DESIGN PATTERNS - FICHE PRATIQUE

Cheatsheet : Adapter et composer des objets

Les sept patrons structurels expliqués : interfaces, délégation, partage, exemples Python et exercices corrigés.

Par Ali ZAINOUL · Cours gratuit

Les sept patrons structurels du catalogue GoF (Gang of Four) organisent les relations entre objets. Avant de choisir, demandez si vous devez traduire une interface, ajouter une responsabilité, contrôler l'accès, grouper ou partager des données.

Une interface est l'ensemble des opérations attendues par le client. La composition conserve un objet collaborateur et lui délègue une partie du travail ; l'héritage reprend ou redéfinit des opérations d'une classe de base. Les exemples supposent les classes, les listes, les dictionnaires et les méthodes Python.

Chaque bloc s'exécute seul avec Python 3 et sa bibliothèque standard. Les sorties attendues figurent en commentaires. Les rendus, services et opérations matérielles sont des simulations ; aucun accès réseau ou matériel n'est effectué.

Adapter

Traduire une interface disponible vers celle qu'attend le client.

Quand le choisir ? Une bibliothèque ou une classe existante réalise le bon travail mais expose d'autres noms, paramètres ou représentations de données.

Exemple : Adapter

Le client appelle request(). L'adaptateur reçoit un Adaptee, appelle specific_request() et adapte la chaîne renvoyée. Il réutilise cet objet par composition : une référence est contenue dans l'adaptateur.

Python

```python
# adapter.py

# Target interface
class Target:
    def request(self):
        return "Target: The default target's behavior."

# Adaptee class
class Adaptee:
    def specific_request(self):
        return ".eetpadA eht fo roivaheb laicepseht"

# Adapter class
class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return f"Adapter: (TRANSLATED) {self.adaptee.specific_request()[::-1]}"

# Client code
def client_code(target):
    print(target.request())

if __name__ == "__main__":
    adaptee = Adaptee()
    adapter = Adapter(adaptee)
    client_code(adapter)

# Expected output:
# Adapter: (TRANSLATED) thespecial behavior of the Adaptee.
```

Résultat expliqué. L'unique appel du client passe par Adapter, qui inverse le texte produit par Adaptee. Le préfixe Adapter: (TRANSLATED) rend cette délégation visible ; le client conserve le même appel request().

Erreur à éviter. Rendre les signatures compatibles ne garantit pas une équivalence métier. Toute conversion doit conserver le sens des données ou signaler ce qu'elle ne peut pas représenter.

▼ Si l'ancien service attend des secondes et le client des millisecondes, quelle responsabilité ajouter ?

Définir explicitement la conversion d'unité et ses règles d'arrondi, puis tester les bornes. Renommer la méthode sans convertir produirait une interface compatible en apparence mais un résultat faux.

Bridge

Séparer deux dimensions de variation en reliant une abstraction à un objet d'implémentation.

Quand le choisir ? Deux axes doivent évoluer indépendamment : formes et couleurs, vues et mécanismes de rendu, par exemple.

Exemple : Bridge

Circle conserve un objet Color et lui demande fill(). Le rayon appartient à la forme ; la couleur est un collaborateur reçu au constructeur. Les méthodes de base sont abstraites par convention, sans ABC.

Python

```python
# bridge.py

# Abstraction
class Shape:
    def __init__(self, color):
        self.color = color

    def draw(self):
        pass

# Refined Abstraction
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def draw(self):
        return f"Circle with radius {self.radius} drawn in {self.color.fill()}"

# Implementor
class Color:
    def fill(self):
        pass

# Concrete Implementor
class RedColor(Color):
    def fill(self):
        return "Red"

class BlueColor(Color):
    def fill(self):
        return "Blue"

# Client code
if __name__ == "__main__":
    red = RedColor()
    blue = BlueColor()
```

```
    circle1 = Circle(red, 5)
    circle2 = Circle(blue, 10)

    print(circle1.draw())
    print(circle2.draw())

# Expected output:
# Circle with radius 5 drawn in Red
# Circle with radius 10 drawn in Blue
```

Résultat expliqué. Les deux messages combinent leur rayon avec Red ou Blue. Modifier l'objet couleur ne change pas le code de Circle ni sa géométrie.

Erreur à éviter. Une délégation supplémentaire n'aide pas si le second axe ne varie jamais. Bridge organise des variations conçues ensemble ; Adapter résout surtout une incompatibilité d'interfaces existantes.

▼ Combien de classes concrètes faut-il pour trois formes et quatre couleurs, sans créer une classe par paire ?

Trois classes de formes et quatre classes de couleurs peuvent produire douze combinaisons par composition. Ce nombre suppose que leurs interfaces permettent bien ces combinaisons.

Composite

Appliquer une même opération à des objets élémentaires et à des groupes organisés en arbre.

Quand le choisir ? Un client doit traiter uniformément une feuille et une composition récursive de feuilles ou de groupes.

Exemple : Composite

Chaque dessin expose render(). CompoundGraphic conserve des enfants, appelle leur render() et assemble les résultats. Un groupe peut contenir un autre groupe : la délégation suit alors l'arbre récursivement.

Python

```python
# composite.py

# Component
class Graphic:
    def render(self):
        pass

# Leaf
class CircleGraphic(Graphic):
    def __init__(self, radius):
        self.radius = radius

    def render(self):
        return f"Rendering circle with radius {self.radius}"

# Leaf
class SquareGraphic(Graphic):
    def __init__(self, side):
        self.side = side

    def render(self):
        return f"Rendering square with side {self.side}"

# Composite
class CompoundGraphic(Graphic):
    def __init__(self):
        self.children = []

    def add(self, graphic):
        self.children.append(graphic)

    def remove(self, graphic):
        self.children.remove(graphic)

    def render(self):
        result = ""
        for child in self.children:
```

```
            result += child.render() + "\n"
        return result

# Client code
if __name__ == "__main__":
    compound_graphic = CompoundGraphic()
    compound_graphic.add(CircleGraphic(5))
    compound_graphic.add(SquareGraphic(10))

    compound_graphic2 = CompoundGraphic()
    compound_graphic2.add(CircleGraphic(2))
    compound_graphic.add(compound_graphic2)

    print(compound_graphic.render())

# Expected output:
# Rendering circle with radius 5
# Rendering square with side 10
# Rendering circle with radius 2
```

Résultat expliqué. La sortie mentionne le cercle de rayon 5, le carré de côté 10 puis le cercle de rayon 2 dans le sous-groupe. Les sauts de ligne proviennent de l'assemblage à chaque niveau.

Erreur à éviter. Définissez les opérations réservées aux branches et interdisez les cycles. Un enfant partagé par plusieurs parents demande une règle de propriété ; cette démonstration n'en impose pas.

▼ Que se passe-t-il si vous ajoutez un groupe comme enfant de lui-même ?

render() s'appellera récursivement sans atteindre de feuille et finira par échouer. Une structure destinée à être un arbre doit refuser cette relation ou détecter les cycles.

Decorator

Ajouter des responsabilités par des enveloppes qui conservent l'interface de l'objet.

Quand le choisir ? Des options doivent se combiner indépendamment sans une sous-classe pour chaque combinaison.

Exemple : Decorator

Coffee fournit get_cost() et get_description(). Chaque enveloppe garde le café reçu, délègue l'opération puis ajoute sa contribution. Le patron objet ne nécessite pas la syntaxe Python @decorator.

Python

```python
# decorator.py

# Component
class Coffee:
    def get_cost(self):
        return 5

    def get_description(self):
        return "Simple coffee"

# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self.coffee = coffee

    def get_cost(self):
        return self.coffee.get_cost()

    def get_description(self):
        return self.coffee.get_description()

# Concrete Decorator
class MilkDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 2

    def get_description(self):
        return super().get_description() + ", with milk"

# Concrete Decorator
class SugarDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 1

    def get_description(self):
        return super().get_description() + ", with sugar"
```

```
# Client code
if __name__ == "__main__":
    coffee = Coffee()
    print(f"{coffee.get_description()} Cost: ${coffee.get_cost()}")

    milk_coffee = MilkDecorator(coffee)
    print(f"{milk_coffee.get_description()} Cost: ${milk_coffee.get_cost()}")

    sugar_milk_coffee = SugarDecorator(milk_coffee)
    print(f"{sugar_milk_coffee.get_description()} Cost:
${sugar_milk_coffee.get_cost()}")

# Expected output:
# Simple coffee Cost: $5
# Simple coffee, with milk Cost: $7
# Simple coffee, with milk, with sugar Cost: $8
```

> **Résultat expliqué.** Le café coûte 5, le lait ajoute 2, le sucre ajoute 1 : le total composé vaut 8. Les descriptions suivent l'ordre d'enveloppement.

> **Erreur à éviter.** Chaque enveloppe doit préserver toutes les opérations attendues. L'ordre des couches peut changer le résultat et rendre le chemin d'exécution moins lisible ; aucune couche ne doit oublier de déléguer.

▼ Inverser lait et sucre change-t-il le coût ? Et la description ?

Dans cet exemple, le coût reste 8 car les suppléments s'additionnent. L'ordre des mots change dans la description : une composition peut être commutative pour une opération et pas pour une autre.

Facade

Présenter une opération simple qui coordonne plusieurs composants d'un sous-système.

Quand le choisir ? Le scénario courant demande un enchaînement de détails que le client n'a pas besoin de piloter lui-même.

Exemple : Facade

ComputerFacade.start() coordonne CPU, Memory et HardDrive. Ces classes ne touchent pas au matériel : elles retournent ou affichent des messages. Les retours intermédiaires ne sont pas tous affichés.

Python

```python
# facade.py

# Subsystem classes
class CPU:
    def freeze(self):
        return "Freezing processor"

    def jump(self, position):
        return f"Jumping to position: {position}"

    def execute(self):
        return "Executing"

class Memory:
    def load(self, position, data):
        return f"Loading from position {position}, data: {data}"

class HardDrive:
    def read(self, lba, size):
        return f"Reading sector {lba}, size: {size}"

# Facade
class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hard_drive = HardDrive()

    def start(self):
        print("Starting computer")
        self.cpu.freeze()
        self.memory.load("0x00", self.hard_drive.read("100", "1024"))
        self.cpu.jump("0x00")
        self.cpu.execute()
        print("Computer started")

# Client code
```

```python
if __name__ == "__main__":
    computer = ComputerFacade()
    computer.start()

# Expected output:
# Starting computer
# Computer started
```

Résultat expliqué. Seuls Starting computer et Computer started sont imprimés. Pour observer l'ordre interne, remplacez les composants par des faux enregistrant les appels plutôt que d'inventer des sorties supplémentaires.

Erreur à éviter. La façade ne garantit ni sécurité ni atomicité. Elle ne doit pas devenir une classe concentrant toutes les responsabilités et n'empêche pas un accès détaillé au sous-système.

▼ Comment vérifier que la lecture du disque précède le chargement en mémoire ?

Fournissez des faux composants qui ajoutent leur nom à une même liste lors des appels. Vérifiez que read est enregistré avant load. Le test observe le scénario coordonné par la façade.

Flyweight

Partager des objets représentant un état intrinsèque identique, en passant séparément l'état propre à chaque usage.

Quand le choisir ? De nombreux éléments réutilisent les mêmes données stables et cette redondance représente un coût mesuré.

Exemple : Flyweight

Le triplet caractère, taille et police est la clé du cache. La position est fournie à render() et n'appartient pas à l'objet partagé. Le cache crée au plus un Character par clé pendant son existence.

Python

```python
# flyweight.py

# Flyweight
class Character:
    def __init__(self, char, font_size, font_family):
        self.char = char
        self.font_size = font_size
        self.font_family = font_family

    def render(self, position_x, position_y):
        return f"Character {self.char} rendered at ({position_x},{position_y})
with font {self.font_family} size {self.font_size}"

# Flyweight Factory
class CharacterFactory:
    _characters = {}

    @staticmethod
    def get_character(char, font_size, font_family):
        key = (char, font_size, font_family)
        if key not in CharacterFactory._characters:
            CharacterFactory._characters[key] = Character(char, font_size,
font_family)
        return CharacterFactory._characters[key]

# Client code
if __name__ == "__main__":
    text = "Hello World"
    x = 0
    for char in text:
        character = CharacterFactory.get_character(char, 12, "Arial")
        print(character.render(x, 0))
        x += 1

# Expected output:
# Character H rendered at (0,0) with font Arial size 12
# Character e rendered at (1,0) with font Arial size 12
```

```
# Character l rendered at (2,0) with font Arial size 12
# Character l rendered at (3,0) with font Arial size 12
# Character o rendered at (4,0) with font Arial size 12
# Character   rendered at (5,0) with font Arial size 12
# Character W rendered at (6,0) with font Arial size 12
# Character o rendered at (7,0) with font Arial size 12
# Character r rendered at (8,0) with font Arial size 12
# Character l rendered at (9,0) with font Arial size 12
# Character d rendered at (10,0) with font Arial size 12
```

Résultat expliqué. Hello World est parcouru caractère par caractère. Les lettres répétées retrouvent le même objet pour la même police et la même taille ; la position x change à chaque appel.

Erreur à éviter. Ne modifiez pas les attributs intrinsèques partagés. Le cache conserve ses entrées sans limite dans cet exemple ; le partage n'est utile que si son coût est inférieur à la duplication évitée.

▼ Deux lettres l avec deux tailles de police différentes doivent-elles partager le même Character ?

Non : la taille fait partie de l'état intrinsèque et de la clé. Deux usages ne partagent l'objet que lorsque tout le triplet est identique.

Proxy

Contrôler l'accès à un service tout en conservant son interface.

Quand le choisir ? L'accès doit être différé, contrôlé ou représenté à distance sans changer l'appel effectué par le client.

Exemple : Proxy

Proxy.operation() vérifie check_access(), délègue à RealService puis journalise par affichage. Le contrôle retourne toujours True ici : ce n'est pas une authentification réelle.

Python

```python
# proxy.py

# Subject interface
class Service:
    def operation(self):
        pass

# Real Subject
class RealService(Service):
    def operation(self):
        return "RealService: Handling request."

# Proxy
class Proxy(Service):
    def __init__(self, real_service):
        self.real_service = real_service

    def operation(self):
        # Add some logic here, like access control, logging, etc.
        print("Proxy: Checking access prior to firing a real request.")
        if self.check_access():
            result = self.real_service.operation()
            self.log_access()
            return result
        else:
            return "Proxy: Access denied."

    def check_access(self):
        # Simulate access check
        return True

    def log_access(self):
        print("Proxy: Logging the time of request.")

# Client code
if __name__ == "__main__":
    real_service = RealService()
```

```
    proxy = Proxy(real_service)
    print(proxy.operation())

# Expected output:
# Proxy: Checking access prior to firing a real request.
# Proxy: Logging the time of request.
# RealService: Handling request.
```

Résultat expliqué. Le contrôle puis le message de journal sont imprimés avant le résultat du vrai service. La délégation réalise une seule opération ; aucun journal externe n'est écrit.

Erreur à éviter. La politique doit préciser les résultats et erreurs du refus, les coûts de délégation et le cycle de vie du service. Un proxy ne rend pas automatiquement une opération distante identique à un appel local.

▼ Si check_access() retourne False, le vrai service doit-il être appelé ?

Non. operation() retourne le message de refus sans délégation ni journal de succès. Un faux service qui compte ses appels permet de vérifier ce cas.

Télécharger cette fiche en PDF · Toutes les fiches par famille

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