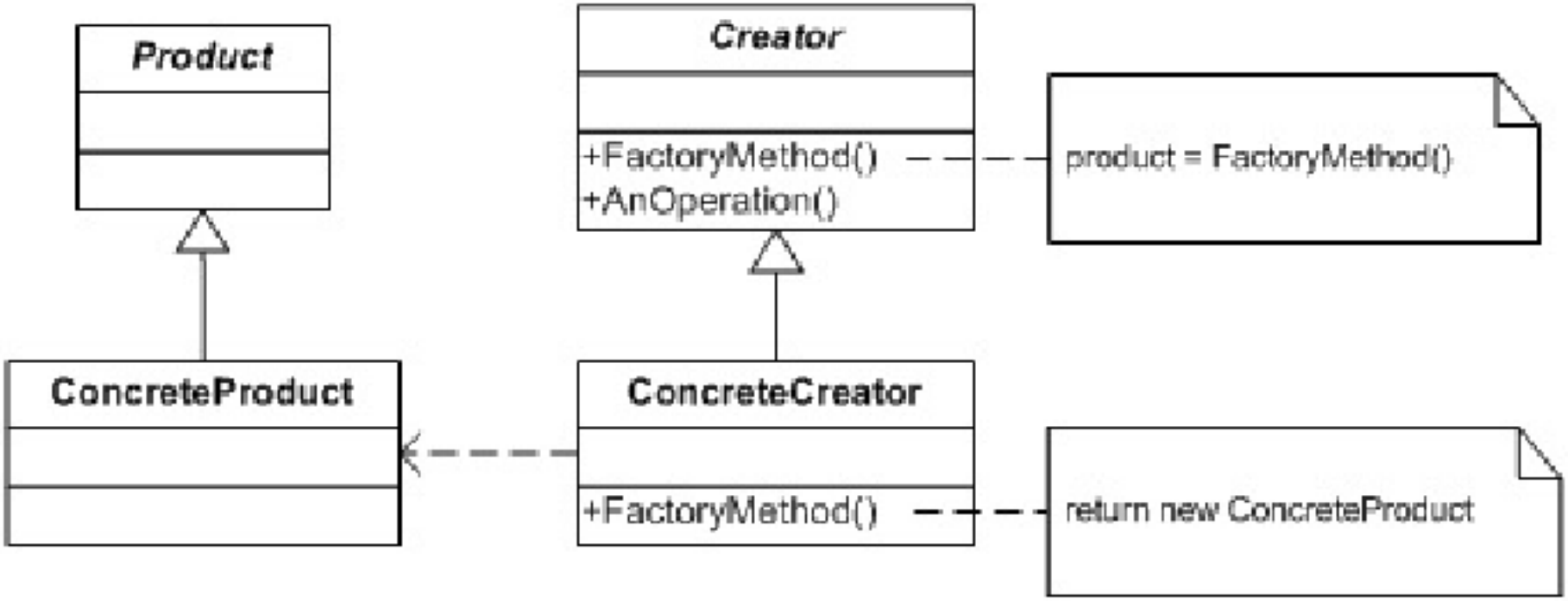
Paradigma Modelelor de Proiectare

Cursul nr. 9

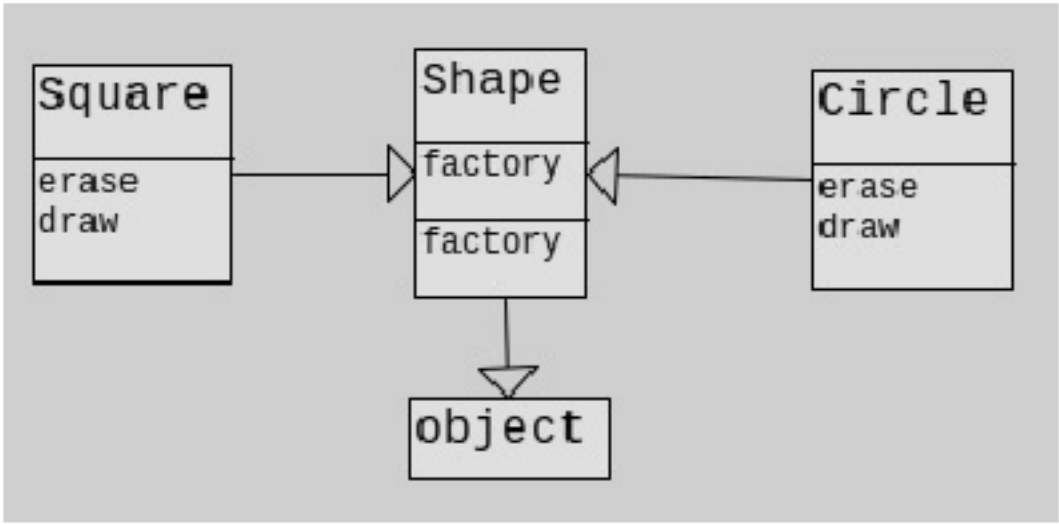
Mihai Zaharia

Modelul Fabrică de obiecte

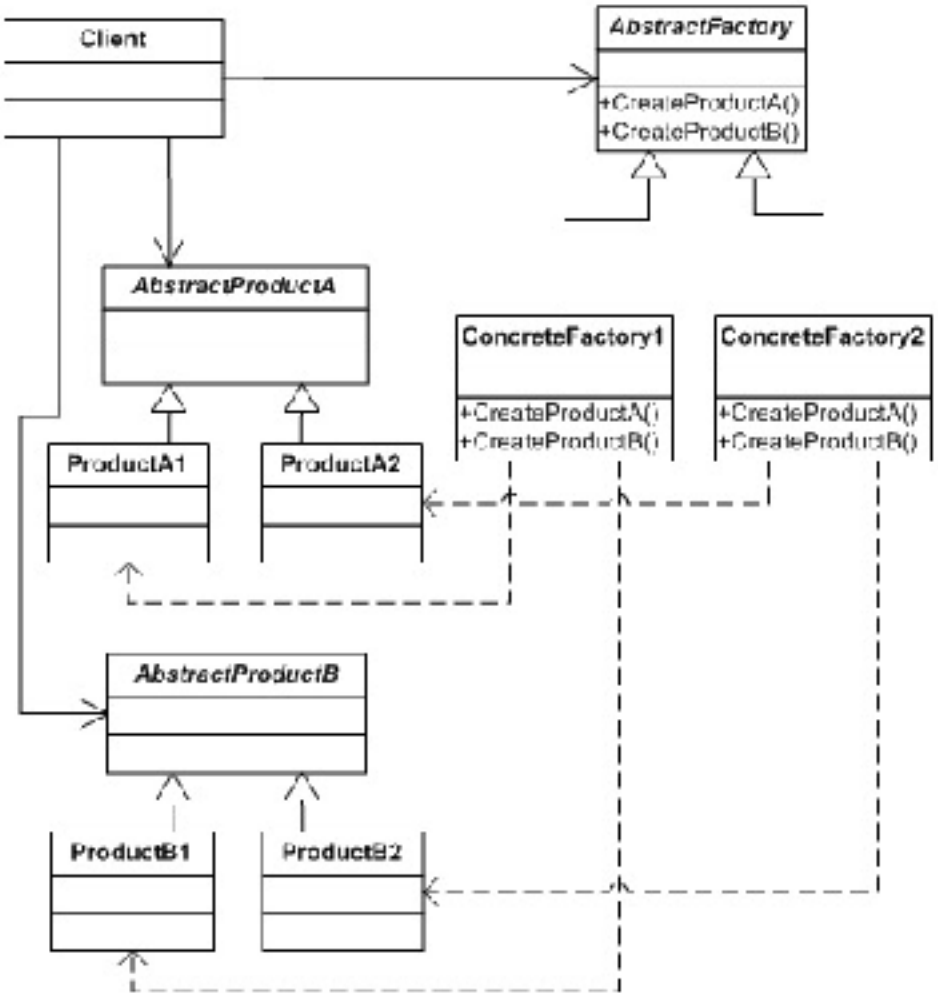


Modelul Fabrică de obiecte - caz de utilizare

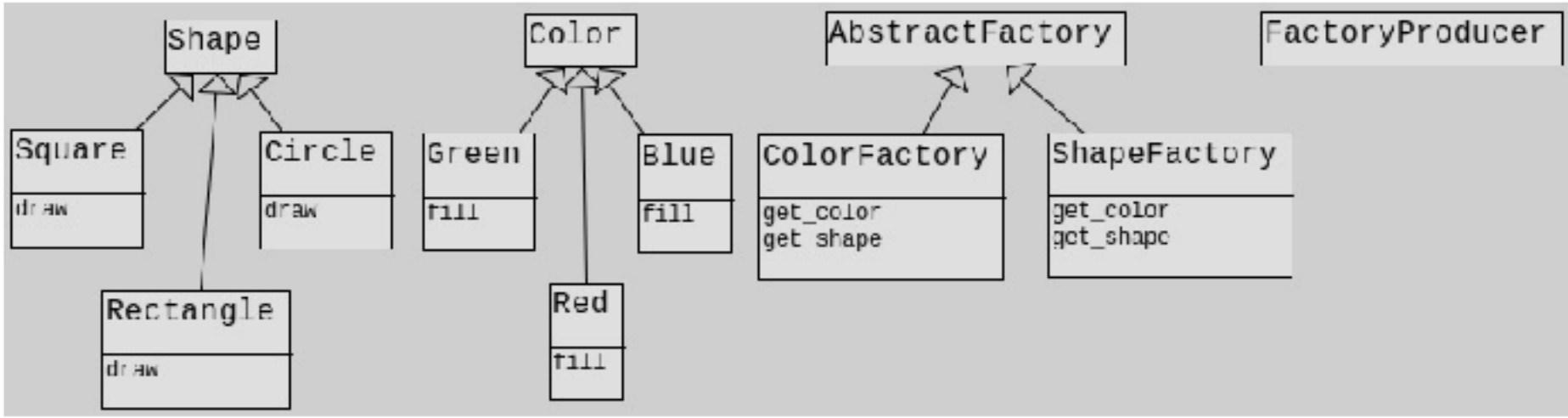
```
class Shape(object):
    def factory(type):  #return eval(type + "()")
        if type == "Circle": return Circle()
        if type == "Square": return Square()
        assert 0, "Bad shape creation: " + type
    factory = staticmethod(factory)
class Circle(Shape):
    def draw(self): print("Circle.draw")
    def erase(self): print("Circle.erase")
class Square(Shape):
    def draw(self): print("Square.draw")
    def erase(self): print("Square.erase")
# se genereaza numele formelor
def shapeNameGen(n):
    types = Shape.__subclasses__()
    for i in range(n):
        yield random.choice(types).__name__
shapes = [ Shape.factory(i) for i in shapeNameGen(7)]
for shape in shapes:
    shape.draw()
    shape.erase()
```



Model Fabrica abstractă



Modelul Fabrică de obiecte - clasic

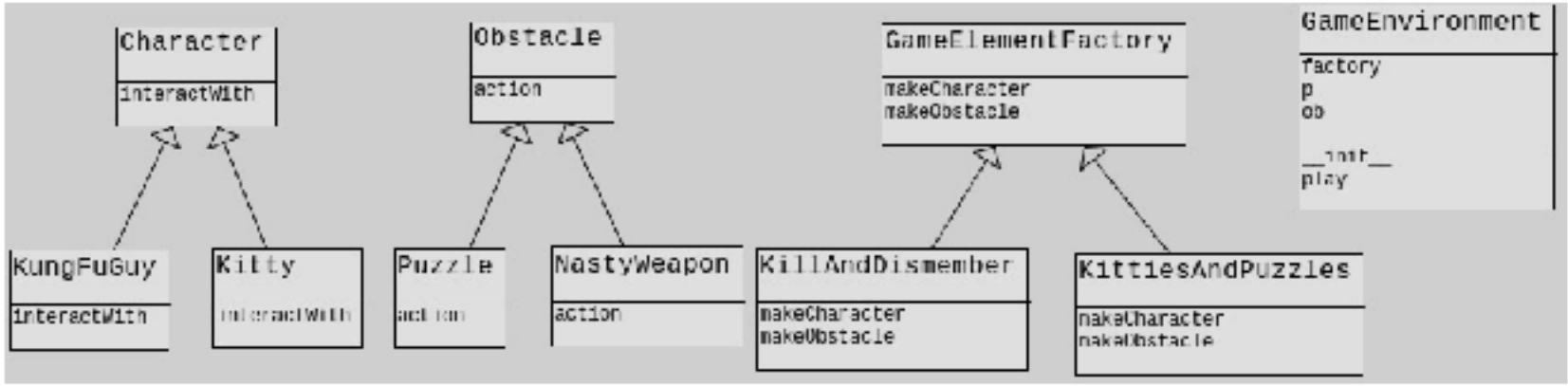


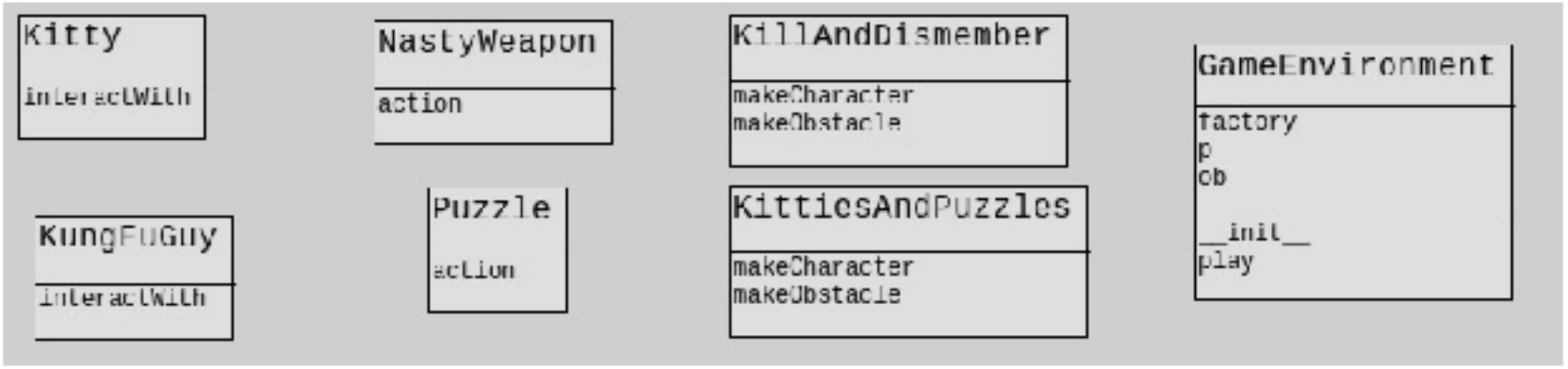
Și implementarea

```
import abc
class Shape(metaclass=abc.ABCMeta): #interfata pentru forme
    @abc.abstractmethod
    def draw(self):
        pass
class Color(metaclass=abc.ABCMeta):#interfata pentru culori
    @abc.abstractmethod
    def fill(self):
        pass
class AbstractFactory(metaclass=abc.ABCMeta):#creare clasa abstracta pt obtinere obj
    @abc.abstractmethod
    def get_color(self):
        pass
    @abc.abstractmethod
    def get_shape(self):
        pass
class Rectangle(Shape):
    def draw(self):
        print("Inside Rectangle::draw() method.")
class Square(Shape):
    def draw(self):
        print("Inside Square::draw() method.")
class Circle(Shape):
    def draw(self):
        print("Inside Circle::draw() method.")
class Red(Color):
    def fill(self):
        print("Inside Red::fill() method.")
class Green(Color):
    def fill(self):
        print("Inside Green::fill() method.")
class Blue(Color):
    def fill(self):
        print("Inside Blue::fill() method.")
```

```
# crearea generator fabrici
class FactoryProducer:
    @staticmethod
    def get_factory(choice):
        if choice == "SHAPE":
            return ShapeFactory()
        elif choice == "COLOR":
            return ColorFactory()
        return None
if __name__ == '__main__':
    shape_factory = FactoryProducer.get_factory("SHAPE")
    shape1 = shape_factory.get_shape("CIRCLE");
    shape1.draw()
    shape2 = shape_factory.get_shape("RECTANGLE");
    shape2.draw()
    shape3 = shape_factory.get_shape("SQUARE");
    shape3.draw()
    color_factory = FactoryProducer.get_factory("COLOR");
    color1 = color_factory.get_color("RED");
    color1.fill()
    color2 = color_factory.get_color("GREEN");
    color2.fill()
    color3 = color_factory.get_color("BLUE");
    color3.fill()
```

Să reanalizăm un pic fabrica de fabrici





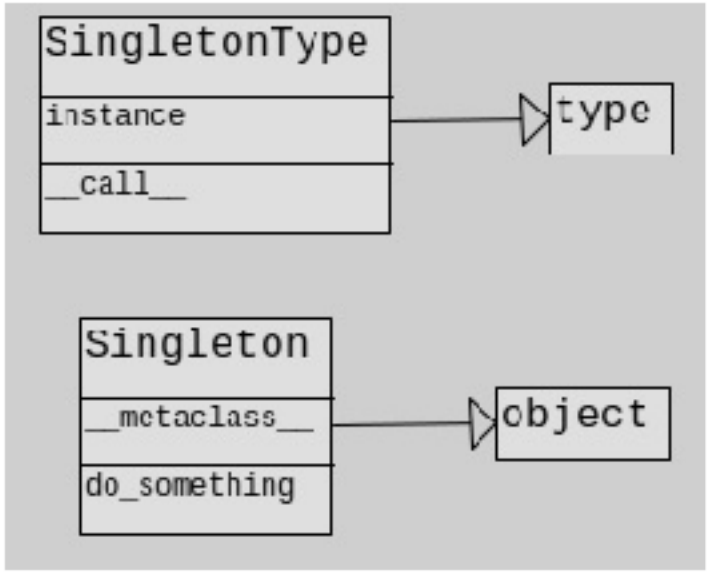
Modelul Fabrica abstractă - OOP vs Structurat

```
class Obstacle : #versiunea generala cf oop
    def action(self) : pass
class Character:
    def interactWith(self, obstacle): pass
class Kitty(Character) :
    def interactWith(self, obstacle):
        print("Kitty has encountered a",
        obstacle.action())
class KungFuGuy(Character) :
    def interactWith(self, obstacle):
        print("KungFuGuy now battles a",
        obstacle.action())
class Puzzle(Obstacle) :
    def action(self) :
        print("Puzzle")
class NastyWeapon(Obstacle):
    def action(self) :
        print("NastyWeapon")
class GameElementFactory:# fabrica abstracta
    def makeCharacter(self): pass
    def makeObstacle(self): pass
class KittiesAndPuzzles(GameElementFactory):# fabrici concrete
    def makeCharacter(self): return Kitty()
    def makeObstacle(self): return Puzzle()
class KillAndDismember(GameElementFactory) :
    def makeCharacter(self): return KungFuGuy()
    def makeObstacle(self): return NastyWeapon()
class GameEnvironment:
    def __init__(self, factory):
        self.factory = factory
        self.p = factory.makeCharacter()
        self.ob = factory.makeObstacle()
    def play(self) :
        self.p.interactWith(self.ob)
g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
```

```
class Kitty:# versiunea front end dist-ului
    def interactWith(self, obstacle):
        print("Kitty has encountered a",
        obstacle.action())
class KungFuGuy:
    def interactWith(self, obstacle):
        print("KungFuGuy now battles a",
        obstacle.action())
class Puzzle:
    def action(self): print("Puzzle")
class NastyWeapon:
    def action(self): print("NastyWeapon")
class KittiesAndPuzzles: #fabrici concrete
    def makeCharacter(self): return Kitty()
    def makeObstacle(self): return Puzzle()
class KillAndDismember:
    def makeCharacter(self): return KungFuGuy()
    def makeObstacle(self): return NastyWeapon()
class GameEnvironment:
    def __init__(self, factory):
        self.factory = factory
        self.p = factory.makeCharacter()
        self.ob = factory.makeObstacle()
    def play(self):
        self.p.interactWith(self.ob)
g1 = GameEnvironment(KittiesAndPuzzles())
g2 = GameEnvironment(KillAndDismember())
g1.play()
g2.play()
```

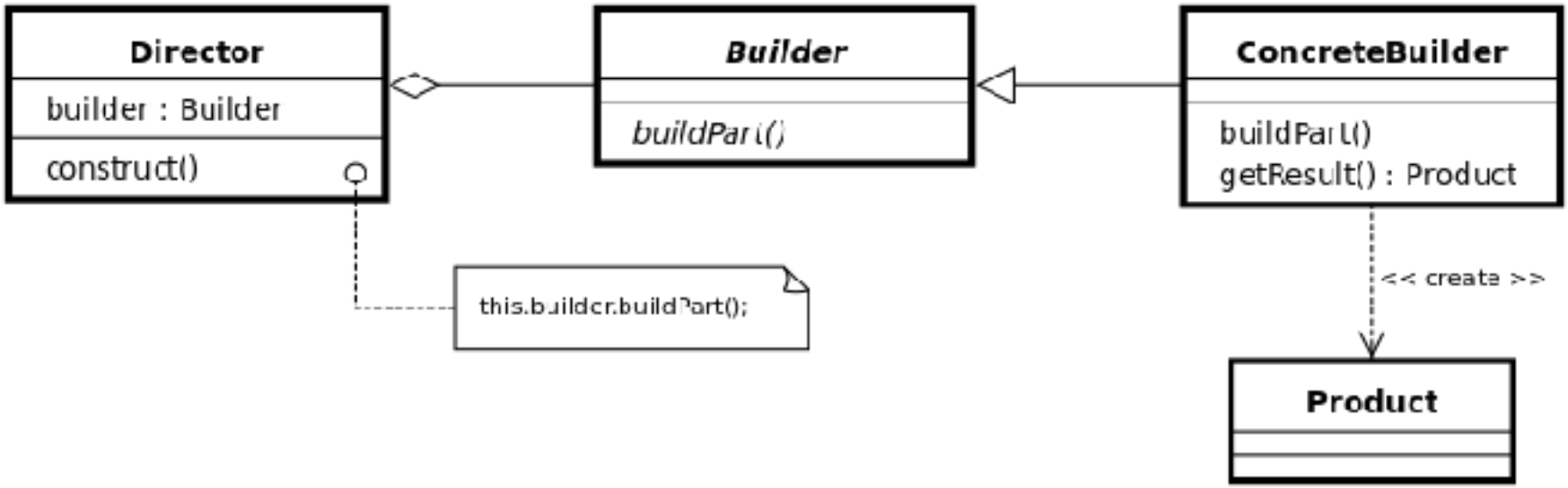
Modelul burlacului

```
class SingletonType(type):
    instance = None

    def __call__(cls, *args, **kw):
        if not cls.instance:
            cls.instance = super(SingletonType, cls).__call__(*args, **kw)
        return cls.instance

class Singleton(object):
    __metaclass__ = SingletonType

    def do_something(self):
        print('Singleton')

s = Singleton()
s.do_something()
```

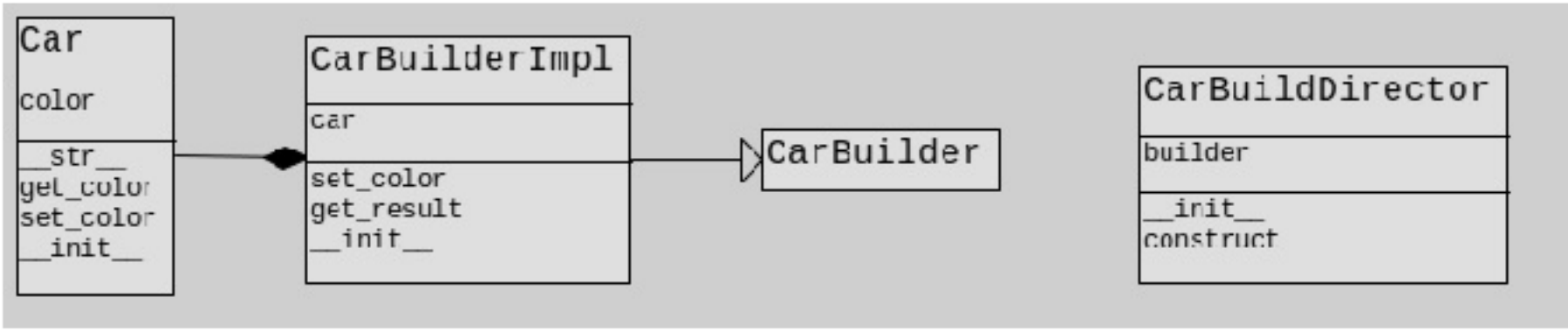


Modelul constructor

Modelul constructor



Modelul constructor - caz de utilizare

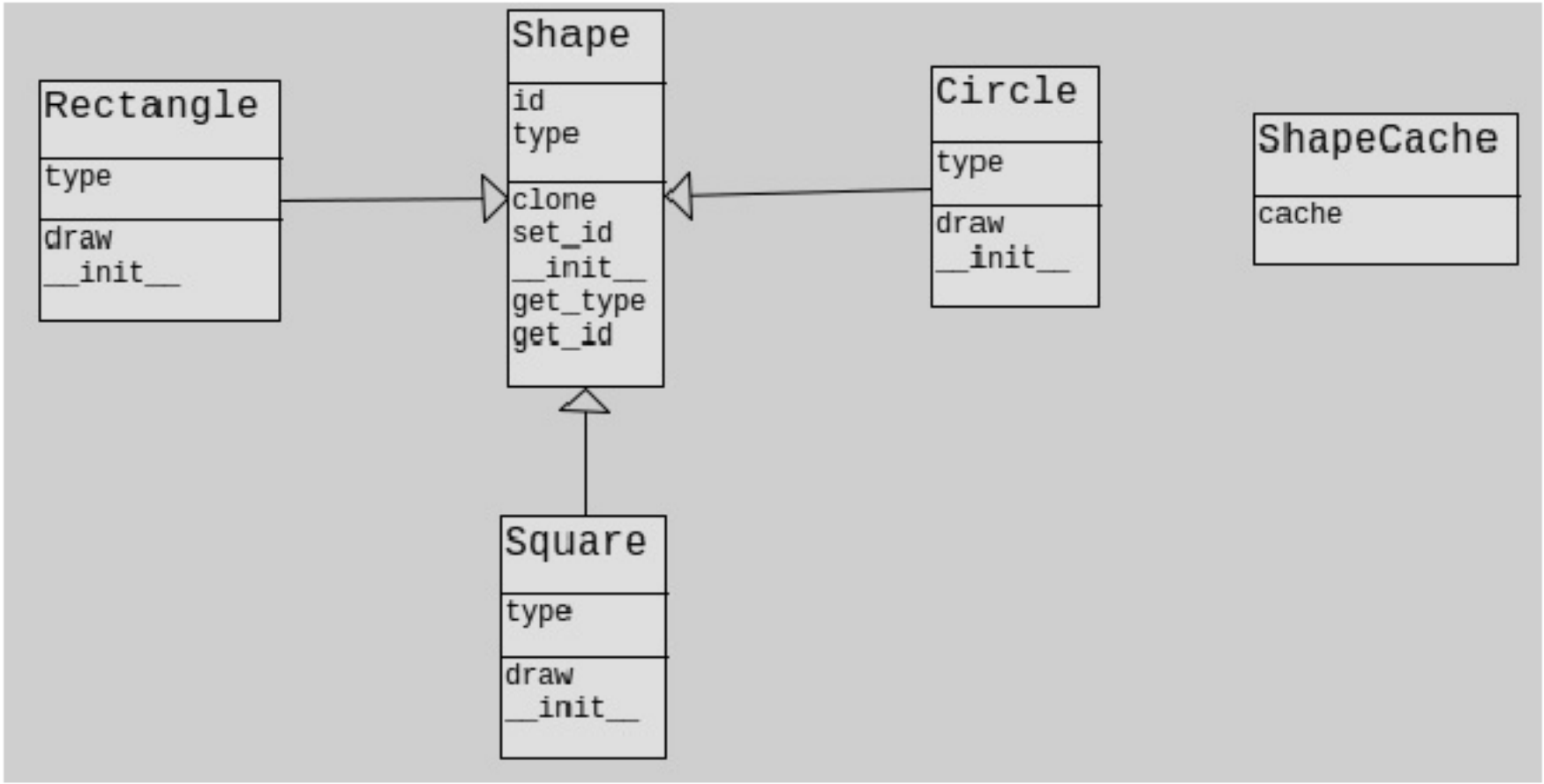


Model constructor - implementare concretă

```
import abc
class Car:#produsul creat de Creator
    def __init__(self):
        self.color = None
    def get_color(self):
        return self.color
    def set_color(self, color):
        self.color = color
    def __str__(self):
        return "Car [color={0}]".format(self.color)
class CarBuilder(metaclass=abc.ABCMeta): #abstractia Creator
    @ abc.abstractmethod
    def set_color(self, color):
        pass
    @ abc.abstractmethod
    def get_result(self):
        pass
class CarBuilderImpl(CarBuilder):
    def __init__(self):
        self.car = Car()
    def set_color(self, color):
        self.car.set_color(color)
    def get_result(self):
        return self.car
```

```
class CarBuildDirector:
    def __init__(self, builder):
        self.builder = builder

    def construct(self):
        self.builder.set_color("Red");
        return self.builder.get_result()

if __name__ == '__main__':
    builder = CarBuilderImpl()
    carBuildDirector = CarBuildDirector(builder)
    print(carBuildDirector.construct())
```

Modelul prototip

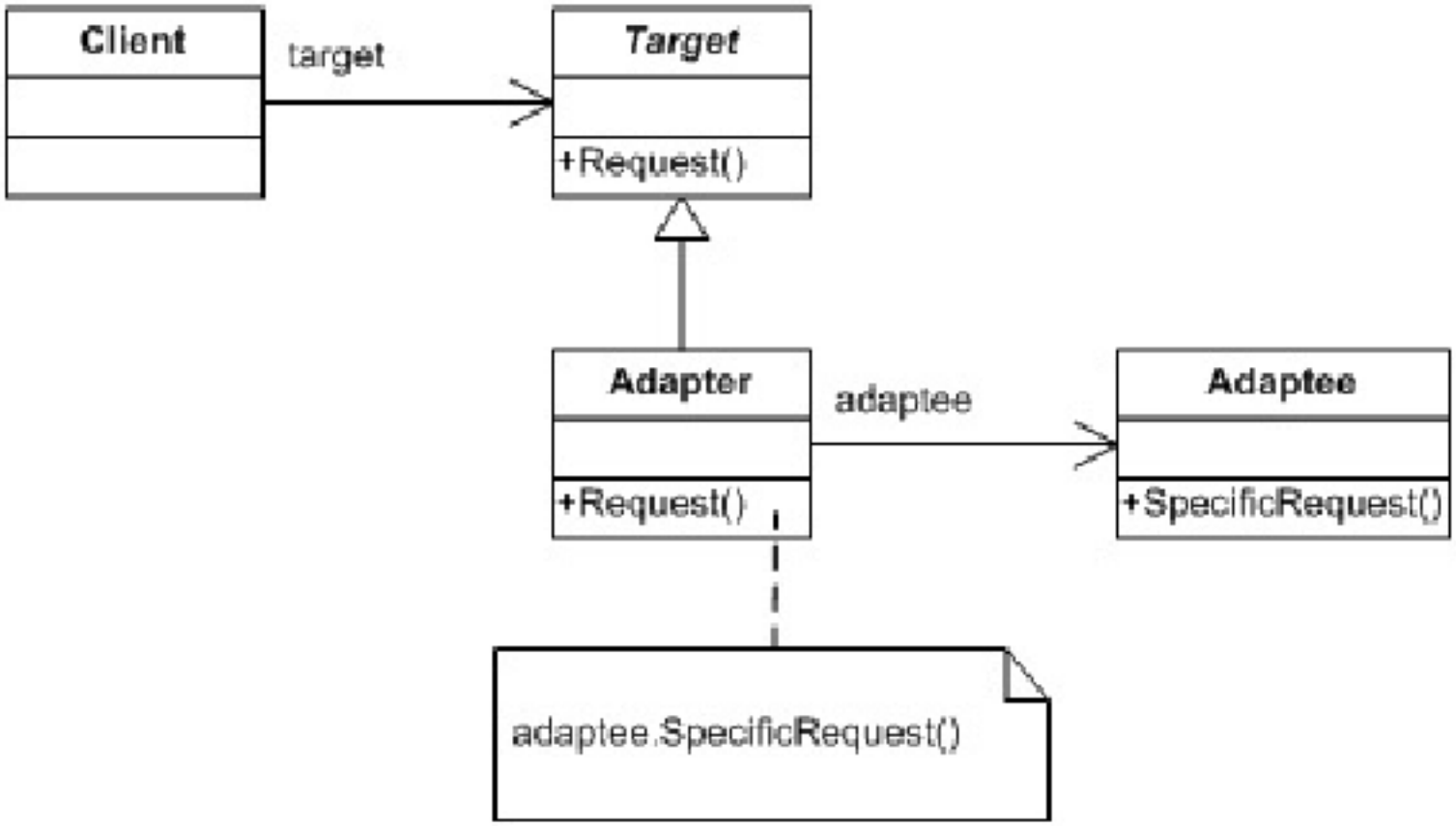


Model protip - implementare de caz

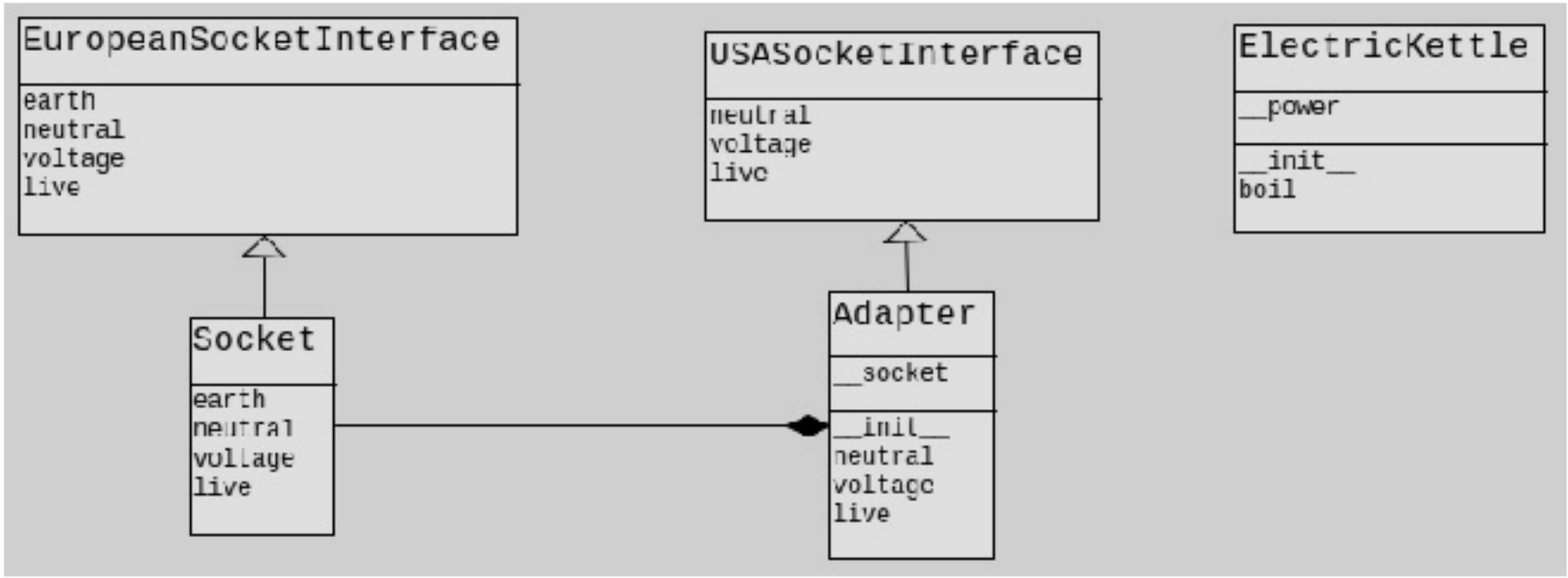
```
import abc
import copy
class Shape(metaclass=abc.ABCMeta):
    def __init__(self):
        self.id = None
        self.type = None
    @abc.abstractmethod
    def draw(self):
        pass
    def get_type(self):
        return self.type
    def get_id(self):
        return self.id
    def set_id(self, sid):
        self.id = sid
    def clone(self):
        return copy.copy(self)
class Rectangle(Shape):
    def __init__(self):
        super().__init__()
        self.type = "Rectangle"
    def draw(self):
        print("Inside Rectangle::draw() method.")
class Square(Shape):
    def __init__(self):
        super().__init__()
        self.type = "Square"
    def draw(self):
        print("Inside Square::draw() method.")
class Circle(Shape):
    def __init__(self):
        super().__init__()
        self.type = "Circle"
    def draw(self):
        print("Inside Circle::draw() method.")
class ShapeCache:
    cache = {}
    @staticmethod
    def get_shape(sid):
        shape = ShapeCache.cache.get(sid, None)
        return shape.clone()
    @staticmethod
    def load():
        circle = Circle()
        circle.set_id("1")
        ShapeCache.cache[circle.get_id()] = circle
        square = Square()
        square.set_id("2")
        ShapeCache.cache[square.get_id()] = square
        rectangle = Rectangle()
        rectangle.set_id("3")
        ShapeCache.cache[rectangle.get_id()] = rectangle
if __name__ == '__main__':
    ShapeCache.load()
    circle = ShapeCache.get_shape("1")
    print(circle.get_type())
    square = ShapeCache.get_shape("2")
    print(square.get_type())
    rectangle = ShapeCache.get_shape("3")
    print(rectangle.get_type())
```

Modele structurale

Modelul Adaptor



Model Adaptor - caz de utilizare

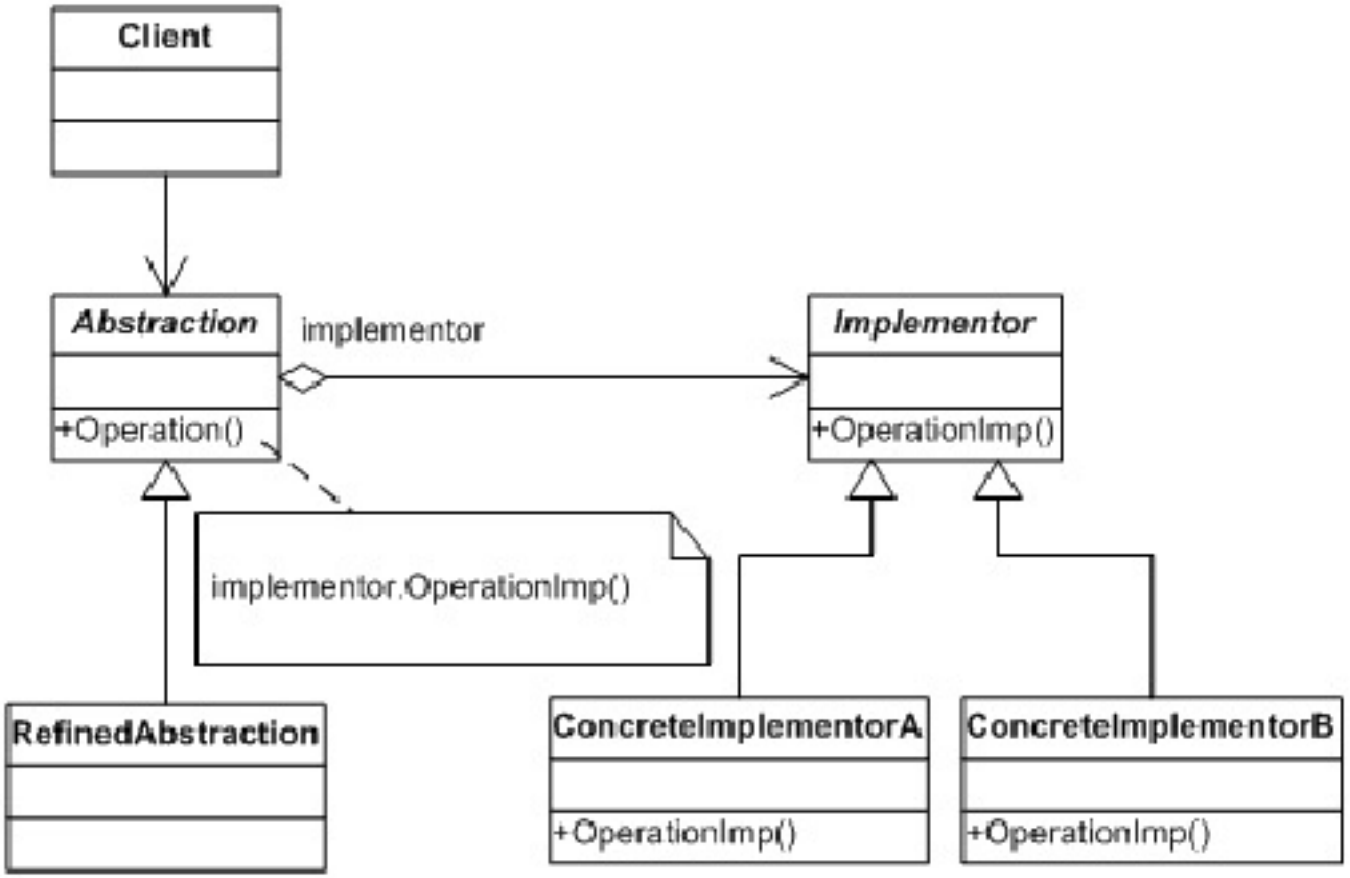


Model Adaptor - implementare

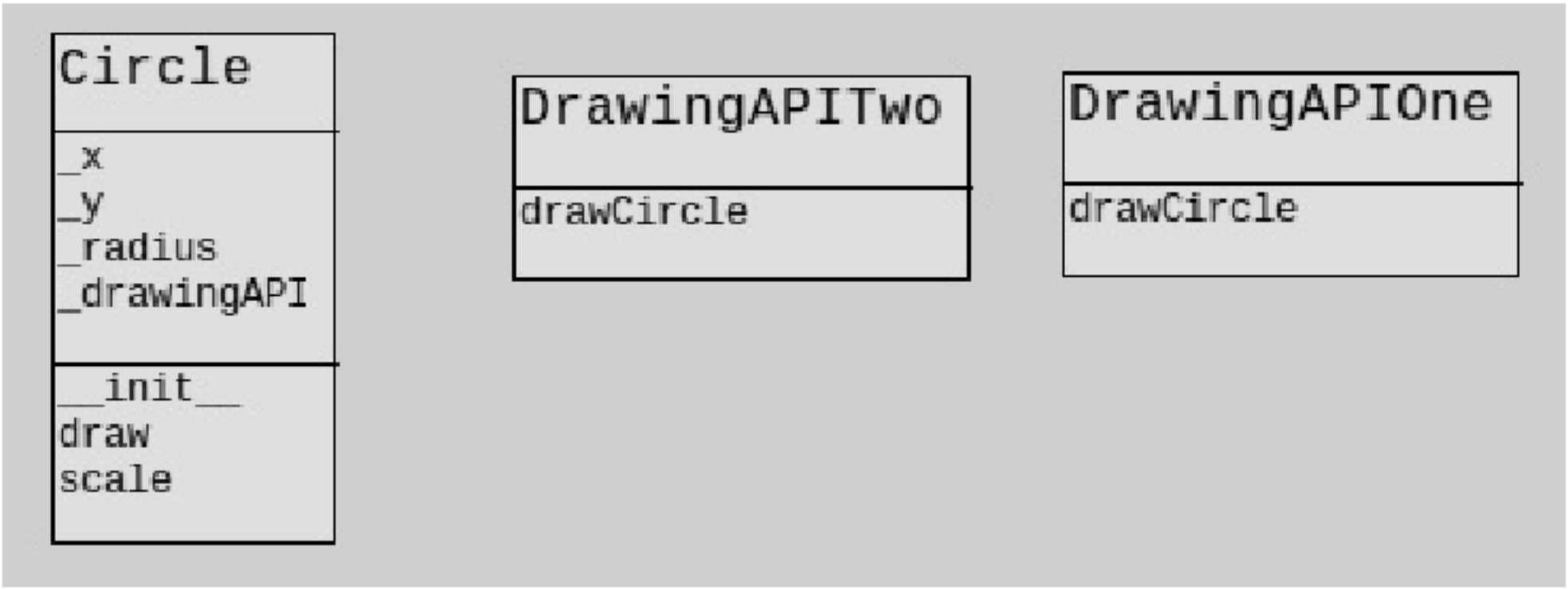
```
class EuropeanSocketInterface:
    def voltage(self): pass
    def live(self): pass
    def neutral(self): pass
    def earth(self): pass
class Socket(EuropeanSocketInterface):# Adaptee
    def voltage(self):
        return 230
    def live(self):
        return 1
    def neutral(self):
        return -1
    def earth(self):
        return 0
class USASocketInterface:#interfata tinta
    def voltage(self): pass
    def live(self): pass
    def neutral(self): pass
class Adapter(USASocketInterface):# The Adapter
    __socket = None
    def __init__(self, socket):
        self.__socket = socket
    def voltage(self):
        return 110
    def live(self):
        return self.__socket.live()
    def neutral(self):
        return self.__socket.neutral()
```

```
class ElectricKettle:# Client
    __power = None
    def __init__(self, power):
        self.__power = power
    def boil(self):
        if self.__power.voltage() > 110:
            print("Kettle on fire!")
        else:
            if self.__power.live() == 1 and \
                    self.__power.neutral() == -1:
                print("Coffee time!")
            else:
                print("No power.")
def main():
    # bagam in priza cu adaptor
    socket = Socket()
    adapter = Adapter(socket)
    kettle = ElectricKettle(adapter)
    # facem cafea
    kettle.boil()
    return 0
```

Modelul Pod


Client
Abstraction
implementor
Implementor
+Operation()
+OperationImp()
implementor.OperationImp()
RefinedAbstraction
ConcreteImplementorA
+OperationImp()
ConcreteImplementorB
+OperationImp()

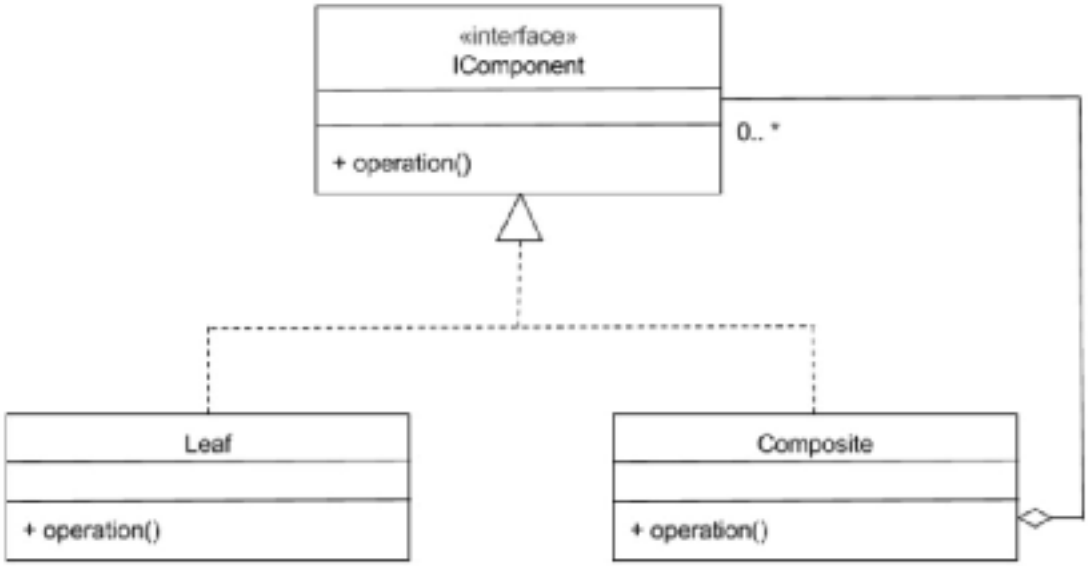
Modelul Pod - caz de utilizare

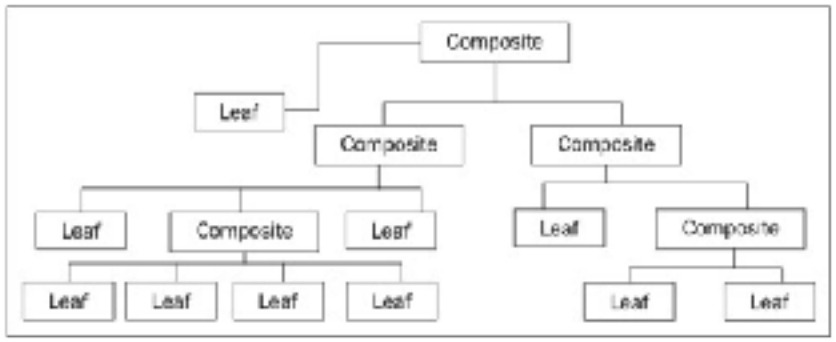


Modelul Pod - implementare

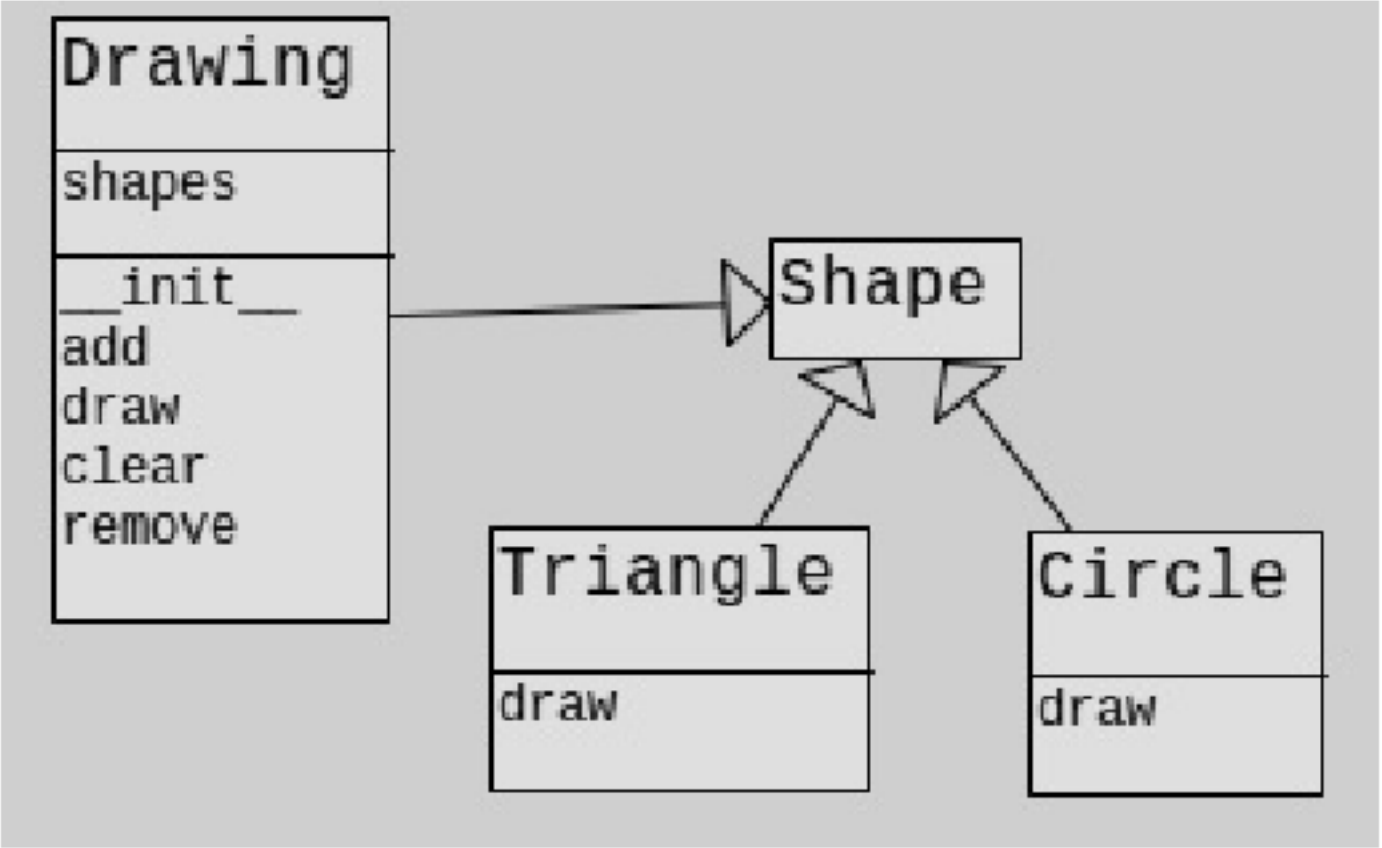
```
class DrawingAPIOne:# concret
    def drawCircle(self, x, y, radius):
        print("API 1 deseneaza un cerc la ({}, {}) cu raza {}".format(x, y, radius))
class DrawingAPITwo: #concret
    def drawCircle(self, x, y, radius):
        print("API 2 deseneaza un cerc la ({}, {}) cu raza {}".format(x, y, radius))
class Circle: #abtract
    def __init__(self, x, y, radius, drawingAPI):
        self._x = x
        self._y = y
        self._radius = radius
        self._drawingAPI = drawingAPI
    def draw(self):#apelul unei implementari concrete specifice
        self._drawingAPI.drawCircle(self._x, self._y, self._radius)
    def scale(self, percent):
        self._radius *= percent
circle1 = Circle(0, 0, 2, DrawingAPIOne())
circle1.draw()
circle2 = Circle(1, 3, 3, DrawingAPITwo())
circle2.draw()
```

Model Compus - forma generală





Model Compus - caz de utilizare

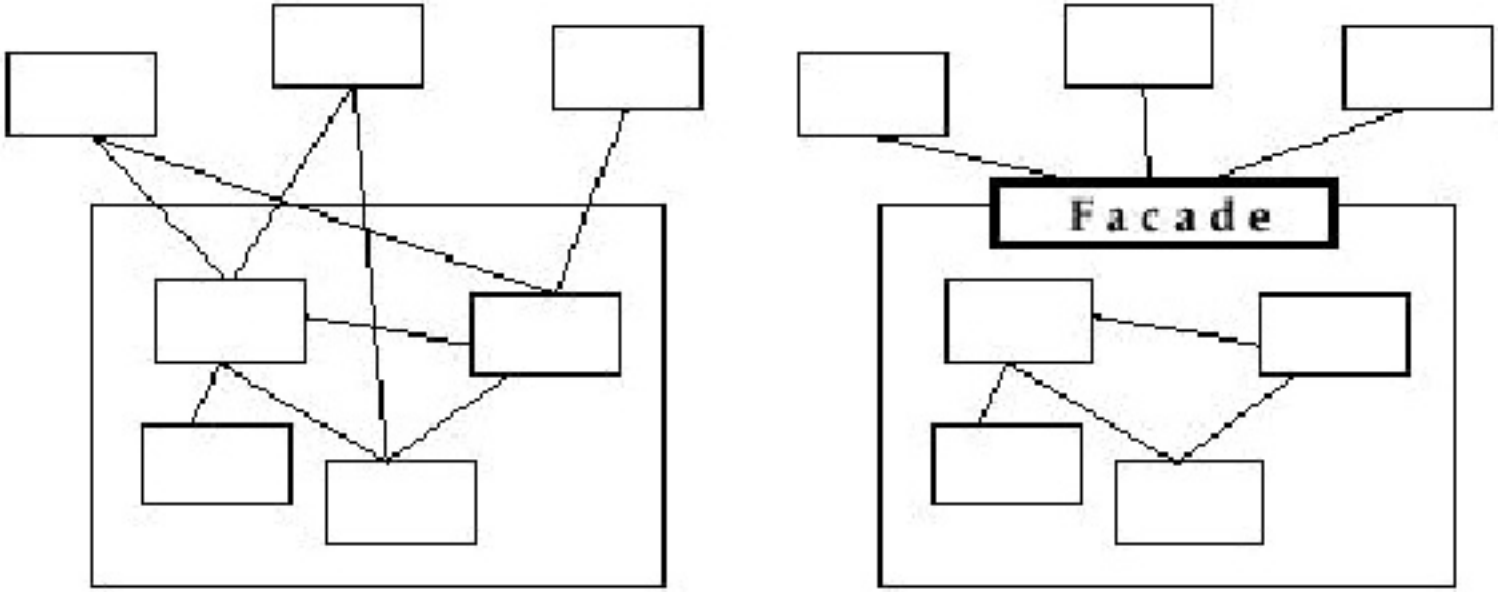


Compus - caz de utilizare - implementare

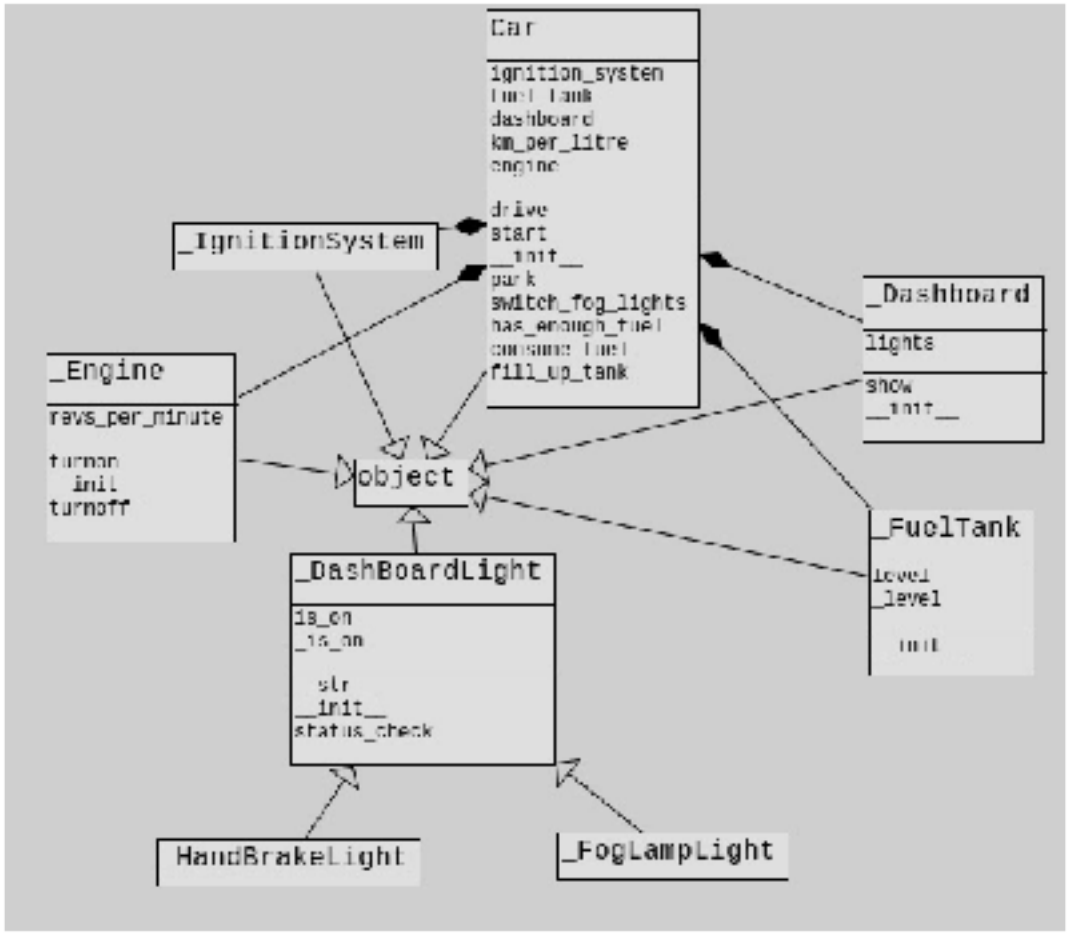
```
import abc
class Shape(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def draw(self, color):
        pass
class Triangle(Shape):
    def draw(self, color):
        print("Desenez un triunghi cu culoarea " + color)
class Circle(Shape):
    def draw(self, color):
        print("Desenez un cerc cu culoarea " + color)
class Drawing(Shape):
    def __init__(self):
        self.shapes = []
    def draw(self, color):
        for sh in self.shapes:
            sh.draw(color)
    def add(self, sh):
        self.shapes.append(sh)
    def remove(self, sh):
        self.shapes.remove(sh)
    def clear(self):
        print("Sterg toate formele din desen")
        self.shapes = []
```

```
if __name__ == '__main__':
    tri1 = Triangle()
    tri2 = Triangle()
    cir  = Circle()

    drawing = Drawing()
    drawing.add(tri1)
    drawing.add(tri2)
    drawing.add(cir)

    drawing.draw("rosu")

    drawing.clear()

    drawing.add(tri1)
    drawing.add(cir)
    drawing.draw("verde")
```

Model Fațadă



Fațadă - Bord


Car
drive
start
__init__
park
switch_fog_lights
fill_up_tank
_IgnitionSystem
_Engine
revs_per_minute
turnon
__init__
turnoff
object
_Dashboard
lights
show
__init__
_FuelTank
level
_level
__init__
_DashBoardLight
is_on
_is_on
__str__
__init__
status_check
HandBrakeLight
_FogLampLight

Model Fațadă - implementare

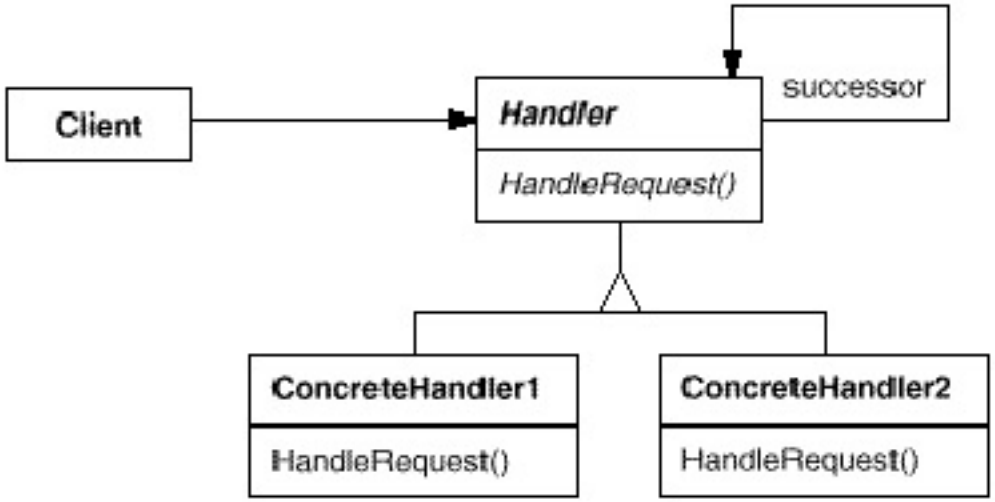
```
class _IgnitionSystem(object):
    @staticmethod
    def produce_spark():
        return True
class _Engine(object):
    def __init__(self):
        self.revs_per_minute = 0
    def turnon(self):
        self.revs_per_minute = 2000
    def turnoff(self):
        self.revs_per_minute = 0
class _FuelTank(object):
    def __init__(self, level=30):
        self._level = level
    @property
    def level(self):
        return self._level
    @level.setter
    def level(self, level):
        self._level = level
```

```
class _DashBoardLight(object):
    def __init__(self, is_on=False):
        self._is_on = is_on
    def __str__(self):
        return self.__class__.__name__
    @property
    def is_on(self):
        return self._is_on
    @is_on.setter
    def is_on(self, status):
        self._is_on = status
    def status_check(self):
        if self._is_on:
            print("{}: Pornit".format(str(self)))
        else:
            print("{}: Oprit".format(str(self)))
class _HandBrakeLight(_DashBoardLight):
    pass
class _FogLampLight(_DashBoardLight):
    pass
```

```
class _Dashboard(object):
    def __init__(self):
        self.lights = {"frana de mana": _HandBrakeLight(), "ceata":
_FogLampLight()}
    def show(self):
        for light in self.lights.values():
            light.status_check()
class Car(object):#Fatada
    def __init__(self):
        self.ignition_system = _IgnitionSystem()
        self.engine = _Engine()
        self.fuel_tank = _FuelTank()
        self.dashboard = _Dashboard()
    @property
    def km_per_litre(self):
        return 17.0
    def consume_fuel(self, km):
        litres = min(self.fuel_tank.level, km / self.km_per_litre)
        self.fuel_tank.level -= litres
    def start(self):
        print("\nPornire...")
        self.dashboard.show()
        if self.ignition_system.produce_spark():
            self.engine.turnon()
        else:
            print("NU pot porni. Eroare la sistemul de aprindere")
    def has_enough_fuel(self, km, km_per_litre):
        litres_needed = km / km_per_litre
        if self.fuel_tank.level > litres_needed:
            return True
        else:
            return False
    def drive(self, km=100):
        print("\n")
        if self.engine.revs_per_minute > 0:
            while self.has_enough_fuel(km, self.km_per_litre):
                self.consume_fuel(km)
                print("S-au condus {}km".format(km))
                print("au mai ramas {:.2f}l de combustibil".format(self.fuel_tank.level))
        else:
            print("Nu se poate merge. Motorul este oprit!")
    def park(self):
        print("\nParcare...")
        self.dashboard.lights["frana de mana"].is_on = True
        self.dashboard.show()
        self.engine.turnoff()
    def switch_fog_lights(self, status):
        print("\nPornire {} lumini ceata...".format(status))
        boolean = True if status == "ON" else False
        self.dashboard.lights["ceata"].is_on = boolean
        self.dashboard.show()
    def fill_up_tank(self):
        print("\nRezervorul a fost umplut!")
        self.fuel_tank.level = 100
def main():# functia main reprezinta clientul
    car = Car()
    car.start()
    car.drive()
    car.switch_fog_lights("ON")
    car.switch_fog_lights("OFF")
    car.park()
    car.fill_up_tank()
    car.drive()
    car.start()
    car.drive()
if __name__ == "__main__":
    main()
```

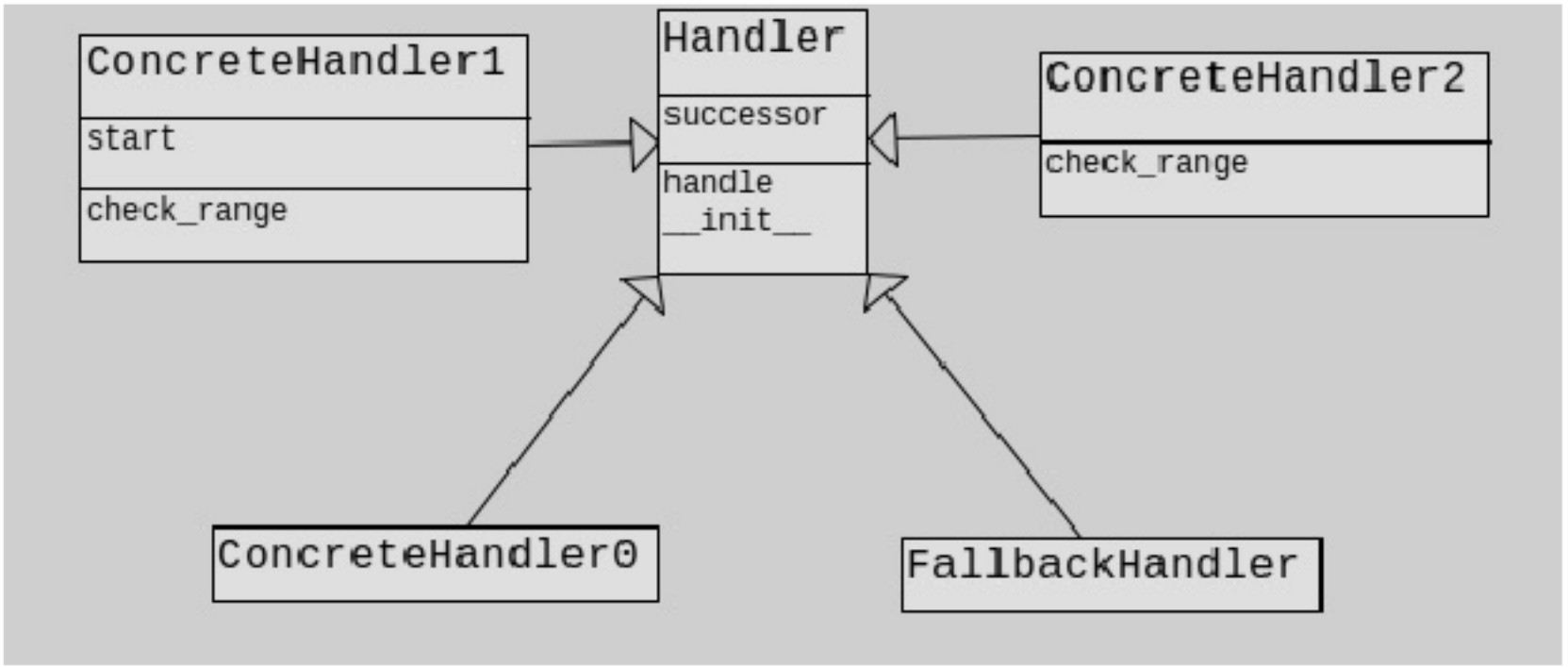
Modele comportamentale

Modelul lanț de responsabilități



Unde o structură tipică de înlănțuire de obiecte ar fi

Caz concret cu trei gestionari diferiți

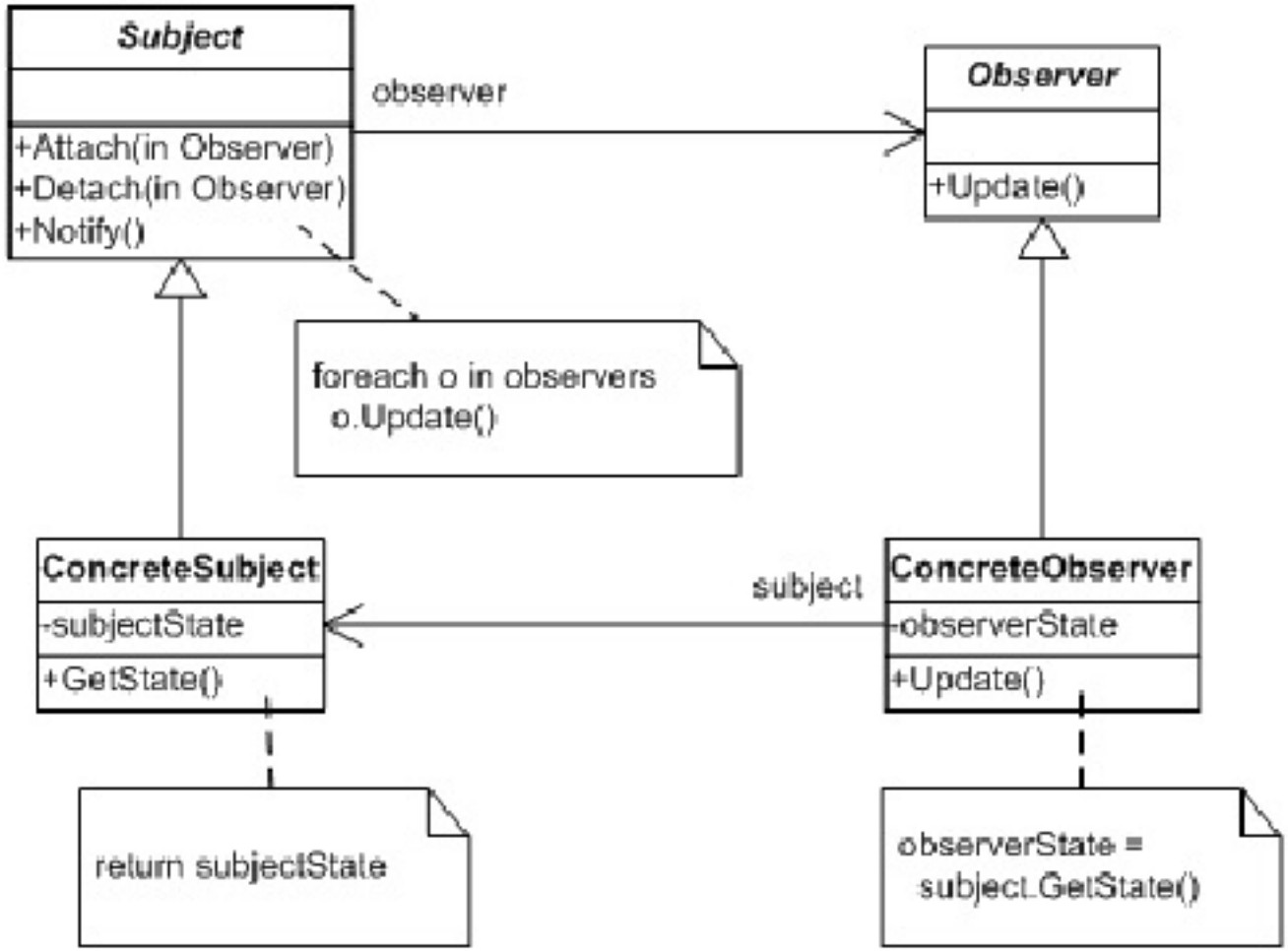


Modelul lanţ de responsabilităţi - implementare

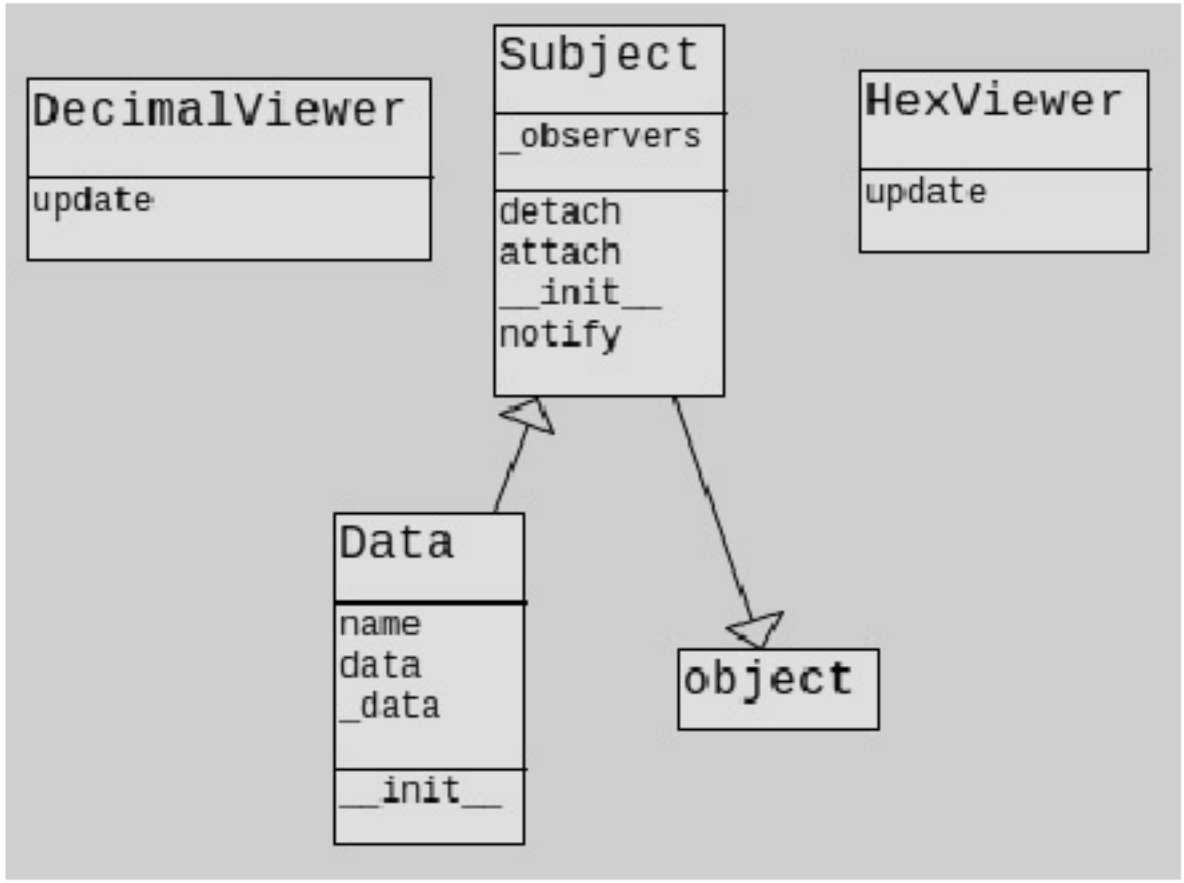
```
import abc
class Handler(metaclass=abc.ABCMeta):
    def __init__(self, successor=None):
        self.successor = successor
    def handle(self, request):
        res = self.check_range(request)
        if not res and self.successor:
            self.successor.handle(request)
    @ abc.abstractmethod
    def check_range(self, request):
        """compara valoarea primita cu un interval predefinita"""
class ConcreteHandler0(Handler):
    @ staticmethod
    def check_range(request):
        if 0 <= request < 10:
            print("cererea {} tratata in gestionarul 0".format(request))
            return True
class ConcreteHandler1(Handler):#are propria stare interna
    start, end = 10, 20
    def check_range(self, request):
        if self.start <= request < self.end:
            print("cererea {} tratata de gestionarul 1".format(request))
            return True
class ConcreteHandler2(Handler):#utilizeaza metode ajutatoare
    def check_range(self, request):
        start, end = self.get_interval_from_db()
        if start <= request < end:
            print("cererea {} tratata de gestionarul 2".format(request))
            return True
    @ staticmethod
    def get_interval_from_db():
        return (20, 30)
class FallbackHandler(Handler):
    @ staticmethod
    def check_range(request):
        print("am terminat de parcurs lantul - nu exista tratare pentru cazul
{}".format(request))
        return False
h0 = ConcreteHandler0() #creez gestionarii
h1 = ConcreteHandler1()
h2 = ConcreteHandler2(FallbackHandler())
h0.successor = h1 #creez lantul
h1.successor = h2
requests = [2, 5, 14, 22, 18, 3, 35, 27, 20]
for request in requests:
    h0.handle(request)
```

Modelul Observator

Modelul Observator



Observator - caz de utilizare



Model Observator - Caz de utilizare

```
class Subject(object):
    def __init__(self):
        self._observers = []
    def attach(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)
    def detach(self, observer):
        try:
            self._observers.remove(observer)
        except ValueError:
            pass
    def notify(self, modifier=None):
        for observer in self._observers:
            if modifier != observer:
                observer.update(self)
class Data(Subject):#exemplu de utilizare
    def __init__(self, name=''):
        Subject.__init__(self)
        self.name = name
        self._data = 0
    @property
    def data(self):
        return self._data
    @data.setter
    def data(self, value):
        self._data = value
        self.notify()
class HexViewer:
    def update(self, subject):
        print(u'Format Hexa: Subiectul %s are valoarea 0x%x' % (subject.name, subject.data))
class DecimalViewer:
    def update(self, subject):
        print(u'Format Zecimal: Subiectul %s are valoarea  %d ' % (subject.name, subject.data))
```

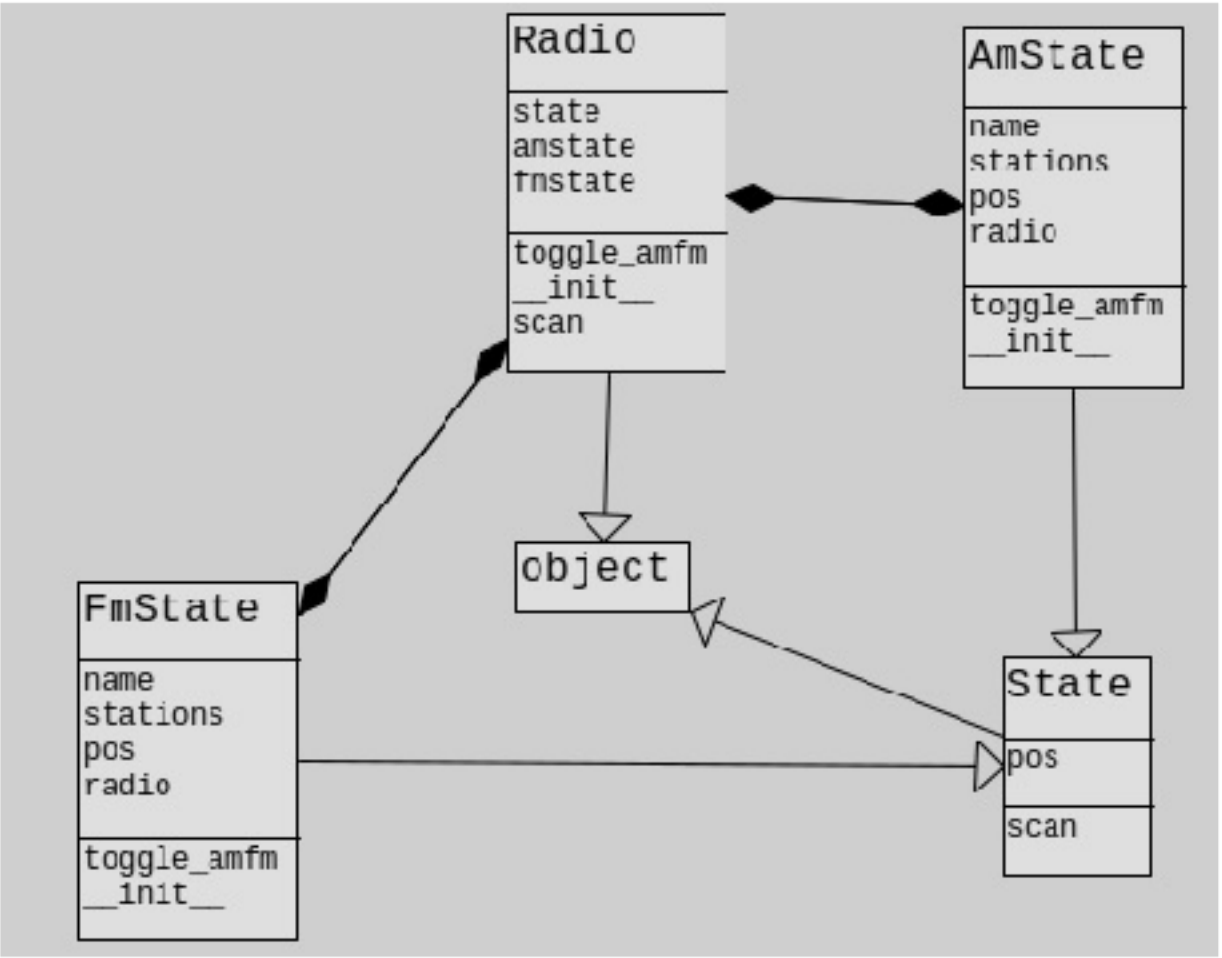
```
def main():#utlizare
    data1 = Data('Data 1')
    data2 = Data('Data 2')
    view1 = DecimalViewer()
    view2 = HexViewer()
    data1.attach(view1)
    data1.attach(view2)
    data2.attach(view2)
    data2.attach(view1)
    print(u"Stabilim valoarea 11 = 10")
    data1.data = 10
    print(u"Stabilim valoarea 12 = 15")
    data2.data = 15
    print(u"Stabilim valoarea 11 = 3")
    data1.data = 3
    print(u"Stabilim valoarea 12 = 5")
    data2.data = 5
    print(u"NU mai utilizam Afisarea Hexa pentru data1 si data2.")
    data1.detach(view2)
    data2.detach(view2)
    print(u"Stabilim valoarea 11 = 10")
    data1.data = 10
    print(u"Stabilim valoarea 12 = 15")
    data2.data = 15
if __name__ == '__main__':
    main()
```

Varianta publish - subscribe

```
class Provider:
    def __init__(self):
        self.msg_queue = []
        self.subscribers = {}
    def notify(self, msg):
        self.msg_queue.append(msg)
    def subscribe(self, msg, subscriber):
        self.subscribers.setdefault(msg, []).append(subscriber)
    def unsubscribe(self, msg, subscriber):
        self.subscribers[msg].remove(subscriber)
    def update(self):
        for msg in self.msg_queue:
            for sub in self.subscribers.get(msg, []):
                sub.run(msg)
        self.msg_queue = []
class Publisher:
    def __init__(self, msg_center):
        self.provider = msg_center
    def publish(self, msg):
        self.provider.notify(msg)
class Subscriber:
    def __init__(self, name, msg_center):
        self.name = name
        self.provider = msg_center
    def subscribe(self, msg):
        self.provider.subscribe(msg, self)
    def unsubscribe(self, msg):
        self.provider.unsubscribe(msg, self)
    def run(self, msg):
        print("{} got {}".format(self.name, msg))
def main():
    message_center = Provider()
    fftv = Publisher(message_center)
    jim = Subscriber("jim", message_center)
    jim.subscribe("cartoon")
    jack = Subscriber("jack", message_center)
    jack.subscribe("music")
    gee = Subscriber("gee", message_center)
    gee.subscribe("movie")
    vani = Subscriber("vani", message_center)
    vani.subscribe("movie")
    vani.unsubscribe("movie")
    fftv.publish("cartoon")
    fftv.publish("music")
    fftv.publish("ads")
    fftv.publish("movie")
    fftv.publish("cartoon")
    fftv.publish("cartoon")
    fftv.publish("movie")
    fftv.publish("blank")
    message_center.update()
if __name__ == "__main__":
    main()
```

Modelul automatului finit State

Automatul - caz concret

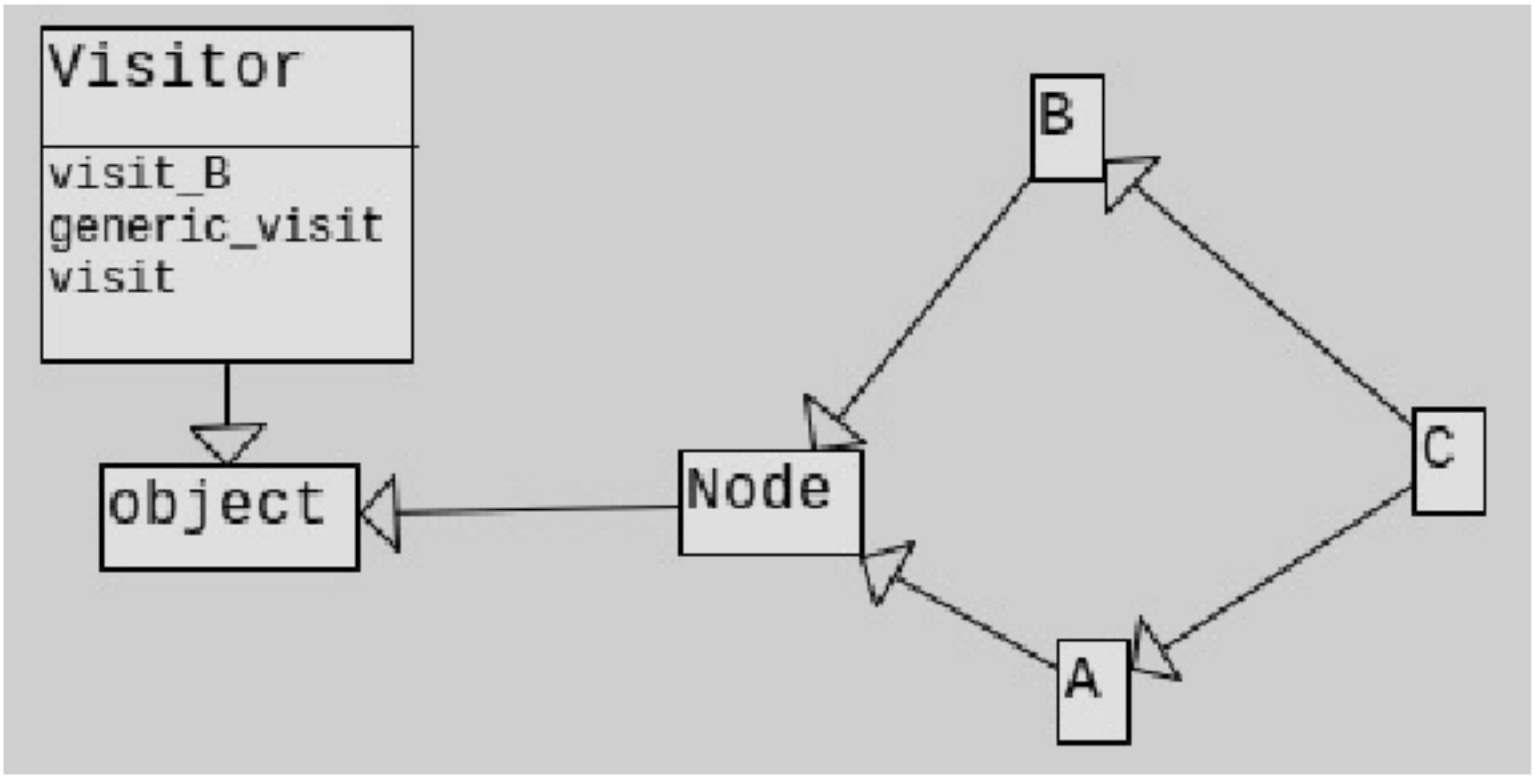


Modelul automatului finit - caz de utilizare

```
class State(object):
    def scan(self): #stare de baza pentru a pune in comun o functionalitate
        self.pos += 1
        if self.pos == len(self.stations):
            self.pos = 0
        print(u"Caut... Am gasit o statie la %s in banda %s" %
(self.stations[self.pos], self.name))
class AmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["1250", "1380", "1510"]
        self.pos = 0
        self.name = "AM"
    def toggle_amfm(self):
        print(u"Comutare in banda FM")
        self.radio.state = self.radio.fmstate
class FmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["81.3", "89.1", "103.9"]
        self.pos = 0
        self.name = "FM"
    def toggle_amfm(self):
        print(u"Comutare in banda AM")
        self.radio.state = self.radio.amstate
```

```
class Radio(object):
    def __init__(self): # are un buton pentru cautare post si unul #
pentru schimbare banda
        self.amstate = AmState(self)
        self.fmstate = FmState(self)
        self.state = self.amstate
    def toggle_amfm(self):
        self.state.toggle_amfm()
    def scan(self):
        self.state.scan()
def main(): #testare radio
    radio = Radio()
    actions = [radio.scan] * 2 + [radio.toggle_amfm] + [radio.scan] * 2
    actions *= 2
    for action in actions:
        action()
if __name__ == '__main__':
    main()
```

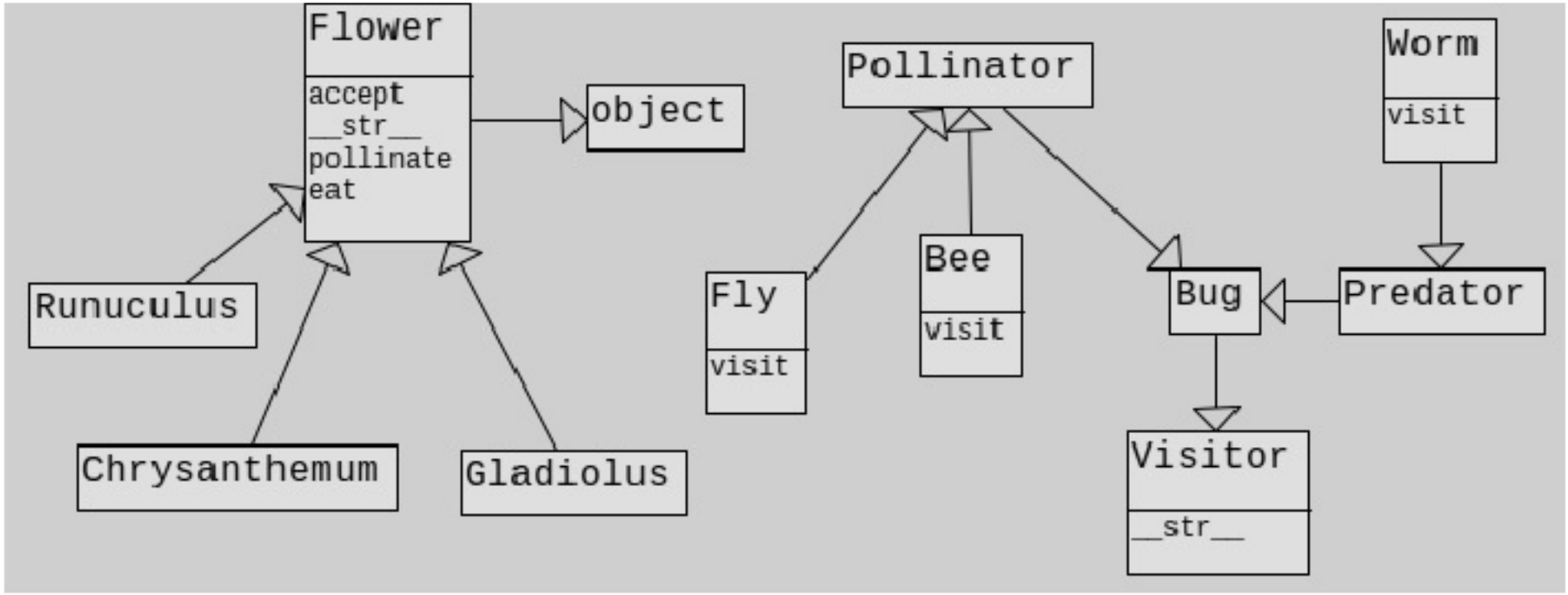
Modelul Vizitator?



Vizitator - Exemplu de tratare

```
class Node(object):
    pass
class A(Node):
    pass
class B(Node):
    pass
class C(A, B):
    pass
class Visitor(object):
    def visit(self, node, *args, **kwargs):
        meth = None
        for cls in node.__class__.__mro__:
            meth_name = 'visit_' + cls.__name__
            meth = getattr(self, meth_name, None)
            if meth:
                break
        if not meth:
            meth = self.generic_visit
        return meth(node, *args, **kwargs)
    def generic_visit(self, node, *args, **kwargs):
        print('generic_visit ' + node.__class__.__name__)
    def visit_B(self, node, *args, **kwargs):
        print('visit_B ' + node.__class__.__name__)

def main():
    a = A()
    b = B()
    c = C()
    visitor = Visitor()
    visitor.visit(a)
    visitor.visit(b)
    visitor.visit(c)

if __name__ == "__main__":
    main()
```

Vizitatori în gradină



Exemplu 2 de vizitator

```
import random
class Flower(object):
    def accept(self, visitor):
        visitor.visit(self)
    def pollinate(self, pollinator):
        print(self, "polenizata de ", pollinator)
    def eat(self, eater):
        print(self, "mancata de ", eater)
    def __str__(self):
        return self.__class__.__name__ #intorc numele clasei
class Gladiolus(Flower): pass
class Runuculus(Flower): pass
class Chrysanthemum(Flower): pass
class Visitor:
    def __str__(self):
        return self.__class__.__name__
class Bug(Visitor): pass
class Pollinator(Bug): pass
class Predator(Bug): pass
class Bee(Pollinator): # ce face o albina
    def visit(self, flower):
        flower.pollinate(self)
class Fly(Pollinator): #ce face o musca
    def visit(self, flower):
        flower.pollinate(self)
class Worm(Predator): # ce face un vierme
    def visit(self, flower):
        flower.eat(self)
def flowerGen(n):
    flwrs = Flower.__subclasses__()
    for i in range(n):
        yield random.choice(flwrs)()
bee = Bee()
fly = Fly()
worm = Worm()
for flower in flowerGen(10):
    flower.accept(bee)
    flower.accept(fly)
    flower.accept(worm)
```

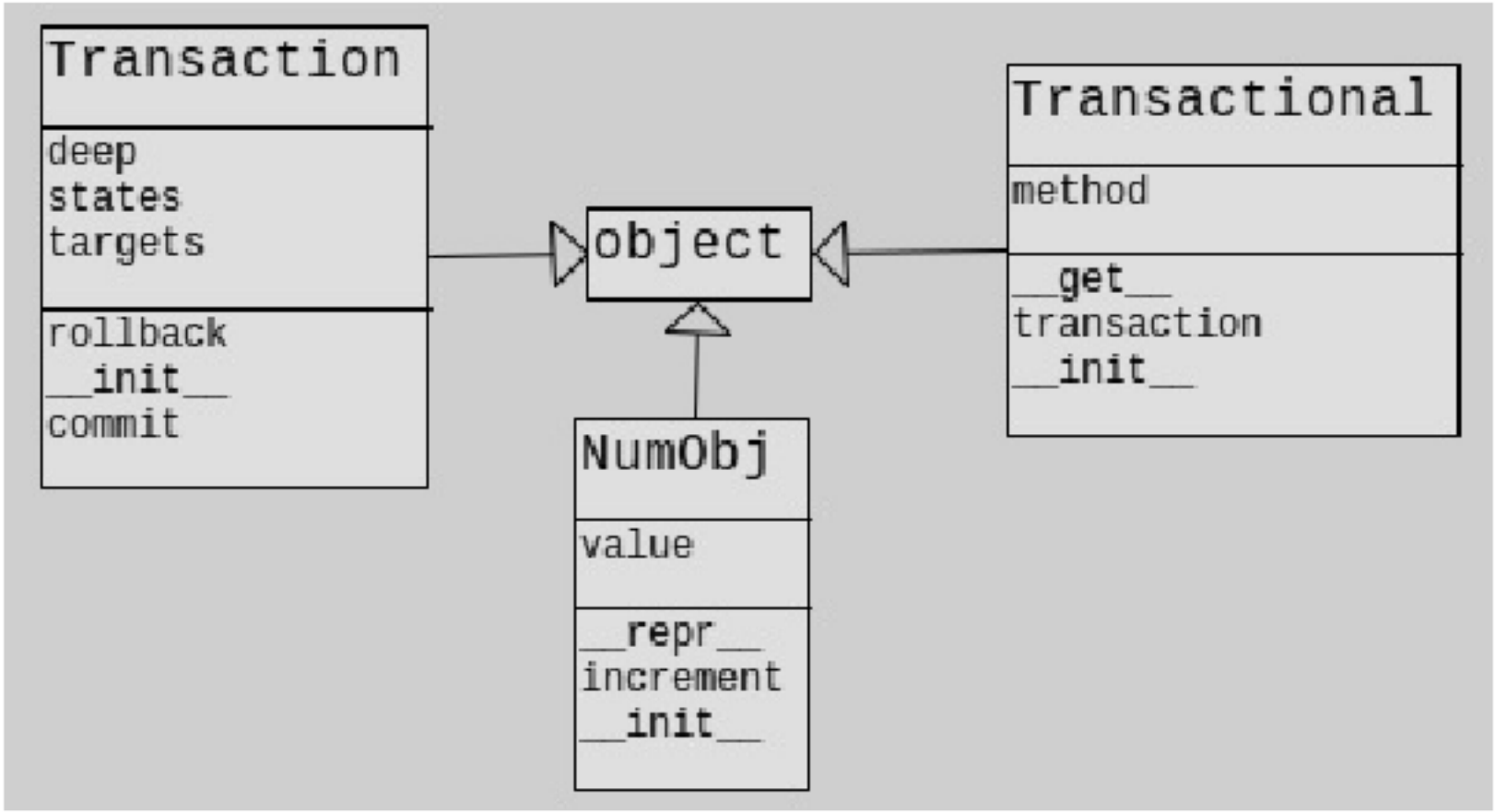
Modelul comandă

```
import os
from os.path import lexists
class MoveFileCommand(object):
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
    def execute(self):
        self.rename(self.src, self.dest)
    def undo(self):
        self.rename(self.dest, self.src)
    def rename(self, src, dest):
        print(u"Redenumesc %s ca %s" % (src, dest))
        os.rename(src, dest)

def main():
    command_stack = []
    # se introduc comenzile in stiva
    command_stack.append(MoveFileCommand('bugs.txt', 'bunny.txt'))
    command_stack.append(MoveFileCommand('bunny.txt',
'stimpy.txt'))
```


```
assert not lexists("bugs.txt")
assert not lexists("bunny.txt")
assert not lexists("stimpy.txt")
try:
    with open("bugs.txt", "w"):  # Creez fisier
        pass # si cam atat
    for cmd in command_stack:#pot fi executate ulterior
        cmd.execute()

    for cmd in reversed(command_stack):
        cmd.undo() #poate fi anulat efectul
finally:#curat in urma mea
    os.unlink("bugs.txt")
```

```
if __name__ == "__main__":
    main()
```

modelul restaurare/reamintire (latină: memento)



si implementarea

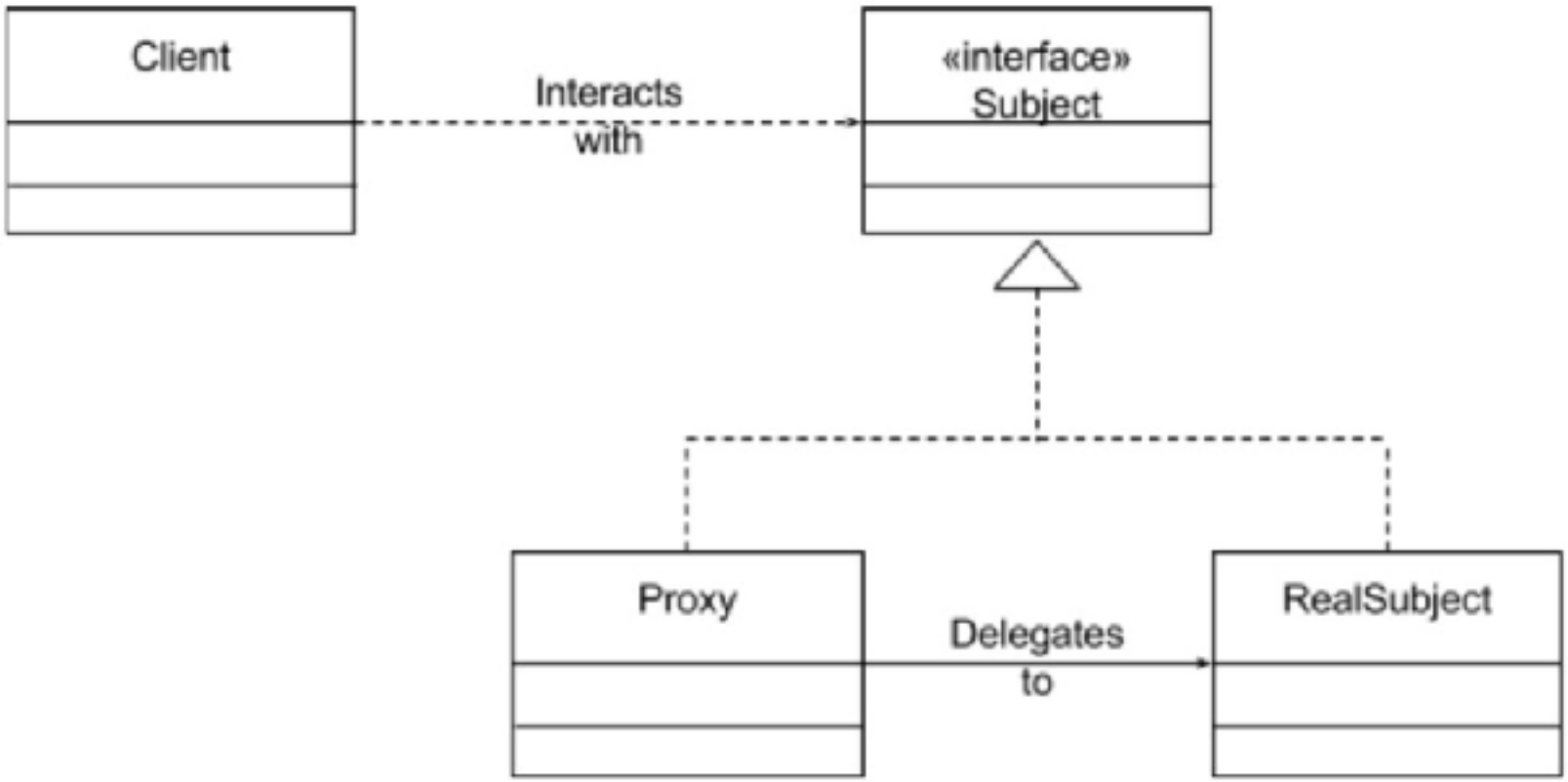
```
ifrom copy import copy
from copy import deepcopy
def memento(obj, deep=False):
    state = deepcopy(obj.__dict__) if deep else
copy(obj.__dict__)
    def restore():
        obj.__dict__.clear()
        obj.__dict__.update(state)
    return restore
class Transaction(object):# o tranzactie cu restaurare
    deep = False
    states = []
    def __init__(self, deep, *targets):
        self.deep = deep
        self.targets = targets
        self.commit()
    def commit(self):
        self.states = [memento(target, self.deep) for target in
self.targets]
    def rollback(self):
        for a_state in self.states:
            a_state()
```

```
class Transactional(object):
    def __init__(self, method):
        self.method = method

    def __get__(self, obj, T):
        def transaction(*args, **kwargs):
            state = memento(obj)
            try:
                return self.method(obj, *args, **kwargs)
            except Exception as e:
                state()
                raise e
        return transaction
class NumObj(object):
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return '<%s: %r>' % (self.__class__.__name__,
self.value)
```

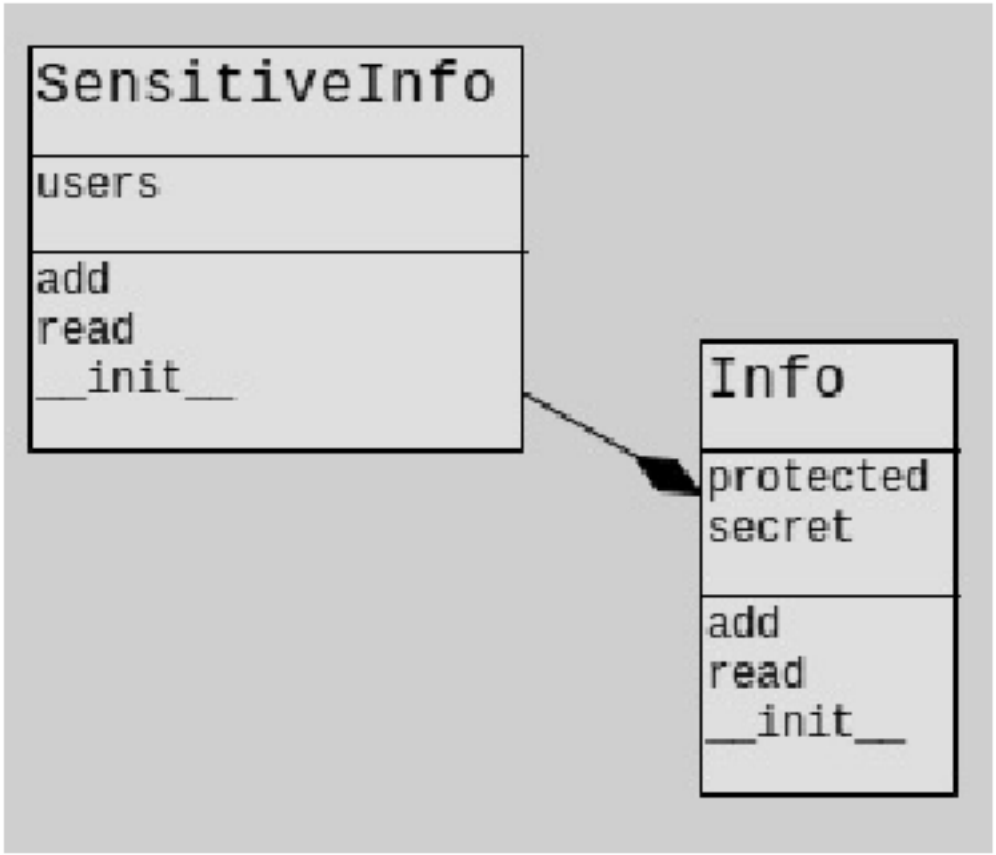
continuare

```
    def increment(self):
        self.value += 1
    @Transactional
    def do_stuff(self): #tot ca sa crape
        self.value = '1111'  # <- valoare incorecta
        self.increment()  # <- crapa si face restaurare
#main
num_obj = NumObj(-1)
print(num_obj)
a_transaction = Transaction(True, num_obj)
try:
    for i in range(3):
        num_obj.increment()
        print(num_obj)
        a_transaction.commit()
        print('-- tranzactie finalizata')
    num_obj.value += 'x'  # incorcet va crapa
    print(num_obj)
except Exception:
    a_transaction.rollback()
    print('-- tranzactie esuata - restauram la situatia anterioara')
print(num_obj)
print('-- alte instructiuni gresite ...')
try:
    num_obj.do_stuff()
except Exception:
    print('-> do_stuff a crapat!')
    import sys
    import traceback
    traceback.print_exc(file=sys.stdout)
#afisez explicit erorile de executie
print(num_obj)
```

Intermediar - forma generală



Intermediar - caz de utilizare

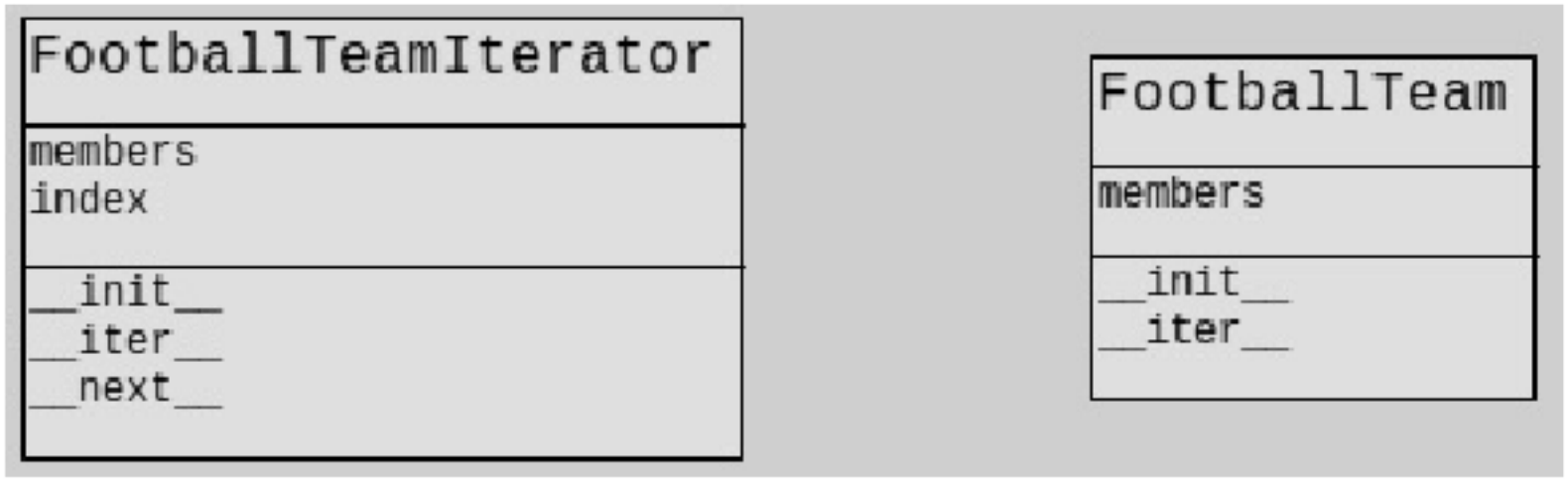


Intermediar - caz de utilizare - implementare

```
class SensitiveInfo:
    def __init__(self):
        self.users = ['bula', 'strula', 'bugs', 'mike']

    def read(self):
        nb = len(self.users)
        print(f"Sunt {nb} utilizatori: {' '.join(self.users)}")

    def add(self, user):
        self.users.append(user)
        print(f'Adauga loser {user}')
class Info:
    def __init__(self):
        self.protected = SensitiveInfo()
        self.secret = '0xdeadbeef'

    def read(self):
        self.protected.read()

    def add(self, user):
        sec = input('dati parola? ')
        self.protected.add(user) if sec == self.secret else print("Mai
incearca!")
```

```
def main():
    info = Info()

    while True:
        print('1. afiseaza lista |==| 2. adauga loser |==| 3. iesire')
        key = input('Alegeti o optiune: ')
        if key == '1':
            info.read()
        elif key == '2':
            name = input('Dati numele utilizatorului: ')
            info.add(name)
        elif key == '3':
            exit()
        else:
            print(f'Optiune invalida: {key}')


if __name__ == '__main__':
    main()
```

Model iterator



Modelul iterator - implementare

```
class FootballTeamIterator:
    def __init__(self, members):# lista de jucatori si
antronori
        self.members = members
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.members):
            val = self.members[self.index]
            self.index += 1
            return val
        else:
            raise StopIteration()
class FootballTeam:
    def __init__(self, members):
        self.members = members
    def __iter__(self):
        return FootballTeamIterator(self.members)
```

```
def main():
    members = []
    for x in range(1, 23):
        members.append(f'jucator_nr_{str(x)}')
    members = members + ['antrenor principal',
'antrenor secund', 'antrenorul cu cafelele']
    team = FootballTeam(members)
    team_it = iter(team)

    while True:
        try:
            print(next(team_it))
        except StopIteration:
            break

if __name__ == '__main__':
    main()
```

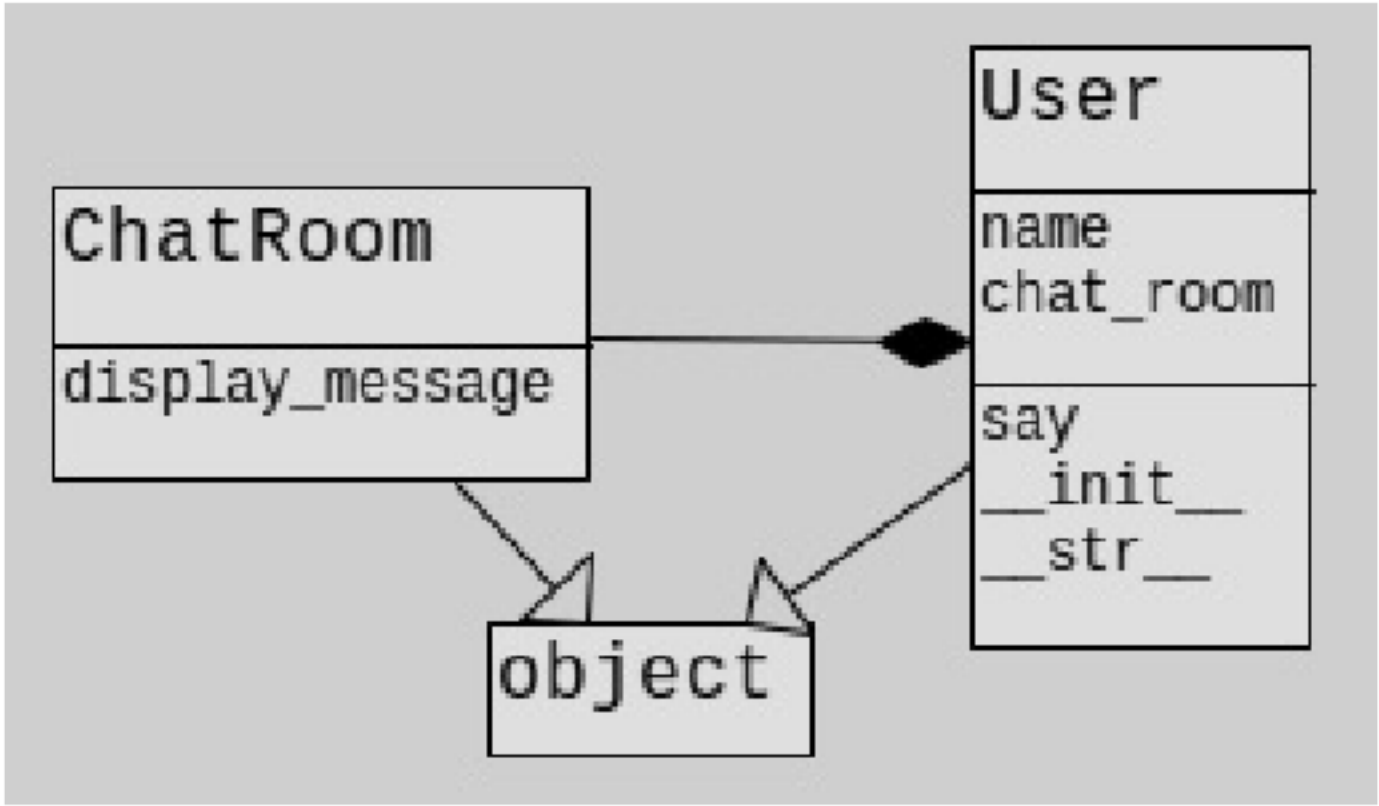
Modelul Strategie

```
import time
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]
SLOW = 3  # in secunde
LIMIT = 5  # in charactere
WARNING = 'nu este bine ai ales un algoritm lent :('
def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True
def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW) #pentru a simula un alg lent
    return True if len(set(s)) == len(s) else False
def allUnique(word, strategy):
    return strategy(word)
```

```
def main():
    WORD_IN_DESC = 'Introducei un cuvant (papa pentru iesire)
>'
    STRAT_IN_DESC = 'Alegeti o strategie: [1] bazta pe un set, [2]
sorteaza si imperecheaza >'
    while True:
        word = None
        while not word:
            word = input(WORD_IN_DESC)
            if word == 'papa':
                print('pa!!!')
                return
            strategy_picked = None
            strategies = {'1': allUniqueSet, '2': allUniqueSort}
            while strategy_picked not in strategies.keys():
                strategy_picked = input(STRAT_IN_DESC)
                try:
                    strategy = strategies[strategy_picked]
                    result = allUnique(word, strategy)
                    print(f'allUnique({word}): {result}')
                except KeyError as err:
                    print(f'Selectie gresita!: {strategy_picked}')
if __name__ == "__main__":
    main()
```

Modelul Mediator

Versiune simplificată



Model Mediator - caz de utilizare

```
class ChatRoom(object):#clasa mediator

    def display_message(self, user, message):
        print("[{} zice]: {}".format(user, message))

class User(object):
    def __init__(self, name):
        self.name = name
        self.chat_room = ChatRoom()

    def say(self, message):
        self.chat_room.display_message(self, message)

    def __str__(self):
        return self.name
```

```
def main():
    vasale = User('Vasale')
    tica = User('Tica2')
    altul = User('Altul')

    vasale.say("Echipa adunarea la ora 3 dupa
amiaza!")
    tica.say("Da sefu pot sa astept pana atunci in
fata usii?")
    altul.say("da' eu pot?")

if __name__ == '__main__':
    main()
```