Widget Application

Joshua Sudduth
Undergraduate Computer Science Capstone
Capstone Advisor: Michael Reale
SUNY Polytechnic Institute
Spring 2024

Introduction

My goal was to develop a windows assistant style python program that allows users to access and use different widgets in one place. I set out to provide a way for users to create their own widgets and import them to the program as well as pick and choose which widgets they want to display for each user's individual purpose. I wanted users to be able to take the framework of my code and design their own widget, so that as long as the user understands how to code a subclass in python, they can implement it seamlessly into the existing application. I visualize this as creating two big puzzle pieces, where the application is the piece the widget fits into, and the widget has to universally fit into this piece, regardless of what the subclass of the widget is. The struggle with this method is ensuring that the user can easily understand that they are creating a subclass of the widget method, so that it can seamlessly integrate into the application itself. The user will not have access to the application code, so it's important that the user never has to change anything regarding the application, and that the framework works as planned and can be replicated with multiple widgets. It is a valuable tool to be able to think about creating an application with the user in mind the whole time, designing the application around the user being able to use it effectively. I also lacked personal experience coding in Python, as my primary languages have been C++ and Java, and this project allowed me the opportunity to understand and learn class structures in python. By prioritizing the design around the user and creating a simple framework to build off, I was able to create a simple widget builder that can take imported python files built from a framework given to the user and display them within the application.

Related Work

A widget manager is admittedly not a new idea, but learning the concepts involved in creating the widget and working with classes in python was a new experience. Labarchives [1] is one example of a company that has created an expansive widget builder. They have a number of existing widgets built in that are already accessible to the user. The expansiveness of the selection is the benefit of the manager, specific widgets allow for user customization. One example is a scientist in need of multiple different specific calculators that a scientific one wouldn't have. An acid and base molarity calculator is much easier than specifically calculating the molarity each time, and being able to cycle through these options can save someone a lot of valuable research time. In addition to this, their widget builder allows the user the ability to create their own widgets inside the application. They open the scripts up to the user to customize and design their own widget, as the manager would have no use to someone if they would have to open up another calculator, it defeats the point of the widget manager. Labarchives takes the approach of allowing the user to create their own to the level of their coding expertise, which I thought was a smart way to handle the boundaries of the widget builder, and I opted for this approach in designing my own. Separate python files can be used and imported into my program, but there needs to be a way to design the classes to fit into my file. CodeProject [4] was interesting to me to learn about python classes and I realized the concept of polymorphing was going to be vital to my project. Polymorphism is the concept of the same function name being used for different things. This means that my application is using a common interface that each subclass must adhere to. Each widget class has to contain specific methods that are crucial to make the widget function. With python, I learned that these methods could be placed in the main widget class of each function, as opposed to requiring the user to create these in the subclass. A

subclass of the main widget class would allow the user the ability to customize the widget while not having to worry about following these rules that I as the programmer set in place for each widget.

Method

Class Design

I decided to design my classes so that it would be as simple as possible for the user to follow. Multiple different classes for every single widget is not practical as I need the user to be able to import the file that they design without ever having to interact with the code of the application itself. Initially, my first idea was to use a constructor for each widget I added, so once imported it was directly in the code. I quickly realized that this is simply not possible. I decided to design my classes so that each widget would automatically fulfill a set of requirements that I needed. Each widget is the main class, and that contains the necessary pieces to complete the puzzle of interacting with my widget application. The specific widget customization is a subclass of the main widget class itself. The widget manager has 3 different kinds of classes that are important to how it runs:

- WidgetManager: this is the main widget application class that the user does not have access to. It handles the importing of the widgets, stores the widgets that have been imported and the displaying and hiding of those stored widgets. This also handles the adding and removing of frames to display the widgets once they are displayed, as a way to build the widget inside the application.

- Widget (main class): this is the class that interacts with the WidgetManager. This contains the frame for the widget to be built in to, as well as the ability to be hidden and displayed by the WidgetManager.
- xxWidget (subclass): this is what the user edits, and is blank on its own. This inherits from the widget class and contains widget specific functionality. In my program, 2 examples of these are the ClockWidget and CalculatorWidget, where the subclasses of those contain the code to run widget specific functionality, such as determining the current time and displaying it as a clock and updating every second.

Functionality / User Interaction

One of the key features to my widget application is the ability to import widgets from a python file that a user creates. The application dynamically loads the python file and prompts the user to name the widget. I decided to prompt the user to name the widget as opposed to loading the widget file name in order to allow multiple of the same widgets. If a user was using a calculator and did not want to clear the results, they can load an additional calculator without deleting the first one by importing a second file. By having the user import their own files, it also expands the potential for the widgets created, as the user can write their own code as opposed to being limited to whatever widgets I provided, or by whatever limited code they could write within the application if that's the route I went down. It is also important that the widget loads inside of the application as opposed to using the widget manager as a widget opener. This is the second feature of the application, the ability for the widget manager to load and display, as well as hiding the widget. The widget is added to a list that the user can select from and use the button above to display and hide the selected widget. I need the application to be intuitive for the user to use, so the user can hit the buttons that never move to perform each functionality. The only

functionality that doesn't have a button on the left above the widget list is the ability for the user to interact with each specific widget. My calculator widget has the built in ability for the user to interact with it with the most commonly used mathematical expressions and numbers, as well as the ability to clear. This was done by assigning each button either a symbol or number, and using pythons built in mathematical functions. The clock widget also interacts with the user, although the user doesn't have any control of the clock. It dynamically updates with the system time on the user's computer using trigonometric functions that I was able to research [5].

Framework

```
import tkinter as tk



class Widget:

    def __init__(self, master, widget_manager, widget_name):

        self.master = master

        self.widget_manager = widget_manager

        self.widget_name = widget_name



        self.frame = tk.Frame(self.master)

        self.frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)



        self.widget_manager.add_widget_frame(self.widget_name, self.frame)

        self.widget_manager.update_widget_listbox()
```

```
        self.window_tag = f"widget_{self.widget_name}"
        self.frame.window_tag = self.window_tag

    def display(self):
        self.frame.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

    def hide(self):
        self.frame.pack_forget()

# Change Calculator to Widget subclass name of choosing
class CalculatorWidget(Widget):

def main():
    root = tk.Tk()
    widget_manager = WidgetManager(root)

    # Change Calculator to Widget name of choosing
    calculator_widget = CalculatorWidget(root, widget_manager, "Calculator")
    calculator_widget.display()

    root.mainloop()

if __name__ == "__main__":
```

```
    main()

"""
```

This is the framework that the user will put into their own python file. The widget class provides all the necessary code to integrate the widget within the application. There are two comments in the framework, explaining to the user that they need to modify two parts of the code beyond just creating the subclass. The user will need to change the name of the widget to match what they are creating. Those are the only two comments in the framework, making it clear to the user that they will have to change it. Instructions on how to manage the framework are found built into the application in a help section at the top. Tkinter was also incredibly valuable in designing the framework, as it contains the frame that allows the widget to be built in.

Calculator Widget

```
class CalculatorWidget(Widget):

    def __init__(self, master, widget_manager, widget_name):

        super().__init__(master, widget_manager, widget_name)



        self.calculator_frame = tk.Frame(self.frame)

        self.calculator_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)




        self.entry = tk.Entry(self.calculator_frame, width=20, font=('Arial', 13))

        self.entry.grid(row=0, column=0, columnspan=5, padx=10, pady=10, sticky="ew")
```

```
        buttons = [
            ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('/', 1, 3), ('C', 1, 4),
            ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('*', 2, 3), ('%', 2, 4),
            ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('-', 3, 3), ('^', 3, 4),
            ('0', 4, 0), ('.', 4, 1), ('=', 4, 2), ('+', 4, 3), ('sqrt', 4, 4)
        ]

        for (text, row, column) in buttons:
            button = tk.Button(self.calculator_frame, text=text, width=5, height=2,
font=('Arial', 14), command=lambda t=text: self.on_button_click(t))
            button.grid(row=row, column=column, padx=5, pady=5)

    def on_button_click(self, char):
        if char == '=':
            try:
                result = eval(self.entry.get())
                self.entry.delete(0, tk.END)
                self.entry.insert(tk.END, str(result))
            except Exception as e:
                self.entry.delete(0, tk.END)
                self.entry.insert(tk.END, "Error")
        elif char == 'C':
            self.entry.delete(0, tk.END)
        elif char == '^':
```

```
            self.entry.insert(tk.END, "**")
        elif char == 'sqrt':
            self.entry.insert(tk.END, "**0.5")
        else:
            self.entry.insert(tk.END, char)
```

The calculator widget starts by creating a calculator frame in which it will store all the buttons that will be shown to the user. I included the ten numbers, as well as six more buttons, to enable decimals, addition, subtraction, multiplication, division, and an equal sign to calculate. I then realized I needed a clear button, so I added three more less frequently used symbols in mod, exponents and the square root of a number in order to make the UI accommodate the extra button. Each button performs the mathematical expression that is assigned to it with the exception of sqrt, ^, clear, and =. These symbols need to have explicit catches in order to function, but the calculator itself is fairly intuitive after learning how to draw the physical calculator.

Clock Widget

```
class ClockWidget(Widget):
    def __init__(self, master, widget_manager, widget_name):
        super().__init__(master, widget_manager, widget_name)

        self.canvas = tk.Canvas(self.frame, width=200, height=200, bg="white")
        self.canvas.pack(expand=True)
```

```
        self.update_clock()



    def update_clock(self):

        self.canvas.delete("clock_hand")



        now = datetime.now()

        hour = now.hour % 12

        minute = now.minute

        second = now.second



        hour_angle = pi / 2 - (hour * 30 + minute * 0.5) * (pi / 180)

        minute_angle = pi / 2 - minute * 6 * (pi / 180)

        second_angle = pi / 2 - second * 6 * (pi / 180)

        self.canvas.create_oval(50, 50, 150, 150, outline="black")



        hour_hand_length = 30

        hour_hand_x = 100 + hour_hand_length * cos(hour_angle)

        hour_hand_y = 100 - hour_hand_length * sin(hour_angle)

        self.canvas.create_line(100, 100, hour_hand_x, hour_hand_y, tags="clock_hand",
fill="black", width=3)



        minute_hand_length = 40

        minute_hand_x = 100 + minute_hand_length * cos(minute_angle)

        minute_hand_y = 100 - minute_hand_length * sin(minute_angle)

        self.canvas.create_line(100, 100, minute_hand_x, minute_hand_y, tags="clock_hand",
fill="black", width=2)
```

```
        second_hand_length = 45
        second_hand_x = 100 + second_hand_length * cos(second_angle)
        second_hand_y = 100 - second_hand_length * sin(second_angle)
        self.canvas.create_line(100, 100, second_hand_x, second_hand_y, tags="clock_hand",
fill="red", width=1)




        self.canvas.after(1000, self.update_clock)
```

The clock widget [5] works by dynamically updating a canvas based on the current system time. It assigns the hour, minute, and second to a value and then calculates the angle that will be used to determine where the line will be drawn. It then, for each of the three hands, calculates the coordinates at which the second hand should be at (using cos and sin for x and y, respectively), It then creates the clock, and updates every second (whenever a new second value would appear). When it updates, all the hands are cleared and new angles are calculated, and the cycle repeats until the widget is hidden.

Results

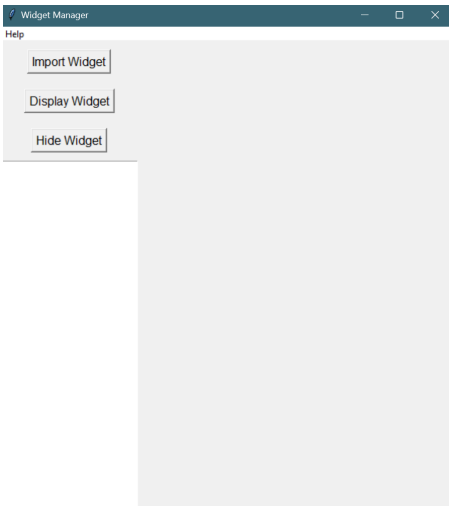

Figure 1: Application upon startup

Immediately upon opening the application as shown in figure 1, the user is greeted with an easy to use interface. There are no widgets imported yet, so the display and hide buttons don't do anything when clicked. There is a clear import widget button that the user is expected to click on, but if they are unsure or don't have any widgets yet to import, they can click on the help button and the application will display 3 options for the the user to choose from: Display Help, How To, and Framework.

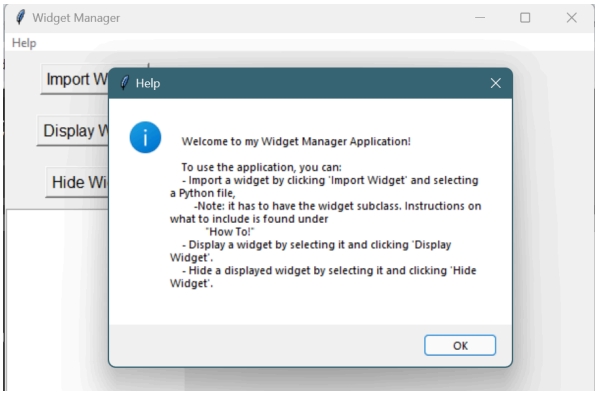

Figure 2: Example Instructions Menu

Shown in Figure 2 is the display help button. It highlights how to use the program and explains for users who are unfamiliar with a widget manager what each button does. It also directs the user to the next help button, the how to button. This explains the framework and what exactly the user has to edit in order to create their own widget. It is a built in guide to explain to the user what to do if they get lost. Lastly, the framework button is a text popup that the user can copy and paste into their IDE.

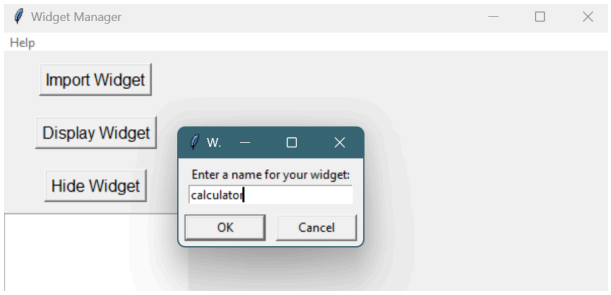

Figure 3: Prompt Upon Importing a File

Upon clicking import widget, the user is prompted to select a python file from the files on their computer. The application verifies that the python file has the widget parent class, and a subclass of that widget. In Figure 3, once it confirms you have a widget file, it prompts the user to name the widget. This widget will then display the name in the widget list, and the user is now allowed to click on the widget they just named and imported.

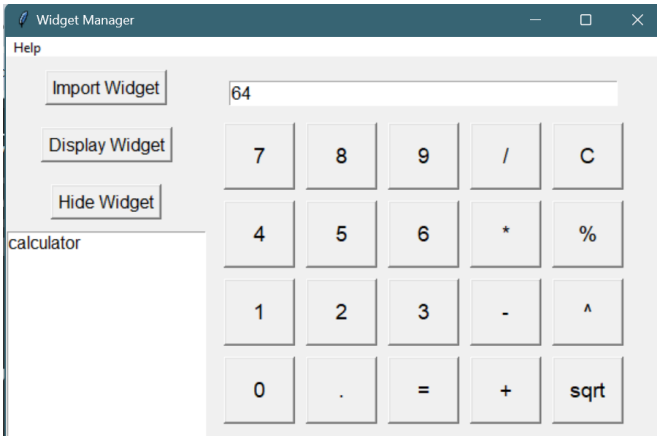

Figure 4: Widget Upon Opening

The widget builder assumes the user wants to open the widget they just imported, so it is loaded and displayed to the user. The calculator widget that was just imported is shown in Figure 4, with the result to 8 x 8 being shown. Now that it is present in the list, the user can toggle whether it's displayed or not.

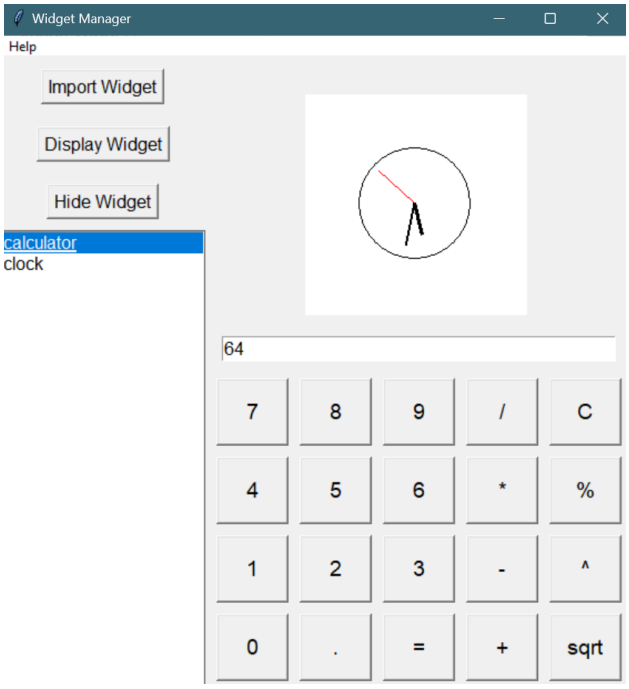

Figure 5: Multiple Widgets At One Time

The widget builder is also built to handle multiple widgets at once. In Figure 5, the calculator was hidden, the clock was imported and the calculator was displayed. Now one widget can be displayed, or multiple can be displayed as needed! Calculator information is also saved when hidden, as the imported file is still the same and under the same name. Multiple calculators could also be shown, and the progress would be saved to each individual one.

Discussion

In order to learn how to create the application, the main piece of technology I had to relearn was python. I took one python class two years ago but besides that, I primarily have coded in C or C++, so I wanted more experience in python class management and learning how different python files interact with each other. I much prefer the python method of inheritance compared to C++, it was a lot more intuitive once I was able to understand the UI of python.

I also had to learn the library of tkinter, and it proved to be a great tool for drawing on canvases. The canvas feature of tkinter was extremely helpful in drawing on my application for my widgets. The frame element took me a while to understand, but once I realized I could put each imported widget inside of a frame that I could add or take away, the project became much easier. Initially, I was using the right window as a "big frame" and trying to force the user to customize the size of the widget and integrate it into my application themselves. I kept encountering an error where the application would load into a different window as opposed to being integrated inside my application. This is obviously a problem because if the widgets load outside the program, the widget manager is functioning as a widget loader, and if that is the case the user may as well just use a calculator application by itself. I was able to fix this problem by using the frames and giving the widget a place to be. If I viewed each widget as a poster, I need a poster frame for each one that only appears when the widget is there.

I also ran into another issue with getting the widget to display and hide accurately. I had no issue getting it to load initially, but once it was gone, I could not get it back. I realized that I had put an accidental limit on the number of widgets that could be displayed. This was originally done to ensure that the widget is not displayed repeatedly when the display button is clicked more than once. I removed this limit and the widget does not repeat so the limit was actually not needed. The actual widget building of my project went really well, since I designed it in a way that I could repeat the framework and not worry about it interacting properly, since when one framework works, they all will. I would definitely do that the same way if I was to repeat this project, but I would change the right frame to a canvas. I envision it would be a lot easier to add additional functionality, specifically with allowing users the ability to customize the GUI when

multiple widgets are downloaded, as right now the user has no control on where the widget is displayed.

Conclusion

In conclusion, the widget manager application was successful in its goal that allows users to access and display widgets. The application can dynamically store the files imported without the user needing access to the application code and automatically integrate them into the window, displaying and hiding the widget as desired. Throughout the project, I encountered a number of difficulties, specifically with learning tkinter and processing the file within the application itself without the user having access to it. However, despite the struggles, widgets are able to be imported, displayed, and hidden as the user desires. There are still a number of improvements that I would make if I was to have more time. Firstly, I would improve the UI of the application when multiple widgets are selected and add a scroll bar. The complication with this comes from the fact that I am using frames to set up the widgets displaying. Based on my brief research into tkinter the best way to set this up would be to set the right side of my application from a window to a canvas. When I tried to do this, I could not get the widgets to display on the canvas, due to them needing the frame that I erased. One possible solution would be to try not use frames entirely and change the framework of the widget to a canvas. Additionally, I would love to add the ability to customize where you put the widgets within the application much like apps on a phone. I think this would be done by customizing the frames that the widgets load in to, perhaps prompting the user on the size when they click display. Ultimately, this project was instrumental in expanding my understanding of python and thinking about designing a program with it being simple for the user to use at the forefront of my mind.

References

[1] A. Gibson, “LabArchives widget manager tips and tricks,” Medium, https://gibson-amandag.medium.com/labarchives-widget-manager-tips-and-tricks-49ad9e306a95 (accessed May 1, 2024).

[2] “Tkinter - Python interface to TCL/TK,” Python documentation, https://docs.python.org/3/library/tkinter.html (accessed May 1, 2024).

[3] “Python 3.12.3 documentation,” 3.12.3 Documentation, https://docs.python.org/3/ (accessed May 1, 2024).

[4] S. Huang, “Python vs C++ series: Polymorphism and duck typing,” CodeProject, https://www.codeproject.com/Articles/5314882/Python-vs-Cplusplus-Series-Polymorphism-and-Duck-T (accessed May 2, 2024).

[5] StormFire1122StormFire1122, “Analog clock using tkinter on python but as soon as I run it it stops after 1 Second,” Stack Overflow, https://stackoverflow.com/questions/75447330/analog-clock-using-tkinter-on-python-but-as-soon-as-i-run-it-it-stops-after-1-se (accessed May 2, 2024).

Appendix

The only requirements to run the widget manager are to have a working version of Python 3.x on your computer that can run the application. Tkinter, the library used in the widget manager, comes included with Python 3.x. Documentation for tkinter and python are found in references [2] and [3]. Any IDE of choice can be used, I personally used Visual Studio Code in the making of this due to my prior experience using it. Since the user is expected to create their own widgets using the framework provided, they need to use the IDE to write the python subclass, and need to be able to run the python application.

GitHub Repository - https://github.com/joshsud/capstone_final