DESIGN PATTERNS - FICHE PRATIQUE

Cheatsheet : Construire, copier et partager des objets

Les cinq patrons de création expliqués : intentions, contrats, compromis, exemples Python et exercices corrigés.

Par Ali ZAINOUL · Cours gratuit

Cette fiche réunit les cinq patrons de création du catalogue GoF (Gang of Four). Ils répondent à des questions différentes : choisir une classe, garantir une famille cohérente, assembler, copier ou partager une identité.

Un contrat précise les entrées, le résultat et les erreurs attendus. Une classe décrit un type d'objets ; une instance est un objet construit. L'héritage spécialise une classe, la composition conserve un collaborateur auquel déléguer. Les exemples supposent les bases des classes, des fonctions, des listes et des exceptions Python.

Chaque bloc est complet et indépendant. Utilisez Python 3 avec sa bibliothèque standard, prévoyez le résultat puis comparez-le aux commentaires Expected output. Les constructions de documents et de composants graphiques sont simulées ; elles ne produisent aucun fichier ni fenêtre.

Singleton

Contrôler l'identité d'un objet en mémorisant une instance accessible par un point commun.

Quand le choisir ? Une contrainte d'unicité locale est explicite et son cycle de vie est connu ; pour une simple configuration, transmettre un objet déjà créé est souvent plus clair.

Exemple : Singleton

__new__ crée normalement l'objet. Ici il retourne l'objet mémorisé ; is compare les identités. La démonstration utilise une seule classe, sans sous-classe, dans un seul processus et sans concurrence.

Python

```python
# singleton.py

class Singleton:
    _instance = None  # Private attribute to hold the single instance

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Initialization logic can be placed here
            # For example: cls._instance.setup(config)
        return cls._instance

    # You can add methods to the Singleton class
    def some_business_logic(self):
        print("Singleton is doing something!")

# Example usage:
s1 = Singleton()
s2 = Singleton()

print(s1 is s2)  # Output: True, both variables point to the same instance
s1.some_business_logic() # Output: Singleton is doing something!

# Expected output:
# True
# Singleton is doing something!
```

> **Résultat expliqué.** True indique que s1 et s2 désignent le même objet. Le second affichage provient d'une méthode de cet objet ; aucune ressource externe n'est ouverte.

Erreur à éviter. Ce cache ne garantit ni unicité entre processus ni sécurité des créations simultanées. Un __init__ éventuel serait encore appelé lors des constructions suivantes ; l'état global complique l'isolation des tests.

▼ Si vous ajoutez un attribut par s1, s2 peut-il le lire ? Pourquoi ?

Oui : les deux noms référencent le même objet. La donnée devient donc partagée. Deux objets distincts reçus en argument permettent au contraire deux configurations indépendantes.

Abstract Factory

Construire une famille de produits compatibles en regroupant leurs opérations de création.

Quand le choisir ? Plusieurs produits doivent changer ensemble : bouton et case à cocher d'un environnement graphique, par exemple.

Exemple : Abstract Factory

Le client demande create_button() et create_checkbox(), puis utilise render(). ABC et abstractmethod, du module standard abc, rendent ces opérations abstraites : une classe incomplète ne peut pas être instanciée.

Python

```python
# abstract-factory.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')

# Expected output:
# ('Rendering a Windows button', 'Rendering a Windows checkbox')
# ('Rendering a MacOS button', 'Rendering a MacOS checkbox')
```

Résultat expliqué. Chaque tuple contient un bouton et une case du même environnement. Remplacer la fabrique change les deux produits sans modifier create_ui.

Erreur à éviter. Ajouter une famille demande une nouvelle fabrique ; ajouter une catégorie de produit demande de modifier toutes les fabriques. Ces classes produisent des messages, pas de véritables composants graphiques.

▼ Comment ajouter un menu à chacune des deux familles ?

Déclarez une opération create_menu et son contrat, puis fournissez un produit menu dans chaque famille. Ce coût transversal distingue l'ajout d'un produit de l'ajout d'une famille.

Builder

Assembler un produit par étapes nommées, puis remettre le résultat au client.

Quand le choisir ? La construction contient plusieurs étapes, des recettes réutilisables ou des contraintes d'ordre ; quelques options indépendantes justifient plutôt des arguments nommés.

Exemple : Builder

Le Builder possède le document en cours. Chaque méthode retourne self pour enchaîner les appels. build() détache le document construit puis prépare un objet neuf. Les chemins d'images sont de simples chaînes.

Python

```python
# builder.py

class Document:
    def __init__(self):
        self.title = None
        self.author = None
        self.creation_date = None
        self.sections = []
        self.images = []

    def __str__(self):
        return f"Document(title={self.title}, author={self.author},
creation_date={self.creation_date}, sections={len(self.sections)}, images=
{len(self.images)})"


class DocumentBuilder:
    def __init__(self):
        self.document = Document()

    def set_title(self, title):
        self.document.title = title
        return self

    def set_author(self, author):
        self.document.author = author
        return self

    def set_creation_date(self, date):
        self.document.creation_date = date
        return self

    def add_section(self, content):
        self.document.sections.append(content)
        return self

    def add_image(self, image_path):
```

```
        self.document.images.append(image_path)
        return self

    def build(self):
        document = self.document
        self.document = Document()
        return document

# Example Usage
builder = DocumentBuilder()
document = (builder.set_title("My Awesome Document")
                   .set_author("Jane Smith")
                   .set_creation_date("2023-11-15")
                   .add_section("Introduction: This is the introduction
section.")
                   .add_section("Main Content: This is the main content of the
document.")
                   .add_image("/path/to/image1.jpg")
                   .build())

print(document)

# Expected output:
# Document(title=My Awesome Document, author=Jane Smith, creation_date=2023-11-
15, sections=2, images=1)
```

Résultat expliqué. Le document contient le titre et l'auteur fournis, deux sections et une image référencée. Aucune image n'est lue. La remise à zéro protège ce document des constructions suivantes.

Erreur à éviter. Un Builder mutable exige une règle de remise à zéro et une validation des produits incomplets. Les étapes ne sont pas automatiquement indépendantes ou commutatives.

▼ Construisez deux documents avec le même Builder. Quel test révèle un partage accidentel des sections ?

Après le premier build, ajoutez une section au second document et vérifiez que la liste du premier reste inchangée. Les objets produits et leurs listes de sections doivent être distincts.

Factory Method

Laisser une sous-classe choisir le produit qu'une opération générale va utiliser.

Quand le choisir ? Un algorithme commun doit varier dans son étape de création tout en conservant le contrat du produit.

Exemple : Factory Method

Exporter.export() appelle create_report(), puis render() sur le produit. ABC et abstractmethod, du module abc, imposent la redéfinition de la création. PlainExporter et HtmlExporter choisissent deux produits qui retournent une chaîne.

Python

```python
# factory-method.py

from abc import ABC, abstractmethod

class Report:
    def render(self):
        return "Plain report"

class HtmlReport:
    def render(self):
        return "<p>HTML report</p>"

class Exporter(ABC):
    @abstractmethod
    def create_report(self):
        """Factory Method: subclasses choose a product with render()."""
        raise NotImplementedError

    def export(self):
        report = self.create_report()
        return report.render()

class PlainExporter(Exporter):
    def create_report(self):
        return Report()

class HtmlExporter(Exporter):
    def create_report(self):
        return HtmlReport()

print(PlainExporter().export())  # Plain report
print(HtmlExporter().export())   # <p>HTML report</p>

# Expected output:
# Plain report
# <p>HTML report</p>
```

Résultat expliqué. Le premier export donne Plain report, le second un paragraphe HTML. L'opération export reste identique ; seul create_report change dans les sous-classes.

Erreur à éviter. Une fonction avec if ou un registre de constructeurs est une fabrique simple, pas à elle seule le Factory Method GoF. L'héritage ajoute une sous-classe par variante ; une fonction reçue en argument peut parfois suffire.

▼ Ajoutez un format qui retourne un titre entouré de crochets. Faut-il modifier export() ?

Non. Ajoutez un produit fournissant render(), puis une sous-classe d'Exporter dont create_report retourne ce produit. Une modification de la logique commune n'est nécessaire que si son contrat change.

Prototype

Créer une variante à partir d'un objet existant plutôt que reconstruire toute sa configuration.

Quand le choisir ? Le modèle contient un état déjà préparé et une règle de copie claire distingue ce qui doit être partagé de ce qui doit être indépendant.

Exemple : Prototype

copy et deepcopy viennent du module standard copy. Une copie superficielle reprend les références imbriquées ; une copie profonde copie récursivement le graphe selon les protocoles des objets.

Python

```python
# prototype.py

from copy import copy, deepcopy

class ReportPrototype:
    def __init__(self, title, sections):
        self.title = title
        self.sections = sections

original = ReportPrototype("Model", ["Introduction"])
shallow = copy(original)
independent = deepcopy(original)
independent.title = "New report"
independent.sections.append("Results")

print(original.sections)     # ['Introduction']
print(independent.sections)  # ['Introduction', 'Results']
assert shallow.sections is original.sections
assert independent.sections is not original.sections

# Expected output:
# ['Introduction']
# ['Introduction', 'Results']
```

> **Résultat expliqué.** Le modèle conserve une section, le clone profond en contient deux. shallow.sections est la liste du modèle ; independent.sections est une liste distincte.

Erreur à éviter. Une copie profonde peut dupliquer trop d'état et coûter cher. Elle ne fournit pas une politique de clonage universelle pour des connexions ou fichiers ouverts. Un constructeur reste préférable si le nouvel état doit être revalidé.

▼ Que devient le modèle si vous ajoutez une section à shallow.sections ?

Sa liste reçoit aussi la section : la copie superficielle ne duplique pas la liste imbriquée. Ce partage peut être voulu ; il doit faire partie du contrat de copie.

Télécharger cette fiche en PDF · **Toutes les fiches par famille**

Version web : https://www.clearcode.fr/courses/programming/design-patterns/cheatsheet/creational