AI Query

I'm working with a complex interface in Python Tkinter with a deep hierarchy.

- The top level is a `Window(Tk)` class, and
- contains a `MainFrame(Frame)`.
- Within `MainFrame`, there are two sub-frames:
    - `PreviewFrame`, and
    - `ViewFrame`.
- Several levels down in `ViewFrame`, each layer of which is another Frame, there is:
    - a `VideoImageInfoSet` Frame containing two Frames:
        - `ResolutionSet`, and
        - `ProjectionSet`.

Each of these `-Set` Frames contains an `OptionMenu`.

Each of these Frames is in its own module.

There is also an `APSettings` module for storing property settings universally for the application.

- Any time the user changes one of the UI elements, `APSettings` needs to know so it can update the relevant property within itself.
- Also, when the application first starts up, `APSettings` needs to wait until the entire UI is drawn before updating its properties from a .json file saved last time the application closed.

I'm thinking an observer pattern would work best for this type of communication between modules. I'm also thinking that `APSettings` needs to be both an *observer* and an *observable*. That would allow it to:

- be notified when the UI is finished drawing,
- notified of any changes in the `OptionMenus` in:
    - `ResolutionSet`, and
    - `ProjectionSet`,
- and also notify all configurable UI elements (including `ResolutionSet` and `ProjectionSet`, but not limited to those two) when they need to update themselves because of changes in `APSettings` property settings.

Outline the code in separate modules so I can easily expand on it and adapt it to my use.

Claude

Certainly, I can help you outline the code for this complex *Tkinter* interface using the observer pattern. I'll break it down into separate modules for clarity. Here's a basic structure you can expand upon:

1. First, let's create a basic observer pattern:

observer.py

```
class Observable:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, *args, **kwargs):
        for observer in self._observers:
            observer.update(*args, **kwargs)

class Observer:
    def update(self, *args, **kwargs):
        pass
```

2. Now, let's create the APSettings module:

ap_settings.py

```
import json
from observer import Observable, Observer

class APSettings(Observable, Observer):
    def __init__(self):
        Observable.__init__(self)
        self.properties = {}
        self.ui_ready = False

    def load_settings(self):
        with open('settings.json', 'r') as f:
            self.properties = json.load(f)
        if self.ui_ready:
            self.notify_observers('settings_loaded')

    def save_settings(self):
        with open('settings.json', 'w') as f:
            json.dump(self.properties, f)

    def update(self, *args, **kwargs):
        if args[0] == 'ui_ready':
            self.ui_ready = True
            self.load_settings()
        elif args[0] == 'option_changed':
            self.properties[args[1]] = args[2]
            self.save_settings()

    def get_property(self, key):
        return self.properties.get(key)

ap_settings = APSettings()
```

3. Now for the main window:

window.py

```
import tkinter as tk
from main_frame import MainFrame
from observer import Observable

class Window(tk.Tk, Observable):
    def __init__(self):
        super().__init__()
        Observable.__init__(self)
        self.main_frame = MainFrame(self)
        self.main_frame.pack(fill = tk.BOTH, expand = True)
        self.after(100, self.notify_ui_ready)

    def notify_ui_ready(self):
        self.notify_observers('ui_ready')
```

4. Main frame:

main_frame.py

```
import tkinter as tk
from preview_frame import PreviewFrame
from view_frame import ViewFrame

class MainFrame(tk.Frame):
   def __init__(self, master):
      super().__init__(master)
      self.preview_frame = PreviewFrame(self)
      self.view_frame = ViewFrame(self)

      self.preview_frame.pack(side = tk.LEFT,
                              fill = tk.BOTH,
                              expand = True)

      self.view_frame.pack(side = tk.RIGHT,
                           fill = tk.BOTH,
                           expand = True)
```

5. View frame (simplified, as it has multiple layers):

view_frame.py

```
import tkinter as tk
from video_image_info_set import VideoImageInfoSet

class ViewFrame(tk.Frame):
   def __init__(self, master):
      super().__init__(master)
      self.video_image_info_set = VideoImageInfoSet(self)
      self.video_image_info_set.pack(fill = tk.BOTH, expand = True)
```

6. VideoImageInfoSet frame:

video_image_info_set.py

```
import tkinter as tk
from resolution_set import ResolutionSet
from projection_set import ProjectionSet
```

```
class VideoImageInfoSet(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.resolution_set = ResolutionSet(self)
        self.projection_set = ProjectionSet(self)
        self.resolution_set.pack(fill = tk.X)
        self.projection_set.pack(fill = tk.X)
```

7. ResolutionSet frame:

resolution_set.py

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import ap_settings

class ResolutionSet(tk.Frame, Observer):
    def __init__(self, master):
        super().__init__(master)
        self.resolution_var = tk.StringVar()

        self.resolution_menu = ttk.OptionMenu(self,
            self.resolution_var, "Select Resolution",
           "1920x1080", "1280x720", "640x480",
           command = self.on_resolution_change)

        self.resolution_menu.pack()
        ap_settings.add_observer(self)

    def on_resolution_change(self, *args):
        ap_settings.update('option_changed', 'resolution',
            self.resolution_var.get())

    def update(self, *args, **kwargs):
        if args[0] == 'settings_loaded':
            self.resolution_var.set(
                ap_settings.get_property('resolution'))
```

8. ProjectionSet frame (similar to ResolutionSet):

projection_set.py

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import ap_settings

class ProjectionSet(tk.Frame, Observer):
  def __init__(self, master):
     super().__init__(master)
     self.projection_var = tk.StringVar()

     self.projection_menu = ttk.OptionMenu(self,
       self.projection_var, "Select Projection",
       "Perspective", "Orthographic",
       command = self.on_projection_change)

     self.projection_menu.pack()
     ap_settings.add_observer(self)

  def on_projection_change(self, *args):
     ap_settings.update('option_changed', 'projection',
       self.projection_var.get())

  def update(self, *args, **kwargs):
     if args[0] == 'settings_loaded':

  self.projection_var.set(ap_settings.get_property('projection'))
```

9. Finally, the main script:

main.py

```
from window import Window
from ap_settings import ap_settings

if __name__ == "__main__":
  window = Window()
  ap_settings.add_observer(window)
  window.mainloop()
```

This structure implements the observer pattern and allows for the communication you described. `APSettings` acts as both an observer and an observable. It observes the UI for changes and the UI ready state, and notifies UI elements when settings are loaded.

You can expand on this structure by adding more UI elements and connecting them to `APSettings` as needed. Remember to add error handling, more robust state management, and any additional features specific to your application.

ChatGPT

Here’s an outline of the observer pattern code across modules for your *Tkinter* project. Each module communicates with others using the observer pattern, allowing `APSettings` to observe and be observed.

Module: APSettings.py

```
class APSettings:
  _instance = None

  def __new__(cls):
    if cls._instance is None:
      cls._instance = super(APSettings, cls).__new__(cls)
      cls._instance._observers = []
      cls._instance.properties = {}
    return cls._instance

  def add_observer(self, observer):
    self._observers.append(observer)

  def notify_observers(self, key):
    for observer in self._observers:
      observer.update(key, self.properties.get(key))

  def update_property(self, key, value):
    self.properties[key] = value
    self.notify_observers(key)

  def update_from_json(self, json_file):
    # Load from JSON, then notify observers of updates
    pass

  def notify_ui_loaded(self):
    # Called when the UI finishes loading
    self.notify_observers("ui_loaded")

## Singleton pattern ensures APSettings is globally accessible.
ap_settings = APSettings()
```

Module: ResolutionSet.py

```
import tkinter as tk
from APSettings import ap_settings
```

```python
class ResolutionSet(tk.Frame):
  def __init__(self, parent, *args, **kwargs):
    super().__init__(parent, *args, **kwargs)

    self.option_menu = tk.OptionMenu(self, tk.StringVar(),
      "1920x1080",     "1280x720")

    self.option_menu.pack()

    ap_settings.add_observer(self)

    # Track user selection
    self.option_menu_var = tk.StringVar()
    self.option_menu_var.trace("w", self.on_option_changed)

  def on_option_changed(self, *args):
    selected_value = self.option_menu_var.get()
    ap_settings.update_property("resolution", selected_value)

  def update(self, key, value):
    if key == "resolution":
      self.option_menu_var.set(value)
    elif key == "ui_loaded":
      self.load_initial_value()

  def load_initial_value(self):
    # Update OptionMenu based on APSettings
    resolution = ap_settings.properties.get("resolution",
      "1920x1080")

    self.option_menu_var.set(resolution)
```

Module: ProjectionSet.py

```python
import tkinter as tk
from APSettings import ap_settings

class ProjectionSet(tk.Frame):
  def __init__(self, parent, *args, **kwargs):
    super().__init__(parent, *args, **kwargs)

    self.option_menu = tk.OptionMenu(self, tk.StringVar(),
      "Perspective", "Orthographic")

    self.option_menu.pack()
    ap_settings.add_observer(self)

    # Track user selection
    self.option_menu_var = tk.StringVar()
    self.option_menu_var.trace("w", self.on_option_changed)

  def on_option_changed(self, *args):
    selected_value = self.option_menu_var.get()
    ap_settings.update_property("projection", selected_value)

  def update(self, key, value):
    if key == "projection":
      self.option_menu_var.set(value)
    elif key == "ui_loaded":
      self.load_initial_value()

  def load_initial_value(self):
    # Update OptionMenu based on APSettings

    projection = ap_settings.properties.get("projection",
      "Perspective")

    self.option_menu_var.set(projection)
```

Module: MainFrame.py

```
import tkinter as tk
from ResolutionSet import ResolutionSet
from ProjectionSet import ProjectionSet
from APSettings import ap_settings

class MainFrame(tk.Frame):
  def __init__(self, parent, *args, **kwargs):
    super().__init__(parent, *args, **kwargs)

    self.resolution_set = ResolutionSet(self)
    self.projection_set = ProjectionSet(self)

    self.resolution_set.pack(side = "left")
    self.projection_set.pack(side = "right")

    # Notify APSettings when UI is fully loaded
    self.after(0, self.notify_ui_loaded)

  def notify_ui_loaded(self):
    ap_settings.notify_ui_loaded()
```

Module: App.py

```
import tkinter as tk
from MainFrame import MainFrame
from APSettings import ap_settings

class App(tk.Tk):
  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)

    self.main_frame = MainFrame(self)
    self.main_frame.pack()

    # Load saved properties from a JSON file
    ap_settings.update_from_json("settings.json")

if __name__ == "__main__":
  app = App()
  app.mainloop()
```

Key Points:

- `APSettings` is a *Singleton*, managing settings across the app.
- Each UI module (e.g., `ResolutionSet`, `ProjectionSet`) updates `APSettings` and listens for changes to update itself.
- `MainFrame` signals when the UI is fully drawn, allowing `APSettings` to update from saved values.

Gemini

Understanding the Problem

Before diving into the code, let's recap the problem:

- Complex `Tkinter` hierarchy.
- Need for communication between different UI elements and a central settings module (`APSettings`).

- `APSettings` needs to be updated after UI is fully drawn.
- *Observer* pattern seems suitable for communication.

Proposed Structure

We'll use the following modules:

- `observer.py`: Defines the *Observer* and *Observable* interfaces.
- `apsettings.py`: Implements `APSettings` as both observer and observable.
- `window.py`: Contains the main window and frame hierarchy.
- `resolution_set.py`: Contains the `ResolutionSet` frame.
- `projection_set.py`: Contains the `ProjectionSet` frame.
- (Other modules for remaining frames as needed)

observer.py

```
class Observer:
    def update(self, observable, *args, **kwargs):
        raise NotImplementedError

class Observable:
    def __init__(self):
        self._observers = set()

    def attach(self, observer):
        self._observers.add(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, *args, **kwargs):
        for observer in self._observers:
            observer.update(self,   *args, **kwargs)
```

apsettings.py

```
import json
from observer import Observable, Observer

class APSettings(Observable, Observer):
    def __init__(self):
        Observable.__init__(self)
        Observer.__init__(self)
        self.settings = {}

    def load_settings(self):
        # Load settings from JSON file
        pass

    def save_settings(self):
        # Save settings to JSON file
        pass

    def update(self, observable, *args, **kwargs):
        # Update internal settings based on changes from observable
        pass

    def notify_settings_changed(self, property_name):
        # Notify observers about changes in specific property
        pass
```

window.py

```
import tkinter as tk
from apsettings import APSettings
from resolution_set import ResolutionSet
from projection_set import ProjectionSet

class Window(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.apsettings = APSettings()
        self.main_frame = MainFrame(self)
        # ... rest of the window setup

class MainFrame(tk.Frame):
    def __init__(self, master):
        tk.Frame.__init__(self, master)
        # ... rest of the frame setup

class ViewFrame(tk.Frame):
    def __init__(self, master):
        tk.Frame.__init__(self, master)
        # ... rest of the frame setup

# ... other frame classes
```

resolution_set.py

```
import tkinter as tk
from observer import Observer
from apsettings import APSettings

class ResolutionSet(tk.Frame, Observer):
  def __init__(self, master, apsettings):
     tk.Frame.__init__(self, master)
     Observer.__init__(self)
     self.apsettings = apsettings
     self.apsettings.attach(self)
     # ... create OptionMenu and other UI elements

  def update(self, observable, *args, **kwargs):
     # Update UI based on changes in APSettings
     pass

  def on_resolution_change(self, new_resolution):
     # Update APSettings and notify observers
     self.apsettings.update_resolution(new_resolution)
```

projection_set.py

Similar structure to `resolution_set.py`

Key Points

- `APSettings` is both an *Observer* and *Observable* to facilitate communication.
- UI elements implement *Observer* to react to changes in `APSettings`.
- UI elements notify `APSettings` when their values change.
- Consider using a dedicated event bus for more complex communication scenarios.
- Implement proper error handling and validation.

Additional Considerations:

- For large-scale applications, consider using a dedicated state management library or framework.
- Optimize performance by avoiding unnecessary updates and using efficient data structures.
- Thoroughly test the observer pattern implementation to ensure correct behavior.

By following this structure, you can create a maintainable and scalable application with clear separation of concerns.

Revision

Using custom state management

Custom State Management for Complex Tkinter Interface

Understanding the Problem

Before diving into the implementation, let's recap the specific challenges we're addressing:

- **Complex UI hierarchy:** Multiple frames and components interacting with each other.
- **Centralized settings:** `APSettings` needs to store and manage application-wide properties.
- **Observer pattern:** For communication between UI elements and `APSettings`.
- **UI updates:** UI elements need to react to changes in `APSettings`.
- **Initial state loading:** `APSettings` needs to load initial settings from a *JSON* file after UI is drawn.

Custom State Management Approach

We can create a custom state management solution using a combination of *Python* dictionaries, events, and observers.

StateManager Class

```
import tkinter as tk

class StateManager:
  def __init__(self):
    self._state = {}
    self._observers = []

  def set_state(self, key, value):
    self._state[key] = value
    self._notify_observers(key)

  def get_state(self, key):
    return self._state.get(key)

  def attach(self, observer):
    self._observers.append(observer)

  def _notify_observers(self, changed_key):
    for observer in self._observers:
      observer.update(self, changed_key)
```

APSettings Class

```
import json
from state_manager import StateManager

class APSettings(StateManager):
 def __init__(self):
     super().__init__()
     self.load_settings()

 def load_settings(self):
     # Load settings from JSON file
     pass

 def save_settings(self):
     # Save settings to JSON file
     pass
```

UI Elements as Observers

```
from state_manager import Observer

class ResolutionSet(tk.Frame, Observer):
    def __init__(self, master, state_manager):
        tk.Frame.__init__(self, master)
        self.state_manager = state_manager
        self.state_manager.attach(self)
        # ... create OptionMenu and other UI elements

    def update(self, observable, changed_key):
        if changed_key == 'resolution':
            # Update UI based on changed resolution
            pass
```

Integration

- Create a `StateManager` instance.
- Pass the `StateManager` instance to `APSettings` and all UI components that need to interact with the state.
- `APSettings` uses `set_state` to update its properties.
- UI components implement the `Observer` interface and react to state changes based on the `changed_key` parameter.
- For initial state loading, trigger a `notify_observers()` call after the UI is fully drawn.

Key Points

- The `StateManager` acts as a central repository for application state.
- The `Observer` pattern enables efficient communication between components.
- Using `changed_key()` in the `update` method optimizes performance by only updating components that are affected by the change.
- Consider using a more robust event system for complex scenarios.

- For large-scale applications, explore using a dedicated state management library.

By following this approach, you can effectively manage state in your Tkinter application without relying on external libraries.

Perplexity

Your approach using the observer pattern for this complex *Tkinter* interface is a good choice. It will help maintain loose coupling between components and allow for efficient communication across the application. Here's an outline of how you could structure this using separate modules:

1. observer.py

This module defines the base `Observer` and `Observable` classes.

```
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, observable, *args, **kwargs):
        pass

class Observable:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, *args, **kwargs):
        for observer in self._observers:
            observer.update(self, *args, **kwargs)
```

2. ap_settings.py

This module defines the `APSettings` class, which is both an `Observer` and an `Observable`.

```
import json
from observer import Observer, Observable

class APSettings(Observer, Observable):
    def __init__(self):
```

```
    Observable.__init__(self)
    self._settings = {}
    self._ui_ready = False

  def load_settings(self):
    with open('settings.json', 'r') as f:
      self._settings = json.load(f)
    if self._ui_ready:
      self.notify_observers('settings_loaded')

  def save_settings(self):
    with open('settings.json', 'w') as f:
      json.dump(self._settings, f)

  def update(self, observable, *args, **kwargs):
    if args[0] == 'ui_ready':
      self._ui_ready = True
      self.load_settings()
    elif args[0] == 'option_changed':
      self._settings[args[1]] = args[2]
      self.save_settings()

  def get_setting(self, key):
    return self._settings.get(key)

ap_settings = APSettings()  # Singleton instance
```

3. window.py

This module defines the main `Window` class.

```
import tkinter as tk
from main_frame import MainFrame
from ap_settings import ap_settings

class Window(tk.Tk):
  def __init__(self):
    super().__init__()
    self.main_frame = MainFrame(self)
    self.main_frame.pack(fill = tk.BOTH, expand = True)
    self.after(100, self.notify_ui_ready)

  def notify_ui_ready(self):
    ap_settings.update(self, 'ui_ready')
```

4. main_frame.py

This module defines the `MainFrame` class.

```
import tkinter as tk
from preview_frame import PreviewFrame
from view_frame import ViewFrame

class MainFrame(tk.Frame):
   def __init__(self, master):
      super().__init__(master)
      self.preview_frame = PreviewFrame(self)
      self.view_frame = ViewFrame(self)

      self.preview_frame.pack(side = tk.LEFT,
         fill = tk.BOTH, expand = True)

      self.view_frame.pack(side = tk.RIGHT,
         fill = tk.BOTH, expand = True)
```

5. view_frame.py

This module defines the `ViewFrame` class.

```
import tkinter as tk
from video_image_info_set import VideoImageInfoSet

class ViewFrame(tk.Frame):
   def __init__(self, master):
      super().__init__(master)
      self.video_image_info_set = VideoImageInfoSet(self)
      self.video_image_info_set.pack()
```

6. video_image_info_set.py

This module defines the `VideoImageInfoSet` class.

```
import tkinter as tk
from resolution_set import ResolutionSet
from projection_set import ProjectionSet

class VideoImageInfoSet(tk.Frame):
   def __init__(self, master):
      super().__init__(master)
      self.resolution_set = ResolutionSet(self)
      self.projection_set = ProjectionSet(self)
      self.resolution_set.pack()
      self.projection_set.pack()
```

7. resolution_set.py

This module defines the `ResolutionSet` class.

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import ap_settings

class ResolutionSet(tk.Frame, Observer):
    def __init__(self, master):
        tk.Frame.__init__(self, master)
        Observer.__init__(self)
        self.resolution_var = tk.StringVar()

        self.resolution_menu = ttk.OptionMenu(self,
            self.resolution_var, "Select Resolution",
            "1080p", "720p", "480p",
            command = self.on_resolution_change)

        self.resolution_menu.pack()
        ap_settings.add_observer(self)

    def on_resolution_change(self, *args):
        ap_settings.update(self, 'option_changed',
        'resolution', self.resolution_var.get())

    def update(self, observable, *args, **kwargs):
        if args[0] == 'settings_loaded':
            self.resolution_var.set(
                ap_settings.get_setting('resolution'))
```

8. projection_set.py

This module is similar to `resolution_set.py` but for projection settings.

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import ap_settings

class ProjectionSet(tk.Frame, Observer):
    def __init__(self, master):
        tk.Frame.__init__(self, master)
        Observer.__init__(self)
        self.projection_var = tk.StringVar()
        self.projection_menu = ttk.OptionMenu(
            self, self.projection_var, "Select Projection",
            "Perspective", "Orthographic",
            command = self.on_projection_change)

        self.projection_menu.pack()
        ap_settings.add_observer(self)

    def on_projection_change(self, *args):
        ap_settings.update(self, 'option_changed', 'projection',
            self.projection_var.get())

    def update(self, observable, *args, **kwargs):
        if args[0] == 'settings_loaded':
            self.projection_var.set(
                ap_settings.get_setting('projection'))
```

9. main.py

This is the entry point of your application.

```
from window import Window

if __name__ == "__main__":
    app = Window()
    app.mainloop()
```

This structure allows for easy communication between components:

1. When the UI is ready, `Window` notifies `APSettings`.
2. `APSettings` then loads the settings and notifies all observers (including `ResolutionSet` and `ProjectionSet`).
3. When a user changes an option in `ResolutionSet` or `ProjectionSet`, it notifies `APSettings`.
4. `APSettings` updates its internal state and saves the settings.

You can expand on this structure by adding more UI elements and having them observe `APSettings` for changes. Remember to add any new configurable UI elements as observers to `APSettings`.

Revision

ap_settings.py (Revised)

Let's correct this and implement `APSettings` as a proper *Singleton*. Here's a revised version of the *ap_settings.py* module:

```
import json
from observer import Observer, Observable

class APSettings(Observer, Observable):
   _instance = None

   def __new__(cls):
      if cls._instance is None:
         cls._instance = super(APSettings, cls).__new__(cls)
         cls._instance._initialized = False
      return cls._instance

   def __init__(self):
      if self._initialized:
         return
      Observable.__init__(self)
      self._settings = {}
      self._ui_ready = False
      self._initialized = True

   def load_settings(self):
      try:
         with open('settings.json', 'r') as f:
            self._settings = json.load(f)
      except FileNotFoundError:
         self._settings = {}  # Use default settings if file
                               # doesn't exist
      if self._ui_ready:
         self.notify_observers('settings_loaded')

   def save_settings(self):
      with open('settings.json', 'w') as f:
         json.dump(self._settings, f)

   def update(self, observable, *args, **kwargs):
      if args[0] == 'ui_ready':
         self._ui_ready = True
         self.load_settings()
      elif args[0] == 'option_changed':
         self._settings[args[1]] = args[2]
         self.save_settings()
```

```
  def get_setting(self, key):
    return self._settings.get(key)

## No need to instantiate here
```

window.py (Revised)

Now, to use this *Singleton* in other modules, you would import and instantiate it like this:

```
import tkinter as tk
from main_frame import MainFrame
from ap_settings import APSettings

class Window(tk.Tk):
  def __init__(self):
    super().__init__()
    self.ap_settings = APSettings()  # Get the singleton instance
    self.main_frame = MainFrame(self)
    self.main_frame.pack(fill = tk.BOTH, expand = True)
    self.after(100, self.notify_ui_ready)

  def notify_ui_ready(self):
    self.ap_settings.update(self, 'ui_ready')
```

resolution_set.py (Revised)

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import APSettings

class ResolutionSet(tk.Frame, Observer):
  def __init__(self, master):
    tk.Frame.__init__(self, master)
    Observer.__init__(self)
    self.ap_settings = APSettings()  # Get the singleton instance
    self.resolution_var = tk.StringVar()

    self.resolution_menu = ttk.OptionMenu(
      self, self.resolution_var, "Select Resolution",
      "1080p", "720p", "480p",
      command = self.on_resolution_change)

    self.resolution_menu.pack()
    self.ap_settings.add_observer(self)
    def on_resolution_change(self, *args):
        self.ap_settings.update(self, 'option_changed',
        'resolution', self.resolution_var.get())

    def update(self, observable, *args, **kwargs):
        if args[0] == 'settings_loaded':
            self.resolution_var.set(
                self.ap_settings.get_setting('resolution'))
```

projection_set.py (Revised)

```
import tkinter as tk
from tkinter import ttk
from observer import Observer
from ap_settings import APSettings

class ProjectionSet(tk.Frame, Observer):
    def __init__(self, master):
        tk.Frame.__init__(self, master)
        Observer.__init__(self)
        self.ap_settings = APSettings()  # Get the singleton instance
        self.projection_var = tk.StringVar()

        self.projection_menu = ttk.OptionMenu(
            self, self.projection_var, "Select Projection",
            "Perspective", "Orthographic",
            command = self.on_projection_change)

        self.projection_menu.pack()
        self.ap_settings.add_observer(self)

    def on_projection_change(self, *args):
        self.ap_settings.update(self, 'option_changed',
            'projection', self.projection_var.get())

    def update(self, observable, *args, **kwargs):
        if args[0] == 'settings_loaded':
            self.projection_var.set(
                self.ap_settings.get_setting('projection'))
```

In this revised version:

1. `APSettings` is implemented as a true *Singleton* using the `__new__()` method.
2. The *Singleton* instance is not created in the `ap_settings`.py module.

3. Each module that needs to use `APSettings` imports the class and creates/gets the *Singleton* instance when needed.

This approach ensures that only one instance of `APSettings` is created and used throughout the application, regardless of where and how many times it's instantiated. The first time `APSettings()` is called, it creates the instance. Subsequent calls return the existing instance.

Thank you for pointing this out. This implementation is more correct and in line with the *Singleton* pattern.

---

h2oGTP

Here's an outline of the code in separate modules using the *Observer* pattern.

observer.py

```
class Observer:
  def update(self, observable, *args, **kwargs):
    raise NotImplementedError
```

observable.py

```
class Observable:
  def __init__(self):
    self.observers = []

  def register_observer(self, observer):
    self.observers.append(observer)

  def remove_observer(self, observer):
    self.observers.remove(observer)

  def notify_observers(self, *args, **kwargs):
    for observer in self.observers:
      observer.update(self, *args, **kwargs)
```

ap_settings.py

```
import json
from observable import Observable
from observer import Observer
```

```
class APSettings(Observable, Observer):
    def __init__(self):
        super().__init__()
        self.properties = {}

    def load_properties(self, filename):
        with open(filename, 'r') as f:
            self.properties = json.load(f)
        self.notify_observers('load_properties')

    def save_properties(self, filename):
        with open(filename, 'w') as f:
            json.dump(self.properties, f)

    def update_property(self, name, value):
        self.properties[name] = value
        self.notify_observers('update_property', name, value)

    def update(self, observable, *args, **kwargs):
        if observable == 'ui_ready':
            self.load_properties('settings.json')
        elif observable == 'option_menu':
            self.update_property(*args, **kwargs)
```

ui_element.py

```python
from observer import Observer

class UIElement(Observer):
    def __init__(self, ap_settings):
        self.ap_settings = ap_settings
        self.ap_settings.register_observer(self)

    def update(self, observable, *args, **kwargs):
        if observable == 'load_properties':
            self.load_properties()
        elif observable == 'update_property':
            self.update_property(*args, **kwargs)

    def load_properties(self):
        raise NotImplementedError

    def update_property(self, name, value):
        raise NotImplementedError
```

resolution_set.py

```
from ui_element import UIElement
from tkinter import OptionMenu

class ResolutionSet(UIElement):
    def __init__(self, master, ap_settings):
        super().__init__(ap_settings)
        self.master = master

        self.option_menu = OptionMenu(master, 'option',
            command = self.option_menu_changed)

        self.option_menu.pack()

    def load_properties(self):
        # Load properties from APSettings
        pass

    def update_property(self, name, value):
        if name == 'resolution':
            self.option_menu.set(value)

    def option_menu_changed(self, value):
        self.ap_settings.update_property('resolution', value)
```

projection_set.py

```
from ui_element import UIElement
from tkinter import OptionMenu

class ProjectionSet(UIElement):
    def __init__(self, master, ap_settings):
        super().__init__(ap_settings)
        self.master = master

        self.option_menu = OptionMenu(master, 'option',
            command = self.option_menu_changed)

        self.option_menu.pack()

    def load_properties(self):
        # Load properties from APSettings
        pass

    def update_property(self, name, value):
        if name == 'projection':
            self.option_menu.set(value)

    def option_menu_changed(self, value):
        self.ap_settings.update_property('projection', value)
```

main.py

```
import tkinter as tk
from ap_settings import APSettings
from resolution_set import ResolutionSet
from projection_set import ProjectionSet

def main():
    root = tk.Tk()
    ap_settings = APSettings()

    # Create UI elements
    resolution_set = ResolutionSet(root, ap_settings)
    projection_set = ProjectionSet(root, ap_settings)

    # Notify APSettings when UI is ready
    def ui_ready():
        ap_settings.notify_observers('ui_ready')

    # Call ui_ready after UI is drawn
    root.after(100, ui_ready)

    root.mainloop()

if __name__ == '__main__':
    main()
```

This is a basic outline of how you can structure your code using the observer pattern. You can expand on this and adapt it to your specific needs.
