Design Patterns is the most essential part of Software Engineering, as they provide the general repeatable solution to a commonly occurring problem in software design. They usually represent some of the best practices adopted by experienced object-oriented software developers.

We can not consider the **Design Patterns** as the finished design that can be directly converted into code. They are only templates that describe how to solve a particular problem with great efficiency. To know more about design patterns basics, refer – Introduction to Design Patterns.

Classification of Design Patterns

Creational Design Pattern:

Creational patterns provides essential information regarding the Class instantiation or the object instantiation. Class Creational Pattern and the Object Creational pattern is the major categorization of the Creational Design Patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Classification of Creational Design Patterns –

- Factory Method
- Abstract Factory Method
- Builder Method
- Prototype Method
- Singleton Method

Structural Design Patterns:

Structural design patterns are about organizing different classes and objects to form larger structures and provide new functionality while keeping these structures flexible and efficient. Mostly they use Inheritance to compose all the interfaces. It also identifies the relationships which led to the simplification of the structure.

Classification of Structural Design Patterns –

- Adapter Method
- Bridge Method
- Composite Method
- Decorator Method
- Facade Method
- Proxy Method
- FlyWeight Method

Behavioral Design Pattern:

Behavioral patterns are all about identifying the common communication patterns between objects and realize these patterns. These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Classification of Behavioral Design Patterns –

- Chain of Responsibility Method
- Command Method
- Iterator Method
- Mediator Method
- Memento Method
- Observer Method
- State Method
- Strategy Method
- Template Method
- Visitor Method

Advantages of using Design Patterns

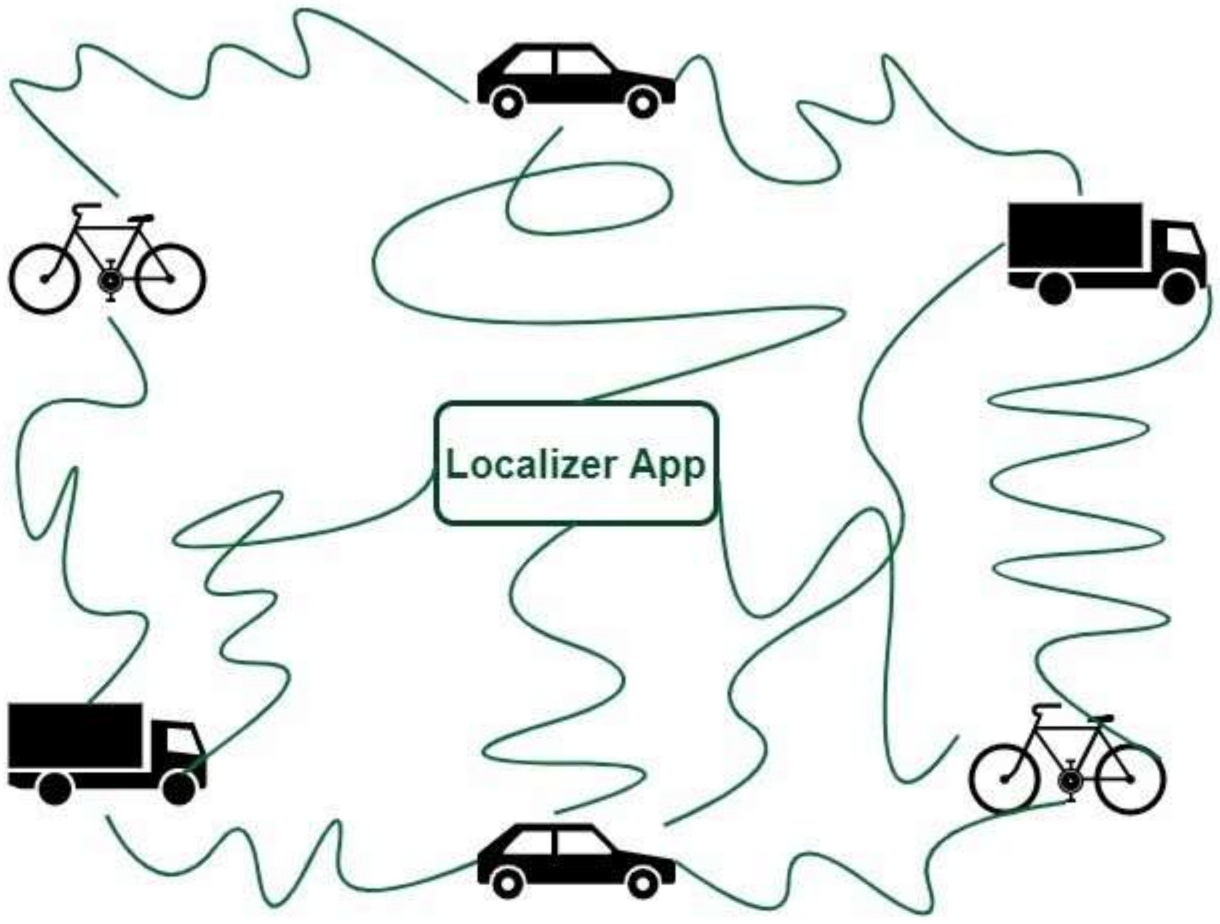
- **Reusability:** By using Inheritance, they helps in making the code reusable and hence we can use them in multiple projects.
- **Transparent:** It improves the transparency of the code for all the developers who are going to use it in future.
- **Established Solution:** We can blindly believe on the solution provided by **Design Patterns** as they are well-proved and testified at critical stages.
- **Established Communication:** Design patterns make communication between designers more efficient. Software professionals can immediately picture the high-level design in their heads when they refer the name of the pattern used to solve a particular issue when discussing system design.
- **Efficient Development:** Design patterns helps in the development of the highly cohesive modules with minimal coupling.

Factory Method – Python Design Patterns

Factory Method is a Creational Design Pattern that allows an interface or a class to create an object, but lets subclasses decide which class or object to instantiate. Using the Factory method, we have the best ways to create an object. Here, objects are created without exposing the logic to the client, and for creating the new type of object, the client uses the same common interface.

Problems we face without Factory Method:

Imagine you are having your own startup which provides ridesharing in different parts of the country. The initial version of the app only provides Two-Wheeler ridesharing but as time passes, your app becomes popular and now you want to add Three and Four-Wheeler ridesharing also.
It’s a piece of great news! but what about the software developers of your startup. They have to change the whole code because now most part of the code is coupled with the Two-Wheeler class and developers have to make changes to the entire codebase.
After being done with all these changes, either the developers end with the messy code or with the resignation letter.



Localizer app

Diagrammatic representation of Problems without using Factory Method

Let’s understand the concept with one more example which is related to the translations and localization of the different languages.
Suppose we have created an app whose main purpose is to translate one language into another and currently our app works with 10 languages only. Now our app has become widely popular among people but the demand has grown suddenly to include 5 more languages.
It’s a piece of great news! only for the owner not for the developers. They have to change the whole code because now most part of the code is coupled with the existing languages only and that’s why developers have to make changes to the entire codebase which is really a difficult task to do.
Let’s look at the code for the problem which we may face without using the factory method.

Note: Following code is written without using the Factory method.

- Python3

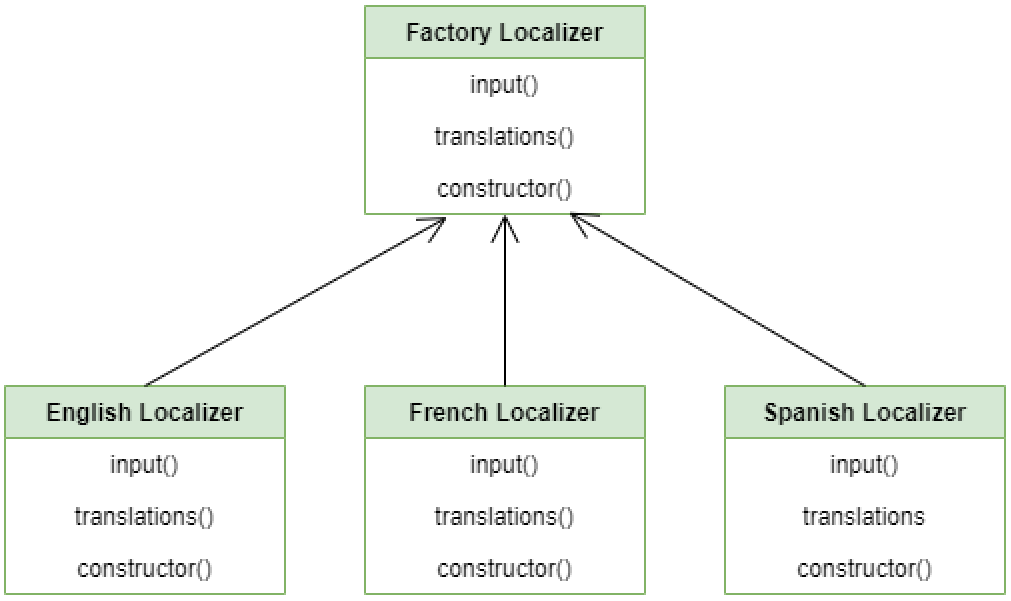
```
# Python Code for Object

# Oriented Concepts without

# using Factory method



class FrenchLocalizer:



    """ it simply returns the french version """



    def __init__(self):



        self.translations = {"car": "voiture", "bike": "bicyclette",

                             "cycle":"cyclette"}



    def localize(self, msg):



        """change the message using translations"""

        return self.translations.get(msg, msg)
```

```
class SpanishLocalizer:
    """it simply returns the spanish version"""

    def __init__(self):

        self.translations = {"car": "coche", "bike": "bicicleta",
                             "cycle":"ciclo"}

    def localize(self, msg):

        """change the message using translations"""
        return self.translations.get(msg, msg)

class EnglishLocalizer:
    """Simply return the same message"""

    def localize(self, msg):
        return msg

if __name__ == "__main__":
```

```
# main method to call others

f = FrenchLocalizer()

e = EnglishLocalizer()

s = SpanishLocalizer()



# list of strings

message = ["car", "bike", "cycle"]



for msg in message:

    print(f.localize(msg))

    print(e.localize(msg))

    print(s.localize(msg))
```

Solution by Factory Method:

Its solution is to replace the straightforward object construction calls with calls to the special factory method. Actually, there will be no difference in the object creation but they are being called within the **factory method**.
For example Our Two_Wheeler, Three_Wheeler, and Four_wheeler classes should implement the **ridesharing** interface which will declare a method called a **ride**. Each class will implement this method uniquely.



solution_factory_diagram

Now let us understand the factory method with the help of an example:

- Python3

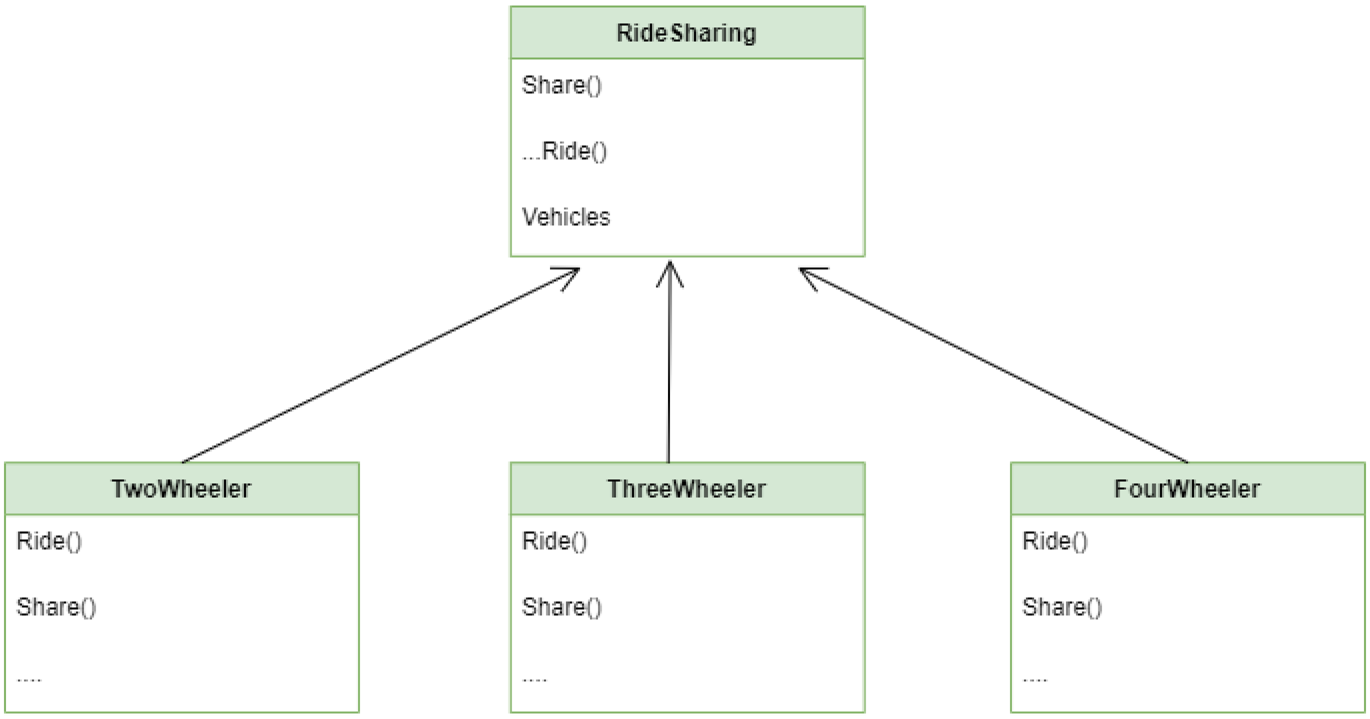
```
# Python Code for factory method

# it comes under the creational

# Design Pattern



class FrenchLocalizer:



    """ it simply returns the french version """



    def __init__(self):
        self.translations = {"car": "voiture", "bike": "bicyclette",

                             "cycle":"cyclette"}


    def localize(self, msg):


        """change the message using translations"""

        return self.translations.get(msg, msg)


class SpanishLocalizer:

    """it simply returns the spanish version"""


    def __init__(self):

        self.translations = {"car": "coche", "bike": "bicicleta",

                             "cycle":"ciclo"}


    def localize(self, msg):


        """change the message using translations"""

        return self.translations.get(msg, msg)
```

```
class EnglishLocalizer:

    """Simply return the same message"""



    def localize(self, msg):

        return msg



def Factory(language ="English"):



    """Factory Method"""

    localizers = {

        "French": FrenchLocalizer,

        "English": EnglishLocalizer,

        "Spanish": SpanishLocalizer,

    }



    return localizers[language]()



if __name__ == "__main__":
```

```
f = Factory("French")

e = Factory("English")

s = Factory("Spanish")


message = ["car", "bike", "cycle"]


for msg in message:

    print(f.localize(msg))

    print(e.localize(msg))

    print(s.localize(msg))
```

Class Diagram for Factory Method:

Let’s look at the class diagram considering the example of ride-sharing.



Factory_pattern_class_diagram

Advantages of using Factory method:

1. We can easily add new types of products without disturbing the existing client code.
2. Generally, tight coupling is being avoided between the products and the creator classes and objects.

Disadvantages of using Factory method:

1. To create a particular concrete product object, the client might have to sub-class the creator class.
2. You end up with a huge number of small files i.e, cluttering the files.
   - In a Graphics system, depending upon the user's input it can draw different shapes like rectangles, Square, Circle, etc. But for the ease of both developers as well as the client, we can use the factory method to create the instance depending upon the user's input. Then we don't have to change the client code for adding a new shape.
   - On a Hotel booking site, we can book a slot for 1 room, 2 rooms, 3 rooms, etc. Here user can input the number of rooms he wants to book. Using the factory method, we can create a factory class Any Rooms

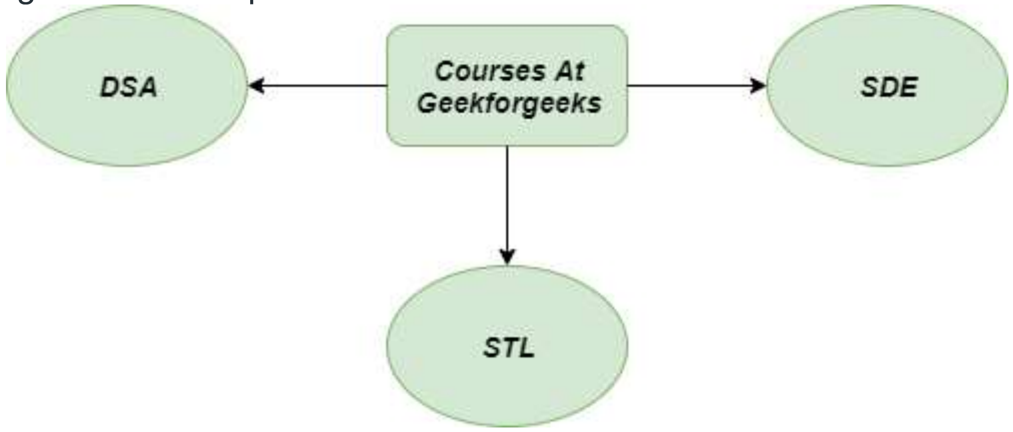
which will help us to create the instance depending upon the user's input. Again we don't have to change the client's code for adding the new facility.

Abstract Factory Method – Python Design Patterns

Abstract Factory Method is a Creational Design pattern that allows you to produce the families of related objects without specifying their concrete classes. Using the abstract factory method, we have the easiest ways to produce a similar type of many objects.
It provides a way to encapsulate a group of individual factories. Basically, here we try to abstract the creation of the objects depending on the logic, business, platform choice, etc.

Problem we face without Abstract Factory Method:

Imagine you want to join one of the elite batches of GeeksforGeeks. So, you will go there and ask about the Courses available, their Fee structure, their timings, and other important things. They will simply look at their system and will give you all the information you required. Looks simple? Think about the developers how they make the system so organized and how their website is so lubricative. Developers will make unique classes for each course which will contain its properties like Fee structure, timings, and other things. But ***how they will call them*** and ***how will they instantiate their objects*****?**
Here arises the problem, suppose initially there are only 3-4 courses available at GeeksforGeeks, but later they added 5 new courses.
So, we have to manually instantiate their objects which is not a good thing according to the developer's side.



Abstract Factory

Diagrammatic representation of Problems without using Abstract Factory Method

Note: Following code is written without using the abstract factory method

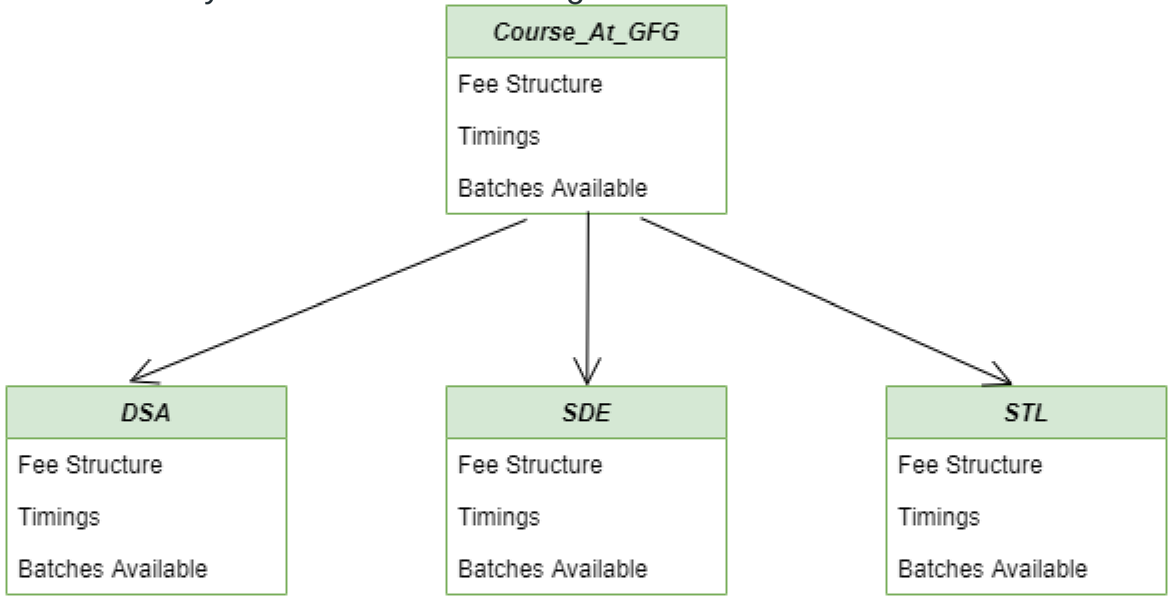
- Python3

```
# Python Code for object

# oriented concepts without

# using the Abstract factory

# method in class



class DSA:



    """ Class for Data Structure and Algorithms """



    def price(self):

        return 11000



    def __str__(self):

        return "DSA"



class STL:
    """Class for Standard Template Library"""

    def price(self):
        return 8000

    def __str__(self):
        return "STL"

class SDE:

    """Class for Software Development Engineer"""

    def price(self):
        return 15000

    def __str__(self):
        return 'SDE'

# main method
if __name__ == "__main__":
```

```
sde = SDE()    # object for SDE class

dsa = DSA()    # object for DSA class

stl = STL()    # object for STL class



print(f'Name of the course is {sde} and its price is {sde.price()}')

print(f'Name of the course is {dsa} and its price is {dsa.price()}')

print(f'Name of the course is {stl} and its price is {stl.price()}')
```

Solution by using Abstract Factory Method:

Its solution is to replace the straightforward object construction calls with calls to the special abstract factory method. Actually, there will be no difference in the object creation but they are being called within the factory method.
Now we will create a unique class whose name is **Course_At_GFG** which will handle all the object instantiation automatically. Now, we don't have to worry about how many courses we are adding after some time.



solution using abstract factory pattern

- Python3

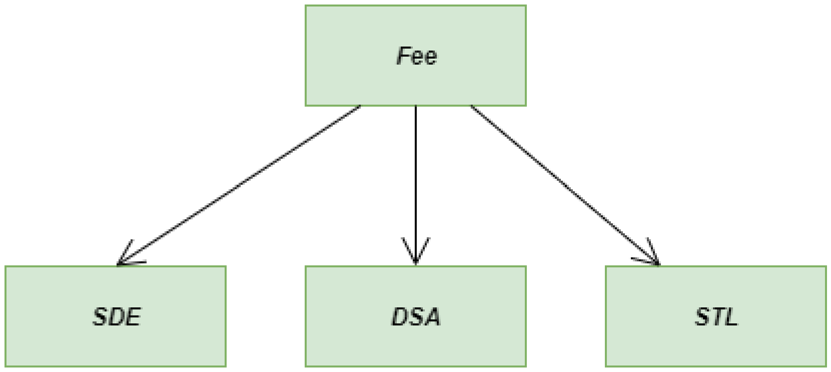
```
# Python Code for object

# oriented concepts using

# the abstract factory

# design pattern



import random



class Course_At_GFG:



    """ GeeksforGeeks portal for courses """



    def __init__(self, courses_factory = None):

        """course factory is out abstract factory"""



        self.course_factory = courses_factory



    def show_course(self):
        """creates and shows courses using the abstract factory"""

        course = self.course_factory()

        print(f'We have a course named {course}')
        print(f'its price is {course.Fee()}')


class DSA:

    """Class for Data Structure and Algorithms"""

    def Fee(self):
        return 11000

    def __str__(self):
        return "DSA"
```

```
class STL:

    """Class for Standard Template Library"""

    def Fee(self):
        return 8000

    def __str__(self):
        return "STL"

class SDE:

    """Class for Software Development Engineer"""

    def Fee(self):
        return 15000

    def __str__(self):
        return 'SDE'
```

```
def random_course():

    """A random class for choosing the course"""

    return random.choice([SDE, STL, DSA])()



if __name__ == "__main__":

    course = Course_At_GFG(random_course)

    for i in range(5):
        course.show_course()
```

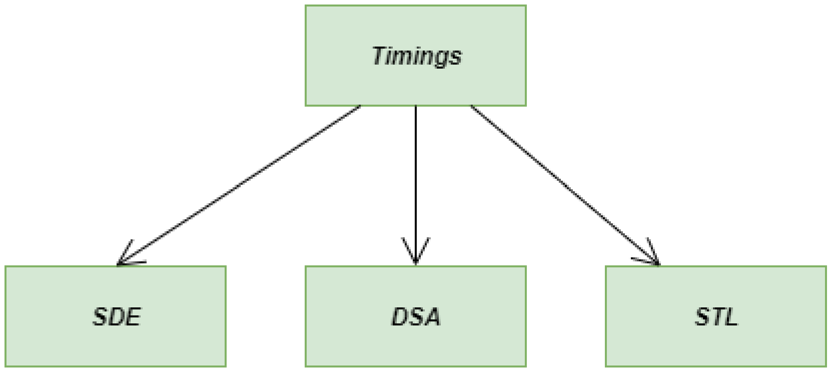
Class Diagram for Abstract Factory Method:

Let’s look at the class diagram considering the example of Courses at GeeksforGeeks.
Fee structure of all the available courses at GeeksforGeeks



class diagram for the abstract factory pattern

Timings of all the available courses at GeeksforGeeks



class diagram 2 abstract method pattern

Advantages of using Abstract Factory method:

This pattern is particularly useful when the client doesn't know exactly what type to create.

1. It is easy to introduce new variants of the products without breaking the existing client code.
2. Products which we are getting from the factory are surely compatible with each other.

Disadvantages of using Abstract Factory method:

1. Our simple code may become complicated due to the existence of a lot of classes.
2. We end up with a huge number of small files i.e, cluttering of files.

Applicability:

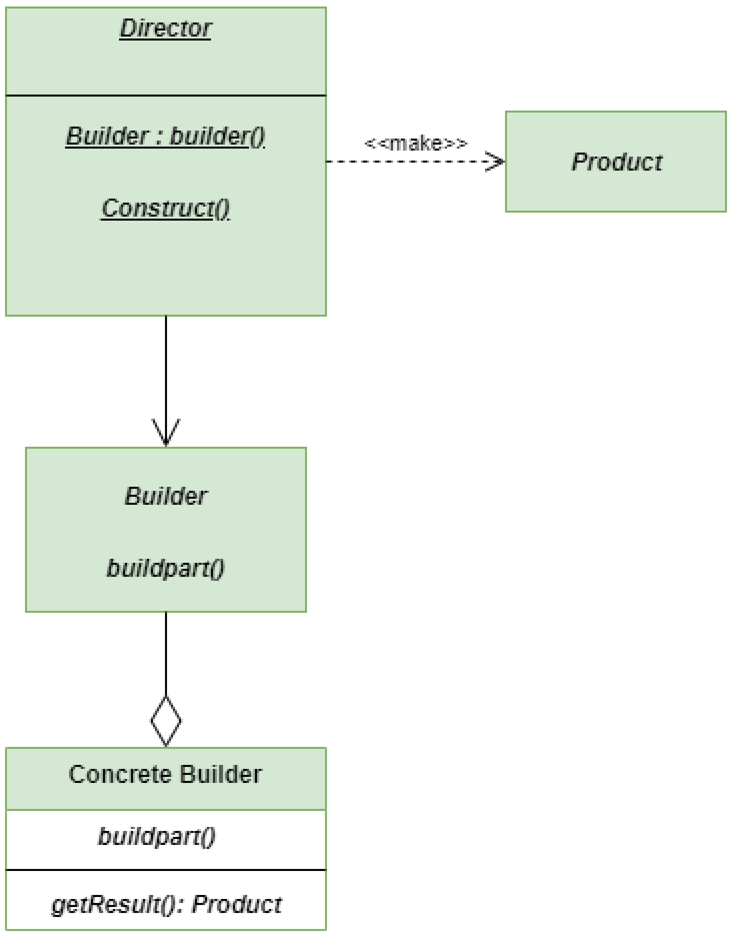
1. Most commonly, abstract factory method pattern is found in the sheet metal stamping equipment used in the manufacture of automobiles.
2. It can be used in a system that has to process reports of different categories such as reports related to input, output, and intermediate transactions.

---

3. Builder Method – Python Design Patterns

Builder Method is a Creation Design Pattern which aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It allows you to construct complex objects step by step. Here using the same construction code, we can produce different types and representations of the object easily.
It is basically designed to provide flexibility to the solutions to various object creation problems in object-oriented programming.

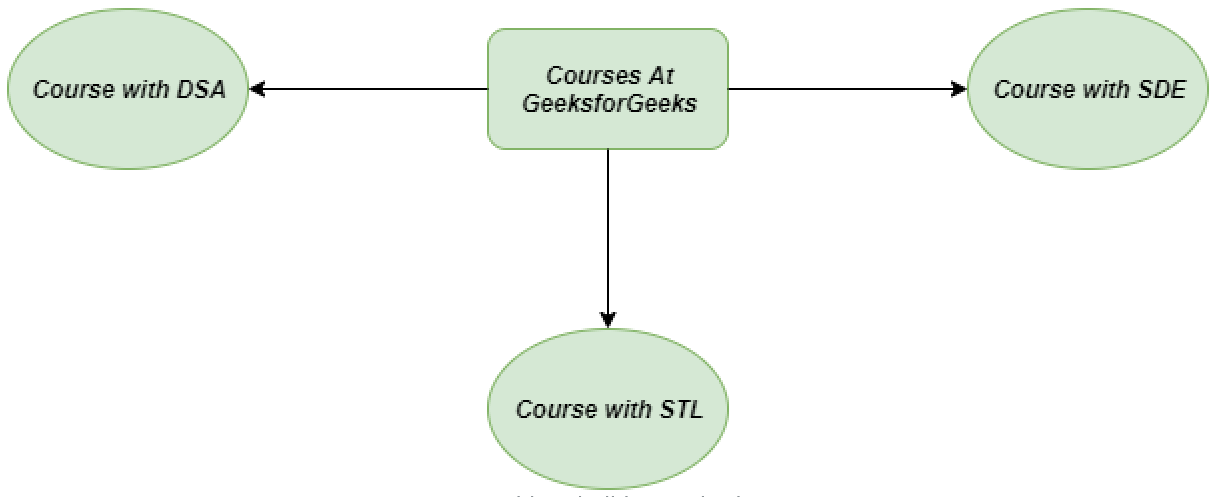
UML Diagram of Builder Design Pattern



Builder-method

Problem without using the Builder Method:

Imagine you want to join one of the elite batches of GeeksforGeeks. So, you will go there and ask about the Fee structure, timings available, and batches about the course you want to join. After looking at the system, they will tell you about the courses, their Fee structures, timings available and batches. That’s it! (No! we are not done yet because we are good developers).
Our main purpose is to design the system flexible, reliable, organized and lubricative. what Unexperienced developers will do is that they will create a separate and unique class for each and every course provided by **GeeksforGeeks**. Then they will create separate object instantiation for each and every class although which is not required every time. The main problem will arise when GeeksforGeeks will start new courses and developers have to add new classes as well because their code is not much flexible.



problem-builder-Method

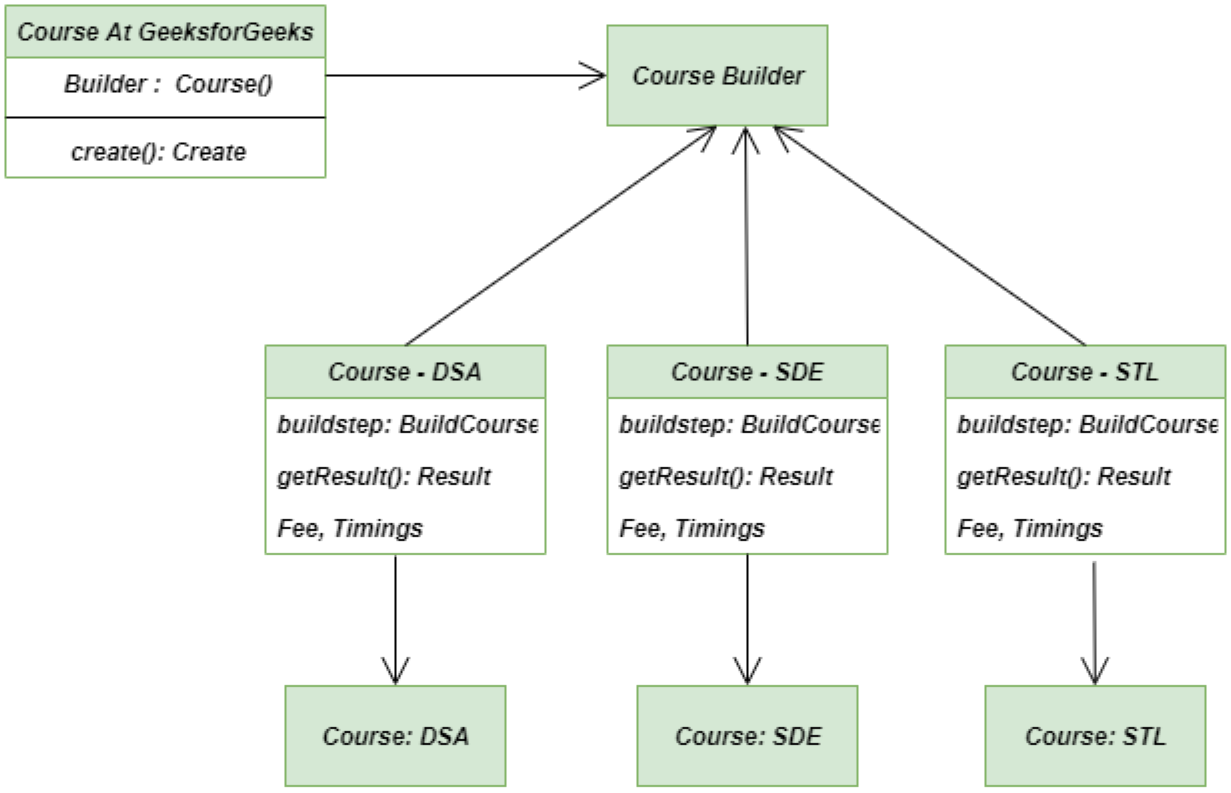
Note: Following code is written without using the Builder Method.

- Python3

```
# concrete course
class DSA():

    """Class for Data Structures and Algorithms"""

    def Fee(self):
        self.fee = 8000

    def available_batches(self):
        self.batches = 5
    def __str__(self):

        return "DSA"


# concrete course

class SDE():


    """Class for Software development Engineer"""


    def Fee(self):

        self.fee = 10000


    def available_batches(self):

        self.batches = 4


    def __str__(self):

        return "SDE"


# concrete course

class STL():
    """class for Standard Template Library of C++"""

    def Fee(self):
        self.fee = 5000

    def available_batches(self):
        self.batches = 7

    def __str__(self):
        return "STL"


# main method
if __name__ == "__main__":

    sde = SDE()   # object for SDE
    dsa = DSA()   # object for DSA
    stl = STL()   # object for STL
    print(f'Name of Course: {sde} and its Fee: {sde.fee}')

    print(f'Name of Course: {stl} and its Fee: {stl.fee}')

    print(f'Name of Course: {dsa} and its Fee: {dsa.fee}')
```

Solution by Builder Method:

Our final end product should be any course from GeeksforGeeks. It might be either SDE, STL or DSA. We have to go through many steps before choosing a particular course such as finding details about the courses, syllabus, fee structure, timings, and batches. Here using the same process we can select different courses available at **GeeksforGeeks**. That's the benefit of using the **builder Pattern.**



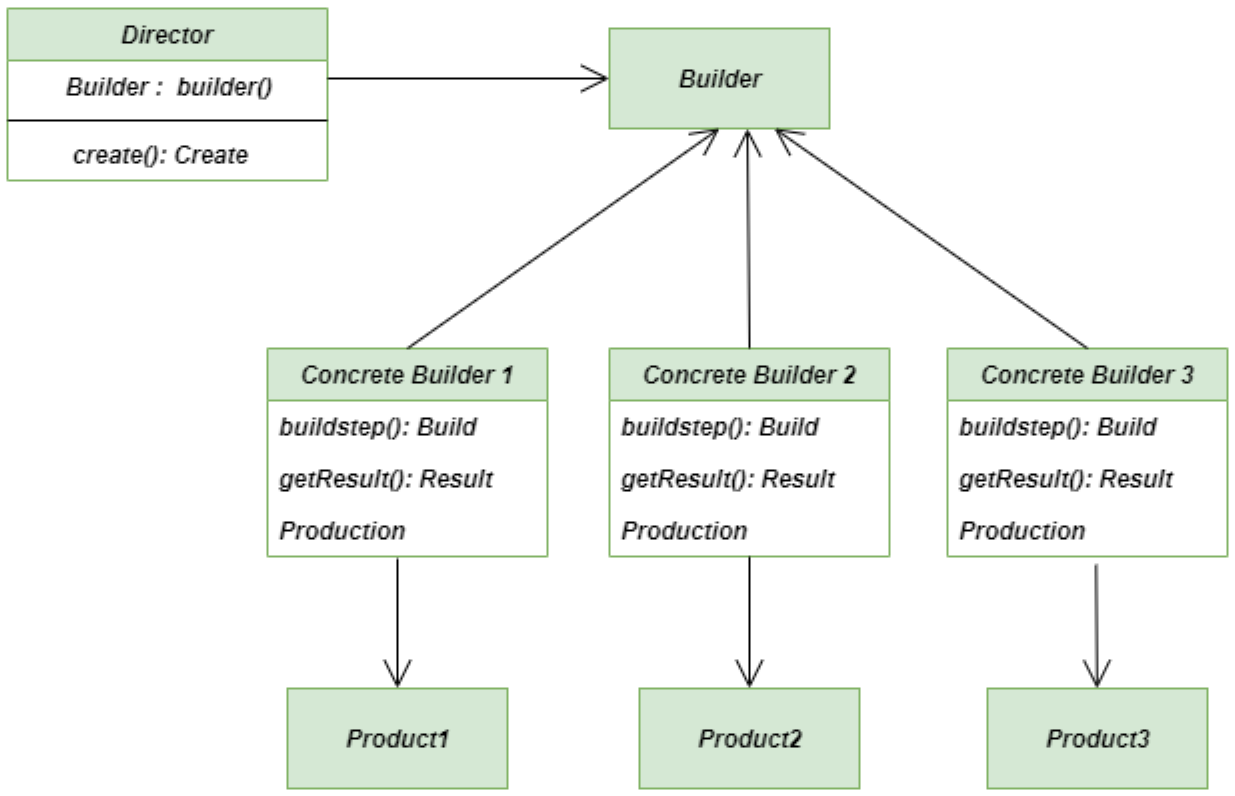
solution-Builder-Method

- Python3

```
# Abstract course

class Course:



    def __init__(self):

        self.Fee()

        self.available_batches()



    def Fee(self):

        raise NotImplementedError



    def available_batches(self):

        raise NotImplementedError



    def __repr__(self):

        return 'Fee : {0.fee} | Batches Available :
{0.batches}'.format(self)



# concrete course

class DSA(Course):
    """Class for Data Structures and Algorithms"""

    def Fee(self):
        self.fee = 8000

    def available_batches(self):
        self.batches = 5

    def __str__(self):
        return "DSA"

# concrete course
class SDE(Course):

    """Class for Software Development Engineer"""

    def Fee(self):
        self.fee = 10000

    def available_batches(self):
        self.batches = 4

    def __str__(self):
        return "SDE"

# concrete course
class STL(Course):

    """Class for Standard Template Library"""

    def Fee(self):
        self.fee = 5000

    def available_batches(self):
        self.batches = 7

    def __str__(self):
        return "STL"

# Complex Course
```

```
class ComplexCourse:



    def __repr__(self):

        return 'Fee : {0.fee} | available_batches:
{0.batches}'.format(self)



# Complex course

class Complexcourse(ComplexCourse):



    def Fee(self):

        self.fee = 7000



    def available_batches(self):

        self.batches = 6



# construct course

def construct_course(cls):



    course = cls()

    course.Fee()
    course.available_batches()


    return course    # return the course object


# main method

if __name__ == "__main__":


    dsa = DSA()  # object for DSA course

    sde = SDE()  # object for SDE course

    stl = STL()  # object for STL course


    complex_course = construct_course(Complexcourse)

    print(complex_course)
```

General Class Diagram for Builder Pattern Method:



General class diagram for builder method

Advantages of using Builder Method:

1. **Reusability:** While making the various representations of the products, we can use the same construction code for other representations as well.
2. **Single Responsibility Principle:** We can separate out both the business logic as well as the complex construction code from each other.
3. **Construction of the object:** Here we construct our object step by step, defer construction steps or run steps recursively.

Disadvantages of using Builder method:

1. **Code complexity increases:** The complexity of our code increases, because the builder pattern requires creating multiple new classes.
2. **Mutability:** It requires the builder class to be mutable
3. **Initialization:** Data members of the class are not guaranteed to be initialized.

Applicability:

1. **Constructing Complex objects :** The Builder Method allows you to construct the products step-by-step. Even, we can defer the execution of some steps without breaking the final product. To create an object tree, it is handy to call the steps recursively.It prevents the client code from fetching the incomplete data because it doesn’t allow the exposing of an unfinished object.

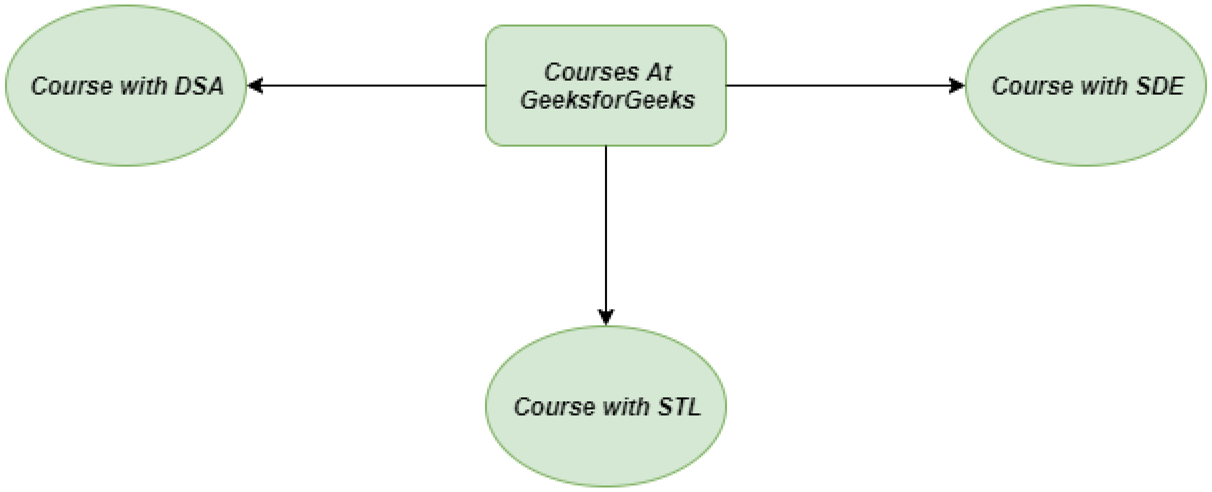
2. **Differ by Representations:** The Builder pattern is applicable when construction of various representations of the product involves similar steps that differ only in the details. The base builder interface is used to define all the construction steps while these steps are implemented by concrete builders.

Prototype Method – Python Design Patterns

Prototype Method is a Creational Design Pattern which aims to reduce the number of classes used for an application. It allows you to copy existing objects independent of the concrete implementation of their classes. Generally, here the object is created by copying a prototypical instance during run-time.
It is highly recommended to use **Prototype Method** when the object creation is an expensive task in terms of time and usage of resources and already there exists a similar object. This method provides a way to copy the original object and then modify it according to our needs.

Problems we face without Prototype Method

Suppose we have a Shape class that produces different shapes such as circle, rectangle, square, etc and we already have an object of it. Now we want to create the exact copy of this object. ***How an ordinary developer will go?***
He/She will create a new object of the same class and will apply all the functionalities of the original objects and copy their values. But we can not copy each and every field of the original object as some may be private and protected and are not available from the outside of the object itself.
Problems are not over here! you also become dependent on the code of other class which is certainly not a good practice in Software Development.
For better understanding, let’s understand the example of **Courses At GeeksforGeeks** that provides courses like SDE, DSA, STL, etc. Creating objects for similar courses, again and again, is not a good task to utilize the resources in a better way.



problem-builder-Method

Note: Following code is written without using Prototype Method

- Python3

```
# concrete course

class DSA():

    """Class for Data Structures and Algorithms"""



    def Type(self):

        return "Data Structures and Algorithms"



    def __str__(self):

        return "DSA"
```

```
# concrete course
class SDE():
    """Class for Software development Engineer"""

    def Type(self):
        return "Software Development Engineer"

    def __str__(self):
        return "SDE"


# concrete course
class STL():
    """class for Standard Template Library of C++"""

    def Type(self):
        return "Standard Template Library"

    def __str__(self):
        return "STL"
```

```
# main method

if __name__ == "__main__":

    sde = SDE()  # object for SDE

    dsa = DSA()  # object for DSA

    stl = STL()  # object for STL



    print(f'Name of Course: {sde} and its type: {sde.Type()}')

    print(f'Name of Course: {stl} and its type: {stl.Type()}')

    print(f'Name of Course: {dsa} and its type: {dsa.Type()}')
```

Solution by Prototype Method:

To deal with such problems, we use the Prototype method. We would create separate classes for **Courses_At_GFG** and **Course_At_GFG_Cache** which will help us in creating the exact copy of already existing object with the same field properties. This method delegates the cloning process to the actual objects that are being cloned. Here we declare a common interface or class which supports object cloning which allows us to clone the object without coupling our code to the class of that method.
An object that supports cloning is called as **Prototype**.

- Python3

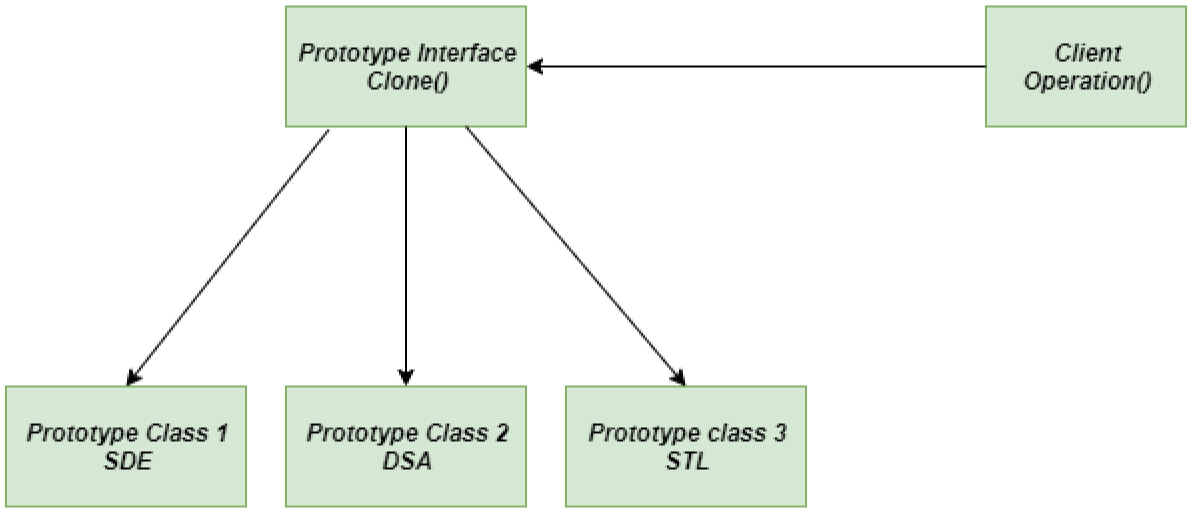
```
# import the required modules
```

```
from abc import ABCMeta, abstractmethod

import copy




# class - Courses at GeeksforGeeks

class Courses_At_GFG(metaclass = ABCMeta):


    # constructor

    def __init__(self):

        self.id = None

        self.type = None


    @abstractmethod

    def course(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 - DSA course

class DSA(Courses_At_GFG):

    def __init__(self):

        super().__init__()

        self.type = "Data Structures and Algorithms"



    def course(self):

        print("Inside DSA::course() method")



# class - SDE Course

class SDE(Courses_At_GFG):
    def __init__(self):

        super().__init__()

        self.type = "Software Development Engineer"



    def course(self):

        print("Inside SDE::course() method.")



# class - STL Course

class STL(Courses_At_GFG):

    def __init__(self):

        super().__init__()

        self.type = "Standard Template Library"



    def course(self):

        print("Inside STL::course() method.")



# class - Courses At GeeksforGeeks Cache

class Courses_At_GFG_Cache:



    # cache to store useful information
```

```
cache = {}


@staticmethod

def get_course(sid):

    COURSE = Courses_At_GFG_Cache.cache.get(sid, None)

    return COURSE.clone()


@staticmethod

def load():

    sde = SDE()

    sde.set_id("1")

    Courses_At_GFG_Cache.cache[sde.get_id()] = sde


    dsa = DSA()

    dsa.set_id("2")

    Courses_At_GFG_Cache.cache[dsa.get_id()] = dsa


    stl = STL()

    stl.set_id("3")

    Courses_At_GFG_Cache.cache[stl.get_id()] = stl
```

```
# main function

if __name__ == '__main__':

    Courses_At_GFG_Cache.load()



    sde = Courses_At_GFG_Cache.get_course("1")

    print(sde.get_type())



    dsa = Courses_At_GFG_Cache.get_course("2")

    print(dsa.get_type())



    stl = Courses_At_GFG_Cache.get_course("3")

    print(stl.get_type())
```

UML Diagram of Prototype Design Pattern



Prototype-Method – UML-Diagram

Advantages

1. **Less number of SubClasses :** All the other Creational Design Patterns provides a lot of new subClasses which are definitely not easy to handle when we are working on a large project. But using Prototype Design Pattern, we get rid of this.
2. **Provides varying values to new objects:** All the highly dynamic systems allows you to define new behavior through object composition by specifying values for an object's variables and not by defining new classes.
3. **Provides varying structure to new objects:** Generally all the applications build objects from parts and subparts. For convenience, such applications often allows you instantiate complex, user-defined structures to use a specific subcircuit again and again.

Disadvantages

1. **Abstraction:** It helps in achieving the abstraction by hiding the concrete implementation details of the class.
2. **Waste of resources at lower level:** It might be proved as the overkill of resources for a project that uses very few objects

Applicability

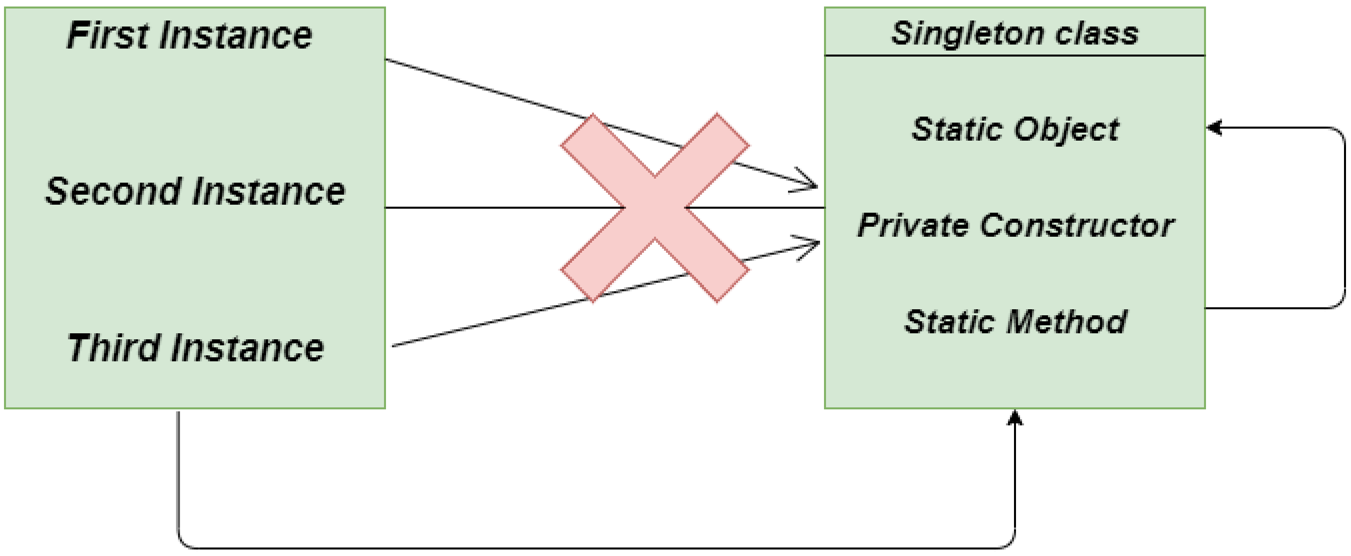
1. **Independency from Concrete Class:** Prototype method provides the way to implement the new objects without depending upon the concrete implementation of the class.

2. **Recurring problems :** Prototype method is also used to solve the recurring and complex problems of the software development.

Singleton Method – Python Design Patterns

What is Singleton Method in Python

Singleton Method is a type of Creational Design pattern and is one of the simplest design patterns00 available to us. It is a way to provide one and only one object of a particular type. It involves only one class to create methods and specify the objects.
Singleton Design Pattern can be understood by a very simple example of Database connectivity. When each object creates a unique Database Connection to the Database, it will highly affect the cost and expenses of the project. So, it is always better to make a single connection rather than making extra irrelevant connections which can be easily done by **Singleton Design Pattern**.



singleton-pattern

Definition: *The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*

Now let's have a look at the different implementations of the Singleton Design pattern.

Method 1: Monostate/Borg Singleton Design pattern

Singleton behavior can be implemented by Borg's pattern but instead of having only one instance of the class, there are multiple instances that share the same state. Here we don't focus on the sharing of the instance identity instead we focus on the sharing state.

Python3

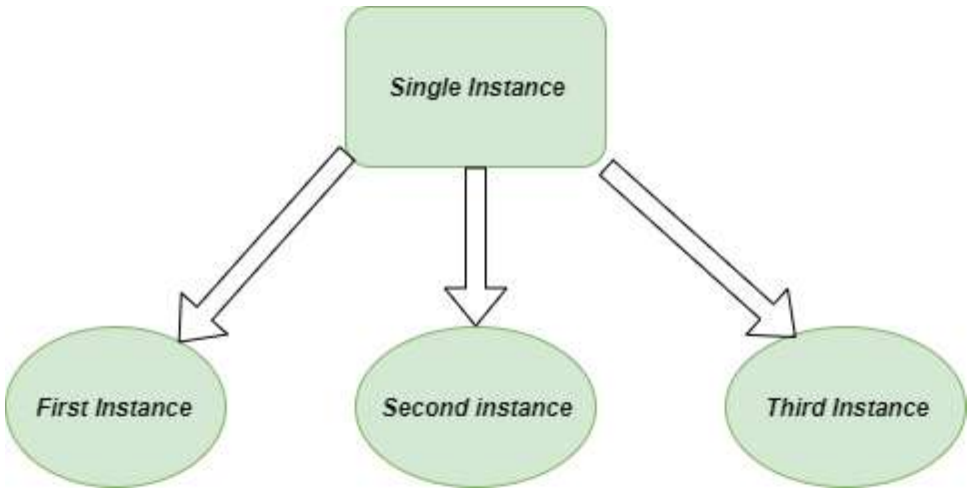
```
# Singleton Borg pattern

class Borg:



    # state shared by each instance

    __shared_state = dict()



    # constructor method

    def __init__(self):



        self.__dict__ = self.__shared_state

        self.state = 'GeeksforGeeks'



    def __str__(self):



        return self.state





# main method
```

```
 if __name__ == "__main__":


     person1 = Borg()    # object of class Borg

     person2 = Borg()    # object of class Borg

     person3 = Borg()    # object of class Borg


     person1.state = 'DataStructures' # person1 changed the state

     person2.state = 'Algorithms'   # person2 changed the state


     print(person1)    # output --> Algorithms

     print(person2)    # output --> Algorithms


     person3.state = 'Geeks' # person3 changed the

     # the shared state


     print(person1)    # output --> Geeks

     print(person2)    # output --> Geeks

     print(person3)    # output --> Geeks
```

Output:

```
Algorithms

Algorithms
```

```
Geeks
Geeks
Geeks
```



Singleton-Design-pattern

Double Checked Locking **Singleton Design pattern**

It is easy to notice that once an object is created, the synchronization of the threading is no longer useful because now the object will never be equal to None and any sequence of operations will lead to consistent results.
So, when the object will be equal to None, then only we will acquire the **Lock** on the getInstance method.

Python3

```
# Double Checked Locking singleton pattern

import threading




class SingletonDoubleChecked(object):



    # resources shared by each and every
    # instance

    __singleton_lock = threading.Lock()
    __singleton_instance = None

    # define the classmethod
    @classmethod
    def instance(cls):

        # check for the singleton instance
        if not cls.__singleton_instance:
            with cls.__singleton_lock:
                if not cls.__singleton_instance:
                    cls.__singleton_instance = cls()

        # return the singleton instance
        return cls.__singleton_instance


# main method
```

```
if __name__ == '__main__':

    # create class X
    class X(SingletonDoubleChecked):
        pass

    # create class Y
    class Y(SingletonDoubleChecked):
        pass

    A1, A2 = X.instance(), X.instance()
    B1, B2 = Y.instance(), Y.instance()

    assert A1 is not B1
    assert A1 is A2
    assert B1 is B2

    print('A1 : ', A1)
    print('A2 : ', A2)
    print('B1 : ', B1)
    print('B2 : ', B2)
```

Output:

```
A1 :  __main__.X object at 0x02EA2590
A2 :  __main__.X object at 0x02EA2590
B1 :  __main__.Y object at 0x02EA25B0
B2 :  __main__.Y object at 0x02EA25B0
```

Creating a singleton in Python

In the classic implementation of the Singleton Design pattern, we simply use the static method for creating the getInstance method which has the ability to return the shared resource. We also make use of the so-called **Virtual private Constructor** to raise the exception against it although it is not much required.

Python3

```
# classic implementation of Singleton Design pattern

class Singleton:



    __shared_instance = 'GeeksforGeeks'



    @staticmethod

    def getInstance():

        """Static Access Method"""

        if Singleton.__shared_instance == 'GeeksforGeeks':

            Singleton()

        return Singleton.__shared_instance
    def __init__(self):

        """virtual private constructor"""

        if Singleton.__shared_instance != 'GeeksforGeeks':

            raise Exception("This class is a singleton class !")

        else:

            Singleton.__shared_instance = self




# main method

if __name__ == "__main__":



    # create object of Singleton Class

    obj = Singleton()

    print(obj)



    # pick the instance of the class

    obj = Singleton.getInstance()

    print(obj)
```

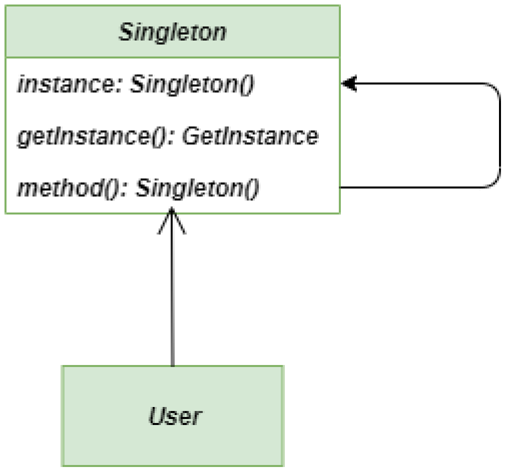
Output:

```
__main__.Singleton object at 0x014FFE90
```

```
 __main__.Singleton object at 0x014FFE90
```

Class diagram

Class Diagram of Singleton Design Pattern



singleton-class-diagram

Advantages of using the Singleton Method:

1. **Initializations: An object** created by the Singleton method is initialized only when it is requested for the first time.
2. **Access to the object:** We got global access to the instance of the object.
3. **Count of instances:** In singleton, method classes can't have more than one instance

Disadvantages of using the Singleton Method:

1. **Multithread Environment: It's** not easy to use the singleton method in a multithread environment, because we have to take care that the multithread wouldn't create a singleton object several times.
2. **Single responsibility principle:** As the Singleton method is solving two problems at a single time, it doesn't follow the single responsibility principle.
3. **Unit testing process:** As they introduce the global state to the application, it makes the unit testing very hard.

Applicability

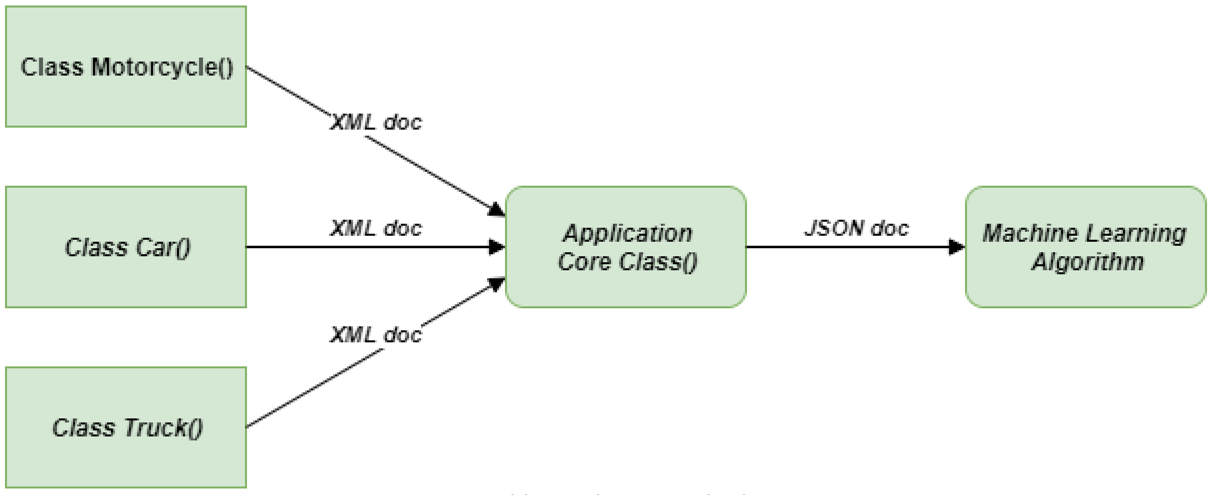
1. **Controlling over global variables:** In the projects where we specifically need strong control over the global variables, it is highly recommended to use **Singleton Method**
2. **Daily Developers use:** Singleton patterns are generally used in providing the logging, caching, thread pools, and configuration settings and are often used in conjunction with Factory design patterns.

Adapter Method – Python Design Patterns

Adapter method is a Structural Design Pattern which helps us in making the incompatible objects adaptable to each other. The Adapter method is one of the easiest methods to understand because we have a lot of real-life examples that show the analogy with it. The main purpose of this method is to create a bridge between two incompatible interfaces. This method provides a different interface for a class. We can more easily understand the concept by thinking about the Cable Adapter that allows us to charge a phone somewhere that has outlets in different shapes. Using this idea, we can integrate the classes that couldn't be integrated due to interface incompatibility.

Problem without using the Adapter Method

Imagine you are creating an application that shows the data about all different types of vehicles present. It takes the data from APIs of different vehicle organizations in XML format and then displays the information. But suppose at some time you want to upgrade your application with a Machine Learning algorithms that work beautifully on the data and fetch the important data only. But there is a problem, it takes data in JSON format only. It will be a really poor approach to make changes in Machine Learning Algorithm so that it will take data in XML format.



Problem-Adapter-Method

Solution using Adapter Method

For solving the problem we defined above, We can use **Adapter Method** that helps by creating an Adapter object. To use an adapter in our code:

1. Client should make a request to the adapter by calling a method on it using the target interface.
2. Using the Adaptee interface, the Adapter should translate that request on the adaptee.
3. Result of the call is received the client and he/she is unaware of the presence of the Adapter's presence.

- Python3

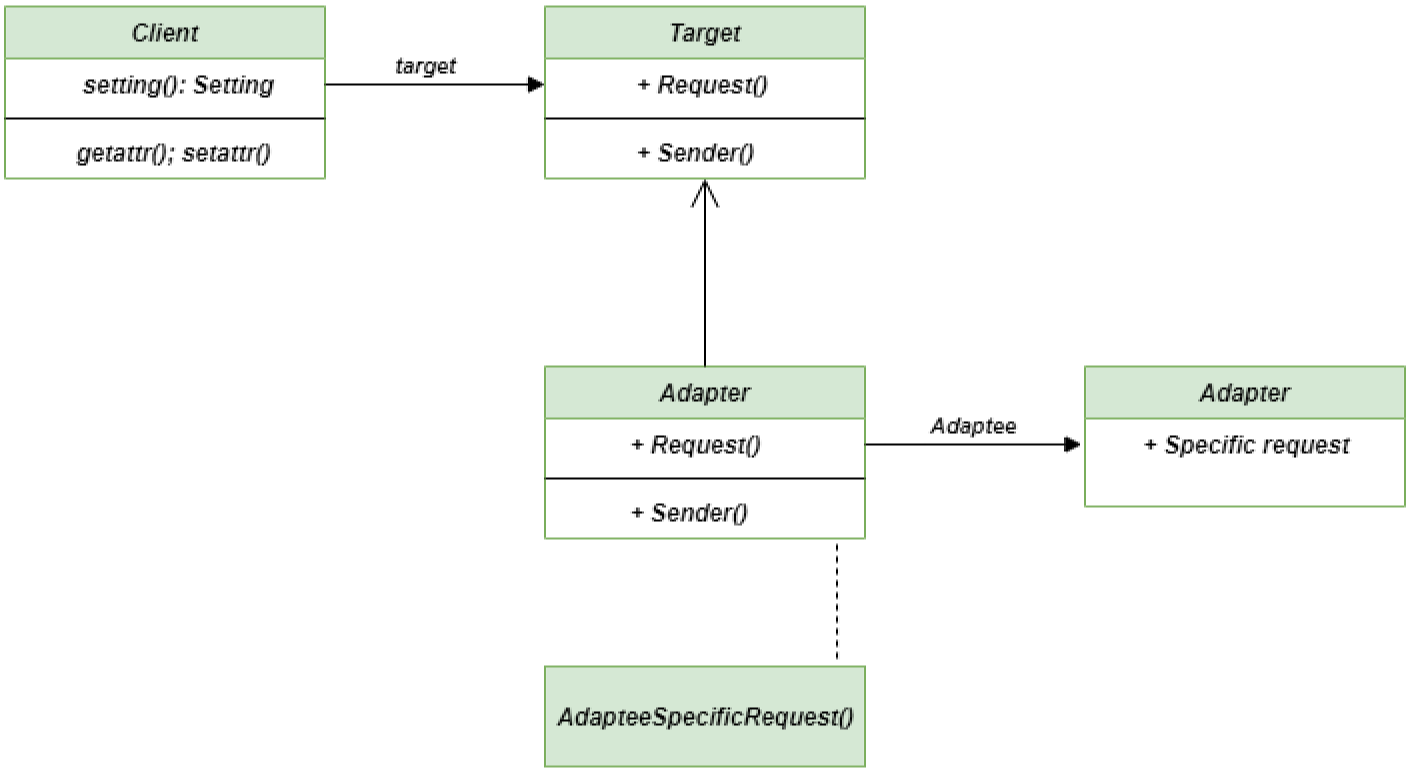
```
# Dog - Cycle

# human - Truck

# car - Car



class MotorCycle:
    """Class for MotorCycle"""

    def __init__(self):
        self.name = "MotorCycle"

    def TwoWheeler(self):
        return "TwoWheeler"


class Truck:

    """Class for Truck"""

    def __init__(self):
        self.name = "Truck"

    def EightWheeler(self):
        return "EightWheeler"
```

```
class Car:

    """Class for Car"""

    def __init__(self):
        self.name = "Car"

    def FourWheeler(self):
        return "FourWheeler"

class Adapter:
    """
    Adapts an object by replacing methods.
    Usage:
    motorCycle = MotorCycle()
    motorCycle = Adapter(motorCycle, wheels = motorCycle.TwoWheeler)
    """

    def __init__(self, obj, **adapted_methods):
        """We set the adapted methods in the object's dict"""

        self.obj = obj

        self.__dict__.update(adapted_methods)



    def __getattr__(self, attr):

        """All non-adapted calls are passed to the object"""

        return getattr(self.obj, attr)



    def original_dict(self):

        """Print original object dict"""

        return self.obj.__dict__




""" main method """

if __name__ == "__main__":



    """list to store objects"""

    objects = []



    motorCycle = MotorCycle()
```

```
objects.append(Adapter(motorCycle, wheels = motorCycle.TwoWheeler))


truck = Truck()

objects.append(Adapter(truck, wheels = truck.EightWheeler))


car = Car()

objects.append(Adapter(car, wheels = car.FourWheeler))


for obj in objects:

   print("A {0} is a {1} vehicle".format(obj.name, obj.wheels()))
```

Class Diagram

Class diagram for the Adapter method which is a type of **Structural Design pattern**:



Adapter-class-diagram

Advantages

- **Principle of Single Responsibility:** We can achieve the principle of Single responsibility with Adapter Method because here we can separate the concrete code from the primary logic of the client.
- **Flexibility:** Adapter Method helps in achieving the flexibility and reusability of the code.
- **Less complicated class:** Our client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.
- **Open/Closed principle:** We can introduce the new adapter classes into the code without violating the Open/Closed principle.

Disadvantages

- **Complexity of the Code:** As we have introduced the set of new classes, objects and interfaces, the complexity of or code definitely rises.
- **Adaptability:** Most of the times, we require many adaptations with the adaptee chain to reach the compatibility what we want.

Applicability

- **To make classes and interfaces compatible :** Adapter method is always used when we are in need to make certain classes compatible to communicate.
- **Relatable to Inheritance:** When we want to reuse some piece of code i.e., classes and interfaces that lack some functionalities, it can be done using the **Adapter Method**.

Bridge Method – Python Design Patterns

The bridge method is a Structural Design Pattern that allows us to separate the **Implementation Specific Abstractions** and **Implementation Independent Abstractions** from each other and can be developed considering as single entities.
The bridge Method is always considered as one of the best methods to organize the class hierarchy.

Bridge Method

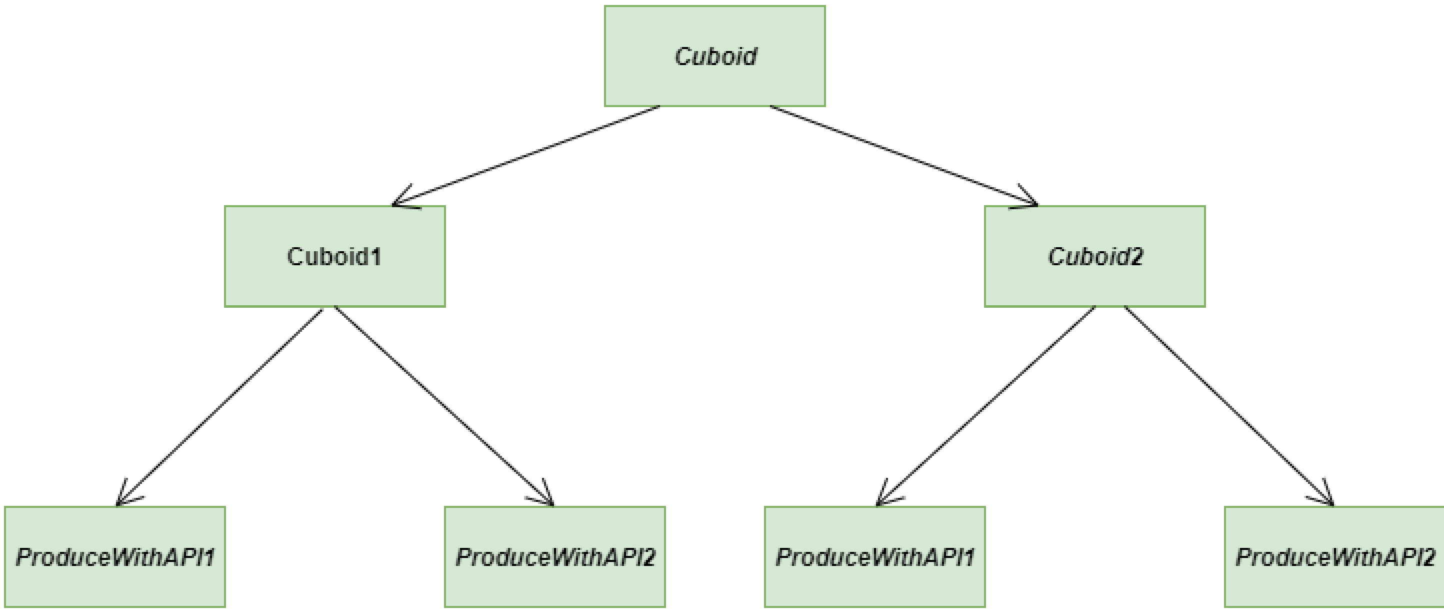
Elements of Bridge Design Pattern

- **Abstraction:** It is the core of the Bridge Design Pattern and it provides the reference to the implementer.
- **Refined Abstraction:** It extends the abstraction to a new level where it takes the finer details one level above and hides the finer element from the implementers.

- **Implementer:** It defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different.
- **Concrete Implementation:** Through the concrete implementation, it implements the above implementer.

Problem without using Bridge Method

Consider the following class **Cuboid** which has three attributes named **length, breadth,** and **height** and three methods named **ProducewithAPI1(), ProduceWithAPI2(),** and **expand()**.
Out of these, producing methods are implementation-specific as we have two production APIs, and one method i.e., expand() method is implementation-independent.
Till now we have only two implementation-specific methods and one implementation-independent method but when the quantity will rise (of course in a large-scale project) things will become messy for the developers to handle.



Problem-Bridge-Method

Note: Following code is written without using the Bridge method.

- Python3

```
""" Code without using the bridge method
    We have a class with three attributes
    named as length, breadth, and height and
    three methods named as ProduceWithAPI1(),
    ProduceWithAPI2(), and expand(). Out of these
    producing methods are implementation-specific
    as we have two production APIs"""

class Cuboid:

    class ProducingAPI1:

        """Implementation Specific Implementation"""

        def produceCuboid(self, length, breadth, height):

            print(f'API1 is producing Cuboid with length = {length}, '
                  f' Breadth = {breadth} and Height = {height}')
```

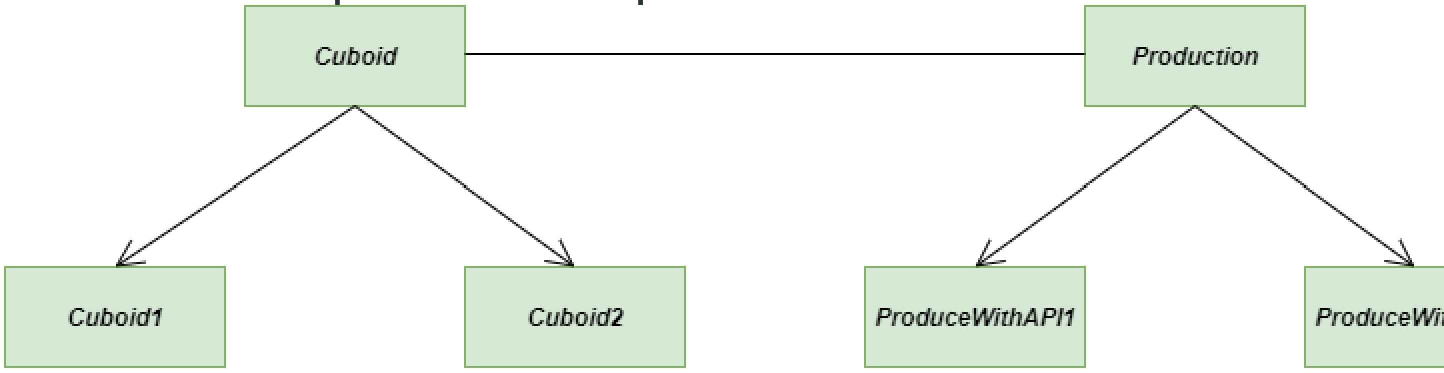
```
class ProducingAPI2:

    """Implementation Specific Implementation"""


    def produceCuboid(self, length, breadth, height):

        print(f'API2 is producing Cuboid with length = {length}, '

              f' Breadth = {breadth} and Height = {height}')




def __init__(self, length, breadth, height):


    """Initialize the necessary attributes"""


    self._length = length

    self._breadth = breadth

    self._height = height


def produceWithAPI1(self):


    """Implementation specific Abstraction"""
```

```
        objectAPIone = self.ProducingAPI1()

        objectAPIone.produceCuboid(self._length, self._breadth,
self._height)



    def producewithAPI2(self):



        """Implementation specific Abstraction"""



        objectAPItwo = self.ProducingAPI2()

        objectAPItwo.produceCuboid(self._length, self._breadth,
self._height)



    def expand(self, times):



        """Implementation independent Abstraction"""



        self._length = self._length * times

        self._breadth = self._breadth * times

        self._height = self._height * times



# Instantiate a Cubiod
```

```
cuboid1 = Cuboid(1, 2, 3)



# Draw it using APIone

cuboid1.produceWithAPI1()



# Instantiate another Cuboid

cuboid2 = Cuboid(19, 20, 21)



# Draw it using APItwo

cuboid2.producewithAPI2()
```

Solution using Bridge method

Now let’s look at the solution for the above problem. **The bridge Method** is one of the best solutions for such kinds of problems. Our main purpose is to separate the codes of **implementation-specific abstractions** and **implementation-independent abstractions**.



Solution-Bridge-Method

Note: Following Code is written using Bridge Method

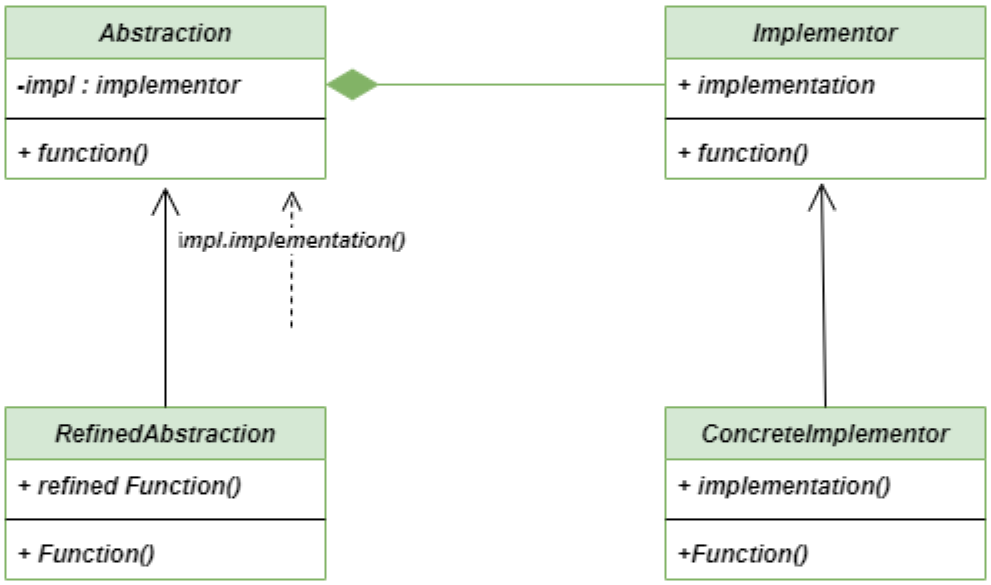
- Python3

```
"""Code implemented with Bridge Method.

   We have a Cuboid class having three attributes

   named as length, breadth, and height and three

   methods named as produceWithAPIOne(), produceWithAPItwo(),

   and expand(). Our purpose is to separate out implementation

   specific abstraction from implementation-independent

   abstraction"""


class ProducingAPI1:


    """Implementation specific Abstraction"""


    def produceCuboid(self, length, breadth, height):


        print(f'API1 is producing Cuboid with length = {length}, '

              f' Breadth = {breadth} and Height = {height}')


class ProducingAPI2:
    """Implementation specific Abstraction"""

    def produceCuboid(self, length, breadth, height):

        print(f'API2 is producing Cuboid with length = {length}, '
              f' Breadth = {breadth} and Height = {height}')

class Cuboid:

    def __init__(self, length, breadth, height, producingAPI):

        """Initialize the necessary attributes
           Implementation independent Abstraction"""

        self._length = length
        self._breadth = breadth
        self._height = height

        self._producingAPI = producingAPI
    def produce(self):

        """Implementation specific Abstraction"""

        self._producingAPI.produceCuboid(self._length, self._breadth,
self._height)

    def expand(self, times):

        """Implementation independent Abstraction"""

        self._length = self._length * times

        self._breadth = self._breadth * times

        self._height = self._height * times


"""Instantiate a cuboid and pass to it an

   object of ProducingAPIone"""

cuboid1 = Cuboid(1, 2, 3, ProducingAPI1())
```

```
cuboid1.produce()



cuboid2 = Cuboid(19, 19, 19, ProducingAPI2())

cuboid2.produce()
```

UML Diagram of Bridge Method

Following is the UML diagram for Bridge Method



UML -diagram-Bridge-Method

Advantages

- **Single Responsibility Principle: The bridge** method clearly follows the Single Responsibility principle as it decouples an abstraction from its implementation so that the two can vary independently.
- **Open/Closed Principle:** It does not violate the Open/Closed principle because at any time we can introduce the new abstractions and implementations independently from each other
- **Platform independent feature:** Bridge Method can be easily used for implementing the platform-independent features.

Disadvantages

- **Complexity:** Our code might become complex after applying the **Bridge method** because we are intruding on new abstraction classes and interfaces.
- **Double Indirection: The bridge** method might have a slight negative impact on the performance because the abstraction needs to pass messages along with the implementation for the operation to get executed.
- **Interfaces with only a single implementation:** If we have only limited interfaces, then it doesn't sweat a breath but if you have an exploded set of interfaces with minimal or only one implementation it becomes hard to manage

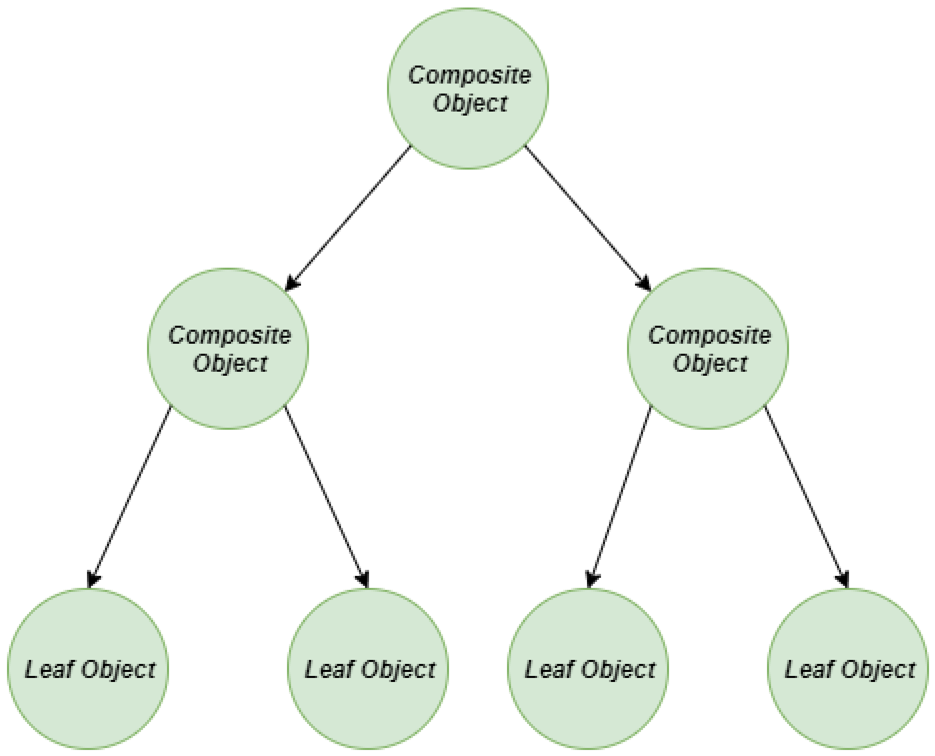
Applicability

- **Run-time Binding:** Generally Bridge method is used to provide the run-time binding of the implementation, here run-time binding refers to what we can call a method at run-time instead of compile-time.
- **Mapping classes: The bridge** method is used to map the orthogonal class hierarchies
- **UI Environment:** A real-life application of the Bridge method is used in the definition of shapes in a UI Environment

---

Composite Method – Python Design Patterns

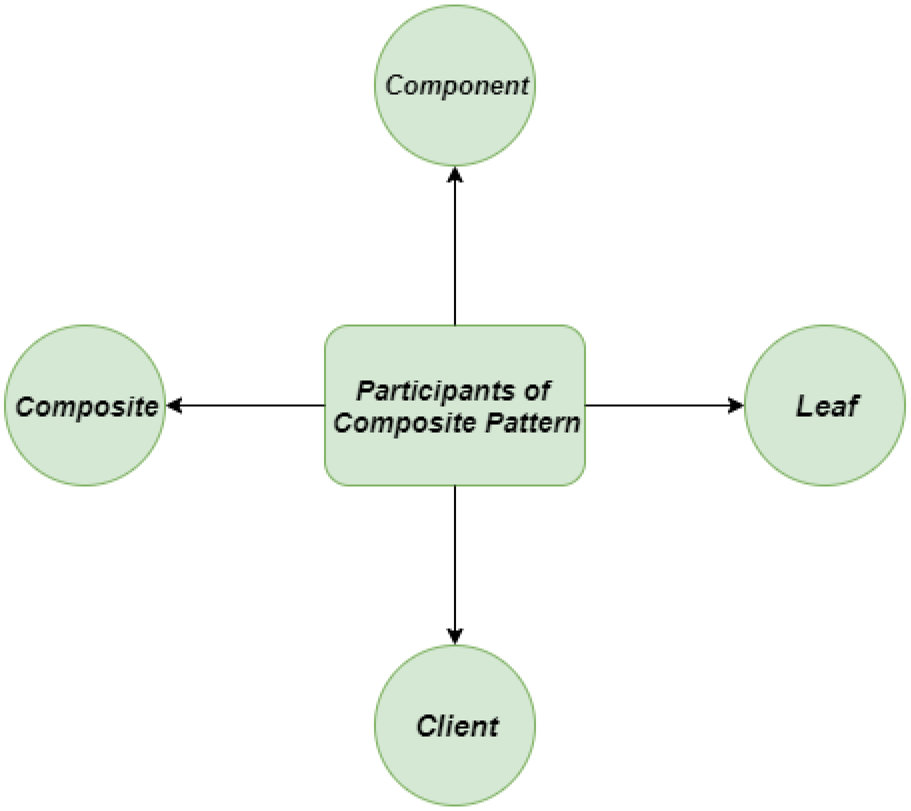
Composite Method is a Structural Design Pattern which describes a group of objects that is treated the same way as a single instance of the same type of the objects. The purpose of the Composite Method is to **Compose** objects into Tree type structures to represent the whole-partial hierarchies.
One of the main advantages of using the Composite Method is that first, it allows you to compose the objects into the **Tree Structure** and then work with these structures as an individual object or an entity.



Composite-Tree-Structure

The operations you can perform on all the composite objects often have the least common denominator relationship.

The Composite Pattern has four participants :



Participants-Composite-Method

- **Component:** Component helps in implementing the default behavior for the interface common to all classes as appropriate. It declares the interface of the objects in the composition and for accessing and managing its child components.
- **Leaf:** It defines the behavior for primitive objects in the composition. It represents the leaf object in the composition.
- **Composite:** It stores the child component and implements child related operations in the component interface.
- **Client:** It is used to manipulate the objects in the composition through the component interface.

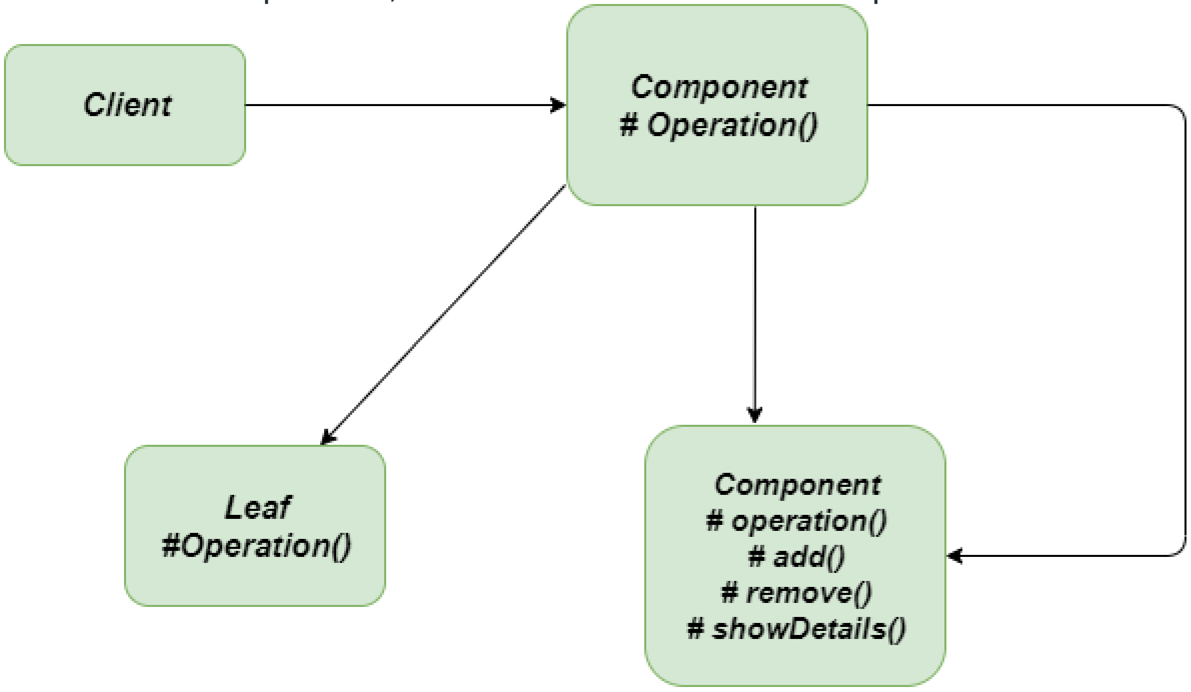
Problem without using Composite Method

Imagine we are studying an organizational structure which consists of General Managers, Managers, and Developers. A General Manager may have many Managers working under him and a Manager may have many developers under him.
Suppose, you have to determine the total salary of all the employees. *So, How would you determine that ?*

An ordinary developer will definitely try the direct approach, go over each employee and calculate the total salary. Looks easy? not so when it comes to implementation. Because we have to know the classes of all the employees General Manager, Manager, and Developers.
It seems even an impossible task to calculate through a direct approach in Tree-based structure.

Solution using Composite Method

One of the best solutions to the above-described problem is using **Composite Method** by working with a common interface that declares a method for calculating the total salary.
We will generally use the Composite Method whenever we have "composites that contain components, each of which could be a composite".



Composite-Running-Example

```
"""Here we attempt to make an organizational hierarchy with sub-
organization,

 which may have subsequent sub-organizations, such as:

GeneralManager                                   [Composite]
```

```
      Manager1                                    [Composite]

              Developer11                         [Leaf]

              Developer12                         [Leaf]

      Manager2                                    [Composite]

              Developer21                         [Leaf]

              Developer22                         [Leaf]"""


class LeafElement:



    '''Class representing objects at the bottom or Leaf of the hierarchy
tree.'''



    def __init__(self, *args):



        ''''Takes the first positional argument and assigns to member
variable "position".'''

        self.position = args[0]



    def showDetails(self):



        '''Prints the position of the child element.'''
        print("\t", end ="")

        print(self.position)




class CompositeElement:


    '''Class representing objects at any level of the hierarchy

     tree except for the bottom or leaf level. Maintains the child

      objects by adding and removing them from the tree structure.'''


    def __init__(self, *args):


        '''Takes the first positional argument and assigns to member

         variable "position". Initializes a list of children elements.'''

        self.position = args[0]

        self.children = []


    def add(self, child):


        '''Adds the supplied child element to the list of children
         elements "children".'''
        self.children.append(child)

    def remove(self, child):

        '''Removes the supplied child element from the list of
        children elements "children".'''
        self.children.remove(child)

    def showDetails(self):

        '''Prints the details of the component element first. Then,
        iterates over each of its children, prints their details by
        calling their showDetails() method.'''
        print(self.position)
        for child in self.children:
            print("\t", end ="")
            child.showDetails()
```

```
"""main method"""


if __name__ == "__main__":


    topLevelMenu = CompositeElement("GeneralManager")

    subMenuItem1 = CompositeElement("Manager1")

    subMenuItem2 = CompositeElement("Manager2")

    subMenuItem11 = LeafElement("Developer11")

    subMenuItem12 = LeafElement("Developer12")

    subMenuItem21 = LeafElement("Developer21")

    subMenuItem22 = LeafElement("Developer22")

    subMenuItem1.add(subMenuItem11)

    subMenuItem1.add(subMenuItem12)

    subMenuItem2.add(subMenuItem22)

    subMenuItem2.add(subMenuItem22)



    topLevelMenu.add(subMenuItem1)

    topLevelMenu.add(subMenuItem2)

    topLevelMenu.showDetails()
```

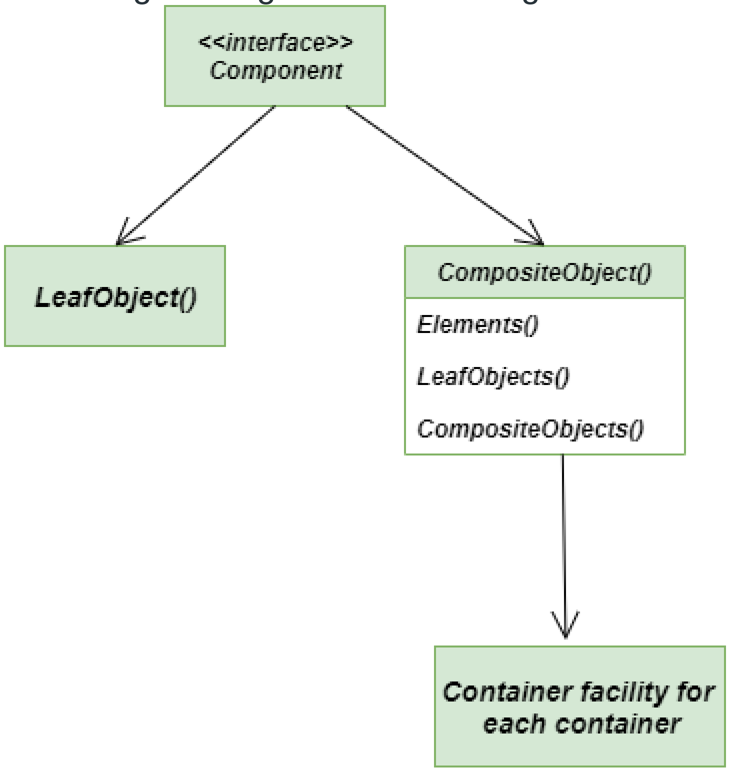
Output:

```
GeneralManager
```

```
Manager1
    Developer11
    Developer12
Manager2
    Developer22
    Developer22
```

Class Diagram of Composite Method

Following is the general class diagram for the **Composite Method**:



Class-diagram-Composite-Method

Advantages

- **Open/Closed Principle:** As the introduction of new elements, classes and interfaces is allowed into the application without breaking the existing code of the client, it definitely follows the **Open/Closed Principle**
- **Less Memory Consumption:** Here we have to create less number of objects as compared to the ordinary method, which surely reduces the

memory usage and also manages to keep us away from errors related to memory
- **Improved Execution Time:** Creating an object in Python doesn't take much time but still we can reduce the execution time of our program by sharing objects.
- **Flexibility:** It provides flexibility of structure with manageable class or interface as it defines class hierarchies that contains primitive and complex objects.

Disadvantages

- **Restriction on the Components:** Composite Method makes it harder to restrict the type of components of a composite. It is not preferred to use when you don't want to represent a full or partial hierarchy of the objects.
- **General Tree Structure:** The Composite Method will produce the overall general tree, once the structure of the tree is defined.
- **Type-System of Language:** As it is not allowed to use the type-system of the programming language, our program must depend on the run-time checks to apply the constraints.

Applicability

- **Requirement of Nested Tree Structure:** It is highly preferred to use **Composite Method** when you are need of producing the nested structure of tree which again include the leaves objects and other object containers.
- **Graphics Editor:** We can define a shape into types either it is **simple** for ex – a straight line or **complex** for ex – a rectangle. Since all the shapes have many common operations, such as rendering the shape to screen so composite pattern can be used to enable the program to deal with all shapes uniformly.

Decorator Method – Python Design Patterns

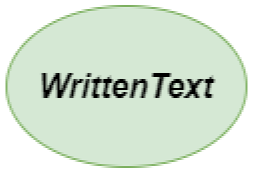
Decorator Method is a **Structural Design Pattern** which allows you to dynamically attach new behaviors to objects without changing their implementation by placing these objects inside the wrapper objects that contains the behaviors.
It is much easier to implement **Decorator Method** in Python because of its built-in feature. It is not equivalent to the **Inheritance** because the new feature is added only to that particular object, not to the entire subclass.

Problem Without Using Decorator Method

Imagine We are working with a formatting tool that provides features likes Bold the text and Underlines the text. But after some time, our formatting tools got famous among the targeted audience and on taking the feedback we got that our audience wants more features in the application such as making the text Italic and many other features.
Looks Simple? It’s not the easy task to implement this or to extend our classes to add more functionalities without disturbing the existing client code because we have to maintain the **Single Responsibility Principle**.

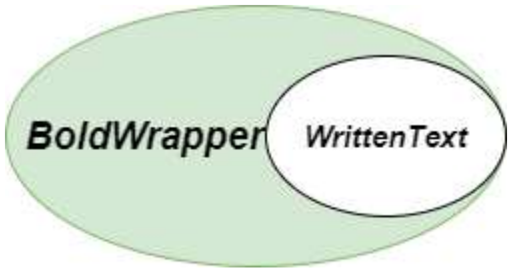
Solution Using Decorator Method

Now let’s look at the solution that we have to avoid such conditions. Initially, we have only **WrittenText** but we have to apply filters like **BOLD, ITALIC, UNDERLINE**. So, we will create separate wrapper classes for each function like BoldWrapperClass, ItalicWrapperClass, and UnderlineWrapperclass.



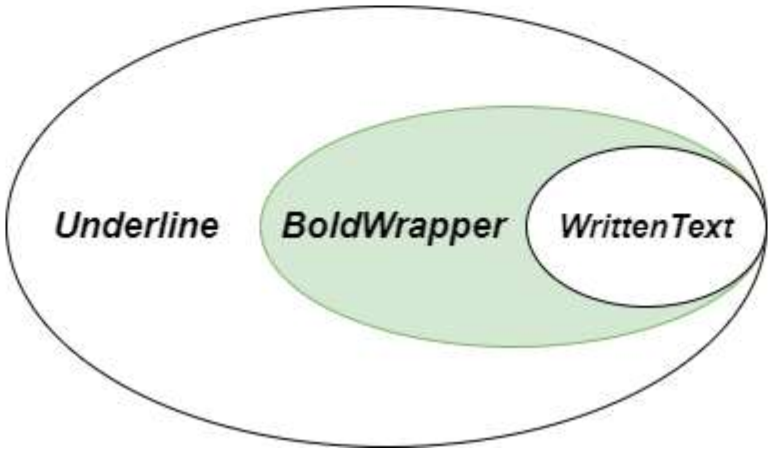
Decorator-Written-Text

First, we will call BoldWrapperclass over the Written text which ultimately converts the text into BOLD letters



Decorator-Wrapper

Then we will apply the ItalicWrapperClass and UnderlineWrapperClass over the Bold text which will give us the result what we want.



Decorator-Underline

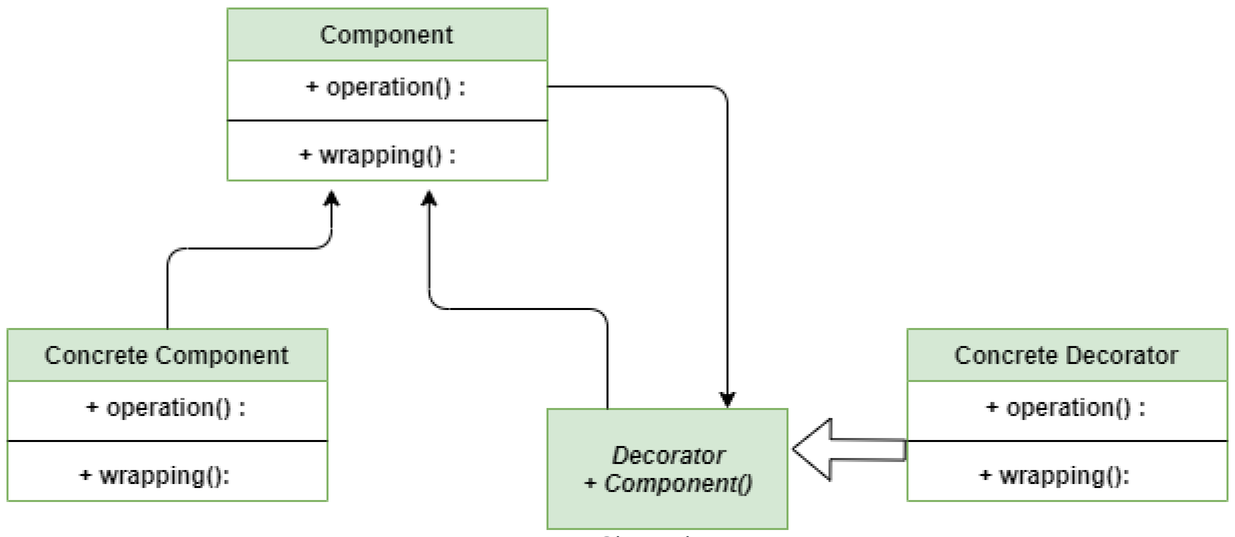
Following Code is written using Decorator Method:

- Python3

```
class WrittenText:

    """Represents a Written text """

    def __init__(self, text):
        self._text = text

    def render(self):
        return self._text

class UnderlineWrapper(WrittenText):
    """Wraps a tag in <u>"""

    def __init__(self, wrapped):
        self._wrapped = wrapped

    def render(self):
        return "<u>{}</u>".format(self._wrapped.render())

class ItalicWrapper(WrittenText):

    """Wraps a tag in <i>"""

    def __init__(self, wrapped):
        self._wrapped = wrapped

    def render(self):
        return "<i>{}</i>".format(self._wrapped.render())

class BoldWrapper(WrittenText):
    """Wraps a tag in <b>"""

    def __init__(self, wrapped):
        self._wrapped = wrapped

    def render(self):
        return "<b>{}</b>".format(self._wrapped.render())

""" main method """

if __name__ == '__main__':

    before_gfg = WrittenText("GeeksforGeeks")
    after_gfg = ItalicWrapper(UnderlineWrapper(BoldWrapper(before_gfg)))

    print("before :", before_gfg.render())
    print("after :", after_gfg.render())
```

Class-Diagram for Decorator Method

Following is the class diagram for Decorator Method:



Decorator-Class-Diagram

Advantages

- **Single Responsibility Principle:** It is easy to divide a monolithic class which implements many possible variants of behavior into several classes using the Decorator method.
- **Runtime Responsibilities:** We can easily add or remove the responsibilities from an object at runtime.
- **SubClassing:** The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.

Disadvantages

1. **removing Wrapper:** It is very hard to remove a particular wrapper from the wrappers stack.

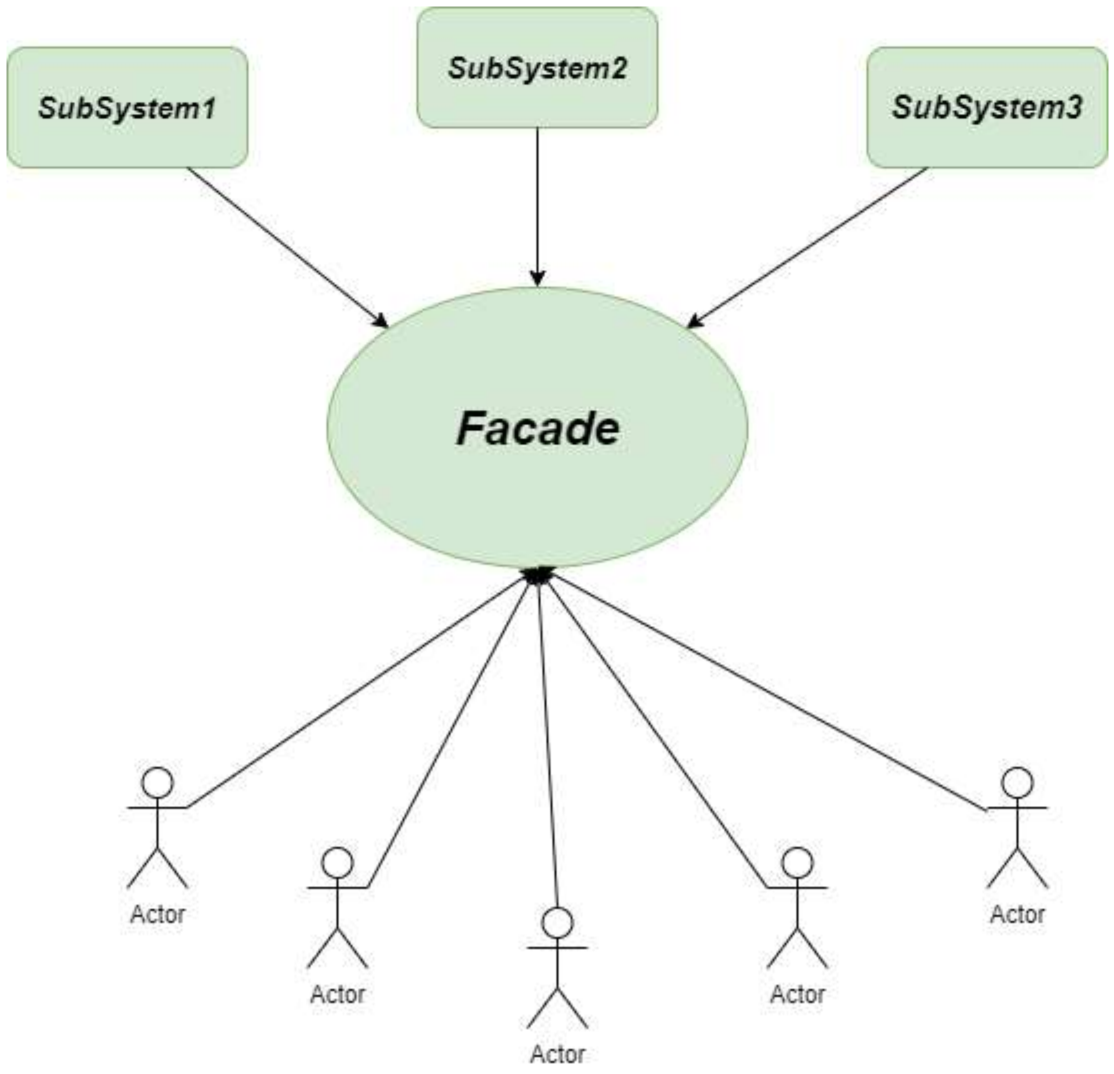
2. **Complicated Decorators:** It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.
3. **Ugly Configuration:** Large number of code of layers might make the configurations ugly.

Applicability

1. **Incapable Inheritance:** Generally, Decorator method is used when it is not possible to extend the behavior of an object using the Inheritance.
2. **Runtime Assignment:** One of the most important feature of Decorator method is to assign different and unique behaviors to the object at the **Runtime**.

Facade Method – Python Design Patterns

Facade Method is a **Structural Design pattern** that provides a simpler unified interface to a more complex system. The word **Facade** means the face of a building or particularly an outer lying interface of a complex system, consists of several sub-systems. It is an essential part **Gang of Four** design patterns. It provides an easier way to access methods of the underlying systems by providing a single entry point.
Here, we create a Facade layer that helps in communicating with subsystems easily to the clients.



Facade-Method

Problem without using Facade Method

Imagine we have a washing machine which can wash the clothes, rinse the clothes and spin the clothes but all the tasks separately. As the whole system is quite complex, we need to abstract the complexities of the subsystems. We need a system that can automate the whole task without the disturbance or interference of us.

Solution using Facade Method

To solve the above-described problem, we would like to hire the Facade Method. It will help us to hide or abstract the complexities of the subsystems as follows.

Following code is written using the Facade Method

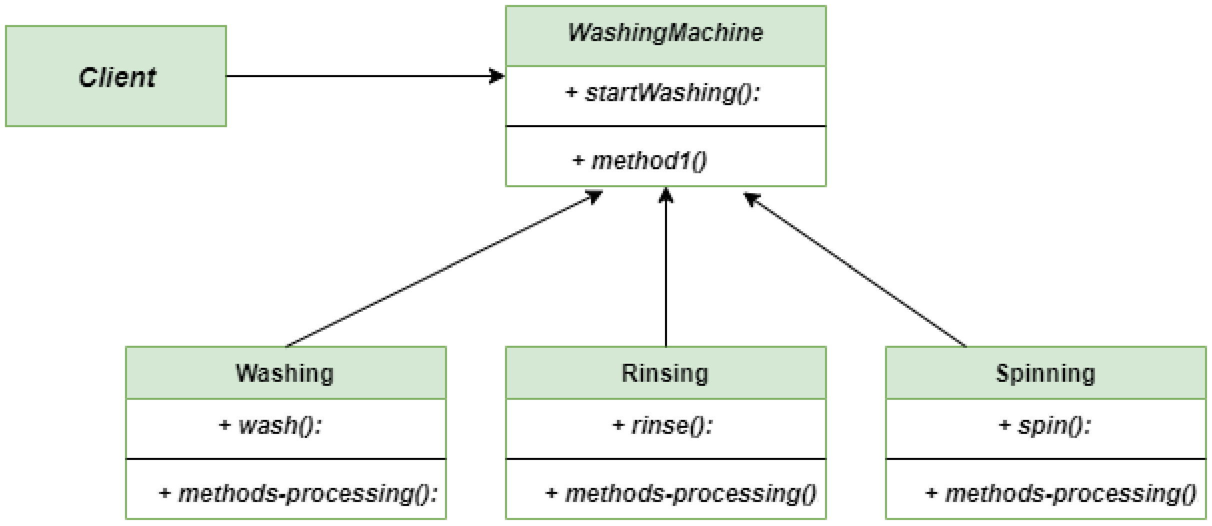
- Python3

```
"""Facade pattern with an example of WashingMachine"""



class Washing:

    '''Subsystem # 1'''



    def wash(self):

        print("Washing...")





class Rinsing:

    '''Subsystem # 2'''



    def rinse(self):

        print("Rinsing...")





class Spinning:
    '''Subsystem # 3'''

    def spin(self):
        print("Spinning...")


class WashingMachine:
    '''Facade'''

    def __init__(self):
        self.washing = Washing()
        self.rinsing = Rinsing()
        self.spinning = Spinning()

    def startWashing(self):
        self.washing.wash()
        self.rinsing.rinse()
        self.spinning.spin()

""" main method """
```

```
if __name__ == "__main__":



    washingMachine = WashingMachine()

    washingMachine.startWashing()
```

Class Diagram for Facade Method

Following is the class diagram for Facade Method:



Facade-method-Class-Diagram

Advantages

- **Isolation:** We can easily isolate our code from the complexity of a subsystem.
- **Testing Process:** Using **Facade Method** makes the process of testing comparatively easy since it has convenient methods for common testing tasks.
- **Loose Coupling:** Availability of loose coupling between the clients and the Subsystems.

Disadvantages

- **Changes in Methods:** As we know that in **Facade method**, subsequent methods are attached to Facade layer and any change in subsequent method may brings change in Facade layer which is not favorable.
- **Costly process:** It is not cheap to establish the Facade method in out application for the system's reliability.
- **Violation of rules:** There is always the fear of violation of the construction of the facade layer.

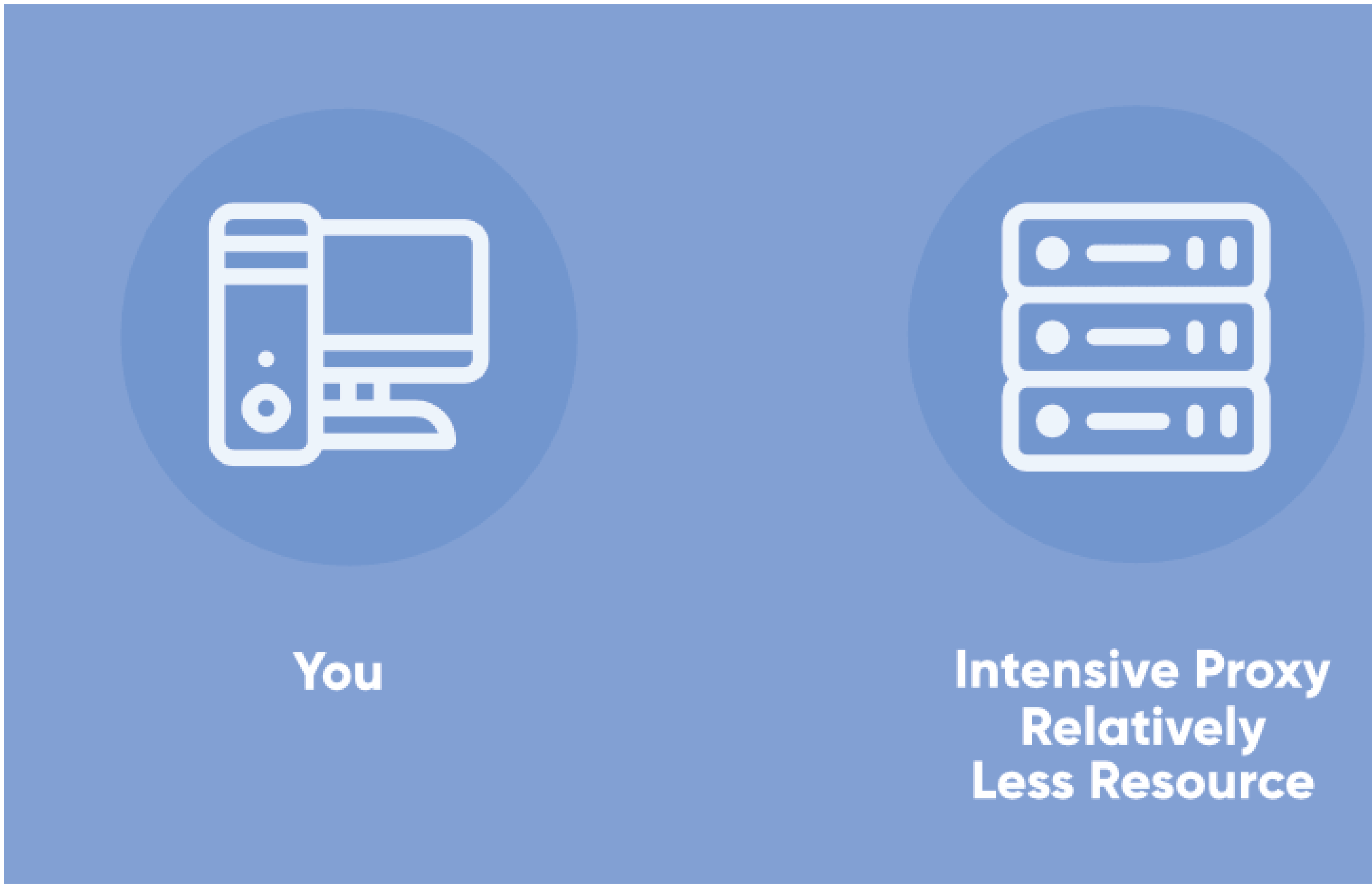
Applicability

- **Providing simple Interface:** One of the most important application of **Facade Method** is that it is used whenever you want to provide the simple interface to the complex sub-system
- **Division into layers:** It is used when we want to provide a unique structure to a sub-system by dividing them into layers. It also leads to loose coupling between the clients and the subsystem.

Proxy Method – Python Design Patterns

The Proxy method is Structural design pattern that allows you to provide the replacement for an another object. Here, we use different classes to represent the functionalities of another class. The most important part is that here we create an object having original object functionality to provide to the outer world.
The meaning of word **Proxy** is "in place of" or "on behalf of" that directly explains the **Proxy Method**.

Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.

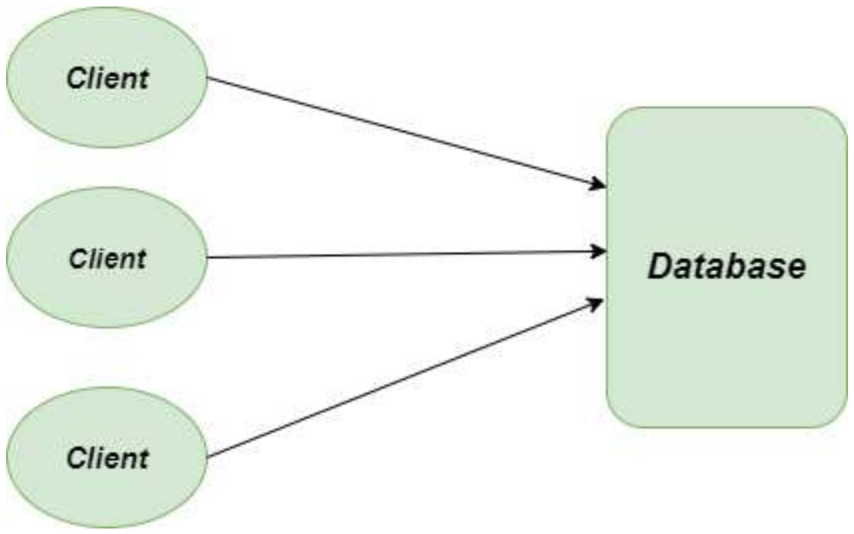


proxy-design-method

A real-world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash and provides a means of accessing that cash when required. And that's exactly what the Proxy pattern does – "Controls and manage access to the object they are protecting".

Problem Without Using Proxy method

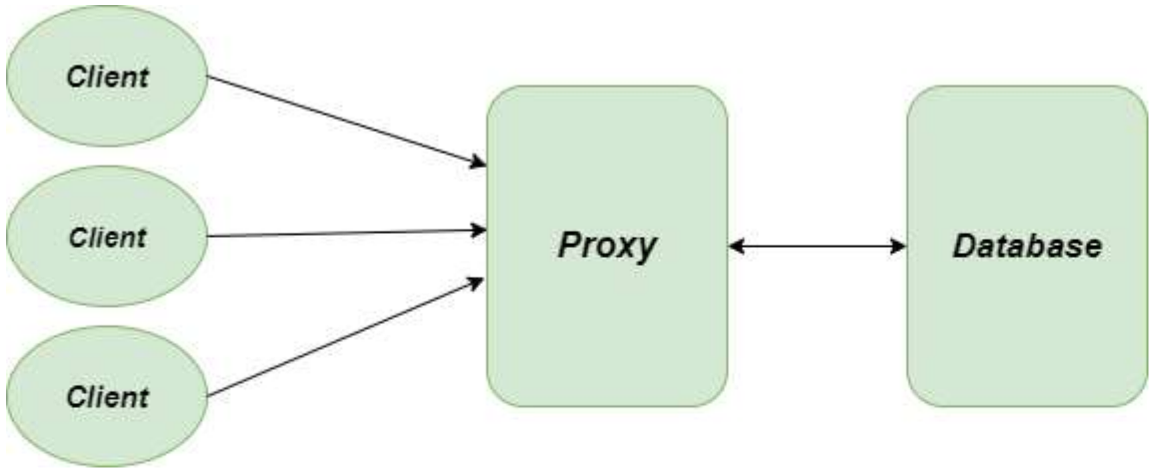
Let's understand the problem by considering the example of the College's Database which takes care of all the student's records. For e.g., we need to find the name of those students from Database whose balance fee if greater than 500. So, if we traverse the whole list of students and for each student object if we make a separate connection to the database then it will be proved as an expensive task.



Proxy-method-Problem

Solution using Proxy Method

Here comes the **Proxy Method** to solve the above-discussed problem. We will create a proxy server or maybe a proxy connection to the database and after that, we don't have to create separate connections to the database for each student object.
We will simply get our needed data using the proxy without wasting a huge amount of memory for the creation of the object.



Proxy-method-Solution

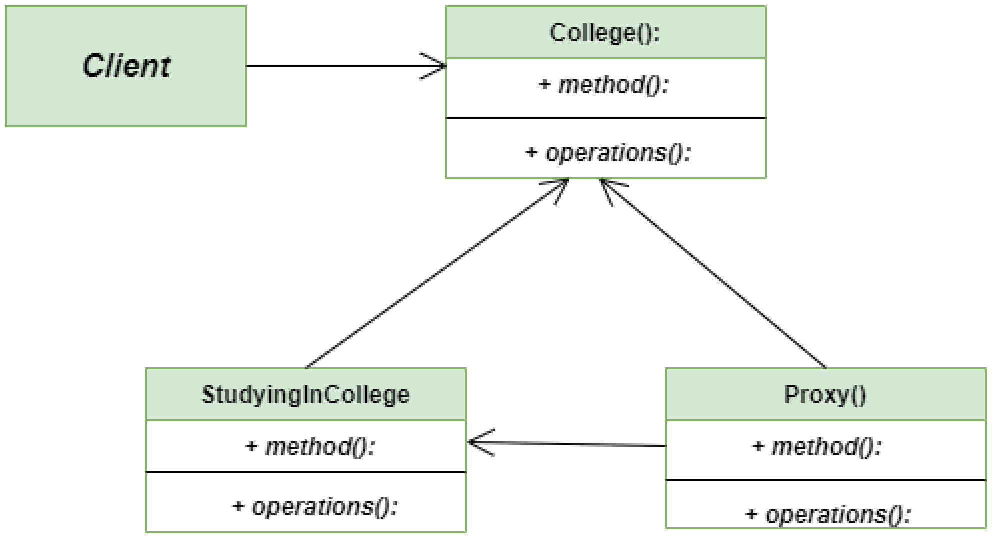
```
class College:
    '''Resource-intensive object'''


    def studyingInCollege(self):
        print("Studying In College....")



class CollegeProxy:
    '''Relatively less resource-intensive proxy acting as middleman.
     Instantiates a College object only if there is no fee due.'''

    def __init__(self):

        self.feeBalance = 1000
        self.college = None

    def studyingInCollege(self):

        print("Proxy in action. Checking to see if the balance of student is clear or not...")
        if self.feeBalance <= 500:
            # If the balance is less than 500, let him study.
            self.college = College()
            self.college.studyingInCollege()
        else:

            # Otherwise, don't instantiate the college object.
            print("Your fee balance is greater than 500, first pay the fee")

"""main method"""

if __name__ == "__main__":
    # Instantiate the Proxy
    collegeProxy = CollegeProxy()

    # Client attempting to study in the college at the default balance of 1000.
    # Logically, since he / she cannot study with such balance,
    # there is no need to make the college object.
    collegeProxy.studyingInCollege()

    # Altering the balance of the student
    collegeProxy.feeBalance = 100

    # Client attempting to study in college at the balance of 100. Should succeed.
    collegeProxy.studyingInCollege()
```

Output:

```
Proxy in action. Checking to see if the balance of student is clear or not...
Your fee balance is greater than 500, first pay the fee

Proxy in action. Checking to see if the balance of student is clear or not...
Studying In College....
```

Class Diagram

Following is the class diagram for the Proxy Method:



Proxy-method-Class-Diagram

Advantages

- **Open/Closed Principle:** Without changing the client code, we can easily introduce the new proxies in our application.
- **Smooth Service:** The proxy that we creates works even when the service object is not ready or is not available at the current scenario.
- **Security:** Proxy method also provides the security to the system.
- **Performance:** It increases the performance of the application by avoiding the duplication of the objects which might be huge size and memory intensive.

Disadvantages

- **Slow response:** It is possible that the service might get slow or delayed.
- **Layer of Abstraction:** This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes
- **Complexity Increases:** Our Code may become highly complicated due to the introduction of lot of new classes.

Applicability

- **Virtual Proxy:** Most importantly used in Databases for example there exist certain heavy resource consuming data in the database and we need it frequently. so. here we can use the proxy pattern which would create multiple proxies and point to the object.
- **Protective Proxy:** It creates a protective layer over the application and can be used in the scenarios of Schools or Colleges where only a few no. of websites are allowed to open with there WiFi.

- **Remote Proxy:** It is particularly used when the service object is located on a remote server. In such cases, the proxy passes the client request over the network handling all the details.
- **Smart proxy:** It is used to provide the additional security to the application by intervene specific actions whenever the object would be accessed.

Flyweight Method – Python Design Patterns

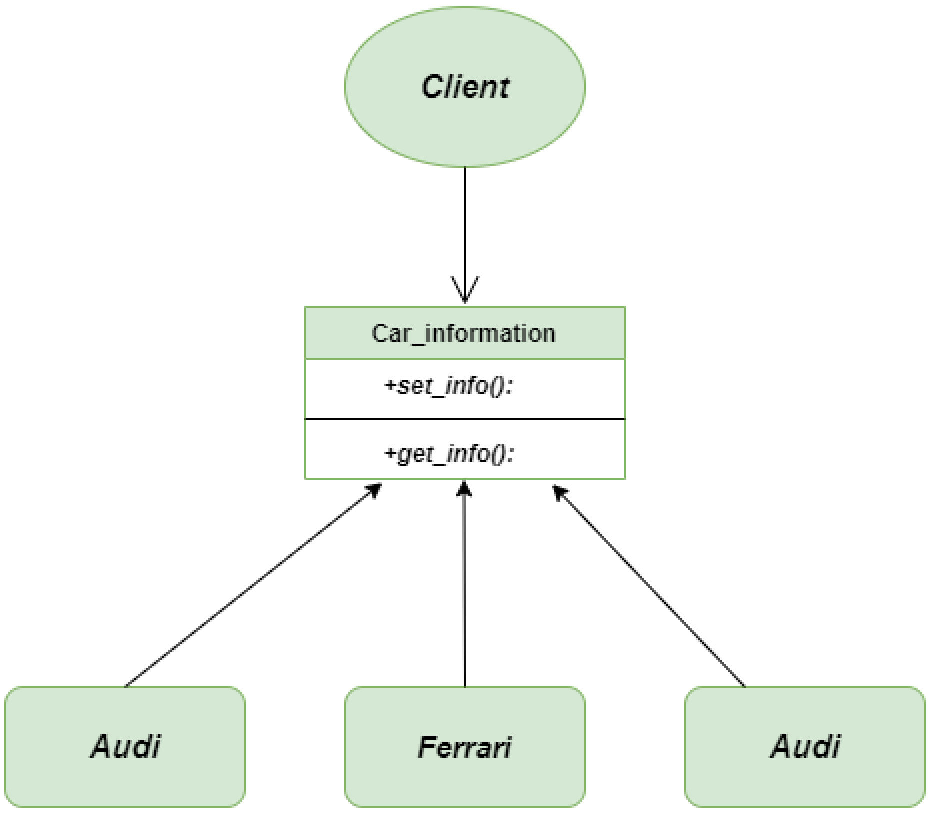
Flyweight method is a **Structural Design Pattern** that focus on minimizing the number of objects that are required by the program at the run-time. Basically, it creates a Flyweight object which is shared by multiple contexts. It is created in such a fashion that you can not distinguish between an object and a Flyweight Object. One important feature of flyweight objects is that they are **immutable**. This means that they cannot be modified once they have been constructed.
To implement the Flyweight method in **Python**, we use **Dictionary** that stores reference to the object which have already been created, every object is associated with a key.

Why we do care for the number of objects in our program ?

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory
- Although creating an object in Python is really fast, we can still reduce the execution time of our program by sharing objects.

Problem without using Flyweight Method

Imagine you are a game developer who likes **Racing Games** much and also wants to develop a racing game for you and your friend. As you are a flawless Game developer, you created one and start enjoying the game. Then you sent the game to your friend also but he did not enjoy the game too much because the game kept crashing after every few minutes.
But Why? (Guess the reason if you think you are a Pro Game Developer).After debugging for several hours, you found that the issue is lack of **RAM** on your friend's system. Your system is much powerful as compared to your friend's system that's why the game was running smoothly on your system but not on your friend's system.



Flyweight-problem-Diagram

Solution using Flyweight Method

so, what will you do as a developer to improve the performance? (of course! not going to upgrade the RAM). The actual problem is related to car objects because each car is represented by separate objects containing plenty of data related to its color, size, seats, maximum speed, etc. Whenever your RAM got filled and unable to add more new objects which are required currently, your game gets crashed. For avoiding such situations in applications, it is the prior duty of the developer to use **Flyweight Method** which allows you to fit more objects into the available amount of RAM by sharing common parts of the objects.

Following Code is written using the Flyweight method

- Python3

```
class ComplexCars(object):
    """Separate class for Complex Cars"""

    def __init__(self):

        pass

    def cars(self, car_name):

        return "ComplexPattern[% s]" % (car_name)



class CarFamilies(object):

    """dictionary to store ids of the car"""

    car_family = {}

    def __new__(cls, name, car_family_id):
        try:
            id = cls.car_family[car_family_id]

        except KeyError:

            id = object.__new__(cls)

            cls.car_family[car_family_id] = id

        return id


    def set_car_info(self, car_info):


        """set the car information"""


        cg = ComplexCars()

        self.car_info = cg.cars(car_info)


    def get_car_info(self):


        """return the car information"""


        return (self.car_info)
```

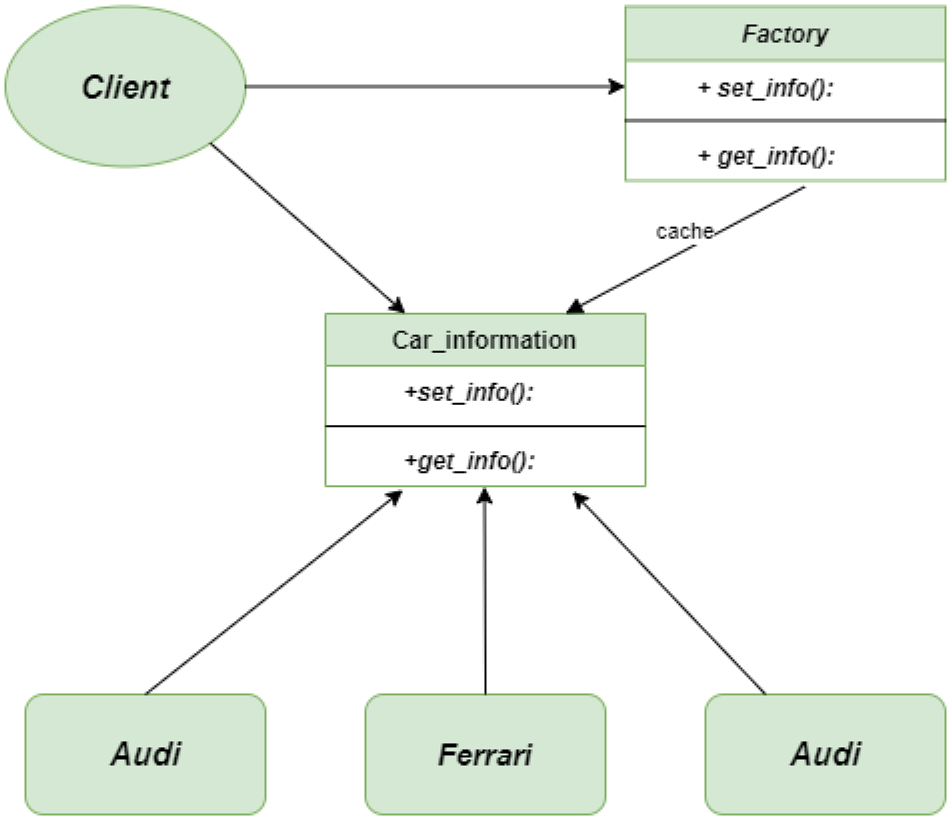
```
if __name__ == '__main__':

    car_data = (('a', 1, 'Audi'), ('a', 2, 'Ferrari'), ('b', 1, 'Audi'))

    car_family_objects = []

    for i in car_data:

        obj = CarFamilies(i[0], i[1])

        obj.set_car_info(i[2])

        car_family_objects.append(obj)



    """similar id's says that they are same objects """



    for i in car_family_objects:

        print("id = " + str(id(i)))

        print(i.get_car_info())
```

Output

```
id = 58598800
ComplexPattern[Audi]
id = 58598832
ComplexPattern[Ferrari]
id = 58598800
ComplexPattern[Audi]
```

Class diagram

Following is the class diagram for the Flyweight method



Flyweight-class-diagram

Advantages

- **Reduced use of RAM:** when we have a lot of similar objects present in our application, its always better to use **Flyweight method** inorder to save a lot of space in RAM
- **Improved Data Caching:** When the need of client or user is High response time, it is always preferred to use Flyweight method because it helps in improving the Data Caching.
- **Improved performance:** It ultimately leads to improve in performance because we are using less number of heavy objects.

Disadvantages

- **Breaking Encapsulation:** Whenever we try to move the state outside the object, we do breaking of encapsulation and may become less efficient then keeping the state inside the object.
- **Hard to handle:** Usage of Flyweight method depends upon the language we use, easy to use in language like Python, Java where all object variables are references but typical to use in language like C, C++ where objects can be allocated as local variables on the stack and destroyed as a result of programmer action.
- **Complicated Code:** Using Flyweight method always increases the complexity of the code to understand for the new developers.

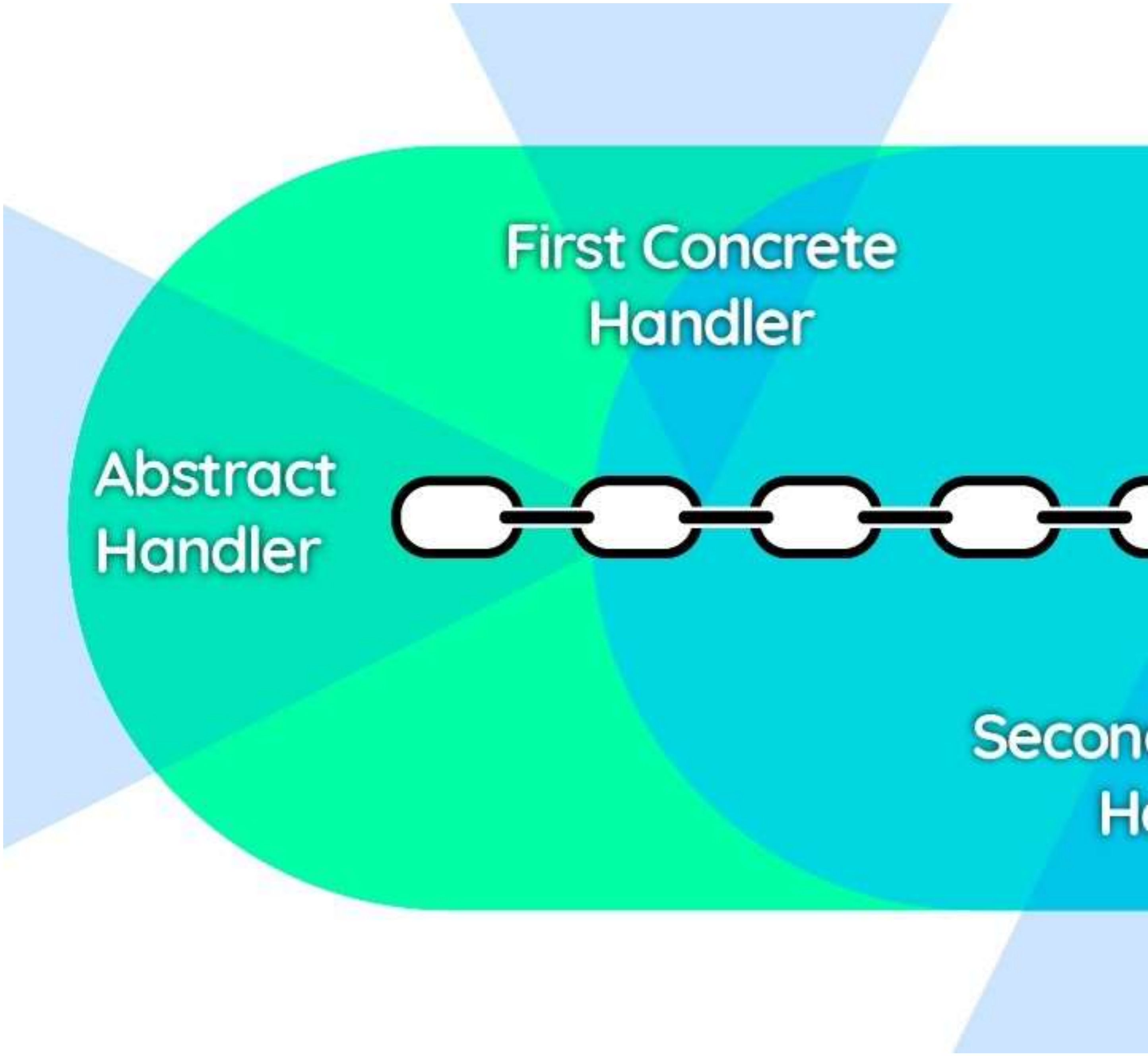
Applicability

- **To Reduce the number of Objects:** Generally, Flyweight method is used when our application has a lot of heavy weight objects, to solve this problem we use Flyweight method to get rid of unnecessary memory consumption.
- **Object independent Applications:** When our application if independent of the object created, then we can make use of this method inorder to save lot of machine space.
- **Project Cost Reduction:** When it is required to reduce the cost of project in terms of space and time complexity, it is always preferred to use the Flyweight method.

Chain of Responsibility – Python Design Patterns

Chain of Responsibility method is **Behavioral design pattern** and it is the object-oriented version of **if … elif … elif … else** and make us capable to rearrange the condition-action blocks dynamically at the run-time. It allows us to pass the requests along the chain of handlers. The processing is simple, whenever any handler received the request it has two choices either to process it or pass it to the next handler in the chain.
This pattern aims to decouple the senders of a request from its receivers by

allowing the request to move through chained receivers until it is handled.



chain-of-responsibility-method

Problem without using Chain of Responsibility Method

Imagine you are building a simple website that takes input strings and tells about the various properties of the strings such as is the string palindrome? is string upperCase? is string lowerCase? and many other properties too. After the complete planning, you decide that these checks for the input string should be performed sequentially. So, here the problem arises for the developer that he/she has to implement such an application that can decide on run-time which action should be performed next.

Solution using Chain of Responsibility Method

Chain of Responsibility Method provides the solution for the above-described problem. It creates a separate **Abstract handler** which is used to handle the sequential operations which should be performed dynamically. For eg., we create four handlers named as **FirstConcreteHandler, SecondConcreteHandler, ThirdConcreteHandler**, and **Defaulthandler** and calls them sequentially from the user class.

- Python3

```
class AbstractHandler(object):

    """Parent class of all concrete handlers"""

    def __init__(self, nxt):

        """change or increase the local variable using nxt"""

        self._nxt = nxt
```

```
def handle(self, request):

    """It calls the processRequest through given request"""

    handled = self.processRequest(request)

    """case when it is not handled"""

    if not handled:
        self._nxt.handle(request)

def processRequest(self, request):

    """throws a NotImplementedError"""

    raise NotImplementedError('First implement it !')
```

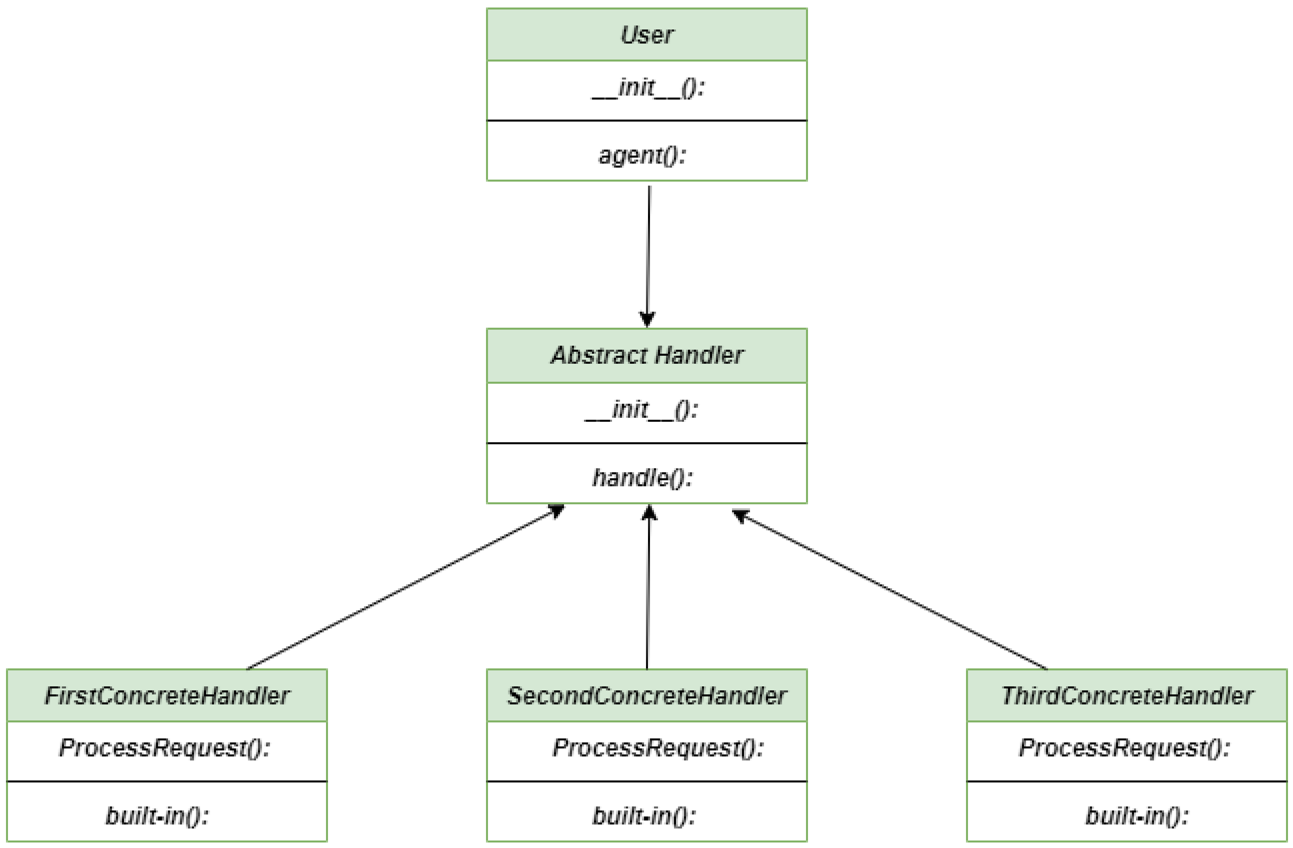
```
class FirstConcreteHandler(AbstractHandler):

    """Concrete Handler # 1: Child class of AbstractHandler"""

    def processRequest(self, request):

        '''return True if request is handled '''

        if 'a' < request <= 'e':
            print("This is {} handling request
'{}'".format(self.__class__.__name__, request))
            return True


class SecondConcreteHandler(AbstractHandler):

    """Concrete Handler # 2: Child class of AbstractHandler"""

    def processRequest(self, request):
        '''return True if the request is handled'''

        if 'e' < request <= 'l':
            print("This is {} handling request
'{}'".format(self.__class__.__name__, request))
            return True

class ThirdConcreteHandler(AbstractHandler):

    """Concrete Handler # 3: Child class of AbstractHandler"""

    def processRequest(self, request):

        '''return True if the request is handled'''

        if 'l' < request <= 'z':
            print("This is {} handling request
'{}'".format(self.__class__.__name__, request))
            return True

class DefaultHandler(AbstractHandler):
    """Default Handler: child class from AbstractHandler"""

    def processRequest(self, request):

        """Gives the message that th request is not handled and returns
true"""

        print("This is {} telling you that request '{}' has no handler
right now.".format(self.__class__.__name__,
                request))
        return True


class User:

    """User Class"""

    def __init__(self):
        """Provides the sequence of handles for the users"""

        initial = None

        self.handler = 
FirstConcreteHandler(SecondConcreteHandler(ThirdConcreteHandler(DefaultHand
ler(initial))))

    def agent(self, user_request):

        """Iterates over each request and sends them to specific handles"""

        for request in user_request:
            self.handler.handle(request)

"""main method"""

if __name__ == "__main__":

    """Create a client object"""
    user = User()
    """Create requests to be processed"""



    string = "GeeksforGeeks"

    requests = list(string)



    """Send the requests one by one, to handlers as per the sequence of
handlers defined in the Client class"""

    user.agent(requests)
```

Output

```
This is DefaultHandler telling you that request 'G' has no handler
right now.
This is FirstConcreteHandler handling request 'e'
This is FirstConcreteHandler handling request 'e'
This is SecondConcreteHandler handling request 'k'
This is ThirdConcreteHandler handling request 's'
This is SecondConcreteHandler handling request 'f'
This is ThirdConcreteHandler handling request 'o'
This is ThirdConcreteHandler handling request 'r'
This is DefaultHandler telling you that request 'G' has no handler
right now.
This is FirstConcreteHandler handling request 'e'
This is FirstConcreteHandler handling request 'e'
This is SecondConcreteHandler handling request 'k'
This is ThirdConcreteHandler handling request 's'
```

Class Diagram

Class Diagram for Chain of Responsibility Method



Chain-Of-Responsibility_class_diagram

Advantages

- **Single Responsibility Principle:** It's easy to decouple the classes here that invoke operations from classes that perform operations.
- **Open/Closed principle:** We can introduce the new code classes without breaking th existing client code.
- **Increases Flexibility:** While giving the responsibilities to the objects, it increases the flexibility of the code.

Disadvantages

- **Unassured about request:** This method doesn't provide any assurance whether the object will be received or not.
- **Spotting characteristics:** Due to debugging, it becomes difficult task to observe the characteristics of operations.
- **Depreciated System Performance:** It might affect the system's performance due to continuous cycle calls

Applicability

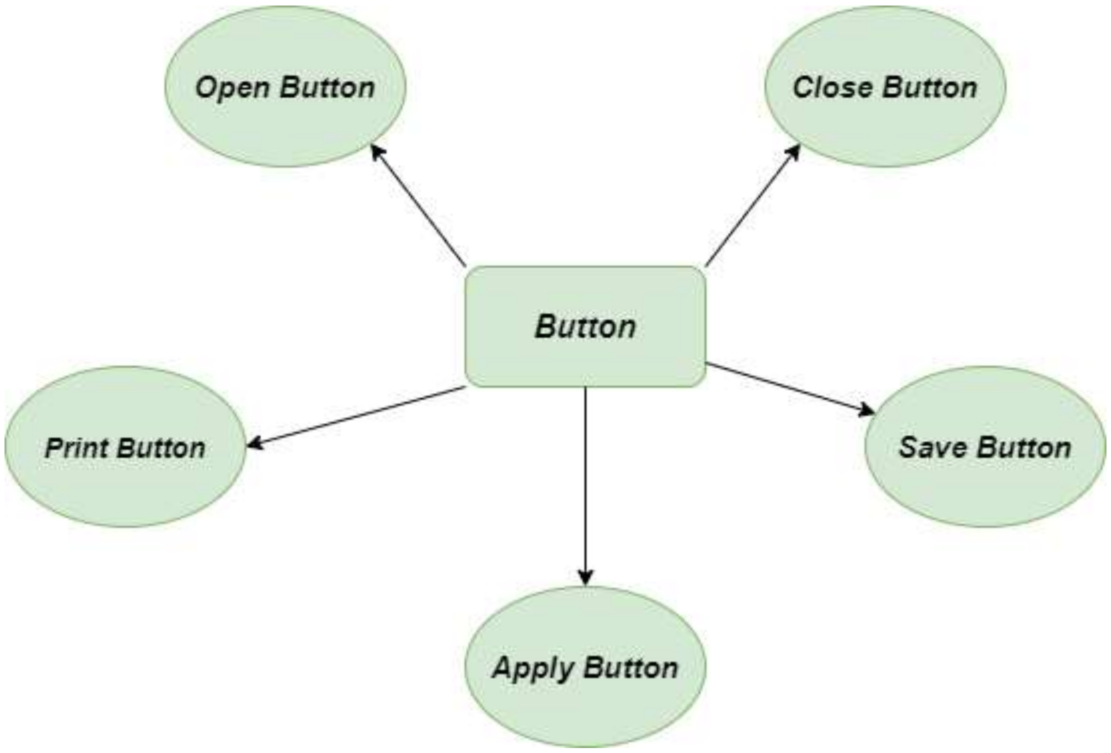
- **Processing several handlers in order:** Chain of responsibility method helps very much when it is required to process several handlers in a particular order because the linking is possible in any order
- **Decoupling requests:** This method is generally used when you want to decouple the request's sender and receiver.
- **Unspecified handlers:** When you don't want to specify handlers in the code, it is always preferred to use the **Chain of Responsibility**.

Command Method – Python Design Patterns

Command Method is **Behavioral Design Pattern** that encapsulates a request as an object, thereby allowing for the parameterization of clients with different requests and the queuing or logging of requests. Parameterizing other objects with different requests in our analogy means that the button used to turn on the lights can later be used to turn on stereo or maybe open the garage door. It helps in promoting the "invocation of a method on an object" to full object status. Basically, it encapsulates all the information needed to perform an action or trigger an event.

Problem without using Command Method

Imagine you are working on a code editor. Your current task is to add the new buttons in the toolbar of the editor for various different operations. It's definitely easy to create a single **Button Class** that can be used for the buttons. As we know that all the buttons used in the editor look similar, so what should we do? Should we create a lot of sub-classes for each place where the button is used?



Problem-without-Command-method

Solution Using Command Method

Let’s have a look at the solution for the above-described problem. It’s always a good idea to divide the software into different layers which helps in easy coding as well as debugging. The command pattern suggests that objects shouldn’t send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate command class with a single method that triggers this request.

- Python3

```
"""Use built-in abc to implement Abstract classes and methods"""

from abc import ABC, abstractmethod
```

```
"""Class Dedicated to Command"""

class Command(ABC):



    """constructor method"""

    def __init__(self, receiver):

        self.receiver = receiver



    """process method"""

    def process(self):

        pass



"""Class dedicated to Command Implementation"""

class CommandImplementation(Command):



    """constructor method"""

    def __init__(self, receiver):

        self.receiver = receiver



    """process method"""

    def process(self):
        self.receiver.perform_action()


"""Class dedicated to Receiver"""

class Receiver:


    """perform-action method"""

    def perform_action(self):

        print('Action performed in receiver.')


"""Class dedicated to Invoker"""

class Invoker:


    """command method"""

    def command(self, cmd):

        self.cmd = cmd


    """execute method"""

    def execute(self):

        self.cmd.process()
```

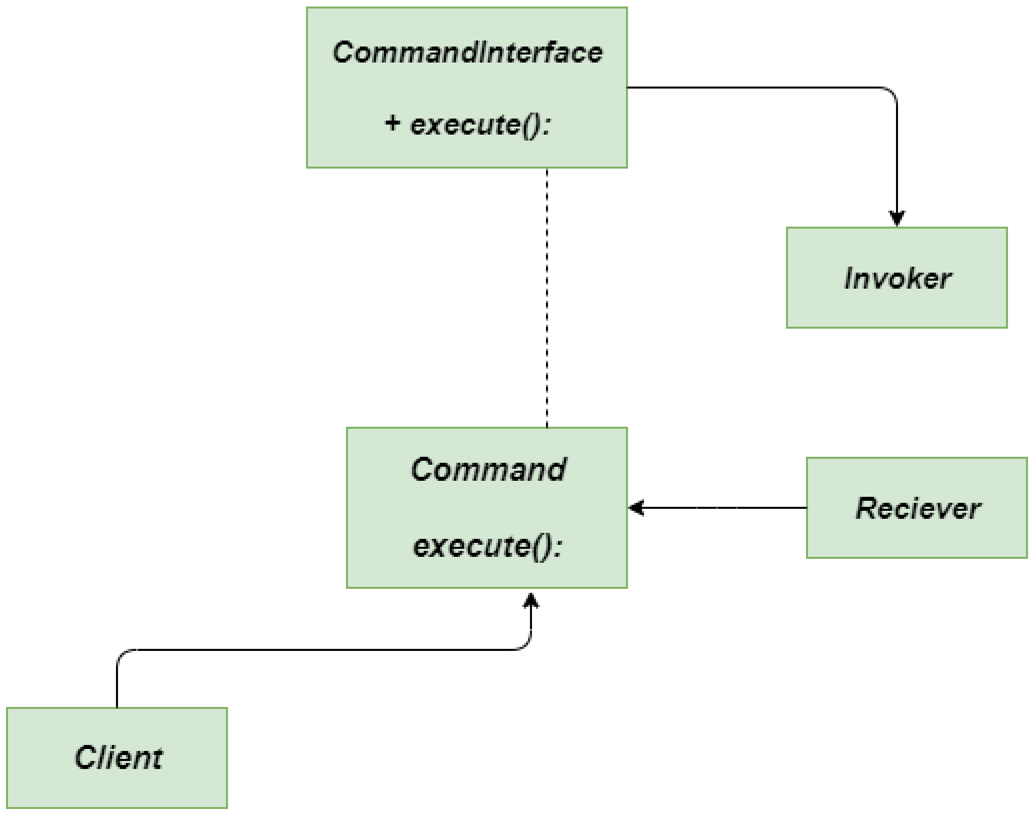
```
"""main method"""

if __name__ == "__main__":



    """create Receiver object"""

    receiver = Receiver()

    cmd = CommandImplementation(receiver)

    invoker = Invoker()

    invoker.command(cmd)

    invoker.execute()
```

Output

```
Action performed in receiver.
```

Class Diagram

Following is the class diagram for the Command method



Class-diagram-Command-Method

Advantages

- **Open/Closed Principle:** We can introduce the new commands into the application without breaking the existing client's code.
- **Single Responsibility Principle:** It's really easy to decouple the classes here that invoke operations from other classes.
- **Implementable UNDO/REDO:** It's possible to implement the functionalities of **UNDO/REDO** with the help of Command method.
- **Encapsulation:** It helps in encapsulating all the information needed to perform an action or an event.

Disadvantages

- **Complexity Increases:** The complexity of the code increases as we are introducing certain layers between the senders and the receivers.
- **Quantity of classes increases:** For each individual command, the quantity of the classes increases.

- **Concrete Command:** Every individual command is a **ConcreteCommand** class that increases the volume of the classes for implementation and maintenance.

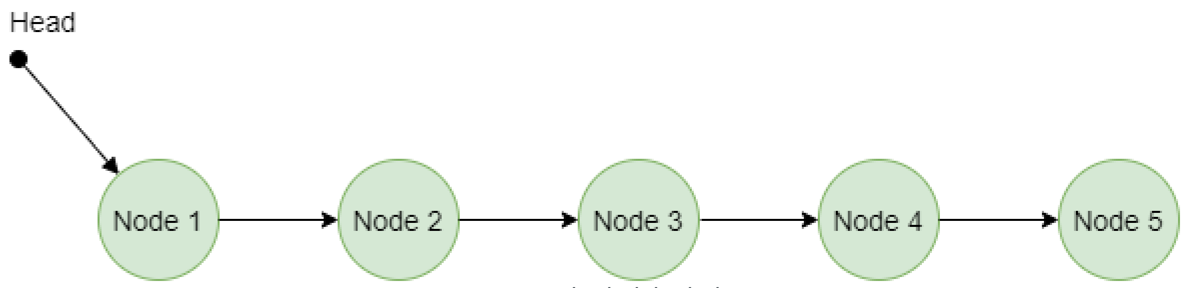
Applicability

- **Implementing Reversible operations:** As the Command method provides the functionalities for **UNDO/REDO** operations, we can possibly reverse the operations.
- **Parameterization:** It's always preferred to use Command method when we have to parameterize the objects with the operations.

Iterator Method – Python Design Patterns

Iterator method is a **Behavioral Design Pattern** that allows us to traverse the elements of the collections without taking the exposure of in-depth details of the elements. It provides a way to access the elements of complex data structure sequentially without repeating them.
According to **GangOfFour**, Iterator Pattern is used **" to access the elements of an aggregate object sequentially without exposing its underlying implementation"**.

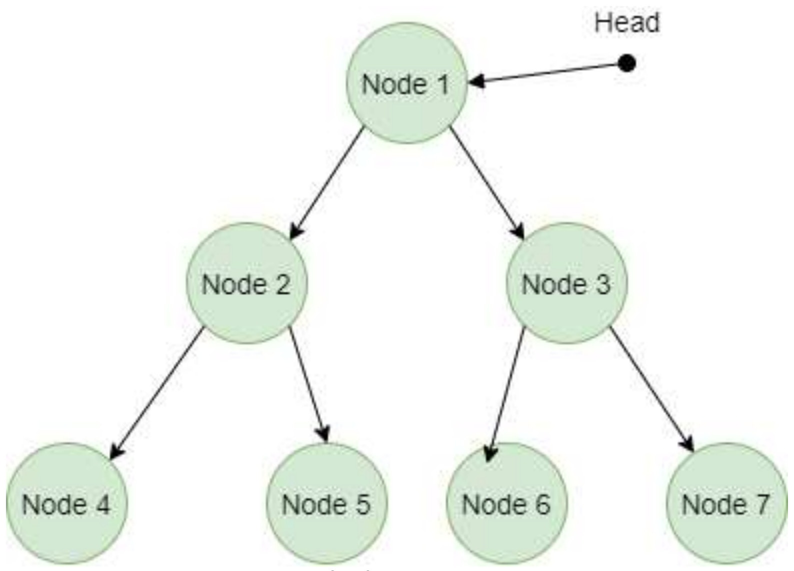
The following diagram depicts the Linked List data structure.



Iterator-method-Linked-List

Problem Without using Iterator Method

Imagine you are creating an application for small kids which takes any valid alphabet as input and return all the alphabets up to that. When this application will be used only a few times, it is okay to run **For loop** and **While loop** again and again but when the frequency of running increases this process becomes quite inefficient. So, we have to find a way in order to avoid these loops. This problem may become bigger when we will work on complex non-linear data structures like Trees, Graphs where traversing is not that simple as in an array. The following diagram depicts the image of the Tree data structure.



Iterator-Method-Tree-Data-Structure

Solution Using Iterator Method

Here we will discuss the solution for the above-described problem. It's always handy for Python users to use **Iterators** for traversing any kind of data structure doesn't matter they are linear or no-linear data structures. We have two options to implement Iterators in Python either we can use the in-built iterators to produce the fruitful output or explicitly we can create iterators with the help of **Generators**. In the following code, we have explicitly created the Iterators with the help of generators.

Note: Following code is the example of an explicitly created Iterator method

- Python3

```
""" helper method for iterator"""
```

```
def alphabets_upto(letter):

    """Counts by word numbers, up to a maximum of five"""

    for i in range(65, ord(letter)+1):

            yield chr(i)




"""main method"""

if __name__ == "__main__":



    alphabets_upto_K = alphabets_upto('K')

    alphabets_upto_M = alphabets_upto('M')



    for alpha in alphabets_upto_K:

        print(alpha, end=" ")



    print()



    for alpha in alphabets_upto_M:

        print(alpha, end=" ")
```

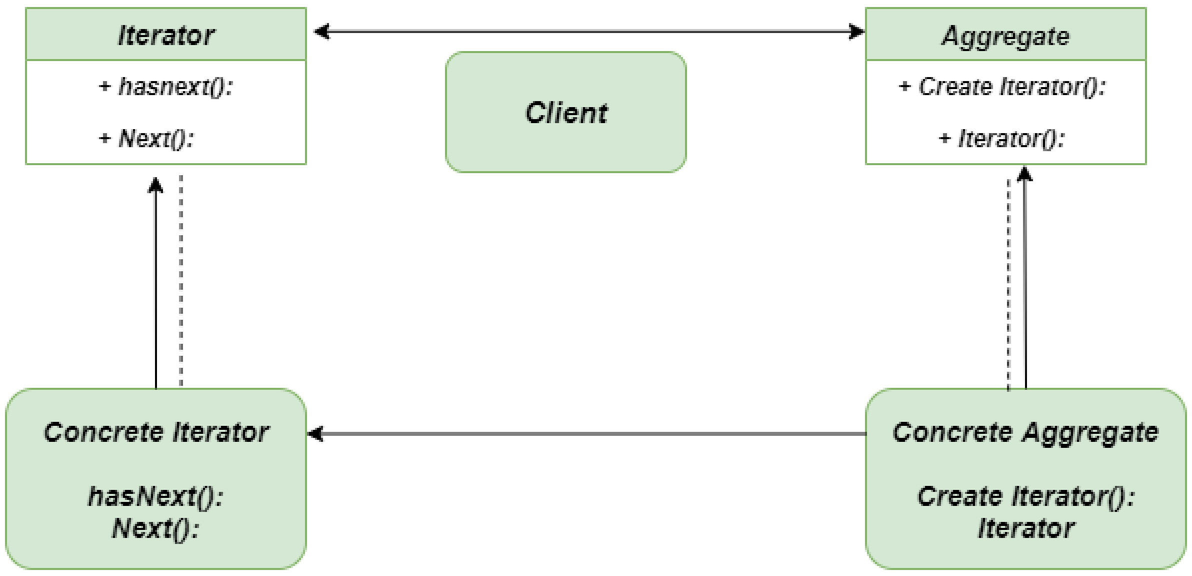
Note: Following code is the example of using an in-built iterator method

- Python3

```
"""utility function"""
def inBuilt_Iterator1():


    alphabets = [chr(i) for i in range(65, 91)]


    """using in-built iterator"""
    for alpha in alphabets:
        print(alpha, end = " ")
    print()


"""utility function"""
def inBuilt_Iterator2():


    alphabets = [chr(i) for i in range(97, 123)]


    """using in-built iterator"""
    for alpha in alphabets:
        print(alpha, end = " ")

    print()



"""main method"""

if __name__ == "__main__" :



    """call the inbuiltIterators"""

    inBuilt_Iterator1()

    inBuilt_Iterator2()
```

Class Diagram

Following is the class diagram for the Iterator Method



Iterator-Method-Class-Diagram

Advantages

- **Single Responsibility Principle:** It’s really easy to extract the huge algorithms into separate classes in the Iterator method.
- **Open/Closed Principle:** Passing the new iterators and collections into the client code does not break the code can easily be installed into it.
- **Easy to use Interface:** It makes the interface really simple to use and also supports the variations in the traversal of the collections.

Disadvantages

- **Unnecessary Wasting resources:** It’s not always a good habit to use the **Iterator Method** because sometimes it may prove as an overkill of resources in a simple application where complex things are not required.
- **Increases Complexity:** As we said earlier, using the Iterator method makes simple applications complex.
- **Decreases Efficiency:** Accessing elements directly is a much better option as compared to accessing elements using the iterator in terms of efficiency.

Applicability

- **Limited Exposure:** When you want to access the elements at the lower level i.e., you are not interested in the internal implementation of the elements then it’s always preferred to use the Iterator Method.
- **Traversing Unknown Data Structures:** The iterator method can be easily used to traverse various types of data structures such as Trees, Graphs, Stack, Queue, etc. as the code provides a couple of generic interfaces for both collections and iterators.

Mediator Method – Python Design Pattern

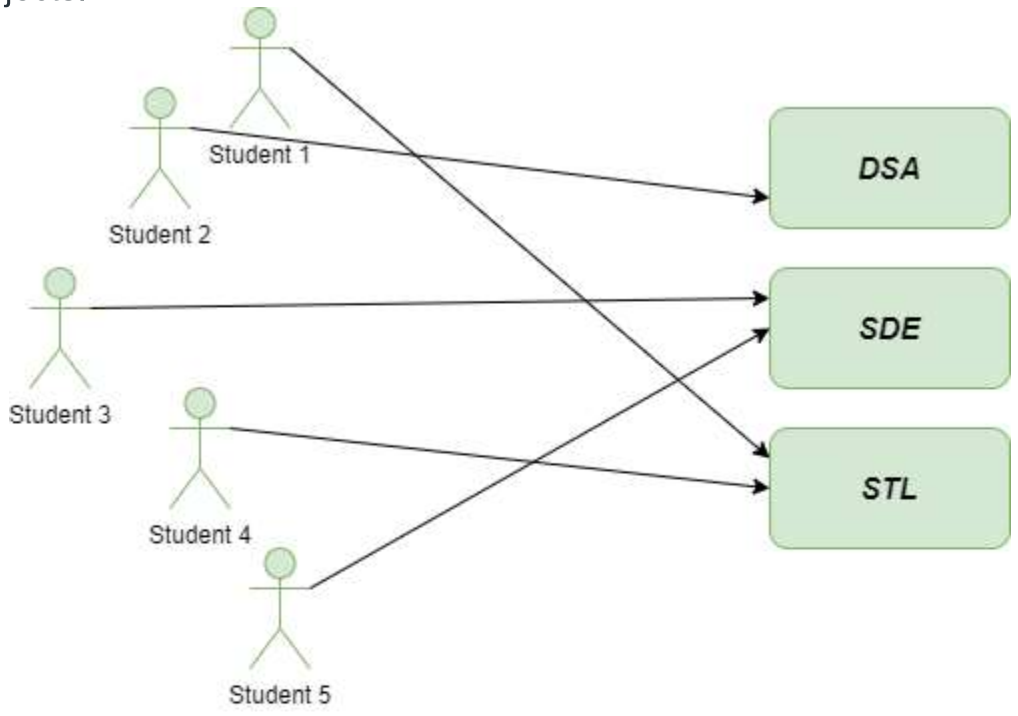
Mediator Method is a **Behavioral Design Pattern** that allows us to reduce the unordered dependencies between the objects. In a mediator environment, objects take the help of mediator objects to communicate with each other. It reduces coupling by reducing the dependencies between communicating objects. The mediator works as a router between objects and it can have it’s own logic to provide a way of communication.

Design Components:

- **Mediator:** It defines the interface for communication between colleague objects.
- **Concrete Mediator:** It implements the mediator interface and coordinates communication between colleague objects.
- **Colleague:** It defines the interface for communication with other colleagues
- **Concrete Colleague:** It implements the colleague interface and communicates with other colleagues through its mediator.

Problem Without using Mediator Method

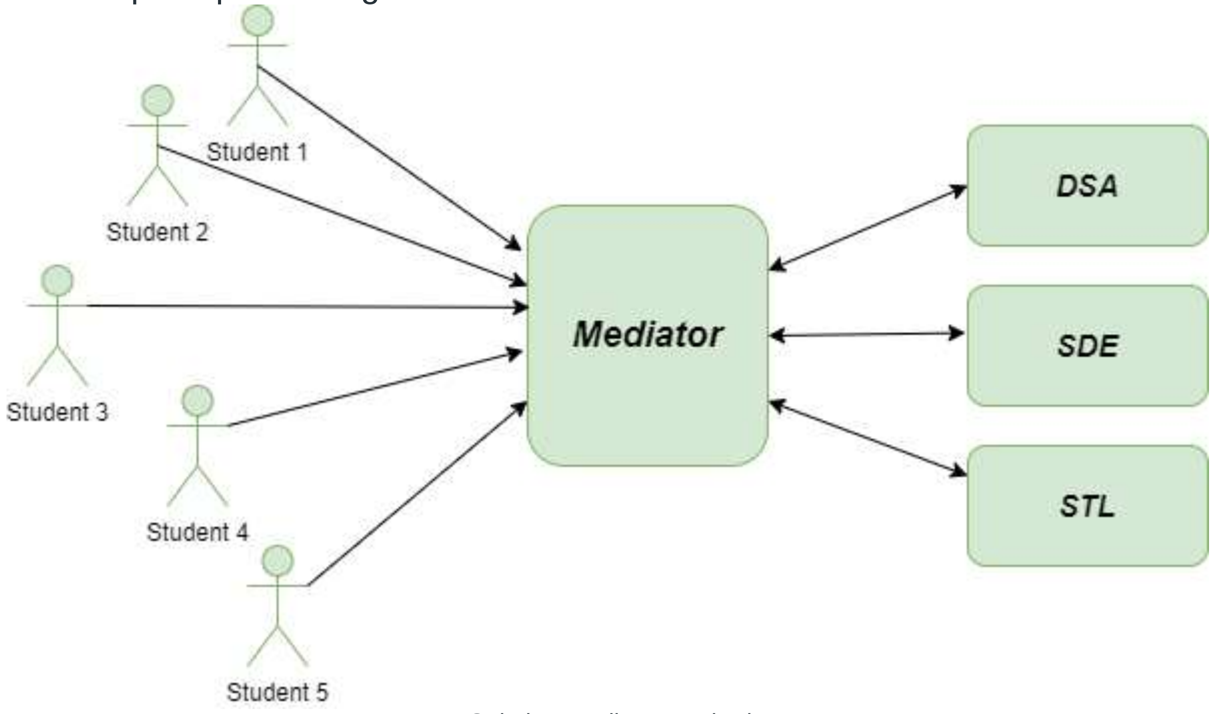
Imagine you are going to take admission in one of the elite courses offered by GeeksforGeeks such as **DSA, SDE**, and **STL**. Initially, there are very few students who are approaching to join these courses. Initially, the developer can create separate objects and classes for the connection between the students and the courses but as the courses become famous among students it becomes hard for developers to handle such a huge number of sub-classes and their objects.



Problem-Mediator-Method

Solution using Mediator Method

Now let us understand how a pro developer will handle such a situation using the **Mediator design pattern**. We can create a separate mediator class named as **Course** and a **User Class** using which we can create different objects of Course class. In the main method, we will create a separate object for each student and inside the User class, we will create the object for Course class which helps in preventing the unordered code.



Solution-mediator-method

- Python3

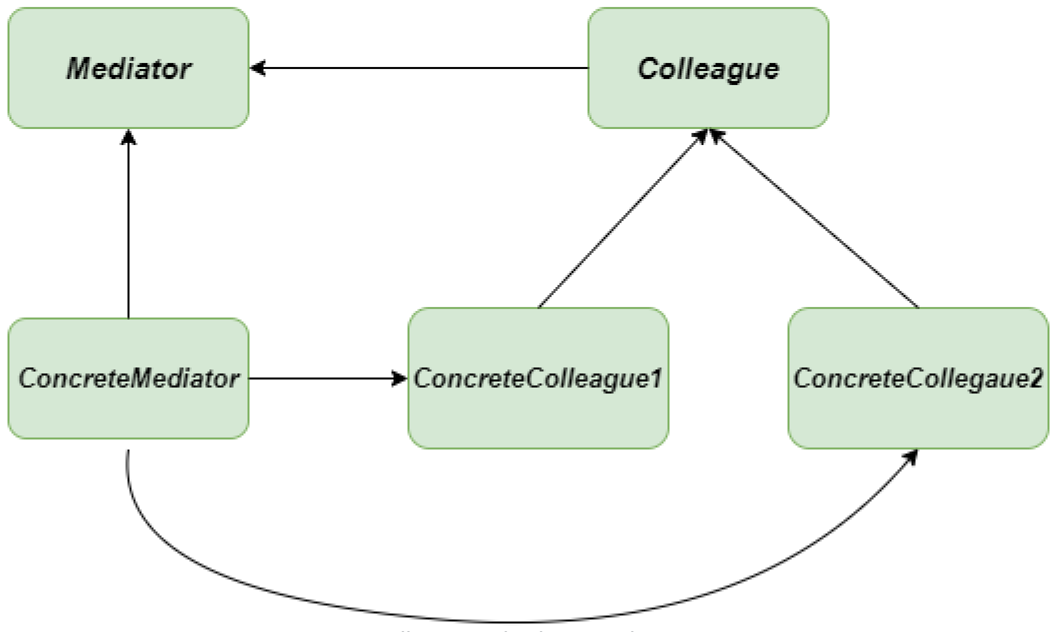
```
class Course(object):

    """Mediator class."""



    def displayCourse(self, user, course_name):

        print("[{}'s course ]: {}".format(user, course_name))
```

```
class User(object):

    '''A class whose instances want to interact with each other.'''


    def __init__(self, name):

        self.name = name

        self.course = Course()


    def sendCourse(self, course_name):

        self.course.displayCourse(self, course_name)


    def __str__(self):

        return self.name


"""main method"""


if __name__ == "__main__":


    mayank = User('Mayank')   # user object
```

```
lakshya = User('Lakshya') # user object

krishna = User('Krishna') # user object



mayank.sendCourse("Data Structures and Algorithms")

lakshya.sendCourse("Software Development Engineer")

krishna.sendCourse("Standard Template Library")
```

UML Diagram

Following is the UML Diagram for Mediator Method:



Mediator-method-UML-Diagram

Advantages

- **Single Responsibility Principle:** Extracting the communications between the various components is possible under **Mediator Method** into a single place which is easier to maintain.

- **Open/Closed Principle:** It's easy to introduce new mediators without disturbing the existing client code.
- **Allows Inheritance:** We can reuse the individual components of the mediators as it follows the **Inheritance**
- **Few Sub-Classes:** Mediator limits the Sub-Classing as a mediator localizes the behavior that otherwise would be disturbed among the several objects.

Disadvantages

- **Centralization:** It completely centralizes the control because the mediator pattern trades complexity of interaction for complexity in the mediator.
- **God Object:** A Mediator can be converted into a **God Object** (an object that knows too much or does too much).
- **Increased Complexity:** The structure of the mediator object may become too much complex if we put too much logic inside it.

Applicability

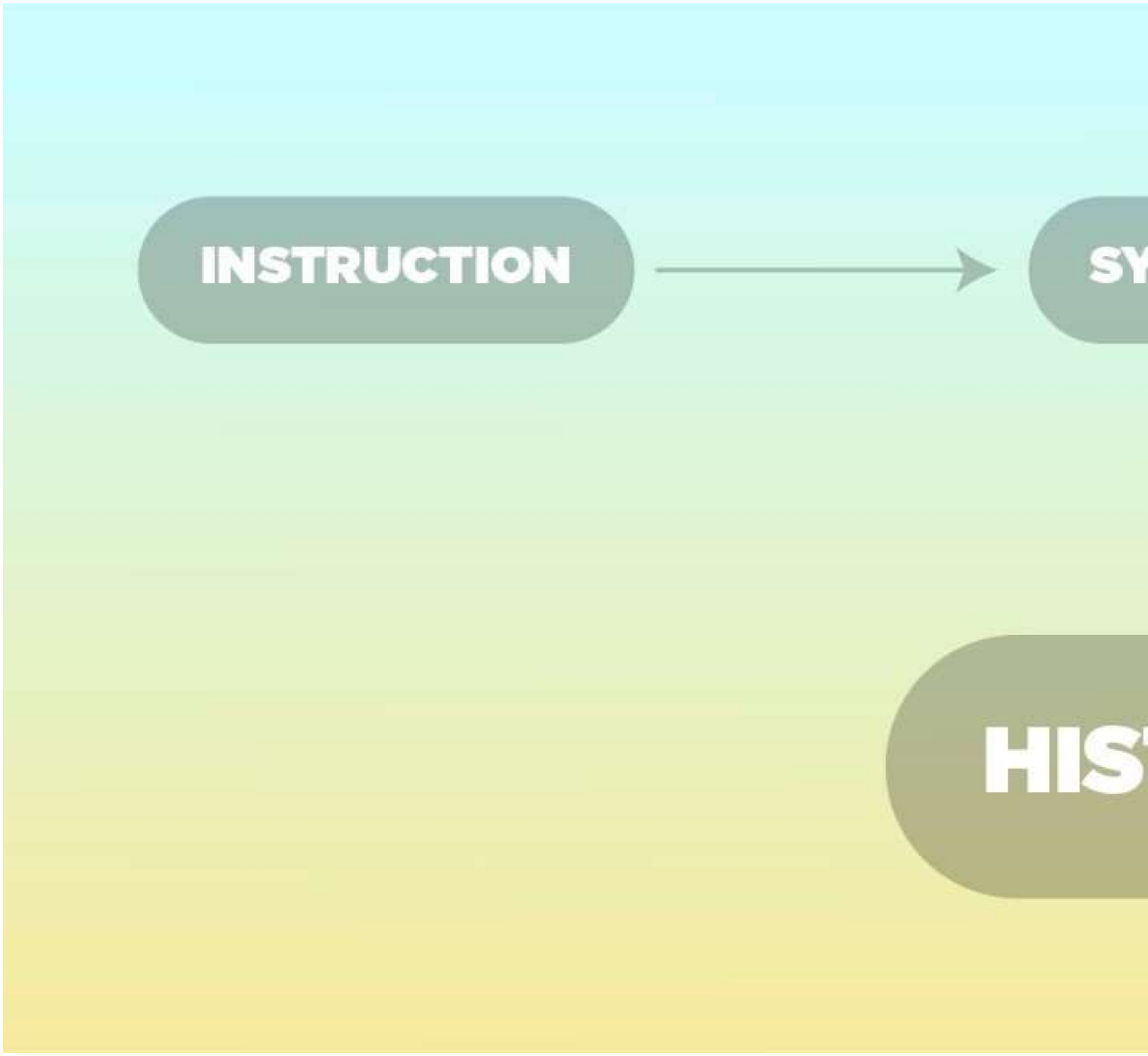
- **Reduce the number of sub-classes:** When you have realized that you have created a lot of unnecessary sub-classes, then it is preferred to use the Mediator method to avoid these unnecessary sub-classes.
- **Air Traffic Controller:** Air traffic controller is a great example of a mediator pattern where the airport control room works as a mediator for communication between different flights.

Memento Method – Python Design Patterns

Memento Method is a **Behavioral Design pattern** which provides the ability to restore an object to its previous state. Without revealing the details of concrete implementations, it allows you to save and restore the previous version of the object. It tries not to disturb the encapsulation of the code and allows you to capture and externalize an object's internal state.

Problem without using Memento Method

Imagine you are a student who wants to excel in the world of competitive programming but you are facing one problem i.e., finding a nice **Code Editor** for programming but none of the present code editors fulfills your needs so you trying to make one for yourself. One of the most important features of any Code Editor is **UNDO** and **REDO** which essentially you also need. As an inexperienced developer, you just used the direct approach of storing all the performed actions. Of course, this method will work but Inefficiently!



Memento-Problem-diagram

Solution by Memento Method

Let’s discuss the solution to the above-discussed problem. The whole problem can be easily resolved by not disturbing the encapsulation of the code. The problem arises when some objects try to perform extra tasks that are not assigned to them, and due to which they invade the private space of other objects. The Memento pattern represents creating the state snapshots to the actual owner of that state, the originator object. Hence, instead of other objects trying to copy the editor’s state from the “outside, ” the editor class itself can make the snapshot since it has full access to its own state.
According to the pattern, we should store the copy of the object’s state in a special object called **Memento** and the content of the memento objects are not accessible to any other object except the one that produced it.

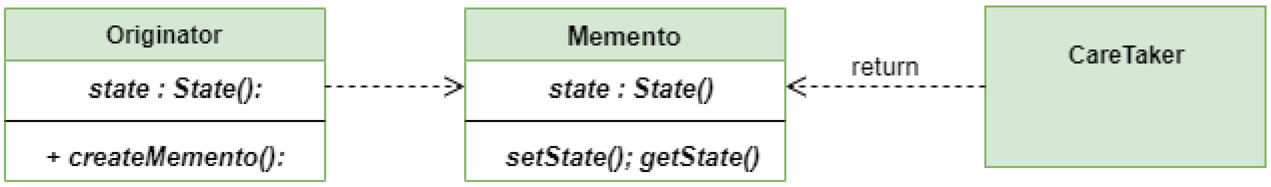
- Python3

```
"""Memento class for saving the data"""



class Memento:



    """Constructor function"""

    def __init__(self, file, content):



        """put all your file content here"""



        self.file = file

        self.content = content
```

```
"""It's a File Writing Utility"""

class FileWriterUtility:

    """Constructor Function"""

    def __init__(self, file):

        """store the input file data"""
        self.file = file
        self.content = ""

    """Write the data into the file"""

    def write(self, string):
        self.content += string

    """save the data into the Memento"""

    def save(self):
        return Memento(self.file, self.content)

    """UNDO feature provided"""

    def undo(self, memento):
        self.file = memento.file
        self.content = memento.content

"""CareTaker for FileWriter"""

class FileWriterCaretaker:

    """saves the data"""

    def save(self, writer):
        self.obj = writer.save()

    """undo the content"""

    def undo(self, writer):
        writer.undo(self.obj)



if __name__ == '__main__':

    """create the caretaker object"""
    caretaker = FileWriterCaretaker()

    """create the writer object"""
    writer = FileWriterUtility("GFG.txt")

    """write data into file using writer object"""
    writer.write("First vision of GeeksforGeeks\n")
    print(writer.content + "\n\n")

    """save the file"""
    caretaker.save(writer)

    """again write using the writer """
    writer.write("Second vision of GeeksforGeeks\n")
    print(writer.content + "\n\n")



    """undo the file"""

    caretaker.undo(writer)



    print(writer.content + "\n\n")
```

UML Diagram

Following is the UML diagram for Memento's Method



Memento-method-UML-Diagram

Advantages

- **Encourages Encapsulation:** Memento method can help in producing the state of the object without breaking the encapsulation of the client's code.
- **Simplifies Code:** We can take the advantage of caretaker who can help us in simplifying the code by maintaining the history of the originator's code.
- **Generic Memento's Implementation:** It's better to use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.

Disadvantages

- **Huge Memory Consumption:** If the Originator's object is very huge then Memento object size will also be huge and use a lot of memory which is definitely not the efficient way to do the work.
- **Problem with Dynamic Languages:** Programming languages like **Ruby**, **Python**, and **PHP** are dynamically typed langauges, can't give the guarantee that the memento object will not be touched.
- **Difficult Deletion:** It's not easy to delete the memento object because the caretaker has to track the originator's lifecycle inorder to get th result.

Applicability

- **UNDO and REDO:**Most of the software applications like Paint, Coding IDEs, text editor, and many others provide **UNDO** and **REDO** features for the ease of client.
- **Providing Encapsulation:** We can use the **Memento's method** for avoiding the breakage of encapsulation in the client's code which might be produced by direct access to the object's internal implementation.

Observer method – Python Design Patterns

The observer method is a **Behavioral design Pattern** which allows you to define or create a subscription mechanism to send the notification to the multiple objects about any new event that happens to the object that they are observing. The subject is basically observed by multiple objects. The subject needs to be monitored and whenever there is a change in the subject, the observers are being notified about the change. This pattern defines one to Many dependencies between objects so that one object changes state, all of its dependents are notified and updated automatically.

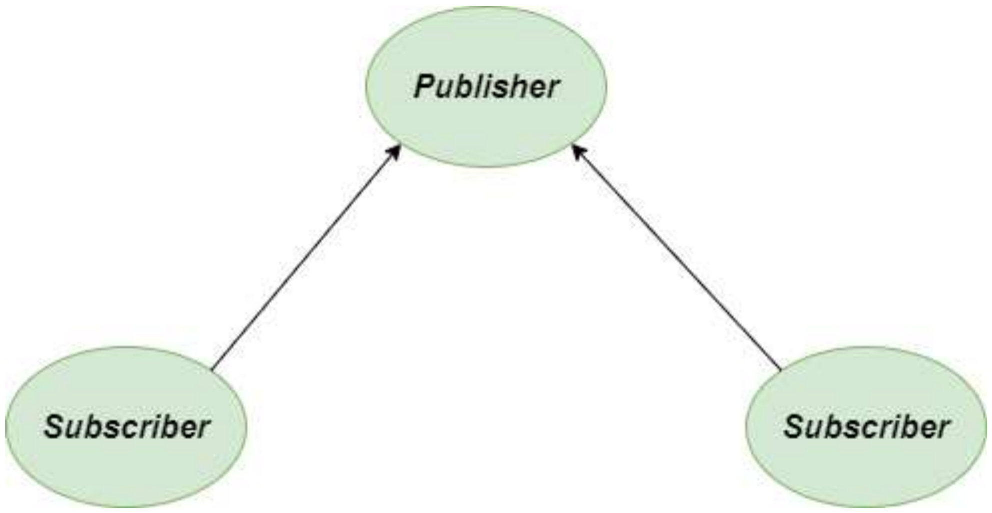
Problem Without using Observer Method

Imagine you want to create a calculator application that has different features such as addition, subtraction, changing base of the numbers to hexadecimal,

decimal, and many other features. But one of your friends is interested in changing the base of his favorite number to Octal base number and you are still developing the application. So, what could be the solution to it? Should your friend check the application daily just to get to know about the status? But don't you think it would result in a lot of unnecessary visits to the application which were definitely not required. Or you may think about that each time you add the new feature and send the notification to each user. Is it OK? Sometimes yes but not every time. Might be some users get offended by a lot of unnecessary notifications which they really don't want.

Solution using Observer Method

Let's discuss the solution to the above-described problem. Here comes the object **Subject** into the limelight. But it also notifies the other objects also that's why we generally call it **Publisher**. All the objects that want to track changes in the publisher's state are called subscribers.



Observer-method-solution-diagram

- Python3

```
class Subject:
```

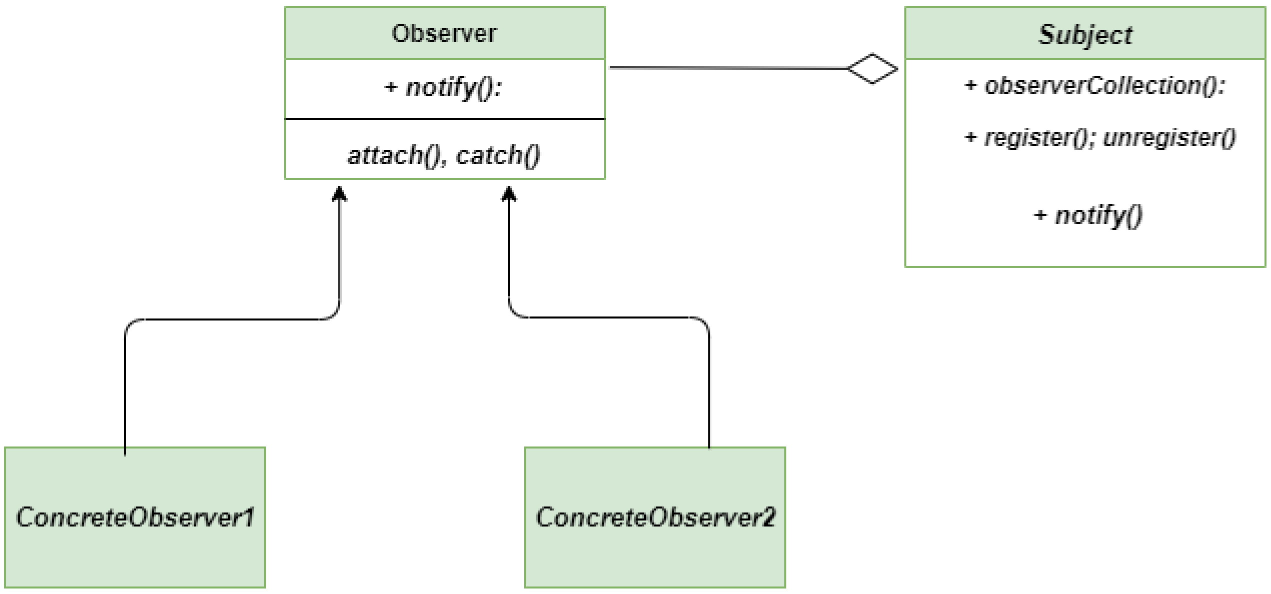
```
"""Represents what is being observed"""

def __init__(self):

    """create an empty observer list"""

    self._observers = []

def notify(self, modifier = None):

    """Alert the observers"""

    for observer in self._observers:
        if modifier != observer:
            observer.update(self)

def attach(self, observer):
        """If the observer is not in the list,

        append it into the list"""


        if observer not in self._observers:

            self._observers.append(observer)


    def detach(self, observer):


        """Remove the observer from the observer list"""


        try:

            self._observers.remove(observer)

        except ValueError:

            pass





class Data(Subject):


    """monitor the object"""
    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:



    """updates the Hewviewer"""
    def update(self, subject):

        print('HexViewer: Subject {} has data
0x{:x}'.format(subject.name, subject.data))



class OctalViewer:



    """updates the Octal viewer"""



    def update(self, subject):

        print('OctalViewer: Subject' + str(subject.name) + 'has data
'+str(oct(subject.data)))




class DecimalViewer:



    """updates the Decimal viewer"""



    def update(self, subject):

        print('DecimalViewer: Subject % s has data % d' % (subject.name,
subject.data))
```

```
"""main function"""

if __name__ == "__main__":

    """provide the data"""

    obj1 = Data('Data 1')
    obj2 = Data('Data 2')

    view1 = DecimalViewer()
    view2 = HexViewer()
    view3 = OctalViewer()

    obj1.attach(view1)
    obj1.attach(view2)
    obj1.attach(view3)

    obj2.attach(view1)
    obj2.attach(view2)
    obj2.attach(view3)
```

```
obj1.data = 10

obj2.data = 15
```

Class Diagram

Following is the class diagram for the Observer Method



Class-diagram-Observer-method

Output

```
DecimalViewer: Subject Data 1 has data 10
HexViewer: Subject Data 1 has data 0xa
OctalViewer: SubjectData 1has data 0o12
DecimalViewer: Subject Data 2 has data 15
HexViewer: Subject Data 2 has data 0xf
```

```
OctalViewer: SubjectData 2has data 0o17
```

Advantages

- **Open/Closed Principle:** Introducing subscriber classes is much easier in Observer method as compared to others without making changes in the client's code.
- **Establishes Relationships:** Its really easy to establishes the relationships at the runtime between the objects.
- **Description:** It carefully describes about the coupling present between the objects and the observer. Hence, there is no need to modify Subject to add or remove observers.

Disadvantages

- **Memory Leakage:** Memory leaks caused by **Lapsed Listener Problem** because of explicit register and unregistering of observers.
- **Random Notifications:** All the subscribers present gets notification in the random order.
- **Risky Implementations:** If the pattern is not implemented carefully, there are huge chances that you will end up with large complexity code.

Applicability

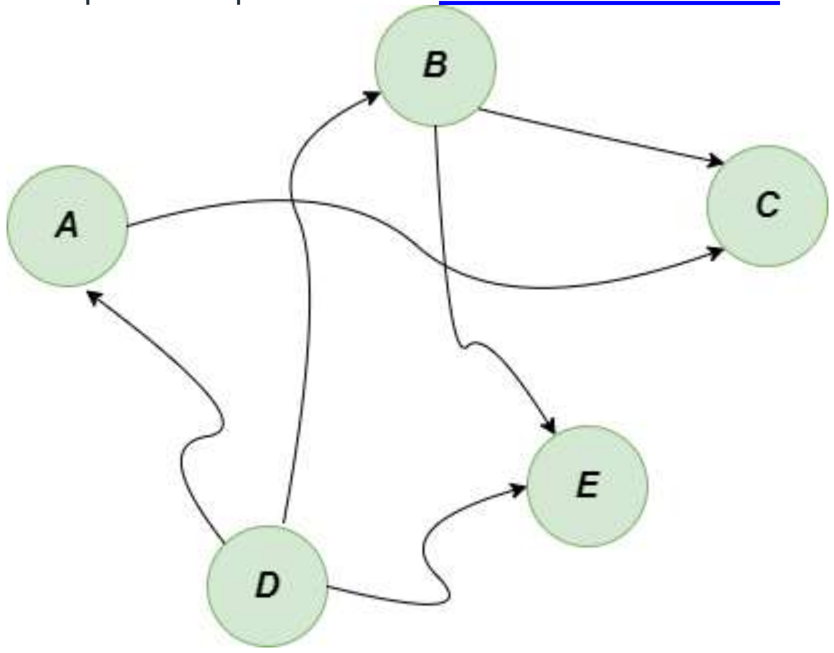
- **Multi-Dependency:** We should use this pattern when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.
- **Getting Notifications:** It is used in social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
- **Reflections of Object:** When we do not coupled the objects tightly, then the change of a state in one object must be reflected in another object.

State Method – Python Design Patterns

State method is **Behavioral Design Pattern** that allows an object to change its behavior when there occurs a change in its internal state. It helps in implementing the state as a derived class of the state pattern interface. If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. It may be termed as the object-oriented state machine. It implements the state transitions by invoking methods from the pattern's superclass.

Problem without using State Method

The State method pattern represents the **Finite State Machine**.

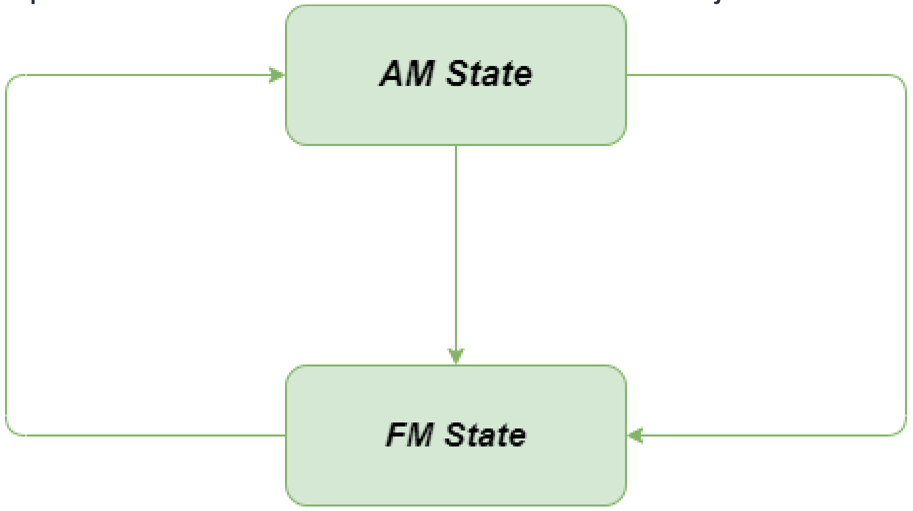


Finite-state-machine-diagram

At any moment, there can be a finite no. of states that can be present in the program. Each and every state is unique in their kind of behavior and other things. Even the program can change itself from one state to another at any moment of time. A program can go from one state to another if and only if the required transition is available in rules. It will certainly become difficult when we add a large number of states. It will become difficult to handle the code as any small change in transition logic may lead to a change in state conditionals in each method. =

Solution using State Method

Let’s have a look at the solution to the above-described problem by considering the example of **Radio**. Here the radio has two states named as **Am State** and **FM State**. We can use the switch to toggle between these two states. The State method suggests that we should create a new class for all possible states of an object and extract all state-specific behaviors into these classes. Instead of implementing all behaviors on its own, the original object called context, stores a reference to one of the state objects that represents its current state, and represents all the state-related work to that object.



Problem-without-State-Method

- Python3

```
"""State class: Base State class"""

class State:



    """Base state. This is to share functionality"""



    def scan(self):
        """Scan the dial to the next station"""

        self.pos += 1



        """check for the last station"""

        if self.pos == len(self.stations):

            self.pos = 0

        print("Visiting... Station is {}
{}".format(self.stations[self.pos], self.name))



"""Separate Class for AM state of the radio"""

class AmState(State):



    """constructor for AM state class"""

    def __init__(self, radio):



        self.radio = radio

        self.stations = ["1250", "1380", "1510"]

        self.pos = 0

        self.name = "AM"
    """method for toggling the state"""
    def toggle_amfm(self):
        print("Switching to FM")
        self.radio.state = self.radio.fmstate


"""Separate class for FM state"""
class FmState(State):


    """Constriuctor for FM state"""
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["81.3", "89.1", "103.9"]
        self.pos = 0
        self.name = "FM"


    """method for toggling the state"""
    def toggle_amfm(self):
        print("Switching to AM")
        self.radio.state = self.radio.amstate
```

```
"""Dedicated class Radio"""

class Radio:


    """A radio. It has a scan button, and an AM / FM toggle switch."""


    def __init__(self):


        """We have an AM state and an FM state"""

        self.fmstate = FmState(self)

        self.amstate = AmState(self)

        self.state = self.fmstate


    """method to toggle the switch"""

    def toggle_amfm(self):

        self.state.toggle_amfm()


    """method to scan """

    def scan(self):

        self.state.scan()
```

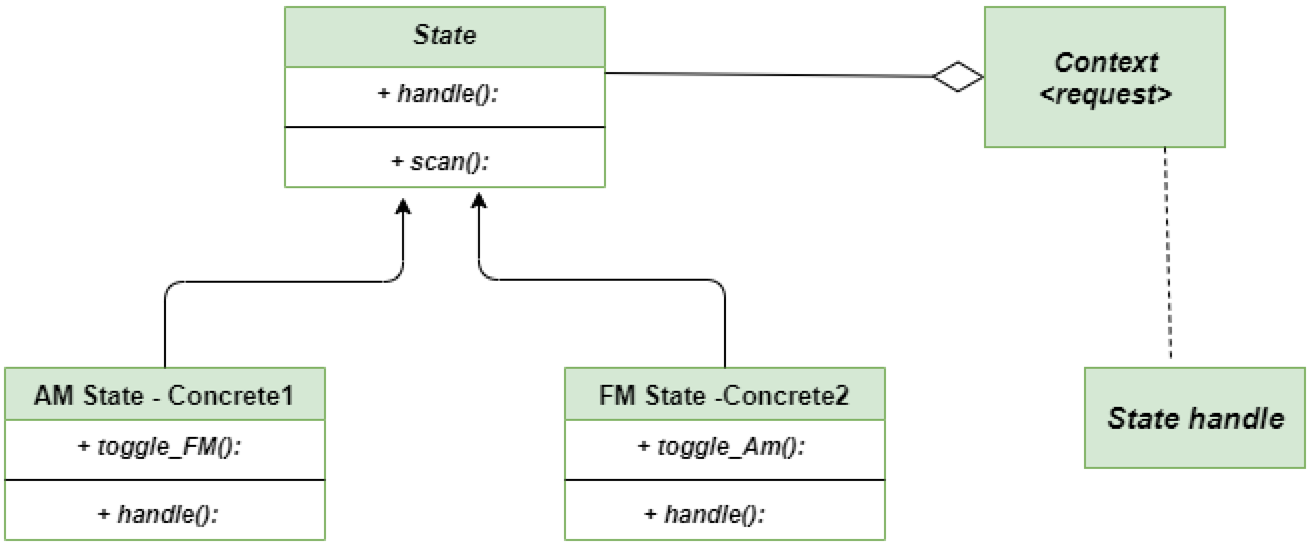
```
""" main method """
if __name__ == "__main__":

    """ create radio object"""
    radio = Radio()
    actions = [radio.scan] * 3 + [radio.toggle_amfm] + [radio.scan] * 3
    actions *= 2

    for action in actions:
        action()
```

Output

```
Visiting... Station is 89.1 FM
Visiting... Station is 103.9 FM
Visiting... Station is 81.3 FM
Switching to AM
Visiting... Station is 1380 AM
Visiting... Station is 1510 AM
Visiting... Station is 1250 AM
Visiting... Station is 1380 AM
Visiting... Station is 1510 AM
Visiting... Station is 1250 AM
Switching to FM
Visiting... Station is 89.1 FM
Visiting... Station is 103.9 FM
Visiting... Station is 81.3 FM
```

UML Diagram

Following is the UML diagram for the state method



UML-diagram-state-method

Advantages

- **Open/Closed Principle:** We can easily introduce the new states without changing the content of existing states of client's code.
- **Single Responsibility Principle:** It helps in organizing the code of particular states into the separate classes which helps in making the code feasible for the other developers too.
- **Improves Cohesion:** It also improves the Cohesion since state-specific behaviors are aggregated into the ConcreteState classes, which are placed in one location in the code.

Disadvantages

- **Making System complex:** If a system has only a few number of states, then its not a good choice to use the **State Method** as you will end up with adding unnecessary code.
- **Changing states at run-time:** State method is used when we need to change the state at run-time by inputting at different sub-classes, this will be considered as a disadvantage as well because we have a clear separate

State classes with some logic and from the other hand the number of classes grows up.
- **Sub-Classes Dependencies:** Here the each state derived class is coupled to its sibling, which directly or indirectly introduces the dependencies between sub-classes.

Strategy Method – Python Design Patterns

The strategy method is **Behavioral Design pattern** that allows you to define the complete family of algorithms, encapsulates each one and putting each of them into separate classes and also allows to interchange there objects. It is implemented in Python by dynamically replacing the content of a method defined inside a class with the contents of functions defined outside of the class. It enables selecting the algorithm at run-time. This method is also called as **Policy Method**.

Problem without using Strategy Method

Imagine you created an application for the departmental store. *Looks simple?* Initially, there was one and only one type of discount called the On-Sale-Discount. So. everything was going with ease and there was no difficulty for maintaining such a simple application for a developer but as time passes, the owner of the departmental store demands for including some other types of discount also for the customers. It is very easy to say to make these changes but definitely not very easy to implement these changes in an efficient way.

Solution using Strategy method

Let's see how can we solve the above-described problem in an efficient way. We can create a specific class that will extract all the algorithms into separate classes called **Strategy**. Out actual class should store the reference to one of the strategy class.

- Python3

```
"""A separate class for Item"""

class Item:
```

```
"""Constructor function with price and discount"""

def __init__(self, price, discount_strategy = None):

    """take price and discount strategy"""

    self.price = price
    self.discount_strategy = discount_strategy

"""A separate function for price after discount"""

def price_after_discount(self):

    if self.discount_strategy:
        discount = self.discount_strategy(self)
    else:
        discount = 0
        return self.price - discount

    def __repr__(self):

        statement = "Price: {}, price after discount: {}"
        return statement.format(self.price, self.price_after_discount())

"""function dedicated to On Sale Discount"""
def on_sale_discount(order):

    return order.price * 0.25 + 20

"""function dedicated to 20 % discount"""
def twenty_percent_discount(order):

    return order.price * 0.20

"""main function"""
if __name__ == "__main__":
```

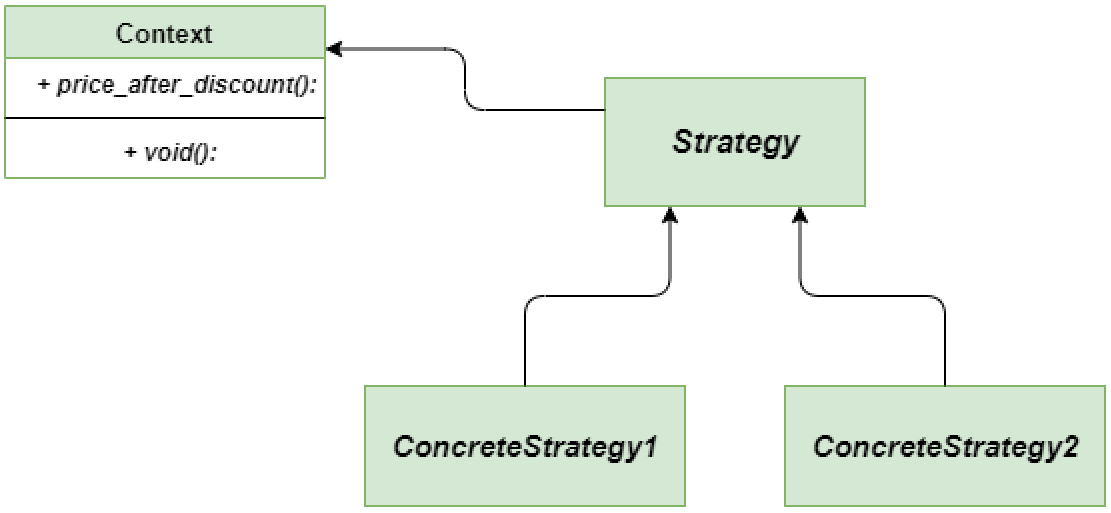
```
print(Item(20000))



"""with discount strategy as 20 % discount"""

print(Item(20000, discount_strategy = twenty_percent_discount))



"""with discount strategy as On Sale Discount"""

print(Item(20000, discount_strategy = on_sale_discount))
```

Output:

```
Price: 20000, price after discount: 20000

Price: 20000, price after discount: 16000.0

Price: 20000, price after discount: 14980.0
```

Class Diagram

Following is the class diagram for the Strategy Method



class-diagram-Strategy-method

Advantages

- **Open/Closed principle:** Its always easy to introduce the new strategies without changing the client’s code.
- **Isolation:** We can isolate the specific implementation details of the algorithms from the client’s code.
- **Encapsulation:** Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class
- **Run-time Switching:** It is possible that application can switch the strategies at the run-time.

Disadvantages

- **Creating Extra Objects:** In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.
- **Awareness among clients:** Difference between the strategies should be clear among the clients to able to select a best one for them.
- **Increases the complexity:** when we have only a few number of algorithms to implement, then its waste of resources to implement the Strategy method.

Applicability

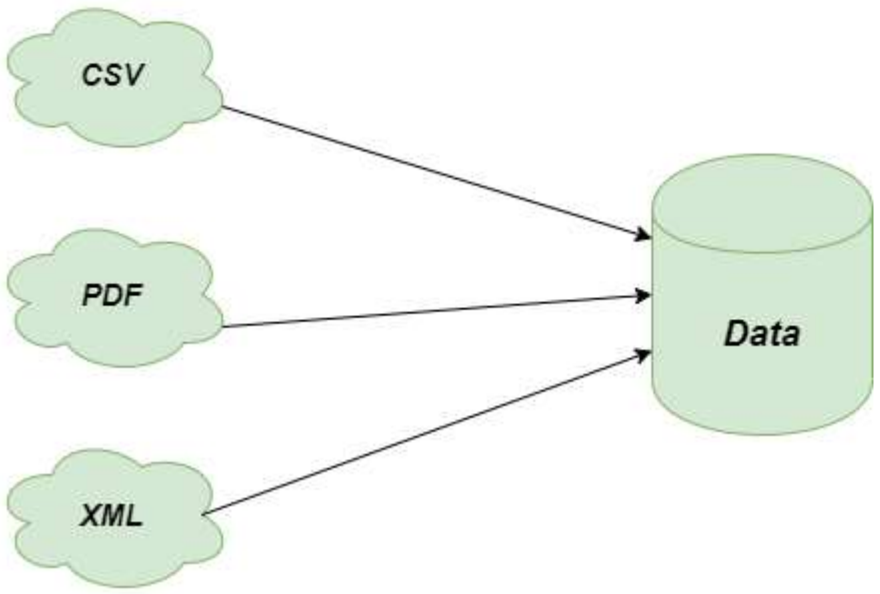
- **Lot of Similar Classes:** This method is highly preferred when we have a lot of similar classes that differs in the way they execute.
- **Conquering Isolation:** It is generally used to isolate the business logic of the class from the algorithmic implementation.

Template Method – Python Design Patterns

The Template method is a **Behavioral Design Pattern** that defines the skeleton of the operation and leaves the details to be implemented by the child class. Its subclasses can override the method implementations as per need but the invocation is to be in the same way as defined by an abstract class. It is one of the easiest among the Behavioral design pattern to understand and implements. Such methods are highly used in framework development as they allow us to reuse the single piece of code at different places by making certain changes. This leads to avoiding code duplication also.

Problem without using Template Method

Imagine you are working on a **Chatbot application** as a software developer which uses data mining techniques to analyze the data of the corporate documents. Initially, your applications were fine with the pdf version of the data only but later your applications also require to collect and convert data from other formats also such as **XML**, **CSV**, and others. After implementing the whole scenario for the other formats also, you noticed that all the classes have lots of similar code. Part of the code like analyzing and processing was identical in almost all classes whereas they differ in dealing with the data.



Problem-Template-Method

Solution using Template Method

Let's discuss the solution to the above-described problem using the template method. It suggests to break down the code into a series of steps and convert these steps into methods and put series call inside the template_function. Hence we created the **template_function** separately and create methods such as **get_xml**, **get_pdf** and **get_csv** for dealing with the code separately.

- Python3

```
""" method to get the text of file"""
def get_text():


    return "plain_text"


""" method to get the xml version of file"""
def get_xml():


    return "xml"


""" method to get the pdf version of file"""
def get_pdf():


    return "pdf"


"""method to get the csv version of file"""
def get_csv():


    return "csv"
```

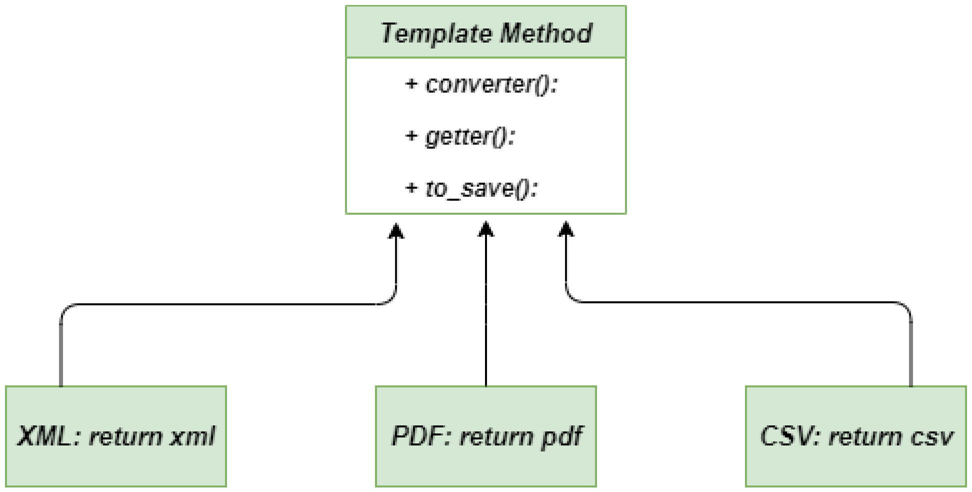
```
"""method used to convert the data into text format"""

def convert_to_text(data):



    print("[CONVERT]")

    return "{} as text".format(data)



"""method used to save the data"""

def saver():



    print("[SAVE]")



"""helper function named as template_function"""

def template_function(getter, converter = False, to_save = False):



    """input data from getter"""

    data = getter()

    print("Got `{}`".format(data))



    if len(data) <= 3 and converter:
        data = converter(data)

    else:

        print("Skip conversion")



    """saves the data only if user want to save it"""

    if to_save:

        saver()



    print("`{}` was processed".format(data))





"""main method"""

if __name__ == "__main__":



    template_function(get_text, to_save = True)



    template_function(get_pdf, converter = convert_to_text)



    template_function(get_csv, to_save = True)
    template_function(get_xml, to_save = True)
```

Output

```
Got `plain_text`
Skip conversion
[SAVE]
`plain_text` was processed
Got `pdf`
[CONVERT]
`pdf as text` was processed
Got `csv`
Skip conversion
[SAVE]
`csv` was processed
```

Class Diagram

Following is the class diagram for the Template Method



Class-diagram-template-method

Advantages

- **Equivalent Content:** It’s easy to consider the duplicate code in the superclass by pulling it there where you want to use it.
- **Flexibility:** It provides vast flexibility such that subclasses are able to decide how to implement the steps of the algorithms.
- **Possibility of Inheritance:** We can reuse our code as the **Template Method** uses inheritance which provides the ability of code reusability.

Disadvantages

- **Complex Code:** The code may become enough complex sometimes while using the template method such that it becomes too much hard to understand the code even by the developers who are writing it.
- **Limitness:** Clients may ask for the extended version because sometimes they feel lack of algorithms in the provided skeleton.
- **Violation:** It might be possible that by using Template method, you may end up with violating the **Liskov Substitution Principle** which is definitely not the good thing to follow.

Applicability

- **Extension by Clients:** This method is always preferred to use when you want to let clients extend the algorithm using particular steps but with not the whole structure of the algorithm.
- **Similar Algorithms:** When you have a lot of similar algorithms with minor changes, its always better to use the template design pattern because if some changes occur in the algorithm, then you don’t have to make changes in each algorithm.
- **Development of Frameworks:** It is highly recommended to use the template design pattern while developing a framework because it will help us to avoid the duplicate code as well as reusing the piece of code again and again by making certain changes.

Visitor Method – Python Design Patterns

Visitor Method is a **Behavioral Design Pattern** which allows us to separate the algorithm from an object structure on which it operates. It helps us to add new features to an existing class hierarchy dynamically without changing it. All the behavioral patterns proved as the best methods to handle the communication between the objects. Similarly, it is used when we have to perform an operation on a group of similar kinds of objects.
A Visitor Method consists of two parts:

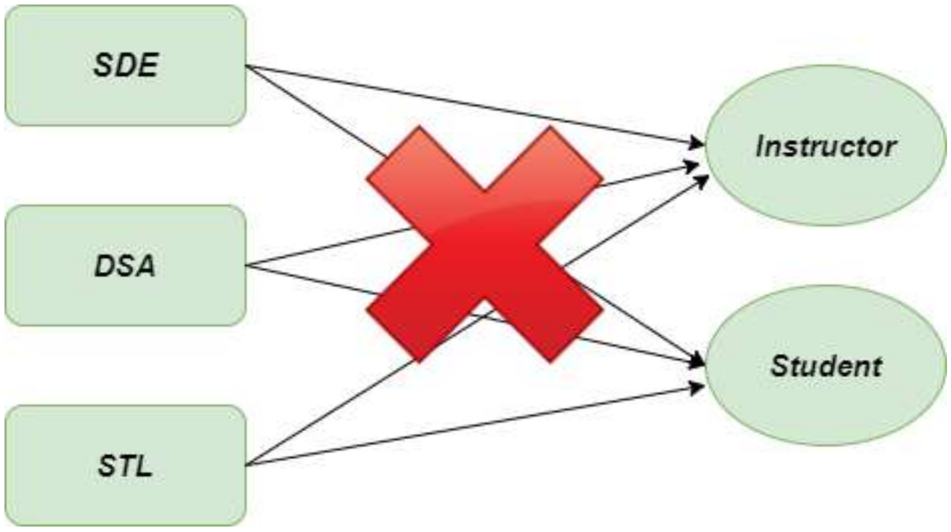
- method named as **Visit()** implemented by the visitor and used and called for every element of the data structure.
- Visitable classes providing **Accept()** methods that accept a visitor

Design Components

- **Client:** The Client class acts as the consumer of the classes of the visitor design pattern. It can access the data structure objects and can instruct them to accept a visitor for the future processing.
- **Visitor:** An Abstract class which is used to declare visit operations for all visitable classes.
- **Concrete Visitor:** Each Visitor will be responsible for different operations. For each type of visitor all the visit methods, declared in abstract visitor, must be implemented.
- **Visitable:** Accept operations is declared by this class. It also act as the entry point which enables an object to be visited by visitor.
- **Concrete Visitable:** These classes implement the Visitable class and defines the accept operation. The visitor object is passed to this object using the accept operation.

Problem without using Visitor Method

Imagine you are handling the Software management of **GeeksforGeeks** and they have started certain courses such as DSA, SDE, and STL which are definitely useful for students who are preparing for the product based companies. But how will you handle all the data of Courses, Instructors, students, classes, IDs in your database? If you go with a simple direct approach to handle such a situation, you will definitely end up with a mess only.



Visitor-problem-diagram

Solution using Visitor Method

Let’s look at the solution to the above-described problem. The Visitor method suggests adding a new behavior in a separate class called Visitor class instead of mixing it with the already existing classes. We will pass the original object to the visitor’s method as parameters such that the method will access all the necessary information.

- Python3

```
""" The Courses hierarchy cannot be changed to add new

   functionality dynamically. Abstract Crop class for

 Concrete Courses_At_GFG classes: methods defined in this class

 will be inherited by all Concrete Courses_At_GFG classes."""



class Courses_At_GFG:
    def accept(self, visitor):

        visitor.visit(self)



    def teaching(self, visitor):

        print(self, "Taught by ", visitor)



    def studying(self, visitor):

        print(self, "studied by ", visitor)




    def __str__(self):

        return self.__class__.__name__




"""Concrete Courses_At_GFG class: Classes being visited."""

class SDE(Courses_At_GFG): pass



class STL(Courses_At_GFG): pass
class DSA(Courses_At_GFG): pass



""" Abstract Visitor class for Concrete Visitor classes:
 method defined in this class will be inherited by all
 Concrete Visitor classes."""
class Visitor:

    def __str__(self):
        return self.__class__.__name__



""" Concrete Visitors: Classes visiting Concrete Course objects.
 These classes have a visit() method which is called by the
 accept() method of the Concrete Course_At_GFG classes."""
class Instructor(Visitor):
    def visit(self, crop):
        crop.teaching(self)
```

```
class Student(Visitor):

    def visit(self, crop):

        crop.studying(self)




"""creating objects for concrete classes"""

sde = SDE()

stl = STL()

dsa = DSA()


"""Creating Visitors"""

instructor = Instructor()

student = Student()


"""Visitors visiting courses"""

sde.accept(instructor)

sde.accept(student)


stl.accept(instructor)
```

```
 stl.accept(student)



 dsa.accept(instructor)

 dsa.accept(student)
```

Output

```
SDE Taught by  Instructor
SDE studied by  Student
STL Taught by  Instructor
STL studied by  Student
DSA Taught by  Instructor
DSA studied by  Student
```

UML Diagram

Following is the UML diagram for Visitor Method

UML-diagram-visitor-method

Advantages

- **Open/Closed principle:** Introducing new behavior in class is easy which can work with objects of different classes without making changes in these classes.
- **Single Responsibility Principle:** Multiple versions of same behavior can be operated into the same class.
- **Addition of entities:** Adding an entity in **Visitor Method** is easy as we have to make changes in visitor class only and it will not affect the existing item.
- **Updating Logic:** If the logic of operation is updated, then we need to make change only in the visitor implementation rather than doing it in all the item classes.

Disadvantages

- **Lots of Updates:** We have to update each and every visitor whenever a class get added or removed form the primary hierarchy
- **Hard to Extend:** If there are too many visitor classes then it becomes really hard to extend the whole interface of the class.
- **Lack of Access:** Sometimes visitors might not have the access to private field of certain classes that they are supposed to work with.

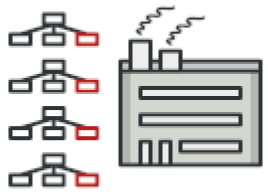
Applicability

- **Recursive structures:** Visitor Method works really well with recursive structures like directory trees or XML structures. The Visitor object can visit each node in the recursive structure
- **Performing Operations:** We cam use the visitor method when we have to perform operations on all the elements of the complex object like Tree.

DESIGN PATTERNSINC++

The Catalog of C++ Examples

Creational Patterns



Abstract Factory

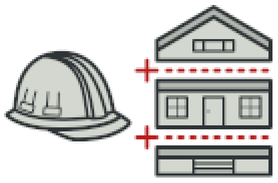
Lets you produce families of related objects without specifying their concrete classes.
Main article
Usage in C++
Code example

```
/**
 * Each distinct product of a product family should have a base interface. All
 * variants of the product must implement this interface.
 */
class AbstractProductA {
 public:
  virtual ~AbstractProductA(){};
  virtual std::string UsefulFunctionA() const = 0;
};

/**
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductA1 : public AbstractProductA {
 public:
  std::string UsefulFunctionA() const override {
    return "The result of the product A1.";
  }
};

class ConcreteProductA2 : public AbstractProductA {
  std::string UsefulFunctionA() const override {
    return "The result of the product A2.";
  }
};

/**
 * Here's the the base interface of another product. All products can interact
 * with each other, but proper interaction is possible only between products of
 * the same concrete variant.
 */
class AbstractProductB {
  /**
   * Product B is able to do its own thing...
   */
 public:
  virtual ~AbstractProductB(){};
  virtual std::string UsefulFunctionB() const = 0;
  /**
   * ...but it also can collaborate with the ProductA.
   *
   * The Abstract Factory makes sure that all products it creates are of the
   * same variant and thus, compatible.
   */
  virtual std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator)
const = 0;
};

/**
 * Concrete Products are created by corresponding Concrete Factories.
 */
class ConcreteProductB1 : public AbstractProductB {
 public:
  std::string UsefulFunctionB() const override {
    return "The result of the product B1.";
  }
  /**
   * The variant, Product B1, is only able to work correctly with the variant,
   * Product A1. Nevertheless, it accepts any instance of AbstractProductA as an
   * argument.
   */
  std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator) const
override {
    const std::string result = collaborator.UsefulFunctionA();
    return "The result of the B1 collaborating with ( " + result + " )";
  }
};

class ConcreteProductB2 : public AbstractProductB {
 public:
  std::string UsefulFunctionB() const override {
    return "The result of the product B2.";
  }
  /**
   * The variant, Product B2, is only able to work correctly with the variant,
   * Product A2. Nevertheless, it accepts any instance of AbstractProductA as an
   * argument.
   */
  std::string AnotherUsefulFunctionB(const AbstractProductA &collaborator) const
override {
    const std::string result = collaborator.UsefulFunctionA();
    return "The result of the B2 collaborating with ( " + result + " )";
  }
};

/**
 * The Abstract Factory interface declares a set of methods that return
 * different abstract products. These products are called a family and are
 * related by a high-level theme or concept. Products of one family are usually
 * able to collaborate among themselves. A family of products may have several
 * variants, but the products of one variant are incompatible with products of
 * another.
 */
class AbstractFactory {
 public:
  virtual AbstractProductA *CreateProductA() const = 0;
  virtual AbstractProductB *CreateProductB() const = 0;
};

/**
 * Concrete Factories produce a family of products that belong to a single
 * variant. The factory guarantees that resulting products are compatible. Note
 * that signatures of the Concrete Factory's methods return an abstract product,
 * while inside the method a concrete product is instantiated.
 */
class ConcreteFactory1 : public AbstractFactory {
 public:
  AbstractProductA *CreateProductA() const override {
    return new ConcreteProductA1();
  }
  AbstractProductB *CreateProductB() const override {
    return new ConcreteProductB1();
  }
```

```cpp
};

/**
 * Each Concrete Factory has a corresponding product variant.
 */
class ConcreteFactory2 : public AbstractFactory {
 public:
  AbstractProductA *CreateProductA() const override {
    return new ConcreteProductA2();
  }
  AbstractProductB *CreateProductB() const override {
    return new ConcreteProductB2();
  }
};

/**
 * The client code works with factories and products only through abstract
 * types: AbstractFactory and AbstractProduct. This lets you pass any factory or
 * product subclass to the client code without breaking it.
 */

void ClientCode(const AbstractFactory &factory) {
  const AbstractProductA *product_a = factory.CreateProductA();
  const AbstractProductB *product_b = factory.CreateProductB();
  std::cout << product_b->UsefulFunctionB() << "\n";
  std::cout << product_b->AnotherUsefulFunctionB(*product_a) << "\n";
  delete product_a;
  delete product_b;
}

int main() {
  std::cout << "Client: Testing client code with the first factory type:\n";
  ConcreteFactory1 *f1 = new ConcreteFactory1();
  ClientCode(*f1);
  delete f1;
  std::cout << std::endl;
  std::cout << "Client: Testing the same client code with the second factory
type:\n";
  ConcreteFactory2 *f2 = new ConcreteFactory2();
  ClientCode(*f2);
  delete f2;
  return 0;
}
```

Output.txt: **Execution result**

```
Client: Testing client code with the first factory type:
The result of the product B1.
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:
The result of the product B2.
The result of the B2 collaborating with the (The result of the product A2.)
```



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
Main article
Usage in C++
Code example

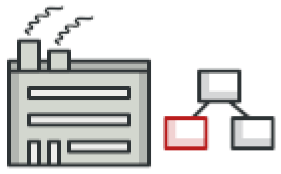
```
/**
 * It makes sense to use the Builder pattern only when your products are quite
 * complex and require extensive configuration.
 *
 * Unlike in other creational patterns, different concrete builders can produce
 * unrelated products. In other words, results of various builders may not
 * always follow the same interface.
 */

class Product1{
    public:
    std::vector<std::string> parts_;
    void ListParts()const{
        std::cout << "Product parts: ";
        for (size_t i=0;i<parts_.size();i++){
            if(parts_[i]== parts_.back()){
                std::cout << parts_[i];
            }else{
                std::cout << parts_[i] << ", ";
            }
        }
        std::cout << "\n\n";
    }
};


/**
 * The Builder interface specifies methods for creating the different parts of
 * the Product objects.
 */
class Builder{
    public:
    virtual ~Builder(){}
    virtual void ProducePartA() const =0;
    virtual void ProducePartB() const =0;
    virtual void ProducePartC() const =0;
};
/**
 * The Concrete Builder classes follow the Builder interface and provide
 * specific implementations of the building steps. Your program may have several
 * variations of Builders, implemented differently.
 */
class ConcreteBuilder1 : public Builder{
    private:

    Product1* product;

    /**
     * A fresh builder instance should contain a blank product object, which is
     * used in further assembly.
     */
    public:

    ConcreteBuilder1(){
        this->Reset();
    }

    ~ConcreteBuilder1(){
        delete product;
    }

    void Reset(){
        this->product= new Product1();
    }
    /**
     * All production steps work with the same product instance.
     */

    void ProducePartA()const override{
        this->product->parts_.push_back("PartA1");
    }

    void ProducePartB()const override{
        this->product->parts_.push_back("PartB1");
    }

    void ProducePartC()const override{
        this->product->parts_.push_back("PartC1");
    }

    /**
     * Concrete Builders are supposed to provide their own methods for
     * retrieving results. That's because various types of builders may create
     * entirely different products that don't follow the same interface.
     * Therefore, such methods cannot be declared in the base Builder interface
     * (at least in a statically typed programming language). Note that PHP is a
     * dynamically typed language and this method CAN be in the base interface.
     * However, we won't declare it there for the sake of clarity.
     *
     * Usually, after returning the end result to the client, a builder instance
     * is expected to be ready to start producing another product. That's why
     * it's a usual practice to call the reset method at the end of the
     * `getProduct` method body. However, this behavior is not mandatory, and
     * you can make your builders wait for an explicit reset call from the
     * client code before disposing of the previous result.
     */

    /**
     * Please be careful here with the memory ownership. Once you call
     * GetProduct the user of this function is responsable to release this
     * memory. Here could be a better option to use smart pointers to avoid
     * memory leaks
     */

    Product1* GetProduct() {
        Product1* result= this->product;
        this->Reset();
        return result;
    }
};

/**
 * The Director is only responsible for executing the building steps in a
 * particular sequence. It is helpful when producing products according to a
 * specific order or configuration. Strictly speaking, the Director class is
 * optional, since the client can control builders directly.
 */
class Director{
    /**
     * @var Builder
     */
    private:
    Builder* builder;
    /**
     * The Director works with any builder instance that the client code passes
     * to it. This way, the client code may alter the final type of the newly
     * assembled product.
     */

    public:

    void set_builder(Builder* builder){
        this->builder=builder;
    }

    /**
     * The Director can construct several product variations using the same
     * building steps.
     */

    void BuildMinimalViableProduct(){
        this->builder->ProducePartA();
    }

    void BuildFullFeaturedProduct(){
        this->builder->ProducePartA();
        this->builder->ProducePartB();
        this->builder->ProducePartC();
    }
```

```
};
/**
 * The client code creates a builder object, passes it to the director and then
 * initiates the construction process. The end result is retrieved from the
 * builder object.
 */
/**
 * I used raw pointers for simplicity however you may prefer to use smart
 * pointers here
 */
void ClientCode(Director& director)
{
    ConcreteBuilder1* builder = new ConcreteBuilder1();
    director.set_builder(builder);
    std::cout << "Standard basic product:\n";
    director.BuildMinimalViableProduct();

    Product1* p= builder->GetProduct();
    p->ListParts();
    delete p;

    std::cout << "Standard full featured product:\n";
    director.BuildFullFeaturedProduct();

    p= builder->GetProduct();
    p->ListParts();
    delete p;

    // Remember, the Builder pattern can be used without a Director class.
    std::cout << "Custom product:\n";
    builder->ProducePartA();
    builder->ProducePartC();
    p=builder->GetProduct();
    p->ListParts();
    delete p;

    delete builder;
}

int main(){
    Director* director= new Director();
    ClientCode(*director);
    delete director;
    return 0;
}
```

Output.txt: Execution result

```
Standard basic product:
Product parts: PartA1

Standard full featured product:
Product parts: PartA1, PartB1, PartC1
```

```
Custom product:
Product parts: PartA1, PartC1
```



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
 Main article
 Usage in C++
 Code example

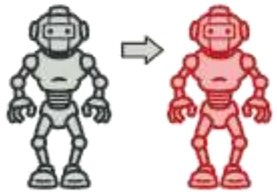
```
/**
 * The Product interface declares the operations that all concrete products must
 * implement.
 */

class Product {
 public:
  virtual ~Product() {}
  virtual std::string Operation() const = 0;
};

/**
 * Concrete Products provide various implementations of the Product interface.
 */
class ConcreteProduct1 : public Product {
 public:
  std::string Operation() const override {
    return "{Result of the ConcreteProduct1}";
  }
};
class ConcreteProduct2 : public Product {
 public:
  std::string Operation() const override {
    return "{Result of the ConcreteProduct2}";
  }
};

/**
 * The Creator class declares the factory method that is supposed to return an
 * object of a Product class. The Creator's subclasses usually provide the
 * implementation of this method.
 */
```

```
class Creator {
  /**
   * Note that the Creator may also provide some default implementation of the
   * factory method.
   */
 public:
  virtual ~Creator(){};
  virtual Product* FactoryMethod() const = 0;
  /**
   * Also note that, despite its name, the Creator's primary responsibility is
   * not creating products. Usually, it contains some core business logic that
   * relies on Product objects, returned by the factory method. Subclasses can
   * indirectly change that business logic by overriding the factory method and
   * returning a different type of product from it.
   */

  std::string SomeOperation() const {
    // Call the factory method to create a Product object.
    Product* product = this->FactoryMethod();
    // Now, use the product.
    std::string result = "Creator: The same creator's code has just worked with " +
product->Operation();
    delete product;
    return result;
  }
};

/**
 * Concrete Creators override the factory method in order to change the
 * resulting product's type.
 */
class ConcreteCreator1 : public Creator {
  /**
   * Note that the signature of the method still uses the abstract product type,
   * even though the concrete product is actually returned from the method. This
   * way the Creator can stay independent of concrete product classes.
   */
 public:
  Product* FactoryMethod() const override {
    return new ConcreteProduct1();
  }
};

class ConcreteCreator2 : public Creator {
 public:
  Product* FactoryMethod() const override {
    return new ConcreteProduct2();
  }
};

/**
 * The client code works with an instance of a concrete creator, albeit through
 * its base interface. As long as the client keeps working with the creator via
 * the base interface, you can pass it any creator's subclass.
 */
```

```
void ClientCode(const Creator& creator) {
  // ...
  std::cout << "Client: I'm not aware of the creator's class, but it still works.\n"
            << creator.SomeOperation() << std::endl;
  // ...
}

/**
 * The Application picks a creator's type depending on the configuration or
 * environment.
 */

int main() {
  std::cout << "App: Launched with the ConcreteCreator1.\n";
  Creator* creator = new ConcreteCreator1();
  ClientCode(*creator);
  std::cout << std::endl;
  std::cout << "App: Launched with the ConcreteCreator2.\n";
  Creator* creator2 = new ConcreteCreator2();
  ClientCode(*creator2);

  delete creator;
  delete creator2;
  return 0;
}
```

Output.txt: Execution result

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the
ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the
ConcreteProduct2}
```



Prototype

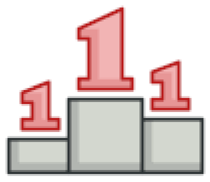
Lets you copy existing objects without making your code dependent on their classes.
Main article
Usage in C++
Code example

```
using std::string;
```

```
// Prototype Design Pattern
//
// Intent: Lets you copy existing objects without making your code dependent on
// their classes.

enum Type {
  PROTOTYPE_1 = 0,
  PROTOTYPE_2
};

/**
 * The example class that has cloning ability. We'll see how the values of field
 * with different types will be cloned.
 */

class Prototype {
 protected:
  string prototype_name_;
  float prototype_field_;

 public:
  Prototype() {}
  Prototype(string prototype_name)
      : prototype_name_(prototype_name) {
  }
  virtual ~Prototype() {}
  virtual Prototype *Clone() const = 0;
  virtual void Method(float prototype_field) {
    this->prototype_field_ = prototype_field;
    std::cout << "Call Method from " << prototype_name_ << " with field : " <<
prototype_field << std::endl;
  }
};

/**
 * ConcretePrototype1 is a Sub-Class of Prototype and implement the Clone Method
 * In this example all data members of Prototype Class are in the Stack. If you
 * have pointers in your properties for ex: String* name_ ,you will need to
 * implement the Copy-Constructor to make sure you have a deep copy from the
 * clone method
 */

class ConcretePrototype1 : public Prototype {
 private:
  float concrete_prototype_field1_;

 public:
  ConcretePrototype1(string prototype_name, float concrete_prototype_field)
      : Prototype(prototype_name),
concrete_prototype_field1_(concrete_prototype_field) {
  }

  /**
   * Notice that Clone method return a Pointer to a new ConcretePrototype1
   * replica. so, the client (who call the clone method) has the responsability
   * to free that memory. I you have smart pointer knowledge you may prefer to
   * use unique_pointer here.
   */
  Prototype *Clone() const override {
    return new ConcretePrototype1(*this);
  }
};

class ConcretePrototype2 : public Prototype {
 private:
  float concrete_prototype_field2_;

 public:
  ConcretePrototype2(string prototype_name, float concrete_prototype_field)
      : Prototype(prototype_name),
concrete_prototype_field2_(concrete_prototype_field) {
  }
  Prototype *Clone() const override {
    return new ConcretePrototype2(*this);
  }
};

/**
 * In PrototypeFactory you have two concrete prototypes, one for each concrete
 * prototype class, so each time you want to create a bullet , you can use the
 * existing ones and clone those.
 */

class PrototypeFactory {
 private:
  std::unordered_map<Type, Prototype *, std::hash<int>> prototypes_;

 public:
  PrototypeFactory() {
    prototypes_[Type::PROTOTYPE_1] = new ConcretePrototype1("PROTOTYPE_1 ", 50.f);
    prototypes_[Type::PROTOTYPE_2] = new ConcretePrototype2("PROTOTYPE_2 ", 60.f);
  }

  /**
   * Be carefull of free all memory allocated. Again, if you have smart pointers
   * knowelege will be better to use it here.
   */

  ~PrototypeFactory() {
    delete prototypes_[Type::PROTOTYPE_1];
    delete prototypes_[Type::PROTOTYPE_2];
  }

  /**
   * Notice here that you just need to specify the type of the prototype you
   * want and the method will create from the object with this type.
   */
  Prototype *CreatePrototype(Type type) {
    return prototypes_[type]->Clone();
  }
};

void Client(PrototypeFactory &prototype_factory) {
  std::cout << "Let's create a Prototype 1\n";

  Prototype *prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_1);
  prototype->Method(90);
  delete prototype;

  std::cout << "\n";

  std::cout << "Let's create a Prototype 2 \n";

  prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_2);
  prototype->Method(10);

  delete prototype;
}

int main() {
  PrototypeFactory *prototype_factory = new PrototypeFactory();
  Client(*prototype_factory);
  delete prototype_factory;

  return 0;
}
```

Output.txt: Execution result

```
Let's create a Prototype 1
Call Method from PROTOTYPE_1  with field : 90

Let's create a Prototype 2
Call Method from PROTOTYPE_2  with field : 10
```



Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

[Main article](#)
[Usage in C++](#)
[Naïve Singleton](#)
[Thread-safe Singleton](#)

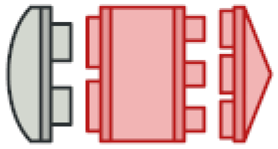
```
/**
```

```
 * The Singleton class defines the `GetInstance` method that serves as an
 * alternative to constructor and lets clients access the same instance of this
 * class over and over.
 */
class Singleton
{

    /**
     * The Singleton's constructor should always be private to prevent direct
     * construction calls with the `new` operator.
     */

protected:
    Singleton(const std::string value): value_(value)
    {
    }

    static Singleton* singleton_;

    std::string value_;

public:

    /**
     * Singletons should not be cloneable.
     */
    Singleton(Singleton &other) = delete;
    /**
     * Singletons should not be assignable.
     */
    void operator=(const Singleton &) = delete;
    /**
     * This is the static method that controls the access to the singleton
     * instance. On the first run, it creates a singleton object and places it
     * into the static field. On subsequent runs, it returns the client existing
     * object stored in the static field.
     */

    static Singleton *GetInstance(const std::string& value);
    /**
     * Finally, any singleton should define some business logic, which can be
     * executed on its instance.
     */
    void SomeBusinessLogic()
    {
        // ...
    }

    std::string value() const{
        return value_;
    }
};

Singleton* Singleton::singleton_= nullptr;;
```

```cpp
/**
 * Static methods should be defined outside the class.
 */
Singleton *Singleton::GetInstance(const std::string& value)
{
    /**
     * This is a safer way to create an instance. instance = new Singleton is
     * dangeruous in case two instance threads wants to access at the same time
     */
    if(singleton_==nullptr){
        singleton_ = new Singleton(value);
    }
    return singleton_;
}

void ThreadFoo(){
    // Following code emulates slow initialization.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("FOO");
    std::cout << singleton->value() << "\n";
}

void ThreadBar(){
    // Following code emulates slow initialization.
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    Singleton* singleton = Singleton::GetInstance("BAR");
    std::cout << singleton->value() << "\n";
}


int main()
{
    std::cout <<"If you see the same value, then singleton was reused (yay!\n" <<
                "If you see different values, then 2 singletons were created
(booo!!)\n\n" <<
                "RESULT:\n";
    std::thread t1(ThreadFoo);
    std::thread t2(ThreadBar);
    t1.join();
    t2.join();

    return 0;
}
```

Output.txt: Execution result

```
If you see the same value, then singleton was reused (yay!
If you see different values, then 2 singletons were created (booo!!)

RESULT:
BAR
FOO
```



Adapter

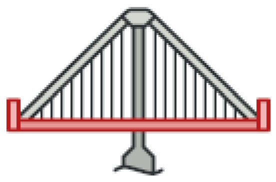
Allows objects with incompatible interfaces to collaborate.
Main article
Usage in C++
Conceptual example
Multiple inheritance

```
/**
 * The Target defines the domain-specific interface used by the client code.
 */
class Target {
 public:
  virtual ~Target() = default;

  virtual std::string Request() const {
    return "Target: The default target's behavior.";
  }
};

/**
 * The Adaptee contains some useful behavior, but its interface is incompatible
 * with the existing client code. The Adaptee needs some adaptation before the
 * client code can use it.
 */
class Adaptee {
 public:
  std::string SpecificRequest() const {
    return ".eetpadA eht fo roivaheb laicepS";
  }
};

/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter : public Target {
 private:
  Adaptee *adaptee_;

 public:
  Adapter(Adaptee *adaptee) : adaptee_(adaptee) {}
  std::string Request() const override {
    std::string to_reverse = this->adaptee_->SpecificRequest();
```

```cpp
    std::reverse(to_reverse.begin(), to_reverse.end());
    return "Adapter: (TRANSLATED) " + to_reverse;
  }
};

/**
 * The client code supports all classes that follow the Target interface.
 */
void ClientCode(const Target *target) {
  std::cout << target->Request();
}

int main() {
  std::cout << "Client: I can work just fine with the Target objects:\n";
  Target *target = new Target;
  ClientCode(target);
  std::cout << "\n\n";
  Adaptee *adaptee = new Adaptee;
  std::cout << "Client: The Adaptee class has a weird interface. See, I don't
understand it:\n";
  std::cout << "Adaptee: " << adaptee->SpecificRequest();
  std::cout << "\n\n";
  std::cout << "Client: But I can work with it via the Adapter:\n";
  Adapter *adapter = new Adapter(adaptee);
  ClientCode(adapter);
  std::cout << "\n";

  delete target;
  delete adaptee;
  delete adapter;

  return 0;
}
```

Output.txt: Execution result

```
Client: I can work just fine with the Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird interface. See, I don't understand it:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
```



Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Main article

Usage in C++

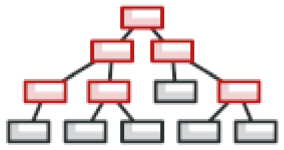
Code example

```
/**
 * The Implementation defines the interface for all implementation classes. It
 * doesn't have to match the Abstraction's interface. In fact, the two
 * interfaces can be entirely different. Typically the Implementation interface
 * provides only primitive operations, while the Abstraction defines higher-
 * level operations based on those primitives.
 */

class Implementation {
 public:
  virtual ~Implementation() {}
  virtual std::string OperationImplementation() const = 0;
};

/**
 * Each Concrete Implementation corresponds to a specific platform and
 * implements the Implementation interface using that platform's API.
 */
class ConcreteImplementationA : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationA: Here's the result on the platform A.\n";
  }
};
class ConcreteImplementationB : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationB: Here's the result on the platform B.\n";
  }
};

/**
 * The Abstraction defines the interface for the "control" part of the two class
 * hierarchies. It maintains a reference to an object of the Implementation
 * hierarchy and delegates all of the real work to this object.
 */

class Abstraction {
  /**
   * @var Implementation
   */
 protected:
  Implementation* implementation_;

 public:
  Abstraction(Implementation* implementation) : implementation_(implementation) {
  }
```

```cpp
  virtual ~Abstraction() {
  }

  virtual std::string Operation() const {
    return "Abstraction: Base operation with:\n" +
           this->implementation_->OperationImplementation();
  }
};
/**
 * You can extend the Abstraction without changing the Implementation classes.
 */
class ExtendedAbstraction : public Abstraction {
 public:
  ExtendedAbstraction(Implementation* implementation) : Abstraction(implementation) {
  }
  std::string Operation() const override {
    return "ExtendedAbstraction: Extended operation with:\n" +
           this->implementation_->OperationImplementation();
  }
};

/**
 * Except for the initialization phase, where an Abstraction object gets linked
 * with a specific Implementation object, the client code should only depend on
 * the Abstraction class. This way the client code can support any abstraction-
 * implementation combination.
 */
void ClientCode(const Abstraction& abstraction) {
  // ...
  std::cout << abstraction.Operation();
  // ...
}
/**
 * The client code should be able to work with any pre-configured abstraction-
 * implementation combination.
 */

int main() {
  Implementation* implementation = new ConcreteImplementationA;
  Abstraction* abstraction = new Abstraction(implementation);
  ClientCode(*abstraction);
  std::cout << std::endl;
  delete implementation;
  delete abstraction;

  implementation = new ConcreteImplementationB;
  abstraction = new ExtendedAbstraction(implementation);
  ClientCode(*abstraction);

  delete implementation;
  delete abstraction;

  return 0;
}
```

Output.txt: **Execution result**

```
Abstraction: Base operation with:
ConcreteImplementationA: Here's the result on the platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationB: Here's the result on the platform B.
```



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Main article

Usage in C++

Code example

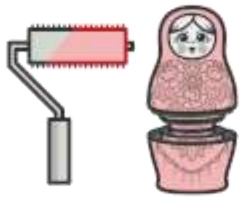
```
#include <algorithm>
#include <iostream>
#include <list>
#include <string>
/**
 * The base Component class declares common operations for both simple and
 * complex objects of a composition.
 */
class Component {
  /**
   * @var Component
   */
 protected:
  Component *parent_;
  /**
   * Optionally, the base Component can declare an interface for setting and
   * accessing a parent of the component in a tree structure. It can also
   * provide some default implementation for these methods.
   */
 public:
  virtual ~Component() {}
  void SetParent(Component *parent) {
    this->parent_ = parent;
  }
  Component *GetParent() const {
    return this->parent_;
  }
  /**
   * In some cases, it would be beneficial to define the child-management
   * operations right in the base Component class. This way, you won't need to
   * expose any concrete component classes to the client code, even during the
   * object tree assembly. The downside is that these methods will be empty for
   * the leaf-level components.
   */
  virtual void Add(Component *component) {}
  virtual void Remove(Component *component) {}
  /**
   * You can provide a method that lets the client code figure out whether a
   * component can bear children.
   */
  virtual bool IsComposite() const {
    return false;
  }
  /**
   * The base Component may implement some default behavior or leave it to
   * concrete classes (by declaring the method containing the behavior as
   * "abstract").
   */
  virtual std::string Operation() const = 0;
};
/**
 * The Leaf class represents the end objects of a composition. A leaf can't have
 * any children.
 *
 * Usually, it's the Leaf objects that do the actual work, whereas Composite
 * objects only delegate to their sub-components.
 */
class Leaf : public Component {
 public:
  std::string Operation() const override {
    return "Leaf";
  }
};
/**
 * The Composite class represents the complex components that may have children.
 * Usually, the Composite objects delegate the actual work to their children and
 * then "sum-up" the result.
 */
class Composite : public Component {
  /**
   * @var \SplObjectStorage
   */
 protected:
  std::list<Component *> children_;

 public:
  /**
   * A composite object can add or remove other components (both simple or
   * complex) to or from its child list.
   */
  void Add(Component *component) override {
    this->children_.push_back(component);
    component->SetParent(this);
  }
  /**
```

```
   * Have in mind that this method removes the pointer to the list but doesn't
   * frees the
   *     memory, you should do it manually or better use smart pointers.
   */
  void Remove(Component *component) override {
    children_.remove(component);
    component->SetParent(nullptr);
  }
  bool IsComposite() const override {
    return true;
  }
  /**
   * The Composite executes its primary logic in a particular way. It traverses
   * recursively through all its children, collecting and summing their results.
   * Since the composite's children pass these calls to their children and so
   * forth, the whole object tree is traversed as a result.
   */
  std::string Operation() const override {
    std::string result;
    for (const Component *c : children_) {
      if (c == children_.back()) {
        result += c->Operation();
      } else {
        result += c->Operation() + "+";
      }
    }
    return "Branch(" + result + ")";
  }
};
/**
 * The client code works with all of the components via the base interface.
 */
void ClientCode(Component *component) {
  // ...
  std::cout << "RESULT: " << component->Operation();
  // ...
}

/**
 * Thanks to the fact that the child-management operations are declared in the
 * base Component class, the client code can work with any component, simple or
 * complex, without depending on their concrete classes.
 */
void ClientCode2(Component *component1, Component *component2) {
  // ...
  if (component1->IsComposite()) {
    component1->Add(component2);
  }
  std::cout << "RESULT: " << component1->Operation();
  // ...
}

/**
 * This way the client code can support the simple leaf components...
 */
```

```cpp
int main() {
  Component *simple = new Leaf;
  std::cout << "Client: I've got a simple component:\n";
  ClientCode(simple);
  std::cout << "\n\n";
  /**
   * ...as well as the complex composites.
   */

  Component *tree = new Composite;
  Component *branch1 = new Composite;

  Component *leaf_1 = new Leaf;
  Component *leaf_2 = new Leaf;
  Component *leaf_3 = new Leaf;
  branch1->Add(leaf_1);
  branch1->Add(leaf_2);
  Component *branch2 = new Composite;
  branch2->Add(leaf_3);
  tree->Add(branch1);
  tree->Add(branch2);
  std::cout << "Client: Now I've got a composite tree:\n";
  ClientCode(tree);
  std::cout << "\n\n";

  std::cout << "Client: I don't need to check the components classes even when
managing the tree:\n";
  ClientCode2(tree, simple);
  std::cout << "\n";

  delete simple;
  delete tree;
  delete branch1;
  delete branch2;
  delete leaf_1;
  delete leaf_2;
  delete leaf_3;

  return 0;
}
```

Output.txt: Execution result

```
Client: I've got a simple component:
RESULT: Leaf

Client: Now I've got a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```



Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Main article

Usage in C++

Code example

```
/**
 * The base Component interface defines operations that can be altered by
 * decorators.
 */
class Component {
 public:
  virtual ~Component() {}
  virtual std::string Operation() const = 0;
};
/**
 * Concrete Components provide default implementations of the operations. There
 * might be several variations of these classes.
 */
class ConcreteComponent : public Component {
 public:
  std::string Operation() const override {
    return "ConcreteComponent";
  }
};
/**
 * The base Decorator class follows the same interface as the other components.
 * The primary purpose of this class is to define the wrapping interface for all
 * concrete decorators. The default implementation of the wrapping code might
 * include a field for storing a wrapped component and the means to initialize
 * it.
 */
class Decorator : public Component {
  /**
   * @var Component
   */
 protected:
  Component* component_;

 public:
  Decorator(Component* component) : component_(component) {
  }
  /**
   * The Decorator delegates all work to the wrapped component.
   */
  std::string Operation() const override {
    return this->component_->Operation();
  }
};
/**
 * Concrete Decorators call the wrapped object and alter its result in some way.
 */
class ConcreteDecoratorA : public Decorator {
  /**
   * Decorators may call parent implementation of the operation, instead of
   * calling the wrapped object directly. This approach simplifies extension of
   * decorator classes.
   */
 public:
  ConcreteDecoratorA(Component* component) : Decorator(component) {
  }
  std::string Operation() const override {
    return "ConcreteDecoratorA(" + Decorator::Operation() + ")";
  }
};
/**
 * Decorators can execute their behavior either before or after the call to a
 * wrapped object.
 */
class ConcreteDecoratorB : public Decorator {
 public:
  ConcreteDecoratorB(Component* component) : Decorator(component) {
  }

  std::string Operation() const override {
    return "ConcreteDecoratorB(" + Decorator::Operation() + ")";
  }
};
/**
 * The client code works with all objects using the Component interface. This
 * way it can stay independent of the concrete classes of components it works
 * with.
 */
void ClientCode(Component* component) {
  // ...
  std::cout << "RESULT: " << component->Operation();
  // ...
}

int main() {
  /**
   * This way the client code can support both simple components...
   */
  Component* simple = new ConcreteComponent;
  std::cout << "Client: I've got a simple component:\n";
  ClientCode(simple);
  std::cout << "\n\n";
  /**
   * ...as well as decorated ones.
   *
   * Note how decorators can wrap not only simple components but the other
   * decorators as well.
   */
  Component* decorator1 = new ConcreteDecoratorA(simple);
  Component* decorator2 = new ConcreteDecoratorB(decorator1);
  std::cout << "Client: Now I've got a decorated component:\n";
  ClientCode(decorator2);
  std::cout << "\n";

  delete simple;
  delete decorator1;
  delete decorator2;

  return 0;
}
```

Output.txt: **Execution result**

```
Client: I've got a simple component:
RESULT: ConcreteComponent

Client: Now I've got a decorated component:
RESULT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))
```

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

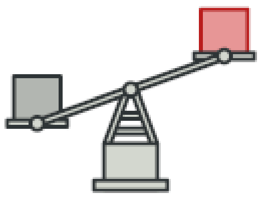
Main article
Usage in C++
Code example

```
/**
 * The Subsystem can accept requests either from the facade or client directly.
 * In any case, to the Subsystem, the Facade is yet another client, and it's not
 * a part of the Subsystem.
 */
class Subsystem1 {
 public:
  std::string Operation1() const {
    return "Subsystem1: Ready!\n";
  }
  // ...
  std::string OperationN() const {
    return "Subsystem1: Go!\n";
  }
```

```
};
/**
 * Some facades can work with multiple subsystems at the same time.
 */
class Subsystem2 {
 public:
  std::string Operation1() const {
    return "Subsystem2: Get ready!\n";
  }
  // ...
  std::string OperationZ() const {
    return "Subsystem2: Fire!\n";
  }
};

/**
 * The Facade class provides a simple interface to the complex logic of one or
 * several subsystems. The Facade delegates the client requests to the
 * appropriate objects within the subsystem. The Facade is also responsible for
 * managing their lifecycle. All of this shields the client from the undesired
 * complexity of the subsystem.
 */
class Facade {
 protected:
  Subsystem1 *subsystem1_;
  Subsystem2 *subsystem2_;
  /**
   * Depending on your application's needs, you can provide the Facade with
   * existing subsystem objects or force the Facade to create them on its own.
   */
 public:
  /**
   * In this case we will delegate the memory ownership to Facade Class
   */
  Facade(
      Subsystem1 *subsystem1 = nullptr,
      Subsystem2 *subsystem2 = nullptr) {
    this->subsystem1_ = subsystem1 ?: new Subsystem1;
    this->subsystem2_ = subsystem2 ?: new Subsystem2;
  }
  ~Facade() {
    delete subsystem1_;
    delete subsystem2_;
  }
  /**
   * The Facade's methods are convenient shortcuts to the sophisticated
   * functionality of the subsystems. However, clients get only to a fraction of
   * a subsystem's capabilities.
   */
  std::string Operation() {
    std::string result = "Facade initializes subsystems:\n";
    result += this->subsystem1_->Operation1();
    result += this->subsystem2_->Operation1();
    result += "Facade orders subsystems to perform the action:\n";
    result += this->subsystem1_->OperationN();
    result += this->subsystem2_->OperationZ();
    return result;
  }
};

/**
 * The client code works with complex subsystems through a simple interface
 * provided by the Facade. When a facade manages the lifecycle of the subsystem,
 * the client might not even know about the existence of the subsystem. This
 * approach lets you keep the complexity under control.
 */
void ClientCode(Facade *facade) {
  // ...
  std::cout << facade->Operation();
  // ...
}
/**
 * The client code may have some of the subsystem's objects already created. In
 * this case, it might be worthwhile to initialize the Facade with these objects
 * instead of letting the Facade create new instances.
 */

int main() {
  Subsystem1 *subsystem1 = new Subsystem1;
  Subsystem2 *subsystem2 = new Subsystem2;
  Facade *facade = new Facade(subsystem1, subsystem2);
  ClientCode(facade);

  delete facade;

  return 0;
}
```

Output.txt: Execution result

```
Facade initializes subsystems:
Subsystem1: Ready!
Subsystem2: Get ready!
Facade orders subsystems to perform the action:
Subsystem1: Go!
Subsystem2: Fire!
```



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Main article

Usage in C++

Code example

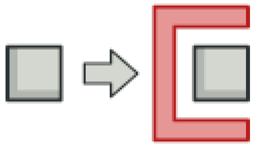
```
/**
 * Flyweight Design Pattern
 *
 * Intent: Lets you fit more objects into the available amount of RAM by sharing
 * common parts of state between multiple objects, instead of keeping all of the
 * data in each object.
 */

struct SharedState
{
    std::string brand_;
    std::string model_;
    std::string color_;

    SharedState(const std::string &brand, const std::string &model, const std::string
&color)
        : brand_(brand), model_(model), color_(color)
    {
    }

    friend std::ostream &operator<<(std::ostream &os, const SharedState &ss)
    {
        return os << "[ " << ss.brand_ << " , " << ss.model_ << " , " << ss.color_ <<
" ]";
    }
};

struct UniqueState
{
    std::string owner_;
    std::string plates_;

    UniqueState(const std::string &owner, const std::string &plates)
        : owner_(owner), plates_(plates)
    {
    }

    friend std::ostream &operator<<(std::ostream &os, const UniqueState &us)
    {
        return os << "[ " << us.owner_ << " , " << us.plates_ << " ]";
    }
};

/**
 * The Flyweight stores a common portion of the state (also called intrinsic
 * state) that belongs to multiple real business entities. The Flyweight accepts
 * the rest of the state (extrinsic state, unique for each entity) via its
 * method parameters.
 */
```

```
class Flyweight
{
private:
    SharedState *shared_state_;

public:
    Flyweight(const SharedState *shared_state) : shared_state_(new
SharedState(*shared_state))
    {
    }
    Flyweight(const Flyweight &other) : shared_state_(new
SharedState(*other.shared_state_))
    {
    }
    ~Flyweight()
    {
        delete shared_state_;
    }
    SharedState *shared_state() const
    {
        return shared_state_;
    }
    void Operation(const UniqueState &unique_state) const
    {
        std::cout << "Flyweight: Displaying shared (" << *shared_state_ << ") and
unique (" << unique_state << ") state.\n";
    }
};
/**
 * The Flyweight Factory creates and manages the Flyweight objects. It ensures
 * that flyweights are shared correctly. When the client requests a flyweight,
 * the factory either returns an existing instance or creates a new one, if it
 * doesn't exist yet.
 */
class FlyweightFactory
{
    /**
     * @var Flyweight[]
     */
private:
    std::unordered_map<std::string, Flyweight> flyweights_;
    /**
     * Returns a Flyweight's string hash for a given state.
     */
    std::string GetKey(const SharedState &ss) const
    {
        return ss.brand_ + "_" + ss.model_ + "_" + ss.color_;
    }

public:
    FlyweightFactory(std::initializer_list<SharedState> share_states)
    {
        for (const SharedState &ss : share_states)
        {
```

```cpp
            this->flyweights_.insert(std::make_pair<std::string, Flyweight>(this-
>GetKey(ss), Flyweight(&ss)));
        }
    }

    /**
     * Returns an existing Flyweight with a given state or creates a new one.
     */
    Flyweight GetFlyweight(const SharedState &shared_state)
    {
        std::string key = this->GetKey(shared_state);
        if (this->flyweights_.find(key) == this->flyweights_.end())
        {
            std::cout << "FlyweightFactory: Can't find a flyweight, creating new
one.\n";
            this->flyweights_.insert(std::make_pair(key, Flyweight(&shared_state)));
        }
        else
        {
            std::cout << "FlyweightFactory: Reusing existing flyweight.\n";
        }
        return this->flyweights_.at(key);
    }
    void ListFlyweights() const
    {
        size_t count = this->flyweights_.size();
        std::cout << "\nFlyweightFactory: I have " << count << " flyweights:\n";
        for (std::pair<std::string, Flyweight> pair : this->flyweights_)
        {
            std::cout << pair.first << "\n";
        }
    }
};

// ...
void AddCarToPoliceDatabase(
    FlyweightFactory &ff, const std::string &plates, const std::string &owner,
    const std::string &brand, const std::string &model, const std::string &color)
{
    std::cout << "\nClient: Adding a car to database.\n";
    const Flyweight &flyweight = ff.GetFlyweight({brand, model, color});
    // The client code either stores or calculates extrinsic state and passes it
    // to the flyweight's methods.
    flyweight.Operation({owner, plates});
}

/**
 * The client code usually creates a bunch of pre-populated flyweights in the
 * initialization stage of the application.
 */

int main()
{
    FlyweightFactory *factory = new FlyweightFactory({{"Chevrolet", "Camaro2018",
"pink"}, {"Mercedes Benz", "C300", "black"}, {"Mercedes Benz", "C500", "red"},
{"BMW", "M5", "red"}, {"BMW", "X6", "white"}});
    factory->ListFlyweights();

    AddCarToPoliceDatabase(*factory,
                            "CL234IR",
                            "James Doe",
                            "BMW",
                            "M5",
                            "red");

    AddCarToPoliceDatabase(*factory,
                            "CL234IR",
                            "James Doe",
                            "BMW",
                            "X1",
                            "red");
    factory->ListFlyweights();
    delete factory;

    return 0;
}
```

Output.txt: Execution result

```
FlyweightFactory: I have 5 flyweights:
BMW_X6_white
Mercedes Benz_C500_red
Mercedes Benz_C300_black
BMW_M5_red
Chevrolet_Camaro2018_pink

Client: Adding a car to database.
FlyweightFactory: Reusing existing flyweight.
Flyweight: Displaying shared ([ BMW , M5 , red ]) and unique ([ CL234IR , James Doe
]) state.

Client: Adding a car to database.
FlyweightFactory: Can't find a flyweight, creating new one.
Flyweight: Displaying shared ([ BMW , X1 , red ]) and unique ([ CL234IR , James Doe
]) state.

FlyweightFactory: I have 6 flyweights:
BMW_X1_red
Mercedes Benz_C300_black
BMW_X6_white
Mercedes Benz_C500_red
BMW_M5_red
Chevrolet_Camaro2018_pink
```



Proxy

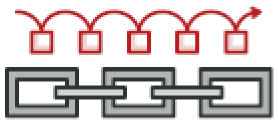
Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
Main article
Usage in C++
Code example

```cpp
#include <iostream>
/**
 * The Subject interface declares common operations for both RealSubject and the
 * Proxy. As long as the client works with RealSubject using this interface,
 * you'll be able to pass it a proxy instead of a real subject.
 */
class Subject {
 public:
  virtual void Request() const = 0;
};
/**
 * The RealSubject contains some core business logic. Usually, RealSubjects are
 * capable of doing some useful work which may also be very slow or sensitive -
 * e.g. correcting input data. A Proxy can solve these issues without any
 * changes to the RealSubject's code.
 */
class RealSubject : public Subject {
 public:
  void Request() const override {
    std::cout << "RealSubject: Handling request.\n";
  }
};
/**
 * The Proxy has an interface identical to the RealSubject.
 */
class Proxy : public Subject {
  /**
   * @var RealSubject
   */
 private:
  RealSubject *real_subject_;

  bool CheckAccess() const {
    // Some real checks should go here.
    std::cout << "Proxy: Checking access prior to firing a real request.\n";
    return true;
  }
  void LogAccess() const {
    std::cout << "Proxy: Logging the time of request.\n";
  }

  /**
   * The Proxy maintains a reference to an object of the RealSubject class. It
   * can be either lazy-loaded or passed to the Proxy by the client.
   */
 public:
  Proxy(RealSubject *real_subject) : real_subject_(new RealSubject(*real_subject)) {
  }

  ~Proxy() {
    delete real_subject_;
  }
  /**
   * The most common applications of the Proxy pattern are lazy loading,
   * caching, controlling the access, logging, etc. A Proxy can perform one of
   * these things and then, depending on the result, pass the execution to the
   * same method in a linked RealSubject object.
   */
  void Request() const override {
    if (this->CheckAccess()) {
      this->real_subject_->Request();
      this->LogAccess();
    }
  }
};
/**
 * The client code is supposed to work with all objects (both subjects and
 * proxies) via the Subject interface in order to support both real subjects and
 * proxies. In real life, however, clients mostly work with their real subjects
 * directly. In this case, to implement the pattern more easily, you can extend
 * your proxy from the real subject's class.
 */
void ClientCode(const Subject &subject) {
  // ...
  subject.Request();
  // ...
}

int main() {
  std::cout << "Client: Executing the client code with a real subject:\n";
  RealSubject *real_subject = new RealSubject;
  ClientCode(*real_subject);
  std::cout << "\n";
  std::cout << "Client: Executing the same client code with a proxy:\n";
  Proxy *proxy = new Proxy(real_subject);
  ClientCode(*proxy);

  delete real_subject;
  delete proxy;
  return 0;
}
```

Output.txt: Execution result

```
Client: Executing the client code with a real subject:
RealSubject: Handling request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling request.
Proxy: Logging the time of request.
```

Behavioral Patterns



Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

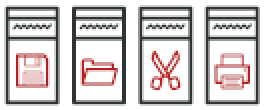
Main article
Usage in C++
Code example

```
/**
 * The Handler interface declares a method for building the chain of handlers.
 * It also declares a method for executing a request.
 */
class Handler {
 public:
  virtual Handler *SetNext(Handler *handler) = 0;
  virtual std::string Handle(std::string request) = 0;
};
/**
 * The default chaining behavior can be implemented inside a base handler class.
 */
class AbstractHandler : public Handler {
  /**
   * @var Handler
   */
 private:
  Handler *next_handler_;

 public:
  AbstractHandler() : next_handler_(nullptr) {
  }
  Handler *SetNext(Handler *handler) override {
    this->next_handler_ = handler;
    // Returning a handler from here will let us link handlers in a convenient
    // way like this:
    // $monkey->setNext($squirrel)->setNext($dog);
    return handler;
  }
  std::string Handle(std::string request) override {
    if (this->next_handler_) {
      return this->next_handler_->Handle(request);
    }

    return {};
  }
};
/**
 * All Concrete Handlers either handle a request or pass it to the next handler
 * in the chain.
 */
class MonkeyHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "Banana") {
      return "Monkey: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
  }
};
class SquirrelHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "Nut") {
      return "Squirrel: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
  }
};
class DogHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "MeatBall") {
      return "Dog: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
  }
};
/**
 * The client code is usually suited to work with a single handler. In most
 * cases, it is not even aware that the handler is part of a chain.
 */
void ClientCode(Handler &handler) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const std::string &f : food) {
```

```cpp
    std::cout << "Client: Who wants a " << f << "?\n";
    const std::string result = handler.Handle(f);
    if (!result.empty()) {
      std::cout << "  " << result;
    } else {
      std::cout << "  " << f << " was left untouched.\n";
    }
  }
}
/**
 * The other part of the client code constructs the actual chain.
 */
int main() {
  MonkeyHandler *monkey = new MonkeyHandler;
  SquirrelHandler *squirrel = new SquirrelHandler;
  DogHandler *dog = new DogHandler;
  monkey->SetNext(squirrel)->SetNext(dog);

  /**
   * The client should be able to send a request to any handler, not just the
   * first one in the chain.
   */
  std::cout << "Chain: Monkey > Squirrel > Dog\n\n";
  ClientCode(*monkey);
  std::cout << "\n";
  std::cout << "Subchain: Squirrel > Dog\n\n";
  ClientCode(*squirrel);

  delete monkey;
  delete squirrel;
  delete dog;

  return 0;
}
```

Output.txt: **Execution result**

```
Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
  Monkey: I'll eat the Banana.
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
  Banana was left untouched.
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.
```



Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Main article

Usage in C++

Code example

```cpp
/**
 * The Command interface declares a method for executing a command.
 */
class Command {
 public:
  virtual ~Command() {
  }
  virtual void Execute() const = 0;
};
/**
 * Some commands can implement simple operations on their own.
 */
class SimpleCommand : public Command {
 private:
  std::string pay_load_;

 public:
  explicit SimpleCommand(std::string pay_load) : pay_load_(pay_load) {
  }
  void Execute() const override {
    std::cout << "SimpleCommand: See, I can do simple things like printing (" <<
this->pay_load_ << ")\n";
  }
};

/**
 * The Receiver classes contain some important business logic. They know how to
 * perform all kinds of operations, associated with carrying out a request. In
 * fact, any class may serve as a Receiver.
 */
class Receiver {
 public:
  void DoSomething(const std::string &a) {
    std::cout << "Receiver: Working on (" << a << ".)\n";
  }
  void DoSomethingElse(const std::string &b) {
    std::cout << "Receiver: Also working on (" << b << ".)\n";
  }
};

/**
 * However, some commands can delegate more complex operations to other objects,
 * called "receivers."
 */
class ComplexCommand : public Command {
  /**
   * @var Receiver
   */
 private:
  Receiver *receiver_;
  /**
   * Context data, required for launching the receiver's methods.
   */
  std::string a_;
  std::string b_;
  /**
   * Complex commands can accept one or several receiver objects along with any
   * context data via the constructor.
   */
 public:
  ComplexCommand(Receiver *receiver, std::string a, std::string b) :
receiver_(receiver), a_(a), b_(b) {
  }
  /**
   * Commands can delegate to any methods of a receiver.
   */
  void Execute() const override {
    std::cout << "ComplexCommand: Complex stuff should be done by a receiver
object.\n";
    this->receiver_->DoSomething(this->a_);
    this->receiver_->DoSomethingElse(this->b_);
  }
};

/**
 * The Invoker is associated with one or several commands. It sends a request to
 * the command.
 */
class Invoker {
  /**
   * @var Command
   */
 private:
  Command *on_start_;
  /**
   * @var Command
   */
  Command *on_finish_;
  /**
   * Initialize commands.
   */
 public:
```

```cpp
  ~Invoker() {
    delete on_start_;
    delete on_finish_;
  }

  void SetOnStart(Command *command) {
    this->on_start_ = command;
  }
  void SetOnFinish(Command *command) {
    this->on_finish_ = command;
  }
  /**
   * The Invoker does not depend on concrete command or receiver classes. The
   * Invoker passes a request to a receiver indirectly, by executing a command.
   */
  void DoSomethingImportant() {
    std::cout << "Invoker: Does anybody want something done before I begin?\n";
    if (this->on_start_) {
      this->on_start_->Execute();
    }
    std::cout << "Invoker: ...doing something really important...\n";
    std::cout << "Invoker: Does anybody want something done after I finish?\n";
    if (this->on_finish_) {
      this->on_finish_->Execute();
    }
  }
};
/**
 * The client code can parameterize an invoker with any commands.
 */

int main() {
  Invoker *invoker = new Invoker;
  invoker->SetOnStart(new SimpleCommand("Say Hi!"));
  Receiver *receiver = new Receiver;
  invoker->SetOnFinish(new ComplexCommand(receiver, "Send email", "Save report"));
  invoker->DoSomethingImportant();

  delete invoker;
  delete receiver;

  return 0;
}
```

Output.txt: Execution result

```
Invoker: Does anybody want something done before I begin?
SimpleCommand: See, I can do simple things like printing (Say Hi!)
Invoker: ...doing something really important...
Invoker: Does anybody want something done after I finish?
ComplexCommand: Complex stuff should be done by a receiver object.
Receiver: Working on (Send email.)
Receiver: Also working on (Save report.)
```

Iterator

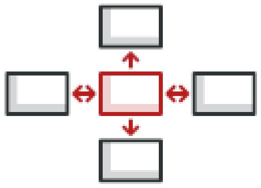
Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Main article
Usage in C++
Code example

```
/**
 * Iterator Design Pattern
 *
 * Intent: Lets you traverse elements of a collection without exposing its
 * underlying representation (list, stack, tree, etc.).
 */

#include <iostream>
#include <string>
#include <vector>

/**
 * C++ has its own implementation of iterator that works with a different
 * generics containers defined by the standard library.
 */

template <typename T, typename U>
class Iterator {
 public:
  typedef typename std::vector<T>::iterator iter_type;
  Iterator(U *p_data, bool reverse = false) : m_p_data_(p_data) {
    m_it_ = m_p_data_->m_data_.begin();
  }

  void First() {
    m_it_ = m_p_data_->m_data_.begin();
  }

  void Next() {
    m_it_++;
  }

  bool IsDone() {
    return (m_it_ == m_p_data_->m_data_.end());
  }

  iter_type Current() {
    return m_it_;
  }
 private:
  U *m_p_data_;
  iter_type m_it_;
};

/**
 * Generic Collections/Containers provides one or several methods for retrieving
 * fresh iterator instances, compatible with the collection class.
 */

template <class T>
class Container {
  friend class Iterator<T, Container>;

 public:
  void Add(T a) {
    m_data_.push_back(a);
  }

  Iterator<T, Container> *CreateIterator() {
    return new Iterator<T, Container>(this);
  }

 private:
  std::vector<T> m_data_;
};

class Data {
 public:
  Data(int a = 0) : m_data_(a) {}

  void set_data(int a) {
    m_data_ = a;
  }

  int data() {
    return m_data_;
  }

 private:
  int m_data_;
};

/**
 * The client code may or may not know about the Concrete Iterator or Collection
 * classes, for this implementation the container is generic so you can used
 * with an int or with a custom class.
 */
void ClientCode() {
  std::cout << "________________Iterator with
int______________________________________" << std::endl;
  Container<int> cont;

  for (int i = 0; i < 10; i++) {
    cont.Add(i);
  }

  Iterator<int, Container<int>> *it = cont.CreateIterator();
  for (it->First(); !it->IsDone(); it->Next()) {
    std::cout << *it->Current() << std::endl;
  }

  Container<Data> cont2;
  Data a(100), b(1000), c(10000);
  cont2.Add(a);
  cont2.Add(b);
  cont2.Add(c);

  std::cout << "________________Iterator with custom 
Class______________________________" << std::endl;
  Iterator<Data, Container<Data>> *it2 = cont2.CreateIterator();
  for (it2->First(); !it2->IsDone(); it2->Next()) {
    std::cout << it2->Current()->data() << std::endl;
  }
  delete it;
  delete it2;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
________________Iterator with int______________________________________
0
1
2
3
4
5
6
7
8
9
________________Iterator with custom Class______________________________
100
1000
10000
```



Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Main article

Usage in C++

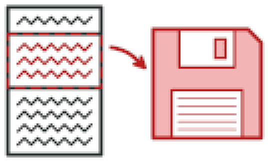
Code example

```
#include <iostream>
#include <string>
/**
 * The Mediator interface declares a method used by components to notify the
 * mediator about various events. The Mediator may react to these events and
 * pass the execution to other components.
 */
class BaseComponent;
class Mediator {
 public:
  virtual void Notify(BaseComponent *sender, std::string event) const = 0;
};

/**
 * The Base Component provides the basic functionality of storing a mediator's
 * instance inside component objects.
 */
class BaseComponent {
 protected:
  Mediator *mediator_;

 public:
  BaseComponent(Mediator *mediator = nullptr) : mediator_(mediator) {
  }
  void set_mediator(Mediator *mediator) {
    this->mediator_ = mediator;
  }
};

/**
 * Concrete Components implement various functionality. They don't depend on
 * other components. They also don't depend on any concrete mediator classes.
 */
class Component1 : public BaseComponent {
 public:
  void DoA() {
    std::cout << "Component 1 does A.\n";
    this->mediator_->Notify(this, "A");
  }
  void DoB() {
    std::cout << "Component 1 does B.\n";
    this->mediator_->Notify(this, "B");
  }
};
```

```
class Component2 : public BaseComponent {
 public:
  void DoC() {
    std::cout << "Component 2 does C.\n";
    this->mediator_->Notify(this, "C");
  }
  void DoD() {
    std::cout << "Component 2 does D.\n";
    this->mediator_->Notify(this, "D");
  }
};

/**
 * Concrete Mediators implement cooperative behavior by coordinating several
 * components.
 */
class ConcreteMediator : public Mediator {
 private:
  Component1 *component1_;
  Component2 *component2_;

 public:
  ConcreteMediator(Component1 *c1, Component2 *c2) : component1_(c1), component2_(c2)
{
    this->component1_->set_mediator(this);
    this->component2_->set_mediator(this);
  }
  void Notify(BaseComponent *sender, std::string event) const override {
    if (event == "A") {
      std::cout << "Mediator reacts on A and triggers following operations:\n";
      this->component2_->DoC();
    }
    if (event == "D") {
      std::cout << "Mediator reacts on D and triggers following operations:\n";
      this->component1_->DoB();
      this->component2_->DoC();
    }
  }
};

/**
 * The client code.
 */

void ClientCode() {
  Component1 *c1 = new Component1;
  Component2 *c2 = new Component2;
  ConcreteMediator *mediator = new ConcreteMediator(c1, c2);
  std::cout << "Client triggers operation A.\n";
  c1->DoA();
  std::cout << "\n";
  std::cout << "Client triggers operation D.\n";
  c2->DoD();

  delete c1;
  delete c2;
  delete mediator;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
Client triggers operation A.
Component 1 does A.
Mediator reacts on A and triggers following operations:
Component 2 does C.

Client triggers operation D.
Component 2 does D.
Mediator reacts on D and triggers following operations:
Component 1 does B.
Component 2 does C.
```



Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

Main article

Usage in C++

Code example

```
/**
 * The Memento interface provides a way to retrieve the memento's metadata, such
 * as creation date or name. However, it doesn't expose the Originator's state.
 */
class Memento {
 public:
  virtual ~Memento() {}
  virtual std::string GetName() const = 0;
  virtual std::string date() const = 0;
  virtual std::string state() const = 0;
};

/**
 * The Concrete Memento contains the infrastructure for storing the Originator's
 * state.
 */
```

```
class ConcreteMemento : public Memento {
 private:
  std::string state_;
  std::string date_;

 public:
  ConcreteMemento(std::string state) : state_(state) {
    this->state_ = state;
    std::time_t now = std::time(0);
    this->date_ = std::ctime(&now);
  }
  /**
   * The Originator uses this method when restoring its state.
   */
  std::string state() const override {
    return this->state_;
  }
  /**
   * The rest of the methods are used by the Caretaker to display metadata.
   */
  std::string GetName() const override {
    return this->date_ + " / (" + this->state_.substr(0, 9) + "...)";
  }
  std::string date() const override {
    return this->date_;
  }
};

/**
 * The Originator holds some important state that may change over time. It also
 * defines a method for saving the state inside a memento and another method for
 * restoring the state from it.
 */
class Originator {
  /**
   * @var string For the sake of simplicity, the originator's state is stored
   * inside a single variable.
   */
 private:
  std::string state_;

  std::string GenerateRandomString(int length = 10) {
    const char alphanum[] =
        "0123456789"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "abcdefghijklmnopqrstuvwxyz";
    int stringLength = sizeof(alphanum) - 1;

    std::string random_string;
    for (int i = 0; i < length; i++) {
      random_string += alphanum[std::rand() % stringLength];
    }
    return random_string;
  }
 public:
  Originator(std::string state) : state_(state) {
    std::cout << "Originator: My initial state is: " << this->state_ << "\n";
  }
  /**
   * The Originator's business logic may affect its internal state. Therefore,
   * the client should backup the state before launching methods of the business
   * logic via the save() method.
   */
  void DoSomething() {
    std::cout << "Originator: I'm doing something important.\n";
    this->state_ = this->GenerateRandomString(30);
    std::cout << "Originator: and my state has changed to: " << this->state_ << "\n";
  }

  /**
   * Saves the current state inside a memento.
   */
  Memento *Save() {
    return new ConcreteMemento(this->state_);
  }
  /**
   * Restores the Originator's state from a memento object.
   */
  void Restore(Memento *memento) {
    this->state_ = memento->state();
    std::cout << "Originator: My state has changed to: " << this->state_ << "\n";
  }
};

/**
 * The Caretaker doesn't depend on the Concrete Memento class. Therefore, it
 * doesn't have access to the originator's state, stored inside the memento. It
 * works with all mementos via the base Memento interface.
 */
class Caretaker {
  /**
   * @var Memento[]
   */
 private:
  std::vector<Memento *> mementos_;

  /**
   * @var Originator
   */
  Originator *originator_;

 public:
     Caretaker(Originator* originator) : originator_(originator) {
     }

     ~Caretaker() {
         for (auto m : mementos_) delete m;
     }
  void Backup() {
    std::cout << "\nCaretaker: Saving Originator's state...\n";
    this->mementos_.push_back(this->originator_->Save());
  }
  void Undo() {
    if (!this->mementos_.size()) {
      return;
    }
    Memento *memento = this->mementos_.back();
    this->mementos_.pop_back();
    std::cout << "Caretaker: Restoring state to: " << memento->GetName() << "\n";
    try {
      this->originator_->Restore(memento);
    } catch (...) {
      this->Undo();
    }
  }
  void ShowHistory() const {
    std::cout << "Caretaker: Here's the list of mementos:\n";
    for (Memento *memento : this->mementos_) {
      std::cout << memento->GetName() << "\n";
    }
  }
};
/**
 * Client code.
 */

void ClientCode() {
  Originator *originator = new Originator("Super-duper-super-puper-super.");
  Caretaker *caretaker = new Caretaker(originator);
  caretaker->Backup();
  originator->DoSomething();
  caretaker->Backup();
  originator->DoSomething();
  caretaker->Backup();
  originator->DoSomething();
  std::cout << "\n";
  caretaker->ShowHistory();
  std::cout << "\nClient: Now, let's rollback!\n\n";
  caretaker->Undo();
  std::cout << "\nClient: Once more!\n\n";
  caretaker->Undo();

  delete originator;
  delete caretaker;
}

int main() {
  std::srand(static_cast<unsigned int>(std::time(NULL)));
  ClientCode();
  return 0;
}
```

Output.txt: **Execution result**

```
Originator: My initial state is: Super-duper-super-puper-super.

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: uOInE8wmckHYPwZS7PtUTwuwZfCIbz

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: te6RGmykRpbqaWo5MEwjji1fpM1t5D

Caretaker: Saving Originator's state...
Originator: I'm doing something important.
Originator: and my state has changed to: hX5xWDVljcQ9ydD7StUfbBt5Z7pcSN

Caretaker: Here's the list of mementos:
Sat Oct 19 18:09:37 2019
 / (Super-dup...)
Sat Oct 19 18:09:37 2019
 / (uOInE8wmc...)
Sat Oct 19 18:09:37 2019
 / (te6RGmykR...)

Client: Now, let's rollback!

Caretaker: Restoring state to: Sat Oct 19 18:09:37 2019
 / (te6RGmykR...)
Originator: My state has changed to: te6RGmykRpbqaWo5MEwjji1fpM1t5D

Client: Once more!

Caretaker: Restoring state to: Sat Oct 19 18:09:37 2019
 / (uOInE8wmc...)
Originator: My state has changed to: uOInE8wmckHYPwZS7PtUTwuwZfCIbz
```

Observer

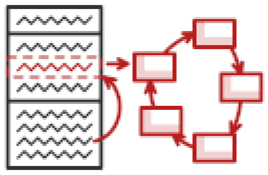
Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Main article
Usage in C++
Code example

```
/**
 * Observer Design Pattern
 *
 * Intent: Lets you define a subscription mechanism to notify multiple objects
 * about any events that happen to the object they're observing.
 *
 * Note that there's a lot of different terms with similar meaning associated
 * with this pattern. Just remember that the Subject is also called the
 * Publisher and the Observer is often called the Subscriber and vice versa.
 * Also the verbs "observe", "listen" or "track" usually mean the same thing.
 */

#include <iostream>
#include <list>
#include <string>

class IObserver {
 public:
  virtual ~IObserver(){};
  virtual void Update(const std::string &message_from_subject) = 0;
};

class ISubject {
 public:
  virtual ~ISubject(){};
  virtual void Attach(IObserver *observer) = 0;
  virtual void Detach(IObserver *observer) = 0;
  virtual void Notify() = 0;
};

/**
 * The Subject owns some important state and notifies observers when the state
 * changes.
 */

class Subject : public ISubject {
 public:
  virtual ~Subject() {
    std::cout << "Goodbye, I was the Subject.\n";
  }

  /**
   * The subscription management methods.
   */
  void Attach(IObserver *observer) override {
    list_observer_.push_back(observer);
  }
  void Detach(IObserver *observer) override {
    list_observer_.remove(observer);
  }
  void Notify() override {
    std::list<IObserver *>::iterator iterator = list_observer_.begin();
    HowManyObserver();
    while (iterator != list_observer_.end()) {
      (*iterator)->Update(message_);
      ++iterator;
    }
  }

  void CreateMessage(std::string message = "Empty") {
    this->message_ = message;
    Notify();
  }
  void HowManyObserver() {
    std::cout << "There are " << list_observer_.size() << " observers in the
list.\n";
  }

  /**
   * Usually, the subscription logic is only a fraction of what a Subject can
   * really do. Subjects commonly hold some important business logic, that
   * triggers a notification method whenever something important is about to
   * happen (or after it).
   */
  void SomeBusinessLogic() {
    this->message_ = "change message message";
    Notify();
    std::cout << "I'm about to do some thing important\n";
  }

 private:
  std::list<IObserver *> list_observer_;
  std::string message_;
};

class Observer : public IObserver {
 public:
  Observer(Subject &subject) : subject_(subject) {
    this->subject_.Attach(this);
    std::cout << "Hi, I'm the Observer \"" << ++Observer::static_number_ << "\".\n";
    this->number_ = Observer::static_number_;
  }
  virtual ~Observer() {
    std::cout << "Goodbye, I was the Observer \"" << this->number_ << "\".\n";
  }

  void Update(const std::string &message_from_subject) override {
    message_from_subject_ = message_from_subject;
    PrintInfo();
  }
  void RemoveMeFromTheList() {
    subject_.Detach(this);
    std::cout << "Observer \"" << number_ << "\" removed from the list.\n";
  }
  void PrintInfo() {
    std::cout << "Observer \"" << this->number_ << "\": a new message is available --
> " << this->message_from_subject_ << "\n";
  }

 private:
  std::string message_from_subject_;
  Subject &subject_;
  static int static_number_;
  int number_;
};

int Observer::static_number_ = 0;

void ClientCode() {
  Subject *subject = new Subject;
  Observer *observer1 = new Observer(*subject);
  Observer *observer2 = new Observer(*subject);
  Observer *observer3 = new Observer(*subject);
  Observer *observer4;
  Observer *observer5;

  subject->CreateMessage("Hello World! :D");
  observer3->RemoveMeFromTheList();

  subject->CreateMessage("The weather is hot today! :p");
  observer4 = new Observer(*subject);

  observer2->RemoveMeFromTheList();
  observer5 = new Observer(*subject);

  subject->CreateMessage("My new car is great! ;)");
  observer5->RemoveMeFromTheList();

  observer4->RemoveMeFromTheList();
  observer1->RemoveMeFromTheList();

  delete observer5;
  delete observer4;
  delete observer3;
  delete observer2;
  delete observer1;
  delete subject;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: **Execution result**

```
Hi, I'm the Observer "1".
Hi, I'm the Observer "2".
Hi, I'm the Observer "3".
There are 3 observers in the list.
Observer "1": a new message is available --> Hello World! :D
Observer "2": a new message is available --> Hello World! :D
Observer "3": a new message is available --> Hello World! :D
Observer "3" removed from the list.
There are 2 observers in the list.
Observer "1": a new message is available --> The weather is hot today! :p
Observer "2": a new message is available --> The weather is hot today! :p
Hi, I'm the Observer "4".
Observer "2" removed from the list.
Hi, I'm the Observer "5".
There are 3 observers in the list.
Observer "1": a new message is available --> My new car is great! ;)
Observer "4": a new message is available --> My new car is great! ;)
Observer "5": a new message is available --> My new car is great! ;)
Observer "5" removed from the list.
Observer "4" removed from the list.
Observer "1" removed from the list.
Goodbye, I was the Observer "5".
Goodbye, I was the Observer "4".
Goodbye, I was the Observer "3".
Goodbye, I was the Observer "2".
Goodbye, I was the Observer "1".
Goodbye, I was the Subject.
```



State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Main article

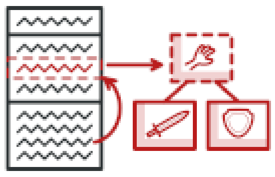
Usage in C++

Code example

```
#include <iostream>
#include <typeinfo>
/**
 * The base State class declares methods that all Concrete State should
 * implement and also provides a backreference to the Context object, associated
 * with the State. This backreference can be used by States to transition the
 * Context to another State.
 */

class Context;

class State {
  /**
   * @var Context
   */
 protected:
  Context *context_;

 public:
  virtual ~State() {
  }
  void set_context(Context *context) {
    this->context_ = context;
  }

  virtual void Handle1() = 0;
  virtual void Handle2() = 0;
};

/**
 * The Context defines the interface of interest to clients. It also maintains a
 * reference to an instance of a State subclass, which represents the current
 * state of the Context.
 */
class Context {
  /**
   * @var State A reference to the current state of the Context.
   */
 private:
  State *state_;

 public:
  Context(State *state) : state_(nullptr) {
    this->TransitionTo(state);
  }
  ~Context() {
    delete state_;
  }
  /**
   * The Context allows changing the State object at runtime.
   */
  void TransitionTo(State *state) {
    std::cout << "Context: Transition to " << typeid(*state).name() << ".\n";
    if (this->state_ != nullptr)
      delete this->state_;
    this->state_ = state;
    this->state_->set_context(this);
  }
  /**
   * The Context delegates part of its behavior to the current State object.
   */
  void Request1() {
    this->state_->Handle1();
  }
  void Request2() {
    this->state_->Handle2();
  }
};

/**
 * Concrete States implement various behaviors, associated with a state of the
 * Context.
 */

class ConcreteStateA : public State {
 public:
  void Handle1() override;

  void Handle2() override {
    std::cout << "ConcreteStateA handles request2.\n";
  }
};

class ConcreteStateB : public State {
 public:
  void Handle1() override {
    std::cout << "ConcreteStateB handles request1.\n";
  }
  void Handle2() override {
    std::cout << "ConcreteStateB handles request2.\n";
    std::cout << "ConcreteStateB wants to change the state of the context.\n";
    this->context_->TransitionTo(new ConcreteStateA);
  }
};

void ConcreteStateA::Handle1() {
  {
    std::cout << "ConcreteStateA handles request1.\n";
    std::cout << "ConcreteStateA wants to change the state of the context.\n";

    this->context_->TransitionTo(new ConcreteStateB);
  }
}

/**
 * The client code.
 */
void ClientCode() {
  Context *context = new Context(new ConcreteStateA);
  context->Request1();
  context->Request2();
  delete context;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
Context: Transition to 14ConcreteStateA.
ConcreteStateA handles request1.
ConcreteStateA wants to change the state of the context.
Context: Transition to 14ConcreteStateB.
ConcreteStateB handles request2.
ConcreteStateB wants to change the state of the context.
Context: Transition to 14ConcreteStateA.
```



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Main article

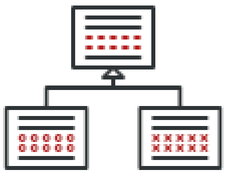
Usage in C++

Code example

```
/**
 * The Strategy interface declares operations common to all supported versions
 * of some algorithm.
 *
 * The Context uses this interface to call the algorithm defined by Concrete
 * Strategies.
 */
class Strategy
{
public:
    virtual ~Strategy() = default;
    virtual std::string doAlgorithm(std::string_view data) const = 0;
};

/**
 * The Context defines the interface of interest to clients.
 */

class Context
{
    /**
     * @var Strategy The Context maintains a reference to one of the Strategy
     * objects. The Context does not know the concrete class of a strategy. It
     * should work with all strategies via the Strategy interface.
     */
private:
    std::unique_ptr<Strategy> strategy_;
    /**
     * Usually, the Context accepts a strategy through the constructor, but also
     * provides a setter to change it at runtime.
     */
public:
    explicit Context(std::unique_ptr<Strategy> &&strategy = {}) :
strategy_(std::move(strategy))
    {
    }
    /**
     * Usually, the Context allows replacing a Strategy object at runtime.
     */
    void set_strategy(std::unique_ptr<Strategy> &&strategy)
    {
        strategy_ = std::move(strategy);
    }
    /**
     * The Context delegates some work to the Strategy object instead of
     * implementing +multiple versions of the algorithm on its own.
     */
    void doSomeBusinessLogic() const
    {
        if (strategy_) {
            std::cout << "Context: Sorting data using the strategy (not sure how
it'll do it)\n";
            std::string result = strategy_->doAlgorithm("aecbd");
            std::cout << result << "\n";
        } else {
            std::cout << "Context: Strategy isn't set\n";
        }
    }
};

/**
 * Concrete Strategies implement the algorithm while following the base Strategy
 * interface. The interface makes them interchangeable in the Context.
 */
class ConcreteStrategyA : public Strategy
{
public:
    std::string doAlgorithm(std::string_view data) const override
    {
        std::string result(data);
        std::sort(std::begin(result), std::end(result));

        return result;
    }
};
class ConcreteStrategyB : public Strategy
{
    std::string doAlgorithm(std::string_view data) const override
    {
        std::string result(data);
        std::sort(std::begin(result), std::end(result), std::greater<>());

        return result;
    }
};
/**
 * The client code picks a concrete strategy and passes it to the context. The
 * client should be aware of the differences between strategies in order to make
 * the right choice.
 */

void clientCode()
{
    Context context(std::make_unique<ConcreteStrategyA>());
    std::cout << "Client: Strategy is set to normal sorting.\n";
    context.doSomeBusinessLogic();
    std::cout << "\n";
    std::cout << "Client: Strategy is set to reverse sorting.\n";
    context.set_strategy(std::make_unique<ConcreteStrategyB>());
    context.doSomeBusinessLogic();
}

int main()
{
    clientCode();
    return 0;
}
```

Output.txt: **Execution result**

```
Client: Strategy is set to normal sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
abcde

Client: Strategy is set to reverse sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
edcba
```



Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Main article

Usage in C++

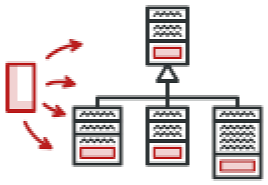
Code example

```
/**
 * The Abstract Class defines a template method that contains a skeleton of some
 * algorithm, composed of calls to (usually) abstract primitive operations.
 *
 * Concrete subclasses should implement these operations, but leave the template
 * method itself intact.
 */
class AbstractClass {
  /**
   * The template method defines the skeleton of an algorithm.
   */
 public:
  void TemplateMethod() const {
    this->BaseOperation1();
    this->RequiredOperations1();
    this->BaseOperation2();
    this->Hook1();
    this->RequiredOperation2();
    this->BaseOperation3();
    this->Hook2();
  }
  /**
   * These operations already have implementations.
   */
 protected:
  void BaseOperation1() const {
    std::cout << "AbstractClass says: I am doing the bulk of the work\n";
  }
  void BaseOperation2() const {
    std::cout << "AbstractClass says: But I let subclasses override some
operations\n";
  }
  void BaseOperation3() const {
    std::cout << "AbstractClass says: But I am doing the bulk of the work anyway\n";
  }
  /**
   * These operations have to be implemented in subclasses.
   */
  virtual void RequiredOperations1() const = 0;
  virtual void RequiredOperation2() const = 0;
  /**
   * These are "hooks." Subclasses may override them, but it's not mandatory
   * since the hooks already have default (but empty) implementation. Hooks
   * provide additional extension points in some crucial places of the
   * algorithm.
   */
  virtual void Hook1() const {}
  virtual void Hook2() const {}
};
/**
 * Concrete classes have to implement all abstract operations of the base class.
 * They can also override some operations with a default implementation.
 */
class ConcreteClass1 : public AbstractClass {
 protected:
  void RequiredOperations1() const override {
    std::cout << "ConcreteClass1 says: Implemented Operation1\n";
  }
  void RequiredOperation2() const override {
    std::cout << "ConcreteClass1 says: Implemented Operation2\n";
  }
};
/**
 * Usually, concrete classes override only a fraction of base class' operations.
 */
class ConcreteClass2 : public AbstractClass {
 protected:
  void RequiredOperations1() const override {
```

```cpp
    std::cout << "ConcreteClass2 says: Implemented Operation1\n";
  }
  void RequiredOperation2() const override {
    std::cout << "ConcreteClass2 says: Implemented Operation2\n";
  }
  void Hook1() const override {
    std::cout << "ConcreteClass2 says: Overridden Hook1\n";
  }
};
/**
 * The client code calls the template method to execute the algorithm. Client
 * code does not have to know the concrete class of an object it works with, as
 * long as it works with objects through the interface of their base class.
 */
void ClientCode(AbstractClass *class_) {
  // ...
  class_->TemplateMethod();
  // ...
}

int main() {
  std::cout << "Same client code can work with different subclasses:\n";
  ConcreteClass1 *concreteClass1 = new ConcreteClass1;
  ClientCode(concreteClass1);
  std::cout << "\n";
  std::cout << "Same client code can work with different subclasses:\n";
  ConcreteClass2 *concreteClass2 = new ConcreteClass2;
  ClientCode(concreteClass2);
  delete concreteClass1;
  delete concreteClass2;
  return 0;
}
```

Output.txt: **Execution result**

```
Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass1 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass1 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway

Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass2 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass2 says: Overridden Hook1
ConcreteClass2 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway
```



Visitor

Lets you separate algorithms from the objects on which they operate.

Main article

Usage in C++

Code example

```
/**
 * The Visitor Interface declares a set of visiting methods that correspond to
 * component classes. The signature of a visiting method allows the visitor to
 * identify the exact class of the component that it's dealing with.
 */
class ConcreteComponentA;
class ConcreteComponentB;

class Visitor {
 public:
  virtual void VisitConcreteComponentA(const ConcreteComponentA *element) const = 0;
  virtual void VisitConcreteComponentB(const ConcreteComponentB *element) const = 0;
};

/**
 * The Component interface declares an `accept` method that should take the base
 * visitor interface as an argument.
 */

class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};

/**
 * Each Concrete Component must implement the `Accept` method in such a way that
 * it calls the visitor's method corresponding to the component's class.
 */
class ConcreteComponentA : public Component {
  /**
   * Note that we're calling `visitConcreteComponentA`, which matches the
   * current class name. This way we let the visitor know the class of the
   * component it works with.
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  /**
```

```
   * Concrete Components may have special methods that don't exist in their base
   * class or interface. The Visitor is still able to use these methods since
   * it's aware of the component's concrete class.
   */
  std::string ExclusiveMethodOfConcreteComponentA() const {
    return "A";
  }
};

class ConcreteComponentB : public Component {
  /**
   * Same here: visitConcreteComponentB => ConcreteComponentB
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentB(this);
  }
  std::string SpecialMethodOfConcreteComponentB() const {
    return "B";
  }
};

/**
 * Concrete Visitors implement several versions of the same algorithm, which can
 * work with all concrete component classes.
 *
 * You can experience the biggest benefit of the Visitor pattern when using it
 * with a complex object structure, such as a Composite tree. In this case, it
 * might be helpful to store some intermediate state of the algorithm while
 * executing visitor's methods over various objects of the structure.
 */
class ConcreteVisitor1 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " +
ConcreteVisitor1\n";
  }

  void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
    std::cout << element->SpecialMethodOfConcreteComponentB() << " +
ConcreteVisitor1\n";
  }
};

class ConcreteVisitor2 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " +
ConcreteVisitor2\n";
  }
  void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
    std::cout << element->SpecialMethodOfConcreteComponentB() << " +
ConcreteVisitor2\n";
  }
};
```

```cpp
/**
 * The client code can run visitor operations over any set of elements without
 * figuring out their concrete classes. The accept operation directs a call to
 * the appropriate operation in the visitor object.
 */
void ClientCode(std::array<const Component *, 2> components, Visitor *visitor) {
  // ...
  for (const Component *comp : components) {
    comp->Accept(visitor);
  }
  // ...
}

int main() {
  std::array<const Component *, 2> components = {new ConcreteComponentA, new
ConcreteComponentB};
  std::cout << "The client code works with all visitors via the base Visitor
interface:\n";
  ConcreteVisitor1 *visitor1 = new ConcreteVisitor1;
  ClientCode(components, visitor1);
  std::cout << "\n";
  std::cout << "It allows the same client code to work with different types of
visitors:\n";
  ConcreteVisitor2 *visitor2 = new ConcreteVisitor2;
  ClientCode(components, visitor2);

  for (const Component *comp : components) {
    delete comp;
  }
  delete visitor1;
  delete visitor2;

  return 0;
}
```

Output.txt: Execution result

```
The client code works with all visitors via the base Visitor interface:
A + ConcreteVisitor1
B + ConcreteVisitor1

It allows the same client code to work with different types of visitors:
A + ConcreteVisitor2
B + ConcreteVisitor2
```