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

Partie 1 : Analyse du code existant problématique

Question 1 : Problèmes dans l'implémentation

Principaux problèmes identifiés :

1. **Duplication excessive de code** : Les méthodes de création des menus (`create_windows_menu`, `create_mac_menu`, `create_linux_menu`) contiennent exactement le même code.
2. **Mauvaise extensibilité** : Pour ajouter un nouveau composant ou système d'exploitation, il faudrait modifier la classe principale et ajouter de nombreuses méthodes.
3. **Violation du principe ouvert/fermé** : La classe n'est pas ouverte à l'extension sans modification directe.
4. **Structure conditionnelle rigide** : L'utilisation de conditions if/elif/else pour déterminer le comportement rend le code difficile à maintenir.
5. **Couplage fort** : La classe `Application` connaît tous les détails d'implémentation des composants spécifiques.
6. **Non-respect du principe de responsabilité unique** : La classe `Application` gère à la fois la détection du système d'exploitation et la création des éléments d'interface.

Si on devait ajouter un nouveau composant ou un nouveau système d'exploitation, il faudrait :

- Ajouter de nouvelles méthodes pour chaque système
- Modifier la méthode `setup_ui` pour inclure ces nouveaux composants
- Dupliquer à nouveau du code similaire

Question 2 : Ajout d'un composant "Formulaire" sans refactorisation

Pour ajouter un composant "Formulaire" au code existant, il faudrait :

```
def create_windows_form(self):
    form_frame = tk.Frame(self.master, relief="raised", borderwidth=2)
    tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=5, pady=5
    tk.Entry(form_frame).grid(row=0, column=1, padx=5, pady=5)
    tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=5, pady
    tk.Entry(form_frame).grid(row=1, column=1, padx=5, pady=5)
    form_frame.pack(padx=10, pady=10)
    return form_frame

def create_mac_form(self):
    form_frame = tk.Frame(self.master, relief="flat", borderwidth=1)
    tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=8, pady=8
    tk.Entry(form_frame).grid(row=0, column=1, padx=8, pady=8)
    tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=8, pady
    tk.Entry(form_frame).grid(row=1, column=1, padx=8, pady=8)
    form_frame.pack(padx=15, pady=15)
    return form_frame

def create_linux_form(self):
    form_frame = tk.Frame(self.master, relief="ridge", borderwidth=1)
    tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=6, pady=6
    tk.Entry(form_frame).grid(row=0, column=1, padx=6, pady=6)
    tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=6, pady
    tk.Entry(form_frame).grid(row=1, column=1, padx=6, pady=6)
    form_frame.pack(padx=12, pady=12)
    return form_frame
```

Et il faudrait modifier la méthode `setup_ui` :

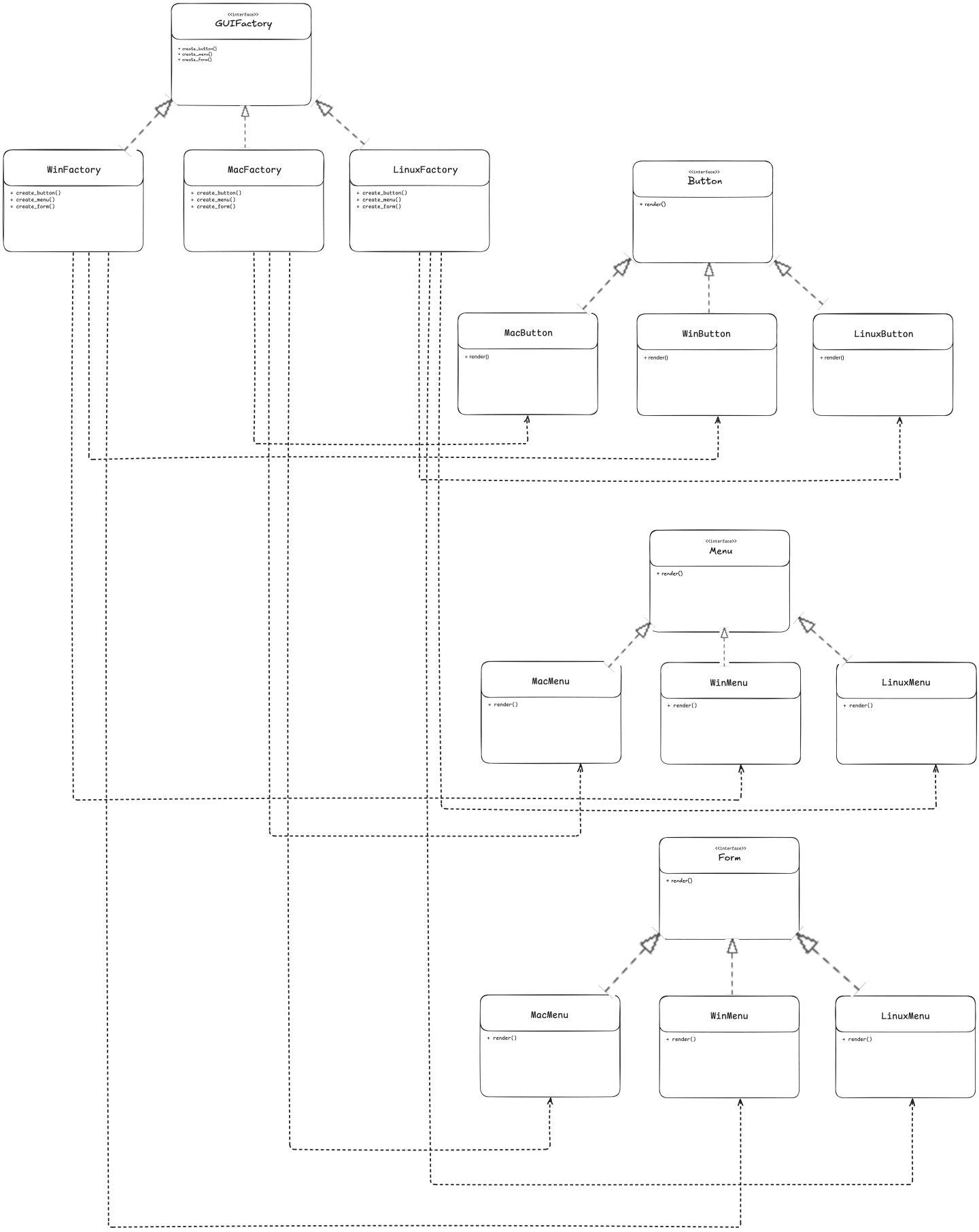
```
def setup_ui(self):
    if self.os_type == "Windows":
        self.create_windows_button()
        self.create_windows_menu()
        self.create_windows_form()
    elif self.os_type == "Darwin":
        self.create_mac_button()
        self.create_mac_menu()
        self.create_mac_form()
    else: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

Cette approche est fastidieuse pour plusieurs raisons :

- Duplication de code très similaire pour chaque système
- Nécessité de modifier la méthode conditionnelle principale
- Risque d'incohérence entre les implémentations
- Maintenance difficile (correction de bugs, modifications)
- Difficulté à réutiliser des composants similaires
- Le code devient de plus en plus volumineux et difficile à comprendre

Partie 2 : Implémentation avec le patron Abstract Factory

Question 1 : Réalisez un diagramme UML



Question 2 : Implémentez

```
import platform
import tkinter as tk
from abc import ABC, abstractmethod

# ========== ABSTRACT FACTORY ==========
class GUIFactory(ABC):
    @abstractmethod
    def create_button(self, master):
        pass

    @abstractmethod
    def create_menu(self, master):
        pass

    @abstractmethod
    def create_form(self, master):
        pass

# ========== CONCRETE FACTORIES ==========
class WinFactory(GUIFactory):
    def create_button(self, master):
        return WinButton(master)

    def create_menu(self, master):
        return WinMenu(master)

    def create_form(self, master):
        return WinForm(master)

class MacFactory(GUIFactory):
    def create_button(self, master):
        return MacButton(master)

    def create_menu(self, master):
        return MacMenu(master)

    def create_form(self, master):
        return MacForm(master)

class LinuxFactory(GUIFactory):
    def create_button(self, master):
        return LinuxButton(master)

    def create_menu(self, master):
```

```
        return LinuxMenu(master)

    def create_form(self, master):
        return LinuxForm(master)

# ========== ABSTRACT PRODUCTS ==========
class Button(ABC):
    def __init__(self, master):
        self.master = master

    @abstractmethod
    def render(self):
        pass

class Menu(ABC):
    def __init__(self, master):
        self.master = master

    @abstractmethod
    def render(self):
        pass

class Form(ABC):
    def __init__(self, master):
        self.master = master

    @abstractmethod
    def render(self):
        pass

# ========== CONCRETE PRODUCTS ==========
class WinButton(Button):
    def render(self):
        btn = tk.Button(self.master, text="Cliquez-moi", bg="lightblue",
                        relief="raised", borderwidth=2)
        btn.pack(pady=10)
        return btn

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

class LinuxButton(Button):
```

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

class WinMenu(Menu):
    def render(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=lambda: print("Fonction appelée
        file_menu.add_command(label="Ouvrir", command=lambda: print("Fonction appelée"
        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)
        return menu

class MacMenu(Menu):
    def render(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=lambda: print("Fonction appelée
        file_menu.add_command(label="Ouvrir", command=lambda: print("Fonction appelée"
        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)
        return menu

class LinuxMenu(Menu):
    def render(self):
        menu = tk.Menu(self.master)
        file_menu = tk.Menu(menu, tearoff=0)
        file_menu.add_command(label="Nouveau", command=lambda: print("Fonction appelée
        file_menu.add_command(label="Ouvrir", command=lambda: print("Fonction appelée"
        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)
        return menu

class WinForm(Form):
    def render(self):
        form_frame = tk.Frame(self.master, relief="raised", borderwidth=2)
        tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=5, pa
```

```
        tk.Entry(form_frame).grid(row=0, column=1, padx=5, pady=5)
        tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=5, |
        tk.Entry(form_frame).grid(row=1, column=1, padx=5, pady=5)
        form_frame.pack(padx=10, pady=10)
        return form_frame

class MacForm(Form):
    def render(self):
        form_frame = tk.Frame(self.master, relief="flat", borderwidth=1)
        tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=8, pa
        tk.Entry(form_frame).grid(row=0, column=1, padx=8, pady=8)
        tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=8, |
        tk.Entry(form_frame).grid(row=1, column=1, padx=8, pady=8)
        form_frame.pack(padx=15, pady=15)
        return form_frame

class LinuxForm(Form):
    def render(self):
        form_frame = tk.Frame(self.master, relief="ridge", borderwidth=1)
        tk.Label(form_frame, text="Nom:").grid(row=0, column=0, sticky="w", padx=6, pa
        tk.Entry(form_frame).grid(row=0, column=1, padx=6, pady=6)
        tk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=6, |
        tk.Entry(form_frame).grid(row=1, column=1, padx=6, pady=6)
        form_frame.pack(padx=12, pady=12)
        return form_frame

# ========== APPLICATION ==========
class Application:
    def __init__(self, master):
        self.master = master
        self.factory = Application.get_factory()
        self.setup_ui()

    @staticmethod
    def get_factory():
        sys_type = platform.system()
        if sys_type == "Windows":
            return WinFactory()
        elif sys_type == "Darwin":
            return MacFactory()
        else:
            return LinuxFactory()

    def setup_ui(self):
        # Création des composants pour le système actuel
        self.menu = self.factory.create_menu(self.master)
```

```
        self.menu.render()

        self.button = self.factory.create_button(self.master)
        self.button.render()

        self.form = self.factory.create_form(self.master)
        self.form.render()

# ========== CLIENT CODE ==========
if __name__ == "__main__":
    root = tk.Tk()
    root.title("Interface adaptative")
    app = Application(root)
    root.mainloop()
```

Partie 3 : Réflexion et extension

1. Comparaison avec l'implémentation originale

Maintenabilité

- **Solution Abstract Factory** : Améliore considérablement la maintenabilité en séparant les responsabilités. Chaque composant est défini dans sa propre classe.
- **Solution originale** : Difficile à maintenir car toute la logique est concentrée dans une seule classe avec beaucoup de code dupliqué.

Extensibilité

- **Solution Abstract Factory** : Très extensible. Pour ajouter un nouveau composant, il suffit d'ajouter une nouvelle interface abstraite et ses implémentations concrètes, puis de mettre à jour l'interface GUIFactory.
- **Solution originale** : Peu extensible, nécessite de modifier de nombreuses parties du code pour ajouter un nouveau composant.

Lisibilité

- **Solution Abstract Factory** : Plus lisible car chaque composant et comportement est clairement défini dans sa propre classe avec des responsabilités précises.
- **Solution originale** : Moins lisible avec des méthodes très similaires et une structure conditionnelle complexe.

Respect des principes SOLID

- **Solution Abstract Factory** :
  - **S** (Responsabilité unique) : Chaque classe a une seule responsabilité
  - **O** (Ouvert/fermé) : Le code est ouvert à l'extension sans modification
  - **L** (Substitution de Liskov) : Les classes concrètes peuvent remplacer leurs abstractions
  - **I** (Ségrégation d'interface) : Interfaces bien séparées
  - **D** (Inversion de dépendance) : Les dépendances sont vers des abstractions
- **Solution originale** : Ne respecte quasiment aucun des principes SOLID.

2. Extension du code

Pour supporter Android

```
class AndroidFactory(GUIFactory):
    def create_button(self, master):
        return AndroidButton(master)

    def create_menu(self, master):
        return AndroidMenu(master)

    def create_form(self, master):
        return AndroidForm(master)

class AndroidButton(Button):
    def render(self):
        btn = tk.Button(self.master, text="Cliquez-moi", bg="white",
                        relief="flat", borderwidth=0, bd=0,
                        highlightthickness=0, padx=12, pady=12)
        btn.pack(pady=10)
        return btn
```

Pour ajouter une boîte de dialogue

```
# Dans GUIFactory (abstrait)
@abstractmethod
def create_dialog(self, master, title, message):
    pass

# Dans WinFactory (concret)
def create_dialog(self, master, title, message):
    return WinDialog(master, title, message)

# Interface abstraite
class Dialog(ABC):
    def __init__(self, master, title, message):
        self.master = master
        self.title = title
        self.message = message

    @abstractmethod
    def show(self):
        pass

# Implémentation concrète
class WinDialog(Dialog):
    def show(self):
        dialog = tk.Toplevel(self.master)
        dialog.title(self.title)
        dialog.geometry("300x150")
        dialog.resizable(False, False)

        tk.Label(dialog, text=self.message, padx=20, pady=20).pack()
        tk.Button(dialog, text="OK", command=dialog.destroy, width=10).pack(pady=10)

        # Rendre modal
        dialog.transient(self.master)
        dialog.grab_set()
        return dialog
```

Pour supporter des thèmes personnalisés

```
# Ajouter un "Theme Manager" qui pourrait être injecté dans les factories
class ThemeManager:
    def __init__(self, theme_name="default"):
        self.theme_name = theme_name
        self.themes = {
            "default": {
                "windows": {"button_bg": "lightblue", "button_fg": "black", "relief": '
                "mac": {"button_bg": "white", "button_fg": "black", "relief": "flat"},
                "linux": {"button_bg": "gray90", "button_fg": "black", "relief": "ridg
            },
            "dark": {
                "windows": {"button_bg": "#333333", "button_fg": "white", "relief": "r
                "mac": {"button_bg": "#222222", "button_fg": "white", "relief": "flat"
                "linux": {"button_bg": "#444444", "button_fg": "white", "relief": "rid
            }
        }

    def get_theme(self, os_type):
        os_key = "windows" if os_type == "Windows" else "mac" if os_type == "Darwin" e
        return self.themes.get(self.theme_name, {}).get(os_key, {})
```