UI based on the JetBrains IntelliJ IDEA Ultimate IDE, although it is relatively simpler in order to ensure that the user is not intimidated.

Class Design

This application is much more complicated than others encountered, with 7 controller classes and 55 classes in the model. In the right, you can see the directory structure of my src folder.



Model

The model, with around 57 classes, was the largest subpackage, and for good reason. Since there were minimal APIs online for electrical circuits, I was forced to code the entire package on my own. The model consists of a sub-subpackage quantity, which contains quantity classes like Resistance, Voltage, Current and Power, which I integrated using a powerful regex-based class called UnitValue, which I had previous experience making, but in other languages. I used these classes as the backbone of my model.

Another sub-subpackage is that of util, where I have implemented JavaFX-based classes like Point, Rotation, and classes to aid me in the File Management System, with my custom File class. I have also taken open-source codes like Gerrit Grunwald's AnglePicker class from GitHub and Michael Berry's DraggableTab class, modified them to fit my needs and placed them in a subpackage of util called fxtools. I used many of these classes to my advantage in the coding process, with the AnglePicker being used for rotation and the DraggableTab allowing more usability of the tabs in my application.

A primary sub-subpackage employed in the model was that of the circuitry sub-subpackage. The circuitry subpackage employed many important classes crucial to the development of this application. One of these was the base class, CircuitObject, which implements Resistance, Voltage, Current and Power, which are the base values of every single component implemented. Another class is the Component class, which extends from the

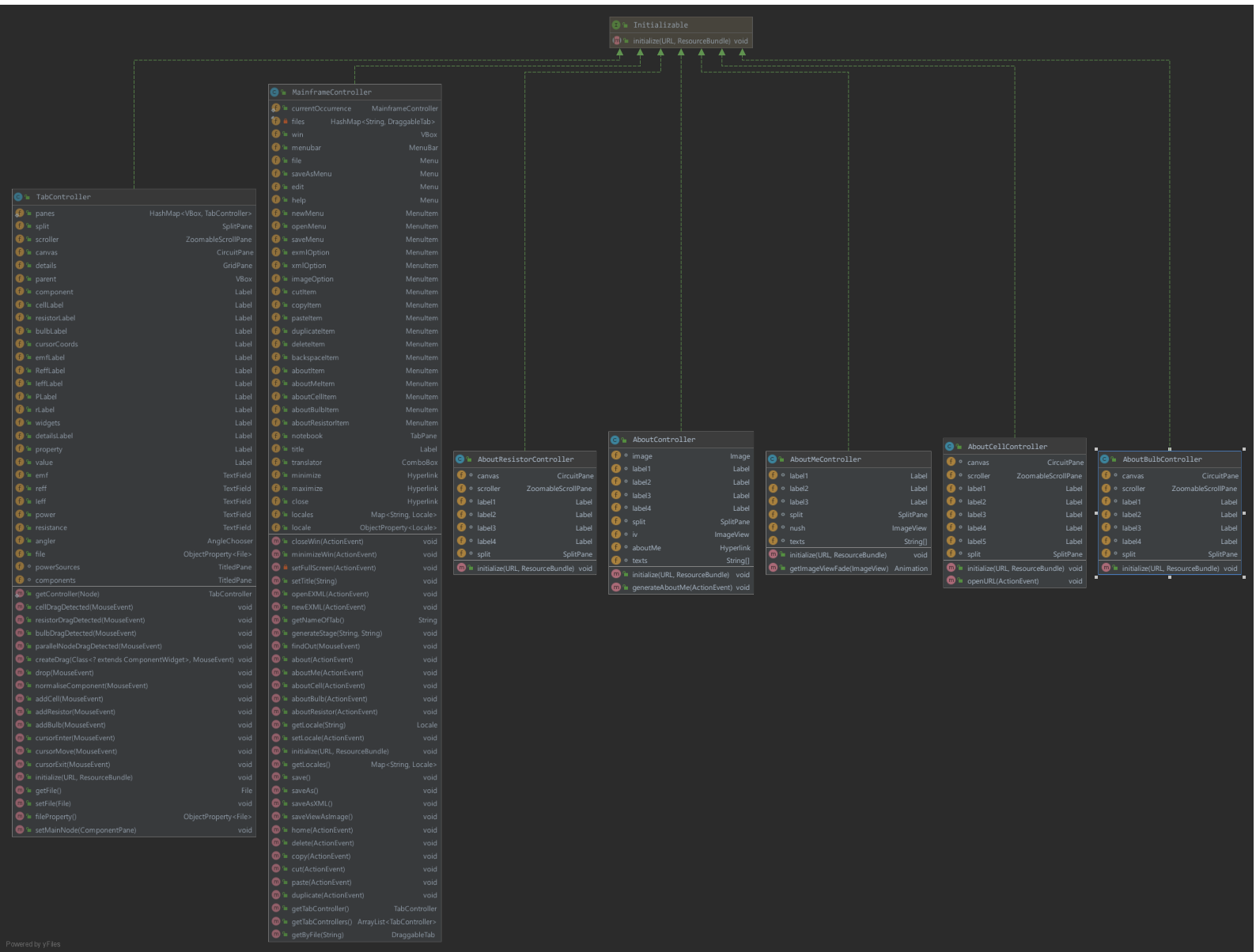
Another sub-subpackage that is equally important is the fxcircuitry sub-subpackage, which provides basic wrappers for each of the classes in the circuitry subpackage, with an abstract ComponentWidget class that all other widget wrappers extend from. In addition, there is the implementation of the ComponentPane and CircuitPane classes, both of which are important classes in the context of the entire project. The ComponentPane widget serves as a way to record the position, rotation and connected components of a particular component. This class also forms a connection between the application and the component object. Additionally, the CircuitPane class serves as the pane used in every tab and creates the Series objects of each Series made.

Controller

There are two main controllers: the MainframeController class and the TabController class.

The MainframeController handles the mainframe.fxml file and is mainly a controller for the overall user interface. Here, methods for cut, copy, paste, delete and many more are implemented. There is also the implementation of internationalization for multiple languages: Spanish, French and Hindi, and is mainly a controller for the menubar. This also accesses the documentation that I have provided for each of the 3 widgets, with clear instructions on how to use various components. It also accesses the about pages I have implemented. It uses the file management system and is the main implementation for the Electric XML files.

Below is the UML diagram for the controller. All controllers are implementations of the Initializable interface, and all in charge of different fxml files.



The TabController, on the other hand, handles the tab.fxml file and thus handles the math, handling text fields in the UI. It also handles creation, drag and drop of widgets. Hence, this class mainly just handles the tab and is a connector between the tab.fxml file and the CircuitPane object.

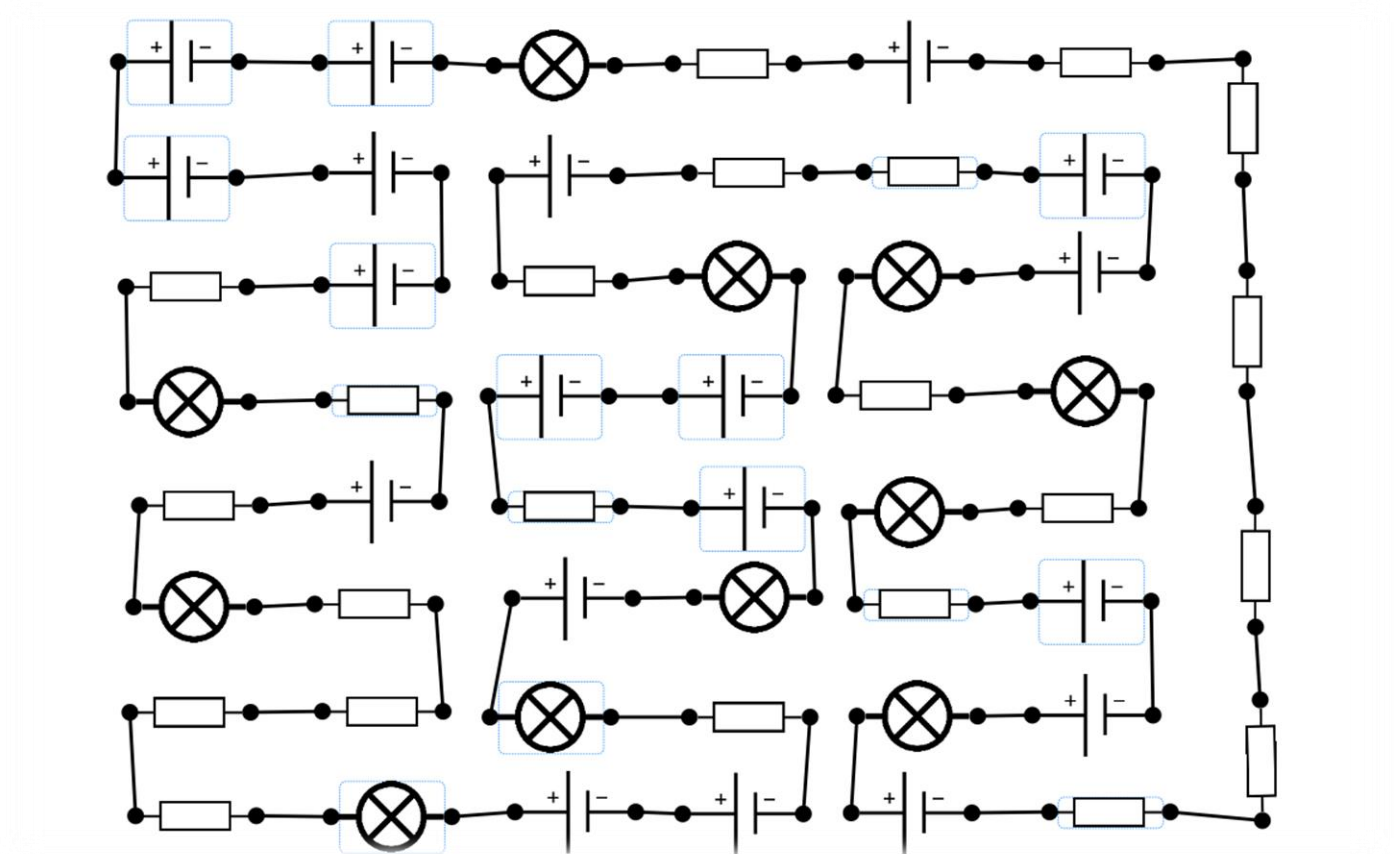
Key Features

Splash Screen

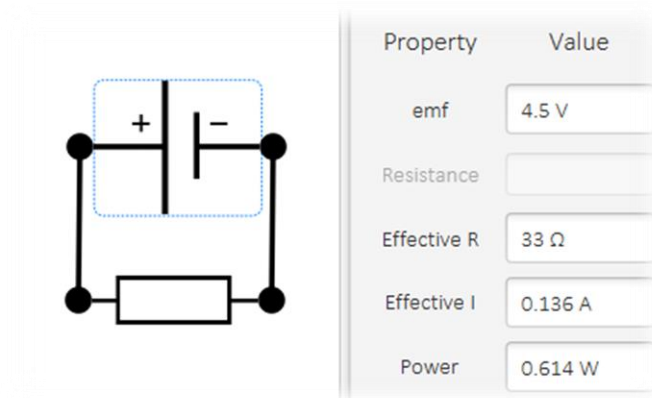


This is a very simplistic Splash Screen that starts playing after about half a second of delay of clicking the JAR. There is an object moving animation that is symbolic of physics, while the lightning is symbolic of electricity. There are also many fun texts shown below the progress bar, upon whose completion, the opacity of the splash screen gradually becomes 0, followed by the application being started up. While this may look very mundane, the splash screen lasts for around 8 seconds after which you are free to use the application. It is mainly inspired by the IntelliJ Splash Screen, seeing with the white bar starting from the horizontal start of the splash screen to the end.

Complex Circuit Diagrams

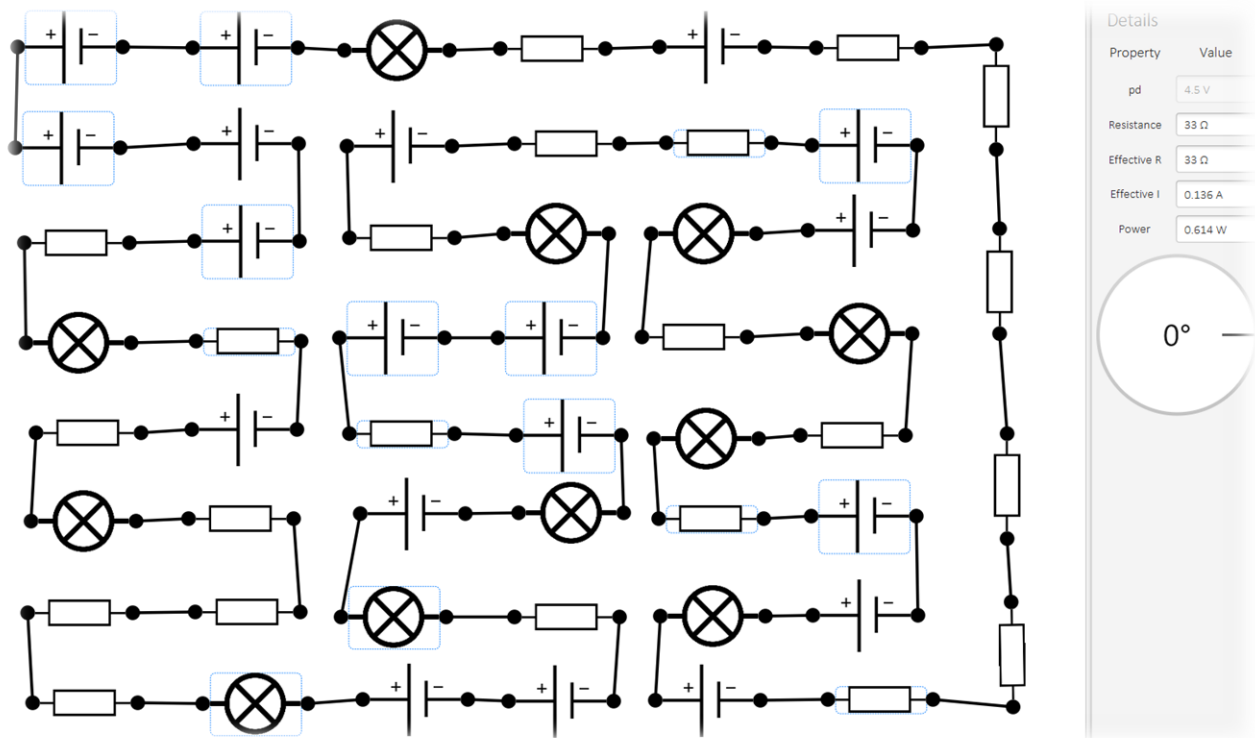


While the above circuit may seem incredibly complicated, this is, in fact a very simplistic diagram, as shown below.

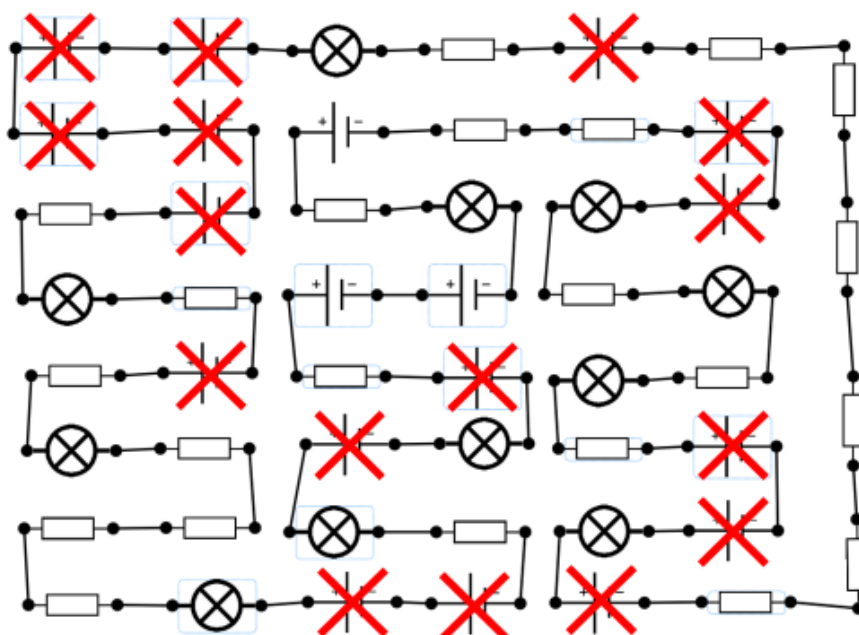


Here, the resistor has been configured to have 33 ohms of resistance instead of the generic 1 ohm that all the resistors and bulbs above contain. At the same time, one can see that the emf of the cell has been configured to be 4.5 V. This results in an effective current of around 0.136 A (3 dp) and power of 0.614 W (3 dp).

Below, one can see that the panel at the right is able to configure an accurate reading.



The mathematical reading behind this is configured by a simple algorithm. Here, once you click on a certain component, it goes towards both directions and checks for the possible direction of current, based on the number of volts facing in either direction. Here, many of these batteries cancel out one another as shown:



This leaves 3 batteries of 1.5 V each, compounding to a total of 4.5 V. There is a total of 33 resistors + bulbs, compounding to 33 ohms. This algorithm does this and then calculates the actual current flowing through it. It thus formulates the potential difference of every resistor / bulb and, from there, calculates the power intake. While this is a simplistic program, it took quite a lot of testing, which many trial and errors involved.



A Simplistic File System

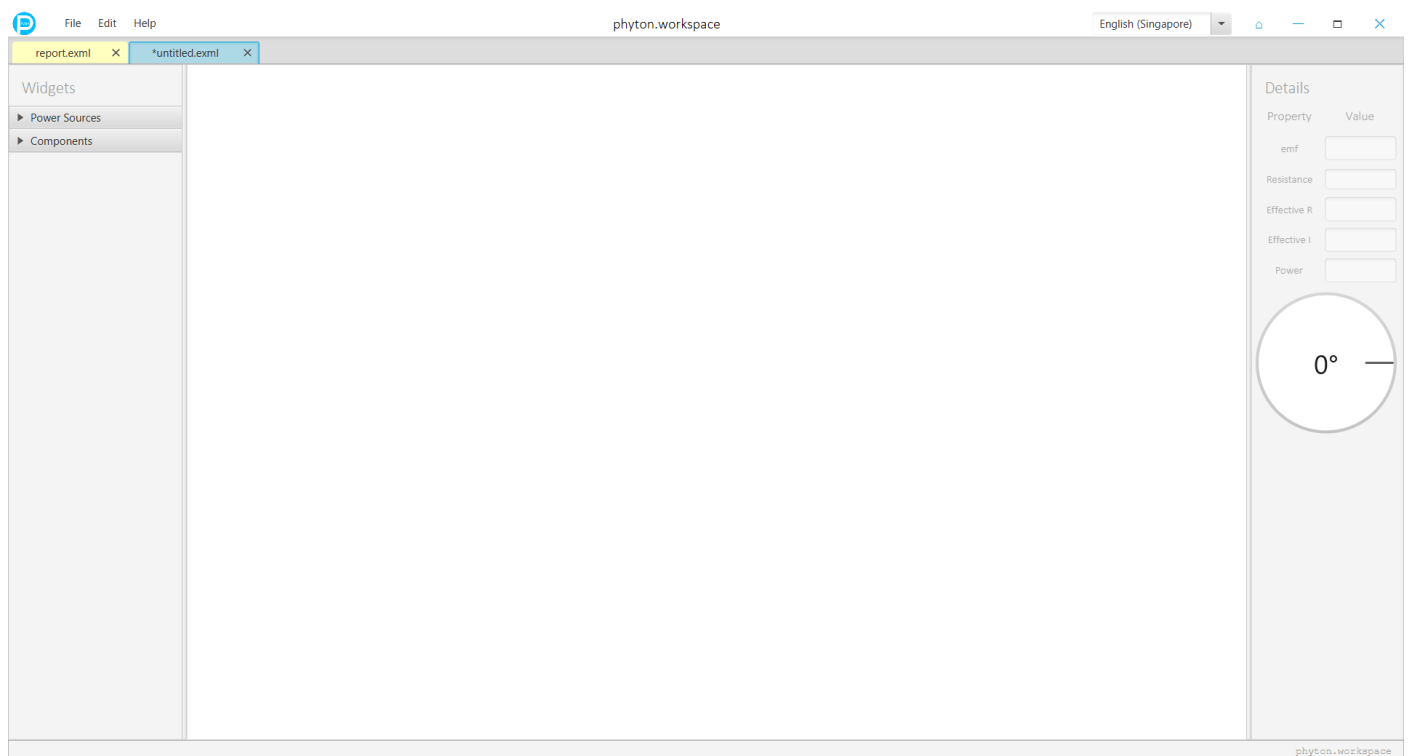
With this program, I implement a simplistic XML-based file system known as Electric XML, or EXML for short. This allows files to be saved in a tag-based layout.

For example, the simplified diagram above can be redrawn as the following:

```
<Series current="0.136A" effectiveR="33?" isClosed="true">
  <Cell P="0.614W" emf="4.5V" x="989.5703125" y="-715.2890625" angle="360.0" prev="c1" />
  <Resistor R="33Ω" I="0.136A" x="988.4263392854773" y="-829.5781249999854" angle="360.0" prev="c1" />
</Series>
```

There are New, Open, Save and Save As commands in the File Menu. Here is a basic overview of all the commands.

New

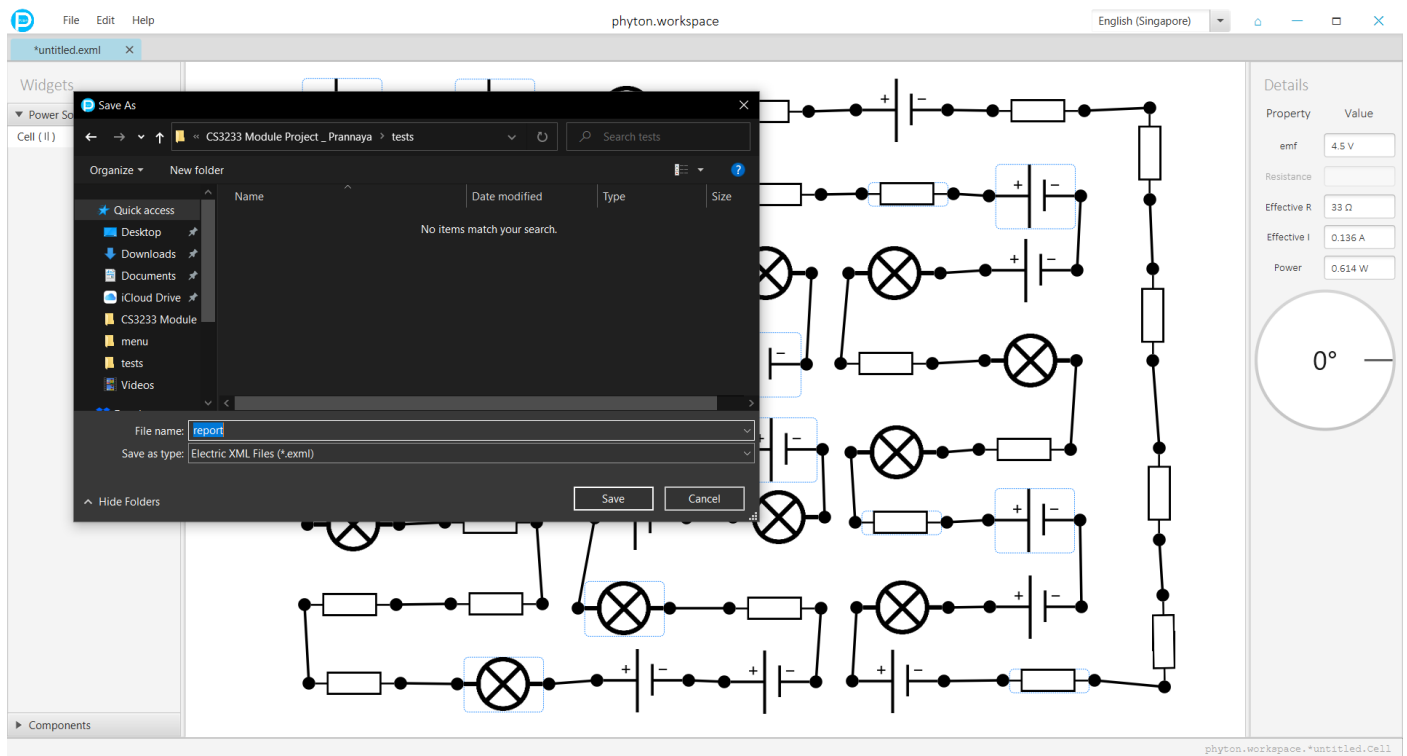


Here, I have clicked on Ctrl-N or the New MenuItem in the File menu and one can see an open *untitled.xml file, where there is currently no component. This is very clearly similar to the start-up screen, because, at the start-up, a new file is always autogenerated.

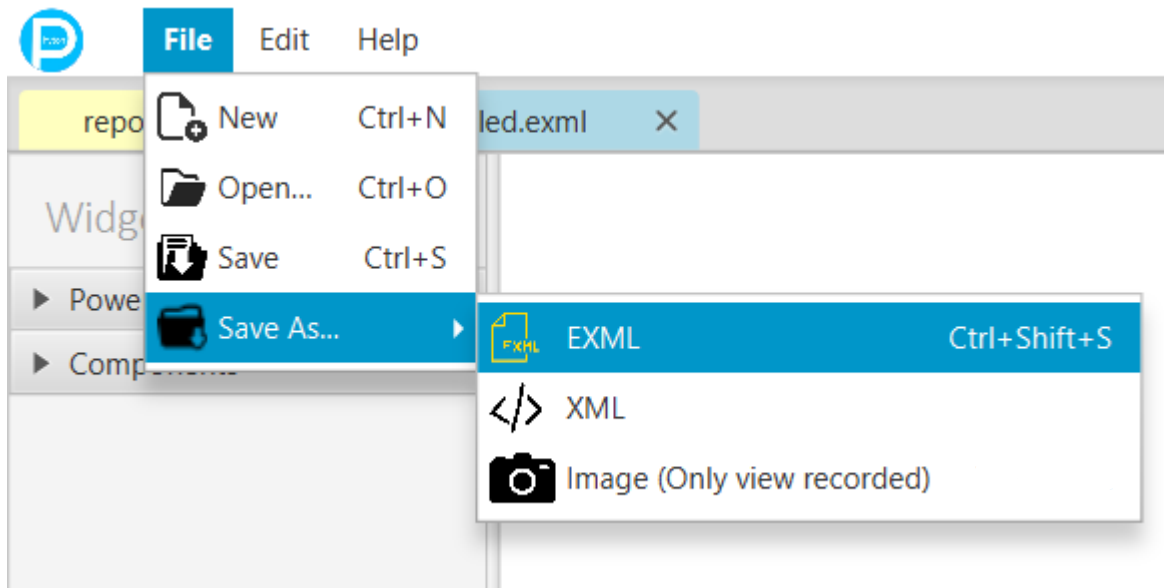
Open

To open any EXML file, you can simply press Ctrl-O and locate an EXML file. To avoid multiple editing, if an already currently opened file is opened, it will not be opened in a new tab.

Save & Save As

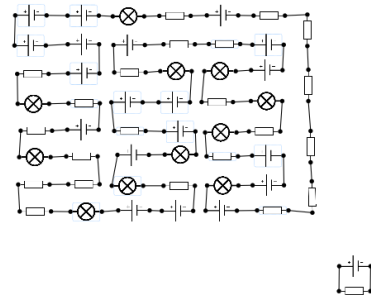


Here, you can see that I am saving the complex circuit diagram below in the currently unzipped version of the zipped folder, under the tests folder as report.exml. This is a very standard and easy-to-use saving system that is very normal in most IDEs that don't offer directory-based editing structures, e.g. the Python IDLE IDE.

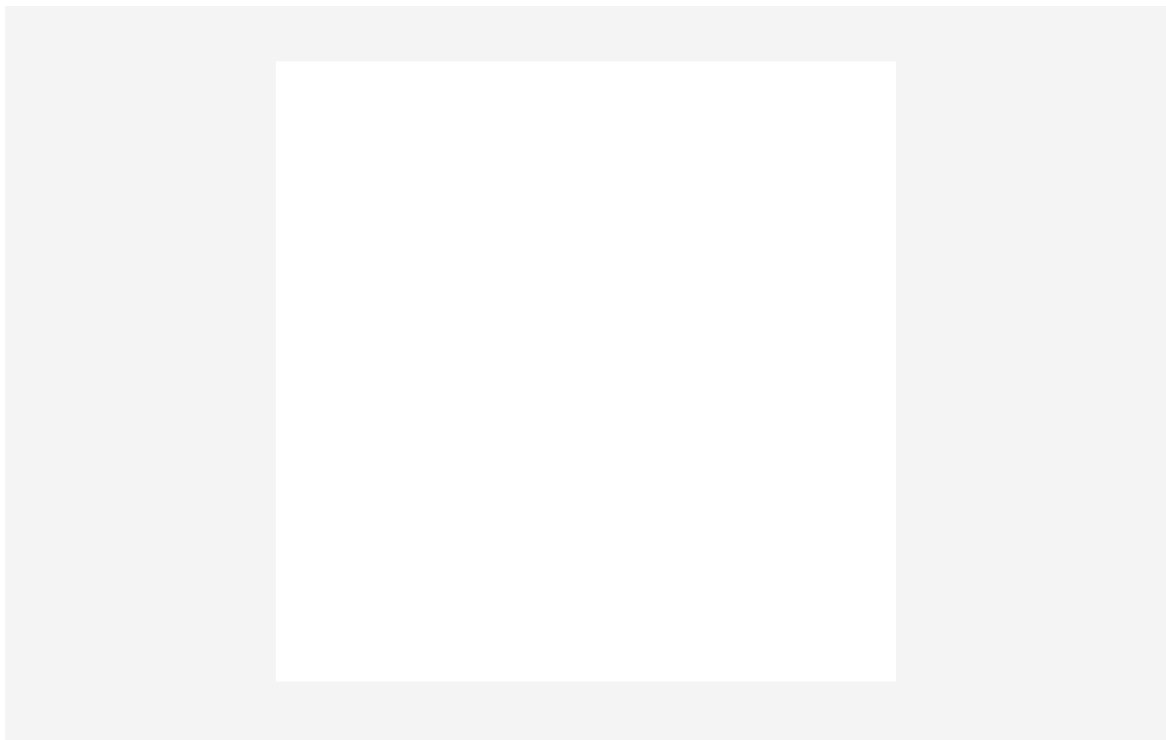


A bonus feature is that one can also save the file as a simple XML file, or as an image. Due to the use of a nearly infinite canvas, one cannot expect to save the entire pane as an image, which is why, as specifically stated above, one can save the view as an image. Unfortunately, in some cases, the circuit may extend beyond the current view, which is why I have implemented a feature that I believe will help you with this situation, and which I believe is much more helpful.

A Infinite, Zoomable, Pannable Canvas



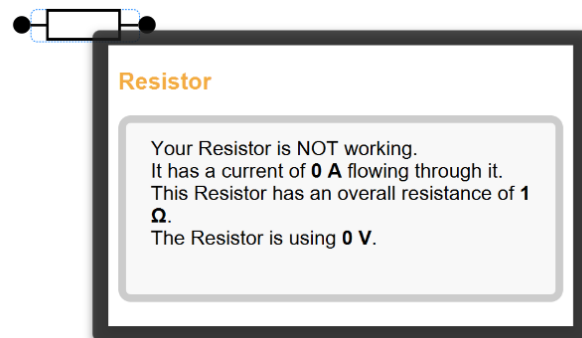
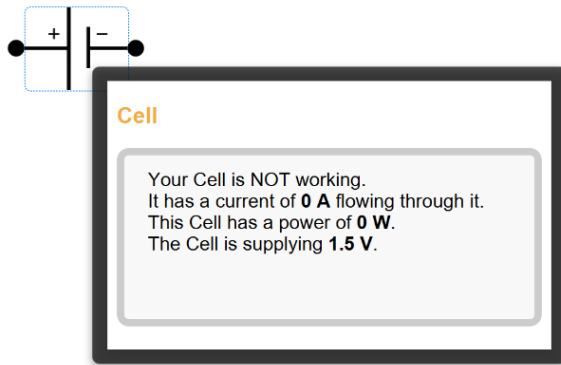
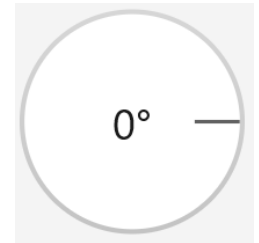
Above is a zoomed-out version of the same canvas you saw above. In this case, you can take a picture of both without much of an issue. The canvas can also be panned about, and it is almost infinite.



If you zoom out too much, you get this screen. While this proves that this pane is, in fact, not infinite, it also shows the degree to which the zoom occurs. Thus, you can get a very small or large view at a custom point and this especially makes it easier for teachers who may wish to create circuit diagrams on this. This also provides an avenue to do art with these circuits, since one can use these complex circuits to create a very interesting looking art piece, similar to Desmos Art.

Draggable and Rotatable Widgets with helpful Tooltips and many other features

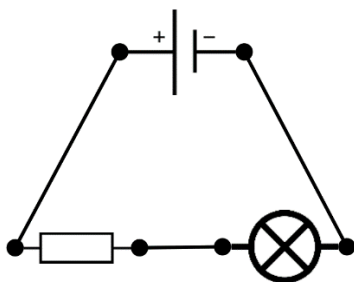
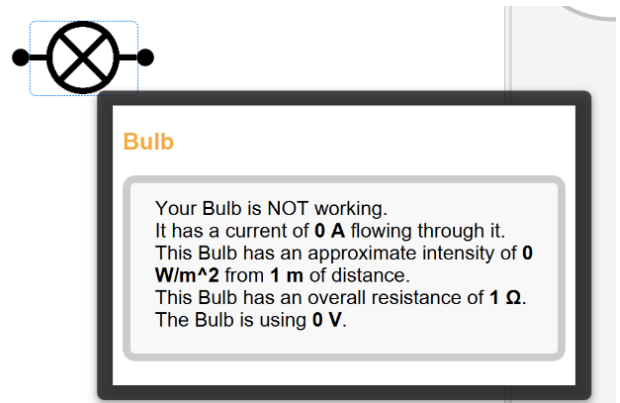
The widgets implemented are easily draggable and can be rotated using the angle picker at the side. The angle picker has been configured by me to start at positive x-axis, with the anti-clockwise direction being considered positive. I have also used a JavaFX WebView and a template html file to create a custom tooltip that can help one understand what exactly is going on.



All these tooltips provide general statements, like whether current is flowing through and what is the voltage it is supplying (Cell) or using (Resistor/Bulb). However, they also come with unique statements, like the power of the Cell and the intensity of the Bulb.

This feature was simply added to boost the person's understanding of how these components work.

The components also have a cut, copy, paste, duplicate and delete mechanism, which is gone into further detail ahead, using the example as follows



Cut

Upon cutting the bulb from the circuit above, it becomes as in the right:

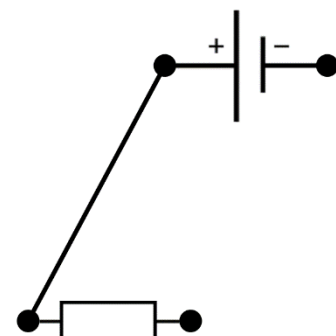
The text in the clipboard is below.

```
<Bulb I="0.75A" R="1Ω" x="365.15382106622565"
y="186.85504283111368" angle="360.0" prev="c1" />
```

Copy

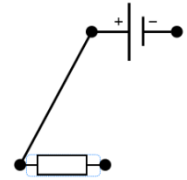
Now, if you copy the Resistor, it gives the following text:

```
<Resistor R="1Ω" I="0A" x="118.71205357147846" y="188.71874999992724" angle="360.0"
prev="c1" />
```



Paste

If pasted, it will produce the bulb widget around 300 pixels below the circuit to prevent overlap. It will not paste with the connections in order to prevent cases where, after removal, the person adds in another widget.



Duplicate

Duplicating the bulb from the original circuit will produce a similar result as in the left.

Delete

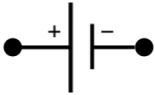
The same situation as cut will apply, but the clipboard will not contain the text above.

All these methods overlap with Cut calling Copy and Delete, Duplicate calling Copy and Paste within the actual code.



Documentation

About the Cell Widget



Cell

A cell is usually initiated with 1.5 V of emf. This is generally the normal EMF of a cell.


Multiple cells can be joined together to form an ideal battery. An ideal battery is such that it has no effective resistance, thus if such a cell is joined together with a simple wire, an infinite current will ideally be generated.

You can also change the EMF by simply going to the side panel and editing the EMF. You can either represent it in the form " n V", where n is the number of volts you want, or with " n kgm²/As³", which is the SI unit form of voltage. You can also simply enter " n " and the press enter and this will register it as " n V".

References:

<http://resources.schoolscience.co.uk/britishenergy/11-14/circh1pg2.html>

About the Bulb Widget




Bulb

A bulb is initiated with 1 ohm of resistance. While this is not generally the normal resistance of a bulb, a bulb is known to have varying resistances. Thus, it was simplified to 1 ohm.

The bulb can be connected in series with a cell to generate a current, GIVEN that the circuit is closed.

You can change the resistance by simply going to the side panel and editing the R property. You can either represent it in the form " n Ω ", where n is the number of ohms you want to set the bulb with, or with " n kgm²/s³A²", which is the SI unit form of ohms. You can also simply enter " n " and the press enter and this will register it as " n Ω ".

About the Resistor Widget



Resistor

A resistor is initiated with 1 ohm of resistance. While this is not generally the normal resistance of a resistor, a resistor is known to be very variable, if you get what I mean. Thus, it was simplified to 1 ohm.

The resistor can be connected in series with a cell to generate a current, GIVEN that the circuit is closed.

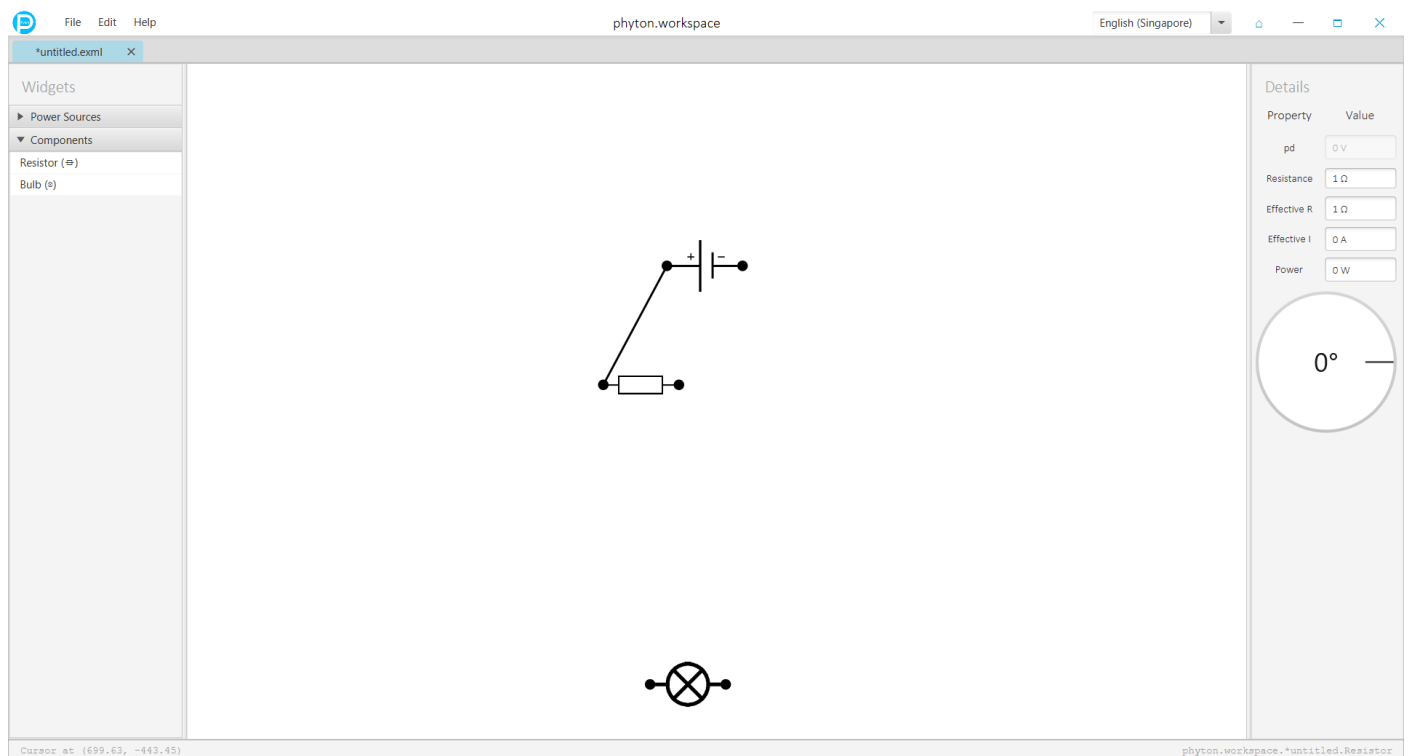
You can change the resistance by simply going to the side panel and editing the R property. You can either represent it in the form " n Ω ", where n is the number of ohms you want to set the bulb with, or with " n kgm²/s³A²", which is the SI unit form of ohms. You can also simply enter " n " and the press enter and this will register it as " n Ω ".

Phyton provides detailed documentation about all 3 widgets, as follows.

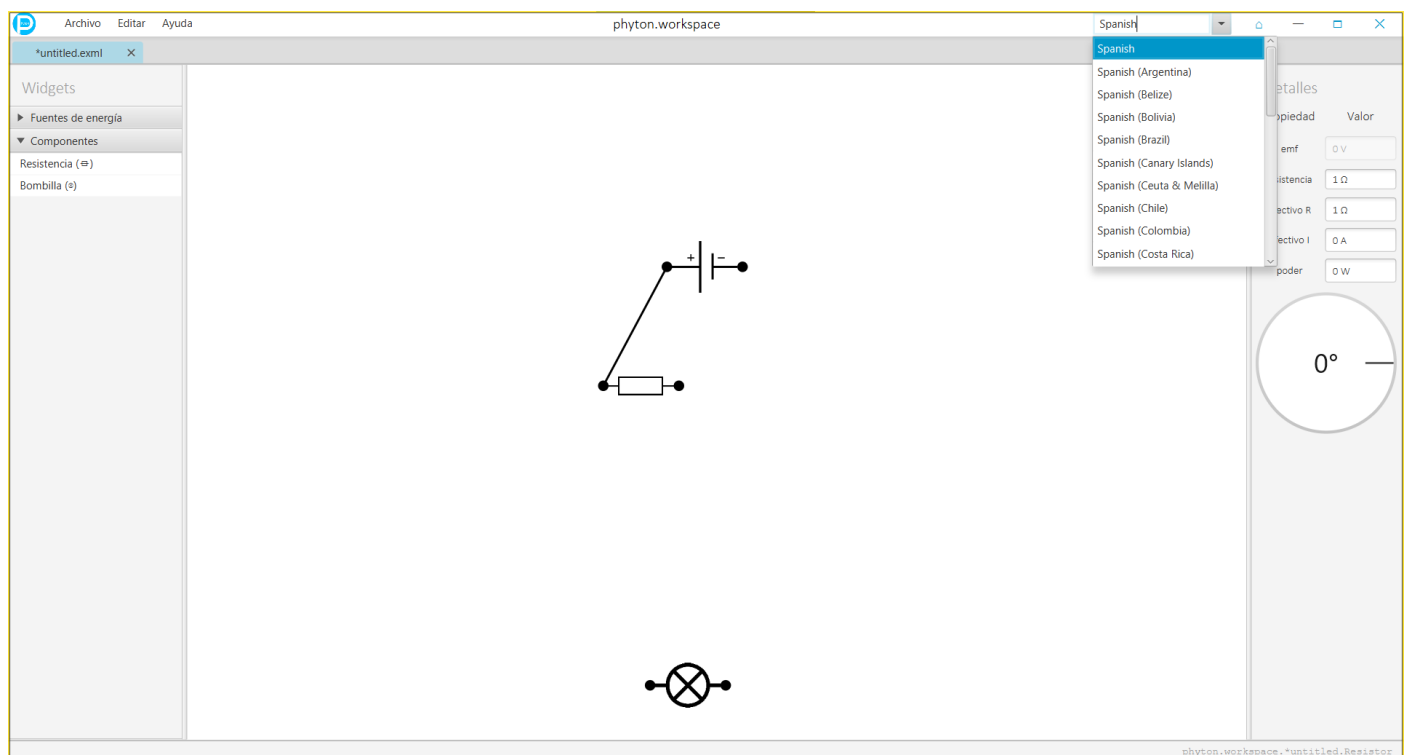
These pages provide detailed insights into each of the widgets, and, since they use a CircuitPane, one can move these about and test them. It also provides detailed instructions for how one can edit the emf (Cell) and resistance (Bulb, Resistor). It also gives reasonings behind each of the values, and, as for the cell, I have cited a source in a hyperlink which, if pressed, will open a page in the default webbrowser. For more details, you can check out the video.

Internationalization

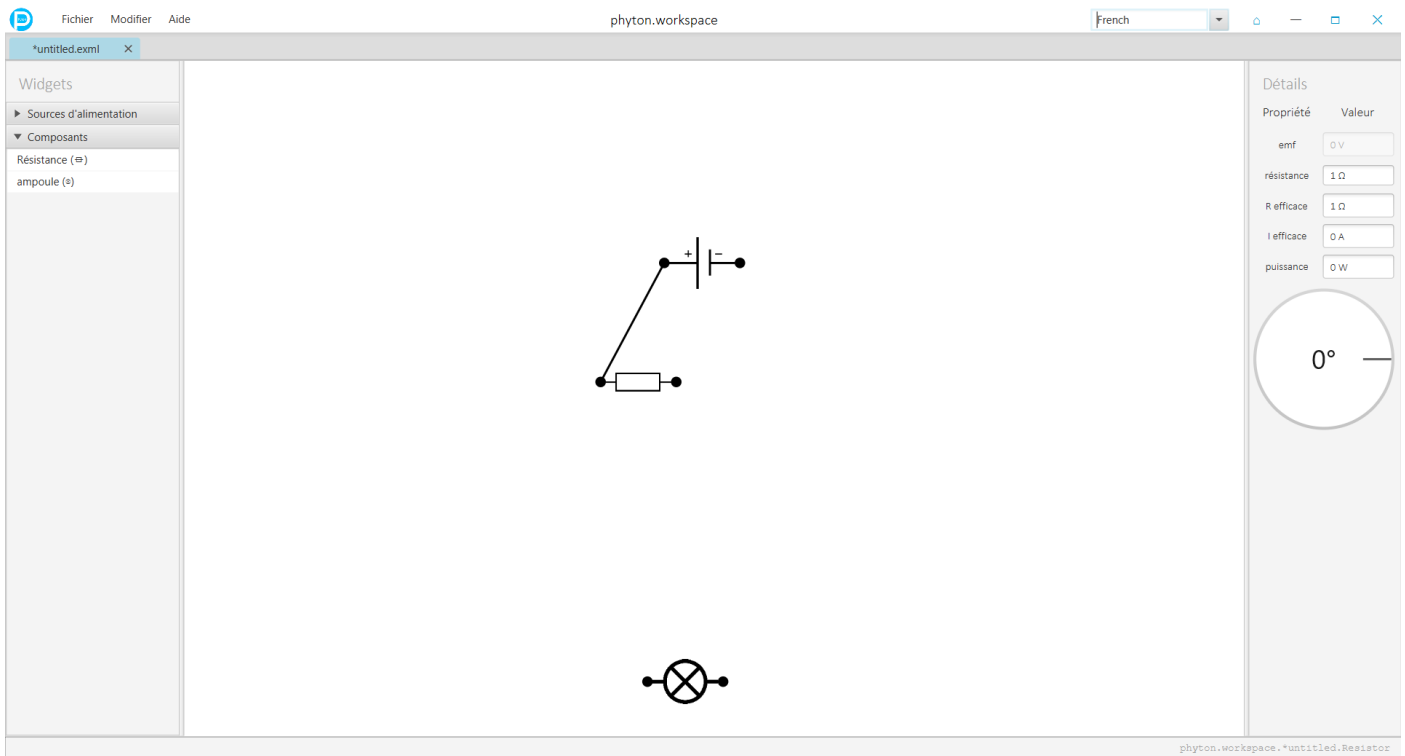
The application has been translated to Spanish, French and Hindi. With each language, the user interface tweaks slightly, and each language tweaks the entire application (besides the tooltip), not just the main UI. In fact, the about pages have also been configured to translate to Spanish, French and Hindi given the situation.



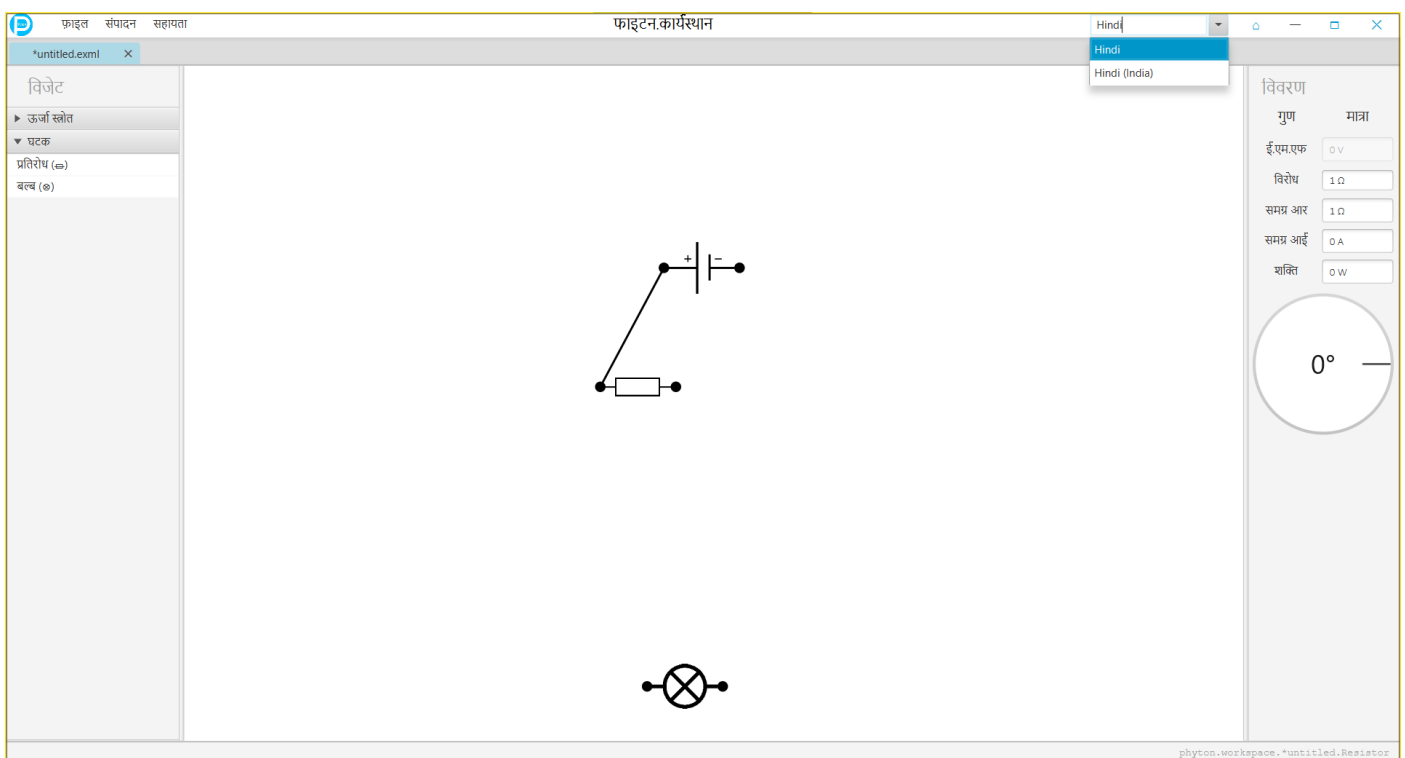
This is the standard English / English (Singapore) UI.



This is the Spanish UI.



This is the French UI.



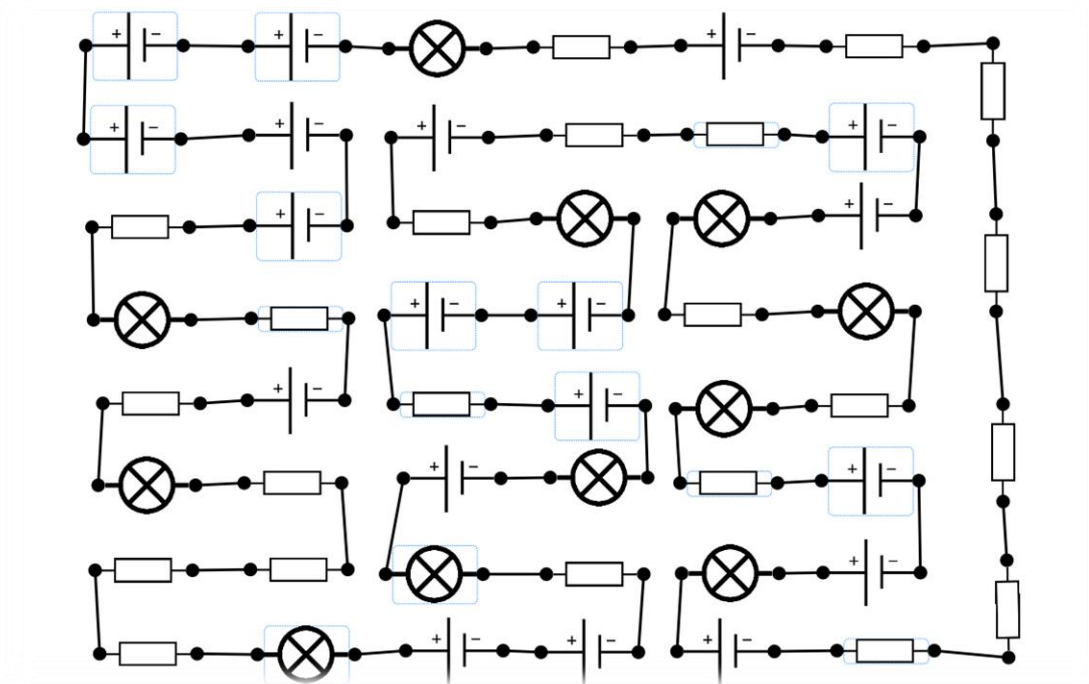
Finally, this is the Hindi UI. As you can see, this is the most drastic change, mainly because Hindi has its own complex set of characters.

These are many features that I implemented in this project.

Testing

There were many issues found in the code over a long period of time, primarily because I had to code this myself without any prior testing having been done. However, some testing that I did was as follows.

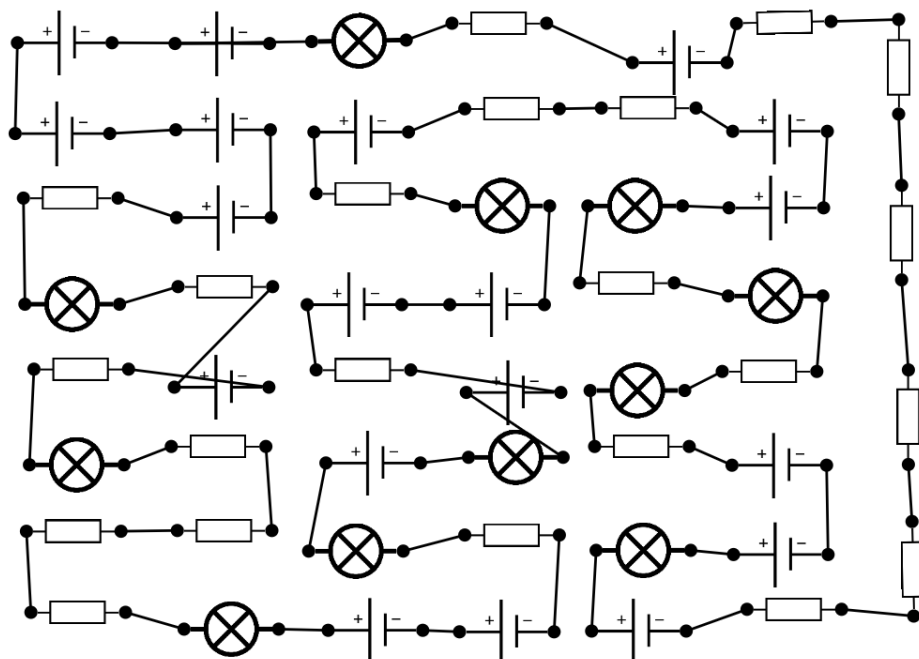
Issue 1: How large a circuit can this program handle?



I tested against this same circuit and the application ran smoothly. There were no bugs and the application didn't crash. However, after a few more widgets were added, the program started to have problems, mainly due to the excessive amount of calculations the program was being forced to do.

Issue 2: Does the file open properly?

In this case, I found a few issues. For one, with large circuits, the opening was mixing up the XML diagrams, and opening it up on its own took a long time. However, the final diagram was not a major deviation from the original. Additionally, because of all the information that the components needed, finding something as such was very time-consuming, mainly on very large diagrams. The final diagram looked like something as follows:



Reflection

- What are some obstacles faced?
- What have you learnt through the project?
- What could you have done better if more time was given?
- How could the task be improved?

What are some obstacles faced?

An obstacle that I had to face was the fact that there was no available API online or provided by JavaFX. Thus, I had to code from scratch which was incredibly difficult and time-consuming. The application also required the use of XML code, but I kept making mistakes in creating the xml files, and the strange way in which it was supposed to be created was quite annoying for me. I also had to focus a lot on many things, and I had a lot of issues in making a parallel circuit to the point that, due to time constraints, I had to leave it. This ultimately just served as a big roadblock for me. Another issue that I faced was the issue of my program being too large. This led to IntelliJ continually crashing due to it being unable to handle such a large application on my laptop's 8 GB RAM.

What have you learnt through the project?

I appreciate this project because I feel like I learnt a lot about how a real-life development of an application is usually formulated and this was my first hands-on opportunity to create such a large project of such a massive scale. I also learnt how xml is really written, and, since the only real XML experience I had was FXML in JavaFX, I never appreciated its relative power until now. I also learnt that there are much easier ways to code the same programs, and the importance of putting comments since I too lost myself many times. I have learnt that simulators are also very difficult to make, and I also learnt how to manage my time well such that I had time for CS as well as my other subjects. Truly, this was an enriching experience.

What could you have done if more time was given?

If I had more time, I could have implemented parallel circuitry, and followed by that I could implement capacitors, potentiometers, solenoids and the ground. These were some widgets that I had planned for from ahead, for example the reason for my distinction between CircuitObject and Component was because a potentiometer, in fact, has three nodes. I had also kept my classes open, and I believe that with more time, all this could have been implemented.

How could the task be improved?

The task, as I previously mentioned was very helpful for me, but I believe the main critique was the limitation of time. Nevertheless, I liked the fact that we could choose our own projects and am thankful to the CS teachers for doing this.