DESIGN PATTERNS - CHAPITRE 3

Abstract Factory en Python

Créer une famille cohérente de produits sans choisir leurs classes dans le client.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Créer une famille cohérente de produits sans choisir leurs classes dans le client.
- Expliquer le coût d'ajouter une famille ou une nouvelle catégorie de produit.

Prérequis. Savoir utiliser classes, héritage, composition, listes, dictionnaires et exceptions Python. Les mécanismes spécifiques sont expliqués avant leur emploi.

Abstract Factory : intention et expérience

Abstract Factory : créer une famille cohérente

Une **fabrique abstraite** fournit plusieurs produits liés : ici bouton et case à cocher d'un même environnement. Le client utilise leurs opérations communes sans sélectionner chaque classe. `ABC` et `abstractmethod`, importés du module standard `abc`, empêchent d'instancier une classe qui ne réalise pas les opérations abstraites.

Comparez les deux familles : changer de fabrique doit changer tous les composants ensemble. Ajouter une famille est simple ; ajouter une nouvelle catégorie de produit oblige à étendre toutes les fabriques.

Python

```python
# example.py
from abc import ABC, abstractmethod

# Abstract Products
class AbstractButton(ABC):
    @abstractmethod
    def render(self):
        pass

class AbstractCheckbox(ABC):
    @abstractmethod
    def render(self):
        pass

# Concrete Products
class WindowsButton(AbstractButton):
    def render(self):
        return "Rendering a Windows button"

class WindowsCheckbox(AbstractCheckbox):
    def render(self):
        return "Rendering a Windows checkbox"

class MacOSButton(AbstractButton):
    def render(self):
        return "Rendering a MacOS button"

class MacOSCheckbox(AbstractCheckbox):
    def render(self):
        return "Rendering a MacOS checkbox"

# Abstract Factory
class AbstractFactory(ABC):
    @abstractmethod
    def create_button(self):
        pass
```

```
    @abstractmethod
    def create_checkbox(self):
        pass

# Concrete Factories
class WindowsFactory(AbstractFactory):
    def create_button(self):
        return WindowsButton()

    def create_checkbox(self):
        return WindowsCheckbox()

class MacOSFactory(AbstractFactory):
    def create_button(self):
        return MacOSButton()

    def create_checkbox(self):
        return MacOSCheckbox()

# Client code
def create_ui(factory: AbstractFactory):
    button = factory.create_button()
    checkbox = factory.create_checkbox()
    return button.render(), checkbox.render()

# Example usage:
windows_ui = create_ui(WindowsFactory())
macos_ui = create_ui(MacOSFactory())

print(windows_ui) # Output: ('Rendering a Windows button', 'Rendering a Windows
checkbox')
print(macos_ui)   # Output: ('Rendering a MacOS button', 'Rendering a MacOS
checkbox')
```

Vérifier et interpréter

Remplacez WindowsFactory par MacOSFactory dans l'appel client. Les deux composants doivent changer ensemble sans modifier create_ui. Imaginez ensuite un troisième produit, comme un menu : chaque fabrique devra fournir cette nouvelle opération. Le gain sur les familles se paie par ce coût d'extension de l'interface.

Version web : https://www.clearcode.fr/courses/programming/design-patterns/creation/abstract-factory