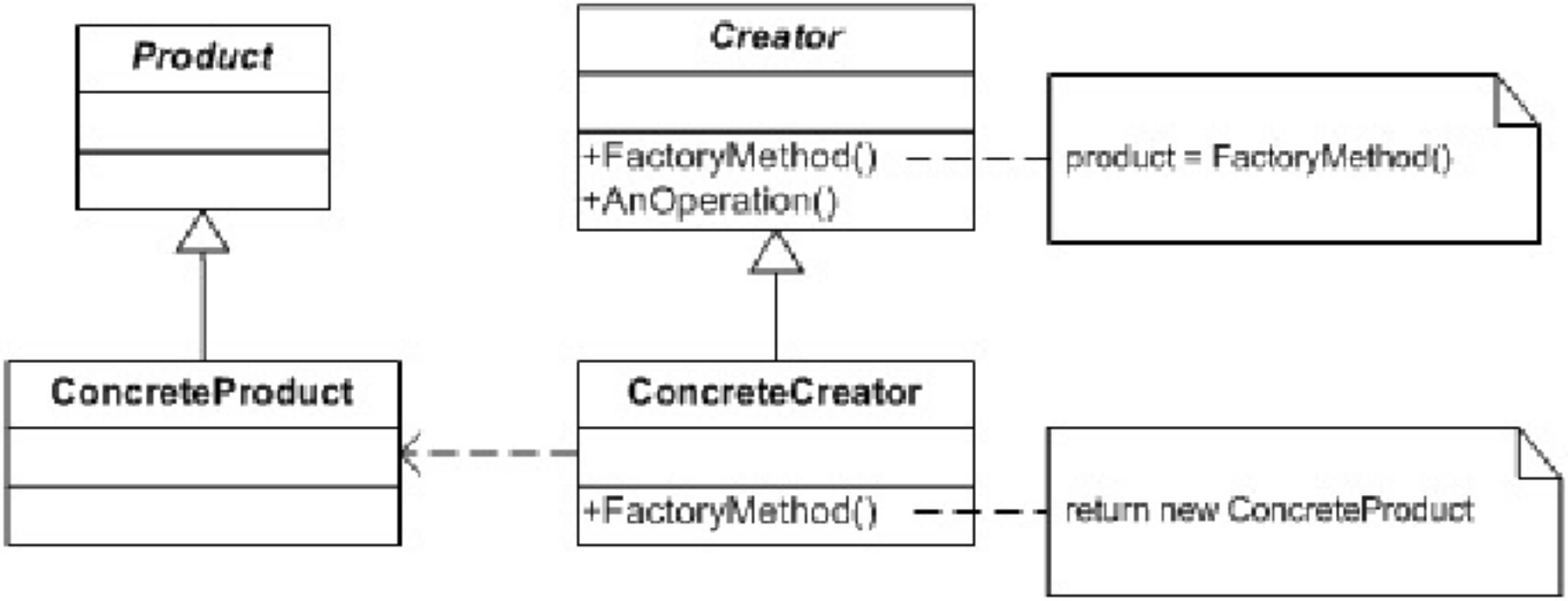
Paradigma Modelelor de Proiectare

Cursul nr. 9

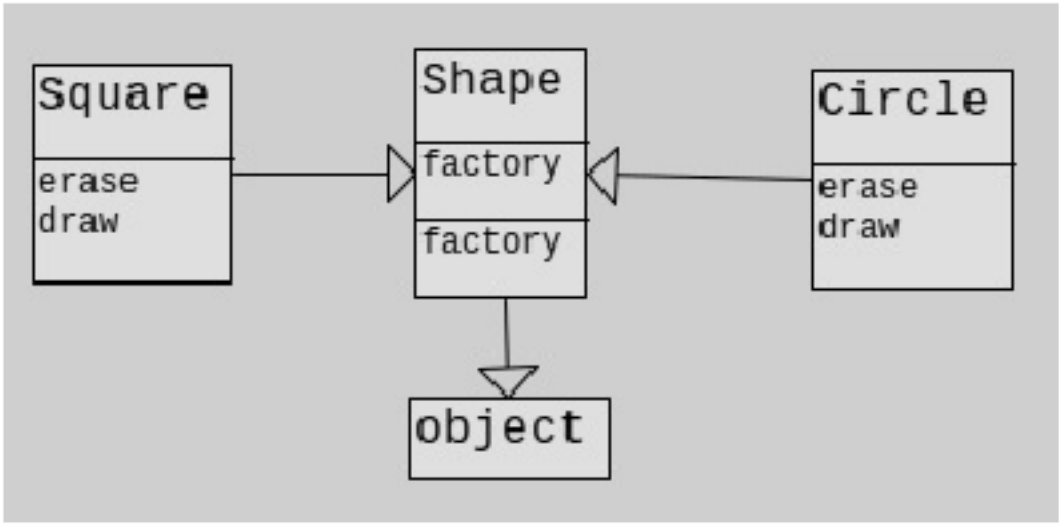
Mihai Zaharia

Modelul Fabrică de obiecte

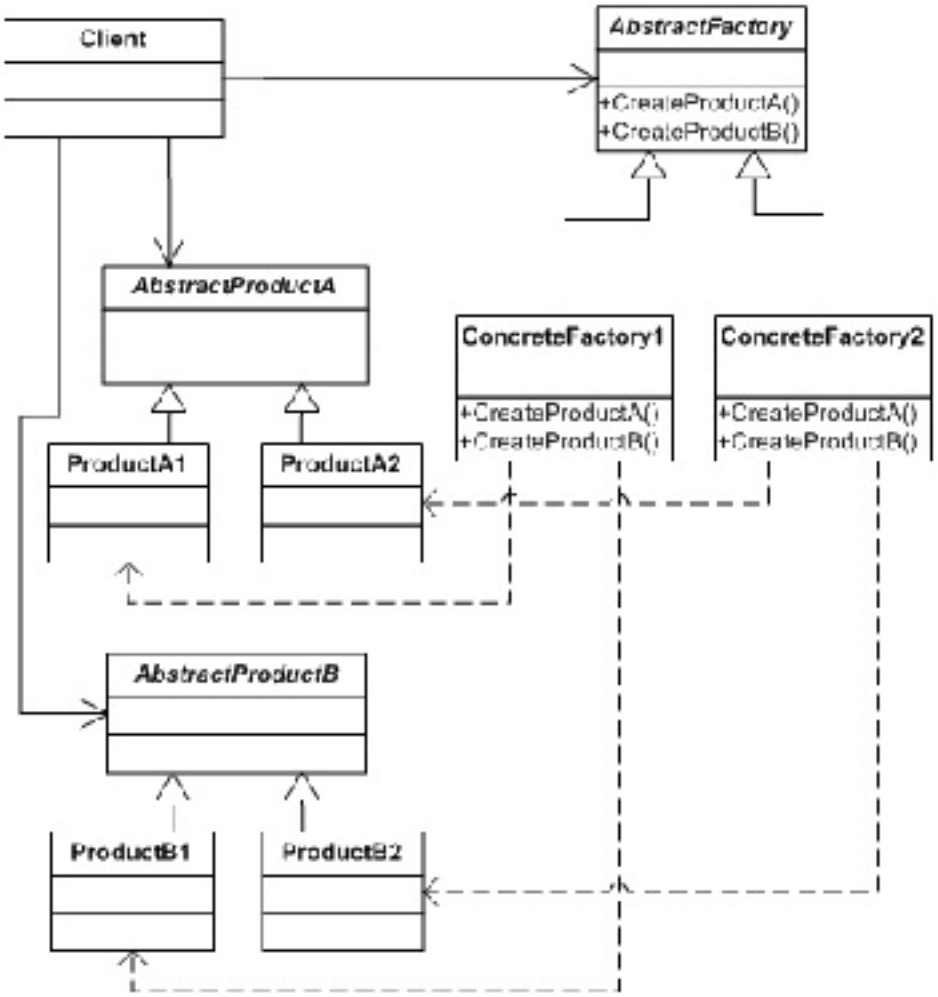


Modelul Fabrică de obiecte - caz de utilizare

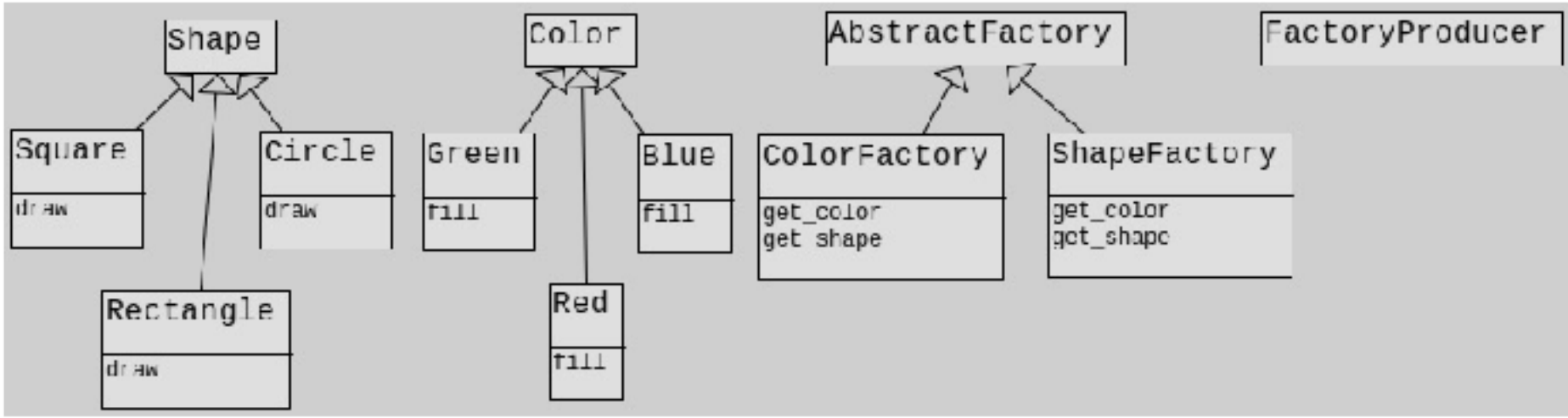
```
class Shape(object):
    def factory(type):  #return eval(type + "()")
        if type == "Circle": return Circle()
        if type == "Square": return Square()
        assert 0, "Bad shape creation: " + type
    factory = staticmethod(factory)
class Circle(Shape):
    def draw(self): print("Circle.draw")
    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

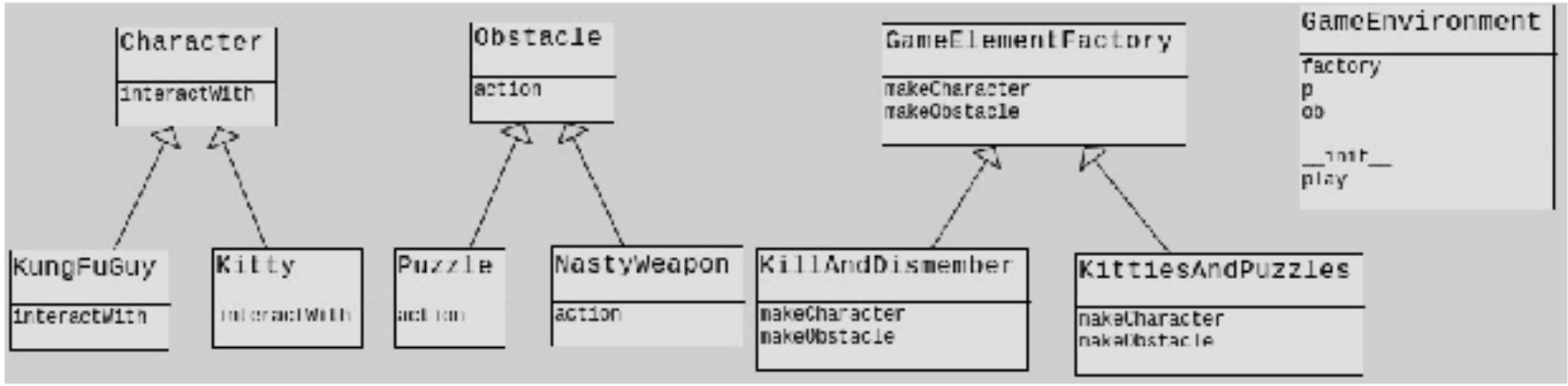


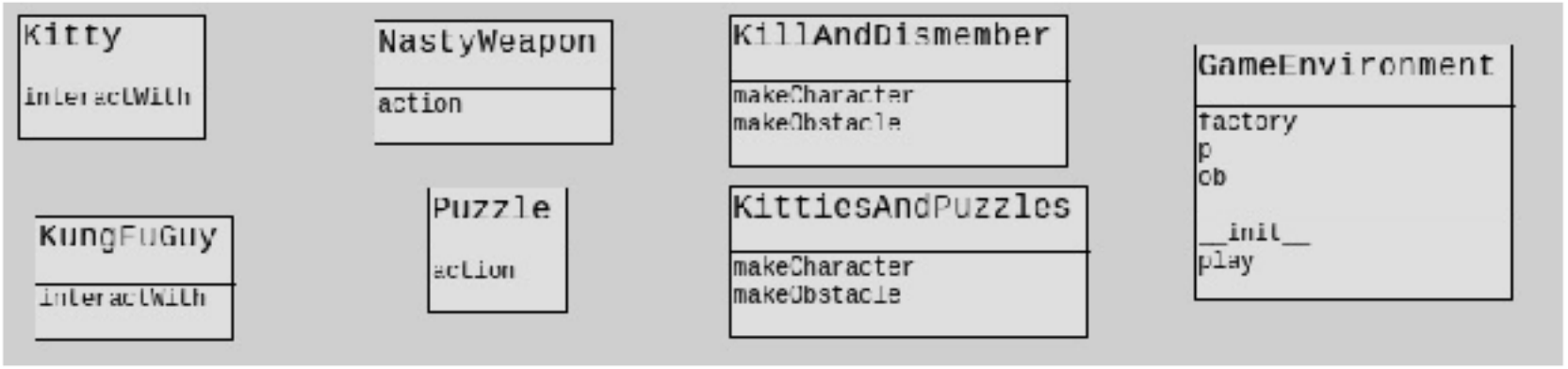
Și implementarea

```
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





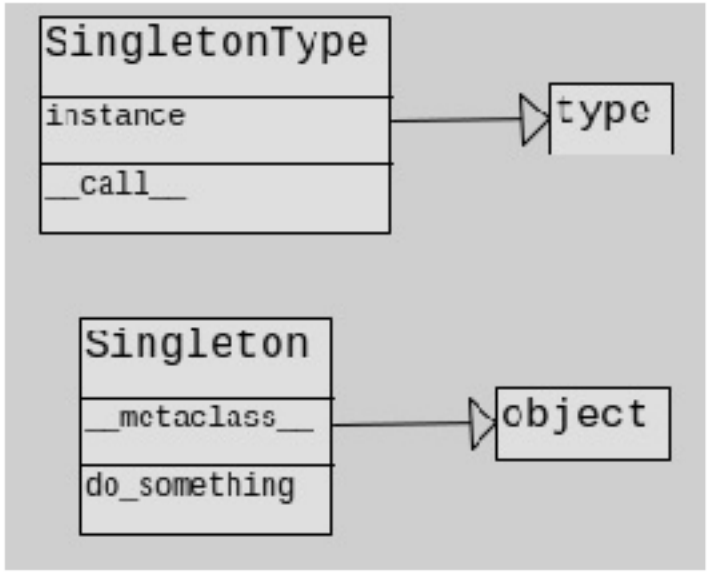
Modelul Fabrica abstractă - OOP vs Structurat

```
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()
```

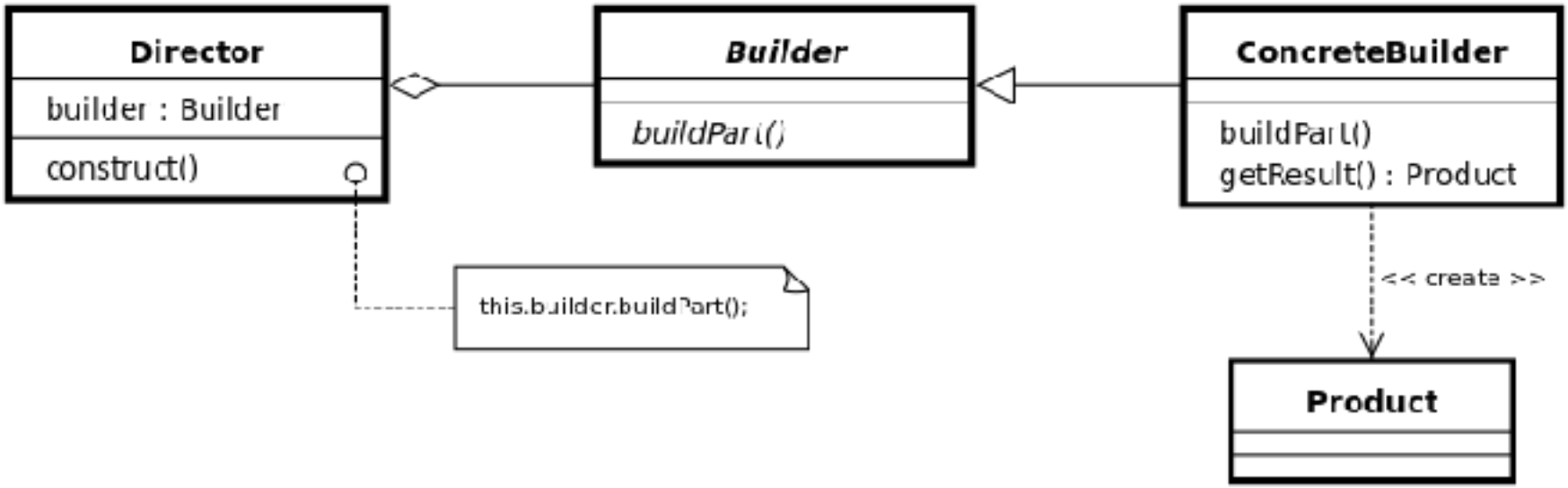
Modelul burlacului

```
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()
```

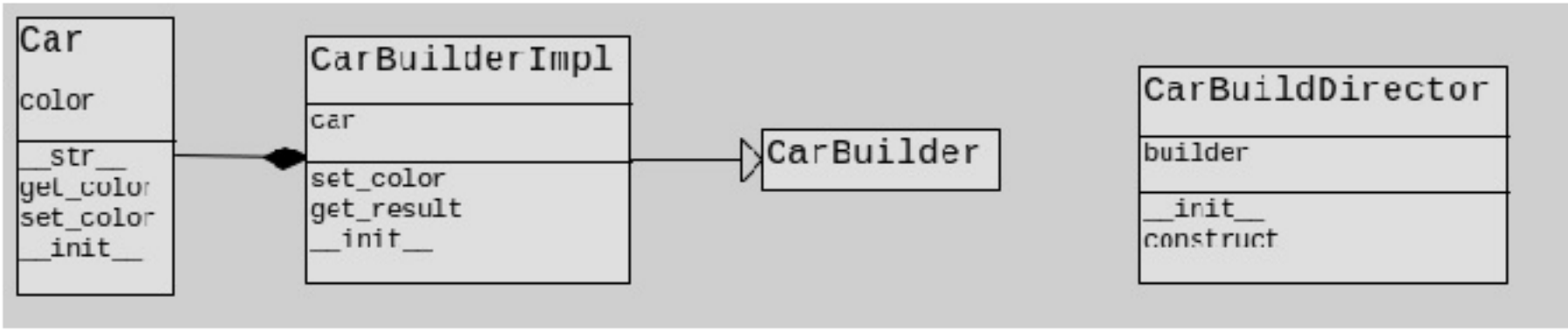


Modelul constructor

Modelul constructor



Modelul constructor - caz de utilizare

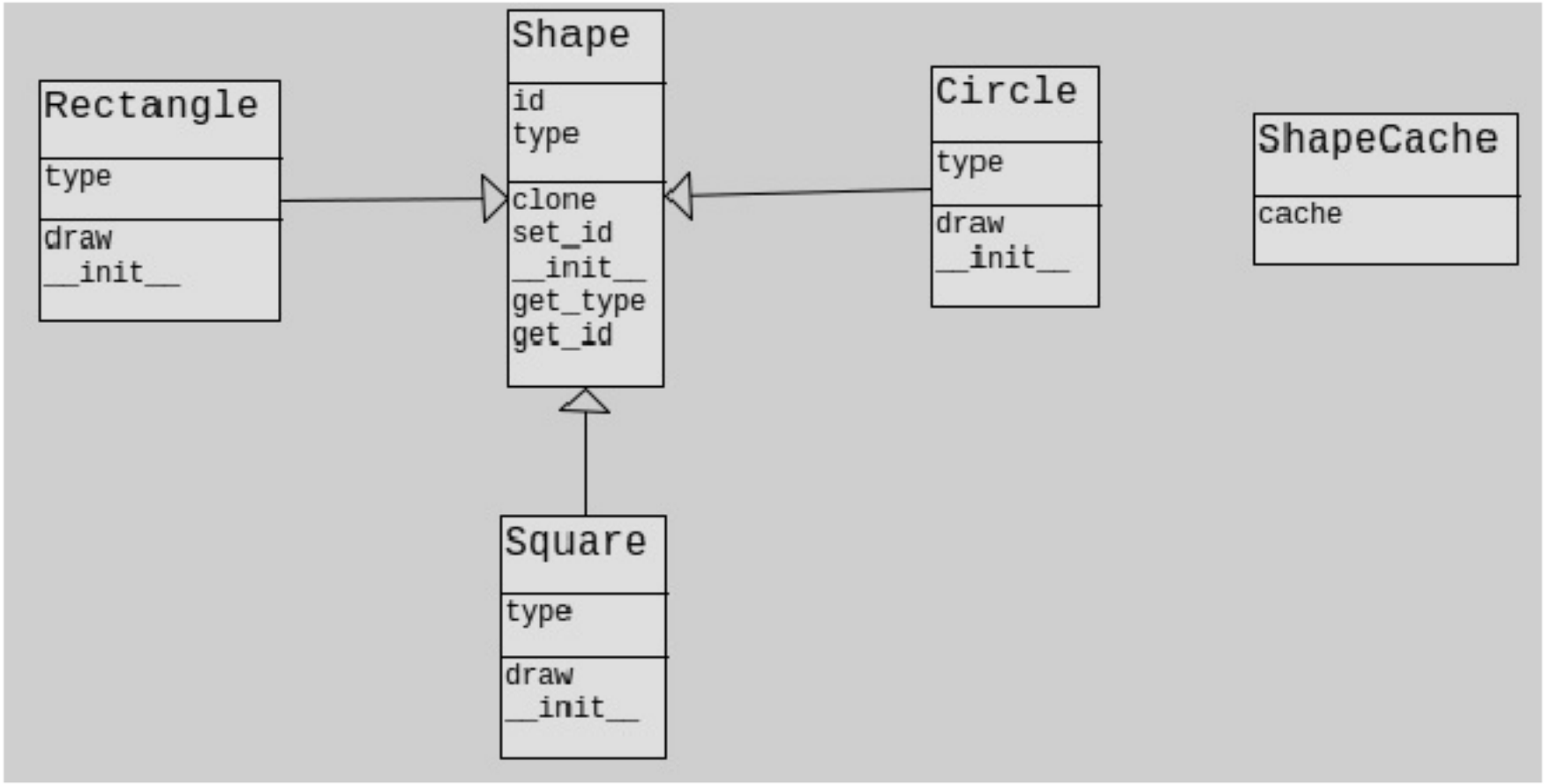


Model constructor - implementare concretă

```
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

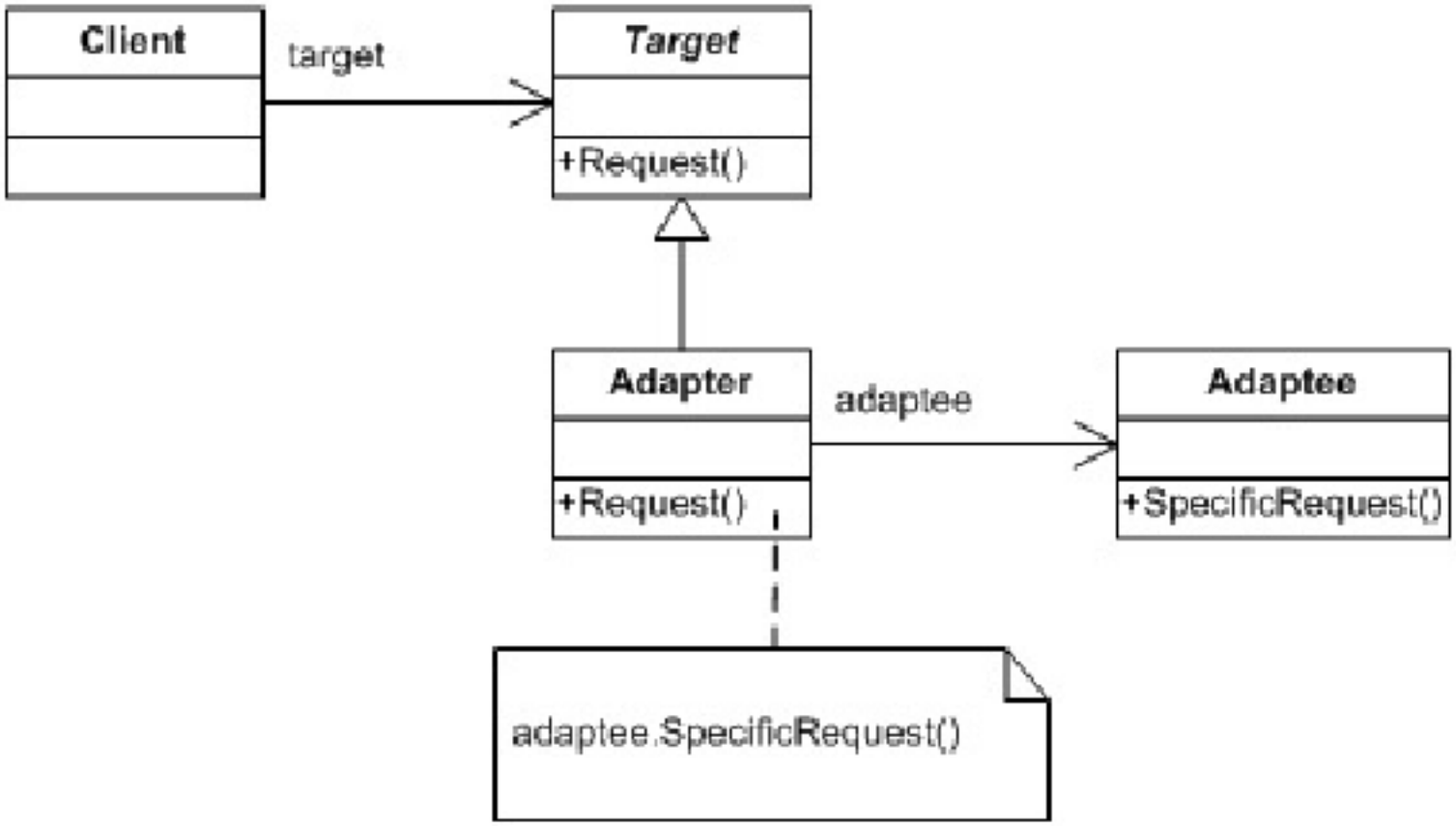


Model protip - implementare de caz

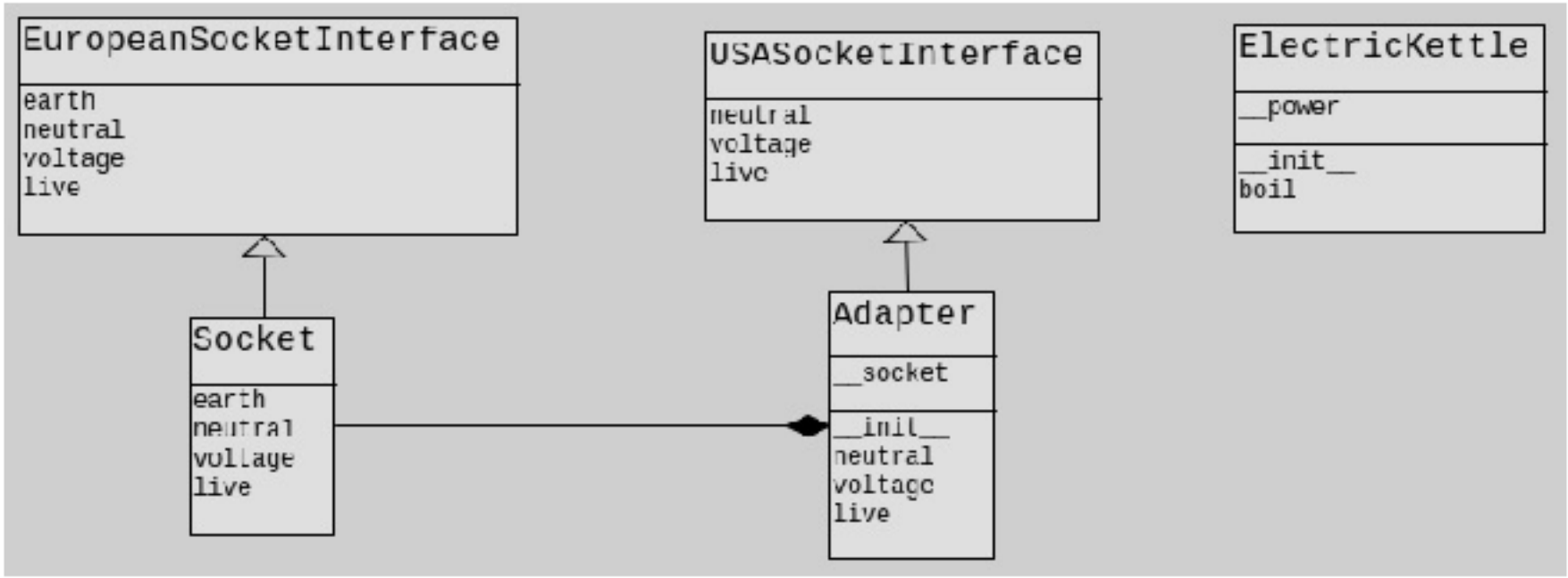
```
import abc
import copy
class Shape(metaclass=abc.ABCMeta):
    def __init__(self):
        self.id = None
        self.type = None
    @abc.abstractmethod
    def draw(self):
        pass
    def get_type(self):
        return self.type
    def get_id(self):
        return self.id
    def set_id(self, 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

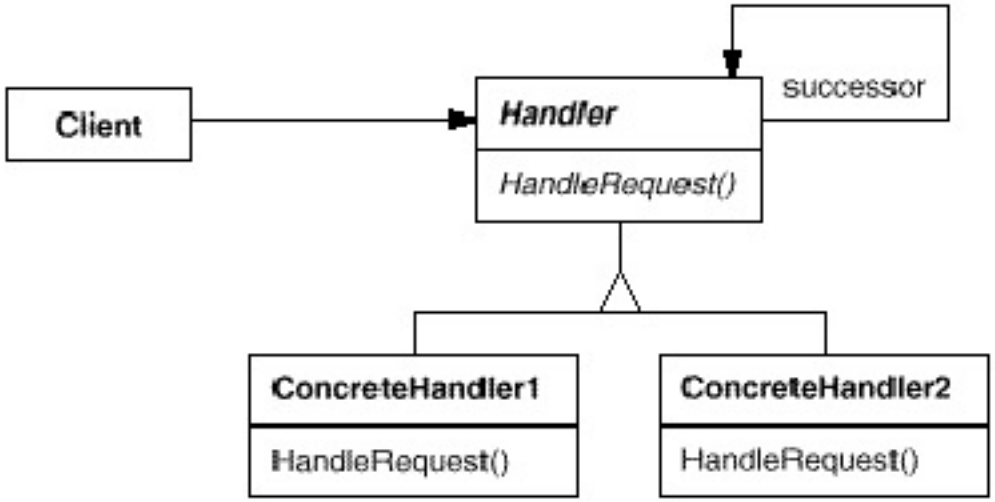


Model Adaptor - implementare

```
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()
```

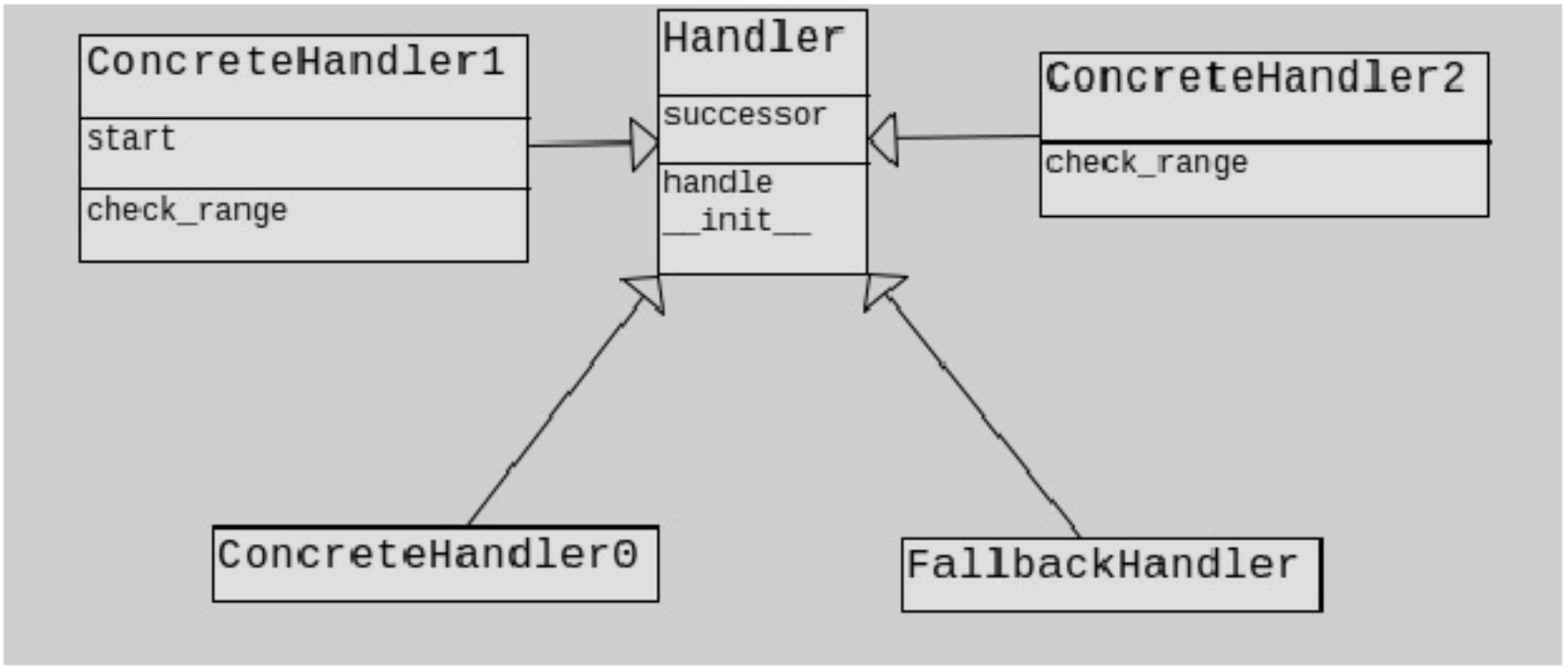
```
class ElectricKettle:# Client
    __power = None
    def __init__(self, power):
        self.__power = power
    def boil(self):
        if self.__power.voltage() > 110:
            print("Kettle on fire!")
        else:
            if self.__power.live() == 1 and \
                    self.__power.neutral() == -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 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)
```