DESIGN PATTERNS - CHAPITRE 4

Builder en Python

Assembler un objet par étapes et détacher le résultat avant de réutiliser le constructeur.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Assembler un objet par étapes et détacher le résultat avant de réutiliser le constructeur.
- Vérifier que deux constructions ne partagent pas accidentellement leur état mutable.

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

Builder : intention et expérience

Builder : construire par étapes

Un **Builder** conserve un objet en cours de construction et expose des étapes nommées. Ici les ingrédients sont ajoutés avant de récupérer la pizza. Pour quelques options indépendantes, des arguments nommés dans un constructeur suffisent ; une classe Builder se justifie surtout par des étapes ou des recettes réutilisables.

Python

```python
# example.py
class Pizza:
    def __init__(self):
        self.dough = None
        self.sauce = None
        self.topping = None

    def __str__(self):
        return f"Pizza with dough: {self.dough}, sauce: {self.sauce}, topping: {self.topping}"

class PizzaBuilder:
    def __init__(self):
        self.pizza = Pizza()

    def add_dough(self, dough):
        self.pizza.dough = dough
        return self  # allows method chaining

    def add_sauce(self, sauce):
        self.pizza.sauce = sauce
        return self

    def add_topping(self, topping):
        self.pizza.topping = topping
        return self

    def build(self):
        return self.pizza

# Example usage:
builder = PizzaBuilder()
pizza = builder.add_dough("thin crust").add_sauce("tomato").add_topping("pepperoni").build()
```

```
print(pizza) # Output: Pizza with dough: thin crust, sauce: tomato, topping:
pepperoni
```

Builder : détacher le résultat construit

Les méthodes retournent `self` pour enchaîner les étapes. `build()` détache le document puis prépare un objet neuf : réutiliser le Builder ne doit pas modifier le résultat déjà obtenu. Les chemins d'images sont de simples chaînes ; aucun fichier n'est lu.

Python

```python
# example.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)
```

Suite du Builder de document : un directeur regroupe une recette et réutilise les étapes publiques. Construisez deux documents puis vérifiez que leurs sections restent indépendantes. Le directeur organise les étapes, le Builder possède l'objet en cours.

Python

```python
# example.py
class DocumentDirector:
    def __init__(self, builder):
        self.builder = builder

    def construct_basic_document(self, title, author, date):
        self.builder.set_title(title)
        self.builder.set_author(author)
        self.builder.set_creation_date(date)

    def construct_full_document(self, title, author, date, sections, images):
        self.construct_basic_document(title, author, date)
        for section in sections:
            self.builder.add_section(section)
        for image in images:
            self.builder.add_image(image)

# Example Usage with Director
builder = DocumentBuilder()
director = DocumentDirector(builder)

director.construct_basic_document("Simple Document", "Alice", "2023-11-16")
simple_document = builder.build()
print(simple_document)

sections = ["Section 1", "Section 2"]
images = ["/path/to/image1.png", "/path/to/image2.png"]
director.construct_full_document("Complete Document", "Bob", "2023-11-16",
sections, images)
complete_document = builder.build()
print(complete_document)
```

Vérifier et interpréter

Construisez un premier document, appelez build(), puis ajoutez une section au Builder pour un second document. Le premier doit conserver ses sections. Un Builder mutable impose de préciser quand le résultat est détaché et si l'ordre des étapes est contraint ; des arguments nommés restent plus directs pour un objet simple.

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