Exercice - Abstract Factory : Adaptation d'interface utilisateur multi-plateforme

Contexte

Une entreprise développe un logiciel interne accessible via Internet. Les employés utilisent différents systèmes d'exploitation (Windows, macOS, Linux) pour accéder à cette application.

Chaque système d'exploitation a ses propres conventions d'interface utilisateur, habitudes d'interaction et dispositions qui lui sont propres. Par exemple :

- **Windows** : Boutons avec bordures prononcées, menus déroulants avec séparateurs visibles, icônes standards Windows
- **macOS** : Design minimaliste, coins arrondis, contrôles plus espacés, animations fluides
- **Linux** : Dépend de l'environnement de bureau (GNOME, KDE, etc.), mais généralement plus personnalisable

Créer une application identique pour tous les systèmes d'exploitation risque de produire une interface peu intuitive pour certains utilisateurs, car elle ne respecterait pas les conventions auxquelles ils sont habitués. Cependant, l'entreprise souhaite développer rapidement avec une seule base de code.

Partie 1 : Analyse du code existant problématique

Voici un exemple simplifié du code actuel, qui présente de nombreuses répétitions et une structure rigide :

```
import platform
import tkinter as tk

class Application:
    def __init__(self, master):
        self.master = master
        self.os_type = platform.system()
        self.setup_ui()

    def setup_ui(self):
        if self.os_type == "Windows":
            self.create_windows_button()
            self.create_windows_menu()
        elif self.os_type == "Darwin":
            self.create_mac_button()
            self.create_mac_menu()
        else:
            self.create_linux_button()
            self.create_linux_menu()

    def create_windows_button(self):
        btn = tk.Button(self.master, text="Cliquez-moi", bg="lightblue",
                        relief="raised", borderwidth=2)
        btn.pack(pady=10)
        return btn

    def create_mac_button(self):
        btn = tk.Button(self.master, text="Cliquez-moi", bg="white",
                        relief="flat", borderwidth=1)
        btn.pack(pady=10)
        return btn

    def create_linux_button(self):
        btn = tk.Button(self.master, text="Cliquez-moi", bg="gray90",
                        relief="ridge", borderwidth=1)
        btn.pack(pady=10)
        return btn

    def create_windows_menu(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=self.dummy_function)
        file_menu.add_command(label="Ouvrir", command=self.dummy_function)
        file_menu.add_separator()
        file_menu.add_command(label="Quitter", command=self.master.quit)
        menu.add_cascade(label="Fichier", menu=file_menu)
```

```
        self.master.config(menu=menu)

    def create_mac_menu(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=self.dummy_function)
        file_menu.add_command(label="Ouvrir", command=self.dummy_function)
        file_menu.add_separator()
        file_menu.add_command(label="Quitter", command=self.master.quit)
        menu.add_cascade(label="Fichier", menu=file_menu)
        self.master.config(menu=menu)

    def create_linux_menu(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=self.dummy_function)
        file_menu.add_command(label="Ouvrir", command=self.dummy_function)
        file_menu.add_separator()
        file_menu.add_command(label="Quitter", command=self.master.quit)
        menu.add_cascade(label="Fichier", menu=file_menu)
        self.master.config(menu=menu)

    def dummy_function(self):
        print("Fonction appelée")

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Application")
    app = Application(root)
    root.mainloop()
```

Questions sur le code existant:

1. **Identifiez les problèmes dans cette implémentation.** Quelles sont les difficultés qui surgiraient si vous deviez ajouter un nouveau composant d'interface (par exemple, une barre d'outils) ou un nouveau système d'exploitation à prendre en charge?
2. **Proposez une modification mineure:** Ajoutez un composant "Formulaire" avec des champs de texte pour chaque système d'exploitation sans refactoriser l'ensemble du code. En quoi cette approche est-elle fastidieuse?

Partie 2 : Introduction au patron de conception Abstract Factory

Le patron de conception Abstract Factory offre une solution élégante à ce problème. Il permet de créer des familles d'objets liés sans avoir à spécifier leurs classes concrètes.

Dans notre contexte, chaque système d'exploitation représente une "famille" de composants d'interface utilisateur qui doivent être cohérents entre eux. L'Abstract Factory nous permettrait de:

- Isoler le code client des classes concrètes
- Assurer que les composants créés sont compatibles entre eux
- Faciliter l'ajout de nouveaux systèmes d'exploitation ou composants

Questions de conception:

1. **Réalisez un diagramme UML** illustrant comment le patron Abstract Factory pourrait être appliqué à notre problème. Incluez au minimum les composants "Bouton" et "Menu" pour les différents systèmes d'exploitation.
2. **Implémentez** la solution basée sur le patron Abstract Factory. Votre code doit permettre de créer facilement des interfaces adaptées à chaque système d'exploitation.

Partie 3 : Réflexion et extension

1. **Comparaison:** Comparez votre implémentation avec Abstract Factory à l'implémentation originale. Quels avantages apporte votre solution en termes de:
    - Maintenabilité
    - Extensibilité
    - Lisibilité
    - Respect des principes SOLID
2. **Extension:** Comment pourriez-vous étendre votre solution pour prendre en charge:
    - Un nouveau système d'exploitation (par exemple, Android)
    - Un nouveau composant d'interface (par exemple, une boîte de dialogue)
    - Des thèmes personnalisés pour chaque système d'exploitation