Design Patterns

O que é um padrão de projeto (Design Pattern)

- Padrão de projeto é uma solução já conhecida e testada para uma classe de problemas.
- Origem:
  - Padrões de arquitetura. (Christofer Alexander)
  - Kent Beck e Ward Cunningham aplicaram esta idéia para programação em 1987.
  - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*
- http://www.oodesign.com/

Modelo

- Intenção:
  - Por que ele foi criado?
- Motivação:
  - Descrição de um problema e como as classes e objetos do padrão resolvem o problema.
- Consequências:
  - Benefícios e custos do padrão.
- Implementação:
  - Questões relativas a implementação em uma linguagem específica.

Tipos de padrões

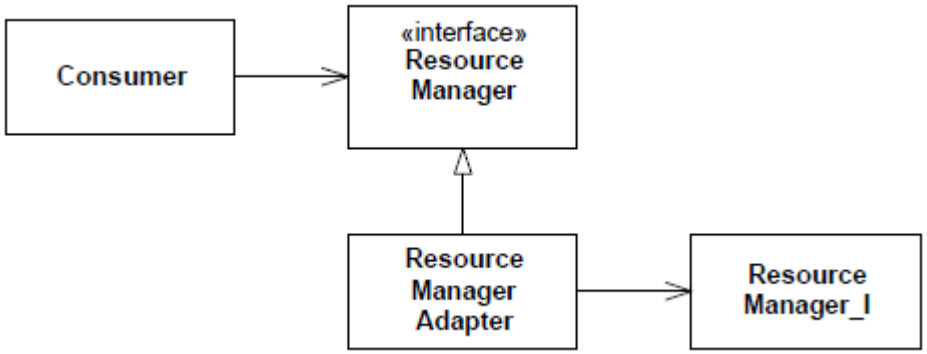
- Padrões para estruturação.
  - Composição de classes ou objetos
- Padrões para comportamento.
  - Como classes e objetos interagem e distribuem responsabilidades entre si.
- Padrões para criação.
  - Meios para instanciação de objetos.

Padrões para Estruturação

- Adaptador
- Façade
- Ponte
- Proxy
- Decorador
- Compósito

Adaptador

- Intenção:
  - Adaptar um modelo de interface para um outro modelo.
- Motivação:
  - Adaptação de software de terceiros para uso em um projeto.
- Implementação:



Exemplo em Python

```
# Adaptee (source) interface
class EuropeanSocketInterface:
    def voltage(self): pass

    def live(self): pass
    def neutral(self): pass
    def earth(self): pass

# Adaptee
class Socket(EuropeanSocketInterface):
    def voltage(self):
        return 230

    def live(self):
        return 1

    def neutral(self):
        return -1

    def earth(self):
        return 0
```

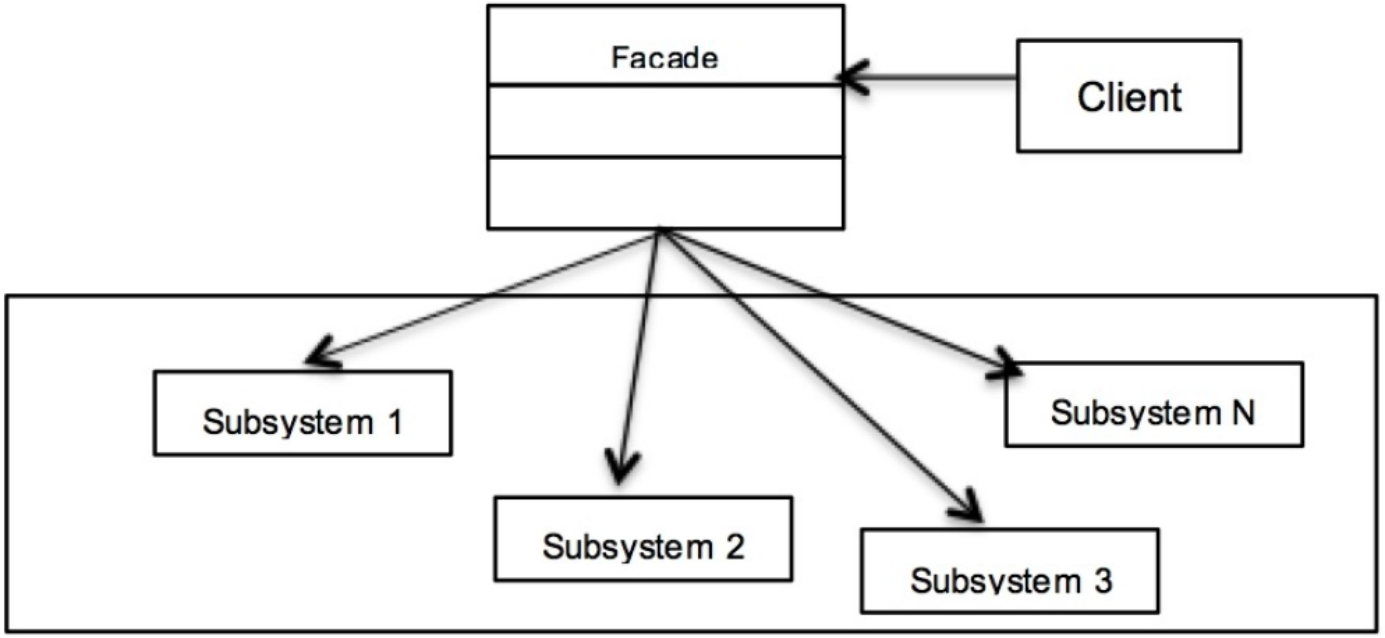
```
# Target interface
class USASocketInterface:
    def voltage(self): pass

    def live(self): pass
    def neutral(self): pass

# The Adapter
class Adapter(USASocketInterface):
    __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()
```

```
# Client
class ElectricKettle:
    __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():
    # Plug in
    socket  = Socket()
    adapter = Adapter(socket)
    kettle  = ElectricKettle(adapter)

    # Make coffee
    kettle.boil()

    return 0

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

Façade

- Intenção: É necessário ter uma interface simplificada para acessar um objeto complexo.
- Motivação: Façade provê uma única interface para um subsistema sem encapsular os diferentes componentes do subsistema.
- Implementação:



```
class You(object):
    def __init__(self):
        print("You:: Whoa! Marriage Arrangements??!!!")
    def askEventManager(self):
        print("You:: Let's Contact the Event Manager\n\n")
        em = EventManager()
        em.arrange()
    def __del__(self):
        print("You:: Thanks to Event Manager, all preparations done!
Phew!")

you = You()
you.askEventManager()
```

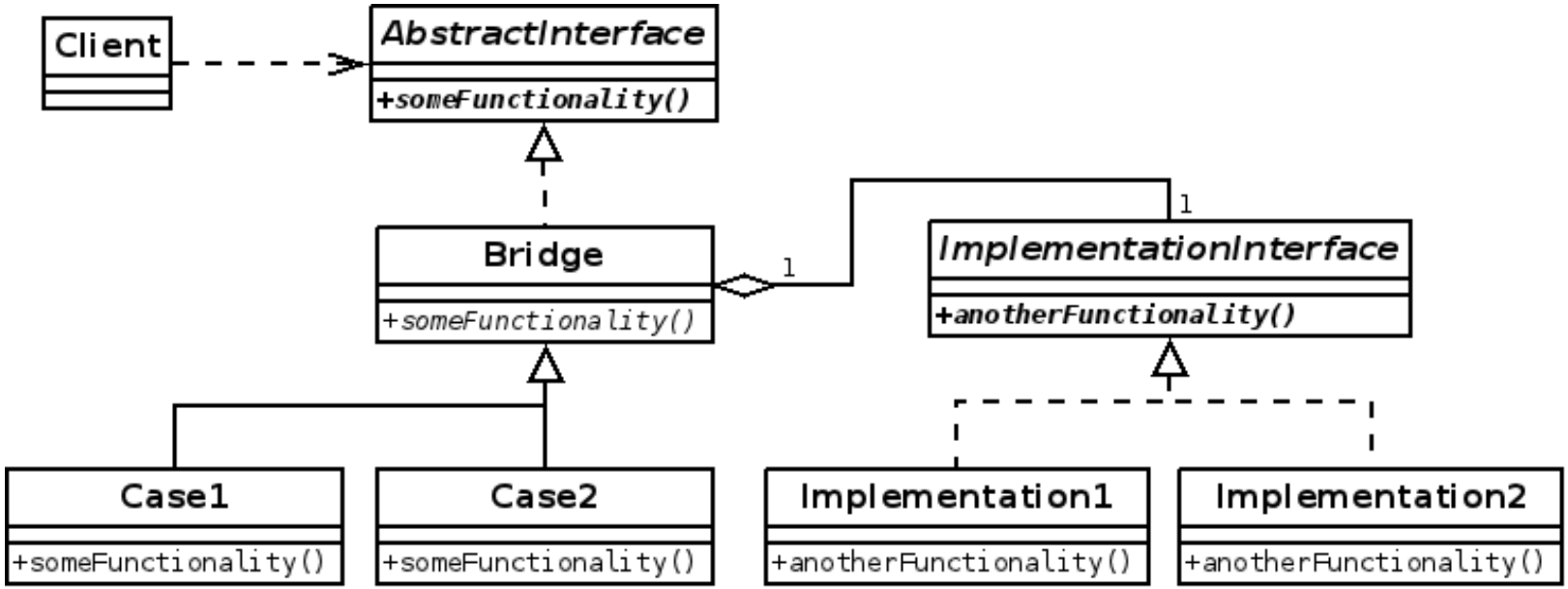
```
class EventManager(object):

    def __init__(self):
        print("Event Manager:: Let me talk to the folks\n")

    def arrange(self):
        self.hotelier = Hotelier()
        self.hotelier.bookHotel()

        self.florist = Florist()
        self.florist.setFlowerRequirements()

        self.caterer = Caterer()
        self.caterer.setCuisine()

        self.musician = Musician()
        self.musician.setMusicType()
```

```
class Hotelier(object):
    def __init__(self):
        print("Arranging the Hotel for Marriage? --")

    def __isAvailable(self):
        print("Is the Hotel free for the event on given day?")
        return True

    def bookHotel(self):
        if self.__isAvailable():
            print("Registered the Booking\n\n")


class Florist(object):
    def __init__(self):
        print("Flower Decorations for the Event? --")

    def setFlowerRequirements(self):
        print("Carnations, Roses and Lilies would be used for
Decorations\n\n")
```

```
class Caterer(object):
    def __init__(self):
        print("Food Arrangements for the Event --")

    def setCuisine(self):
        print("Chinese & Continental Cuisine to be served\n\n")


class Musician(object):
    def __init__(self):
        print("Musical Arrangements for the Marriage --")

    def setMusicType(self):

        print("Jazz and Classical will be played\n\n")
```

Ponte (Bridge)

- Intenção:
  - Podem existir várias opções para uma abstração, cada opção por sua vez pode utilizar diferentes implementações.
- Motivação:
  - A separação facilita a introdução de novas implementações sem alterar as abstrações (e vice-versa).
- Implementação:



```
class AbstractInterface:

    """ Target interface.

    This is the target interface, that clients use.
    """

    def someFunctionality(self):
        raise NotImplemented()

class Bridge(AbstractInterface):

    """ Bridge class.

    This class forms a bridge between the target
    interface and background implementation.
    """

    def __init__(self):
        self.__implementation = None
```

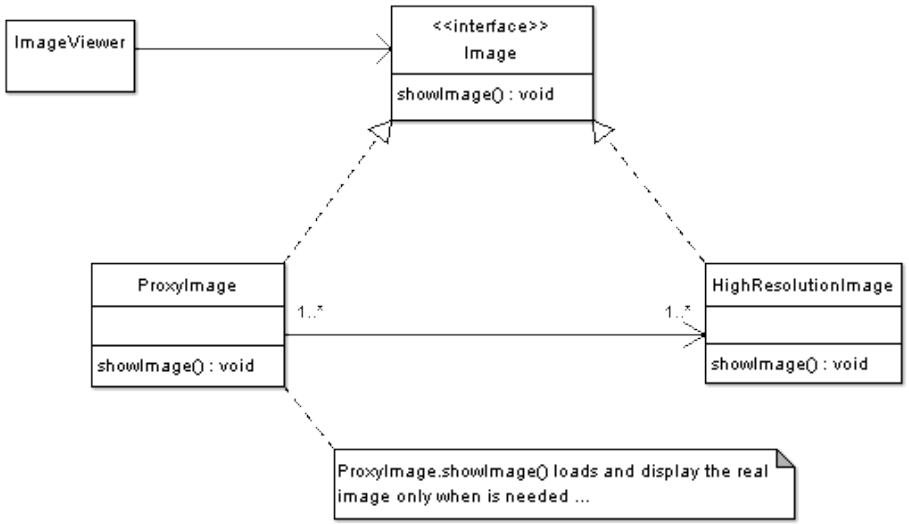
```
class UseCase1(Bridge):

    """ Variant of the target interface.

    This is a variant of the target Abstract interface.
    It can do something little differently and it can
    also use various background implementations through
    the bridge.
    """

    def __init__(self, implementation):
        self.__implementation = implementation

    def someFunctionality(self):
        print "UseCase1: ",
        self.__implementation.anotherFunctionality()

class UseCase2(Bridge):
    def __init__(self, implementation):
        self.__implementation = implementation

    def someFunctionality(self):
        print "UseCase2: ",
        self.__implementation.anotherFunctionality()
```

```
class ImplementationInterface:

    """ Interface for the background implementation.

    This class defines how the Bridge communicates
    with various background implementations.
    """

    def anotherFunctionality(self):
        raise NotImplemented

class Linux(ImplementationInterface):

    """ Concrete background implementation.

    A variant of background implementation, in this
    case for Linux!
    """

    def anotherFunctionality(self):
        print "Linux!"

class Windows(ImplementationInterface):
    def anotherFunctionality(self):
        print "Windows."
```

```
def main():
    linux = Linux()
    windows = Windows()

    # Couple of variants under a couple
    # of operating systems.
    useCase = UseCase1(linux)
    useCase.someFunctionality()

    useCase = UseCase1(windows)
    useCase.someFunctionality()

    useCase = UseCase2(linux)
    useCase.someFunctionality()

    useCase = UseCase2(windows)
    useCase.someFunctionality()
```

Delegado (Proxy)

- Intenção: controlar um objeto sem necessitar acessar todo o conteúdo do objeto.
- Motivação: Permitir a criação de clientes para um objeto complexo. O cliente pode acessar apenas algumas partes mais simples do objeto.
- Implementação:



Tipos de proxy

- **Proxy remoto**: o cliente age como um representante local de um serviço remoto.
- **Lazy Proxy**: o cliente adota a estratégia de executar um serviço completo apenas quando necessário.
- **Protective proxy**: o cliente é usado como uma barreira de proteção para um objeto sensitivo.
- **Proxy inteligente**: executa ações adicionais quando o objeto é acionado.

```
class You:
    def __init__(self):
        print("You:: Lets buy the Denim shirt")
        self.debitCard = DebitCard()
        self.isPurchased = None

    def make_payment(self):
        self.isPurchased = self.debitCard.do_pay()

    def __del__(self):
        if self.isPurchased:
            print("You:: Wow! Denim shirt is Mine :-)")
        else:
            print("You:: I should earn more :(")

you = You()
you.make_payment()
```

Interface para o serviço

```
from abc import ABCMeta, abstractmethod

class Payment(metaclass=ABCMeta):

    @abstractmethod
    def do_pay(self):
        pass
```

```
class Bank(Payment):

    def __init__(self):
        self.card = None
        self.account = None

    def __getAccount(self):
        self.account = self.card # Assume card number is account number
        return self.account

    def __hasFunds(self):
        print("Bank:: Checking if Account", self.__getAccount(), "has
enough funds")
        return True

    def setCard(self, card):
        self.card = card

    def do_pay(self):
        if self.__hasFunds():
            print("Bank:: Paying the merchant")
            return True
        else:
            print("Bank:: Sorry, not enough funds!")
            return False
```

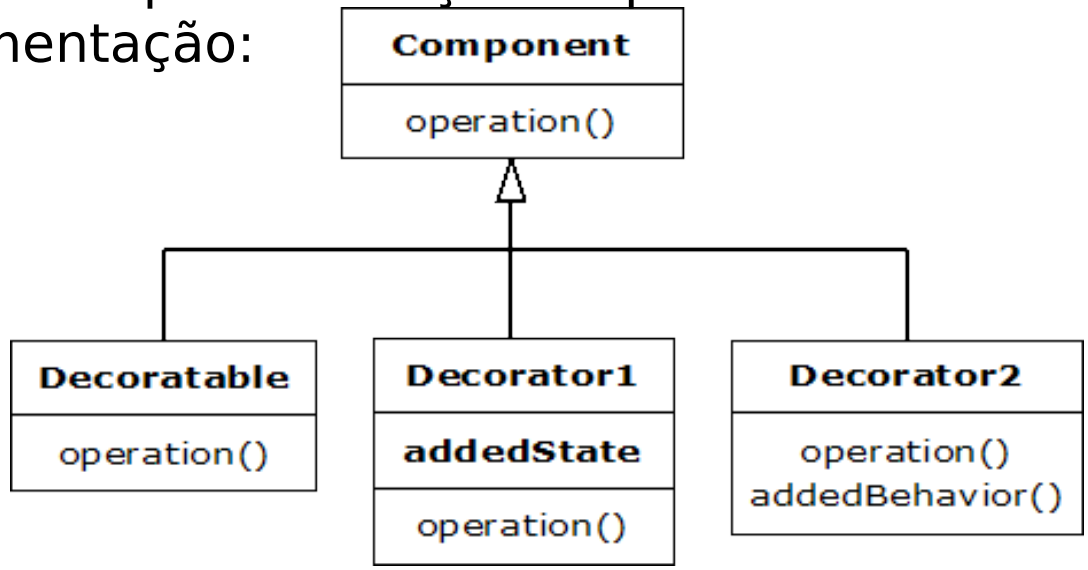
Classe Proxy

```
class DebitCard(Payment):

    def __init__(self):
        self.bank = Bank()

    def do_pay(self):
        card = input("Proxy:: Punch in Card Number: ")
        self.bank.setCard(card)
        return self.bank.do_pay()
```

```
You:: Lets buy the Denim shirt
Proxy:: Punch in Card Number: 23-2134-222
Bank:: Checking if Account 23-2134-222 has enough funds
Bank:: Paying the merchant
You:: Wow! Denim shirt is Mine :-)
```

Decorador

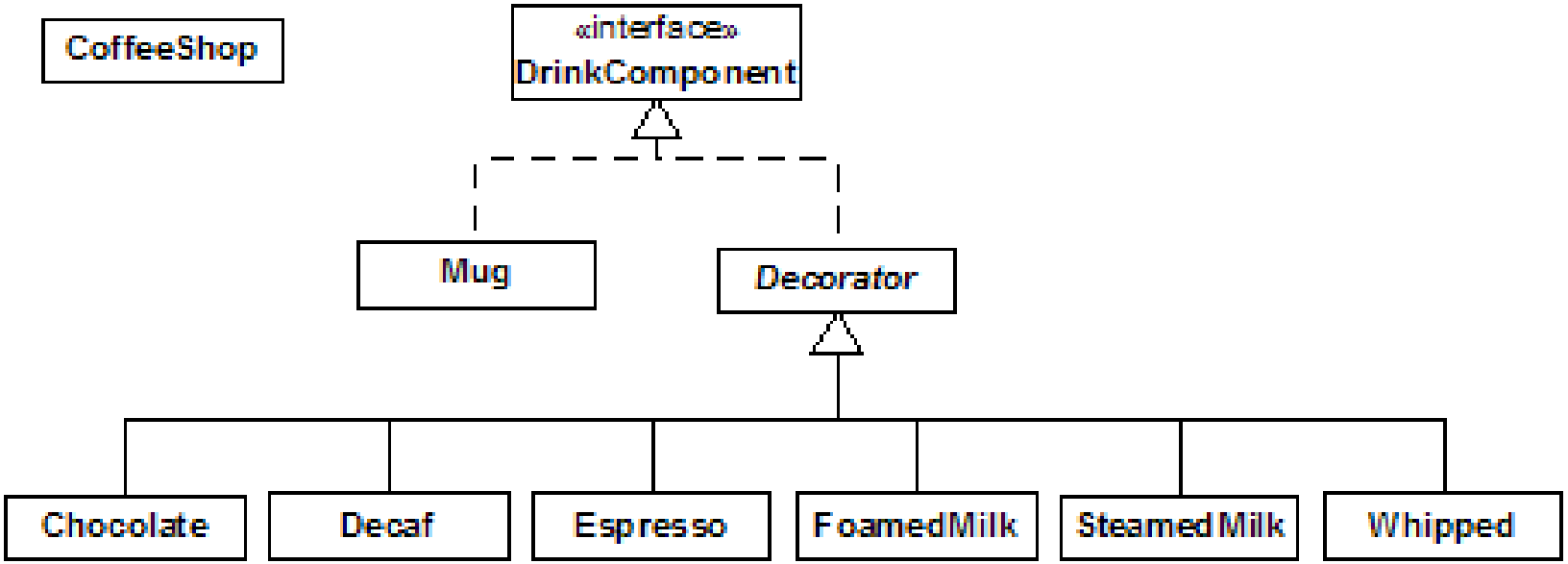
- Intenção: Uma classe pode ter muitas variações. Implementar uma subclasse para cada variação é improdutivo.
- Motivação: Criar uma abstração para as diferentes opções. E permitir que estas opções possam ser combinadas para a criação do produto final.
- Implementação:



Coffee Shop


CoffeeShop
Espresso
DoubleEspresso
Cappuccino
CappuccinoDecaf
CappuccinoDecafWhipped
CappuccinoWhipped
CappuccinoDry
CappuccinoExtraEspresso
CappuccinoExtraEspressoWhipped
CappuccinoDryWhipped
CafeMocha
CafeMochaDecaf
CafeMochaDecafWhipped
CafeMochaWhipped
CafeMochaWet
CafeMochaExtraEspresso
CafeMochaExtraEspressoWhipped
CafeMochaWetWhipped
CafeLatte
CafeLatteDecaf
CafeLatteDecafWhipped
CafeLatteWhipped
CafeLatteWet
CafeLatteExtraEspresso
CafeLatteExtraEspressoWhipped
CafeLatteWetWhipped

Implementação do Coffe Shop com Decorador



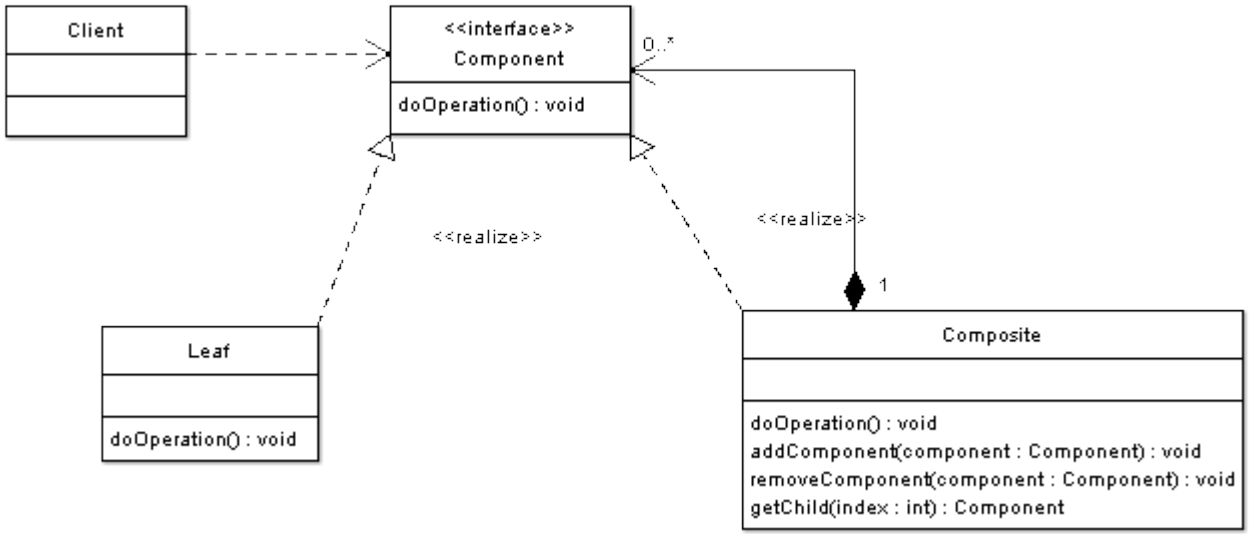
```
# Decorator/alldecorators/CoffeeShop.py
# Coffee example using decorators

class DrinkComponent:
    def getDescription(self):
        return self.__class__.__name__
    def getTotalCost(self):
        return self.__class__.cost

class Mug(DrinkComponent):
    cost = 0.0

class Decorator(DrinkComponent):
    def __init__(self, drinkComponent):
        self.component = drinkComponent
    def getTotalCost(self):
        return self.component.getTotalCost() +
DrinkComponent.getTotalCost(self)
    def getDescription(self):
        return self.component.getDescription() + '
'+DrinkComponent.getDescription(self)
```

```
class Espresso(Decorator):
    cost = 0.75
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Decaf(Decorator):
    cost = 0.0
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class FoamedMilk(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class SteamedMilk(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Whipped(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)

class Chocolate(Decorator):
    cost = 0.25
    def __init__(self, drinkComponent):
        Decorator.__init__(self, drinkComponent)
```

```
cappuccino = Espresso(FoamedMilk(Mug()))
print(cappuccino.getDescription()+ ": $" +
cappuccino.getTotalCost())

cafeMocha =
Espresso(SteamedMilk(Chocolate( Whipped(Decaf(Mug())))))

print(cafeMocha.getDescription()+ ": $" +
cafeMocha.getTotalCost())
```

Compósito

- Intenção: Muitas vezes é necessário a manipulação de árvores de objetos com ramos e folhas tratados de forma uniforme.
- Motivação: Compor objetos em estruturas de árvores formando hierarquias de partes.
- Implementação:



```
class Component(object):
    def __init__(self, *args, **kw):
        pass

    def component_function(self):
        pass


class Leaf(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self,
*args, **kw)

    def component_function(self):
        print( "some function“)
```

```
class Composite(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self, *args, **kw)
        self.children = []

    def append_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        self.children.remove(child)

    def component_function(self):
        map(lambda x: x.component_function(),
self.children)
```

```
c = Composite()
l = Leaf()
l_two = Leaf()
c.append_child(l)
c.append_child(l_two)
c.component_function()
```

Exercicio

- Criar exemplos que ilustrem os padrões Ponte, Decorador e Compósito.