Design Patterns

MAS Foundations Course WS-19
Hochschule Bonn-Rhein-Sieg

Lokesh Veeramacheneni

Outline

- What are Design Patterns?
- Strategy Pattern
- Decorator Pattern
- Singleton Pattern

Design Pattern

- Design Patterns are used to represent the flow used by developers for the development of a particular software.

Why Design Patterns ?

- Design Patterns describe about
  - Solution to the problem.
  - When and where to apply that solution.
  - Consequences of applying that solution.

Design Patterns (cont'd)

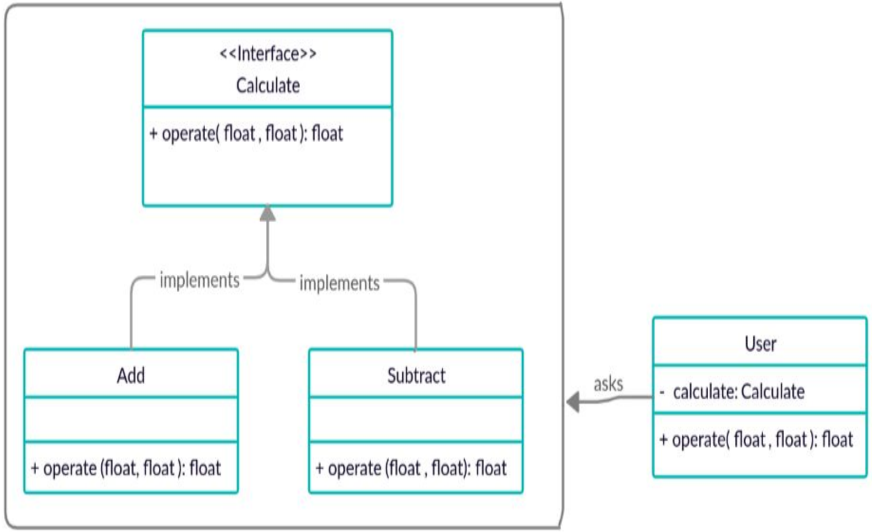
- Design Patterns are independent of language of implementation.
- Types of Design Patterns:
  - **Creational Design Patterns** - more concern on the way of object creation.
  - **Structural Design Patterns** - gives importance on relations between the classes and simplifying the structure
  - **Behavioral Patterns** - concern on the interaction of objects and their responsibilities.
- Some Design Patterns:
  - Singleton
  - Factory
  - Prototype
  - Decorator
  - Flyweight, etc...

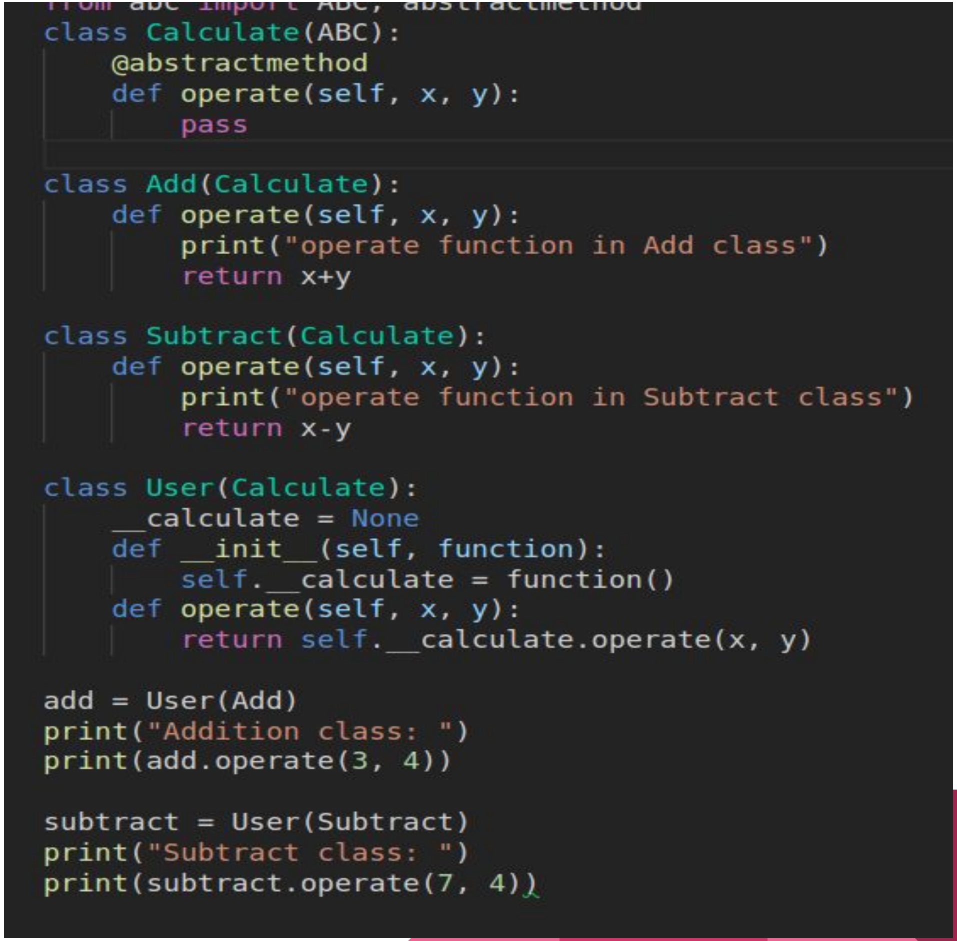
Strategy Pattern

- Behavioral Pattern
- It enables the user to use different algorithms for a particular task.
- Also known as **Policy**.
- Advantage is ease of incorporation of new behaviours into the existing code.

Strategy Pattern (cont'd)

UML Diagram:


<<Interface>>
Calculate
+ operate(float , float): float
implements
implements
Add
+ operate (float, float): float
Subtract
+ operate (float , float): float
asks
User
- calculate: Calculate
+ operate(float , float): float



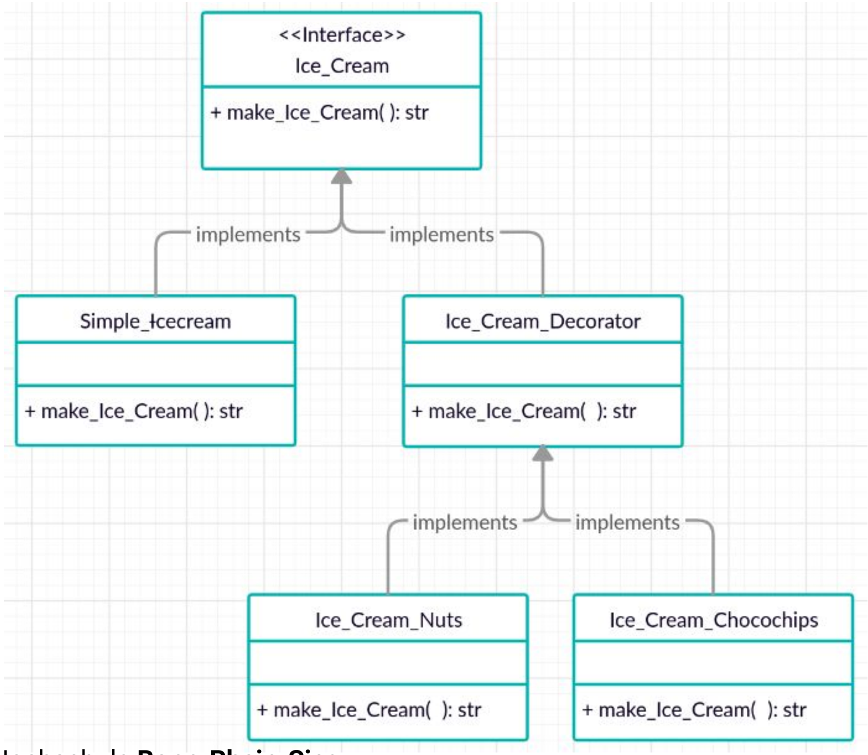
```python
from abc import ABC, abstractmethod
class Calculate(ABC):
    @abstractmethod
    def operate(self, x, y):
        pass

class Add(Calculate):
    def operate(self, x, y):
        print("operate function in Add class")
        return x+y

class Subtract(Calculate):
    def operate(self, x, y):
        print("operate function in Subtract class")
        return x-y

class User(Calculate):
    __calculate = None
    def __init__(self, function):
        self.__calculate = function()
    def operate(self, x, y):
        return self.__calculate.operate(x, y)

add = User(Add)
print("Addition class: ")
print(add.operate(3, 4))

subtract = User(Subtract)
print("Subtract class: ")
print(subtract.operate(7, 4))
```

Decorator Pattern

- Structural Pattern
- It is used in the scenarios where the object is added with extra responsibilities dynamically.
- For example let object is used to make Ice Cream an additional seasoning is added as decorator for that object.
- Also called as **Wrapper**.

Decorator Pattern (cont'd)

UML Diagram:



<<Interface>>
Ice_Cream
+ make_Ice_Cream( ): str

implements

implements

Simple_Icecream
+ make_Ice_Cream( ): str

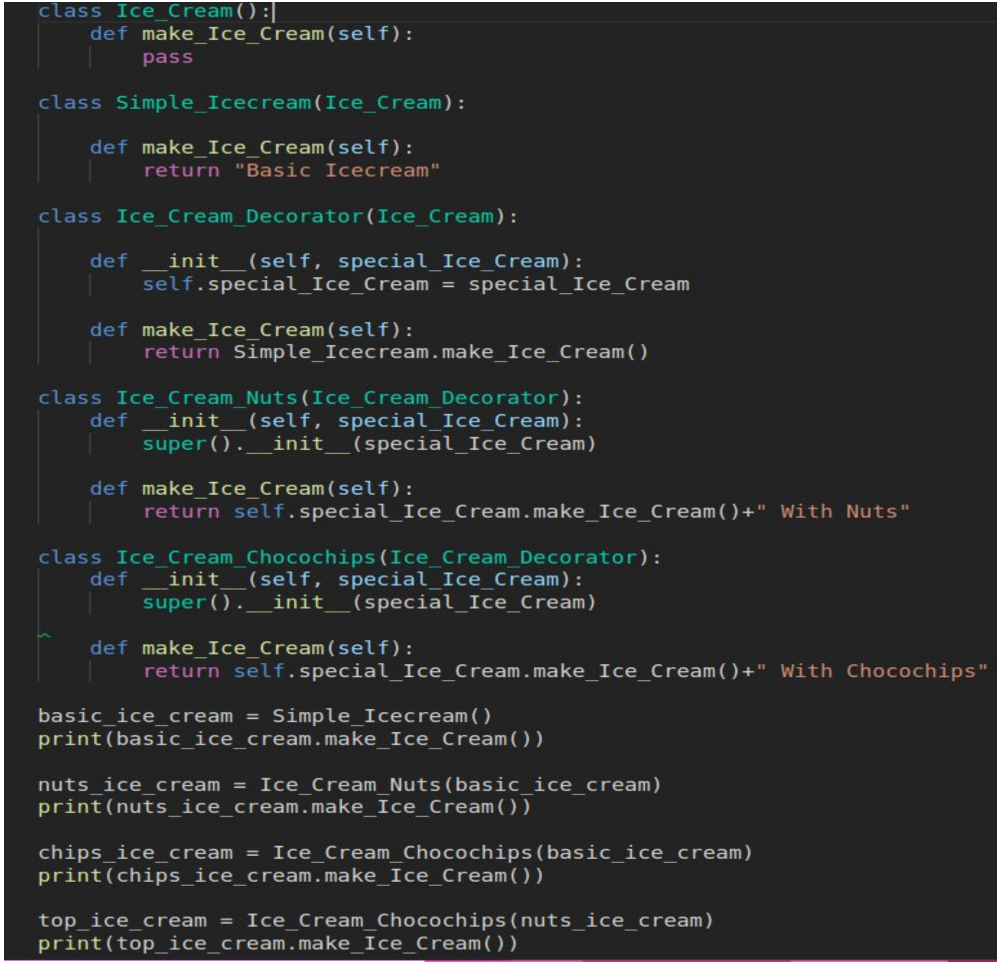
Ice_Cream_Decorator
+ make_Ice_Cream( ): str

implements

implements

Ice_Cream_Nuts
+ make_Ice_Cream( ): str

Ice_Cream_Chocochips
+ make_Ice_Cream( ): str



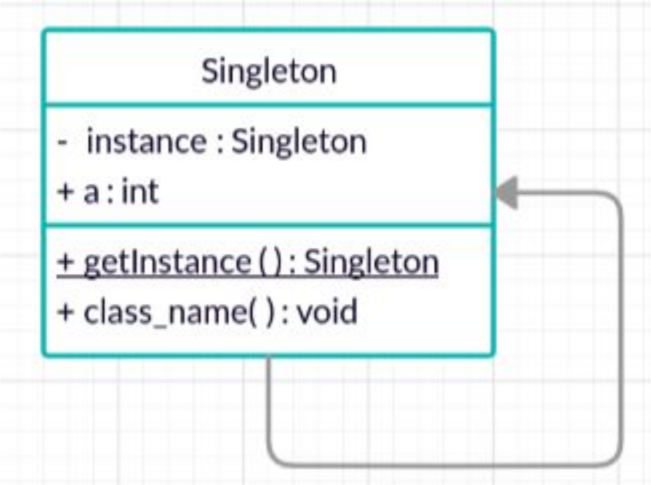
```
class Ice_Cream():
    def make_Ice_Cream(self):
        pass

class Simple_Icecream(Ice_Cream):

    def make_Ice_Cream(self):
        return "Basic Icecream"

class Ice_Cream_Decorator(Ice_Cream):

    def __init__(self, special_Ice_Cream):
        self.special_Ice_Cream = special_Ice_Cream

    def make_Ice_Cream(self):
        return Simple_Icecream.make_Ice_Cream()

class Ice_Cream_Nuts(Ice_Cream_Decorator):
    def __init__(self, special_Ice_Cream):
        super().__init__(special_Ice_Cream)

    def make_Ice_Cream(self):
        return self.special_Ice_Cream.make_Ice_Cream()+" With Nuts"

class Ice_Cream_Chocochips(Ice_Cream_Decorator):
    def __init__(self, special_Ice_Cream):
        super().__init__(special_Ice_Cream)

    def make_Ice_Cream(self):
        return self.special_Ice_Cream.make_Ice_Cream()+" With Chocochips"

basic_ice_cream = Simple_Icecream()
print(basic_ice_cream.make_Ice_Cream())

nuts_ice_cream = Ice_Cream_Nuts(basic_ice_cream)
print(nuts_ice_cream.make_Ice_Cream())

chips_ice_cream = Ice_Cream_Chocochips(basic_ice_cream)
print(chips_ice_cream.make_Ice_Cream())

top_ice_cream = Ice_Cream_Chocochips(nuts_ice_cream)
print(top_ice_cream.make_Ice_Cream())
```

Singleton Pattern

- Creational Pattern
- This pattern allows the single instance of the class and this instance acts as global access point.
- This pattern is used when memory requirements are strict and it widely used in multi-threading and database applications such as logging, caching, etc.
- Based on time of creation of instance , Singleton Pattern can be divided into two forms:
  - **Early Instantiation:** Instances are created at time of loading.
  - **Lazy Instantiation:** Instances are created when required.

Singleton Pattern (cont'd)

UML Diagram:



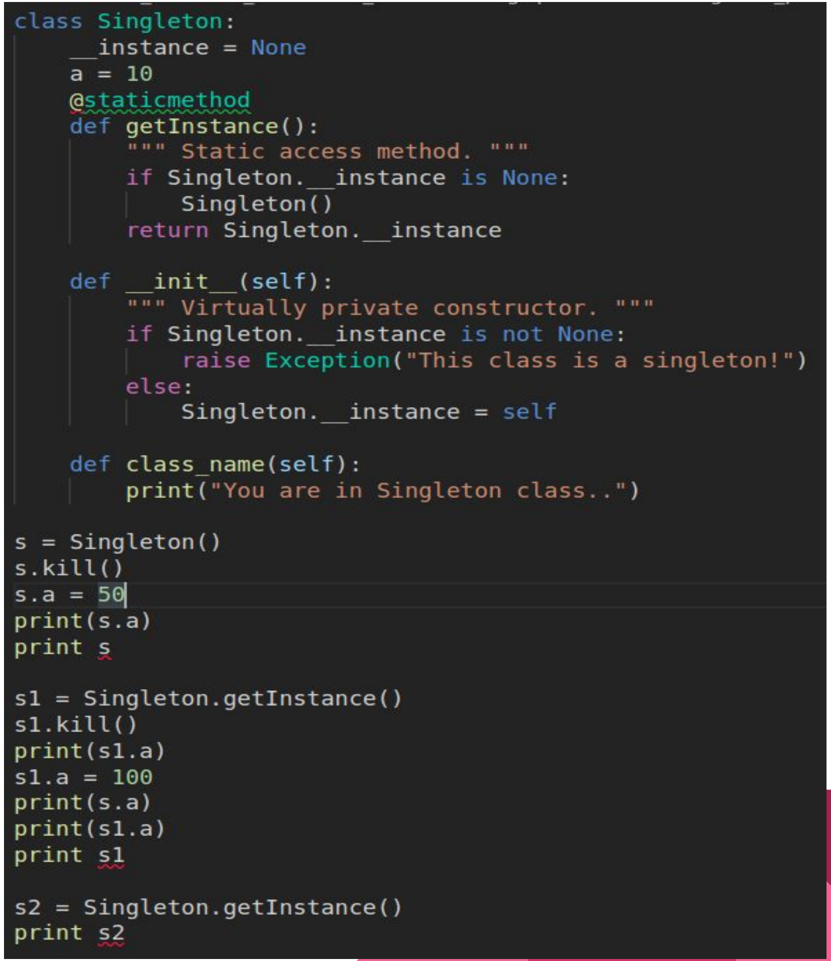
Singleton

- instance : Singleton

+ a : int

+ getInstance ( ) : Singleton

+ class_name( ) : void



```
class Singleton:
    __instance = None
    a = 10
    @staticmethod
    def getInstance():
        """ Static access method. """
        if Singleton.__instance is None:
            Singleton()
        return Singleton.__instance

    def __init__(self):
        """ Virtually private constructor. """
        if Singleton.__instance is not None:
            raise Exception("This class is a singleton!")
        else:
            Singleton.__instance = self

    def class_name(self):
        print("You are in Singleton class..")

s = Singleton()
s.kill()
s.a = 50
print(s.a)
print s

s1 = Singleton.getInstance()
s1.kill()
print(s1.a)
s1.a = 100
print(s.a)
print(s1.a)
print s1

s2 = Singleton.getInstance()
print s2
```

References:

- Learn Design Patterns for absolute beginners - tutorialspoint
- Design Patterns in Java - Javatpoint.
- Design Patterns - Christopher Okhravi.