Code Review of
‘feature-rich desktop calendar software’

Course Name: **Software Development Project**

Course No: **CSE 3106**

Submitted to:
Dr. Amit Kumar Mondal
Associate Professor
Computer Science & Engineering Discipline,
Khulna University,
Khulna.

Submitted by:

Name: Muhammad Fahim
Student ID: 210210

Name: Umme Talha
Student ID: 210223

Project Title: feature-rich desktop calendar software

Project Developers:

Name: Sharmika Das Banhi
Student ID: 210204

Name: Redwan
Student ID: 210207

Code Reviewed By:

Name: Muhammad Fahim
Student ID: 210210

Name: Umme Talha
Student ID: 210223

Code Smells

1. **Large/Complex Methods:**

Code Smell: Some methods like `update_calendar()`, `display_events()`, and `setup_gui()` in `CalendarGUI` class are relatively long, which can make them harder to understand and maintain.

update_calendar:

```
    def update_calendar(self):
        self.label_month_year.config(
```

```
            text=calendar.month_name[self.current_date.month] +
" " + str(self.current_date.year))

        month_calendar =
calendar.monthcalendar(self.current_date.year,
self.current_date.month)

        # Display the days of the week above the dates
        days_of_week = [calendar.day_abbr[i] for i in range(7)]
        for col, day in enumerate(days_of_week):
            day_label = tk.Label(self.calendar_frame, text=day,
fg="blue")  # Change the color to blue
            day_label.grid(row=0, column=col)

        for week in month_calendar:
            for day in week:
                if day != 0:
                    label = tk.Label(self.calendar_frame,
text=str(day))
                    label.bind("<Button-1>", lambda event,
d=day: self.select_date(d))

                    # Set background color for Saturdays (index
5) and Fridays (index 4)
                    if week.index(day) == 5:  # Saturday
                        label.config(bg="Red")
                    elif week.index(day) == 4:  # Friday
                        label.config(bg="Red")

                    current_date =
datetime(self.current_date.year, self.current_date.month,
day).date()

                    # Check if the date is a public holiday
                    if current_date in
self.schedule_manager.public_holidays:
                        label.config(bg="LightGreen")  # Adjust
color as needed

                    if current_date == datetime.today().date():
```

```
                        label.config(bg="yellow")

                    label.grid(row=month_calendar.index(week) +
1, column=week.index(day))

        self.display_events()
```

display_events:

```
   def display_events(self):
        self.event_display.delete(1.0, tk.END)
        if self.selected_date:
            selected_date = datetime(self.current_date.year,
self.current_date.month, self.selected_date).date()
            events =
self.schedule_manager.get_events(selected_date)

            if self.is_birthday_today(events):
                birthday_person =
self.get_birthday_person(events)
                self.event_display.insert(tk.END, f"Happy
Birthday, {birthday_person}!\n")
                self.event_display.tag_add("birthday", "1.0",
tk.END)
                self.event_display.tag_config("birthday",
foreground="red")

            elif events:
                for event, _, person_name in events:
                    event_text = f"{event} ({person_name})"
                    self.event_display.insert(tk.END,
f"{event_text}\n")
                    self.event_display.tag_add("event", "1.0",
tk.END)
                    self.event_display.tag_config("event",
foreground="blue")
            else:
                self.event_display.insert(tk.END, f"
{selected_date}")
```

```
                if selected_date in
self.schedule_manager.public_holidays:
                    holiday_name =
self.schedule_manager.public_holidays[selected_date]
                    self.event_display.insert(tk.END, f"Public
Holiday: {holiday_name}\n")

self.event_display.tag_add("public_holiday", "1.0", tk.END)

self.event_display.tag_config("public_holiday",
foreground="green")
        else:
            self.event_display.insert(tk.END, "Select a date to
display events.")

    def show_all_events(self):
        all_events_text =
self.schedule_manager.show_all_events(self.current_date)
        self.event_display.delete(1.0, tk.END)
        if all_events_text:
            self.event_display.insert(tk.END, all_events_text)
            self.event_display.tag_add("event", "1.0", tk.END)
            self.event_display.tag_config("event",
foreground="green")

    def add_birthday(self):
        person_name = askstring("Input", "Enter person's
name:")
        if person_name:
            birthday_description = f"Happy Birthday!
({person_name})"
            self.event_display.delete(1.0, tk.END)
            self.event_display.insert(tk.END,
birthday_description)
            self.save_event()

    def check_birthday(self):
        if self.selected_date:
            selected_date = datetime(self.current_date.year,
```

```
self.current_date.month, self.selected_date).date()
            events =
self.schedule_manager.get_events(selected_date)
            if any(event[1] for event in events):  # Check if
there are birthdays
                messagebox.showinfo("Birthday Checker", "This
date has birthdays!")
            else:
                messagebox.showinfo("Birthday Checker", "No
birthdays on this date.")
        else:
            messagebox.showinfo("Birthday Checker", "Select a
date to check birthdays.")

    def set_alarm(self):
        alarm_time_str = askstring("Set Alarm", "Enter alarm
time (hh:mm AM/PM):")
        if alarm_time_str:
            try:
                alarm_time = datetime.strptime(alarm_time_str,
"%I:%M %p").time()
                self.schedule_alarm(alarm_time)
                messagebox.showinfo("Alarm Set", f"Alarm set
for {alarm_time_str}")
            except ValueError:
                messagebox.showerror("Invalid Time", "Please
enter a valid time in the format hh:mm AM/PM")
```

Setup_gui:

```
    def setup_gui(self):

        self.master.title("Desktop Calendar")
        self.master.geometry("800x700")

        modern_font = ("Helvetica", 14)
```

```
        # Create a frame for the month and year label
        label_frame = tk.Frame(self.master, bg="#EFEFEF")  #
Light Gray
        label_frame.pack(pady=25)

        # Display the month and year label
        self.label_month_year = tk.Label(label_frame, text="",
font=("Arial", 36),
                                         bg="#FFFFE0")  #
Adjust font style and size
        self.label_month_year.pack()

        # Create a frame for the calendar
        self.calendar_frame = tk.Frame(self.master,
bg="#FFFFFF")
        self.calendar_frame.pack()

        # Display the days of the week above the dates
        days_of_week = [calendar.day_abbr[i] for i in range(7)]
        for col, day in enumerate(days_of_week):
            day_label = tk.Label(self.calendar_frame, text=day,
fg="blue",
                                 bg="#E0E0E0")  # Increase font
size
            day_label.grid(row=0, column=col, padx=10)

        # Create a frame for the navigation buttons
        nav_button_frame = tk.Frame(self.master, bg="#E0E0E0")
# Light Gray
        nav_button_frame.pack(pady=10)

        # Add navigation buttons
        self.prev_year_button = tk.Button(nav_button_frame,
text="Previous Year", command=self.prev_year,
                                          bg="#FFD700",
relief=tk.FLAT, font=modern_font)
        self.prev_year_button.pack(side=tk.LEFT, padx=5,
pady=5)
```

```
        self.prev_month_button = tk.Button(nav_button_frame,
text="Previous Month", command=self.prev_month,
                                           bg="#98FB98",
font=modern_font)
        self.prev_month_button.pack(side=tk.LEFT, padx=5)

        self.next_month_button = tk.Button(nav_button_frame,
text="Next Month", command=self.next_month,

bg="#98FB98",font=modern_font)  # Pale Green
        self.next_month_button.pack(side=tk.RIGHT, padx=5)

        self.next_year_button = tk.Button(nav_button_frame,
text="Next Year", command=self.next_year,
bg="#FFD700",font=modern_font)
        self.next_year_button.pack(side=tk.RIGHT, padx=5)

        # Create a frame for the additional buttons
        additional_button_frame = tk.Frame(self.master,
bg="#E0E0E0")
        additional_button_frame.pack(pady=10)

        self.todo_button = tk.Button(additional_button_frame,
text="To-Do List", command=self.manage_todo_list,
                                     bg="#87CEEB",
font=modern_font)
        self.todo_button.pack(side=tk.LEFT, padx=5)

        self.add_birthday_button =
tk.Button(additional_button_frame, text="Add Birthday",
command=self.add_birthday,
                                          bg="#98FB98",
font=modern_font)
        self.add_birthday_button.pack(side=tk.LEFT, padx=5)

        self.set_alarm_button =
tk.Button(additional_button_frame, text="Set Alarm",
command=self.set_alarm,

bg="#87CEEB",font=modern_font)
```

```
        self.set_alarm_button.pack(side=tk.RIGHT, padx=5)

        self.show_all_events_button =
tk.Button(additional_button_frame, text="Show All Events",

command=self.show_all_events, bg="#98FB98",font=modern_font)
        self.show_all_events_button.pack(side=tk.RIGHT, padx=5)

        # Create a frame for the event display
        event_display_frame = tk.Frame(self.master,
bg="#FFFFFF")
        event_display_frame.pack(pady=10)

        # Display the event text box
        self.event_display = tk.Text(event_display_frame,
height=10, width=40, wrap=tk.WORD, bg="#FFFFFF",
                                     font=("Arial", 12))
        self.event_display.pack(pady=5)

        # Create a frame for the save event button
        save_event_frame = tk.Frame(self.master, bg="#E0E0E0")
# Light Gray
        save_event_frame.pack()

        # Add the save event button
        self.save_event_button = tk.Button(save_event_frame,
text="Save Event", command=self.save_event,

bg="#FFD700",font=modern_font)  # Gold
        self.save_event_button.pack(side=tk.RIGHT, padx=5)

        self.check_birthday_button =
tk.Button(additional_button_frame, text="Check Birthday",

command=self.check_birthday, bg="#e74c3c", font=modern_font,
                                                fg="white")  #
Red background
        self.check_birthday_button.pack(side=tk.LEFT, padx=5)

        reminder_frame = tk.Frame(self.master, bg="#E0E0E0")
```

```
        reminder_frame.pack(pady=10)

        # Add entry for reminder message
        self.reminder_entry = tk.Entry(reminder_frame,
width=40, font=("Arial", 12))
        self.reminder_entry.pack(side=tk.LEFT, padx=5)

        # Add button to set reminder
        set_reminder_button = tk.Button(reminder_frame,
text="Set Reminder", command=self.set_reminder,
                                        bg="#87CEEB",
font=("Arial", 12))
        set_reminder_button.pack(side=tk.LEFT, padx=5)

        self.update_calendar()

    # Inside the CalendarGUI class in calendar_gui.py
```

Possible Solution:

- Refactoring long methods into smaller, more modular functions

```
def update_calendar(self):
    self.update_month_year_label()
    self.display_days_of_week()
    self.display_month_calendar()
    self.display_events()
```

```
def display_events(self):
    self.clear_event_display()
    if self.selected_date:
        self.display_selected_date_events()
    else:
        self.display_no_date_selected_message()
```

Define helper methods to break down the logic into smaller, more manageable pieces

2. **Magic Numbers/hard coded values:**

Code Smell: The `CalendarGUI` class contains magic numbers and strings (e.g., colors, date formats) without explicit meaning or context.

```
        # Display the days of the week above the dates
        days_of_week = [calendar.day_abbr[i] for i in range(7)]
        for col, day in enumerate(days_of_week):
            day_label = tk.Label(self.calendar_frame, text=day,
fg="blue")  # Change the color to blue
            day_label.grid(row=0, column=col)

        for week in month_calendar:
            for day in week:
                if day != 0:
                    label = tk.Label(self.calendar_frame,
text=str(day))
                    label.bind("<Button-1>", lambda event,
d=day: self.select_date(d))
```

```
                    # Set background color for Saturdays (index
5) and Fridays (index 4)
                    if week.index(day) == 5:  # Saturday
                        label.config(bg="Red")
                    elif week.index(day) == 4:  # Friday
                        label.config(bg="Red")
```

Possible Solution:

- Defining constants for magic numbers and strings

```
class CalendarGUI:
    BACKGROUND_COLOR_SELECTED_DATE = "yellow"
    BACKGROUND_COLOR_SATURDAY = "red"
    BACKGROUND_COLOR_FRIDAY = "red"
    BACKGROUND_COLOR_PUBLIC_HOLIDAY = "LightGreen"

    def __init__(self, master, schedule_manager):
        # Constructor code

    # Rest of the class definition
```

By defining constants for magic numbers and strings, developers provide explicit meaning and context, improving code readability and maintainability.

3. **Too Many if/else Statements:**

Code Smell: The `CalendarGUI` class contains `display_events()` method and the Event class contains `show_all_events()` which has too many if/else statements

display_events:

```
    def display_events(self):
        self.event_display.delete(1.0, tk.END)
        if self.selected_date:
            selected_date = datetime(self.current_date.year,
self.current_date.month, self.selected_date).date()
            events =
self.schedule_manager.get_events(selected_date)

            if self.is_birthday_today(events):
                birthday_person =
self.get_birthday_person(events)
                self.event_display.insert(tk.END, f"Happy
Birthday, {birthday_person}!\n")
                self.event_display.tag_add("birthday", "1.0",
tk.END)
                self.event_display.tag_config("birthday",
foreground="red")

            elif events:
                for event, _, person_name in events:
                    event_text = f"{event} ({person_name})"
                    self.event_display.insert(tk.END,
f"{event_text}\n")
                    self.event_display.tag_add("event", "1.0",
tk.END)
                    self.event_display.tag_config("event",
foreground="blue")
            else:
                self.event_display.insert(tk.END, f"
{selected_date}")
```

```
                if selected_date in
self.schedule_manager.public_holidays:
                    holiday_name =
self.schedule_manager.public_holidays[selected_date]
                    self.event_display.insert(tk.END, f"Public
Holiday: {holiday_name}\n")

self.event_display.tag_add("public_holiday", "1.0", tk.END)

self.event_display.tag_config("public_holiday",
foreground="green")
        else:
            self.event_display.insert(tk.END, "Select a date to
display events.")
```

show_all_events:

```
    def show_all_events(self, month):
        all_events = []
        today_date = datetime.today().date()

        for day in range(1, calendar.monthrange(month.year,
month.month)[1] + 1):
            date = datetime(month.year, month.month,
day).date()
            events = self.get_events(date)

            if events:
                for event in events:
                    person_name = event[2] if event[2] is not
None else ''  # Return empty string if the name is None
                    event_text = f"{event[0]} ({date} -
{calendar.day_name[date.weekday()]} - {person_name})"

                    if event[1]:
```

```
                        next_year_date = datetime(month.year +
1, month.month, day).date()
                        event_text += f" - Next year on
{next_year_date}"
                    elif date == today_date:
                        event_text += " - Today is the
birthday!"

                    all_events.append(event_text)

        return "\n".join(all_events)
```

Possible Solution:

- Simplify the method using switch case statements

```
def display_events(self):
    self.event_display.delete(1.0, tk.END)
    selected_date = datetime(self.current_date.year,
self.current_date.month, self.selected_date).date()
    events = self.schedule_manager.get_events(selected_date)


    match events:
          . . .

          # All the cases here
```

4. **Incomplete Error Handling:**

Code Smell: In the `set_alarm()` and `set_reminder()` method in `CalendarGUI` class, the issue lies in the way it handles invalid input for the alarm time.

- It doesn't provide specific details about what went wrong with the input.
- Users might not understand why their input is invalid or how to correct it.

set_alarm:

```
    def set_alarm(self):
        alarm_time_str = askstring("Set Alarm", "Enter alarm
time (hh:mm AM/PM):")
        if alarm_time_str:
            try:
                alarm_time = datetime.strptime(alarm_time_str,
"%I:%M %p").time()
                self.schedule_alarm(alarm_time)
                messagebox.showinfo("Alarm Set", f"Alarm set
for {alarm_time_str}")
            except ValueError:
                messagebox.showerror("Invalid Time", "Please
enter a valid time in the format hh:mm AM/PM")
```

set_reminder:

```
def set_reminder(self):
    reminder_message = self.reminder_entry.get().strip()
```

```
    if reminder_message:
        try:
            reminder_time_str = askstring("Set Reminder Time",
"Enter reminder time (hh:mm AM/PM):")

            if reminder_time_str:
                reminder_datetime =
datetime.strptime(reminder_time_str, "%I:%M %p").time()

                reminder_thread =
Thread(target=self.run_reminder, args=(reminder_message,
reminder_datetime))
                reminder_thread.start()

                messagebox.showinfo("Reminder Set", f"Reminder
set for {reminder_time_str}")

        except ValueError:
            messagebox.showerror("Invalid Time", "Please enter
a valid time in the format hh:mm AM/PM")
```

Possible Solution:

Instead of a generic error message, consider providing more informative feedback. For example:

- “Invalid time format. Please use the format hh:mm AM/PM.”
- “Invalid hour. Hours should be between 1 and 12.”
- “Invalid minutes. Minutes should be between 0 and 59.”
- “Invalid AM/PM designation. Use ‘AM’ or ‘PM’.”

Improving exception handling by providing informative error messages helps users understand and correct input errors more easily.

5. **Unnecessary dependencies:**

Code Smell: The `CalendarGUI` class imports modules that are not used, leading to unnecessary clutter and potentially confusion.

```
import tkinter as tk
from datetime import datetime, timedelta
from tkinter.simpledialog import askstring
from tkinter import messagebox
from threading import Thread
import calendar
import winsound
from tkinter import PhotoImage
```

Possible Solution:

- Remove unused imports to declutter the code

```
import tkinter as tk
from datetime import datetime, timedelta
from tkinter.simpledialog import askstring
from tkinter import messagebox
from threading import Thread
```

```
import calendar
import winsound

# Remove unused imports to declutter the code
# from tkinter import PhotoImage
```

6. **Large Class:**

Code Smell: The `CalendarGUI` class appears to handle multiple responsibilities, violating the Single Responsibility Principle (SRP) and making it difficult to understand and maintain.

```
class CalendarGUI:
    def __init__(self, master, schedule_manager):
        self.master = master
        self.schedule_manager = schedule_manager
        self.current_date = datetime.now()
        self.selected_date = None

        self.setup_gui()

        . . .

        # Rest of the Class
```

Possible Solution:

- Splitting into smaller classes

```
class CalendarGUI:
    def __init__(self, master, schedule_manager):
        self.master = master
        self.schedule_manager = schedule_manager
        self.current_date = datetime.now()
        self.selected_date = None
        self.setup_gui()

    def setup_gui(self):
        # GUI setup code
```

```
class AlarmManager:
    def __init__(self):
        pass
    # Add methods to handle alarm-related functionality
```

```
class ReminderManager:
    def __init__(self):
        pass
    # Add methods to handle reminder-related functionality
```

By splitting the responsibilities into smaller, focused classes, each class can adhere to the Single Responsibility Principle, making the codebase more maintainable and easier to understand.

Architecture

➢ **Does the codebase follow Layered Architecture or Not?**

The provided codebase does not strictly adhere to a layered architecture for several reasons:

1. **Direct Interaction with Business Logic**: The `CalendarGUI` class directly interacts with the `Event` class for managing events. In a layered architecture, such interaction should typically occur through well-defined interfaces or services provided by the business logic layer.
2. **Limited Separation of Layers**: There is no clear separation between layers such as presentation, business logic, and data access. Instead, the codebase consists of a single class (`CalendarGUI`) that combines elements of presentation and business logic.
3. **Lack of Abstraction**: The code lacks clear abstraction between different layers. In a layered architecture, each layer should encapsulate its functionality and expose only necessary interfaces to other layers. However, in the provided codebase, the GUI elements and event management logic are tightly coupled within the `CalendarGUI` class.

4. **Dependencies Flowing in Both Directions**: In a layered architecture, dependencies should flow in a single direction, typically from higher-level layers to lower-level layers. However, in the provided codebase, there are bidirectional dependencies between the GUI layer and the event management logic.

➢ **Is layered Architecture design pattern appropriate for the codebase or not?**

Layered architecture may not be the most suitable design pattern for the provided codebase due to several reasons:

1. **Mixing of Concerns**: The codebase exhibits a significant mixing of concerns, with GUI-related logic tightly coupled with business logic. This mixing makes it challenging to cleanly separate layers according to the principles of a layered architecture.
2. **Tight Coupling**: There is a high degree of coupling between the GUI components and the event management logic. Layers in a layered architecture should ideally have loose coupling, but this is not the case here.
3. **Single Class Handling Multiple Responsibilities**: The `CalendarGUI` class seems to handle both GUI presentation and event management. In a layered architecture, each layer should have a distinct responsibility, but here, the class violates the Single Responsibility Principle (SRP).
4. **Limited Abstraction**: There's a lack of clear abstraction between layers. In a layered architecture, layers should interact through well-defined interfaces, but in this codebase, the interaction is direct, leading to tight coupling.

Given these considerations, an alternative design pattern that might be more suitable is the Model-View-Controller (MVC) pattern. In MVC:

- **Model**: Represents the data and business logic of the application. It encapsulates the behavior of the application and interacts with the data layer.
- **View**: Represents the presentation layer of the application. It is responsible for rendering the user interface based on the data provided by the model.
- **Controller**: Acts as an intermediary between the model and the view. It handles user input, updates the model based on user actions, and updates the view accordingly.

Applying MVC to the codebase would involve:

- Separating the business logic related to event management into a distinct model layer.
- Implementing the GUI presentation and user interaction logic in the view layer.
- Introducing controllers to handle user inputs and coordinate interactions between the model and the view.

This separation of concerns would lead to a more modular, maintainable, and testable codebase compared to a layered architecture, given the current structure and requirements of the application.

Modularity or Separation of Concerns

The provided codebase does not fully adhere to modularity or separation of concerns. While there is an attempt to organize functionality into different methods and classes, the overall structure still lacks clear separation of concerns. Here's why:

1. **Mixing of Concerns**: The `CalendarGUI` class, which represents the GUI and user interaction logic, also contains significant portions of business logic related to event management. This mixing of concerns violates the principle of separation of concerns, making it challenging to understand and maintain the codebase.
2. **Limited Modularity**: Although the codebase is divided into methods and classes, the boundaries between modules are not clearly defined. For example, there are methods within `CalendarGUI` that handle both GUI rendering and event management tasks, leading to a lack of modularity.
3. **Tight Coupling**: There is a high degree of coupling between the GUI components and the event management logic. Ideally, modules should be loosely coupled, but in this codebase, changes in one area could have unintended effects on other areas due to tight coupling.
4. **Single Responsibility Principle (SRP)**: Many methods and classes in the codebase violate the SRP, which states that a

class or method should have only one reason to change. For example, the `CalendarGUI` class handles both GUI presentation and event management, violating the SRP.

In summary, while there is an attempt to organize functionality into methods and classes, the codebase lacks clear modularity and separation of concerns. To improve modularity and separation of concerns, it would be beneficial to refactor the codebase to clearly separate GUI-related logic from business logic and adhere more closely to principles like the SRP. This could involve restructuring the codebase into separate modules or classes for GUI presentation, event management, and possibly data access and persistence.