Programmation Orientée Objet en Python

#6 POO Avancée

par David Albert

2023

Table des matières

01

SOLID principles *

*cette section reprend les exemples de @dmmeteo

Single responsability

Explication

Chaque classe / composant logiciel doit avoir une responsable unique.

✓ GOOD

```python
class Animal:
    def __init__(self, name: str):
            self.name = name

    def get_name(self):
        pass


class AnimalDB:
    def get_animal(self) -> Animal:
        pass

    def save(self, animal: Animal):
        pass
```

✕ BAD

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

    def save(self, animal: Animal):
        pass
```

↑ Ici, la classe `Animal` a deux responsabilités: gérer la base de données et gérer les attributs.

Open-Closed

Explication

Chaque entité logiciel (classe, module, fonction) doit être ouverte à l'extension mais fermée à la modification.

✓ GOOD

```python
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def get_discount(self):
            return self.price * 0.2


class VIPDiscount(Discount):
    def get_discount(self):
        return super().get_discount() * 2
```

✕ BAD

```python
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def give_discount(self):
            if self.customer == 'fav':
                return self.price * 0.2
            if self.customer == 'vip':
                return self.price * 0.4
```

↑ Ici, au moindre souhait d'ajouter un nouveau type de remise, on devra changer le code existant !

Substitution de Liskov

Explication

Une sous-classe doit être substituable à sa super-classe. Le but est de s'assurer qu'une sous-classe peut prendre la place de sa super-classe sans erreur.

✓ GOOD

```
def animal_leg_count(animals: list):
    for animal in animals:
        print(animal.leg_count())

animal_leg_count(animals)
```

✕ BAD

```
def animal_leg_count(animals: list):
    for animal in animals:
        if isinstance(animal, Lion):
            print(lion_leg_count(animal))
        elif isinstance(animal, Mouse):
            print(mouse_leg_count(animal))
        elif isinstance(animal, Pigeon):
            print(pigeon_leg_count(animal))

animal_leg_count(animals)
```

↑ Ici, le type de classe est vérifié, le principe de substitution de Liskov est donc violé !

Séparation des Interfaces

Explication

Fournir des interfaces simples et spécifiques. Ne pas contraindre la personne qui voudra étendre notre logiciel de dépendre d'interfaces qu'il n'utilise pas.

✓ GOOD

```python
class IShape:
    def draw(self):
        raise NotImplementedError

class Circle(IShape):
    def draw(self):
        pass

class Square(IShape):
    def draw(self):
        pass

class Rectangle(IShape):
    def draw(self):
        pass
```

✕ BAD

```python
class IShape:
    def draw_square(self):
        raise NotImplementedError

    def draw_rectangle(self):
        raise NotImplementedError

    def draw_circle(self):
        raise NotImplementedError
```

← Ici, on pourra utiliser une combinaison de plusieurs interfaces pour créer des formes particulières: demi cercle, triancle rectangle, ...

Inversion des Dépendances

Explication

La dépendance doit porter sur les abstractions et non sur les concrétions.

✓ GOOD

```python
class Connection:
    def request(self, url: str, options: dict):
        raise NotImplementedError

class Http:
    def __init__(self, http_connection: Connection):
        self.http_connection = http_connection

    def get(self, url: str, options: dict):
        self.http_connection.request(url, 'GET')

    def post(self, url, options: dict):
        self.http_connection.request(url, 'POST')

class XMLHttpService(Connection):
    xhr = XMLHttpRequest()
    def request(self, url: str, options:dict):
        self.xhr.open()
        self.xhr.send()
```

✕ BAD

```python
class XMLHttpService(XMLHttpRequestService):
    pass

class Http:
    def __init__(self, xml_http_service: XMLHttpService):
        self.xml_http_service = xml_http_service

    def get(self, url: str, options: dict):
        self.xml_http_service.request(url, 'GET')

    def post(self, url, options: dict):
        self.xml_http_service.request(url, 'POST')
```

↑ Ici, la classe de haut-niveau (http) dépend d'une implémentation spécifique pour le bas niveau et non d'une abstraction.

02

Design patterns

Introduction

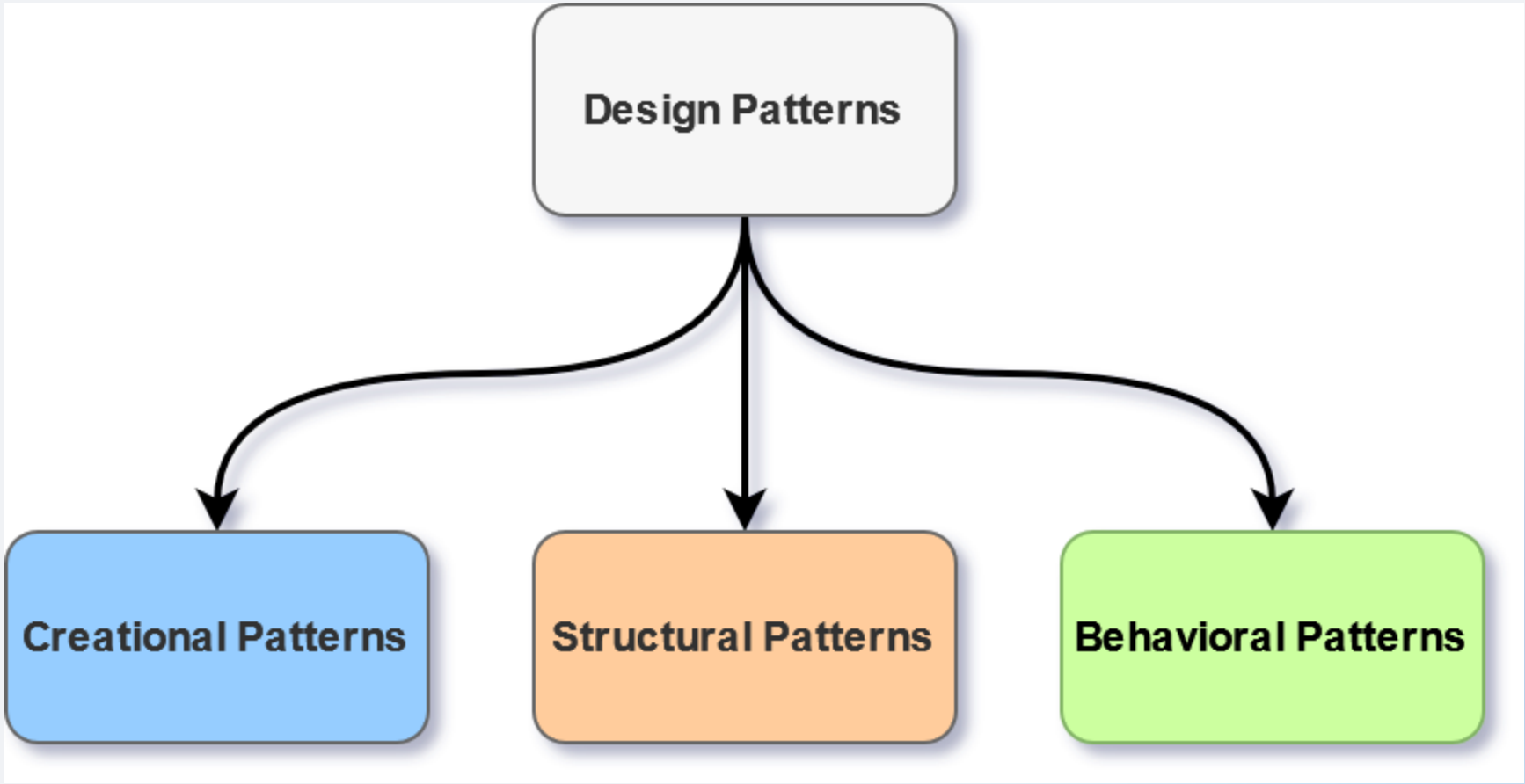
Les **patrons de conception** (design patterns) sont des solutions classiques à des problèmes récurrents de la conception de logiciels.

Chaque patron est une sorte de plan ou de schéma que vous pouvez personnaliser afin de résoudre un problème récurrent dans votre code.

Dans ce cours, nous présenterons 4 patrons de conception classiques.

Refactoring Guru contient les explications de 22 patrons de conceptions classiques et leur utilisation.

Types de patterns



Factory Method creational

La **"factory method"** permet de construire des objets depuis du texte.

Exemple

```python
class FrenchLocalizer:
    """it simply returns the french version"""
    def __init__(self):
        self.translations = {"car": "voiture", "bike": "bicyclette"}

    def localize(self, msg):
        """change the message using translations"""
        return self.translations.get(msg, msg)

class SpanishLocalizer:
    """it simply returns the spanish version"""
    def __init__(self):
        self.translations = { "car": "coche", "bike": "bicicleta"}

    def localize(self, msg):
        """change the message using translations"""
        return self.translations.get(msg, msg)

class EnglishLocalizer:
    """Simply return the same message"""
    def localize(self, msg):
        return msg
```

```python
# The factory
def TranslateFactory(language ="English"):

    """Factory Method"""
    localizers = {
        "French": FrenchLocalizer,
        "English": EnglishLocalizer,
        "Spanish": SpanishLocalizer,
    }

    return localizers[language]()
```

```python
f = Factory("French")
print(f.localize("car")) # voiture
```

Adapter structural

La méthode **"adapter"** permet de réutiliser le comportement fonctionnel d'une classe mais dont l'interface ne correspond pas aux attentes.

Exemple

```python
class Target:
    """
    Interface utilisé par le code client.
    """
    def request(self) -> str:
        return "Target: The default target's behavior."

class Adaptee:
    """
    Contient des comportements utiles mais  l'interface
    ne correspond pas à l'interface existante.
    """
    def specific_request(self) -> int:
        return 12301384
```

```python
class Adapter(Target):
    """
    L'adapteur permet de faire concorder la classe
    véhicule avec l'interface souhaitée.
    """
    def __init__(self, adaptee: Adaptee):
        self.__adaptee = adaptee

    def request(self) -> str:
        return "Adapted request : " +
            str(self.__adaptee.specific_request())
```

Bridge structural

Le méthode **"bridge"** permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies — abstraction et implémentation — qui peuvent évoluer indépendamment l'une de l'autre.

Exemple :

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    def __init__(self, x: int, y: int, drawAPI: DrawAPI) -> None:
        self._x = x
        self._y = y
        self._drawAPI = drawAPI

    @abstractmethod
    def draw(self) -> None:
        pass

class Circle(Shape):
    def draw(self) -> None:
        self._drawAPI.draw_circle(color)
```

```python
class DrawAPI(ABC):
    """ Définit l'interface de notre système de dessin. """
    @abstractmethod
    def draw_circle(self, x: int, y: int) -> None:
        pass

    @abstractmethod
    def draw_rectangle(self, x: int, y: int) -> None:
        pass

class ShellDrawAPI(DrawAPI):
    # ... impélemente DrawAPI pour un affichage en terminal ...

class TkinterDrawAPI(DrawAPI):
    # ... impélemente DrawAPI pour un affichage en via Tkinter ...
```

```
api_type : DrawAPI = new ShellDrawAPI()
circle1 : Shape = new Circle(0, 0, api_type)
circle2 : Shape = new Circle(20, 10, api_type)
circle1.draw()
circle2.draw()
```

Strategy behavioral

La méthode **"strategy"** permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

Exemple :

```python
import random
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    """
    L'interface Strategy déclare les opération communes
    à toutes les versions supportées d'un algorithme.
    """
    @abstractmethod
    def sort(self, data: List):
        pass

class StandardSort(SortStrategy):
    def sort(self, data: List) -> List:
        return sorted(data)

class ReverseSort(SortStrategy):
    def sort(self, data: List) -> List:
        return reversed(sorted(data))
```

```python
class Context():
    def __init__(self, sort_strategy: SortStrategy) -> None:
        """
        Souvent, le contexte accepte la stratégie comme
        entrée du constructeur, mais fournit aussi un
        setter pour la modifier pendant l'exécution.
        """
        self._sort_strategy = sort_strategy

    def get_sort_strategy(self) -> SortStrategy:
        return self._sort_strategy

    def set_sort_strategy(self, sort_strategy: SortStrategy) -> None:
        self._sort_strategy = sort_strategy

    def do_some_business_logic(self) -> None:
        print("Context: Sorting data using the strategy (not sure how \it'll do it)")
        result = self._sort_strategy.sort(["a", "b", "c", "d", "e"])
        print(",".join(result))
```

```python
context = Context(new StandardSort())

context.do_some_business_logic() # a,b,c,d,e
context.set_sort_strategy(new ReverseSort())
context.do_some_business_logic() # e,d,c,b,a
```

03

Annexes

Annexes 1 : Patrons de conception

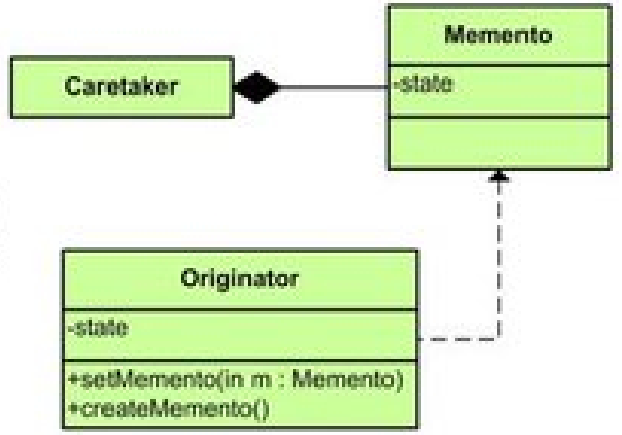
Patterns de comportement

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

Memento

Type: Behavioral

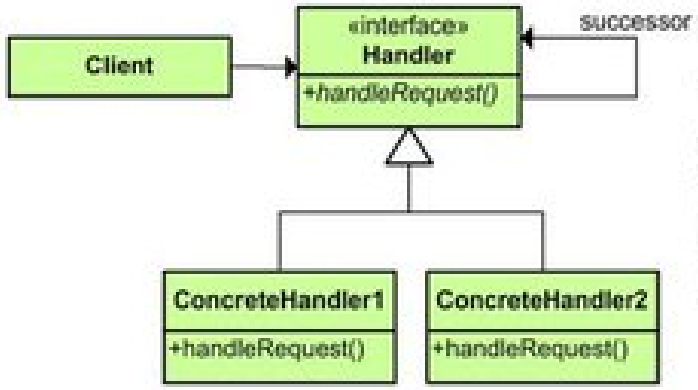
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

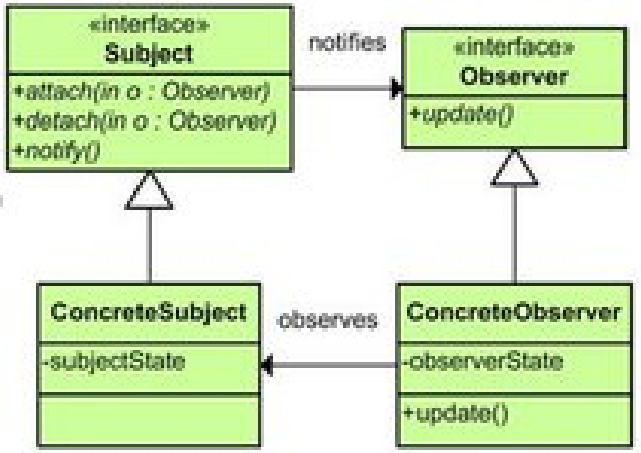
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



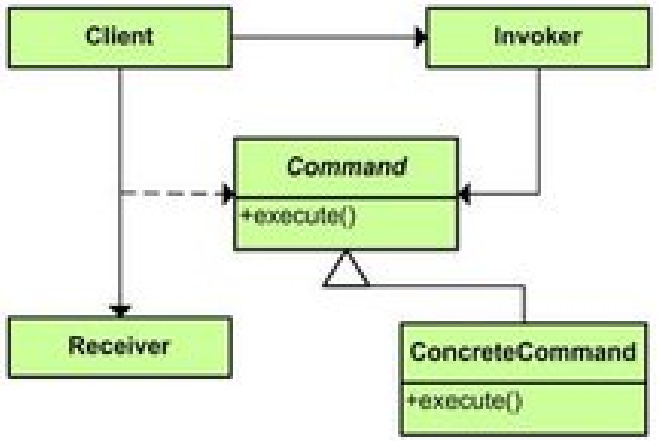
Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Patterns de comportement



Command

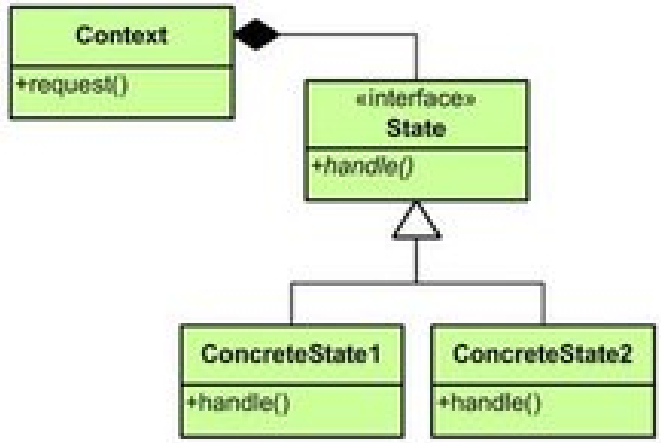
Type: Behavioral

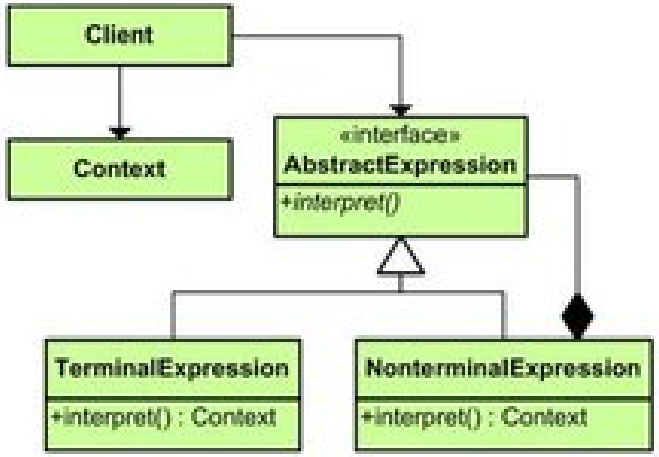
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.





Interpreter

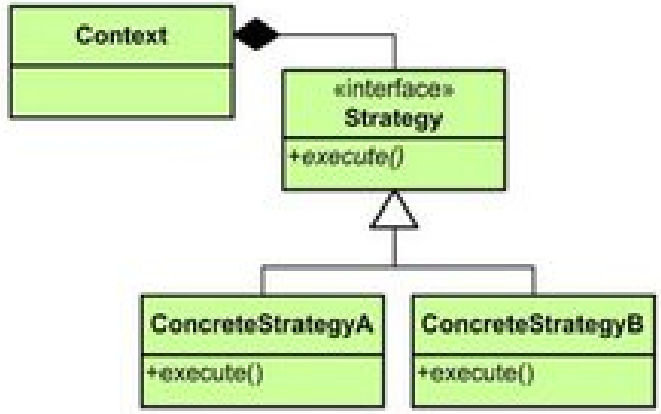
Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

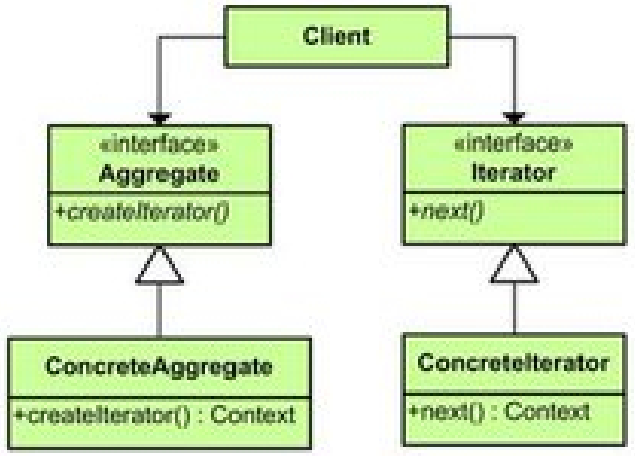
Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Patterns de comportement



Iterator

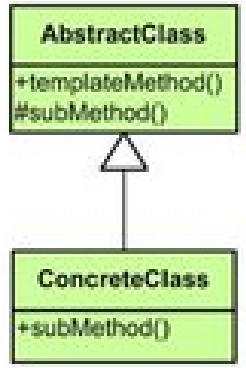
Type: Behavioral

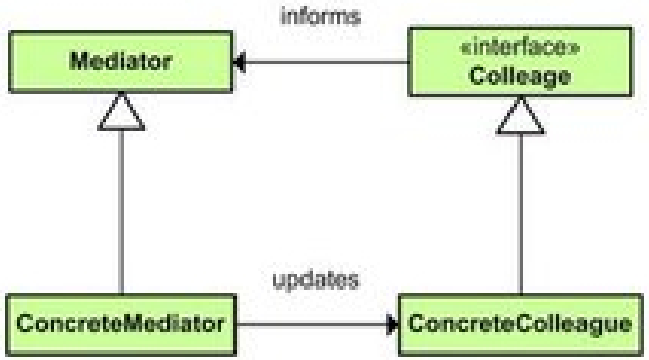
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.





Mediator

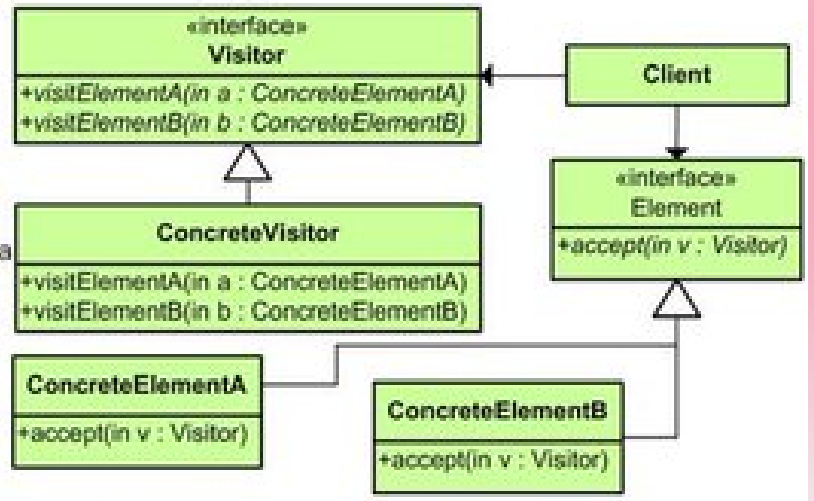
Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Visitor

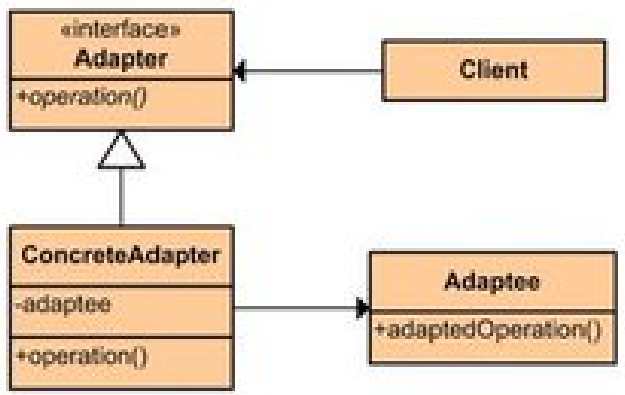
Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

Patterns structurels



Adapter

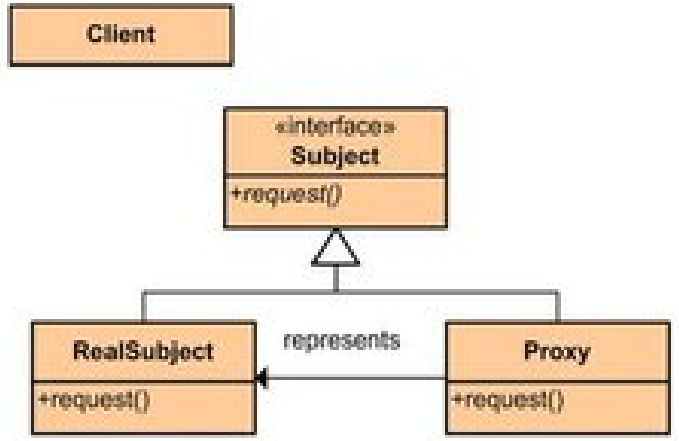
Type: Structural

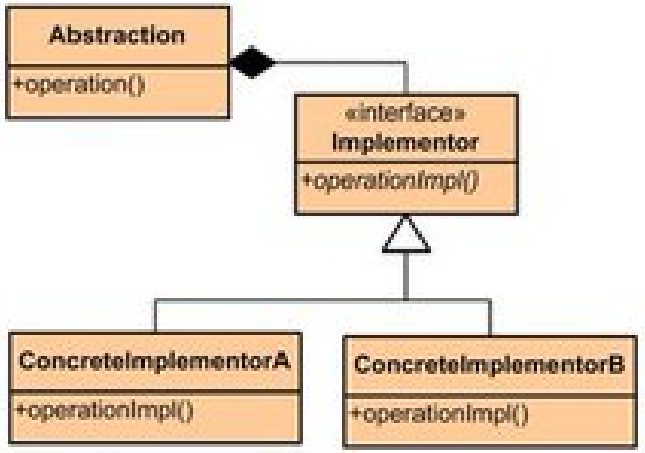
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

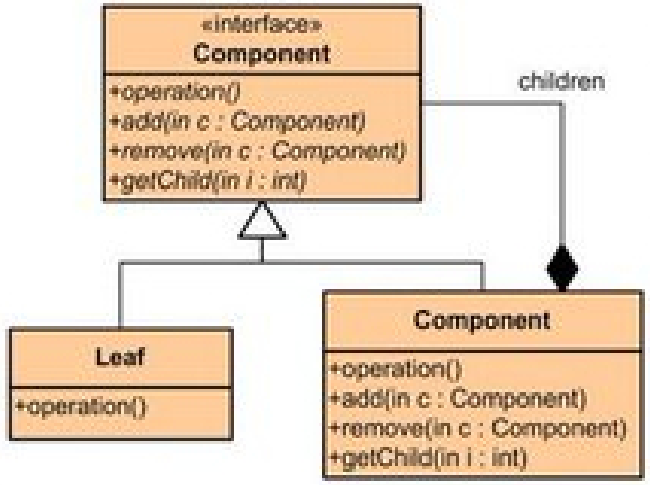




Bridge

Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

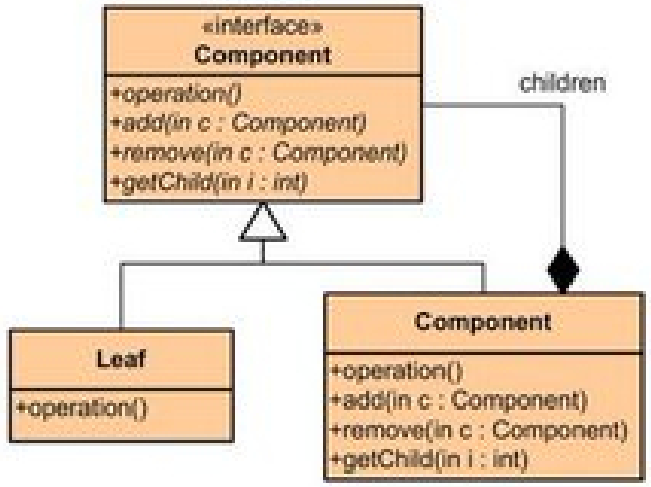


Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

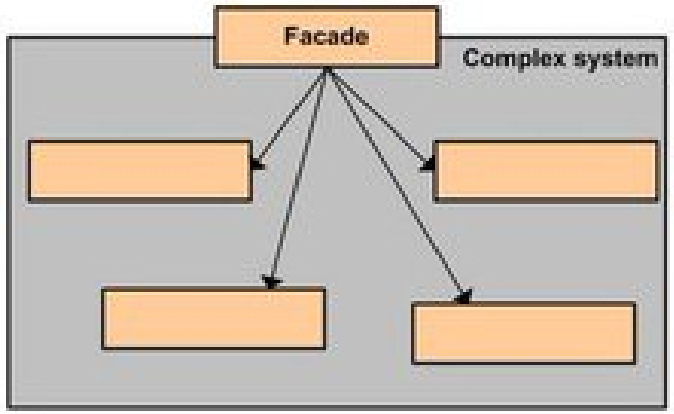
Patterns structurels



Composite

Type: Structural

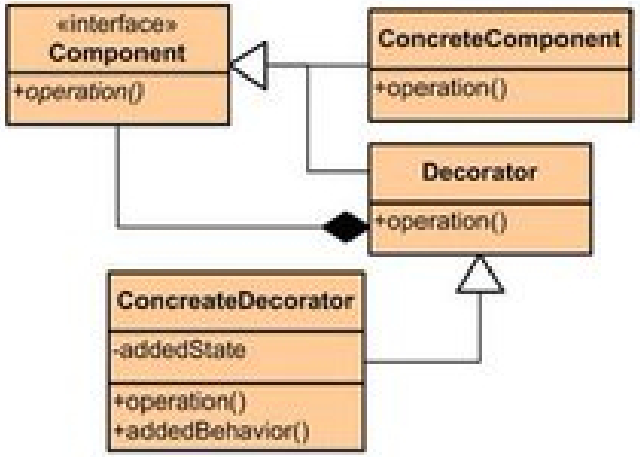
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Facade

Type: Structural

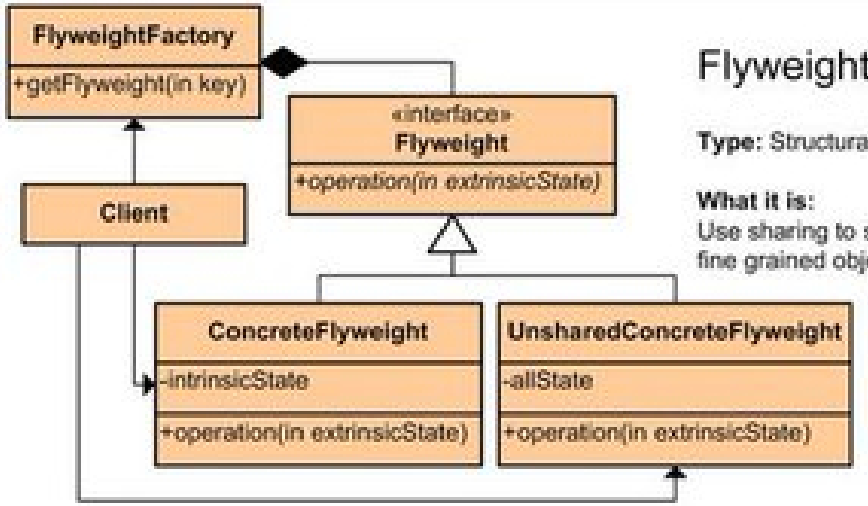
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Decorator

Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Flyweight

Type: Structural

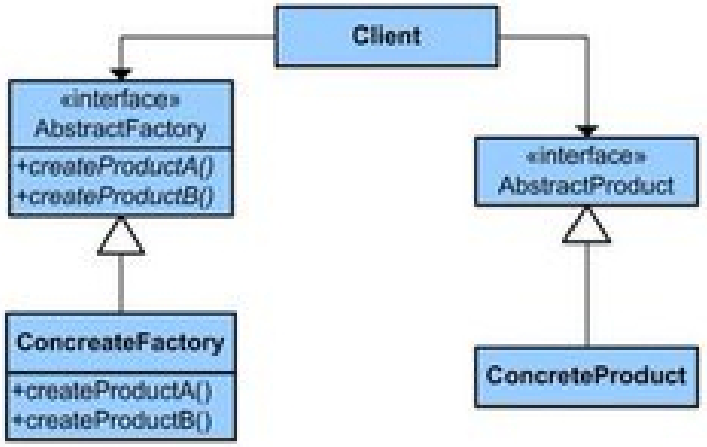
What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Patterns de création

Abstract Factory

Type: Creational

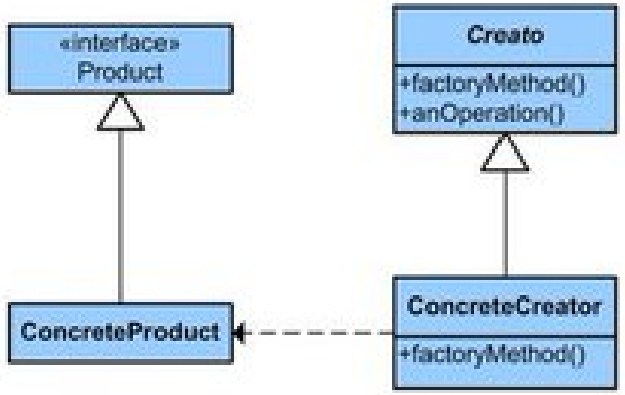
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Factory Method

Type: Creational

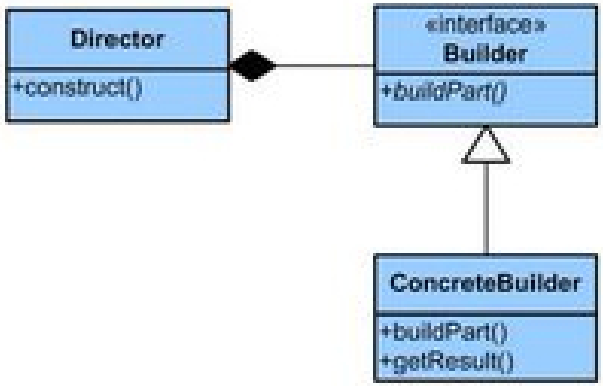
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Builder

Type: Creational

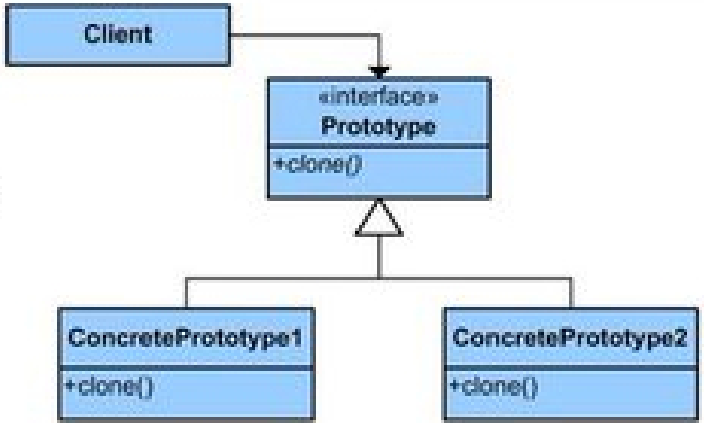
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.