Summer Fellowship Report

on

Chemical PFD Tool

Rishikesh Anand

Under the guidance of

Prof. Kannan Moudgalya

Department of Chemical Engineering

FOSSEE, Python

IIT Bombay

Acknowledgement

I would like to thank the FOSSEE project from IIT Bombay for giving me an opportunity to for an internship in developing a full fledged Software, using python. The internship opportunity provided me to put my skills to the test all the while helping me to enhance my knowledge in the strict and method process of developing software, exploring libraries and framework, proper documentation, team work and general UI/UX.

I feel immense gratitude to have met so many wonderful people and professionals who guided me through this internship period. I would like to specially acknowledge Mr. Pravin Kumar Dalve with my deepest gratitude who mentored and guided me during this internship and also and provided enough value in discussing and considering my thoughts on various things, keeping me motivated. I consider this opportunity as a substantial milestone in my career that I desire to see grow.

I shall keep exploring, and improving with the skills and experience that I have obtained.

Contents

Chapter 1

Introduction to PFDs

A process flow diagram or PFD is a diagram commonly used in chemical and process engineering to indicate the general flow of plant processes and equipment. The PFD displays the relationship between *major* equipment of a plant facility and does not show minor details such as piping details and designations. Another commonly used term for a PFD is a *flowsheet*.

Process flow diagram - Wikipedia

1.1 Why?

While working on a chemical plant, it is extremely important to know beforehand the detailed *process* of the *flow* of the various components involved. A PFD is required for just that.

A PFD is a diagram representing the exact process flow, involved . It enables engineers to plan ahead, and make it easier to understand what needs to be done.

Chapter 2

Software Development using PyQt5

The Qt framework was originally developed to make it easier for developers to quickly implement GUI elements and not reinvent the wheel every time they start a new project. The original framework, was made for the C/C++ development environment, but with python allowing C/C++ extensions, and the quick development process of python, PyQt the python binding for the Qt framework was born

2.1 Why PyQt?

For our project, PyQt made the most sense, as most of what we needed, from displaying PFD elements to manipulating them, connecting them, along with additional GUI and Quality of Life features like Undo-Redo, multiple Diagrams etc. PyQt made the most sense, due to its extensive documentation, proper age, active development, and meaningful C/C++ to python code translation.

2.2 The FBS dev environment

Michael Herrmann developed the fman build system (fbs), to make it easier to continuously develop his file manager program called fman, without worrying about having to configure installers and binary compilation procedures every time a new version was released. The beauty of fbs is that it

keeps the code structure clean, all the while allowing easy compilation procedure for each os distribution. We decided to adopt fbs early, as sooner or later we would have seen the benefit of using it when we decided to ship binaries. As a bonus, even python let alone the other dependencies are not required to be pre-installed for the user to use the "*compiled*" binaries!

2.3 Licensing

The software is planned to be licensed under the GNU GPL, as both PyQt5 and FBS are served under the same license, which means the software will remain free to use and distribute and any products derived from this must also be open sourced as well.

2.4 PySide2

With the more active development of PySide2, better documentation and more lenient licensing (PyQt vs PySide2...), there are plans to switch to PySide2 instead. For now, as there no conclusive differences between PyQt5 and PySide2. It is safe to stick to PyQt5 for now.

Chapter 3

The Tool

The main goal of our tool is to remain intuitive, easy to use yet as feature complete as possible. We were to focus on designing the UI in very much the same way.

I was tasked with working on the UI, and my main source reference were the adobe suite. As an experienced After FX user, I understand the importance of procedural components and have done my best to implement the same design idea.

3.1 The UI

The UI consists of the multiple diagram and files the user may work on, along with all the components needed to provide the array of features

3.1.1 Canvas Dimensions

We began with a simple *QGraphicsScene* used to display the PFD elements in. It was important that the scene had a fixed width and height in accordance to the various paper sizes available for example from A0 to A4.

It is important to note that these paper sizes are in human scales and not pixel dimensions. As such, it is important to convert them to accurately depict the paper size. For the conversion, A value knows as **Pixels Per Inch** or **PPI** is used to calculate the pixel dimension. Below is a table showing the exact values for the various paper sizes

Size	Dim(mm)	72 ppi	96 ppi	150 ppi	300 ppi
A0	841 x 1189 mm	2384 x 3370	3179 x 4494	4967 x 7022	9933 x 14043
A1	594 x 841 mm	1684 x 2384	2245 x 3179	3508 x 4967	7016 x 9933
A2	420 x 594 mm	1191 x 1684	1587 x 2245	2480 x 3508	4960 x 7016
A3	297 x 420 mm	842 x 1191	1123 x 1587	1754 x 2480	3508 x 4960
A4	210 x 297 mm	595 x 842	794 x 1123	1240 x 1754	2480 x 3508

So the first plan of action was to implement this! We built a look up table as a dictionary in python, and saved it as a json file.

```
{
    "A0": {
        "72": [2384, 3370],
        "96": [3179, 4494],
        "150": [4967, 7022],
        "300": [9933, 14043]
    },
    "A1": {
        "72": [1684, 2384],
        "96": [2245, 3179],
        "150": [3508, 4967],
        "300": [7016, 9933]
    },
    "A2": {
        "72": [1191, 1684],
        "96": [1587, 2245],
        "150": [2480, 3508],
        "300": [4960, 7016]
    },
    "A3": {
        "72": [842, 1191],
        "96": [1123, 1587],
        "150": [1754, 2480],
        "300": [3508, 4960]
    },
    "A4": {
        "72": [595, 842],
        "96": [794, 1123],
        "150": [1240, 1754],
        "300": [2480, 3508]
    }
}
```

Inside the canvas class, the properties for the size and ppi was implemented, and a setter method was created to set the dimensions of the scene whenever they were changed.

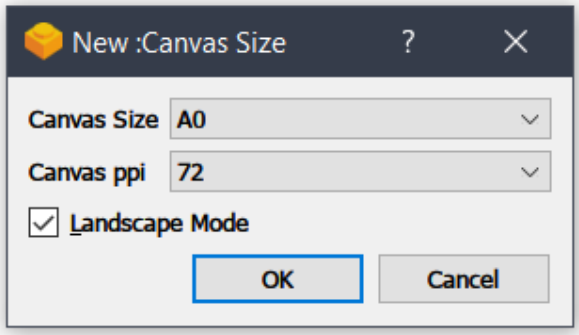
Also note that the landscape feature was implemented later on, and a

simple *hack* was used.

```
@property
def canvasSize(self):
    return self._canvasSize

@canvasSize.setter
def canvasSize(self, size):
    self._canvasSize = size
    if self.painter:
        self.resizeView(*(sorted(paperSizes[self.canvasSize][self.ppi],
        reverse = self.landscape)))
```

A dialog box was also added to allow the user to change between available sizes



All of this is done progressively. To add more paper sizes, update the json file, and the program will load it upon execution.

3.1.2 Multiple Diagrams and Files

Our next feature target was enable the user to be able to edit multiple diagrams at once. The solution was simple, create multiple instance of our canvas object!

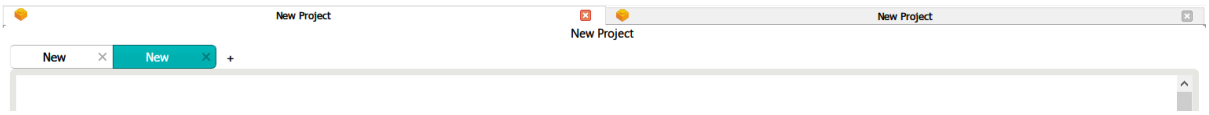
For this we looked into *QTabWidget*, where each tab consisted of a unique self complete canvas.

To make creating new tabs easier, we built a chrome like new tab button, from scratch!

It was to move to the appropriate location whenever the tab count changed, and we implemented just that.

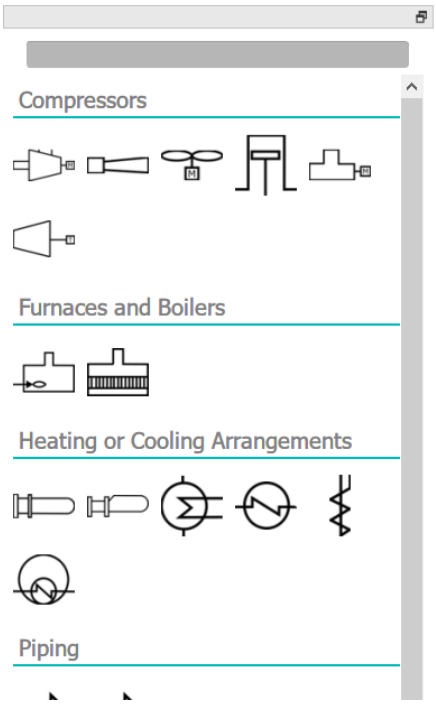
```
def movePlusButton(self):
    #move the new tab button to correct location
    size = sum([self.tab.tabRect(i).width() for i in range(self.tab.count())])
    # calculate width of all tabs
    h = max(self.tab.geometry().bottom() - self.plusButton.height(), 0)
    #align with bottom of tabbar
    w = self.tab.width()
    if size > w: #if all the tabs do not overflow the tab bar, add at the end
        self.plusButton.move(w-self.plusButton.width(), h)
    else:
        self.plusButton.move(size+5, h)
```

Next was abstracting the tab widget as a file. We used QMdiSubWindow along with a QMdiArea. An mdi sub-window is a free window movable inside the app itself, to be exact, in the mdi area.



3.1.3 The toolbar

To build a non-obtrusive toolbar that the user could move at will, two approaches were considered, One being *QToolBar* and the other being *QDockWidget*, the latter made the cut, due to it requiring a child widget as such all properties of the contents of the toolbar were easily sand boxed inside the child widget without implementing additional behavior on the toolbar widget itself, the float-ability and look of the dock widget as just an added bonus.



Item list

The toolbar consisted of a search box and an area that contained all the elements to be added to the scene. The scroll area is a *QScrollArea* that contains a *QWidget*. A custom layout was implemented and assigned to the widget, so that the items are first arranged width wise, and when no more width is available the widget is placed in the immediate next line, and the process repeats until no more widgets can be placed. This is one of the example in Qt called, flow layout and was tweaked for our use case.

Search bar

For the search box, the items in the toolbar need to be filtered out as per the search query. When the search query is empty, the dict.keys() is passed as is, otherwise a filter object is sent. The filter object is built with an anonymous function with regex search from the re library, and dict.keys() iterable. This allows seamless search queries without over complicating things.

```
def populateToolbar(self, filterFunc=None):
    #called everytime the button box needs to be updated(incase of a filter)
    self.clearLayout() #clears layout
    for itemClass in self.toolbarButtonDict.keys():
        self.diagAreaLayout.addWidget(self.toolbarLabelDict[itemClass])
        for item in filter(filterFunc, self.toolbarButtonDict[itemClass].keys()):
            self.diagAreaLayout.addWidget(self.toolbarButtonDict[itemClass][item])
    self.resize()

def searchQuery(self):
    # shorten toolbaritems list with search items
    # self.populateToolbar() # populate with toolbar items
    text = self.searchBox.text() #get text
    if text == '':
        self.populateToolbar() # restore everything on empty string
    else:
        # use regex to search filter through button list and add the remainder
        # to toolbar
        self.populateToolbar(lambda x: search(text, x, IGNORECASE))
```

Toolbar buttons

The toolbar items contains of two things, a category label and buttons for each pfd symbol, to avoid using useless space, a json file is prepared, consisting of the categories and the corresponding items.

```
{
    "Compressors": {
        "Centrifugal Compressor": {
            "name": "Centrifugal Compressor",
            "icon": ".\\Compressors\\Centrifugal Compressor.png",
            "class": "Compressors",
            "object": "CentrifugalCompressor",
            "args": []
        },
        "Ejector Compressor": {
            "name": "Ejector Compressor",
            "icon": ".\\Compressors\\Ejector Compressor.png",
            "class": "Compressors",
            "object": "EjectorCompressor",
            "args": []
        },
        ...
    },
    "Furnaces and Boilers": {
        "Oil Gas or Pulverized Fuel Furnace": {
```

```
            "name": "Oil Gas or Pulverized Fuel Furnace",
            "icon": ".\\Furnaces and Boilers\\Oil Gas or Pulverized Fuel Furnace.png",
            "class": "Furnaces and Boilers",
            "object": "OilGasOrPulverizedFuelFurnace",
            "args": []
        },
        ...
    },
    ...
```

The first set of keys defines the set of categories and is used to prepare labels, while the inner set of keys represent the items, their class info, icon etc. and is used to prepare a list of *QToolButtons* that have their logo set to the icon value from the dictionary.

Button click and Drag & Drop

The final thing was to implement the button click event as well as drag and drop.

The button click event was simple, as qt provides button click events itself.

However, to implement drag and drop, there were two objectives,

1. Differentiate between button click and drag-drop intents

   This is done by checking if the mouse has moved the equivalent of something known inside qt as *the Manhattan length* from the button with the mouse pressed, if it has the drag and drop intent is registered otherwise button press intent is registered.

```
    def mousePressEvent(self, event):
        #check if button was pressed or there was a drag intent
        super(toolbarButton, self).mousePressEvent(event)
        if event.button() == Qt.LeftButton:
            self.dragStartPosition = event.pos() #set dragstart position

    def mouseMoveEvent(self, event):
        #handles drag
        if not (event.buttons() and Qt.LeftButton):
            return #ignore if left click is not held
        if (event.pos() - self.dragStartPosition).manhattanLength() <
            app.app.startDragDistance():
            return
```

```
            #check if mouse was dragged enough, manhattan length is a rough
            #and quick method in qt

        -- drag and drop code --
```

2. Implement Drag and Drop itself

   To do this, we must first understand about mime data,

   Now known as media type, mime data is a two part identifier used to identify the data type.

   While dragging and dropping, the object name is attached as text to *QMimeData* which is then attached to the mouse by using *QDrag* . The canvas accepts mime drops, and if a valid class is dropped to the scene it creates an object with the class name and then sets is position at the drop position.

Create drop

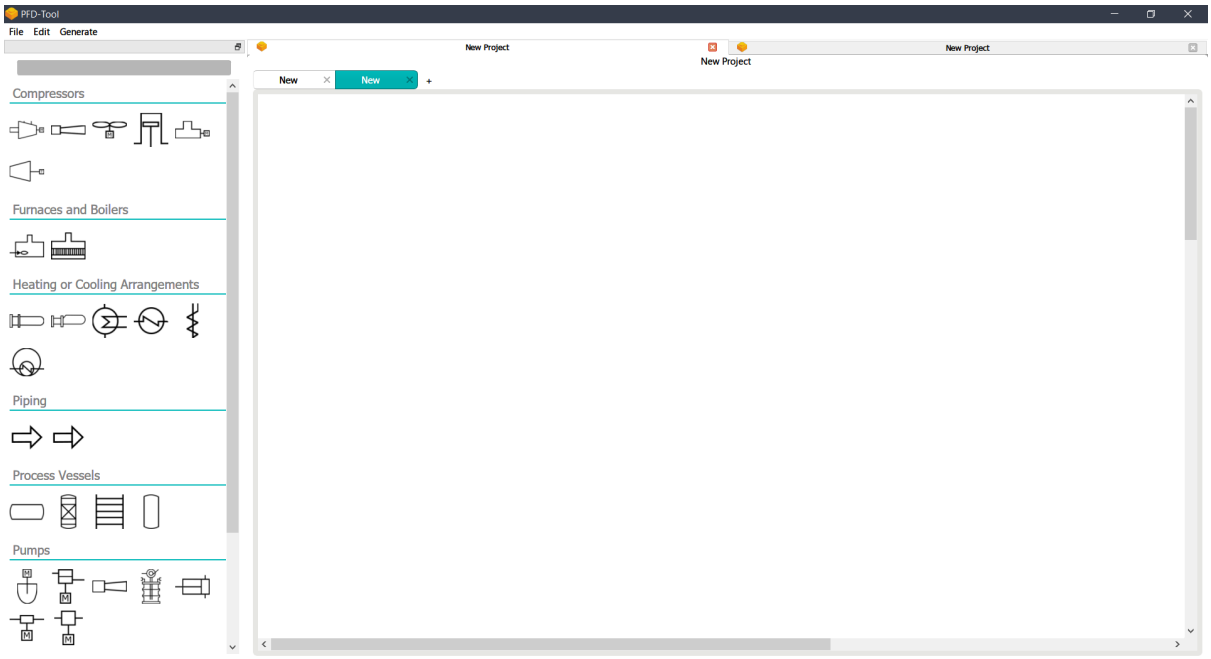
```
drag = QDrag(self) #create drag object
mimeData = QMimeData() #create drag mime
mimeData.setText(self.itemObject) # set mime value for view to accept
drag.setMimeData(mimeData) # attach mime to drag
drag.exec(Qt.CopyAction) #execute drag
```

Accept drop

```
def dropEvent(self, QDropEvent):
    #defines item drop, fetches text, creates corresponding QGraphicItem and adds it to
    #scene
    if QDropEvent.mimeData().hasText():
        #QDropEvent.mimeData().text() defines intended drop item, the pos values define
        #position
        obj = QDropEvent.mimeData().text().split('/')
        graphic = getattr(shapes, obj[0])(*map(lambda x: int(x) if x.isdigit() else x,
                                               obj[1:]))
        self.scene().addItemPlus(graphic)
        graphic.setPos(QDropEvent.pos().x(), QDropEvent.pos().y())
        QDropEvent.acceptProposedAction()
```

3.1.4 The main window

The main window is built using *QMainWindow*, along with a central mdi area. A menu bar is built with actions for creating a new project, saving the project and opening a new one.

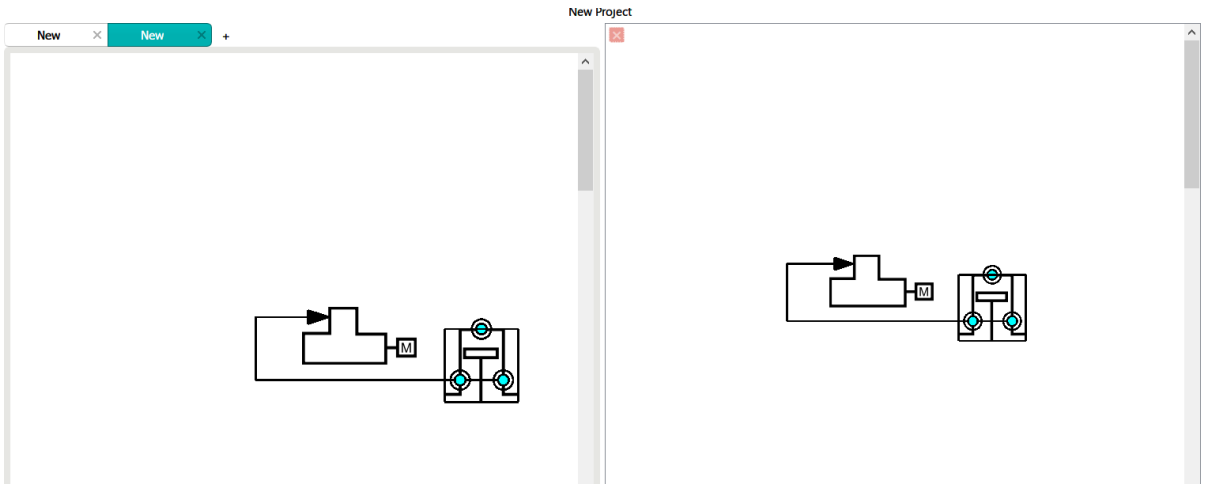


The save and open file windows are made using *QFileDialog*'s methods, *getSaveFileName* and *getOpenFileNames* respectively.

Multiple files can be opened at once. But a separate save file dialog will be opened for each file open currently.

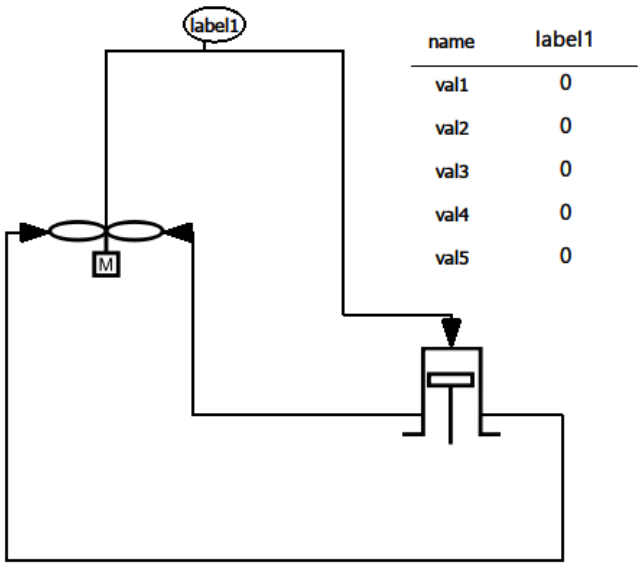
3.1.5 Side View

More of a quality of life feature, A side view was implemented to have a persistent diagram on screen, for a file. It is added to the *QMdiSubWindow*, at the request of the user, and can be removed as required, and even allows free switching either by right clicking any canvas and selecting view side-by-side, or right clicking the side view itself to open a separate dialog box to switch views.



3.1.6 Stream Table

A late addition, a stream table defining the various details about a diagram was added by adding a customized *QTableView* to the *QGraphicsScene* using a *QGraphicsProxyWidget* with a moveable parent rect Item to allow moving the diagram freely inside the scene. The stream table can be added to a scene by simply right clicking the scene



3.2 More Quality of Life features

3.2.1 Zoom in/out

A very important feature while working with diagrams. It is implemented by using the *setScale* method on the *QGraphicsView* , with the scale amount equivalent to the degrees scrolled in from the scroll wheel. Moreover, It works with track pad zoom-in gestures that come on laptops and supports high-precision mice thanks to qt.

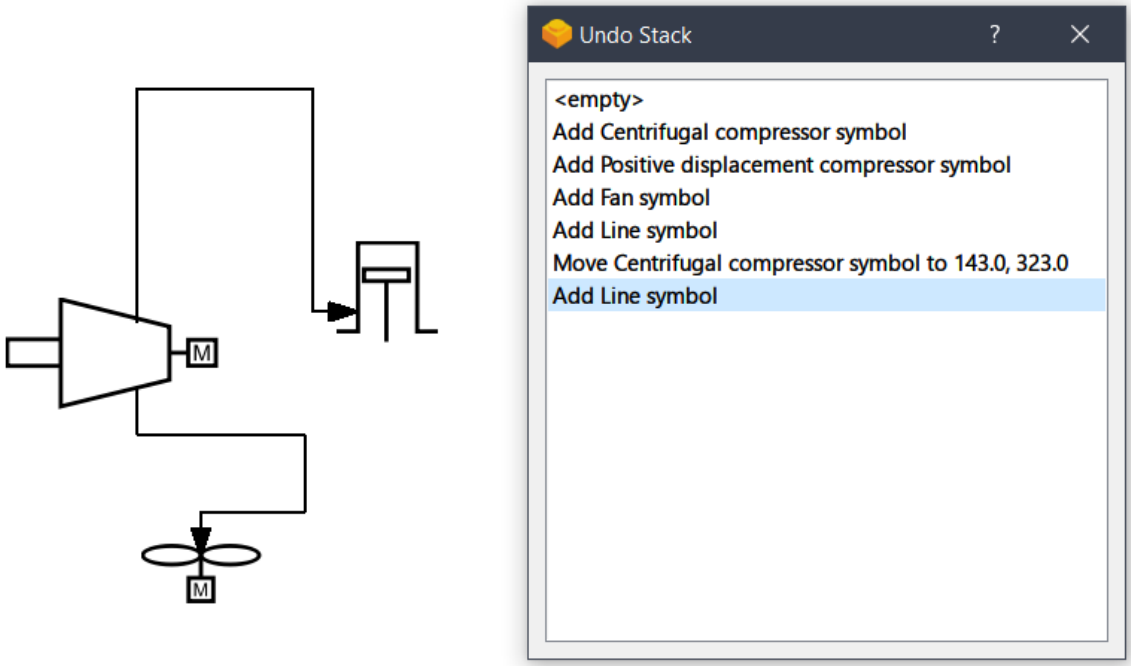
```
def wheelEvent(self, QWheelEvent):
    #overload wheelevent, to zoom if control is pressed, else scroll normally
    if QWheelEvent.modifiers() & Qt.ControlModifier: #check if control is pressed
        if QWheelEvent.source() == Qt.MouseEventNotSynthesized: #check if precision
        #mouse(mac)
            # angle delta is 1/8th of a degree per scroll unit
            if self.zoom + QWheelEvent.angleDelta().y()/2880 > 0.1:
            # hit and trial value (2880)
                self.zoom += QWheelEvent.angleDelta().y()/2880
        else:
            # precision delta is exactly equal to amount to scroll
            if self.zoom + QWheelEvent.pixelDelta().y() > 0.1:
                self.zoom += QWheelEvent.angleDelta().y()
        QWheelEvent.accept() # accept event so that scrolling doesnt happen
        #simultaneously
    else:
        return super(customView, self).wheelEvent(QWheelEvent) # scroll if ctrl not
        #pressed
```

3.2.2 Undo-Redo

Using a *QUndoStack*, a user can simply use **Ctrl+Z**, **Ctrl+Y** to undo or redo the last action.

Also added is a stack view, that allows the user to shift through multiple actions at once, while maintaining the consistency of the items.

The main thing here is that all possible undo-redo actions need to be implement as a subclass of *QUndoCommand* as part of the entire *QUndoFramework*. Whenever a new undo-able is performed it is to be executed by pushing to the undo stack instead.



Current, there are sub-classes for resizing, moving, adding and deleting items along with adjusting the canvas dimensions.

3.2.3 Saving and Loading project

The most critical and hardest feature to implement, this took quite the development time to implement.

The original idea was to use pickle to save the projects in a file with .pfd format, but the decision was made to switch to json so that converting to future versions is much easier, since a json file can easily be viewed in a simple text editor.

Its important to note, that what we have implemented is not true saving or loading, as rather than saving the state of the items we write information about how to recreate the items from scratch.

Furthermore, it is important that before saving and after loading, the object references remain consitent, as it is impossible to recreate information at the same memory location, writing pointers to json would be pointless. Instead, what we do is store references in the forms of id's and instead of generating a new id for every object we use the hex value of its memory location.

This allows us to store references and when loading recreate the refrences by preparing a lookup dictionary with the id as the key and reference pointer as the value.

Here is how objects are loaded, inside the canvas.

```
def __setstate__(self, dict):
    self._ppi = dict['ppi']
    self._canvasSize = dict['canvasSize']
    self.landscape = dict['landscape']
    self.setObjectName(dict['ObjectName'])

    for item in dict['symbols']:
        graphic = getattr(shapes, item['_classname_'])()
        graphic.__setstate__(dict = item)
        self.painter.addItem(graphic)
        graphic.setPos(*item['pos'])
        graphic.updateLineGripItem()
        graphic.updateSizeGripItem()
        for gripitem in item['lineGripItems']:
            shapeGrips[gripitem[0]] = (graphic, gripitem[1])
        if item['label']:
            graphicLabel = shapes.ItemLabel(pos = QPointF(*item['label']['pos']),
                                            parent = graphic)
            graphicLabel.__setstate__(item['label'])
            self.painter.addItem(graphicLabel)

    for item in dict['lines']:
        line = shapes.Line(QPointF(*item['startPoint']), QPointF(*item['endPoint']))
        lines[item['id']] = line
        line.__setstate__(dict = item)
        self.painter.addItem(line)
        graphic, index = shapeGrips[item['startGripItem']]
        line.startGripItem = graphic.lineGripItems[index]
        graphic.lineGripItems[index].line = line
        if item['endGripItem']:
            graphic, index = shapeGrips[item['endGripItem']]
            line.endGripItem = graphic.lineGripItems[index]
            graphic.lineGripItems[index].line = line
        else:
            line.refLine = lines[item['refLine']]
            lines[item['refLine']].midLines.append(line)
            line.refIndex = item['refIndex']
        for label in item['label']:
            labelItem = shapes.LineLabel(QPointF(*label['pos']), line)
            line.label.append(labelItem)
            labelItem.__setstate__(label)
            self.painter.addItem(labelItem)
        line.updateLine()
        line.addGrabber()
    shapeGrips.clear()
    lines.clear()
```

and Here is how objects are serialized to be saved as a dict. The following example is of a pfd symbol on the scene.

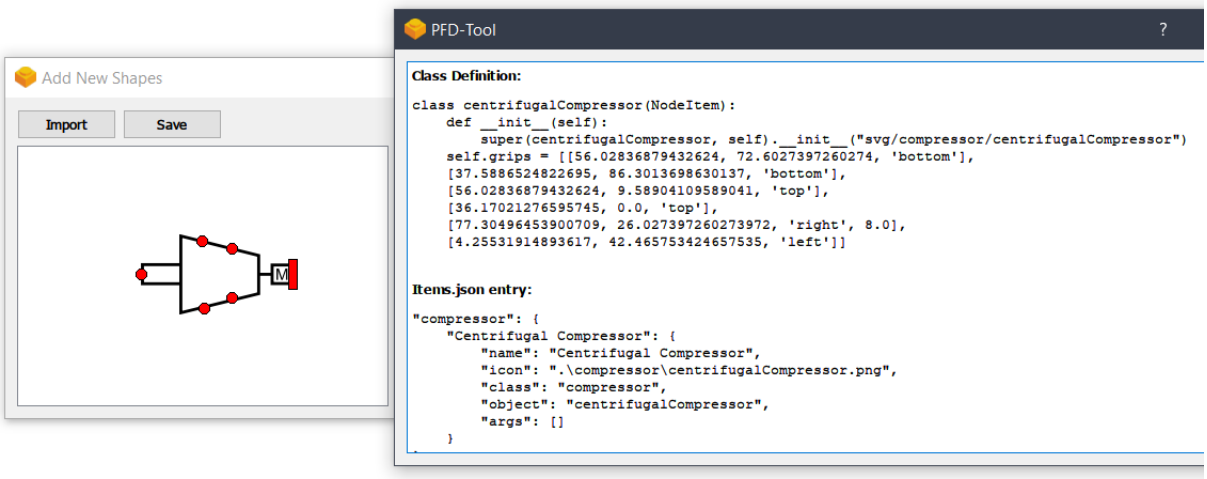
```
def __getstate__(self):
    return {
        "_classname_": self.__class__.__name__,
        "width": self.width,
        "height": self.height,
        "pos": (self.pos().x(), self.pos().y()),
        "lineGripItems": [(hex(id(i)), i.m_index) for i in self.lineGripItems],
        "label": self.label
    }
```

3.2.4 Custom Symbol tool

Built right into the tool, and available separately, is a simple dialog box to visually place grip items over an svg file, create the necessary class definition, items.json entry and the toolbar icon.

The following template is used, and rendered out as html on a QTextEdit window.

```
output = OutputBox(temp, f"""
<b> Class Definition:</b>
<pre>
class {className}(NodeItem):
    def __init__(self):
        super({className}, self).__init__("
                    svg/{category}/{str.split(name[0], "/")[-1][:-4]}")
    {grips}
</pre>
<b> Items.json entry:</b>
<pre>
"{category}": {{
    "{itemName}": {{
        "name": "{itemName}",
        "icon": ".\\{category}\\{str.split(name[0], "/")[-1]}",
        "class": "{category}",
        "object": "{className}",
        "args": []
    }}
}}</pre>""")
```



3.3 Styling the UI

It is so very important that the UI looks appealing, and is not obnoxious for a user to use. Therefore, a Qt Style Sheet, or QSS was written from scratch. Specific properties like the color palette, component sizes and behavior on interaction, along with the minor details such as the round edges of corner edges was implemented. QSS syntax is very similar to web CSS , and experience in web-dev was very useful.

app.qss

src > main > resources > base > app.qss > ...

```
47  tabBarPlus:focus {
48      border: 0px transparent black;
49  }
50
51  tabBarPlus::close-button {
52      padding: 0px;
53      margin: 0px;
54      border-radius: 2px;
55      background-image: url("src/main/resources/base/ui/close.png");
56      background-position: center center;
57      background-repeat: none;
58  }
```

Chapter 4

Conclusion

On the whole, this internship was a very fruitful experience. I have gained new knowledge, skills and have also met a few new people. I realized the importance of discussion and feedback, analytic thinking and team work. I have achieved several learning goals, and have moved a step further in achieving a few other. I got insight into professional practice, and it has allowed as an opportunity to get an exposure of the practical implementation of theoretical fundamentals of software engineering. During this internship I gained a solid grasp on the following tools/software/services,

1. Version Control using GIT
2. UI development using QT framework
3. The nuances of python, one-liners and proper class properties
4. Implementing feature branches using Github, along with PRs and Issue tracker
5. Slack, Hangouts and Google Meet for communication

Furthermore, I found that several things are important during the development of software in a professional environment,

4.1 Critical and Analytical Thinking

Whenever a new feature is to be implemented, it is critical to think about the various approaches one could take, what each approach might affect to whatever is already implemented, how new feature friendly will it be

in the future, what are the pros and cons of each approach, what is the resource cost both in development time and computation time and if there is scope for optimization. Everyone likes to have a feature complete tool, but not at the expense of minor annoyances, like long wait times, sluggish behavior or straight up broken components.

4.2 Time Management

Managing time is another important aspect as unlike a casual environment, there is always a deadline to meet which means that one cannot take huge amounts of time just exploring and a compromise must most of the times be made. It is useful to not stick to a single approach if it leads to bugs that require additional time to squash.

4.3 Intuitive thinking

Components of the program should be procedural, and not case based. UI components should have a central information pool, and should scale automatically with additional data added to the pool, without manual input. This can be seen from my implementation of the toolbar, where the list of buttons is pulled from a json file which then populates the toolbar, without manual intervention. As for algorithms, they should have human like thinking logic instead of a case based implementation leading to a spaghetti of if else statements, this prevents having to implement every possible case which in some situations might not even be possible.

4.4 Goal management

It is important to divide a goal to sub tasks the need to be completed first in order to achieve that goal, it allows one to keep track and even stay motivated while working on something.

4.5 Documentation

While not of use for the user, as part of open source, it is a developer's duty to properly document and comment their code, so that if someone were to improve on the project, they don't have to waste precious development

time in understanding the code base. Documentation is critical even for tools and not just frameworks, as it specifies what each component does and maybe even how.