TMPS – Identificati si descrieti sablonul de proiectare + exemplu de cod

CREATIONAL PATTERNS

Factory Method

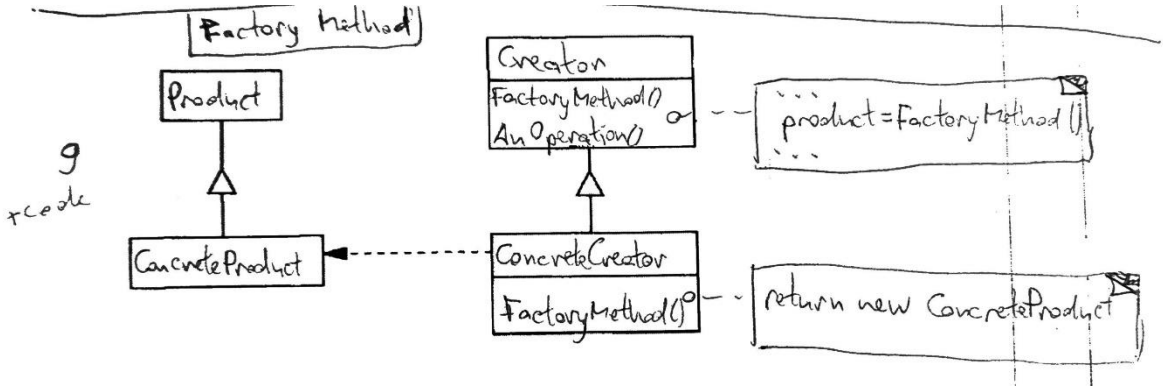
- Intentia sablonului

Factory Method este un sablon de proiectare creational care ofera o interfata pentru crearea obiectelor intr-o clasa, dar permite subclaselor sa schimbe tipul de obiecte care sunt create.

- Problema care o rezolva

Uneori este necesar sa se creeze obiecte in timpul rularii programului, dar nu se poate decide ce tip de obiect trebuie creat pana la momentul executarii. De exemplu, poate fi necesar sa se decida ce tip de conexiune de baza de date trebuie folosita in functie de configuratia sistemului. Factory Method rezolva aceasta problema prin permiterea subclaselor sa decida tipul obiectelor care trebuie create.

- Componente cu o mica descriere
  1. Creator: Clasa abstracta care defineste interfata pentru crearea obiectelor.
  2. ConcreteCreator: Subclasa care implementeaza Creator si returneaza obiecte de un anumit tip.
  3. Product: Clasa abstracta a obiectului care trebuie creat.
  4. ConcreteProduct: Clasa care implementeaza Product si reprezinta obiectul concret care trebuie creat.



Factory Method (exemplu de cod)

```
from abc import ABCMeta, abstractmethod


class AbstractDegree(metaclass=ABCMeta):
    @abstractmethod
    def info(self): pass


class BachelorEngineer(AbstractDegree):
    def info(self):
        print('Bachelor of engineering')

    def __str__(self):
        return 'Bachelor of engineering'


class MasterEngineer(AbstractDegree):
    def info(self):
        print('Master of engineering')

    def __str__(self):
        return 'Master of engineering'

class
MasterBusinessAdministration(AbstractDegree
):
    def info(self):
        print('Master of business
administration')

    def __str__(self):
        return 'Master of business
administration'


class ProfileAbstractFactory(object):
    def __init__(self):
        self._degrees = []
        self.create_profile()


    @abstractmethod
    def create_profile(self): pass

    def add_degree(self, degree):
        self._degrees.append(degree)

    def get_degrees(self):
        return self._degrees


class
ManagerFactory(ProfileAbstractFactory):
    def create_profile(self):
        self.add_degree(BachelorEngineer())

self.add_degree(MasterBusinessAdministratio
n())


class
EngineerFactory(ProfileAbstractFactory):
    def create_profile(self):
        self.add_degree(BachelorEngineer())
        self.add_degree(MasterEngineer())


class ProfileCreatorFactory(object):
    @classmethod
    def create_profile(cls, name):
        return
eval(f'{profile_type}Factory')()


if __name__ == '__main__':
    profile_type = input('Which Profile
would you like to create? Manager/Engineer
- ')
    profile =
ProfileCreatorFactory.create_profile(profil
e_type)
    print(f'Creating Profile of
{profile_type}')
    print('Profile has following degrees:')
    for deg in profile.get_degrees():
        print(f' - {deg}')
```

Abstract Factory

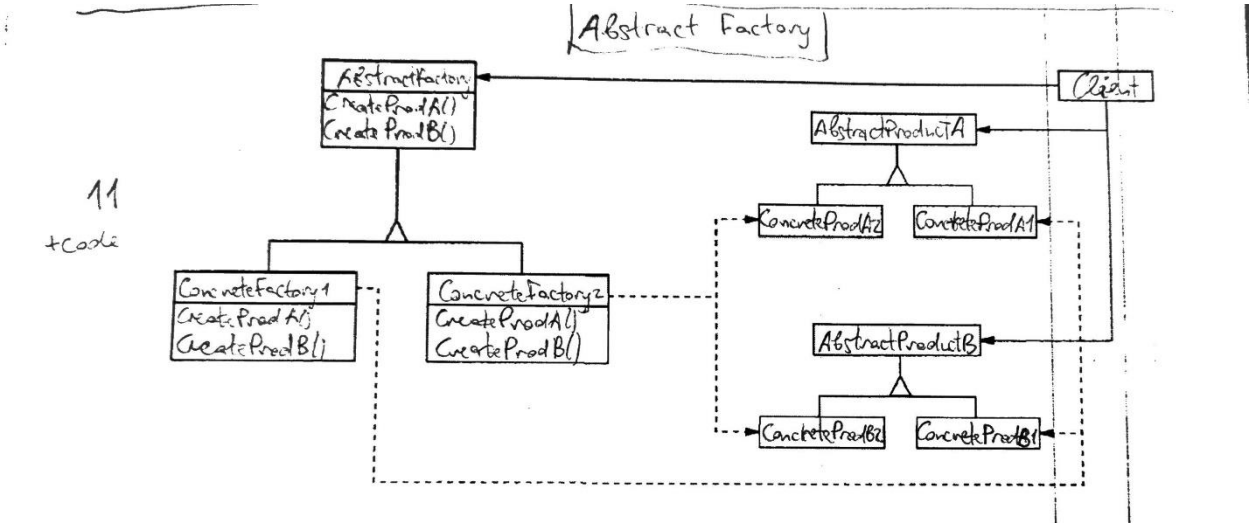
- Intentia sablonului

Abstract Factory este un sablon de proiectare creational care ofera o interfata pentru crearea unui set de obiecte legate logic, fara a specifica clasele concrete.

- Problema care o rezolva

Uneori este necesar sa se creeze un set de obiecte legate logic, cum ar fi obiecte care trebuie sa functioneze impreuna pentru a realiza o anumita functionalitate. Dar nu se poate decide ce clase concrete trebuie utilizate pana la momentul rularii programului. Abstract Factory rezolva aceasta problema prin oferirea unei interfete pentru crearea acestui set de obiecte, fara a specifica clasele concrete.

- Componente cu o mica descriere
  1. AbstractFactory: Clasa abstracta care defineste interfata pentru crearea obiectelor legate logic.
  2. ConcreteFactory: Clasa care implementeaza AbstractFactory si returneaza un set de obiecte legate logic.
  3. AbstractProduct: Clasa abstracta a obiectului legat logic.
  4. ConcreteProduct: Clasa care implementeaza AbstractProduct si reprezinta un obiect concret legat logic.



Abstract Factory (exemplu de cod)

```
from abc import ABCMeta, abstractmethod


class CarFactory(metaclass=ABCMeta):
    @abstractmethod
    def build_parts(self): pass

    @abstractmethod
    def build_car(self): pass


class SedanCarFactory(CarFactory):
    def build_parts(self):
        return SedanCarPartsFactory()

    def build_car(self):
        return SedanCarAssembleFactory()


class SUVCarFactory(CarFactory):
    def build_parts(self):
        return SUVCarPartsFactory()

    def build_car(self):
        return SUVCarAssembleFactory()


class CarPartsFactory(metaclass=ABCMeta):
    @abstractmethod
    def build(self): pass


class
SedanCarPartsFactory(CarPartsFactory):
    def build(self):
        print('Sedan car parts are built')

    def __str__(self):
        return '\'Sedan car parts\''


class SUVCarPartsFactory(CarPartsFactory):
    def build(self):
        print('SUV Car parts are built')

    def __str__(self):
        return '\'SUV car parts\''


class
CarAssembleFactory(metaclass=ABCMeta):
    @abstractmethod
    def assemble(self, parts): pass


class
SedanCarAssembleFactory(CarAssembleFactory)
:
    def assemble(self, parts):
        print(f'Sedan car is assembled here
using {parts}')


class
SUVCarAssembleFactory(CarAssembleFactory):
    def assemble(self, parts):
        print(f'SUV car is assembled here
using {parts}')


if __name__ == '__main__':
    for factory in (SedanCarFactory(),
SUVCarFactory()):
        car_parts = factory.build_parts()
        car_parts.build()
        car_builder = factory.build_car()
        car_builder.assemble(car_parts)
```

Builder

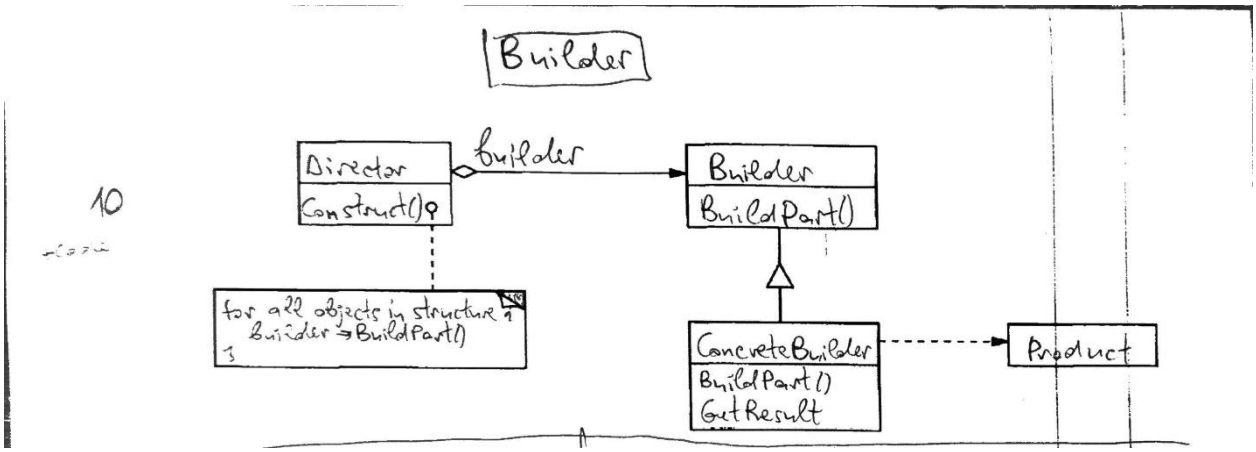
- Intentia sablonului

Builder este un sablon de proiectare creational care separa procesul de constructie a unui obiect complex de reprezentarea acestuia, astfel incat aceleasi procese de constructie pot fi utilizate pentru crearea de obiecte diferite.

- Problema care o rezolva

Uneori este necesar sa se creeze obiecte complexe, cu multiple componente sau pasi de constructie, cum ar fi documente sau structuri de date. Builder rezolva aceasta problema prin separarea procesului de constructie de reprezentarea obiectului final, astfel incat aceleasi procese de constructie pot fi utilizate pentru crearea de obiecte diferite.

- Componente cu o mica descriere
  1. Builder: Interfata care defineste metodele necesare pentru construirea obiectului.
  2. ConcreteBuilder: Clasa care implementeaza Builder si construieste obiectul complex.
  3. Product: Clasa care reprezinta obiectul complex care este construit de ConcreteBuilder.
  4. Director: Clasa care controleaza procesul de constructie si utilizeaza ConcreteBuilder pentru a construi obiectul complex.



Builder (exemplu de cod)

```
class Director:
    __builder = None

    def set_builder(self, builder):
        self.__builder = builder

    def get_car(self):
        car = Car()

        body = self.__builder.get_body()
        car.set_body(body)

        engine =
self.__builder.get_engine()
        car.set_engine(engine)

        i = 0
        while i < 4:
            wheel =
self.__builder.get_wheel()
            car.attach_wheel(wheel)
            i += 1

        return car


class Car:
    def __init__(self):
        self.__wheels = list()
        self.__engine = None
        self.__body = None

    def set_body(self, body):
        self.__body = body

    def attach_wheel(self, wheel):
        self.__wheels.append(wheel)

    def set_engine(self, engine):
        self.__engine = engine

    def specification(self):
        print(f'body: {self.__body.shape}')
        print(f'engine horsepower:
{self.__engine.horsepower}')
        print(f'tire size:
{self.__wheels[0].size}')


class Builder:
    def get_wheel(self): pass

    def get_engine(self): pass

    def get_body(self): pass


class JeepBuilder(Builder):
    def get_wheel(self):
        wheel = Wheel()
        wheel.size = 22
        return wheel

    def get_engine(self):
        engine = Engine()
        engine.horsepower = 400
        return engine

    def get_body(self):
        body = Body()
        body.shape = "SUV"
        return body


class NissanBuilder(Builder):
    def get_wheel(self):
        wheel = Wheel()
        wheel.size = 16
        return wheel

    def get_engine(self):
        engine = Engine()
        engine.horsepower = 85
        return engine

    def get_body(self):
        body = Body()
        body.shape = "hatchback"
        return body


class Wheel:
    size = None


class Engine:
    horsepower = None


class Body:
    shape = None


if __name__ == "__main__":
    jeep_builder = JeepBuilder()
    nissan_builder = NissanBuilder()

    director = Director()

    print('Jeep')
    director.set_builder(jeep_builder)
    jeep = director.get_car()
    jeep.specification()

    print('Nissan')
    director.set_builder(nissan_builder)
    nissan = director.get_car()
    nissan.specification()
```

Prototype

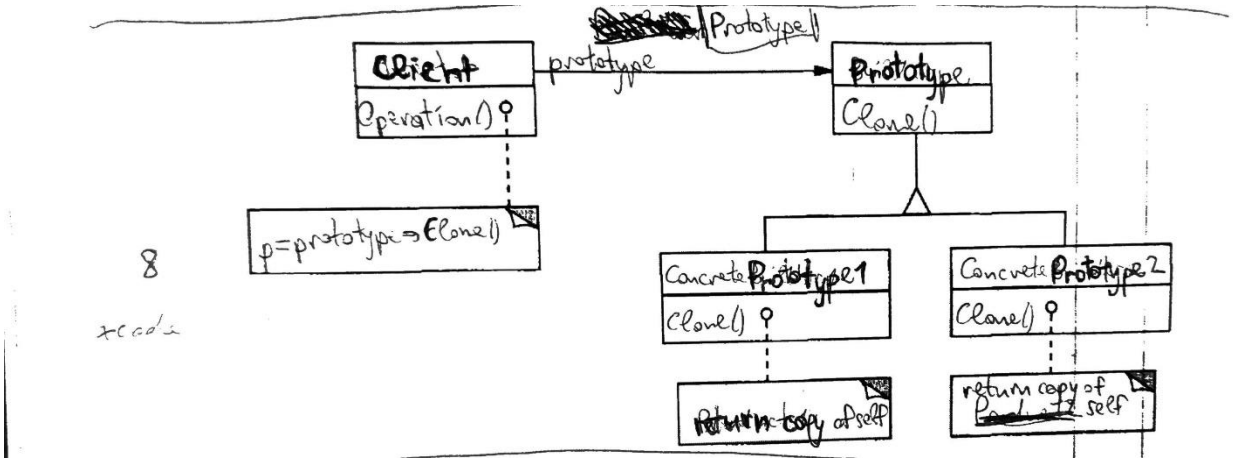
- Intentia sablonului

Prototype este un sablon de proiectare creational care permite crearea de obiecte noi prin clonarea unui obiect existent, fara a depinde de clasele concrete.

- Problema care o rezolva

Uneori este necesar sa se creeze obiecte noi cu proprietati similare cu cele ale unui obiect existent, dar fara a crea o dependenta intre obiectele noi si cele existente. Prototype rezolva aceasta problema prin permiterea clonarii unui obiect existent pentru a crea unul nou.

- Componente cu o mica descriere
  1. Prototype: Interfata care defineste metoda pentru clonarea obiectelor.
  2. ConcretePrototype: Clasa care implementeaza Prototype si ofera o metoda pentru clonarea obiectelor.
  3. Client: Clasa care utilizeaza ConcretePrototype pentru a crea obiecte noi prin clonare.



Prototype (exemplu de cod)

```
from abc import ABCMeta, abstractmethod
import copy


class CoursesPrototype(metaclass=ABCMeta):
    def __init__(self):
        self.id = None
        self.type = None

    def get_type(self):
        return self.type

    def get_id(self):
        return self.id

    def set_id(self, sid):
        self.id = sid

    def clone(self):
        return copy.copy(self)


class Python(CoursesPrototype):
    def __init__(self):
        super().__init__()
        self.type = 'Python Basic and
Algorithm'


class Java(CoursesPrototype):
    def __init__(self):
        super().__init__()
        self.type = 'Java Basics and Spring
Boot'


class R(CoursesPrototype):
    def __init__(self):
        super().__init__()
        self.type = "R programming
language"


class CoursesCache:
    cache = {}

    @staticmethod
    def get_course(sid):
        course =
CoursesCache.cache.get(sid, None)
        return course.clone()

    @staticmethod
    def load():
        python = Python()
        python.set_id('1')
        CoursesCache.cache[python.get_id()]
= python

        java = Java()
        java.set_id('2')
        CoursesCache.cache[java.get_id()] =
java

        r = R()
        r.set_id('3')
        CoursesCache.cache[r.get_id()] = r


if __name__ == '__main__':
    CoursesCache.load()

    Python = CoursesCache.get_course('1')
    print(Python.get_type())

    Java = CoursesCache.get_course('2')
    print(Java.get_type())

    R = CoursesCache.get_course('3')
    print(R.get_type())
```

STRUCTURAL PATTERNS

Adapter

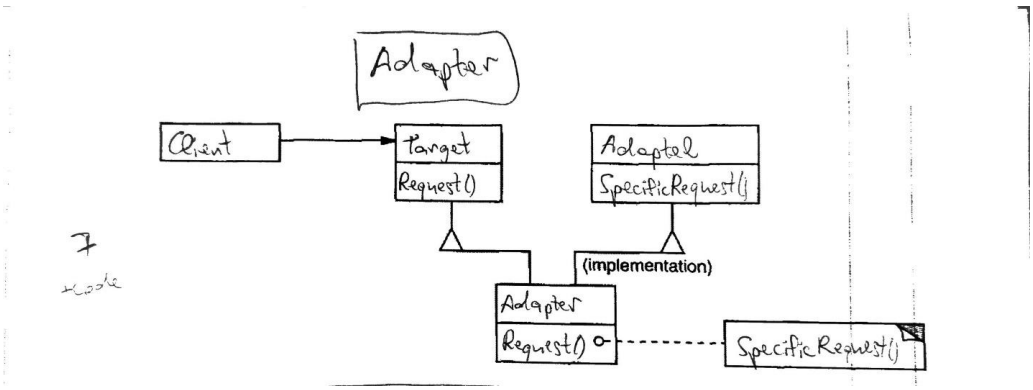
- Intentia sablonului

Adapter este un sablon de proiectare structural care permite obiectelor incompatibile sa lucreze impreuna prin intermediul unui obiect adaptor.

- Problema care o rezolva

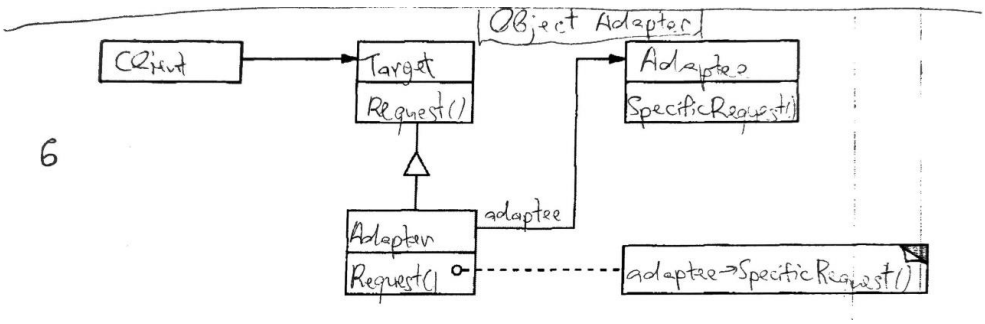
Uneori, se poate intampla ca doua obiecte sa fie incompatibile din punct de vedere al interfetei, ceea ce inseamna ca nu pot lucra impreuna fara a modifica unul dintre acestea. Adapter rezolva aceasta problema prin utilizarea unui obiect adaptor care converteşte interfata unui obiect in alta, astfel incat cele doua obiecte sa poata lucra impreuna.

- Componente cu o mica descriere
  1. Target: Interfata care reprezinta interfata dorita de client.
  2. Adaptee: Clasa care contine functionalitatea existenta, dar nu are interfata dorita de client.
  3. Adapter: Clasa care converteşte interfata Adaptee in interfata Target, astfel incat Adaptee poate fi utilizat de client prin intermediul Adapter.



Un Adapter este un sablon de proiectare care permite unui obiect sa fie adaptat la interfata altui obiect, astfel incat acestea sa poata colabora. Aceasta se face prin crearea unui obiect intermediar, numit adapter, care transforma interfata unui obiect in alta interfata asteptata de client.

Un Object Adapter este o varianta a sablonului de proiectare Adapter care utilizeaza compozitia pentru a adapta obiectul la interfata dorita. Acesta utilizeaza un obiect existent si il adapteaza la interfata necesara prin intermediul compozitiei. In acest fel, un obiect poate fi adaptat la mai multe interfete diferite.



Adapter (exemplu de cod)

```
import math


class RoundHole:
    def __init__(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def fits(self, peg):
        return self.get_radius() >=
peg.get_radius()


class RoundPeg:
    def __init__(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius


class SquarePeg:
    def __init__(self, width):
        self.__width = width

    def get_width(self) -> float:
        return self.__width


class SquarePegAdapter:
    peg: SquarePeg

    def __init__(self, peg: SquarePeg):
        self.peg = peg

    def get_radius(self):
        return self.peg.get_width() *
math.sqrt(2) / 2


if __name__ == "__main__":
    hole = RoundHole(5)
    round_peg = RoundPeg(5)

    print(f"Hole radius:
{hole.get_radius()}\n")
    print(f"Round Peg radius:
{round_peg.get_radius()}")
    print(f"The round peg fits the hole?
{hole.fits(round_peg)}\n")

    sm_square_peg = SquarePeg(7)
    lg_square_peg = SquarePeg(10)

    sm_square_peg_adapter =
SquarePegAdapter(sm_square_peg)
    lg_square_peg_adapter =
SquarePegAdapter(lg_square_peg)
    print(f"Small peg radius:
{sm_square_peg_adapter.get_radius()}")
    print(f"The small square peg fits the
hole? {hole.fits(sm_square_peg_adapter)}")
    print(f"Large peg radius:
{lg_square_peg_adapter.get_radius()}")
    print(f"The large square peg fits the
hole? {hole.fits(lg_square_peg_adapter)}")
```

Bridge

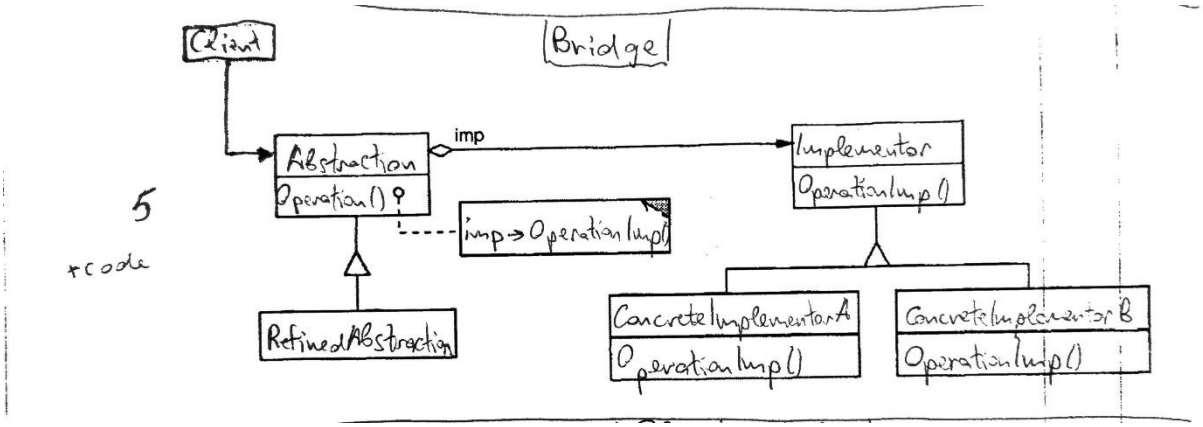
- Intentia sablonului

Bridge este un sablon de proiectare structural care permite separarea interfetei si implementarii unei clase, astfel incat acestea sa poata fi modificate independent.

- Problema care o rezolva

Uneori, este necesar sa se modifice interfata sau implementarea unei clase fara a afecta celelalte. Bridge rezolva aceasta problema prin separarea interfetei si implementarii unei clase, astfel incat acestea sa poata fi modificate independent.

- Componente cu o mica descriere
  1. Abstraction: Clasa abstracta care defineste interfata pentru client.
  2. RefinedAbstraction: Clasa care extinde Abstraction si adauga functionalitate suplimentara.
  3. Implementor: Interfata care defineste interfata pentru implementare.
  4. ConcreteImplementor: Clasa care implementeaza Implementor.



Bridge (exemplu de cod)

```
from abc import ABCMeta


class Carrier(metaclass=ABCMeta):
    def carry_military(self, items):
        pass

    def carry_commercial(self, items):
        pass


class Cargo(Carrier):
    def carry_military(self, items):
        print(f"The plane carries {items}
military cargo goods")

    def carry_commercial(self, items):
        print(f"The plane carries {items}
commercial cargo goods")


class Passenger(Carrier):
    def carry_military(self, passengers):
        print(f"The plane carries
{passengers} military passengers")

    def carry_commercial(self, passengers):
        print(f"The plane carries
{passengers} commercial passengers")


class Plane:
    def __init__(self, carrier):
        self.carrier = carrier

    def display_description(self):
        pass

    def add_objects(self, new_objects):
        pass


class Commercial(Plane):
    def __init__(self, carrier, objects):
        super().__init__(carrier)
        self.objects = objects

    def display_description(self):

self.carrier.carry_commercial(self.objects)

    def add_objects(self, new_objects):
        self.objects += new_objects


class Military(Plane):
    def __init__(self, carrier, objects):
        super().__init__(carrier)
        self.objects = objects

    def display_description(self):

self.carrier.carry_military(self.objects)

    def add_objects(self, new_objects):
        self.objects += new_objects


if __name__ == "__main__":
    cargo = Cargo()
    passenger = Passenger()

    military_plane = Military(passenger,
100)
    military_plane.display_description()
    military_plane.add_objects(25)
    military_plane.display_description()

    commercialPlane = Commercial(cargo,
200)
    commercialPlane.display_description()
```

Composite

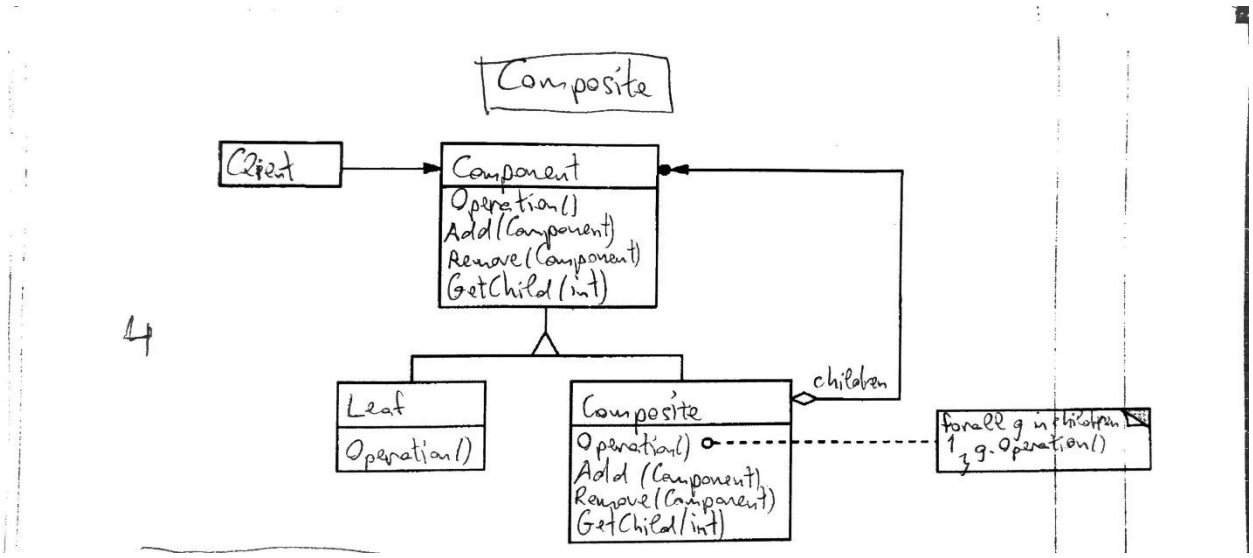
- Intentia sablonului

Composite este un sablon de proiectare structural care permite crearea de obiecte ierarhice si tratarea lor la fel ca si obiecte individuale.

- Problema care o rezolva

Uneori, este necesar sa se trateze un grup de obiecte intr-un mod similar cu un obiect individual. Composite rezolva aceasta problema prin permiterea gruparii de obiecte intr-o ierarhie si tratarea lor la fel ca si obiecte individuale.

- Componente cu o mica descriere
  1. Component: Interfata care defineste metodele comune pentru obiecte individuale si compozite.
  2. Leaf: Clasa care reprezinta obiectele individuale.
  3. Composite: Clasa care contine subcomponente si le gestioneaza.



Composite (exemplu de cod)

```
from abc import ABC, abstractmethod

# Clasa abstracta Componenta care defineste
o interfata comuna pentru toate
componentele
class Component(ABC):

    @abstractmethod
    def afiseaza_informatii(self):
        pass


# Clasa Frunza care reprezinta un angajat
class Angajat(Component):

    def __init__(self, nume, salariu):
        self.nume = nume
        self.salariu = salariu

    def afiseaza_informatii(self):
        print(f"Angajat: {self.nume},
Salariu: {self.salariu}")


# Clasa Composite care reprezinta un
departament sau o subdiviziune a unei
companii
class Departament(Component):

    def __init__(self, nume):
        self.nume = nume
        self.componente = []

    def adauga(self, componenta):
        self.componente.append(componenta)

    def sterge(self, componenta):
        self.componente.remove(componenta)

    def afiseaza_informatii(self):
        print(f"Departament: {self.nume}")
        for componente in self.componente:

componente.afiseaza_informatii()


# Exemplu de utilizare a sablonului de
proiectare Composite pentru a crea o
companie
if __name__ == '__main__':
    companie = Departament("Companie")

    it = Departament("IT")
    it.adauga(Angajat("John Smith", 5000))
    it.adauga(Angajat("Jane Doe", 6000))

    contabilitate =
Departament("Contabilitate")
    contabilitate.adauga(Angajat("Michael
Brown", 4000))

    marketing = Departament("Marketing")
    marketing.adauga(Angajat("Sarah
Johnson", 7000))
    marketing.adauga(Angajat("David Lee",
8000))

    companie.adauga(it)
    companie.adauga(contabilitate)
    companie.adauga(marketing)

    companie.afiseaza_informatii()
```

Decorator

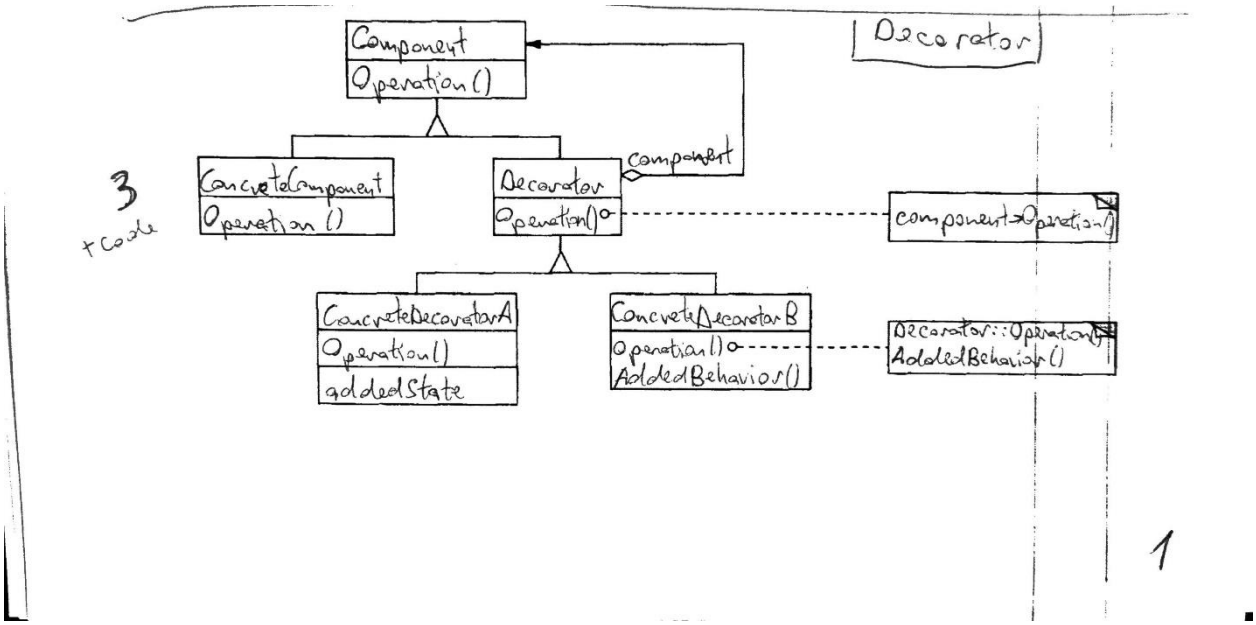
- Intentia sablonului

Decorator este un sablon de proiectare structural care permite adaugarea de functionalitati suplimentare la un obiect existent, fara a afecta alte instante ale acestei clase.

- Problema care o rezolva

Uneori, este necesar sa se adauge functionalitati suplimentare la un obiect, dar nu este posibil sau practic sa se modifice clasa originala. Decorator rezolva aceasta problema prin utilizarea de obiecte decoratoare care adauga functionalitati suplimentare la un obiect existent.

- Componente cu o mica descriere
    1. Component: Clasa abstracta care defineste interfata pentru obiectul de baza si obiectele decorate.
    2. ConcreteComponent: Clasa care implementeaza Component si ofera functionalitatea de baza.
    3. Decorator: Clasa abstracta care defineste interfata pentru obiectele care adauga functionalitati suplimentare.
    4. ConcreteDecorator: Clasa care implementeaza Decorator si adauga functionalitati suplimentare la un obiect.



Decorator (exemplu de cod)

```
from abc import ABCMeta


class AbstractCar(metaclass=ABCMeta):
    def get_cost(self) -> float: pass

    def get_parts(self): pass

    def get_tax(self):
        return 0.05 * self.get_cost()


class ConcreteCar(AbstractCar):
    def get_cost(self):
        return 10000

    def get_parts(self):
        return "initial car"


class AbstractCarDecorator(AbstractCar):
    def __init__(self, decorated_car):
        self.decorated_car = decorated_car

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

    def get_parts(self):
        return
self.decorated_car.get_parts()


class Engine(AbstractCarDecorator):
    def __init__(self, decorated_car):
        AbstractCarDecorator.__init__(self,
decorated_car)

    def get_cost(self):
        return
self.decorated_car.get_cost() + 5000

    def get_parts(self):
        return
self.decorated_car.get_parts() + " +
engine"


class Turbocharger(AbstractCarDecorator):
    def __init__(self, decorated_car):
        AbstractCarDecorator.__init__(self,
decorated_car)

    def get_cost(self):
        return
self.decorated_car.get_cost() + 900

    def get_parts(self):
        return
self.decorated_car.get_parts() + " +
turbocharger"


class Exhaust(AbstractCarDecorator):
    def __init__(self, decorated_car):
        AbstractCarDecorator.__init__(self,
decorated_car)

    def get_cost(self):
        return
self.decorated_car.get_cost() + 500

    def get_parts(self):
        return
self.decorated_car.get_parts() + " +
exhaust"


if __name__ == "__main__":
    car = ConcreteCar()
    print(f"Parts ({car.get_parts()}):\n\t"
          f"Cost: ${car.get_cost()} | Sales
tax: ${car.get_tax()}")

    car = Turbocharger(car)
    print(f"Parts ({car.get_parts()}):\n\t"
          f"Cost: ${car.get_cost()} | Sales
tax: ${car.get_tax()}")

    car = Exhaust(car)
    print(f"Parts ({car.get_parts()}):\n\t"
          f"Cost: ${car.get_cost()} | Sales
tax: ${car.get_tax()}")

    car = Engine(car)
    print(f"Parts ({car.get_parts()}):\n\t"
          f"Cost: ${car.get_cost()} | Sales
tax: ${car.get_tax()}")
```

Facade

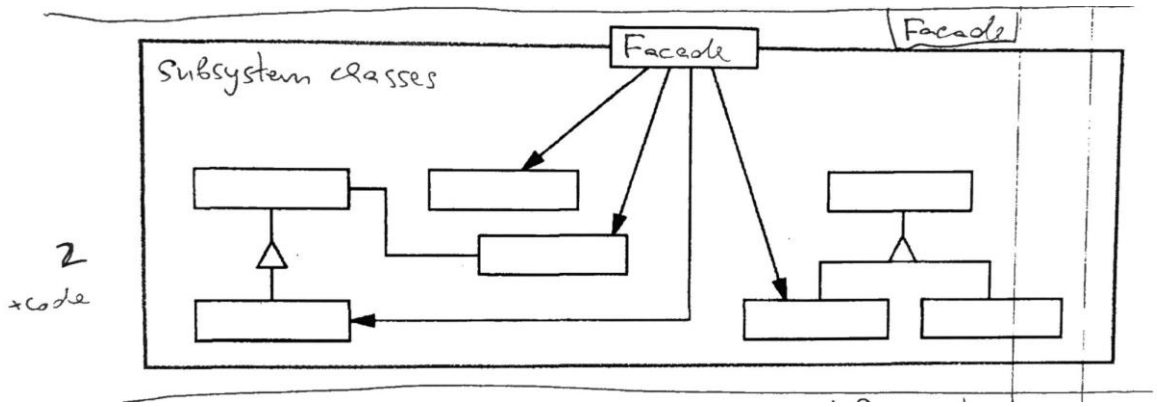
- Intentia sablonului

Facade este un sablon de proiectare structural care ofera o interfata simplificata catre un ansamblu complex de clase.

- Problema care o rezolva

Uneori, o aplicatie poate fi formata dintr-un ansamblu de clase interconectate intr-un mod complex. Facade rezolva aceasta problema prin oferirea unei interfete simple catre acest ansamblu de clase, astfel incat utilizatorii sa poata interactiona cu aplicatia fara a cunoaste complexitatea din spate.

- Componente cu o mica descriere
  1. Facade: Clasa care ofera o interfata simplificata catre ansamblul complex de clase.
  2. Subsystem classes: Clasele care formeaza ansamblul complex si care sunt ascunse de catre Facade.



Facade (exemplu de cod)

```
class CPU:
    def freeze(self):
        print("Freezing processor.")

    def jump(self, position):
        print("Jumping to:", position)

    def execute(self):
        print("Executing.")


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


class SolidStateDrive:
    def read(self, lba, size):
        return f"Some data from sector {lba} with size {size}"


class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.ssd = SolidStateDrive()

    def start(self):
        self.cpu.freeze()
        self.memory.load("0x00", self.ssd.read("100", "1024"))
        self.cpu.jump("0x00")
        self.cpu.execute()


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

Flyweight

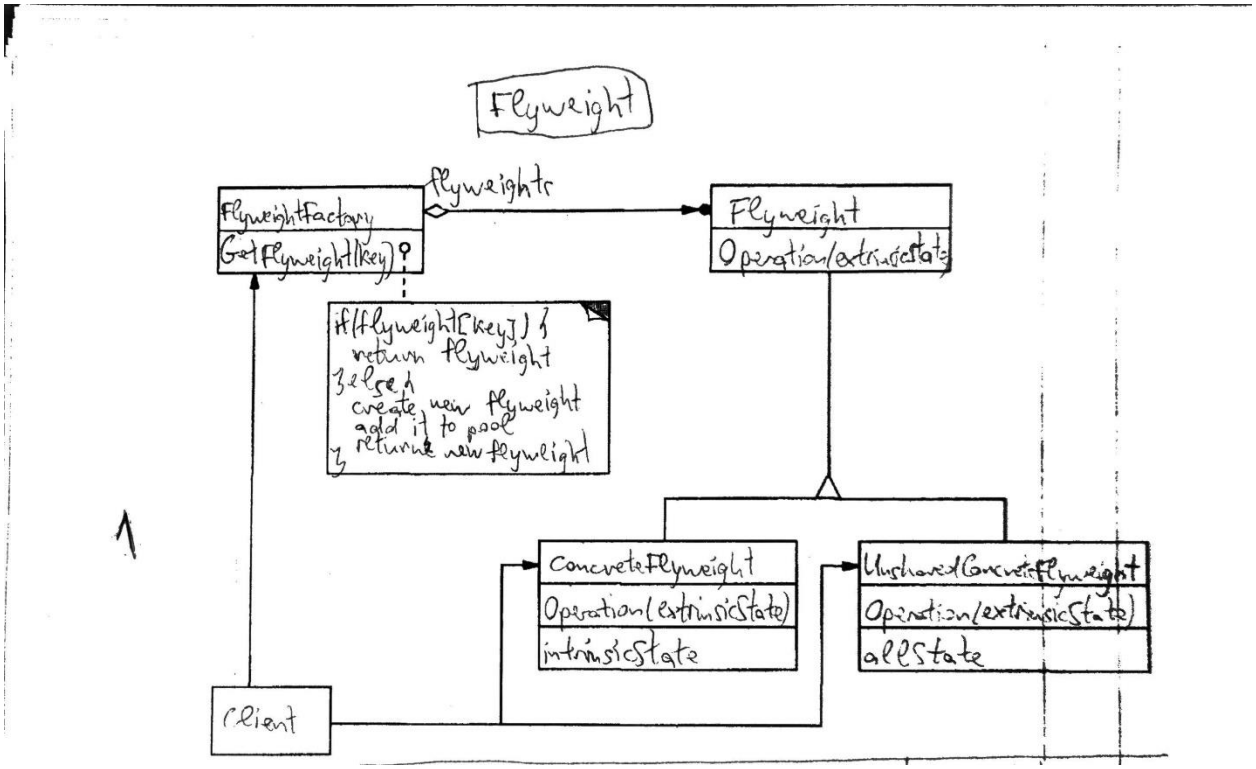
- Intentia sablonului

Flyweight este un sablon de proiectare structural care permite utilizarea eficienta a unui numar mare de obiecte mici, prin partajarea lor intre mai multe entitati.

- Problema care o rezolva

Uneori, o aplicatie poate utiliza un numar mare de obiecte mici care consuma multa memorie. Flyweight rezolva aceasta problema prin partajarea obiectelor intre mai multe entitati, astfel incat acestea sa poata fi utilizate eficient.

- Componente cu o mica descriere
  1. Flyweight: Clasa abstracta care defineste interfata pentru obiectele Flyweight.
  2. ConcreteFlyweight: Clasa care implementeaza Flyweight si contine informatii specifice unui obiect.
  3. FlyweightFactory: Clasa care creeaza si mentine obiectele Flyweight. Aceasta asigura ca obiectele Flyweight sunt partajate intre mai multe entitati.



Flyweight (exemplu de cod)

```
from typing import Dict, Tuple

# Clasa Flyweight care reprezinta obiectul partajat
class Forma:

    def deseneaza(self, x: int, y: int):
        pass


# Clasa FlyweightConcret care reprezinta o implementare a obiectului partajat
class Cerc(Forma):

    def __init__(self, culoare: str):
        self.culoare = culoare

    def deseneaza(self, x: int, y: int):
        print(f"Desenare cerc cu culoarea {self.culoare} la coordonatele ({x}, {y})")


# Clasa Factory Flyweight care gestioneaza obiectele partajate
class FabricaForme:

    def __init__(self):
        self.forme = {}

    def get_forma(self, culoare: str) -> Forma:
        forma = self.forme.get(culoare)
        if not forma:
            forma = Cerc(culoare)
            self.forme[culoare] = forma
        return forma


# Exemplu de utilizare a sablonului de proiectare Flyweight pentru a desena cercuri de culori
diferite
if __name__ == '__main__':
    fabrica_forme = FabricaForme()

    cercuri = [
        (1, 2, 'rosu'),
        (3, 4, 'verde'),
        (5, 6, 'rosu'),
        (7, 8, 'albastru'),
        (9, 10, 'verde'),
        (11, 12, 'rosu'),
        (13, 14, 'albastru'),
        (15, 16, 'verde'),
    ]

    for coord in cercuri:
        x, y, culoare = coord
        cerc = fabrica_forme.get_forma(culoare)
        cerc.deseneaza(x, y)
```

Proxy

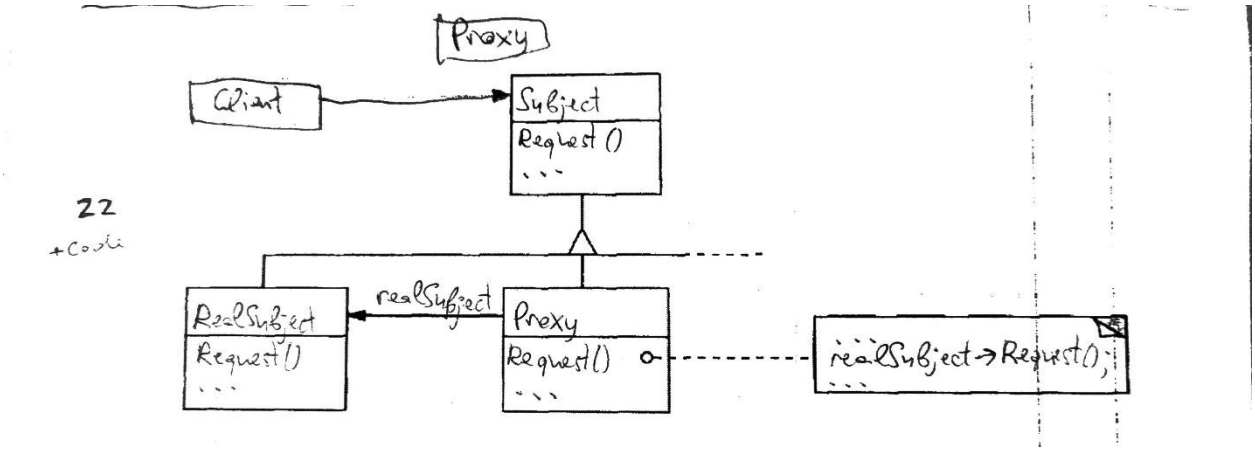
- Intentia sablonului

Proxy este un sablon de proiectare structural care ofera o interfata pentru un obiect existent, permitand astfel controlul accesului la acesta.

- Problema care o rezolva

Uneori, este necesar sa se controleze accesul la un obiect, de exemplu pentru a restricta anumite operatii sau pentru a gestiona resursele limitate. Proxy rezolva aceasta problema prin oferirea unei interfete pentru obiectul existent, dar cu un control suplimentar asupra accesului la acesta.

- Componente cu o mica descriere
  1. Subject: Clasa abstracta care defineste interfata pentru obiectul existent si Proxy.
  2. RealSubject: Clasa care implementeaza Subject si reprezinta obiectul existent.
  3. Proxy: Clasa care implementeaza Subject si controleaza accesul la RealSubject.



Proxy (exemplu de cod)

```
class Image:
    def __init__(self, filename):
        self._filename = filename

    def load_image_from_disk(self):
        print("loading " + self._filename)

    def display_image(self):
        print("display " + self._filename)


class Proxy:
    def __init__(self, subject):
        self._subject = subject
        self._proxy_state = None


class ProxyImage(Proxy):
    def display_image(self):
        if self._proxy_state is None:
            self._subject.load_image_from_disk()
            self._proxy_state = 1
        print("display " + self._subject._filename)


if __name__ == "__main__":
    proxy_image1 = ProxyImage(Image("HiRes_10Mb_Photo1"))
    proxy_image2 = ProxyImage(Image("HiRes_10Mb_Photo2"))

    proxy_image1.display_image()  # loading necessary
    proxy_image1.display_image()  # loading unnecessary
    proxy_image2.display_image()  # loading necessary
    proxy_image2.display_image()  # loading unnecessary
    proxy_image1.display_image()  # loading unnecessary
```

BEHAVIORAL PATTERNS

Chain of Responsibility

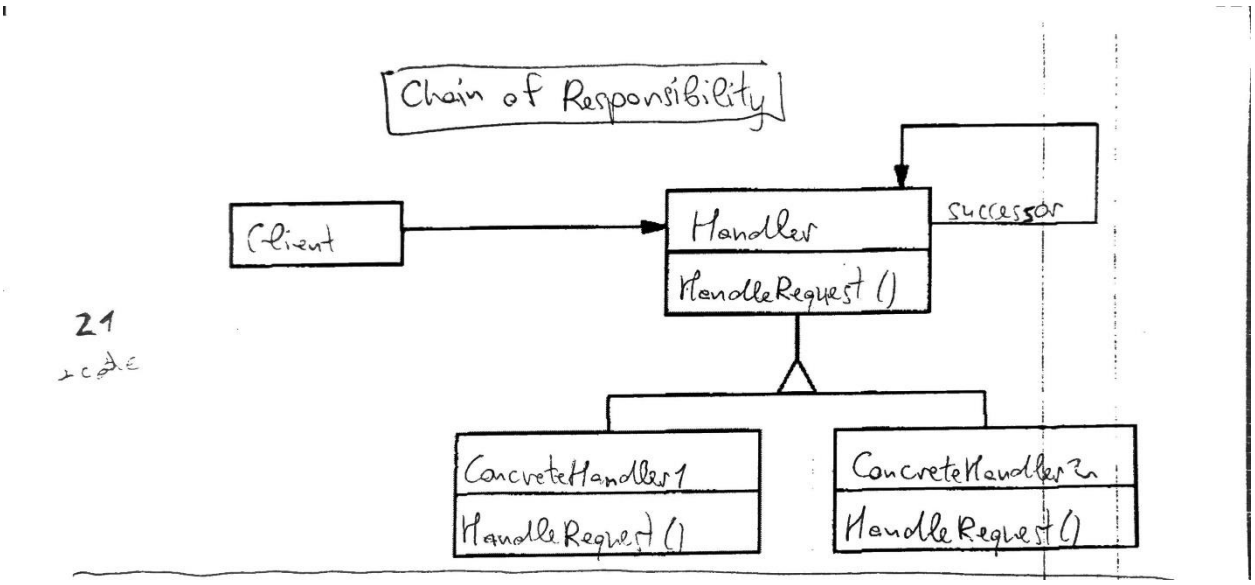
- Intentia sablonului

Chain of Responsibility este un sablon de proiectare comportamental care permite construirea unui lant de obiecte care pot procesa un anumit request, astfel incat request-ul sa fie trimis catre unul dintre aceste obiecte care il poate procesa.

- Problema care o rezolva

Uneori, este necesar sa se proceseze un request primit intr-un mod flexibil si eficient. Chain of Responsibility rezolva aceasta problema prin permiterea construirii unui lant de obiecte care pot procesa request-ul si trimite request-ul catre unul dintre aceste obiecte care il poate procesa.

- Componente cu o mica descriere
  1. Handler: Clasa abstracta care defineste interfata pentru procesarea request-urilor si pentru a seta urmatorul handler in lant.
  2. ConcreteHandler: Clasa care implementeaza Handler si poate procesa anumite tipuri de request-uri. Aceasta poate trimite request-ul mai departe catre urmatorul handler in lant.



Chain of Responsibility (exemplu de cod)

```
import pdfkit

path_wkhtmltopdf = r'C:\Program
Files\wkhtmltopdf\bin\wkhtmltopdf.exe'
config =
pdfkit.configuration(wkhtmltopdf=path_wkhtm
ltopdf)


class ReportFormat:
    PDF = 0
    TEXT = 1


class Report:
    def __init__(self, format_):
        self.title = 'Monthly report'
        self.text = ['Things are going',
'really, really well.']
        self.format_ = format_


class Handler:
    def __init__(self):
        self.nextHandler = None

    def handle(self, request):
        self.nextHandler.handle(request)


class PDFHandler(Handler):
    def handle(self, request):
        if request.format_ ==
ReportFormat.PDF:

self.output_report(request.title,
request.text)
        else:
            super(PDFHandler,
self).handle(request)

    def output_report(self, title, text):
        string = '<html>'
        string += ' <head>'
        string += ' <title>%s</title>' %
title
        string += ' </head>'
        string += ' <body>'
        for line in text:
            string += ' <p>%s' % line
        string += ' </body>'
        string += '</html>'
        print(string)
        pdfkit.from_string(string,
'GfG.pdf', configuration=config)


class TextHandler(Handler):

    def handle(self, request):
        if request.format_ ==
ReportFormat.TEXT:

self.output_report(request.title,
request.text)
        else:
            super(TextHandler,
self).handle(request)

    def output_report(self, title, text):
        print(5 * '*' + title + 5 * '*')
        for line in text:
            print(line)


class ErrorHandler(Handler):
    def handle(self, request):
        print("Invalid request")


if __name__ == '__main__':
    report = Report(ReportFormat.PDF)
    pdf_handler = PDFHandler()
    text_handler = TextHandler()
    pdf_handler.handle(report)
    text_handler.handle(report)
    pdf_handler.nextHandler = text_handler
    text_handler.nextHandler = pdf_handler
    text_handler.handle(report)
```

Command

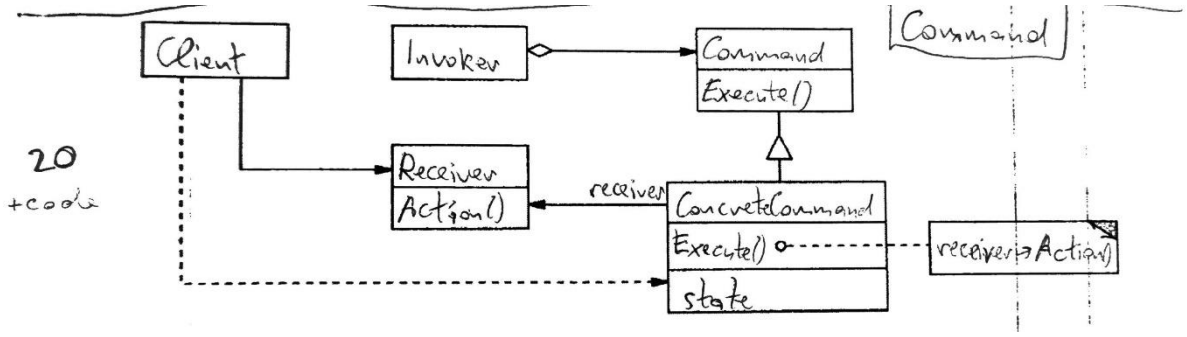
- Intentia sablonului

Command este un sablon de proiectare comportamental care transforma o solicitare intr-un obiect independent de solicitant si care poate fi utilizat pentru a parametriza obiecte diferite cu diferite solicitari.

- Problema care o rezolva

Uneori, este necesar sa se trimita o solicitare catre un obiect, dar nu se cunoaste in prealabil obiectul specific care va procesa solicitarea sau ca procesarea solicitarilor necesita stocarea istoricului solicitărilor pentru undo/redo. Command rezolva aceasta problema prin encapsularea unei solicitari intr-un obiect independent de solicitant si care poate fi utilizat pentru a parametriza obiecte diferite cu diferite solicitari.

- Componente cu o mica descriere
  1. Command: Clasa abstracta care defineste interfata pentru toate comenzile. Aceasta ofera o metoda execute pentru a procesa comanda.
  2. ConcreteCommand: Clasa care implementeaza Command si contine o referinta catre obiectul receptor si o metoda execute care apeleaza o anumita actiune asupra receptorului.
  3. Receiver: Clasa care contine logica efectiva pentru procesarea comenzilor.
  4. Invoker: Clasa care trimite comenzi catre obiecte Receiver. Aceasta stie cum sa execute o anumita comanda si stie care comanda sa trimita la care receptor.
  5. Client: Clasa care creaza obiectele ConcreteCommand si configureaza obiectele Invoker cu aceste comenzi.



Command (exemplu de cod)

```
from abc import ABC, abstractmethod


class BaseCommand(ABC):
    @abstractmethod
    def execute(self): pass


class EMailCommand(BaseCommand):
    def __init__(self, receiver, data):
        self.receiver = receiver
        self.data = data

    def execute(self):
        self.receiver.send_email(self.data)


class SMSCommand(object):
    def __init__(self, receiver, data):
        self.receiver = receiver
        self.data = data

    def execute(self):
        self.receiver.send_sms(self.data)


class NotificationService(object):
    def send_email(self, data):
        print("Sending Email", data)

    def send_sms(self, data):
        print("Sending SMS", data)


class NotificationInvoker(object):
    def __init__(self):
        self.notification_history = []

    def invoke(self, command):
        self.notification_history.append(command)
        command.execute()


if __name__ == "__main__":
    invoker = NotificationInvoker()
    sender = NotificationService()
    invoker.invoke(EMailCommand(sender, {"subject": "Test Email subject", "body": "test Email
body"}))
    invoker.invoke(SMSCommand(sender, {"message": "Test SMS message"}))
    print(f"Notification history: {invoker.notification_history}")
```

Iterator

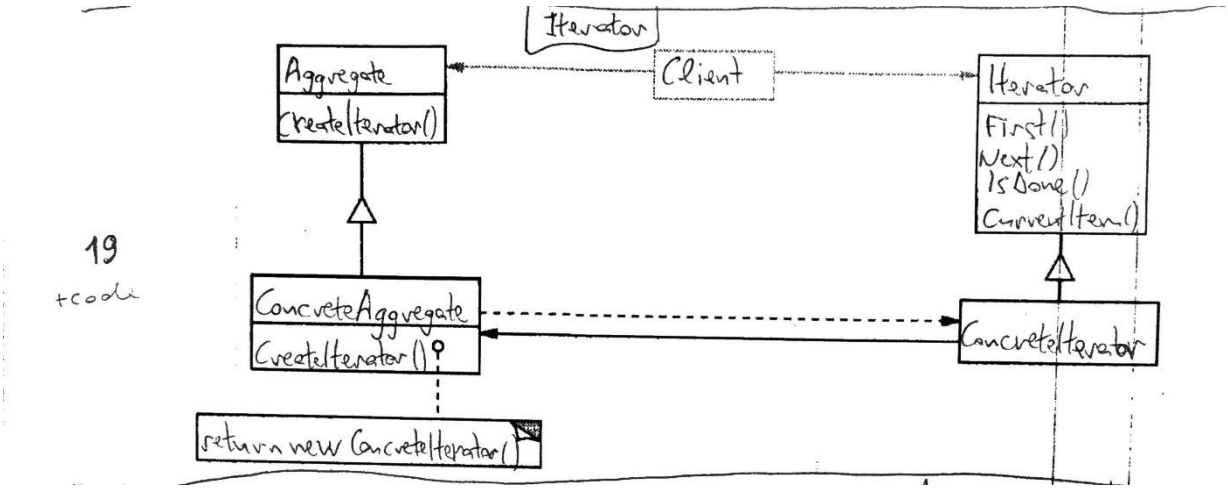
- Intentia sablonului

Iterator este un sablon de proiectare comportamental care permite iterarea prin obiectele unei colectii fara a expune reprezentarea interna a colectiei.

- Problema care o rezolva

Uneori, este necesar sa se acceseze elementele unei colectii, dar nu se doreste expunerea reprezentarii interne a colectiei sau sa se afecteze iterarea prin colectie in cursul utilizarii colectiei. Iterator rezolva aceasta problema prin definirea unei interfete comune pentru iterarea prin elementele colectiei si izolarea codului care face iterarea prin colectie de codul care utilizeaza colectia.

- Componente cu o mica descriere
  1. Iterator: Clasa abstracta care defineste o interfata pentru iterarea prin elementele colectiei.
  2. ConcreteIterator: Clasa care implementeaza Iterator si care contine informatii despre pozitia curenta in colectie si implementeaza metodele definite in Iterator pentru a naviga prin colectie.
  3. Aggregate: Clasa abstracta care defineste o interfata pentru crearea unui iterator.
  4. ConcreteAggregate: Clasa care implementeaza Aggregate si care creeaza un iterator specific colectiei.



Iterator (exemplu de cod)

```
from abc import ABC, abstractmethod


class AbstractIterator(ABC):
    @abstractmethod
    def has_next(self): pass

    @abstractmethod
    def next(self): pass


class FoodItem:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __str__(self):
        return f"{self.name}: {self.price}
$"


class Menu:
    def __init__(self):
        self.items = []

    def add(self, it):
        self.items.append(it)

    def remove(self):
        return self.items.pop()

    def iterator(self):
        return MenuIterator(self.items)
```

```
class MenuIterator(AbstractIterator):
    def __init__(self, items):
        self.index = 0
        self.items = items

    def has_next(self):
        return True if self.index <
len(self.items) else False

    def next(self):
        it = self.items[self.index]
        self.index += 1
        return it


if __name__ == "__main__":
    item1 = FoodItem("Burger", 7)
    item2 = FoodItem("Pizza", 8)
    item3 = FoodItem("Chicken", 10)

    menu = Menu()
    menu.add(item1)
    menu.add(item2)
    menu.add(item3)

    print("-- Displaying Menu --")
    iterator = menu.iterator()

    while iterator.has_next():
        item = iterator.next()
        print(item)

    iterator.next()
```

Mediator

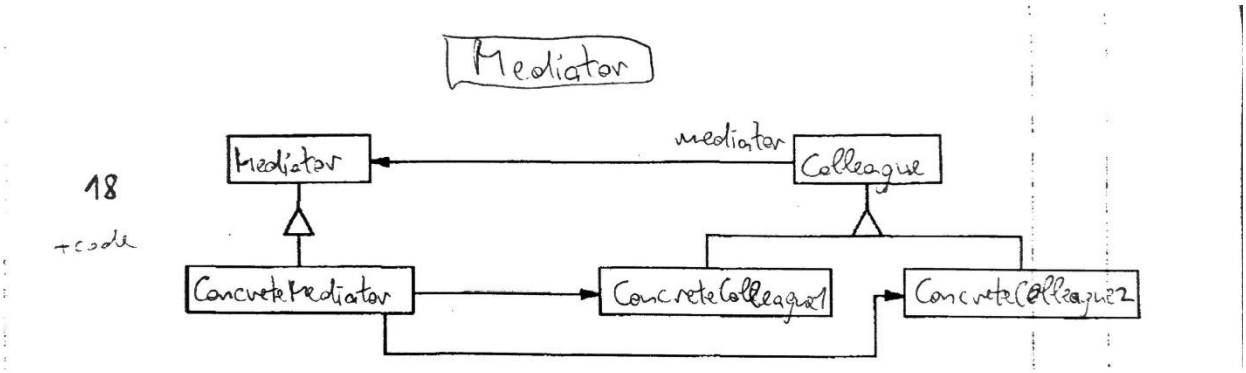
- Intentia sablonului

Mediator este un sablon de proiectare comportamental care permite reducerea cuplajului intre obiecte si promoveaza colectivitatea si modularitatea.

- Problema care o rezolva

Uneori, obiectele trebuie sa comunice intre ele pentru a efectua anumite operatii. Cu toate acestea, aceasta comunicare poate duce la cresterea cuplajului intre obiecte, ceea ce poate face dificila modificarea sistemului in viitor. Mediator rezolva aceasta problema prin definirea unui obiect mediator care gestioneaza comunicarea intre obiectele participante.

- Componente cu o mica descriere
  1. Mediator: Clasa abstracta care defineste interfata pentru comunicarea intre obiecte.
  2. ConcreteMediator: Clasa care implementeaza Mediator si care gestioneaza comunicarea intre obiecte.
  3. Colleague: Clasa abstracta care defineste interfata pentru obiectele care trebuie sa comunice intre ele.
  4. ConcreteColleague: Clasa care implementeaza Colleague si care comunica cu celelalte obiecte prin intermediul Mediatorului.



Mediator (exemplu de cod)

```
from abc import ABC, abstractmethod


class User(ABC):
    def __init__(self, med, name):
        self.mediator = med
        self.name = name

    @abstractmethod
    def send(self, msg):
        pass

    @abstractmethod
    def receive(self, msg):
        pass


class ChatMediator:
    def __init__(self):
        self.users = []

    def add_user(self, user):
        self.users.append(user)

    def send_message(self, msg, user):
        for u in self.users:
            if u != user:
                u.receive(msg)


class ConcreteUser(User):
    def send(self, msg):
        print(self.name + ": [Sending Message] -> " + msg)
        self.mediator.send_message(msg, self)

    def receive(self, msg):
        print(self.name + ": [Received Message] <- " + msg)


if __name__ == '__main__':
    mediator = ChatMediator()

    user1 = ConcreteUser(mediator, "Andrei")
    user2 = ConcreteUser(mediator, "Ana")
    user3 = ConcreteUser(mediator, "Cristi")
    user4 = ConcreteUser(mediator, "Vasile")

    mediator.add_user(user1)
    mediator.add_user(user2)
    mediator.add_user(user3)
    mediator.add_user(user4)

    user1.send("Hello everyone")
```

Memento

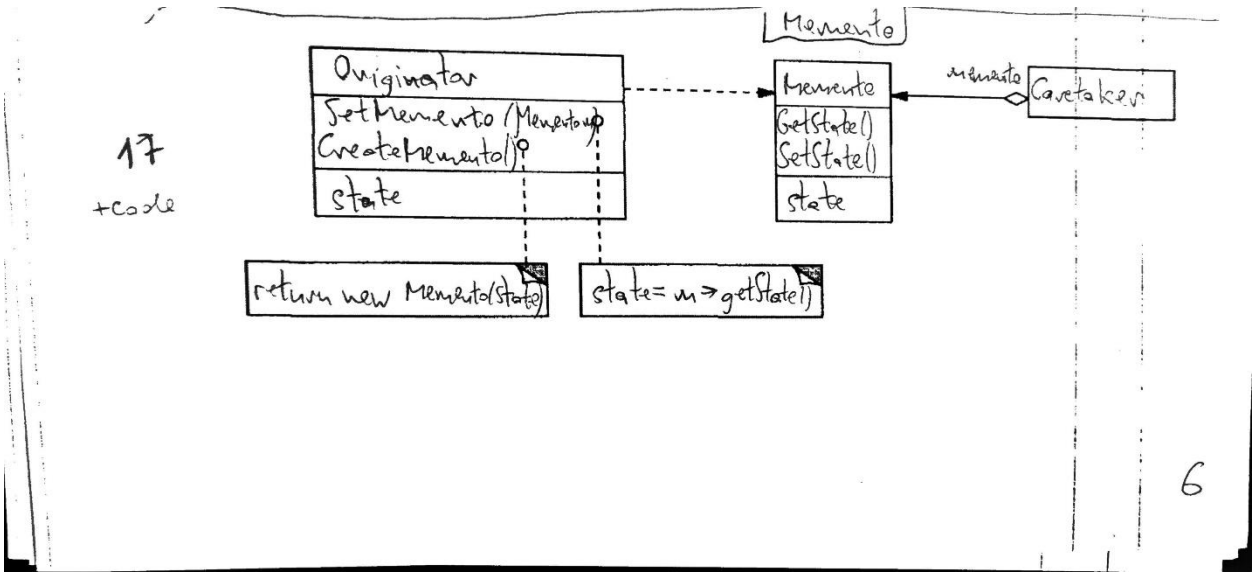
- Intentia sablonului

Memento este un sablon de proiectare comportamental care permite salvarea si restaurarea starii anterioare a unui obiect fara a dezvalui detaliile de implementare ale obiectului.

- Problema care o rezolva

Uneori, trebuie sa salvam starea anterioara a unui obiect pentru a permite restaurarea ulterioara a acesteia. Cu toate acestea, daca obiectul salveaza starea interna in mod public, acest lucru poate duce la o expunere neintentionata a detaliilor de implementare ale obiectului. Memento rezolva aceasta problema prin definirea a trei clase: Originator (obiectul pentru care se salveaza starea), Memento (clasa care contine starea salvata a obiectului) si Caretaker (clasa care gestioneaza starea salvata a obiectului).

- Componente cu o mica descriere
    1. Originator: Clasa care contine starea obiectului si care poate crea un obiect Memento care contine starea salvata a obiectului.
    2. Memento: Clasa care contine starea salvata a obiectului.
    3. Caretaker: Clasa care gestioneaza starea salvata a obiectului si care poate restaura starea obiectului la o anumita stare anterioara.



Memento (exemplu de cod)

```
from abc import ABC, abstractmethod


class Memento(ABC):
    @abstractmethod
    def get_saved_state(self): pass


class ConcreteMemento(Memento, object):
    def __init__(self, state):
        self._state = state

    def get_saved_state(self):
        return self._state


class Originator:
    _state = ""

    def set(self, state):
        self._state = state
        print("Originator: Setting state to", self._state)

    def save_to_memento(self):
        print("Originator: Saving to Memento.")
        return ConcreteMemento(self._state)

    def restore_from_memento(self, memento):
        self._state = memento.get_saved_state()
        print("Originator: State after restoring from Memento:", self._state)


if __name__ == "__main__":
    saved_states = []
    originator = Originator()
    originator.set("State-1")
    originator.set("State-2")
    saved_states.append(originator.save_to_memento())

    originator.set("State-3")
    saved_states.append(originator.save_to_memento())

    originator.set("State-4")

    originator.restore_from_memento(saved_states[0])
```

Observer

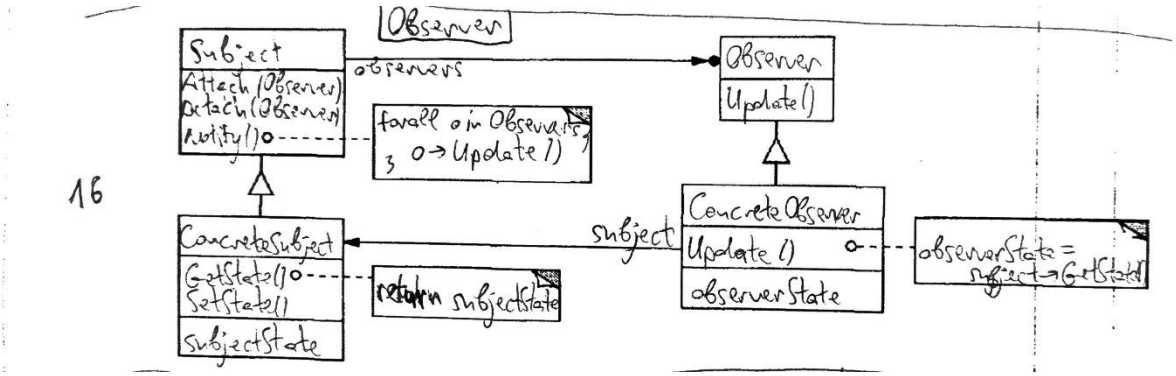
- Intentia sablonului

Observer este un sablon de proiectare comportamental care permite notificarea obiectelor interesate atunci cand se produce o schimbare a starii obiectului observabil.

- Problema care o rezolva

Uneori, mai multe obiecte trebuie sa reactioneze la schimbarile starii unui obiect. In acest caz, solutia ar fi sa se defineasca metode de notificare pentru fiecare obiect interesat, dar acest lucru poate duce la cresterea cuplajului intre obiecte si poate face dificila adaugarea sau eliminarea obiectelor care reactioneaza la schimbarile starii obiectului observabil. Observer rezolva aceasta problema prin permiterea notificarii automatice a obiectelor interesate atunci cand se produce o schimbare a starii obiectului observabil.

- Componente cu o mica descriere
  1. Subject: Clasa abstracta care defineste interfata pentru obiectul observabil.
  2. ConcreteSubject: Clasa care implementeaza Subject si care notifica obiectele interesate atunci cand se produce o schimbare a starii sale.
  3. Observer: Clasa abstracta care defineste interfata pentru obiectele interesate de schimbarile starii obiectului observabil.
  4. ConcreteObserver: Clasa care implementeaza Observer si care reactioneaza la schimbarile starii obiectului observabil.



Observer (exemplu de cod)

```
from abc import ABC, abstractmethod
from typing import List

# Clasa Observer abstracta care specifica metoda de update
class Observer(ABC):

    @abstractmethod
    def update(self, message: str):
        pass


# Clasa Subject care gestioneaza lista de obiecte Observator si notifica aceste obiecte de
schimbari
class Subiect:

    def __init__(self):
        self.observatori: List[Observer] = []

    def adauga_observator(self, observator: Observer):
        self.observatori.append(observator)

    def elimina_observator(self, observator: Observer):
        self.observatori.remove(observator)

    def notifica_observatori(self, message: str):
        for observator in self.observatori:
            observator.update(message)


# Clasa ConcretObservator care implementeaza metoda update
class Abonat(Observer):

    def __init__(self, nume: str):
        self.nume = nume

    def update(self, message: str):
        print(f"{self.nume} a primit urmatorul mesaj: {message}")


# Exemplu de utilizare a sablonului Observer pentru a notifica abonatii cand se intampla un
eveniment
if __name__ == '__main__':
    subiect = Subiect()
    abonat1 = Abonat("Ion")
    abonat2 = Abonat("Maria")
    abonat3 = Abonat("Alex")

    subiect.adauga_observator(abonat1)
    subiect.adauga_observator(abonat2)
    subiect.adauga_observator(abonat3)

    subiect.notifica_observatori("A avut loc o schimbare!")

    subiect.elimina_observator(abonat2)

    subiect.notifica_observatori("O alta schimbare a avut loc!")
```

State

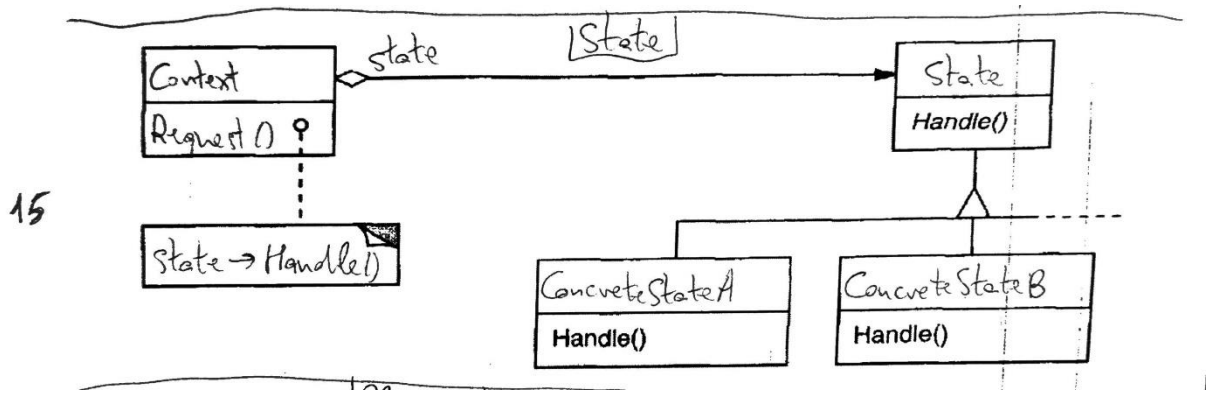
- Intentia sablonului

State este un sablon de proiectare comportamental care permite unui obiect sa isi modifice comportamentul in functie de starea sa interna.

- Problema care o rezolva

Uneori, comportamentul unui obiect trebuie sa se schimbe in functie de starea sa interna. Daca se utilizeaza un set de instructiuni if-else pentru a schimba comportamentul obiectului in functie de starea sa interna, atunci acest lucru poate duce la cod greu de intretinut si dificil de extins. State rezolva aceasta problema prin definirea unei clase abstracte State si a claselor ConcreteState care reprezinta stari diferite ale obiectului si permit schimbarea comportamentului obiectului in functie de starea sa interna.

- Componente cu o mica descriere
  1. Context: Clasa care contine starea interna a obiectului si care foloseste obiectele ConcreteState pentru a schimba comportamentul obiectului.
  2. State: Clasa abstracta care defineste interfata pentru starea obiectului.
  3. ConcreteState: Clasa care implementeaza State si care reprezinta o stare diferita a obiectului.



State (exemplu de cod)

```
from abc import ABC, abstractmethod

# Starea abstracta
class WashingState(ABC):
    @abstractmethod
    def handle(self):
        pass

# Starea Concrete: Oprire
class Off(WashingState):
    def handle(self):
        print("Masina de spalat este
oprita.")

# Starea Concrete: Pornit
class On(WashingState):
    def handle(self):
        print("Masina de spalat este
pornita.")

# Starea Concrete: Clatire
class Rinsing(WashingState):
    def handle(self):
        print("Masina de spalat clateste
rufele.")

# Starea Concrete: Stoarcere
class Spinning(WashingState):
    def handle(self):
        print("Masina de spalat stoarce
rufele.")

# Starea Concrete: Terminat
class Done(WashingState):
    def handle(self):
        print("Spalarea a fost
finalizata.")
```

```
# Contextul: Masina de spalat
class WashingMachine:
    def __init__(self):
        # Initial starea este "oprit"
        self.state = Off()

    # Metoda pentru schimbarea starii
    def set_state(self, state):
        self.state = state

    # Metoda pentru manevrarea masinii de
spalat
    def start_washing(self):
        self.state.handle()
        self.set_state(On())

    def rinse(self):
        self.state.handle()
        self.set_state(Rinsing())

    def spin(self):
        self.state.handle()
        self.set_state(Spinning())

    def done(self):
        self.state.handle()
        self.set_state(Done())

# Utilizarea contextului
washing_machine = WashingMachine()
washing_machine.start_washing()
washing_machine.rinse()
washing_machine.spin()
washing_machine.done()
```

Strategy

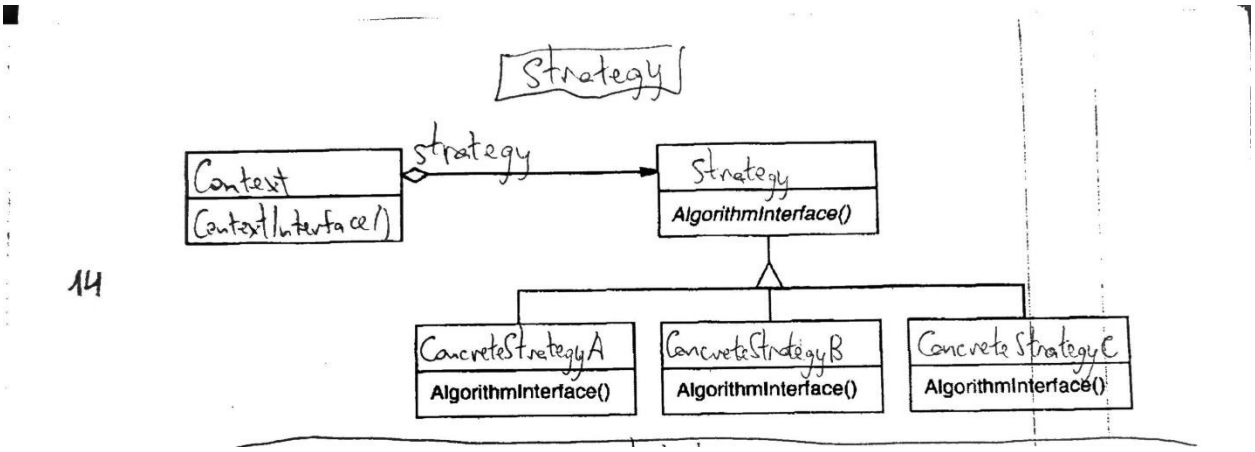
- Intentia sablonului

Strategy este un sablon de proiectare comportamental care permite schimbarea comportamentului unui obiect in timpul executiei.

- Problema care o rezolva

Uneori, comportamentul unui obiect trebuie sa se schimbe in functie de contextul in care este utilizat. Daca se utilizeaza un set de instructiuni if-else pentru a schimba comportamentul obiectului in functie de context, atunci acest lucru poate duce la cod greu de intretinut si dificil de extins. Strategy rezolva aceasta problema prin definirea unei clase abstracte Strategy si a claselor ConcreteStrategy care reprezinta strategii diferite ale obiectului si permit schimbarea comportamentului obiectului in timpul executiei.

- Componente cu o mica descriere
  1. Context: Clasa care contine obiectul care utilizeaza strategia si care poate schimba strategia in timpul executiei.
  2. Strategy: Clasa abstracta care defineste interfata pentru strategiile obiectului.
  3. ConcreteStrategy: Clasa care implementeaza Strategy si care reprezinta o strategie diferita a obiectului.



Strategy (exemplu de cod)

```
from abc import ABC, abstractmethod

# Clasa abstracta Strategie care defineste metoda de plata
class StrategiePlata(ABC):

    @abstractmethod
    def plateste(self, suma: float) -> str:
        pass

# Clasa ConcretStrategie care implementeaza o metoda de plata specifica
class PlataCard(StrategiePlata):

    def plateste(self, suma: float) -> str:
        return f"Plata cu cardul a fost acceptata pentru suma de {suma} RON."

# Clasa ConcretStrategie care implementeaza o metoda de plata specifica
class PlataPayPal(StrategiePlata):

    def plateste(self, suma: float) -> str:
        return f"Plata cu PayPal a fost acceptata pentru suma de {suma} RON."

# Clasa ConcretStrategie care implementeaza o metoda de plata specifica
class PlataBitcoin(StrategiePlata):

    def plateste(self, suma: float) -> str:
        return f"Plata cu Bitcoin a fost acceptata pentru suma de {suma} RON."

# Clasa Context care utilizeaza o strategie pentru a efectua o plata
class ContextPlata:

    def __init__(self, strategie: StrategiePlata):
        self.strategie = strategie

    def schimba_strategie(self, strategie: StrategiePlata):
        self.strategie = strategie

    def efectueaza_plata(self, suma: float) -> str:
        return self.strategie.plateste(suma)


# Exemplu de utilizare a sablonului Strategy pentru a efectua diferite tipuri de plati
if __name__ == '__main__':
    plata_card = PlataCard()
    plata_paypal = PlataPayPal()
    plata_bitcoin = PlataBitcoin()

    context_plata = ContextPlata(plata_card)
    print(context_plata.efectueaza_plata(100.0))

    context_plata.schimba_strategie(plata_paypal)
    print(context_plata.efectueaza_plata(150.0))

    context_plata.schimba_strategie(plata_bitcoin)
    print(context_plata.efectueaza_plata(200.0))
```

Template Method

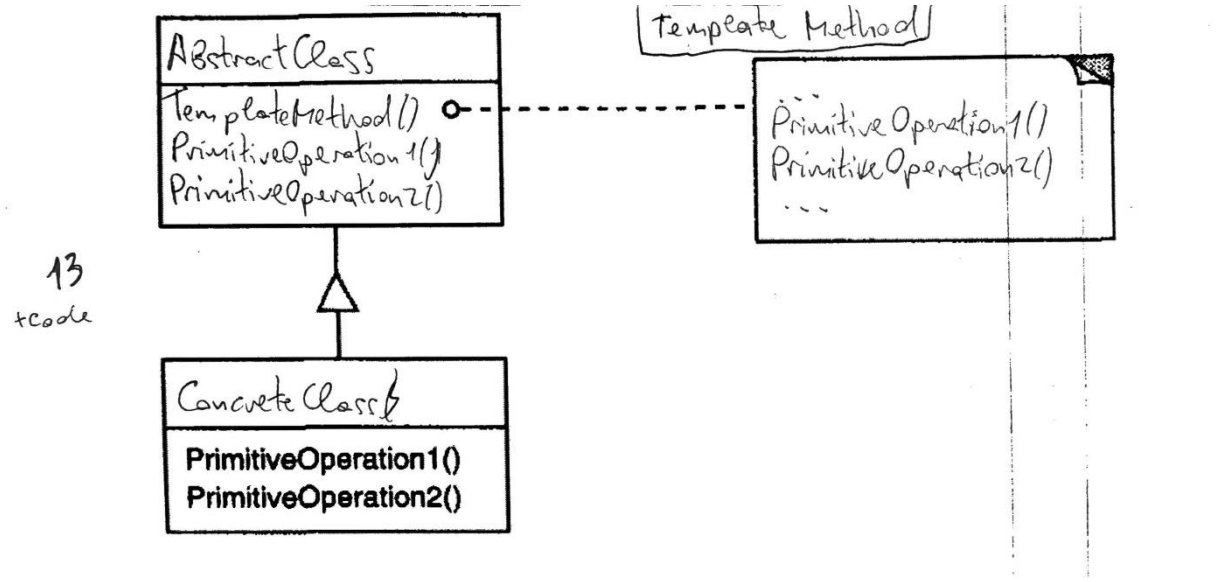
- Intentia sablonului

Template Method este un sablon de proiectare comportamental care permite definirea scheletului unui algoritm, lasand unele dintre pasii specifici implementarii subclaselor.

- Problema care o rezolva

Uneori, avem un algoritm general care trebuie executat, dar anumiti pasi din algoritm pot varia in functie de subclasele care il implementeaza. Daca incercam sa implementam algoritmul intr-o singura clasa, acesta poate deveni greu de extins si de intretinut. Template Method rezolva aceasta problema prin definirea unei clase abstracte care contine scheletul algoritmului, iar subclasele pot implementa pasii specifici care variaza.

- Componente cu o mica descriere
  1. AbstractClass: Clasa abstracta care contine scheletul algoritmului si care defineste metodele abstracte care vor fi implementate de subclase.
  2. ConcreteClass: Clasa care mosteneste AbstractClass si care implementeaza metodele abstracte specifice acelei subclase.



Template Method (exemplu de cod)

```
from abc import ABC, abstractmethod


class ResearchGuideline(ABC):
    def template_method(self):
        self.request_letter()
        self.complete_information()
        self.apply()
        self.get_documents()

    def request_letter(self):
        pass

    def complete_information(self):
        pass

    @abstractmethod
    def apply(self):
        pass

    def get_documents(self):
        pass


class TechnicalUniversityOfMoldova(ResearchGuideline):
    def complete_information(self):
        print(" - Completing personal information for Technical University of Moldova")

    def apply(self):
        print(" - Applying for Technical University of Moldova")


class StateUniversityOfMoldova(ResearchGuideline):
    def request_letter(self):
        print(" - Requesting a letter for State University of Moldova")

    def apply(self):
        print(" - Applying for State University of Moldova")

    def get_documents(self):
        print(" - Getting documents for State University of Moldova")


def client_call(research_guideline: ResearchGuideline):
    research_guideline.template_method()


if __name__ == '__main__':
    print("\nTechnical University of Moldova:")
    client_call(TechnicalUniversityOfMoldova())

    print("\nState University of Moldova:")
    client_call(StateUniversityOfMoldova())
```

Visitor

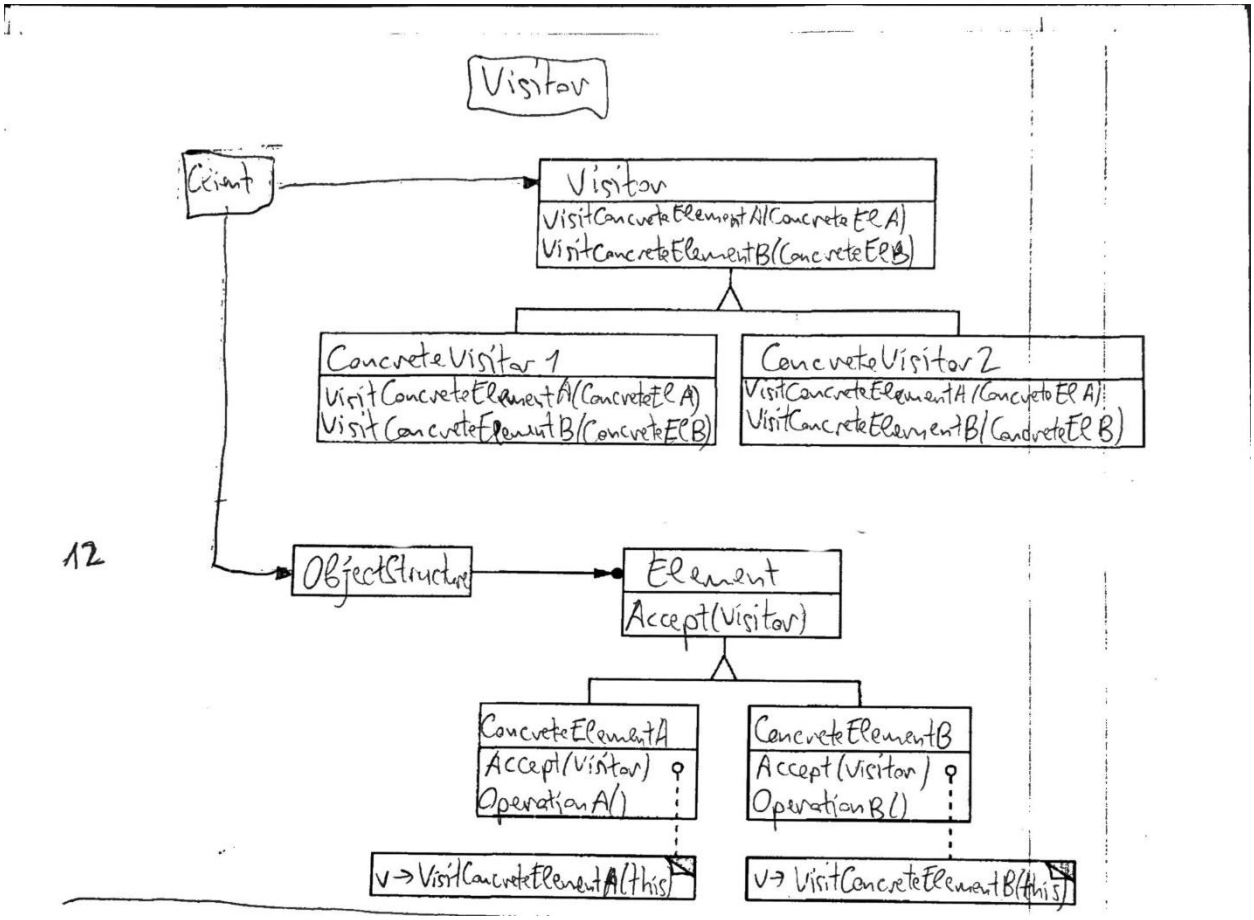
- Intentia sablonului

Visitor este un sablon de proiectare comportamental care permite separarea algoritmului de operarea pe obiecte.

- Problema care o rezolva

Uneori, trebuie sa aplicam un algoritm pe un set de obiecte de diferite tipuri. Daca incercam sa implementam algoritmul in fiecare clasa de obiecte, atunci acesta poate deveni greu de intretinut si extins. Visitor rezolva aceasta problema prin definirea a doua clase: Visitor si ConcreteVisitor. Clasa Visitor contine metodele care vor fi utilizate pentru a aplica algoritmul pe obiecte, iar clasa ConcreteVisitor contine implementarea algoritmului pentru fiecare tip de obiect.

- Componente cu o mica descriere
  1. Visitor: Clasa abstracta care defineste metodele care vor fi utilizate pentru a aplica algoritmul pe obiecte.
  2. ConcreteVisitor: Clasa care implementeaza Visitor si care contine implementarea algoritmului pentru fiecare tip de obiect.
  3. Element: Clasa abstracta care defineste metodele accept() care accepta un visitor.
  4. ConcreteElement: Clasa care implementeaza Element si care permite unui visitor sa acceseze starea obiectului.



Visitor (exemplu de cod)

```
from abc import ABC, abstractmethod

# Vizitatorul
class GameVisitor(ABC):
    @abstractmethod
    def visit_enemy(self, enemy):
        pass

    @abstractmethod
    def visit_obstacle(self, obstacle):
        pass

# Obiectul vizitat 1
class Enemy(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass

    @abstractmethod
    def attack(self):
        pass

# Obiectul vizitat 2
class Obstacle(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass

    @abstractmethod
    def move(self):
        pass

# Implementarea obiectului vizitat 1
class Goblin(Enemy):
    def accept(self, visitor):
        visitor.visit_enemy(self)

    def attack(self):
        print("Goblin attacks the player!")

# Implementarea obiectului vizitat 2
class Wall(Obstacle):
    def accept(self, visitor):
        visitor.visit_obstacle(self)

    def move(self):
        print("Wall cannot move.")

# Implementarea vizitatorului
class CombatVisitor(GameVisitor):
    def visit_enemy(self, enemy):
        print("Player attacks the goblin!")
        enemy.attack()

    def visit_obstacle(self, obstacle):
        print("Player tries to move through
the wall.")
        obstacle.move()

# Clientul
if __name__ == '__main__':
    enemies = [Goblin() for _ in range(5)]
    obstacles = [Wall() for _ in range(3)]

    visitor = CombatVisitor()

    for enemy in enemies:
        enemy.accept(visitor)

    for obstacle in obstacles:
        obstacle.accept(visitor)
```