

Submitted By,

Sharmika Das Banhi

Student ID: 210204

Redwan

Student ID: 210207

Computer Science & Engineering Discipline

Khulna University, Khulna

Submitted To,

Amit Kumar Mondal

Associate Professor

Computer Science & Engineering Discipline

Khulna University, Khulna

**Code Review of ‘Text Editor’ (210210, 210223)**

**Code Reviews:**

1. **Code smells:**
2. **Incomplete error handling:**

**import** tkinter **as** tk

**from** tkinter **import** filedialog,messagebox

**import** GUI

**def** **AutoSave**(master: GUI.App):

**def** **inner\_function**(event=None):

**if**(master.autoSave == False): **return**

content=master.main\_text\_box.get('1.0',tk.END)

# print(content)

**if** (master.file\_path==''):

print("No file path was given")

**return**

print(master.file\_path)

**with** open(master.file\_path,'w') **as** fw:

fw.write(content)

**return** inner\_function;

The problem with the code is that it doesn’t handle the error properly. There can occur attribute error, value error, IO error.

1. **Large or Complex Methods:**

import tkinter as tk

from tkinter import scrolledtext

import utils

class App(tk.Tk):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.file\_path: str = ""

self.content: str = ""

self.isSaved: bool = False;

self.isEdited:bool = False;

self.autoSave:bool = False

self.title("Text Editor")

self.option\_add("\*Font",'aerial 13')

self.geometry("700x500")

self.menu\_bar = tk.Menu(master=self)

self.menu\_item\_File = tk.Menu(self.menu\_bar,tearoff=0)

self.menu\_item\_Edit = tk.Menu(self.menu\_bar,tearoff=0)

self.menu\_bar.add\_cascade(label="File",menu=self.menu\_item\_File)

self.menu\_bar.add\_cascade(label="Edit",menu=self.menu\_item\_Edit)

self.config(menu=self.menu\_bar)

## Find all

self.find\_frame = tk.Frame(master=self)

self.find\_frame.pack()

self.find\_text\_label = tk.Label(master=self.find\_frame,text='Search: ')

self.find\_text\_entry = tk.Entry(master=self.find\_frame)

self.find\_text\_btn = tk.Button(master=self.find\_frame,

text="Find All Occurances",

command=utils.FindAll(master=self)

)

self.find\_text\_label.grid(row=0,column=0,pady=2)

self.find\_text\_entry.grid(row=0,column=1,columnspan=2,pady=2)

self.find\_text\_btn.grid(row=0,column=3,pady=2,padx=8)

Top of Form

## Replace all

self.replace\_frame = tk.Frame(master=self)

self.replace\_frame.pack()

self.replace\_text\_label = tk.Label(master=self.replace\_frame,text='replace: ')

self.replace\_text\_entry = tk.Entry(master=self.replace\_frame)

self.replace\_text\_btn = tk.Button(master=self.replace\_frame,

text="replace All Occurances",

command=utils.ReplaceAll(master=self)

)

self.replace\_text\_label.grid(row=0,column=0,pady=2)

self.replace\_text\_entry.grid(row=0,column=1,columnspan=2,pady=2)

self.replace\_text\_btn.grid(row=0,column=3,pady=2,padx=8)

## Text Box

self.main\_text\_box = scrolledtext.ScrolledText(master=self,undo=True)

self.main\_text\_box.pack(padx=10,pady=10,expand='yes',fill='both')

self.menu\_item\_File.add\_command(label="Open",

command=utils.Open(master=self),

accelerator="Ctrl+O".rjust(15),

)

self.menu\_item\_File.add\_command(label="Save As",

command=utils.SaveAs(master=self),

accelerator="Ctrl+Shift+S".rjust(15),

)

self.menu\_item\_File.add\_command(label="Save",

command=utils.Save(master=self),

accelerator="Ctrl+S".rjust(15),

)

self.menu\_item\_File.add\_command(label="Exit",

command=utils.Exit(master=self),

accelerator="Ctrl+Shift+X".rjust(15),

)

self.menu\_item\_Edit.add\_command(label="Copy All",

command=utils.CopyAll(master=self),

accelerator="Ctrl+Shift+C".rjust(15),

)

self.menu\_item\_Edit.add\_command(label="Clear",

command=utils.Clear(master=self),

accelerator="Ctrl+Shift+D".rjust(15),

)

self.menu\_item\_Edit.add\_command(label="Undo",

command=self.main\_text\_box.edit\_undo,

accelerator="Ctrl+Z".rjust(15),

)

self.menu\_item\_Edit.add\_command(label="Redo",

command=self.main\_text\_box.edit\_redo,

accelerator="Ctrl+Y".rjust(15),

self.menu\_item\_Edit.add\_command(label="Highlight",

command=utils.Highlight(master=self),

)

self.menu\_item\_Edit.add\_command(label="Toggle Auto Save",

command=utils.toggleAutoSave(master=self),

)

self.menu\_item\_Edit.add\_command(label="Word count",

command=utils.CountWords(master=self),

)

## Bindings

self.bind\_all('<Control-o>', utils.Open(master=self))

self.bind\_all('<Control-s>', utils.Save(master=self))

self.bind\_all('<Control-S>', utils.SaveAs(master=self))

self.bind\_all('<Control-X>', utils.Exit(master=self))

self.bind\_all('<Control-C>', utils.CopyAll(master=self))

self.wm\_protocol("WM\_DELETE\_WINDOW", utils.Exit(master=self))

def auto\_save\_text\_file(self):

utils.AutoSave(master=self)()

self.after(3000,self.auto\_save\_text\_file)

if \_\_name\_\_ == '\_\_main\_\_':

app = App()

app.auto\_save\_text\_file()

app.mainloop()

There exists a large and complex method **def** \_\_init\_\_**(self, \*args, \*\*kwargs).** It is difficult to handle such a complex and large method**.** This method is responsible for both managing the user interface (UI) components (like menus, buttons, text boxes) and integrating them with the application's functionality (like file operations, editing actions). This violates SRP, as a method should ideally have only one reason to change.

1. **Duplicate code:**

**Save.py:**

**import** tkinter **as** tk

**from** tkinter **import** filedialog,messagebox

**import** GUI

**def** **Save**(master: GUI.App):

**def** **inner\_function**(event=None):

content=master.main\_text\_box.get('1.0',tk.END)

print(content)

**if** (master.file\_path==''):

master.file\_path=filedialog.asksaveasfilename(filetypes=[("txt files ",".txt")],defaultextension=".txt")

print(master.file\_path)

**with** open(master.file\_path,'w') **as** fw:

fw.write(content)

**return** inner\_function;

**Saveas.py:**

**import** tkinter **as** tk

**from** tkinter **import** filedialog,messagebox

**import** GUI

**def** **SaveAs**(master: GUI.App):

**def** **inner\_function**(event=None):

content=master.main\_text\_box.get('1.0',tk.END)

print(content)

master.file\_path=filedialog.asksaveasfilename(filetypes=[("txt files ",".txt")],defaultextension=".txt")

print(master.file\_path)

**with** open(master.file\_path,'w') **as** fw:

fw.write(content)

**return** inner\_function

There exists duplication of code between the Save and SaveAs functions. Both functions contain similar logic for saving the content of the text box to a file using the filedialog.asksaveasfilename dialog.

1. **Architecture:**
2. **Does the code follows the architecture pattern or not:**

**Single Responsibility Principle (SRP):** The GUI class is responsible for both managing the user interface (UI) components (like menus, buttons, text boxes) and integrating them with the application's functionality (like file operations, editing actions). This violates SRP, as a class should ideally have only one reason to change.

**Separation of Concerns:** The GUI class intertwines UI-related tasks (like widget layout, menu creation) with application logic (like file operations, editing actions). In a repository architecture, concerns should be separated more clearly, with UI tasks handled separately from business logic.

**Modularity and Extensibility:** While the GUI class does encapsulate the UI-related functionalities, it's tightly coupled with the application's specific functionalities. This could make it harder to extend or modify the application in the future without affecting the UI.

**Testability:** Testing the application becomes more complex when UI and application logic are tightly coupled in the same class. Ideally, UI components and application functionalities should be testable independently.

**Flexibility:** The tight coupling between UI and application logic could limit the flexibility of the application, making it harder to replace or modify UI components without affecting the underlying functionalities.

For fully coping up with the proposed architecture ‘Repository architecture pattern’ the above functionality should be fulfilled properly.

1. **Does the design pattern is appropriate for the project or not:**

After thoroughly reviewing the codebase and considering the problem context, it appears that the **Repository Architecture pattern** is not fully appropriate for the codebase. While the code does demonstrate modularity and separation of concerns to some extent, the tight coupling between UI-related tasks and application logic within the GUI class indicates that a different architectural pattern might be more suitable.

A more suitable design pattern for this codebase could be the **Model-View-Controller (MVC) pattern**. MVC separates an application into three main components: the Model (handles data), the View (handles user interface and presentation logic), and the Controller (mediates between the Model and View, handling user input and updating the Model).

In this context:

**Model:** This would encapsulate the core functionality of the text editor, including file operations, text manipulation, manages the system data and associated operations on that data.

**View:** The GUI class and associated UI components would fall under the View layer. They would be responsible for rendering the user interface and handling user interactions and defines and manages how the data is presented to the user.

**Controller:** This layer would handle user input events from the View, delegate tasks to the appropriate components in the Model, and update the View accordingly.

Adopting the MVC pattern would promote better **separation of concerns, making the codebase more modular**, **maintainable**, and **testable**. It would also facilitate future enhancements and modifications to the application, as each component's responsibilities are clearly defined.

Therefore, while the Repository Architecture pattern demonstrates some beneficial aspects in the codebase, considering the context and problem requirements, the MVC pattern would likely be a more appropriate choice for achieving a well-structured and maintainable design.

1. **Modularity or separation of concern:**

**Separation of Concerns:** The GUI class intertwines UI-related tasks (like widget layout, menu creation) with application logic (like file operations, editing actions). Concerns should be separated more clearly, with UI tasks handled separately from business logic.

**Modularity:** While the GUI class does encapsulate the UI-related functionalities, it's tightly coupled with the application's specific functionalities. This could make it harder to extend or modify the application in the future without affecting the UI.