1. Class Diagram (Chapter-6)

A class diagram shows the static structure of a system, including classes, their attributes, methods, and relationships.

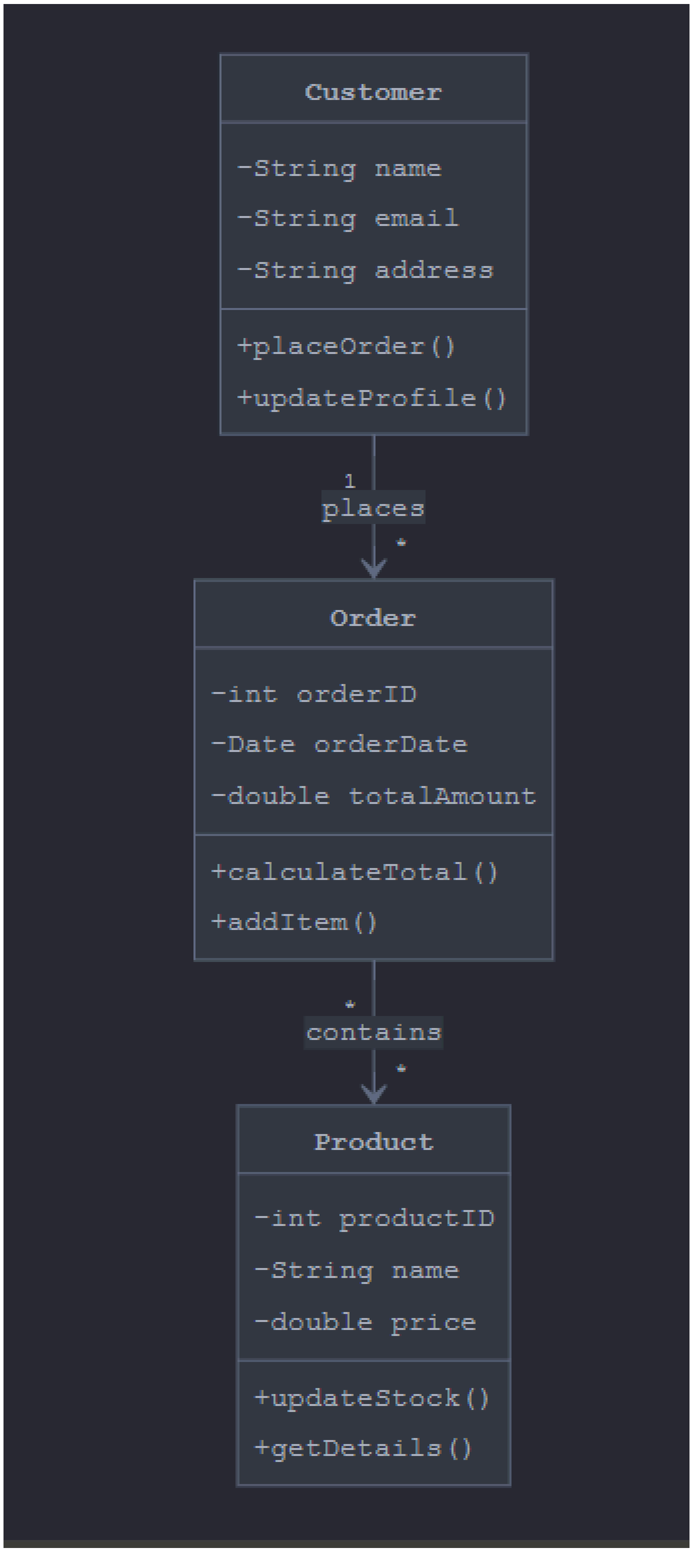
Key Components:

- Class name (top compartment)
- Attributes (middle compartment)
- Methods (bottom compartment)
- Relationships between classes (lines with different symbols)

Steps to draw a Class Diagram in exam:

1. Identify all the classes from the scenario
2. For each class, list attributes and methods
3. Draw class boxes with three compartments
4. Identify relationships between classes:
   - Association (simple line)
   - Inheritance (arrow with hollow triangle)
   - Aggregation (line with hollow diamond)
   - Composition (line with filled diamond)
   - Dependency (dashed line with arrow)
5. Add multiplicity (1, 0..*, 1..*, etc.) at line ends
6. Add relationship labels if needed

Let me create a visualization to help you understand better:


Customer
-String name
-String email
-String address
+placeOrder()
+updateProfile()
1
places
*
Order
-int orderID
-Date orderDate
-double totalAmount
+calculateTotal()
+addItem()
*
contains
*
Product
-int productID
-String name
-double price
+updateStock()
+getDetails()

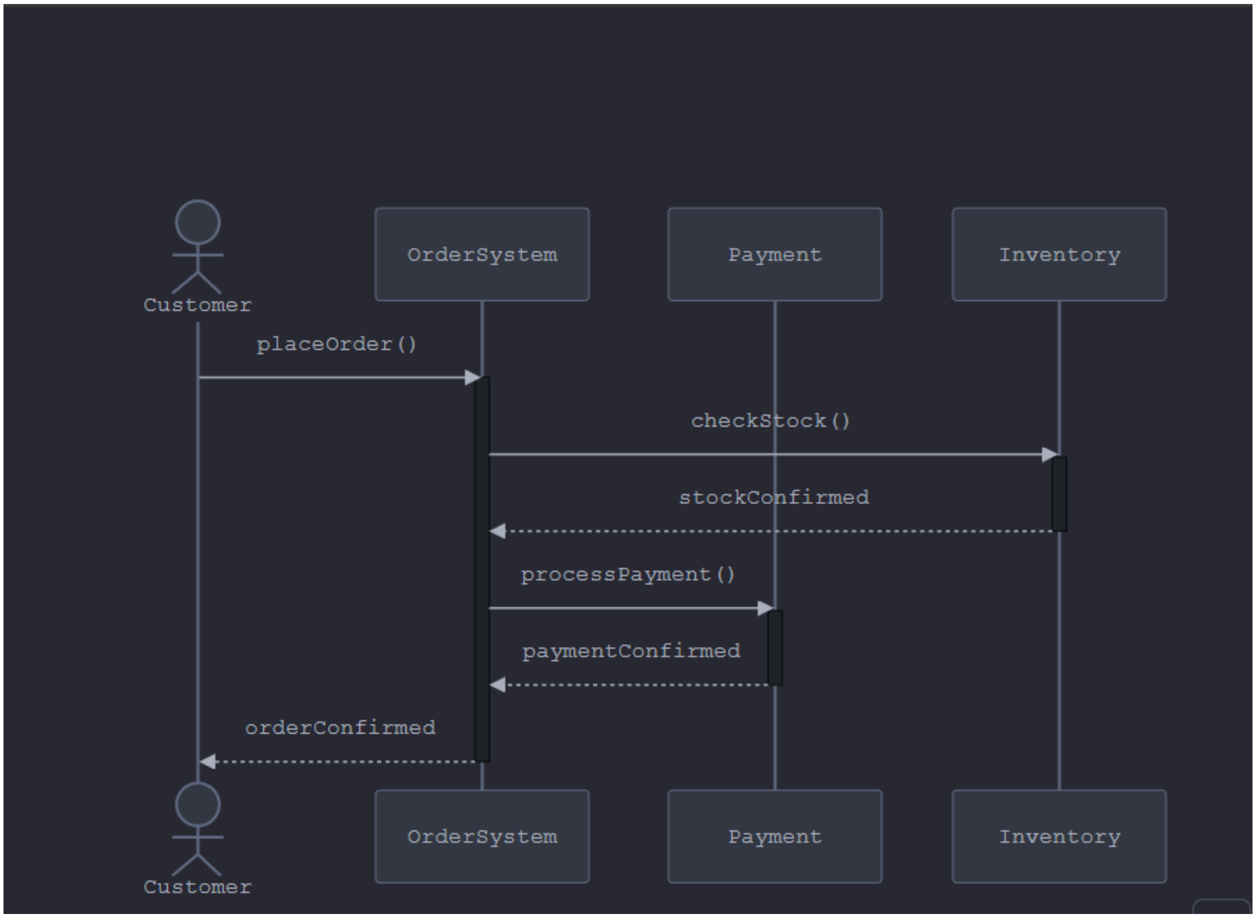
2. Sequence Diagram

A sequence diagram shows object interactions arranged in time sequence, illustrating how objects communicate with each other.

Steps to draw a Sequence Diagram:

1. Identify objects/actors involved in the scenario
2. Draw lifelines (vertical dashed lines) for each object
3. Draw activation boxes (rectangles) when object is active
4. Add messages between objects (arrows):
   - Synchronous (filled arrow)
   - Asynchronous (open arrow)
   - Return (dashed arrow)
5. Add time constraints or conditions if needed
6. Show object creation/destruction if applicable

Let me create a sequence diagram example:



3. Use Case Diagram

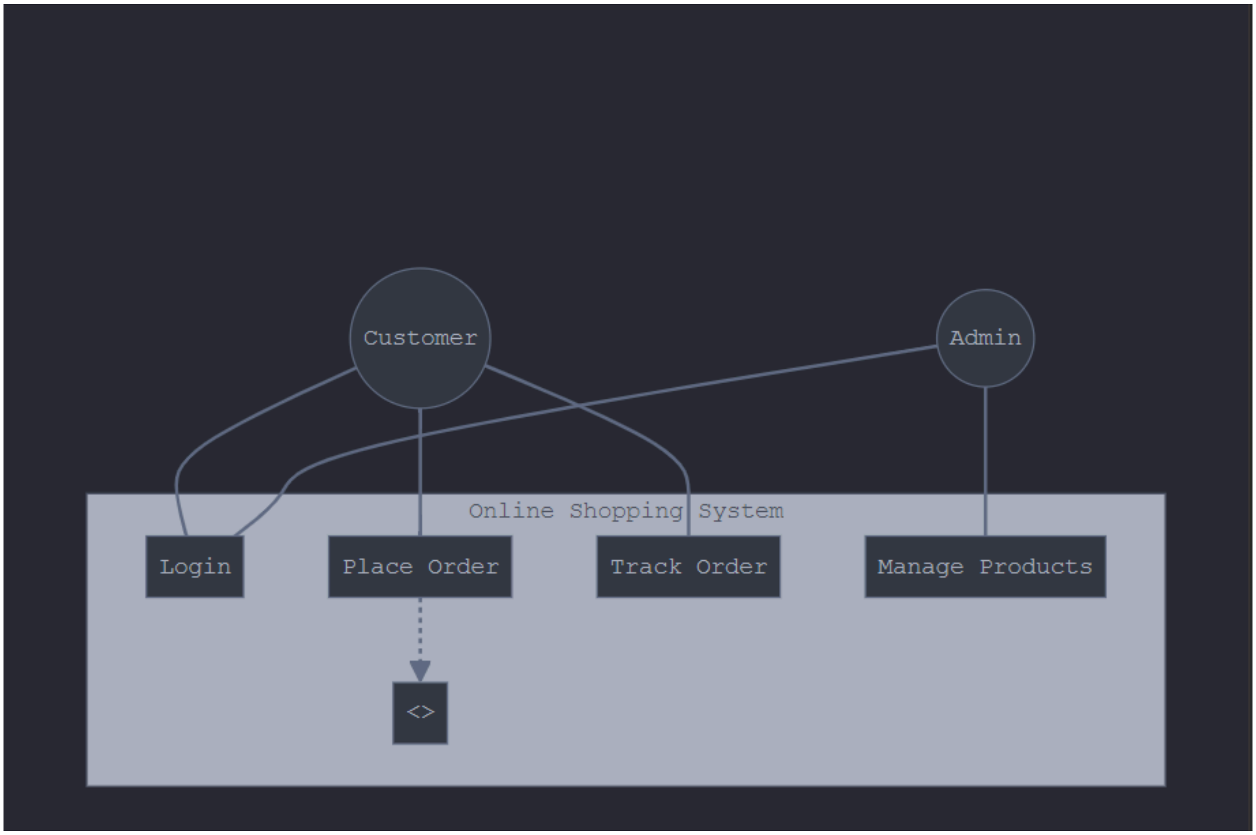
A use case diagram shows interactions between users (actors) and the system, representing functional requirements.

Steps to draw a Use Case Diagram:

1. Identify actors (stick figures)
   - Primary actors on left
   - Secondary actors on right
2. Identify use cases (ovals with text)
3. Draw system boundary (rectangle)
4. Connect actors to use cases with lines
5. Identify relationships between use cases:

- Include (dashed arrow with <<include>>)
- Extend (dashed arrow with <<extend>>)
- Generalization (solid line with arrow)

Here's a use case diagram example:



Agile Model and Scrum (Chapter-7)

2. **Agile Methodology**

- **Agile Philosophy**: Agile emerged as a solution to the problems of the waterfall model. It focuses on rapid, iterative development and deployment of software. Instead of building the entire application at once, Agile breaks the application into smaller, manageable parts (microservices).

- **Iterative Development**: Agile works in iterations (cycles). In each iteration, a specific task or microservice is developed, tested, and deployed. This allows for continuous improvement and rapid delivery of functional software.

3. **Key Terms of Agile**

- **People over Processes and Tools**: Agile prioritizes human interaction and collaboration over rigid processes and tools.
- **Working Software over Documentation**: The focus is on delivering functional software rather than extensive documentation.
- **Customer Collaboration over Contracts**: Agile relies heavily on customer feedback and collaboration, rather than sticking to fixed contracts.

4. **Principles of Agile**

➔ - Agile emphasizes customer satisfaction, welcoming changing requirements, and delivering working software frequently.

➔ - It promotes frequent interaction with stakeholders, motivated individuals, and face-to-face communication.

➔ - Agile also focuses on technical excellence, simplicity, self-organizing teams, and continuous reflection and adjustment.

5. **Advantages of Agile**

1. Agile ensures continuous delivery of software.

2. Frequent updates and collaboration lead to higher stakeholder satisfaction.
3. Agile welcomes changes at any stage of development.
4. Regular communication ensures that the team is aligned and any issues are addressed promptly.

6. **Implementing Agile**

- SCRUM,
- Extreme Programming (XP),
- LEAN
- KANBAN
- CRYSTAL

7. **SCRUM Framework**

- **Iterative Approach**: SCRUM is an iterative framework that involves planning, building, testing, and reviewing in cycles.
- **Roles in SCRUM**:
  - **Product Owner**: Responsible for the product backlog and deployment. They don’t need to be technical.
  - **Scrum Master**: Acts as a team leader, handling day-to-day tasks and ensuring the team follows Agile practices.
  - **Team**: Includes developers, testers, and other roles involved in the development process.

8. **Development in SCRUM**

- **Product Backlog**: A list of tasks to be completed, prioritized by the product owner and scrum master.

- **User Stories**: Each task in the backlog is called a user story. These stories are prioritized and worked on in iterations (sprints).

- **Sprint Backlog**: The development team works on the tasks in the sprint backlog. Sprint planning involves setting goals for the sprint.

- **Daily Scrum**: A 15-minute stand-up meeting where the team discusses progress, plans for the day, and any blockers.

- **Sprint Reviews**: At the end of each sprint (usually 2-4 weeks), the team demonstrates the completed work and reviews the progress before moving on to the next sprint.

- Agile, particularly through the SCRUM framework, allows for rapid, iterative development, continuous feedback, and adaptability to changing requirements. It emphasizes collaboration, simplicity, and delivering functional software frequently.

This detailed breakdown of the slide content highlights the key concepts of Agile and SCRUM, explaining how they address the limitations of traditional development models like Waterfall and promote a more flexible, customer-centric approach to software development.

Chapter-9

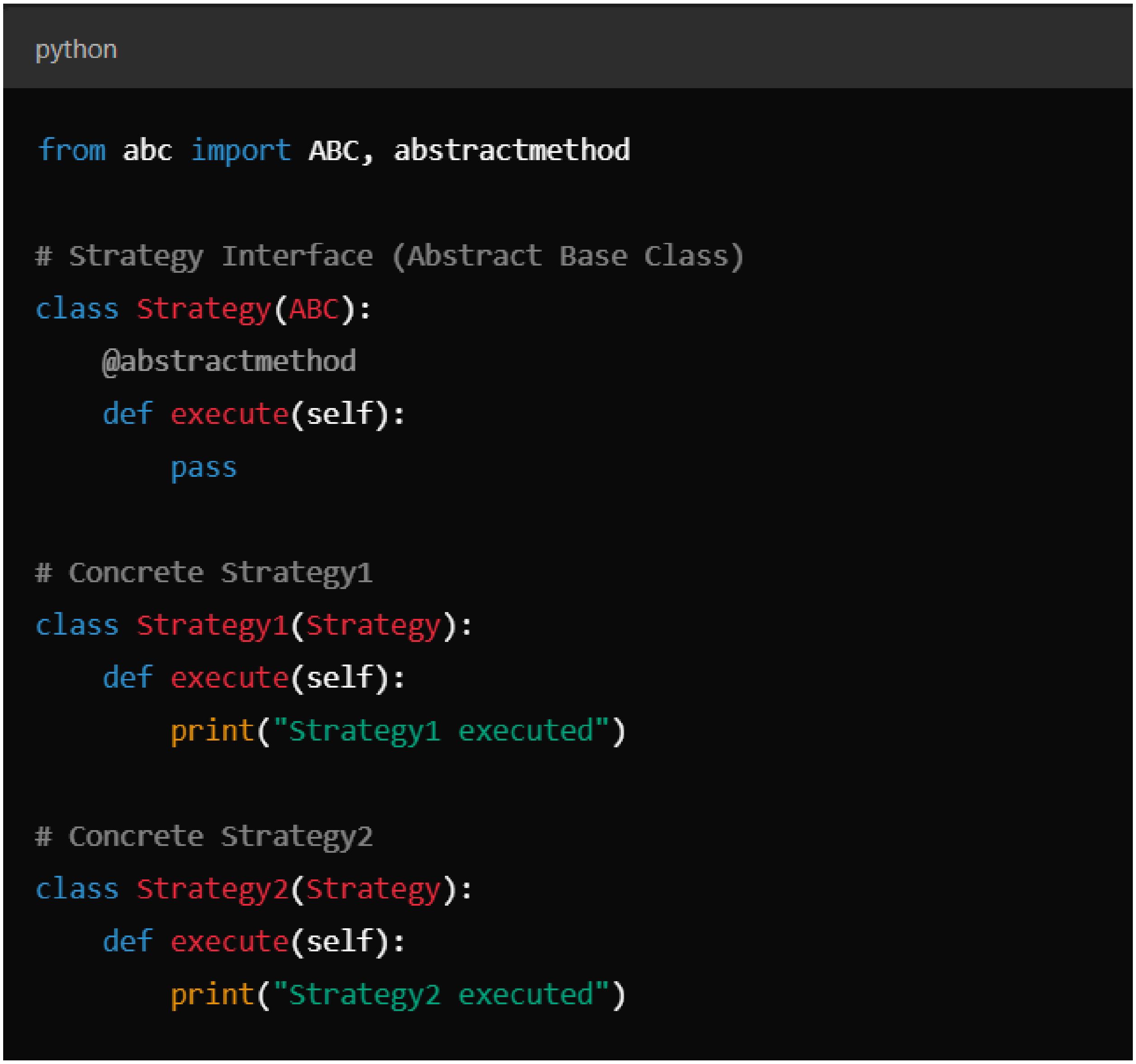
Explanation of Strategy Pattern Components

1. **Context**
    - The main class that interacts with the client.
    - Holds a reference to a Strategy object and delegates execution.
    - Can switch between different strategies at runtime.
2. **Strategy Interface**

    - Serves as a common contract for all strategies.
    - Declares a method that all strategies must implement.
    - Ensures consistency and allows the Context to use any strategy interchangeably.

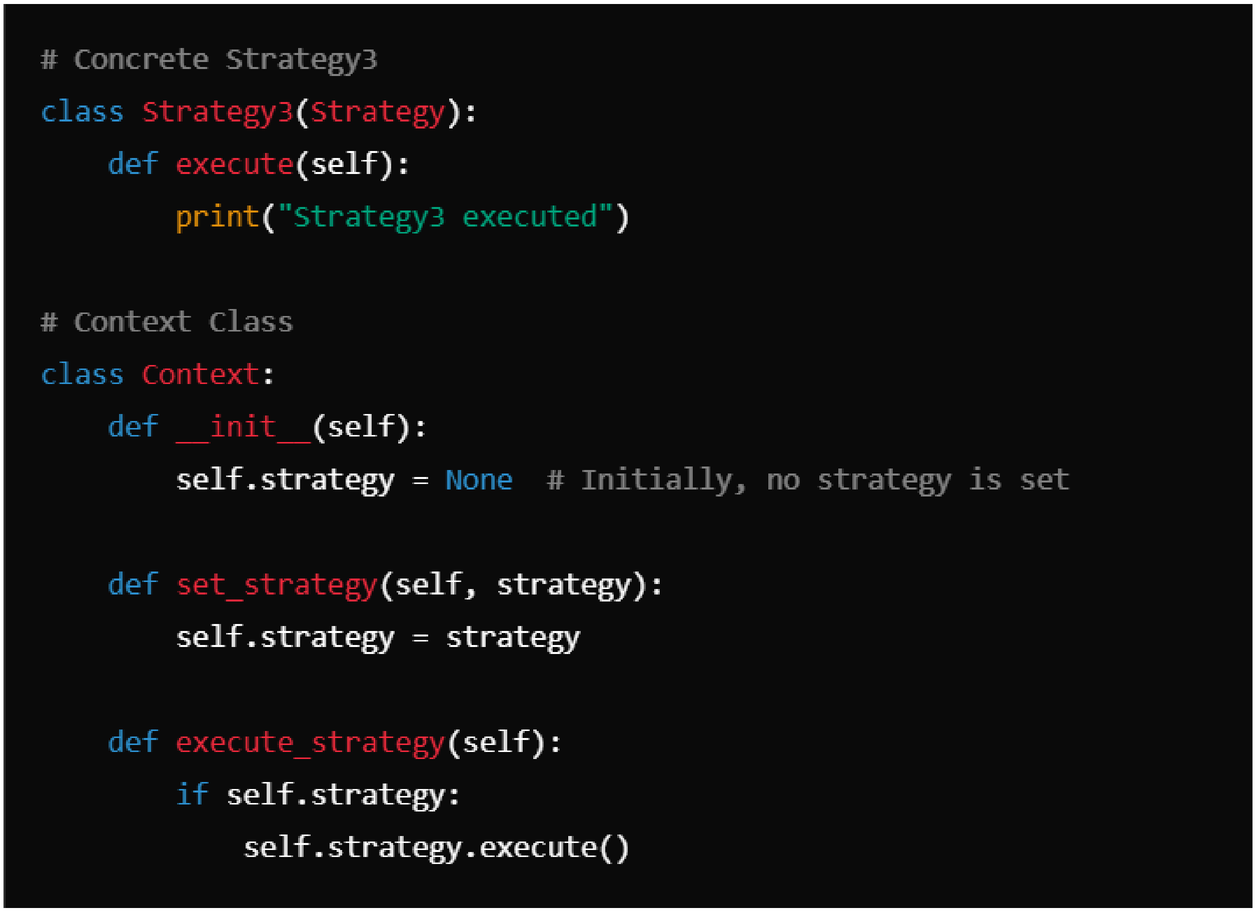
3. **Concrete Strategies**
    - Provide specific implementations of the algorithm.
    - Allow adding new behaviors without modifying the Context.

Code:

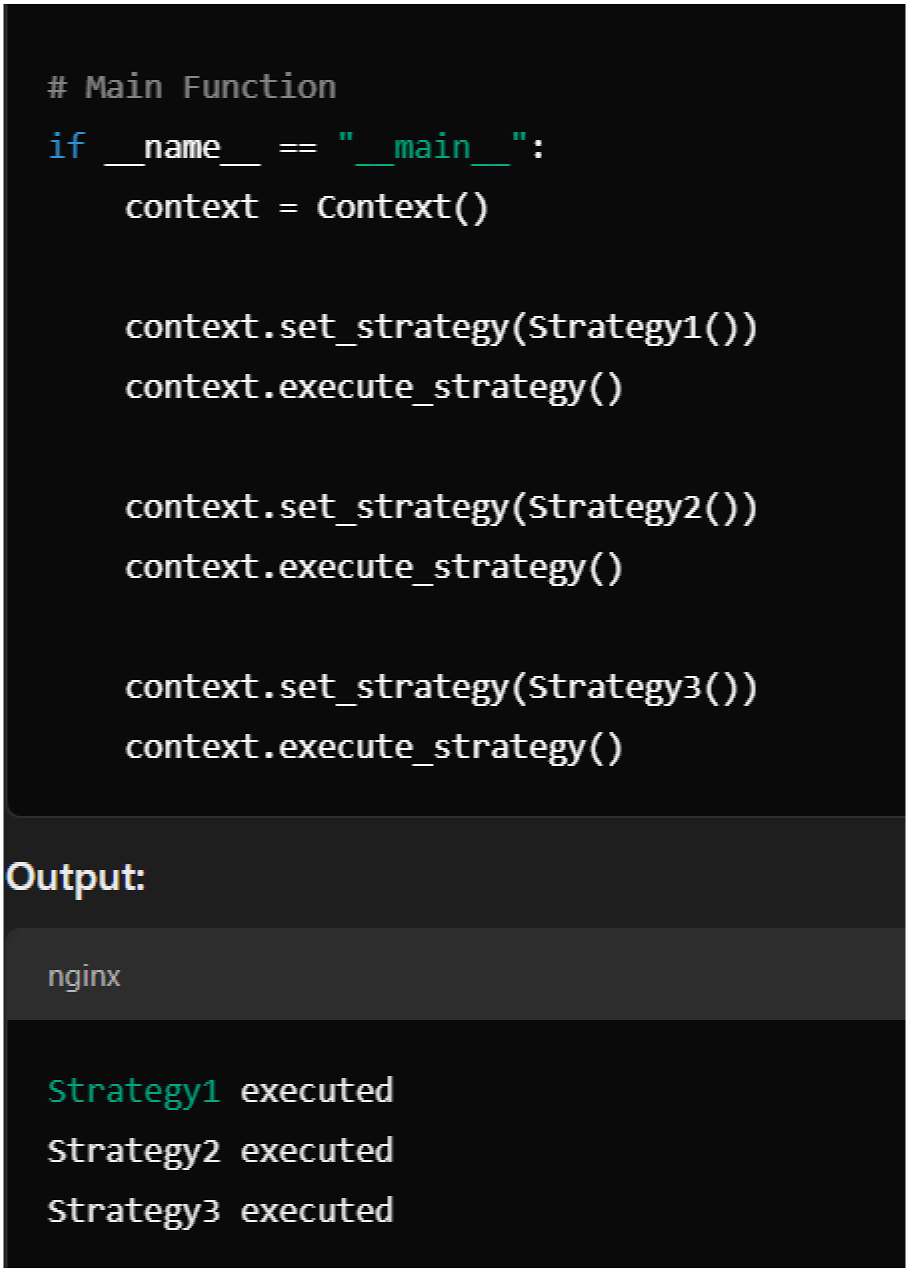


python

```python
from abc import ABC, abstractmethod

# Strategy Interface (Abstract Base Class)
class Strategy(ABC):
    @abstractmethod
    def execute(self):
        pass

# Concrete Strategy1
class Strategy1(Strategy):
    def execute(self):
        print("Strategy1 executed")

# Concrete Strategy2
class Strategy2(Strategy):
    def execute(self):
        print("Strategy2 executed")
```



```python
# Concrete Strategy3
class Strategy3(Strategy):
    def execute(self):
        print("Strategy3 executed")

# Context Class
class Context:
    def __init__(self):
        self.strategy = None  # Initially, no strategy is set

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

    def execute_strategy(self):
        if self.strategy:
            self.strategy.execute()
```



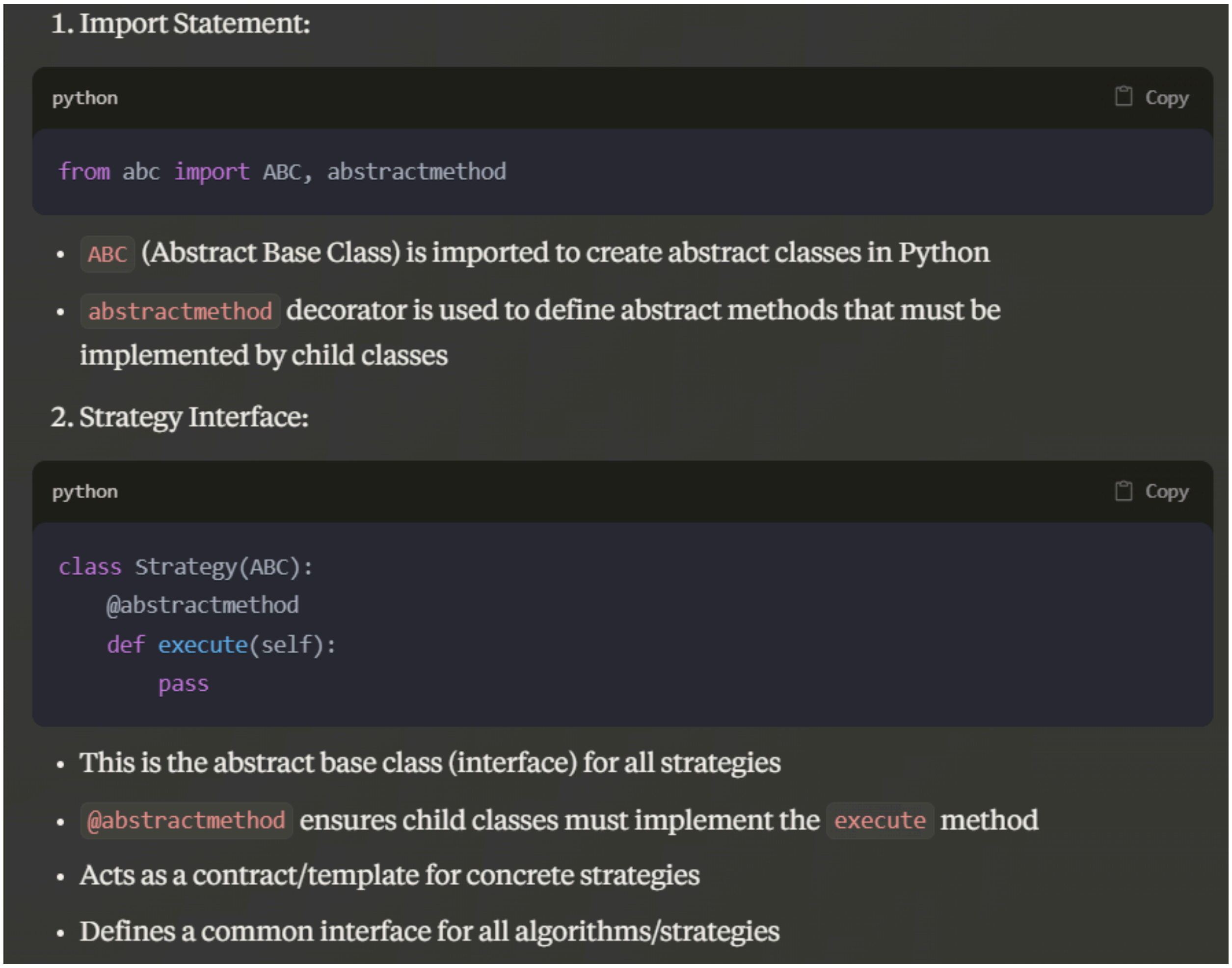
```
# Main Function
if __name__ == "__main__":
    context = Context()

    context.set_strategy(Strategy1())
    context.execute_strategy()

    context.set_strategy(Strategy2())
    context.execute_strategy()

    context.set_strategy(Strategy3())
    context.execute_strategy()
```

Output:

nginx

```
Strategy1 executed
Strategy2 executed
Strategy3 executed
```

Explanation:



1. Import Statement:

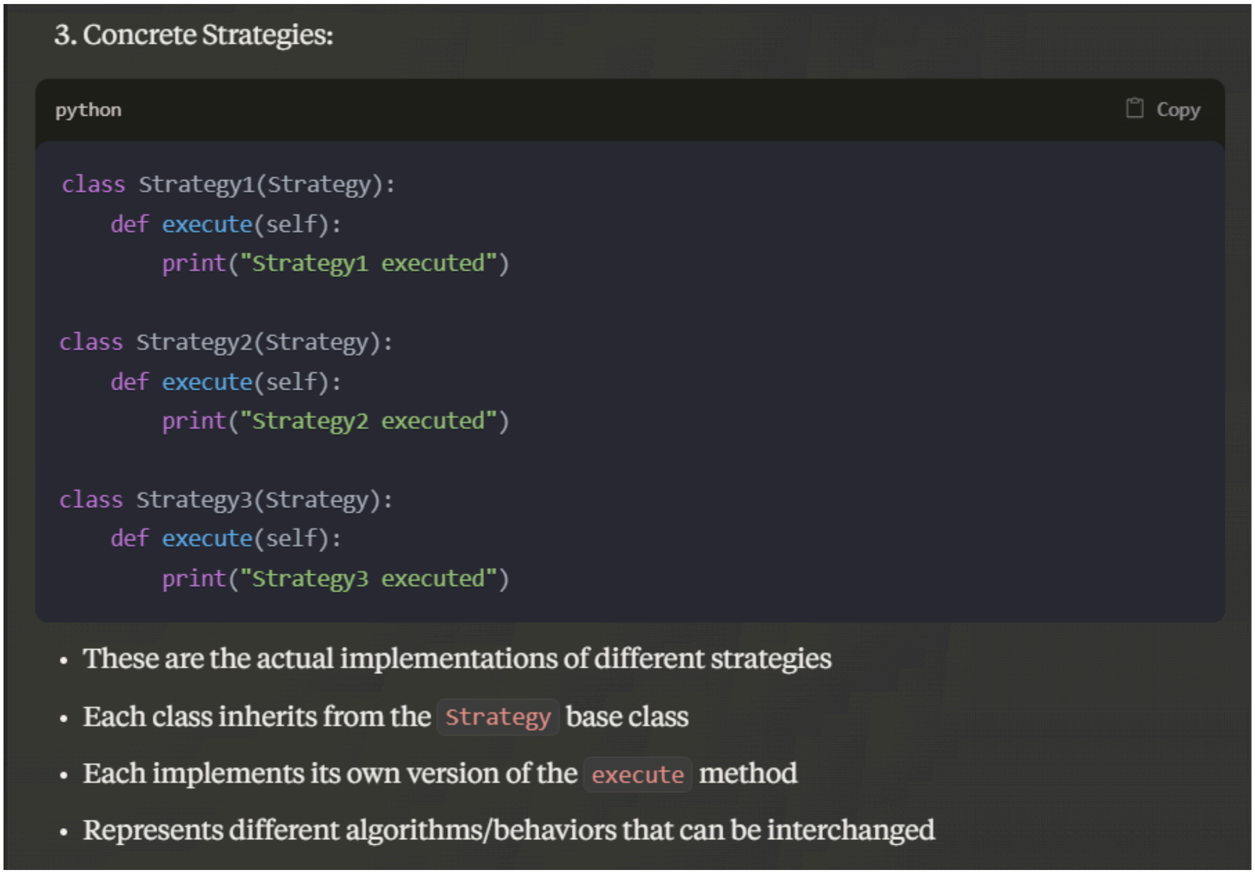
```python
from abc import ABC, abstractmethod
```

- `ABC` (Abstract Base Class) is imported to create abstract classes in Python
- `abstractmethod` decorator is used to define abstract methods that must be implemented by child classes

2. Strategy Interface:

```python
class Strategy(ABC):
    @abstractmethod
    def execute(self):
        pass
```

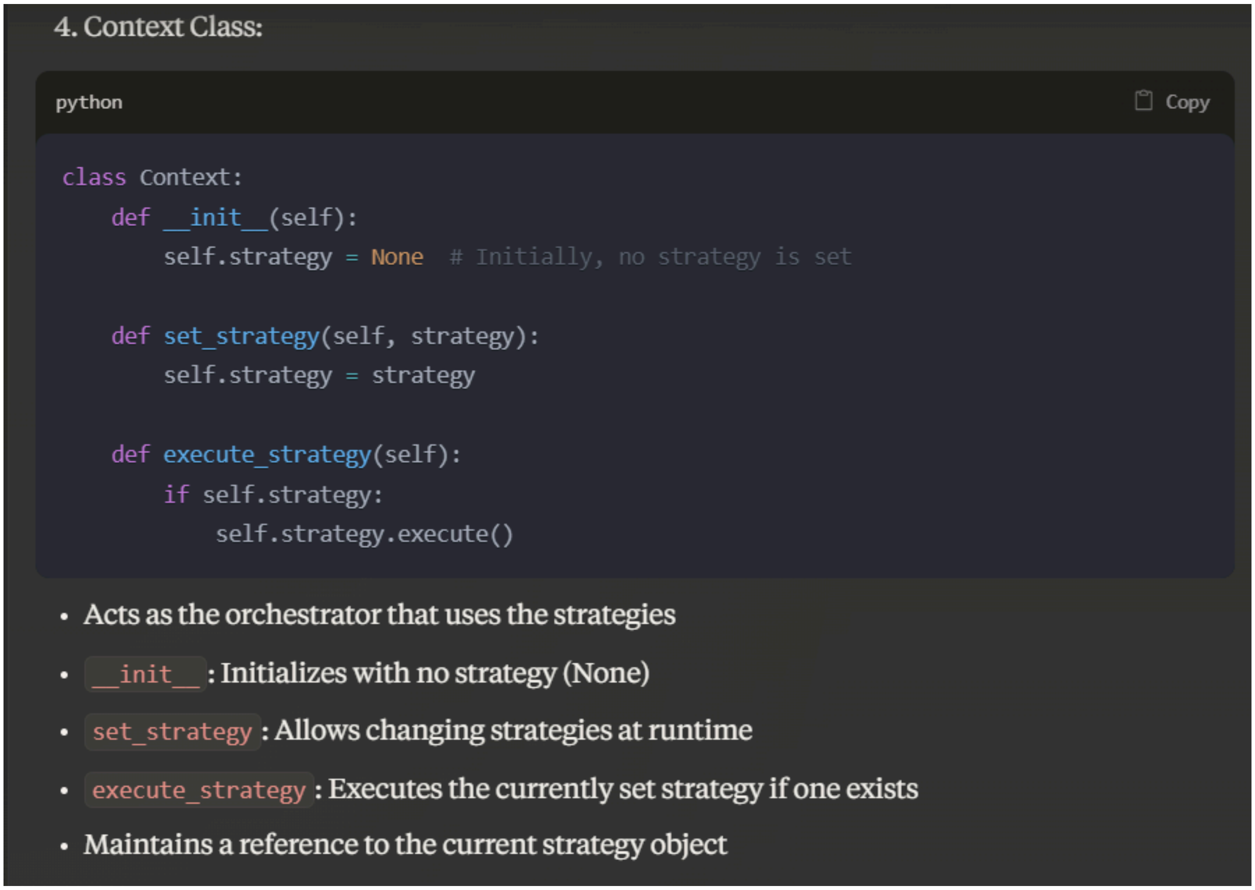
- This is the abstract base class (interface) for all strategies
- `@abstractmethod` ensures child classes must implement the `execute` method
- Acts as a contract/template for concrete strategies
- Defines a common interface for all algorithms/strategies



3. Concrete Strategies:

```python
class Strategy1(Strategy):
    def execute(self):
        print("Strategy1 executed")

class Strategy2(Strategy):
    def execute(self):
        print("Strategy2 executed")

class Strategy3(Strategy):
    def execute(self):
        print("Strategy3 executed")
```

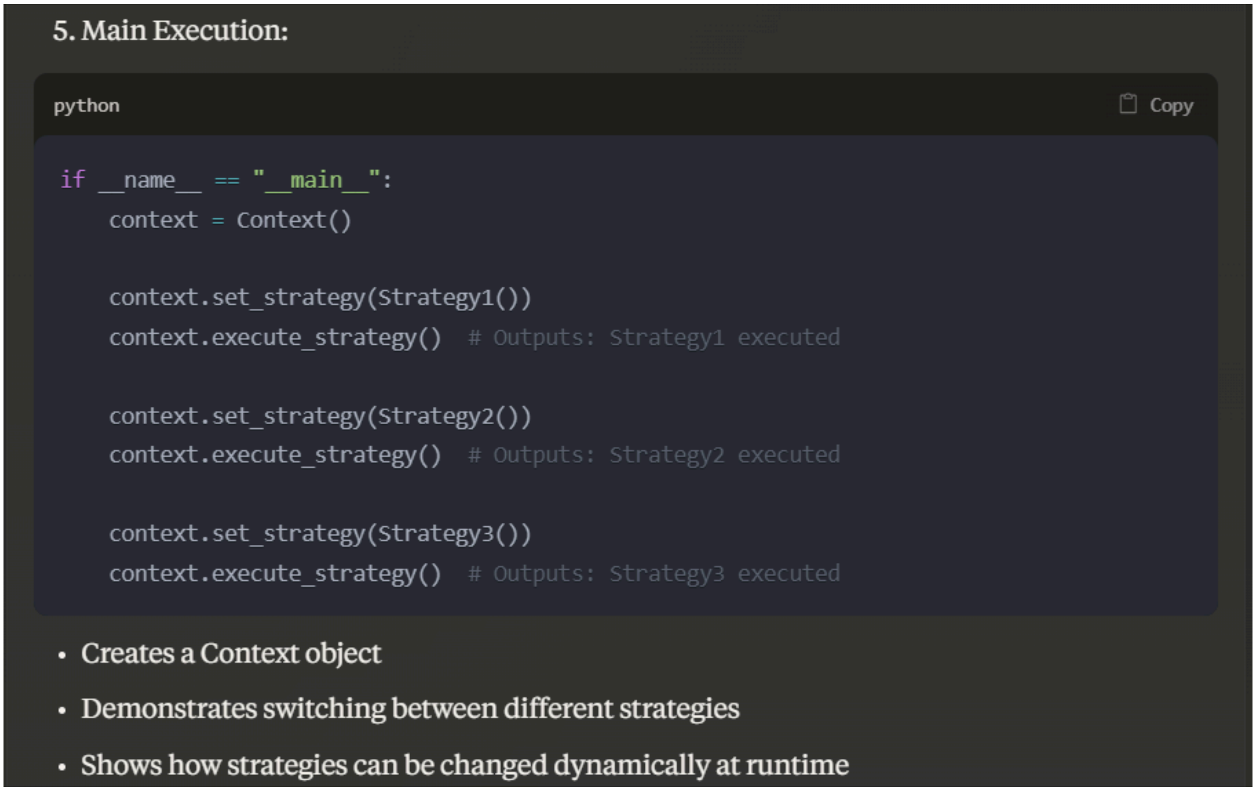
- These are the actual implementations of different strategies
- Each class inherits from the `Strategy` base class
- Each implements its own version of the `execute` method
- Represents different algorithms/behaviors that can be interchanged



4. Context Class:

```python
class Context:
    def __init__(self):
        self.strategy = None  # Initially, no strategy is set

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

    def execute_strategy(self):
        if self.strategy:
            self.strategy.execute()
```

- Acts as the orchestrator that uses the strategies
- `__init__`: Initializes with no strategy (None)
- `set_strategy`: Allows changing strategies at runtime
- `execute_strategy`: Executes the currently set strategy if one exists
- Maintains a reference to the current strategy object

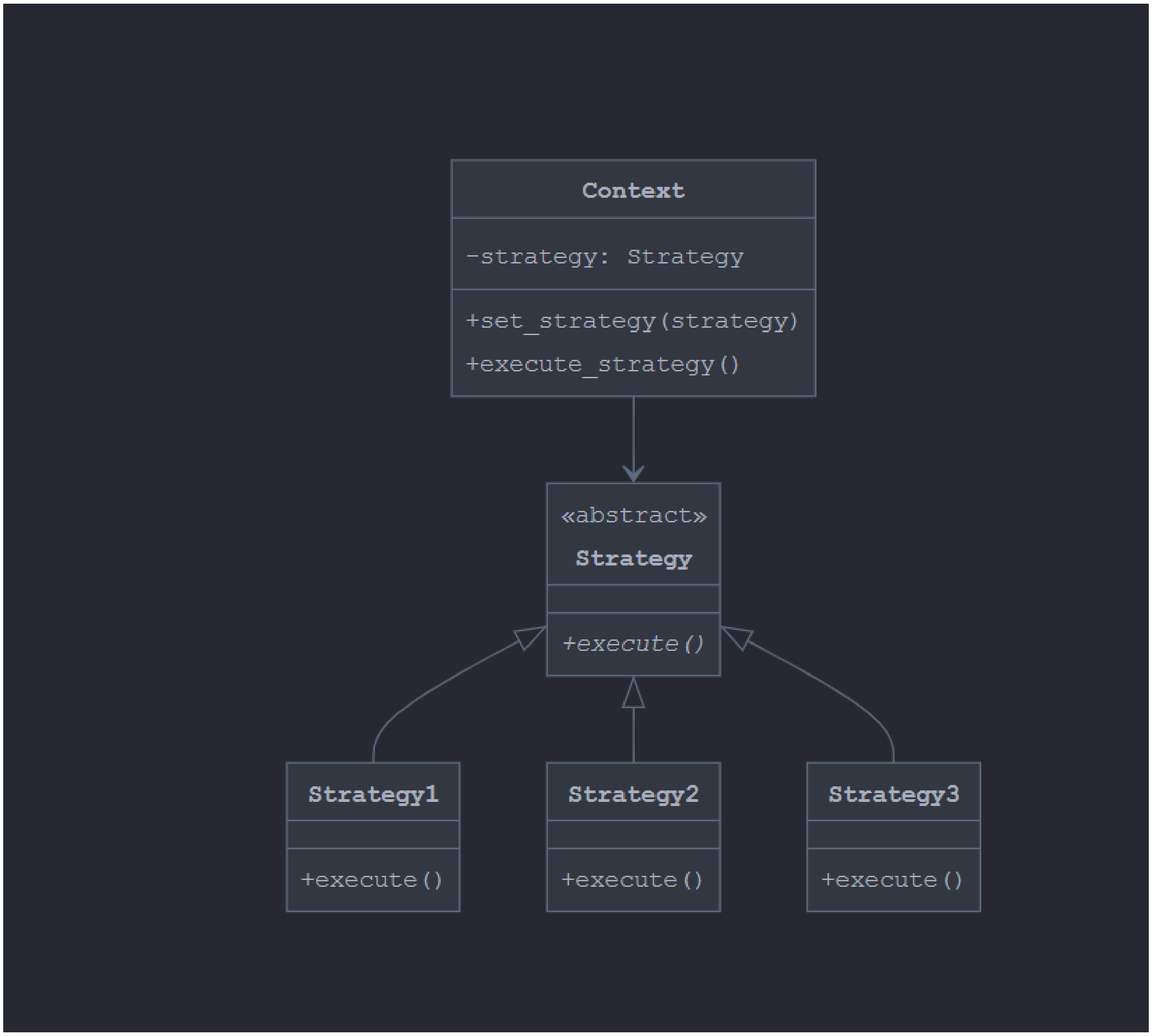


5. Main Execution:

```python
if __name__ == "__main__":
    context = Context()

    context.set_strategy(Strategy1())
    context.execute_strategy()  # Outputs: Strategy1 executed

    context.set_strategy(Strategy2())
    context.execute_strategy()  # Outputs: Strategy2 executed

    context.set_strategy(Strategy3())
    context.execute_strategy()  # Outputs: Strategy3 executed
```

- Creates a Context object
- Demonstrates switching between different strategies
- Shows how strategies can be changed dynamically at runtime

Diagram for visual explain:



Strategy Pattern Components:

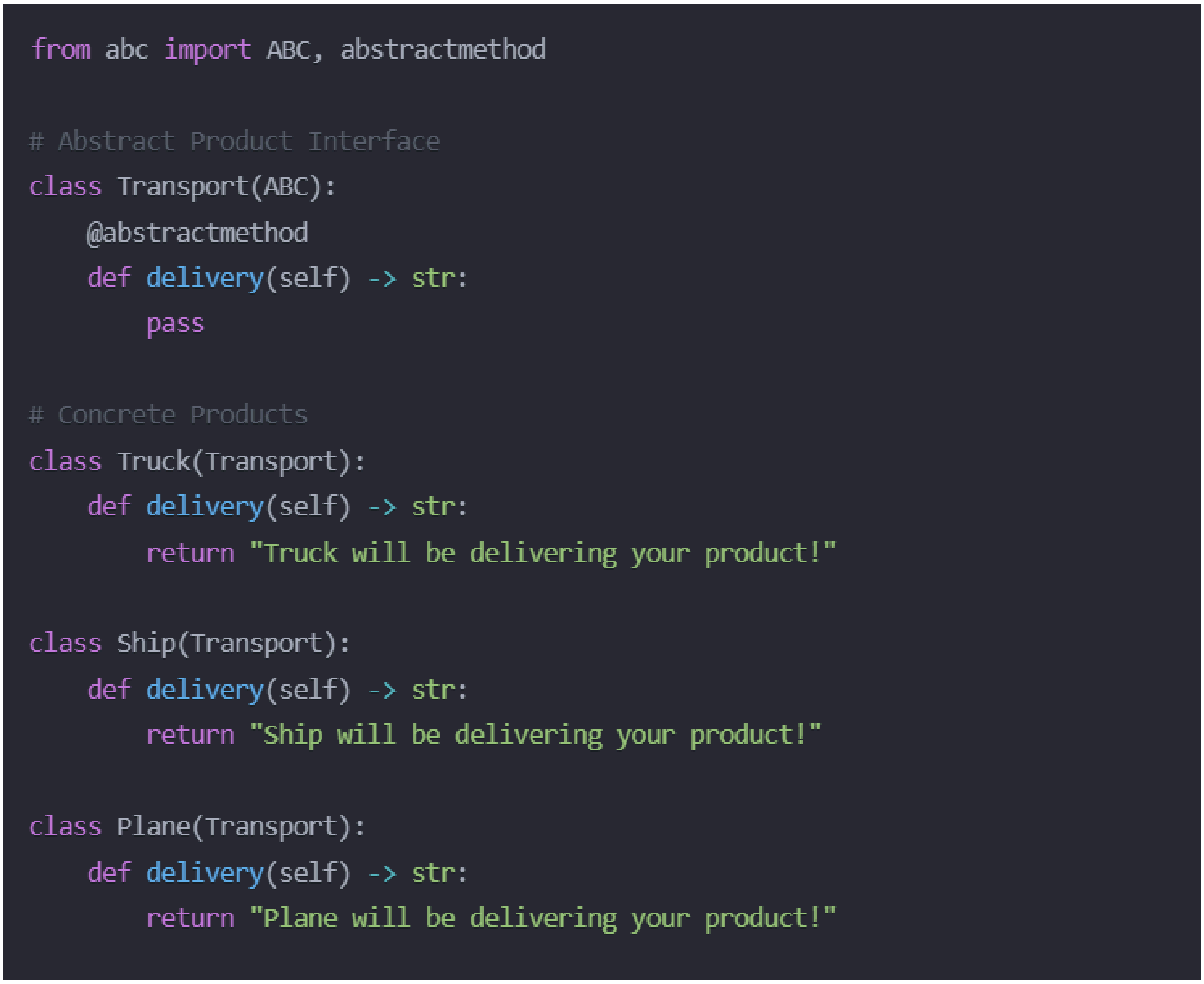
1. Strategy (Abstract Class):
   - Defines the interface common to all supported algorithms
   - Declares the `execute` method that concrete strategies must implement
2. Concrete Strategies (Strategy1, Strategy2, Strategy3):
   - Implement different variations of an algorithm

    - Must follow the interface defined by the Strategy abstract class
    - Can be swapped in and out without changing the client code
3. Context:
    - Maintains a reference to a Strategy object
    - Provides a way to change the strategy at runtime
    - Delegates the algorithm execution to the Strategy object

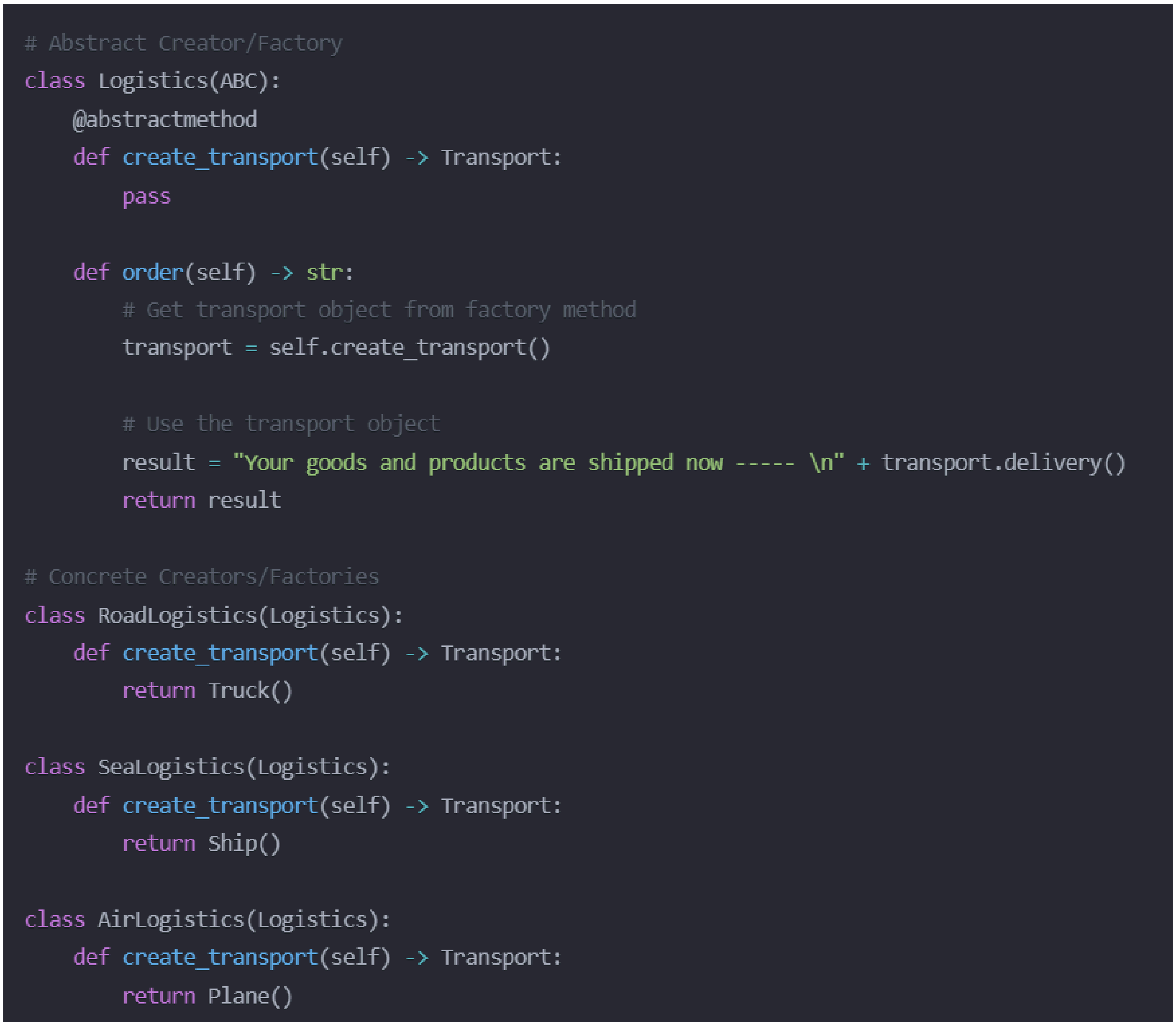
Benefits of this Pattern:

1. Encapsulates algorithms in separate classes
2. Makes algorithms interchangeable
3. Enables runtime algorithm switching
4. Promotes code reuse and maintainability
5. Follows Open/Closed Principle (open for extension, closed for modification)

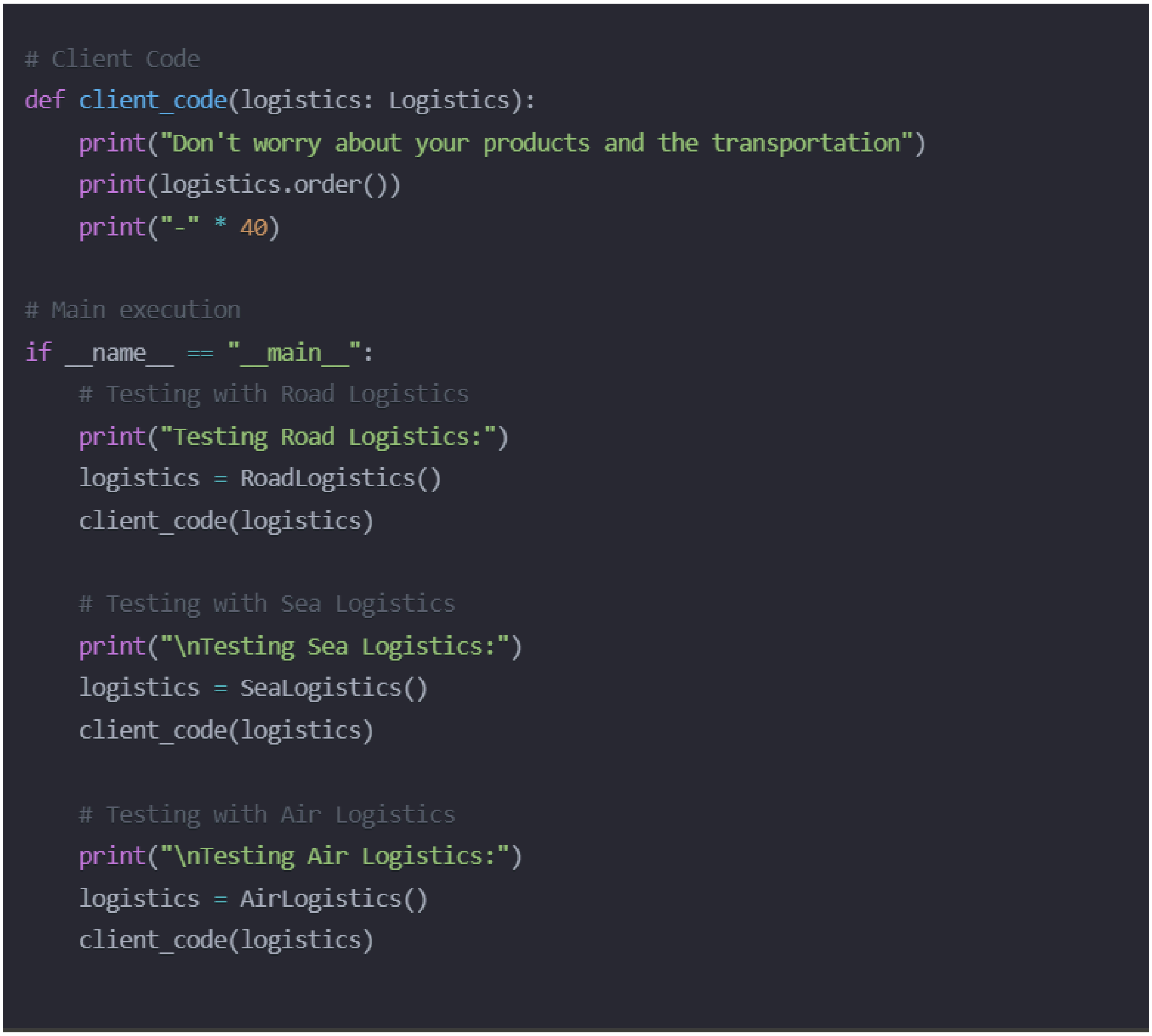
Important Question Ans:



```python
from abc import ABC, abstractmethod

# Abstract Product Interface
class Transport(ABC):
    @abstractmethod
    def delivery(self) -> str:
        pass

# Concrete Products
class Truck(Transport):
    def delivery(self) -> str:
        return "Truck will be delivering your product!"

class Ship(Transport):
    def delivery(self) -> str:
        return "Ship will be delivering your product!"

class Plane(Transport):
    def delivery(self) -> str:
        return "Plane will be delivering your product!"
```



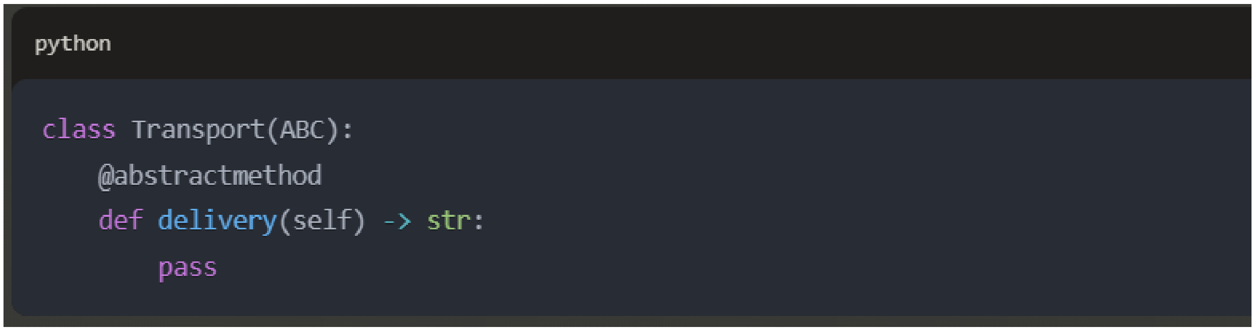
```python
# Abstract Creator/Factory
class Logistics(ABC):
    @abstractmethod
    def create_transport(self) -> Transport:
        pass

    def order(self) -> str:
        # Get transport object from factory method
        transport = self.create_transport()

        # Use the transport object
        result = "Your goods and products are shipped now ----- \n" + transport.delivery()
        return result

# Concrete Creators/Factories
class RoadLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Truck()

class SeaLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Ship()

class AirLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Plane()
```



```python
# Client Code
def client_code(logistics: Logistics):
    print("Don't worry about your products and the transportation")
    print(logistics.order())
    print("-" * 40)

# Main execution
if __name__ == "__main__":
    # Testing with Road Logistics
    print("Testing Road Logistics:")
    logistics = RoadLogistics()
    client_code(logistics)

    # Testing with Sea Logistics
    print("\nTesting Sea Logistics:")
    logistics = SeaLogistics()
    client_code(logistics)

    # Testing with Air Logistics
    print("\nTesting Air Logistics:")
    logistics = AirLogistics()
    client_code(logistics)
```

Explain::

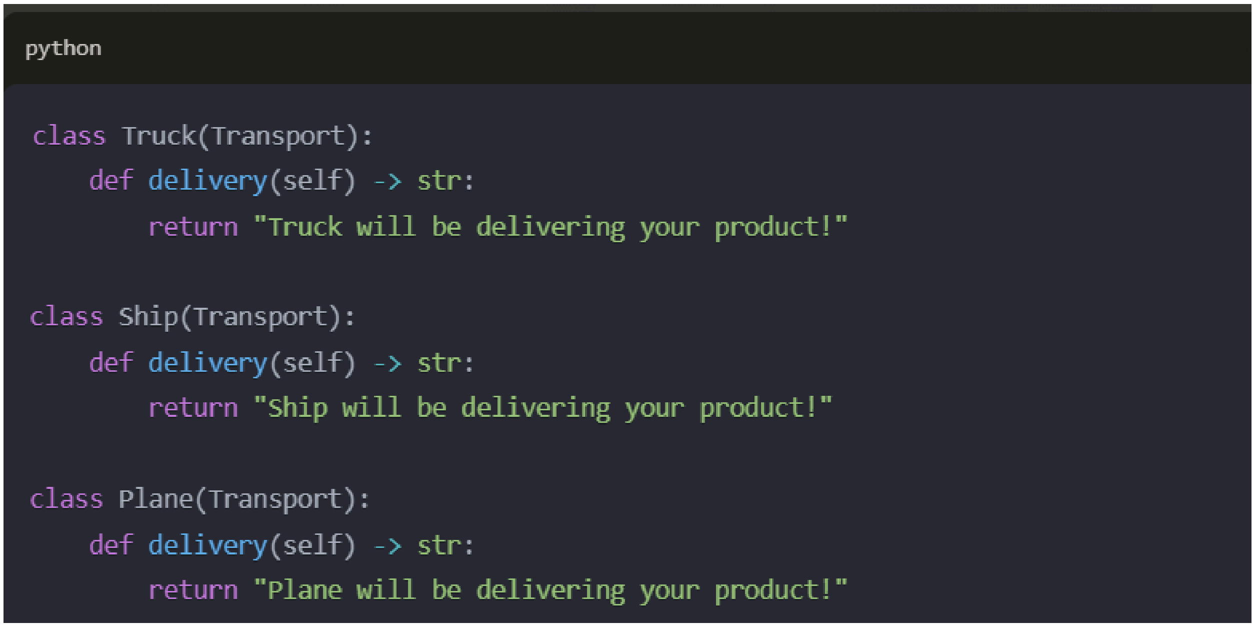
Abstract Product (Transport):



```python
class Transport(ABC):
    @abstractmethod
    def delivery(self) -> str:
        pass
```

- This is the interface that all transport types must implement
- Defines the common method `delivery()` that each transport type will provide
- Uses ABC (Abstract Base Class) to make it an interface

2. Concrete Products:

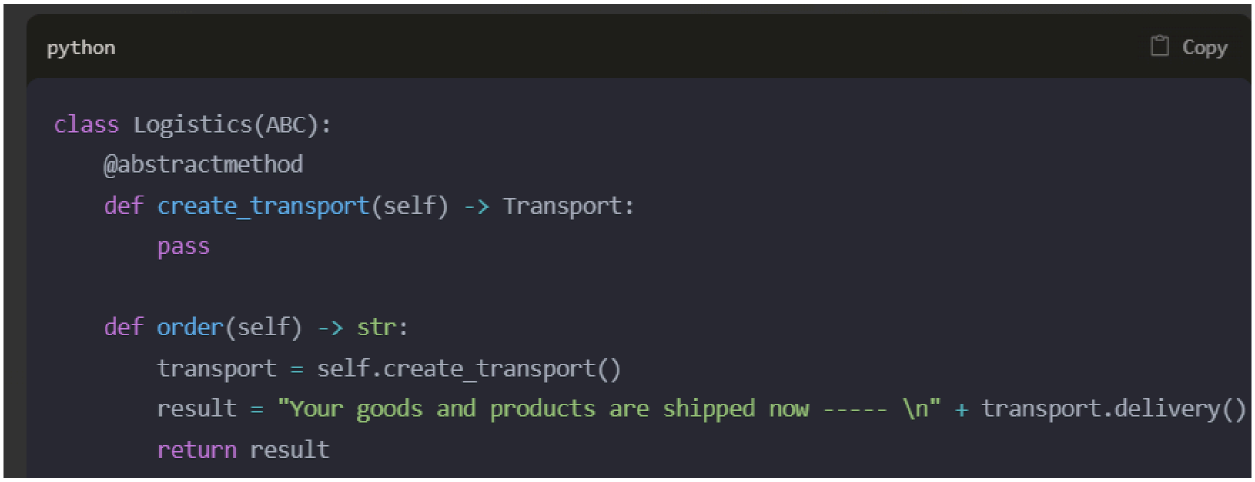


```python
class Truck(Transport):
    def delivery(self) -> str:
        return "Truck will be delivering your product!"

class Ship(Transport):
    def delivery(self) -> str:
        return "Ship will be delivering your product!"

class Plane(Transport):
    def delivery(self) -> str:
        return "Plane will be delivering your product!"
```

These are specific implementations of the Transport interface

- Each provides its own implementation of the `delivery()` method
- Can be extended with more transport types without changing existing code

3. Abstract Creator (Logistics):

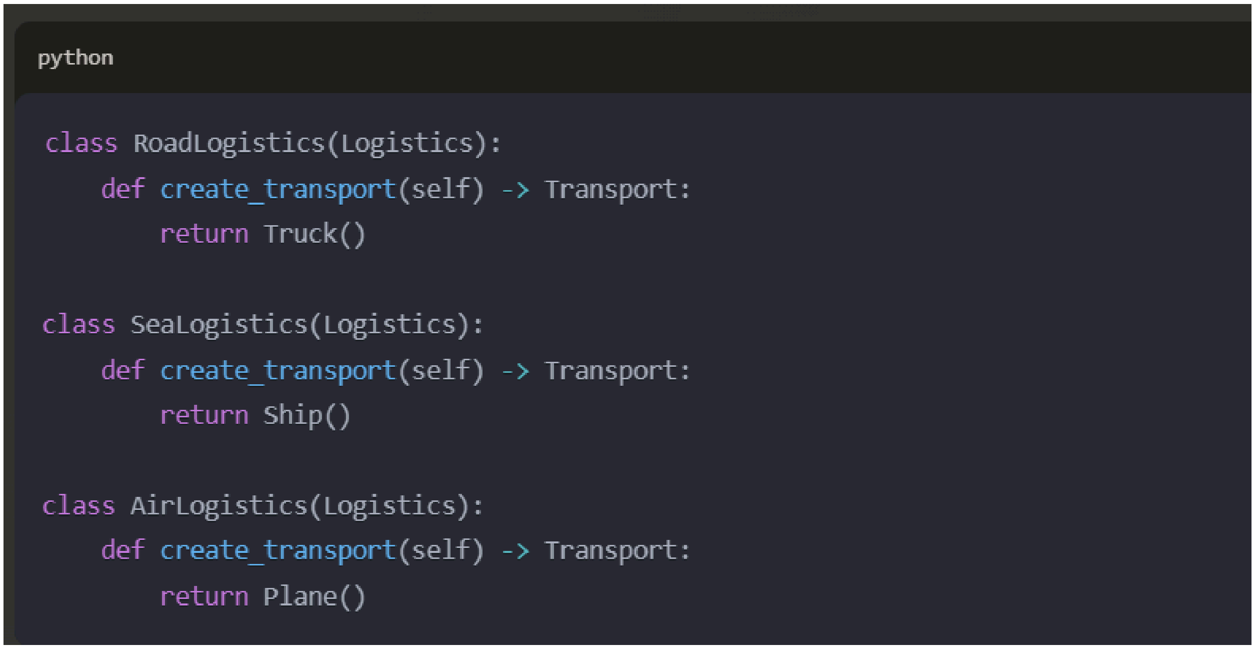


```python
class Logistics(ABC):
    @abstractmethod
    def create_transport(self) -> Transport:
        pass

    def order(self) -> str:
        transport = self.create_transport()
        result = "Your goods and products are shipped now ----- \n" + transport.delivery()
        return result
```

Abstract factory class that declares the factory method

- Contains common business logic in `order()` method
- Factory method `create_transport()` must be implemented by subclasses

4. Concrete Creators:

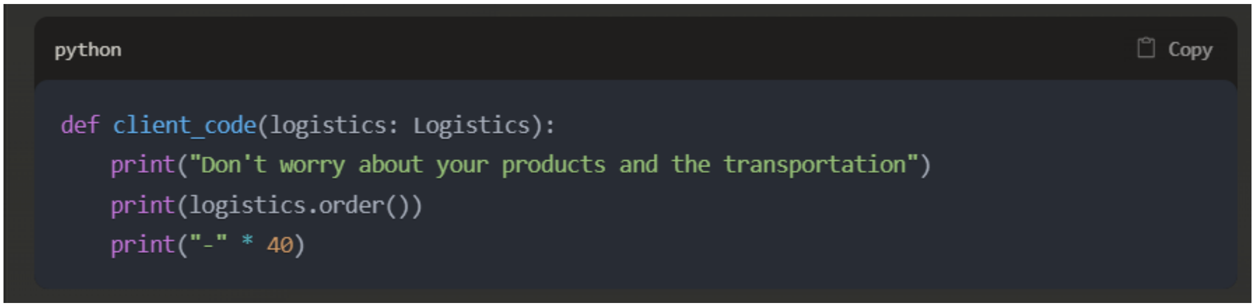


```python
class RoadLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Truck()

class SeaLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Ship()

class AirLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Plane()
```

Specific implementations of the Logistics factory

- Each creates its own type of transport
- New logistics types can be added without modifying existing code

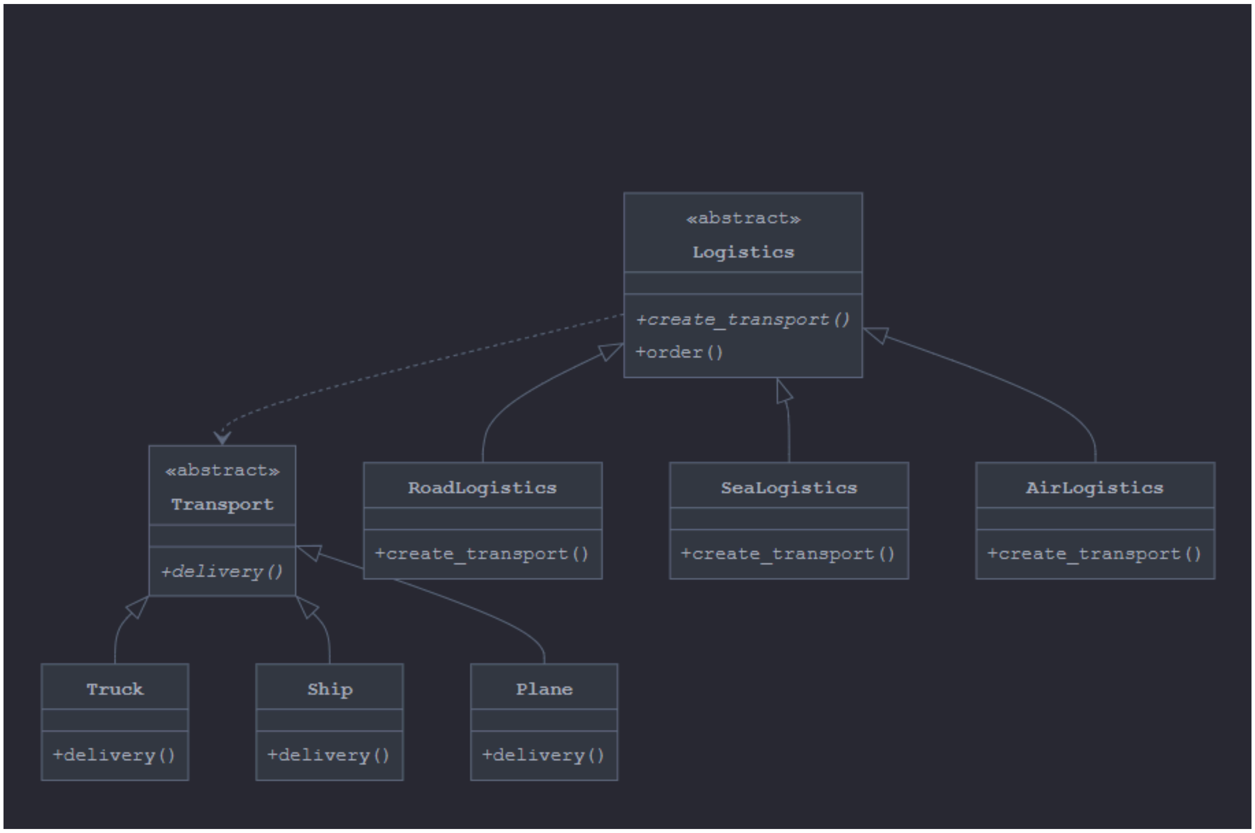
5. Client Code:



```python
def client_code(logistics: Logistics):
    print("Don't worry about your products and the transportation")
    print(logistics.order())
    print("-" * 40)
```

- Works with any type of logistics without knowing concrete classes
- Demonstrates the flexibility of the factory pattern

Visual :



«abstract»
Logistics
+create_transport()
+order()
«abstract»
Transport
+delivery()
RoadLogistics
+create_transport()
SeaLogistics
+create_transport()
AirLogistics
+create_transport()
Truck
+delivery()
Ship
+delivery()
Plane
+delivery()