Design Patterns

Design patterns help programmers in many ways; as these patterns make programmes more understandable, easily adaptable across other systems, easier to upgrade, etc. They are basically blueprints for common programming problems.

* Creational

These patterns are for creating flexible, efficient objects.

* + Builder
    - To create complex object piecewise & succinity

To create an object with too many specifications can be done by using parameters for every specification. Yet, implementing like this is costful since there can be many unused parameters. Instead of using parameters we can use ‘builder’s. They are simply helper functions for other vicinities of an object. Sometimes a ‘director’ is needed for directing building steps.

Example: There can be many options for a car (color, park censors, cruise control, speed limiter, etc.).

class Car():

def \_\_init\_\_(self) -> None:

self.options = []

def add(self, option) -> None:

self.options.append(option)

class Builder():

def \_\_init\_\_(self) -> None:

self.car=Car()

def option\_A(self) -> None:

self.car.add("option\_A")

print("Car has option A")

def option\_B(self) -> None:

self.car.add("option\_B")

print("Car has option B")

def option\_C(self) -> None:

self.car.add("option\_C")

print("Car has option C")

* + Factory
    - To create wholesale objects unlike builder

While the builder method works from piece by piece to form an object, the factory method works as constructing objects from one source (class). Subclasses help us to define the factory method to create an instance of the appropriate product.

Example: Most cars are created by factories.

#Define a Factory class

class Creator():

def factory\_method(self):

pass

def operation(self):

product = self.factory\_method()

if product:

product.operation()

#Define Factory subclasses

class ConcreteCreator\_1(Creator):

def factory\_method(self):

return Car\_1()

class ConcreteCreator\_2(Creator):

def factory\_method(self):

return Car\_2()

#Define a Car class

class Car():

def operation(self):

pass

#Define Car subclasses

class Car\_1(Car):

def operation(self):

print("Producing Car\_1")

class Car\_2(Car):

def operation(self):

print("Producing Car\_2")

* + Prototype
    - To create new object cheaply

Instead of creating a new object from ground-up, copying an already existing object is simpler and faster (with proper background).

Example: With prototype class an object can be instantiated, registered, and cloned (if object exists).

class Prototype:

def \_\_init\_\_(self):

self.objects = {}

def register\_object(self, name, obj):

self.objects[name] = obj

def unregister\_object(self, name):

del self.objects[name]

def clone(self, name, \*\*kwargs):

obj = self.objects.get(name)

if obj:

cloned\_obj = obj.clone(\*\*kwargs)

return cloned\_obj

else:

raise ValueError(f"Object with name '{name}' does not exist.")

class ConcreteObject:

def \_\_init\_\_(self, x, y, color):

self.x = x

self.y = y

self.color = color

def clone(self, \*\*kwargs):

x = kwargs.get('x', self.x)

y = kwargs.get('y', self.y)

color = kwargs.get('color', self.color)

obj = self.\_\_class\_\_(x, y, color)

return obj

* + Singleton
    - To create one and only on instance of class

There can be only one instance of a class. This can be implemented by checking if there exists that class before adding a new class.

Example:

class Singleton:

\_instance = None

@staticmethod

def get\_instance():

if Singleton.\_instance is None:

Singleton.\_instance = Singleton()

return Singleton.\_instance

def \_\_init\_\_(self):

if Singleton.\_instance is not None:

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

* Structural

These patterns are for constructing a system; using objects and classes, maintaining flexibility and efficiency.

* + Adapter
    - To get the interface you want from interface you have

Objects we have and objects that are wanted from us (by somebody else, or by another program) can be in different forms. To solve this we use adapters, we alter objects for expectations.

There are two main ways to implement this: Object Composition, inheriting interface of one object while wrapping other object (can be implemented in all programming languages); Inheritance, inheriting interface of both objects (programming language must support multiple inheritance).

Example:

class Target:

def request(self) -> str:

return "Target: The default target's behavior."

class Adaptee:

def specific\_request(self) -> str:

return ".eetpadA eht fo roivaheb laicepS"

class Adapter(Target):

#This is an example of object composition.

#To adapt with inheritance we take Adaptee as an argument, and we will not need an initiator (constructor).

def \_\_init\_\_(self, adaptee: Adaptee) -> None:

self.adaptee = adaptee

def request(self) -> str:

return f"Adapter: (TRANSLATED) {self.adaptee.specific\_request()[::-1]}"

def client\_code(target: Target) -> None:

#Client code is crucial (requests can change) but the target interface must be met.

try:

print(target.request(), end="")

except AttributeError as error:

print(f"Error: {error}")

* + Bridge
    - To separate the interface from implementation

Thisis a structural design pattern that 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. For an object (circle) ‘abstraction’ includes initiators of the object (radius, location, etc.), ‘implementation’ includes more intricate options of the object (color, fullness of inside, etc.); and the bridge method builds a bridge between these two hierarchies.

Example:

# Implementing the Abstraction class

class Shape:

def \_\_init\_\_(self, color):

self.color = color

def draw(self):

pass

# Implementing the Concrete Implementor classes

class RedColor:

def fill\_color(self):

return "Red"

class BlueColor:

def fill\_color(self):

return "Blue"

# Implementing the Refined Abstraction classes

class Circle(Shape):

def draw(self):

return f"Drawing Circle with {self.color.fill\_color()} color"

class Square(Shape):

def draw(self):

return f"Drawing Square with {self.color.fill\_color()} color"

* + Composite
    - To treat individual & group of objects uniformly

This is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects. Working with this method only makes sense if the core model of the app is representable as a tree.

Example: A component is an object structured as a tree. Composite and Leaf classes build this object.

# Base class for components

class Component:

def operation(self):

pass

# Leaf component

class Leaf(Component):

def operation(self):

print("Leaf operation")

# Composite component

class Composite(Component):

def \_\_init\_\_(self):

self.\_children = []

def add(self, component):

self.\_children.append(component)

def remove(self, component):

self.\_children.remove(component)

def operation(self):

print("Composite operation")

for child in self.\_children:

child.operation()

* + Decorator
    - To facilitate the additional functionality to objects

This is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. Instead of creating new subclasses for different combinations of classes, create composition/aggregation (Aggregation: object A contains objects B; B can live without A. Composition: object A consists of objects B; A manages life cycle of B; B can’t live without A.) links between classes.

Example:

# Define the base component interface

class Component:

def operation(self):

pass

# Concrete component

class ConcreteComponent(Component):

def operation(self):

print("Performing the operation in the concrete component.")

# Base decorator class

class Decorator(Component):

def \_\_init\_\_(self, component):

self.component = component

def operation(self):

self.component.operation()

# Concrete decorator class

class ConcreteDecoratorA(Decorator):

def operation(self):

super().operation()

self.additional\_operation()

def additional\_operation(self):

print("Adding additional operation in ConcreteDecoratorA.")

# Another concrete decorator class

class ConcreteDecoratorB(Decorator):

def operation(self):

super().operation()

self.additional\_operation()

def additional\_operation(self):

print("Adding additional operation in ConcreteDecoratorB.")

* + Facade
    - To provide unified interface by hiding system complexities

This is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Example: To hide complex subclasses write a simple interface.

# Subsystem 1

class Subsystem1:

def method1(self):

print("Subsystem 1: Method 1")

def method2(self):

print("Subsystem 1: Method 2")

# Subsystem 2

class Subsystem2:

def method1(self):

print("Subsystem 2: Method 1")

def method2(self):

print("Subsystem 2: Method 2")

# Facade

class Facade:

def \_\_init\_\_(self):

self.subsystem1 = Subsystem1()

self.subsystem2 = Subsystem2()

def operation(self):

self.subsystem1.method1()

self.subsystem1.method2()

self.subsystem2.method1()

self.subsystem2.method2()

* + Flyweight
    - To avoid redundancy when storing data

This is a structural design pattern that lets you fit more objects into the available amount of memory by sharing common parts of state between multiple objects instead of keeping all of the data in each object. When objects share similar parts keep their shared data in the same memory location for memory efficiency, yet this memory efficiency gets traded with increasing CPU cycles.

Example:

class Flyweight:

#keeps data. by default initialized to shared data, it can be changed to unique data.

def \_\_init\_\_(self, shared\_state):

self.shared\_state = shared\_state

def operation(self, unique\_state):

print(f"Flyweight: shared state({self.shared\_state}) and unique state({unique\_state})")

class FlyweightFactory:

#creates new objects.

def \_\_init\_\_(self):

self.flyweights = {}

def get\_flyweight(self, shared\_state):

if shared\_state not in self.flyweights:

self.flyweights[shared\_state] = Flyweight(shared\_state)

return self.flyweights[shared\_state]

* + Proxy
    - An interface for accessing a particular resource

This is a structural design pattern that 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. This is needed because real objects can not always be present and yet it may get called any time. Initializing an object after a call can be impossible since there can be private classes, functions; and it would be slow. Therefore, we will initialize a proxy object that is a placeholder of the real object; do the updates, read necessary data from this proxy object.

Example: An image database with proxy object implementation

# Subject interface

class Image:

def display(self):

pass

# Real Subject class

class RealImage(Image):

def \_\_init\_\_(self, filename):

self.\_filename = filename

self.load\_from\_disk()

def load\_from\_disk(self):

print(f"Loading image: {self.\_filename}")

def display(self):

print(f"Displaying image: {self.\_filename}")

# Proxy class

class ProxyImage(Image):

def \_\_init\_\_(self, filename):

self.\_filename = filename

self.\_real\_image = None

def display(self):

if self.\_real\_image is None:

self.\_real\_image = RealImage(self.\_filename)

self.\_real\_image.display()

* Behavioural

These patterns are for assigning responsibilities and communicating effectively between objects.

* + Chain of Responsibility
    - To handle the request by more than one object

This is a behavioral design pattern that 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. This works when there are multiple security levels in a project such as an e-commerce website with users: buyer, seller, admin there should be different levels of authorization.

Example: There is a Handler class with subclasses A, B, C which can have requests and successors.

class Handler:

def \_\_init\_\_(self, successor=None):

self.\_successor = successor

def handle\_request(self, request):

pass

class ConcreteHandlerA(Handler):

def handle\_request(self, request):

if request == 'A':

print("ConcreteHandlerA handles the request.")

elif self.\_successor is not None:

self.\_successor.handle\_request(request)

class ConcreteHandlerB(Handler):

def handle\_request(self, request):

if request == 'B':

print("ConcreteHandlerB handles the request.")

elif self.\_successor is not None:

self.\_successor.handle\_request(request)

class ConcreteHandlerC(Handler):

def handle\_request(self, request):

if request == 'C':

print("ConcreteHandlerC handles the request.")

elif self.\_successor is not None:

self.\_successor.handle\_request(request)

* + Command
    - To decouple the sender & receiver

This is a behavioral design pattern that 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. This method does not help with writing an uncomplicated code since we need to write every layer for both sender and receiver.

Example:

# Receiver class

class Light:

def turn\_on(self):

print("Light is on.")

def turn\_off(self):

print("Light is off.")

# Command interface

class Command:

def execute(self):

pass

# Concrete command classes

class TurnOnCommand(Command):

def \_\_init\_\_(self, light):

self.light = light

def execute(self):

self.light.turn\_on()

class TurnOffCommand(Command):

def \_\_init\_\_(self, light):

self.light = light

def execute(self):

self.light.turn\_off()

# Invoker class (sender interface)

class RemoteControl:

def \_\_init\_\_(self):

self.command = None

def set\_command(self, command):

self.command = command

def press\_button(self): #executes command

if self.command is not None:

self.command.execute()

else:

print("No command is set.")

* + Interpreter
    - To process structured text data

This is a behavioral design pattern that defines a way to evaluate or interpret sentences in a language. It provides a way to represent and evaluate grammar or language rules. The pattern is commonly used in the field of programming languages, compilers, and natural language processing.

Example:

# Abstract Expression

class Expression:

def interpret(self, context):

pass

# Terminal Expression

class Number(Expression):

def \_\_init\_\_(self, value):

self.value = value

def interpret(self, context):

return self.value

# Terminal Expression

class Plus(Expression):

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

def interpret(self, context):

return self.left.interpret(context) + self.right.interpret(context)

# Context

class Context:

def \_\_init\_\_(self):

self.variables = {}

def set\_variable(self, variable, value):

self.variables[variable] = value

def get\_variable(self, variable):

return self.variables.get(variable)

* + Iterator
    - To facilitate the traversal of data structure

This is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Example: This is a list iterator. After initializing data to a list we can go through the list.

class MyIterator:

def \_\_init\_\_(self, data):

self.data = data

self.index = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.index >= len(self.data):

raise StopIteration

value = self.data[self.index]

self.index += 1

print(value) # Print the value inside the iterator

return value

* + Mediator
    - To facilitate communication between objects

This is a behavioral design pattern that 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. This makes communications centralized.

Example: There are component objects, between these objects communications done by mediator.

class Mediator:

def \_\_init\_\_(self):

self.components = []

def add\_component(self, component):

self.components.append(component)

def notify(self, sender, event):

for component in self.components:

if component != sender:

component.receive(event)

class Component:

def \_\_init\_\_(self, mediator):

self.mediator = mediator

def send(self, event):

self.mediator.notify(self, event)

def receive(self, event):

print(f"Received event: {event}")

* + Memento
    - To store/restore the state of the object

This is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Example:

class Originator:

#user interface for changing and restoring states

def \_\_init\_\_(self):

self.\_state = None

self.\_caretaker = Caretaker()

def set\_state(self, state):

print(f"Setting state to: {state}")

self.\_state = state

self.\_caretaker.add\_memento(Memento(self.\_state))

def restore\_state(self, index):

memento = self.\_caretaker.get\_memento(index)

self.\_state = memento.get\_state()

print(f"State restored to: {self.\_state}")

class Memento:

#is a container that stores states

def \_\_init\_\_(self, state):

self.\_state = state

def get\_state(self):

return self.\_state

class Caretaker:

#keeps track of the stored data of states

def \_\_init\_\_(self):

self.\_mementos = []

def add\_memento(self, memento):

self.\_mementos.append(memento)

def get\_memento(self, index):

return self.\_mementos[index]

* + Observer
    - To get notifications when events happen

This is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

Example: Observers are subscribers to a subject. After updates occur observers get notified (this can be done automatically).

class Subject:

def \_\_init\_\_(self):

self.\_observers = []

def attach(self, observer):

self.\_observers.append(observer)

def detach(self, observer):

self.\_observers.remove(observer)

def notify(self, message):

for observer in self.\_observers:

observer.update(message)

class Observer:

def update(self, message):

pass

# Example notifications (users)

class EmailNotification(Observer):

def update(self, message):

print("Sending email notification:", message)

class SMSNotification(Observer):

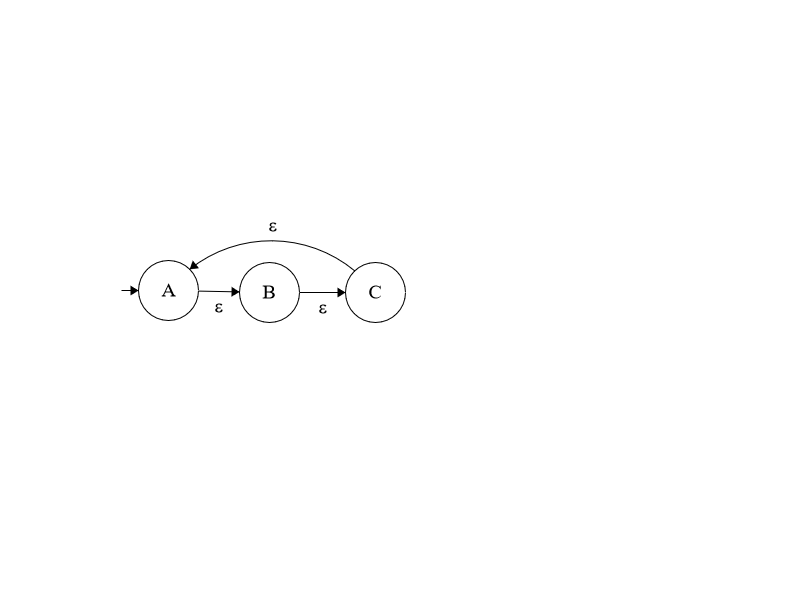
def update(self, message):

print("Sending SMS notification:", message)

* + State
    - To implement the object's behavior by its state

This method shows similarity to ‘Finite State Machine’s as state changes can be done with conditions and in different states different operations can be done. This is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. Different from Strategy method states may know about each other. Throughout the states different alterations can happen to the object, and this can be done multiple times.

*Ex*: This code represents shown finite state machine. (State changes and operations can be way more complex.)



class State:

def do\_action(self, context):

pass

class StateA(State):

def do\_action(self, context):

print("State A action")

context.state = StateB()

class StateB(State):

def do\_action(self, context):

print("State B action")

context.state = StateC()

class StateC(State):

def do\_action(self, context):

print("State C action")

context.state = StateA()

class Context:

def \_\_init\_\_(self):

self.state = StateA()

def request(self):

self.state.do\_action(self)

* + Strategy
    - To facilitate the algorithm's flexibility

This is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. Different from State method strategies almost never know about each other.

Example:

from typing import Union

# Strategies

class Strategy:

def do\_operation(self, num1, num2):

pass

class AddStrategy(Strategy):

def do\_operation(self, num1, num2):

return num1 + num2

class SubtractStrategy(Strategy):

def do\_operation(self, num1, num2):

return num1 - num2

class MultiplyStrategy(Strategy):

def do\_operation(self, num1, num2):

return num1 \* num2

# Interface for user

class Context:

def \_\_init\_\_(self, strategy: Union[AddStrategy, SubtractStrategy, MultiplyStrategy]):

self.strategy = strategy

def execute\_strategy(self, num1, num2):

return self.strategy.do\_operation(num1, num2)

* + Template
    - To provide high-level blueprints in base class

This is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Example: After initializing an AbstractClass, initiate template\_method and series of operations get done. Some operations get overridden by subclasses (in this example single) to unify different types of objects’ cases under AbstractClass.

from abc import ABC, abstractmethod

class AbstractClass(ABC):

def template\_method(self):

self.common\_operation1()

self.specialized\_operation1()

self.common\_operation2()

self.specialized\_operation2()

def common\_operation1(self):

print("AbstractClass: Performing common operation 1")

def common\_operation2(self):

print("AbstractClass: Performing common operation 2")

@abstractmethod

def specialized\_operation1(self):

pass

@abstractmethod

def specialized\_operation2(self):

pass

class ConcreteClass(AbstractClass):

def specialized\_operation1(self):

print("ConcreteClass: Performing specialized operation 1")

def specialized\_operation2(self):

print("ConcreteClass: Performing specialized operation 2")

* + Visitor
    - To define a new operation on a group of objects or hierarchy

This is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Example: There are elements and visitors. When elements (tried to) get visited by a visitor; elements can accept or reject this visitor, visitor may do different operations on this element.

# Define the Visitor interface

class Visitor:

def visit(self, element):

pass

# Define the elements that can be visited

class ElementA:

def accept(self, visitor):

visitor.visit(self)

class ElementB:

def accept(self, visitor):

visitor.visit(self)

# Define concrete visitors

class ConcreteVisitor1(Visitor):

def visit(self, element):

print("ConcreteVisitor1 visiting", element.\_\_class\_\_.\_\_name\_\_)

class ConcreteVisitor2(Visitor):

def visit(self, element):

print("ConcreteVisitor2 visiting", element.\_\_class\_\_.\_\_name\_\_)

For Further Reading:

* [Design Patterns](https://refactoring.guru/design-patterns)
* Design Patterns Elements of Reusable Object-Oriented Software by Gamma et al (Gang of Four)