Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

System Design Pattern

They help engineers build systems that are scalable, maintainable, reliable, and efficient.

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

1. Singleton Pattern

Ensures a class has only one instance and provides a global point of access to it.

```python
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Singleton(metaclass=SingletonMeta):
    def some_business_logic(self):
        pass

obj1 = Singleton()
obj2 = Singleton()
print(obj1 is obj2)  # True
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

2. Factory Method Pattern

Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

```python
class Button:
    def render(self):
        pass

class WindowsButton(Button):
    def render(self):
        return "Rendering Windows button"

class HTMLButton(Button):
    def render(self):
        return "Rendering HTML button"

class Dialog:
    def create_button(self):
        pass

class WindowsDialog(Dialog):
    def create_button(self):
        return WindowsButton()

class WebDialog(Dialog):
    def create_button(self):
        return HTMLButton()

dialog = WindowsDialog()
button = dialog.create_button()
print(button.render())  # Rendering Windows
button
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

3. Abstract Factory Pattern

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

```python
class GUIFactory:
    def create_button(self): pass
    def create_checkbox(self): pass

class WinFactory(GUIFactory):
    def create_button(self):
        return WinButton()
    def create_checkbox(self):
        return WinCheckbox()

class MacFactory(GUIFactory):
    def create_button(self):
        return MacButton()
    def create_checkbox(self):
        return MacCheckbox()

class WinButton:
    def click(self): return "Windows Button Clicked"

class WinCheckbox:
    def check(self): return "Windows Checkbox
Checked"
class MacButton:
    def click(self): return "Mac Button Clicked"

class MacCheckbox:
    def check(self): return "Mac Checkbox Checked"

factory = WinFactory()
button = factory.create_button()
print(button.click())  # Windows Button Clicked
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

4. Builder Pattern

Separates object construction from its representation, allowing the same construction process to create different representations.

```python
class Burger:
    def __init__(self):
        self.ingredients = []

    def add(self, ingredient):
        self.ingredients.append(ingredient)

    def __str__(self):
        return f"Burger with: {',
'.join(self.ingredients)}"
class BurgerBuilder:
    def __init__(self):
        self.burger = Burger()

    def add_bun(self):
        self.burger.add("bun")
        return self

    def add_patty(self):
        self.burger.add("patty")
        return self

    def add_lettuce(self):
        self.burger.add("lettuce")
        return self

    def build(self):
        return self.burger

builder = BurgerBuilder()
burger =
print(burger)
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

5. Prototype Pattern

Creates new objects by copying an existing object (deep copy or shallow copy).

```python
import copy

class Prototype:
    def clone(self):
        return copy.deepcopy(self)

class Person(Prototype):
    def __init__(self, name, skills):
        self.name = name
        self.skills = skills

    def __str__(self):
        return f"{self.name} with skills: {self.skills}"

original = Person("Alice", ["Python", "ML"])
clone = original.clone()
clone.name = "Bob"
clone.skills.append("AI")

print(original)  # Alice with skills: ['Python', 'ML',
print(clone)     # Bob with skills: ['Python', 'ML', 'AI']
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

6. Adapter Pattern

Allows incompatible interfaces to work together by wrapping one class with an interface expected by clients.

```python
class EuropeanPlug:
    def connect(self):
        return "220V from European plug"

class USPlugAdapter:
    def __init__(self, plug):
        self.plug = plug

    def connect(self):
        return f"Adapter converts -> {self.plug.connect()} to 110V"

euro_plug = EuropeanPlug()
adapter = USPlugAdapter(euro_plug)
print(adapter.connect())  # Adapter converts -> 220V from European plug to
110V
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

7. Bridge Pattern

Decouples abstraction from implementation, allowing them to vary independently.

```python
class RemoteControl:
    def __init__(self, device):
        self.device = device

    def turn_on(self):
        return self.device.enable()

class TV:
    def enable(self):
        return "TV turned ON"

class Radio:
    def enable(self):
        return "Radio turned ON"

remote = RemoteControl(TV())
print(remote.turn_on())  # TV turned
ON
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

8. Composite Pattern

Used to treat individual objects and compositions of objects uniformly. Ideal for tree-like structures.

```python
class Component:
    def operation(self):
        pass

class Leaf(Component):
    def __init__(self, name):
        self.name = name

    def operation(self):
        return f"Leaf {self.name}"

class Composite(Component):
    def __init__(self):
        self.children = []

    def add(self, component):
        self.children.append(component)

    def operation(self):
        results = [child.operation() for child in
self.children]
        return f"Branch({', '.join(results)})"

leaf1 = Leaf("A")
leaf2 = Leaf("B")
tree = Composite()
tree.add(leaf1)
tree.add(leaf2)
print(tree.operation())  # Branch(Leaf A, Leaf B)
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

9. Decorator Pattern

Adds functionality to an object dynamically without altering its structure.

```python
class Coffee:
    def cost(self):
        return 5

class MilkDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost() + 2

class SugarDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost() + 1

coffee = Coffee()
coffee_with_milk = MilkDecorator(coffee)
coffee_with_milk_sugar = SugarDecorator(coffee_with_milk)
print(coffee_with_milk_sugar.cost())  # 8
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

10. Facade Pattern

Simplifies complex subsystems by providing a unified interface.

```python
class CPU:
    def freeze(self): return "CPU freeze"
    def execute(self): return "CPU execute"

class Memory:
    def load(self): return "Memory load"

class HardDrive:
    def read(self): return "HardDrive read"

class Computer:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hard_drive = HardDrive()

    def start_computer(self):
        return [self.cpu.freeze(), self.memory.load(), self.hard_drive.read(),
self.cpu.execute()]
computer = Computer()
print(computer.start_computer())
# ['CPU freeze', 'Memory load', 'HardDrive read', 'CPU execute']
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

11. Flyweight Pattern

Reduces memory usage by sharing common data between similar objects.

```python
class TreeType:
    def __init__(self, name, color, texture):
        self.name = name
        self.color = color
        self.texture = texture

    def draw(self, x, y):
        return f"Drawing {self.name} at ({x}, {y})"

class TreeFactory:
    _tree_types = {}

    @staticmethod
    def get_tree_type(name, color, texture):
        key = (name, color, texture)
        if key not in TreeFactory._tree_types:
            TreeFactory._tree_types[key] = TreeType(name, color,
texture)return TreeFactory._tree_types[key]

tree1 = TreeFactory.get_tree_type("Oak", "Green", "Rough")
tree2 = TreeFactory.get_tree_type("Oak", "Green", "Rough")
print(tree1 is tree2)  # True
print(tree1.draw(10, 20))  # Drawing Oak at (10, 20)
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

12. Proxy Pattern

Acts as a placeholder for another object to control access, reduce cost, or add extra logic.

```python
class RealSubject:
    def request(self):
        return "RealSubject handling request"

class Proxy:
    def __init__(self, real_subject):
        self.real_subject = real_subject

    def request(self):
        print("Proxy: Logging access before forwarding to RealSubject")
        return self.real_subject.request()

real = RealSubject()
proxy = Proxy(real)
print(proxy.request())  # Logs and then returns: RealSubject handling request
```

Hasnain Ahmed Shaikh
Software Development
Engineer （ SDE ）

12. Proxy Pattern

Acts as a placeholder for another object to control access, reduce cost, or add extra logic.

```python
class RealSubject:
    def request(self):
        return "RealSubject handling request"

class Proxy:
    def __init__(self, real_subject):
        self.real_subject = real_subject

    def request(self):
        print("Proxy: Logging access before forwarding to RealSubject")
        return self.real_subject.request()

real = RealSubject()
proxy = Proxy(real)
print(proxy.request())  # Logs and then returns: RealSubject handling
request
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

13. Chain of Responsibility Pattern

Passes a request along a chain of handlers until one handles it.

```python
class Handler:
    def __init__(self, successor=None):
        self.successor = successor

    def handle(self, request):
        if self.successor:
            return self.successor.handle(request)
        return None

class AuthHandler(Handler):
    def handle(self, request):
        if request == "auth":
            return "Handled by AuthHandler"
        return super().handle(request)

class DataHandler(Handler):
    def handle(self, request):
        if request == "data":
            return "Handled by DataHandler"
        return super().handle(request)

chain = AuthHandler(DataHandler())
print(chain.handle("data"))  # Handled by DataHandler
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

14. Command Pattern

Encapsulates a request as an object, separating the command from the receiver.

```python
class Light:
    def on(self):
        return "Light is ON"

    def off(self):
        return "Light is OFF"

class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        return self.light.on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        return self.light.off()

class RemoteControl:
    def submit(self, command):
        return command.execute()

light = Light()
remote = RemoteControl()
print(remote.submit(LightOnCommand(light)))   # Light is ON
print(remote.submit(LightOffCommand(light)))  # Light is
OFF
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

15. Interpreter Pattern

Interprets a simple grammar or expression language.

```python
class Context:
    def __init__(self, text):
        self.text = text

class Expression:
    def interpret(self):
        pass

class Number(Expression):
    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

class Plus(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() +
self.right.interpret()
expr = Plus(Number(5), Number(3))
print(expr.interpret())  # 8
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

16. Iterator Pattern

Provides a way to access the elements of an aggregate object without exposing its underlying representation.

```python
class MyIterator:
    def __init__(self, collection):
        self.collection = collection
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.collection):
            val = self.collection[self.index]
            self.index += 1
            return val
        raise StopIteration

class Numbers:
    def __init__(self):
        self.nums = [1, 2, 3, 4]

    def __iter__(self):
        return MyIterator(self.nums)

numbers = Numbers()
for num in numbers:
    print(num)
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

17. Mediator Pattern

Defines an object that encapsulates how a set of objects interact.

```python
class ChatRoom:
    def show_message(self, user, message):
        print(f"[{user}] says: {message}")

class User:
    def __init__(self, name, chatroom):
        self.name = name
        self.chatroom = chatroom

    def send(self, message):
        self.chatroom.show_message(self.name,
message)
room = ChatRoom()
alice = User("Alice", room)
bob = User("Bob", room)

alice.send("Hi Bob!")
bob.send("Hello Alice!")
```

Hasnain Ahmed Shaikh
Software Development Engineer (SDE)

18. Memento Pattern

Captures and externalizes an object's internal state so it can be restored later without violating encapsulation.

```python
class Memento:
    def __init__(self, state):
        self._state = state

    def get_state(self):
        return self._state

class Originator:
    def __init__(self):
        self._state = ""

    def set_state(self, state):
        self._state = state

    def save(self):
        return Memento(self._state)

    def restore(self, memento):
        self._state =
memento.get_state()
originator = Originator()
originator.set_state("Version 1")
memento = originator.save()

originator.set_state("Version 2")
print(originator._state)  # Version 2

originator.restore(memento)
print(originator._state)  # Version 1
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

19. Observer Pattern

Defines a one-to-many dependency between objects, so when one changes state, all dependents are notified.

```python
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def notify(self, message):
        for obs in self._observers:
            obs.update(message)

class Observer:
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received:
{message}")
subject = Subject()
obs1 = Observer("Observer1")
obs2 = Observer("Observer2")

subject.attach(obs1)
subject.attach(obs2)

subject.notify("Event happened!")
```

Hasnain Ahmed Shaikh
Software Development Engineer (SDE)

Here are the final 4 classic design patterns, explained with concise Python examples:

20. State Pattern

Allows an object to alter its behavior when its internal state changes.

```python
class State:
    def handle(self):
        pass

class ConcreteStateA(State):
    def handle(self):
        return "State A handling"

class ConcreteStateB(State):
    def handle(self):
        return "State B handling"

class Context:
    def __init__(self, state):
        self._state = state

    def set_state(self, state):
        self._state = state

    def request(self):
        return self._state.handle()

context = Context(ConcreteStateA())
print(context.request())  # State A handling
context.set_state(ConcreteStateB())
print(context.request())  # State B handling
```

Hasnain Ahmed Shaikh
Software Development Engineer (SDE)

21. Strategy Pattern

Defines a family of algorithms and makes them interchangeable.

```python
class Strategy:
    def execute(self, a, b):
        pass

class AddStrategy(Strategy):
    def execute(self, a, b):
        return a + b

class MultiplyStrategy(Strategy):
    def execute(self, a, b):
        return a * b

class Context:
    def __init__(self, strategy):
        self._strategy = strategy

    def set_strategy(self, strategy):
        self._strategy = strategy

    def execute_strategy(self, a, b):
        return self._strategy.execute(a,
b)
context = Context(AddStrategy())
print(context.execute_strategy(5, 3))  # 8

context.set_strategy(MultiplyStrategy())
print(context.execute_strategy(5, 3))  #
15
```

Hasnain Ahmed Shaikh
Software Development Engineer (SDE)

22. Template Method Pattern

Defines the skeleton of an algorithm, letting subclasses fill in specific steps.

```python
class Game:
    def play(self):
        self.start()
        self.play_turn()
        self.end()

    def start(self):
        pass

    def play_turn(self):
        pass

    def end(self):
        pass

class Chess(Game):
    def start(self):
        print("Chess game started")

    def play_turn(self):
        print("Playing a turn in chess")
    def end(self):
        print("Chess game ended")

game = Chess()
game.play()
```

Hasnain Ahmed Shaikh
Software Development
Engineer (SDE)

23. Visitor Pattern

Adds operations to existing class hierarchies without modifying them.

```python
class Visitor:
    def visit_concrete_element_a(self, element):
        pass

    def visit_concrete_element_b(self, element):
        pass

class ConcreteVisitor(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"Visitor works on
{element.feature()}")
    def visit_concrete_element_b(self, element):
        print(f"Visitor analyzes {element.detail()}")

class Element:
    def accept(self, visitor):
        pass

class ConcreteElementA(Element):
    def accept(self, visitor):
        visitor.visit_concrete_element_a(self)

    def feature(self):
        return "Element A"

class ConcreteElementB(Element):
    def accept(self, visitor):
        visitor.visit_concrete_element_b(self)

    def detail(self):
        return "Element B"

elements = [ConcreteElementA(), ConcreteElementB()]
visitor = ConcreteVisitor()
for e in elements:
    e.accept(visitor)
```

Follow

To get more Knowledge about
Developing a Software

Hasnain Ahmed Shaikh

Software Development Engineer （ SDE ）