Design Patterns Lab Manual(Hand Book)

1Q. Design and implement ShapeFactory class that generates different types of Shape objects (Circle, Square, Rectangle) based on input parameters using Factory Design Pattern.

ANS:

Code Implementation

```
class Shape:
    def draw(self): pass

class Circle(Shape):
    def draw(self): return "Drawing Circle"

class Square(Shape):
    def draw(self): return "Drawing Square"

class Rectangle(Shape):
    def draw(self): return "Drawing Rectangle"

class ShapeFactory:
    @staticmethod
    def create_shape(shape_type):
        shapes = {'circle': Circle, 'square': Square, 'rectangle': Rectangle}
        return shapes.get(shape_type.lower())()

# Client Code
shape1 = ShapeFactory.create_shape('circle')
shape2 = ShapeFactory.create_shape('square')

print(shape1.draw())  # Output: Drawing Circle
print(shape2.draw())  # Output: Drawing Square
```

```
OUTPUT:
Drawing Circle
Drawing Square
```

Explained

1. Abstract Base Class: Shape

- **`Shape`**: This is an abstract base class that defines a common interface for all shapes.
- The `draw` method is defined but not implemented (using `pass`), which means that any subclass must implement this method.

2. Concrete Shape Classes

- **`Circle`**, **`Square`**, and **`Rectangle`** are concrete implementations of the `Shape` class.
- Each of these classes implements the `draw` method to provide specific behavior. For instance, `Circle` returns "Drawing Circle", and so on.

3. ShapeFactory Class

- `ShapeFactory`: This is the factory class responsible for creating shape objects.
- The `create_shape` method is a static method that takes `shape_type` as an argument.
- A dictionary named `shapes` maps shape type names (in lowercase) to their corresponding classes.
- The method retrieves the appropriate class from the dictionary based on the input string and creates an instance of that class using `()`

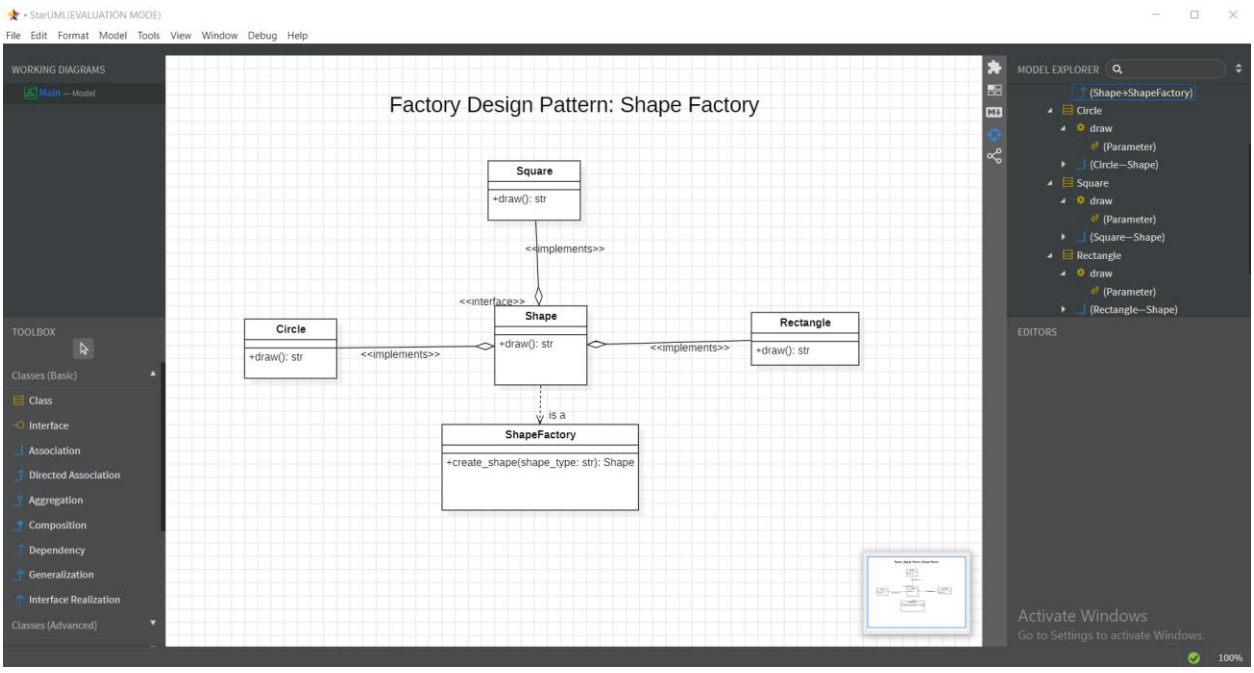
4. Client Code

- The client code calls the `create_shape` method on the `ShapeFactory` to create a `Circle` and a `Square`.
- It then calls the `draw` method on each shape instance and prints the output.

Summary

- This implementation of the **Factory Design Pattern** encapsulates the creation logic of different shape objects within the `ShapeFactory`.
- Clients can create different shapes without needing to know the specifics of how those shapes are implemented, adhering to the principle of separation of concerns.

UML Diagram



2 Q. Design and Implement an AbstractFactory class to create families of related or dependent objects with respect to decathlon store without specifying their concrete classes using Abstract Factory?

ANS:

Code implementation

```
from abc import ABC, abstractmethod

# Abstract Products
class Shoes(ABC):
    @abstractmethod
    def type(self): pass

class Clothes(ABC):
    @abstractmethod
    def type(self): pass

# Concrete Products
class RunningShoes(Shoes):
    def type(self): return "Running Shoes"

class SwimmingSuit(Clothes):
    def type(self): return "Swimming Suit"

# Abstract Factory
class SportsFactory(ABC):
    @abstractmethod
    def create_shoes(self): pass

    @abstractmethod
    def create_clothes(self): pass

# Concrete Factory
class DecathlonFactory(SportsFactory):
    def create_shoes(self): return RunningShoes()
    def create_clothes(self): return SwimmingSuit()

# Client Code
factory = DecathlonFactory()
print(factory.create_shoes().type())  # Output: Running Shoes
print(factory.create_clothes().type())  # Output: Swimming Suit
```

OUTPUT:

Running Shoes

Swimming Suit

Explained

1 Abstract Product Classes

- **`Shoes`** and **`Clothes`**: These are abstract base classes that define the interface for different types of products.
- The `type` method is declared as an abstract method, meaning any subclass must implement this method to provide specific behavior.

2 Concrete Product Classes

- **`RunningShoes`** and **`SwimmingSuit`**: These classes are concrete implementations of the abstract product classes.
- Each class implements the `type` method to return a specific product description.

3. Abstract Factory Class

`SportsFactory`: This is an abstract factory class that declares methods for creating products. It defines the methods `create_shoes` and `create_clothes`, which must be implemented by any concrete factory.

4. Concrete Factory Class

`DecathlonFactory`: This class is a concrete implementation of the `SportsFactory`. It provides the implementation for the methods to create specific products (i.e., `RunningShoes` and `SwimmingSuit`).

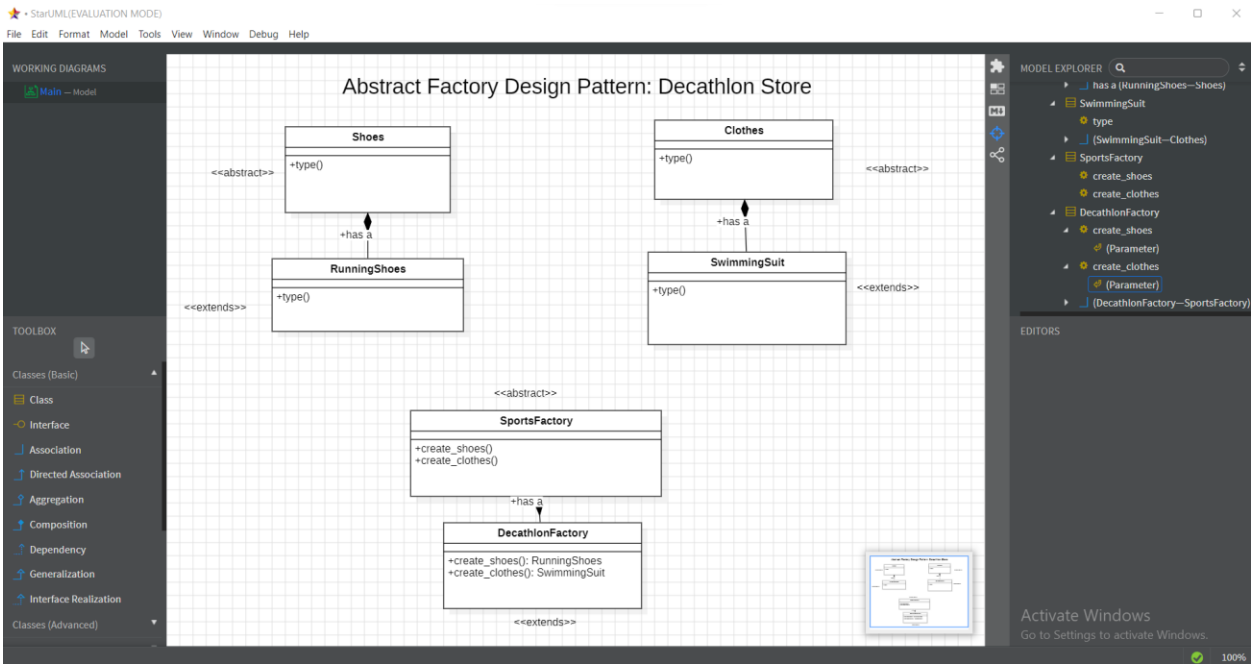
5. Client Code

- The client code creates an instance of `DecathlonFactory`.
- It calls the `create_shoes` method to create a `RunningShoes` object and then calls the `type` method to get its description.
- Similarly, it creates a `SwimmingSuit` object and retrieves its description.

Summary

- This implementation demonstrates the **Abstract Factory Pattern**, which allows the creation of families of related or dependent objects (in this case, shoes and clothes) without specifying their concrete classes.
- The use of abstract classes and methods helps in providing a clear interface and facilitates the creation of new products by simply extending the existing structure without modifying the client code.

UML Diagram



3 Q. Design and implement a complex object like a House using a step-by-step Builder pattern, allowing different representations of the house (wooden, brick, etc.).?

ANS:

```
class House:
    def __init__(self, foundation, roof, interior):
        self.foundation = foundation
        self.roof = roof
        self.interior = interior
    def __str__(self):
        return f"House with {self.foundation}, {self.roof}, and {self.interior}"

# Builder Interface
class HouseBuilder:
    def build(self):
        pass

# Wooden House Builder
class WoodenHouseBuilder(HouseBuilder):
    def build(self):
        return House("Wooden foundation", "Wooden roof", "Wooden interior")
```

```
# Brick House Builder
class BrickHouseBuilder(HouseBuilder):
    def build(self):
        return House("Concrete foundation", "Tile roof", "Modern interior")

# Director
class Director:
    def __init__(self, builder):
        self.builder = builder

    def construct_house(self):
        return self.builder.build()

# Client Code
if __name__ == "__main__":
    # Build Wooden House
    wooden_house = Director(WoodenHouseBuilder()).construct_house()
    print(wooden_house)

    # Build Brick House
    brick_house = Director(BrickHouseBuilder()).construct_house()
    print(brick_house)
```

OUTPUT

House with Wooden foundation, Wooden roof, and Wooden interior

House with Concrete foundation, Tile roof, and Modern interior

Explained

1.House Class:

- Represents the complex object (House) that is being constructed.
- The constructor takes three parameters: `foundation`, `roof`, and `interior`.
- The `__str__` method provides a string representation of the house, detailing its components.

2.Builder Interface:

- An abstract base class that defines a method `build` that concrete builders will implement.

3.Concrete Builders:

- **WoodenHouseBuilder**: Implements the `build` method to create a wooden house with specific components.

- **BrickHouseBuilder**: Implements the `build` method to create a brick house with its own set of components.

4.Director Class:

- This class takes a builder instance and orchestrates the construction process. It provides a `construct_house` method to build the house.

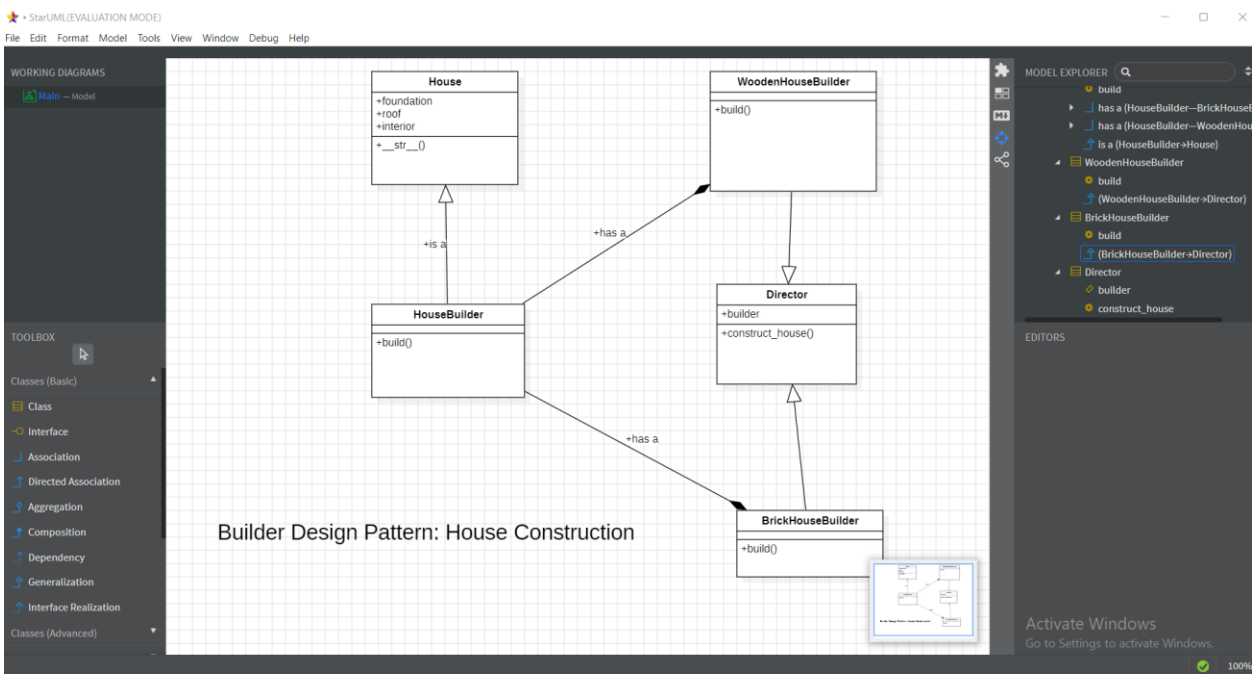
5.Client Code:

- The `if __name__ == "__main__":` block demonstrates how to use the builder and director to construct a wooden house and a brick house, and print their details.

Summary

- This implementation of the **Builder Pattern** allows for the creation of different representations of a complex object (House) by using specific builders.
- The use of the **Director** class simplifies the process of constructing the object, encapsulating the construction logic away from the client code.
- This pattern promotes separation of concerns and enhances code maintainability and flexibility, as adding new types of houses can be done by creating new builder classes without modifying existing code.

UML Diagram



4 Q.Design and Implement to Extend a Coffee object with dynamic features (e.g., milk, sugar, whipped cream) using Decorators?

AND

Code Implementation

```
# Base Coffee class
class Coffee:
    def cost(self):
        return 5  # Base cost of coffee

# Base Decorator class
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

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

# Concrete Decorators
class MilkDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1  # Add cost of milk

class SugarDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 0.5  # Add cost of sugar

class WhippedCreamDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1.5  # Add cost of whipped cream

# Client Code
if __name__ == "__main__":
    coffee = Coffee()  # Basic coffee
    coffee_with_milk = MilkDecorator(coffee)  # Coffee with milk
    coffee_with_milk_sugar = SugarDecorator(coffee_with_milk)  # Coffee with milk and sugar
    coffee_full = WhippedCreamDecorator(coffee_with_milk_sugar)  # Coffee with milk, sugar,
and whipped cream

    print(f"Total cost: ${coffee_full.cost():.2f}")  # Output: Total cost: $8.00
```

OUTPUT

```
Total cost: $8.00
```

`Explained`

1 **Base Coffee Class**:

- Represents the core object with a base cost. In this case, the base cost of coffee is set to `$5`.

2 **Base Decorator Class**:

- This class inherits from `Coffee` and serves as a base for all decorators. It holds a reference to a `Coffee` object.

3 Concrete Decorators:

- Each concrete decorator enhances the base coffee object by adding a specific cost:
  - **MilkDecorator**: Adds `$1` to the coffee cost.
  - **SugarDecorator**: Adds `$0.5` to the coffee cost.
  - **WhippedCreamDecorator**: Adds `$1.5` to the coffee cost

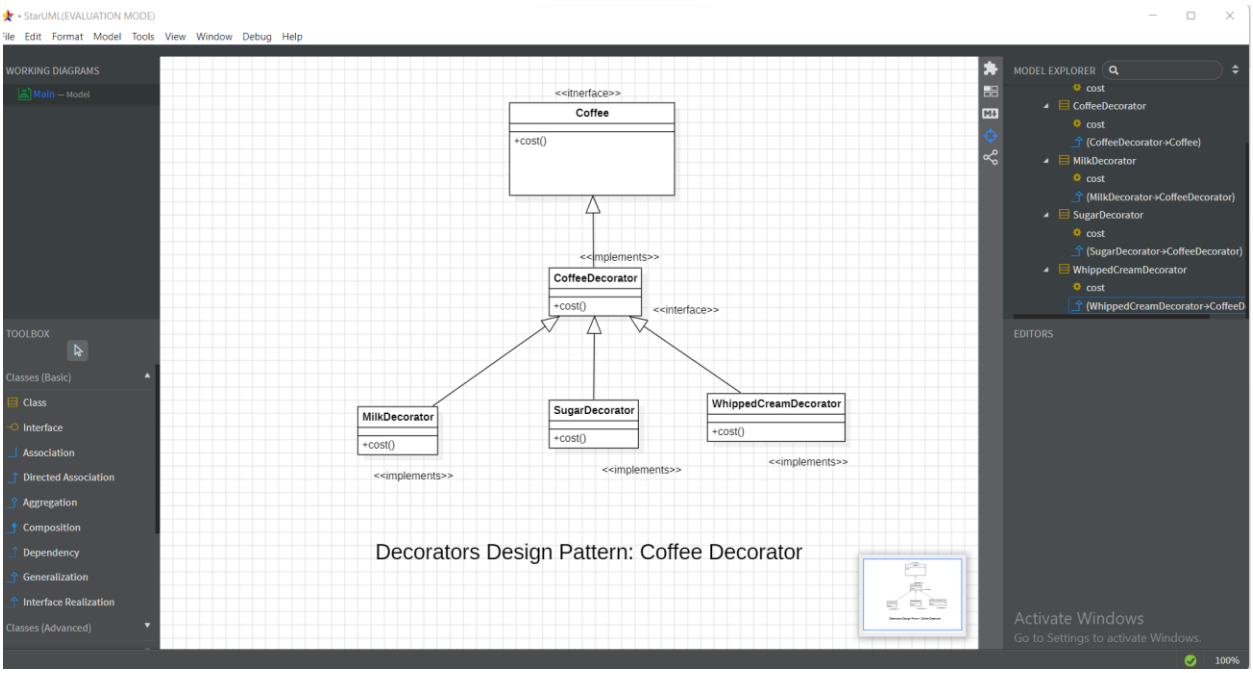
4. Client Code:

- This section of the code demonstrates how to use the decorators to create a customized coffee order.
- It starts with a basic `Coffee` object and then decorates it step by step:
  - Adding milk.
  - Adding sugar.
  - Finally, adding whipped cream.
- The total cost is printed out.

Summary

- This implementation of the **Decorator Pattern** allows for the dynamic addition of functionalities (in this case, additional ingredients) to an object (coffee) without modifying its structure.
- The pattern promotes code reusability and adheres to the **Open/Closed Principle**, as new decorators can be added without changing the existing codebase.
- The total cost is computed at runtime based on the selected decorators, demonstrating flexibility in object enhancement.

UML Diagram



5Q.Design and Implement a Logger class ensuring a single instance throughout the application?

ANS:

```
class Logger:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Logger, cls).__new__(cls)
        return cls._instance

    def log(self, message):
        print(f"Log: {message}")
# Client Code
logger1 = Logger()
logger2 = Logger()

logger1.log("This is a log message.")
print(logger1 is logger2)  # Output: True
```

OUTPUT:

Log: This is a log message.

True

- The `Logger` class is defined to implement the Singleton pattern, ensuring that only one instance of the class exists throughout the application.
- `_instance` is a class variable that holds the single instance of the `Logger` class. Initially, it's set to `None`.

- The `__new__` method is responsible for creating a new instance of the class. It is a static method called before `__init__`.
- The method checks if `_instance` is `None` (i.e., no instance exists yet). If so, it creates a new instance using `super(Logger, cls).__new__(cls)` and assigns it to `_instance`.
- If an instance already exists, it simply returns the existing instance stored in `_instance`.
- This ensures that any call to create a `Logger` instance will return the same instance.

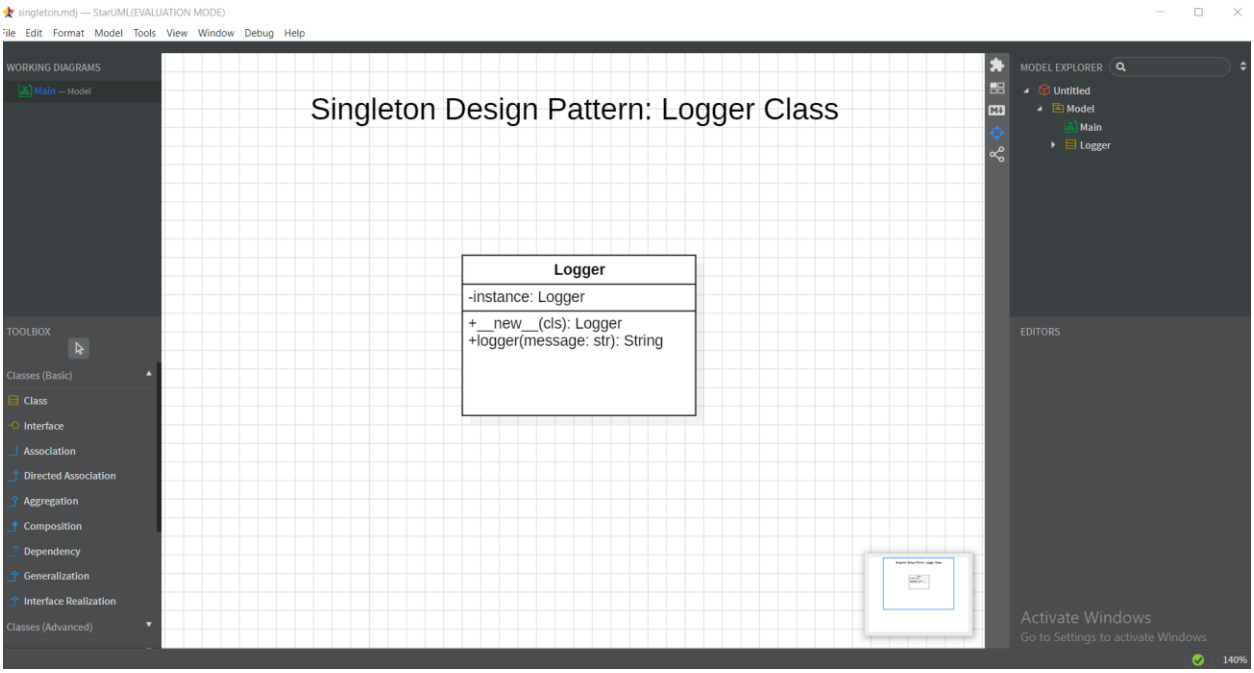
- The `log` method takes a `message` as a parameter and prints it prefixed by "Log: ".
- This method is used to log messages to the console.

- Two variables, `logger1` and `logger2`, are assigned to instances of the `Logger` class.
- Because of the Singleton implementation, both `logger1` and `logger2` point to the same instance of `Logger`. Hence, the `print(logger1 is logger2)` statement evaluates to `True`, confirming that both references point to the same object in memory.
- The line `logger1.log("This is a log message.")` logs the message using the `log` method of the `Logger` instance.

Summary

- This implementation effectively creates a Singleton Logger class that ensures only one instance is created and used across the application. Whenever you need to log a message, you can use the `Logger` class without worrying about creating multiple instances. This is particularly useful for maintaining consistent logging behavior in applications.

UML Diagram



6 Q. Design and implement an Adapter Pattern for a Music System?

AND:

Code Implementation

```
# Step 1: Define the Old Music Player interface
class OldMusicPlayer:
    def play_cd(self, cd_name: str):
        raise NotImplementedError("This method should be overridden.")

# Step 2: Implement the Modern Music Player
class ModernMusicPlayer:
    def play_streaming(self, song_name: str):
        print(f"Playing {song_name} from a streaming service.")

# Step 3: Create an Adapter that adapts ModernMusicPlayer to OldMusicPlayer
class StreamingAdapter(OldMusicPlayer):
    def __init__(self, modern_player: ModernMusicPlayer):
        self.modern_player = modern_player

    def play_cd(self, cd_name: str):
        # Adapting the play_cd method to use the play_streaming method of ModernMusicPlayer
        print(f"Adapting CD '{cd_name}' to a streaming song.")
        self.modern_player.play_streaming(cd_name)
```

```
# Step 4: Client code demonstrating the Adapter Pattern
if __name__ == "__main__":
    # Create a modern music player
    modern_player = ModernMusicPlayer()

    # Use the adapter to connect the modern player to the old system
    adapter = StreamingAdapter(modern_player)

    # Now play a "CD" (which is really a streaming song)
    adapter.play_cd("Classic Hits")
```

OUTPU:

Adapting CD 'Classic Hits' to a streaming song.

Playing Classic Hits from a streaming service.

Explained

1 **Old Music Player Interface**:

- This is the target interface that we want to adapt to. It defines a method `play_cd`, which is supposed to be implemented by any class that follows this interface.

2 **Modern Music Player**:

- This class represents the existing implementation that does not adhere to the `OldMusicPlayer` interface. It has a method `play_streaming` for playing songs from a streaming service.

3 **Streaming Adapter**:

- This is the adapter class that allows `ModernMusicPlayer` to be used where `OldMusicPlayer` is expected. It inherits from `OldMusicPlayer` and implements the `play_cd` method.
- Inside `play_cd`, it adapts the call to use `play_streaming` of `ModernMusicPlayer`.

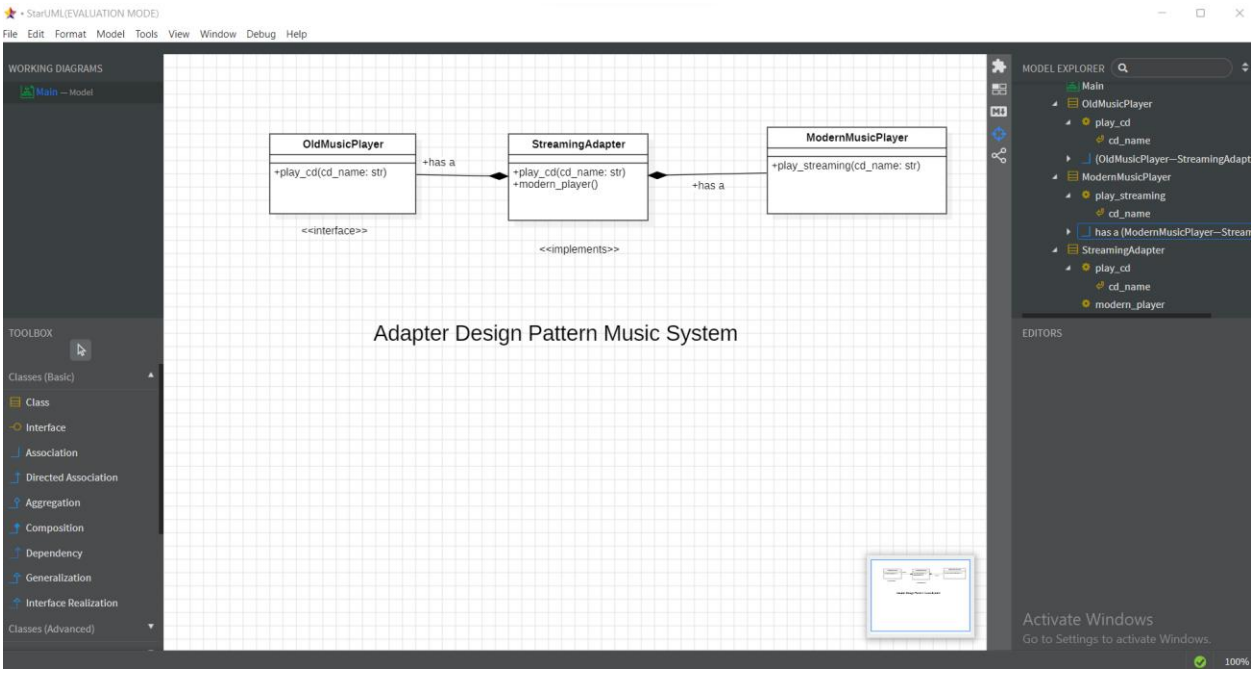
4 **Client Code**:

- In the `if __name__ == "__main__":` block, we create an instance of `ModernMusicPlayer` and then create an adapter (`StreamingAdapter`) that connects it to the `OldMusicPlayer` interface.
- When `play_cd` is called on the adapter, it internally calls the modern player's method, effectively bridging the gap between the two interfaces.

Summary

- The **Adapter Pattern** is useful when you want to allow two incompatible interfaces to work together.
- In this example, `StreamingAdapter` allows the modern music player to be used in a system that expects an old music player interface.
- This approach promotes code reuse and flexibility, allowing new implementations to be added without altering existing code structures.

UML Diagram



7 Q. Design and Implement an Observer pattern for a news agency to notify subscribers of updates?

AND:

Code Implementation

```
# Subject (News Agency)
class NewsAgency:
    def __init__(self):
        self.subscribers = []
        self.news = ""
```

```
    # Subscribe a subscriber to the agency
    def subscribe(self, subscriber):
        self.subscribers.append(subscriber)

    # Notify all subscribers about the new news
    def notify(self):
        for subscriber in self.subscribers:
            subscriber.update(self.news)

    # Set the news and notify all subscribers
    def set_news(self, news):
        self.news = news
        self.notify()

# Observer (Subscriber)
class Subscriber:
    def update(self, news):
        print(f"News received: {news}")

# Client Code
if __name__ == "__main__":
    # Create the news agency (Subject)
    agency = NewsAgency()

    # Create subscriber instances (Observers)
    sub1 = Subscriber()
    sub2 = Subscriber()

    # Subscribe the subscribers to the news agency
    agency.subscribe(sub1)
```

```
    agency.subscribe(sub2)

    # Set the news and notify all subscribers

    agency.set_news("Breaking News: Observer Pattern in Python!")
```

```
OUTPUT:
News received: Breaking News: Observer Pattern in Python!

News received: Breaking News: Observer Pattern in Python!
```

Explained

1 Subject (News Agency):

- This class maintains a list of subscribers and manages the news updates. It provides methods to subscribe and unsubscribe subscribers and notify them when there is new news.
- **Key Methods**:

- `subscribe`: Adds a subscriber to the list.
- `unsubscribe`: Removes a subscriber from the list.
- `notify`: Calls the `update` method on all subscribed observers when news is available.
- `add_news`: Sets the news and triggers the notification.

2 Observer (Subscriber):

- This class represents the observers who are interested in receiving updates from the `NewsAgency`. Each subscriber has a name and an `update` method that receives the news.

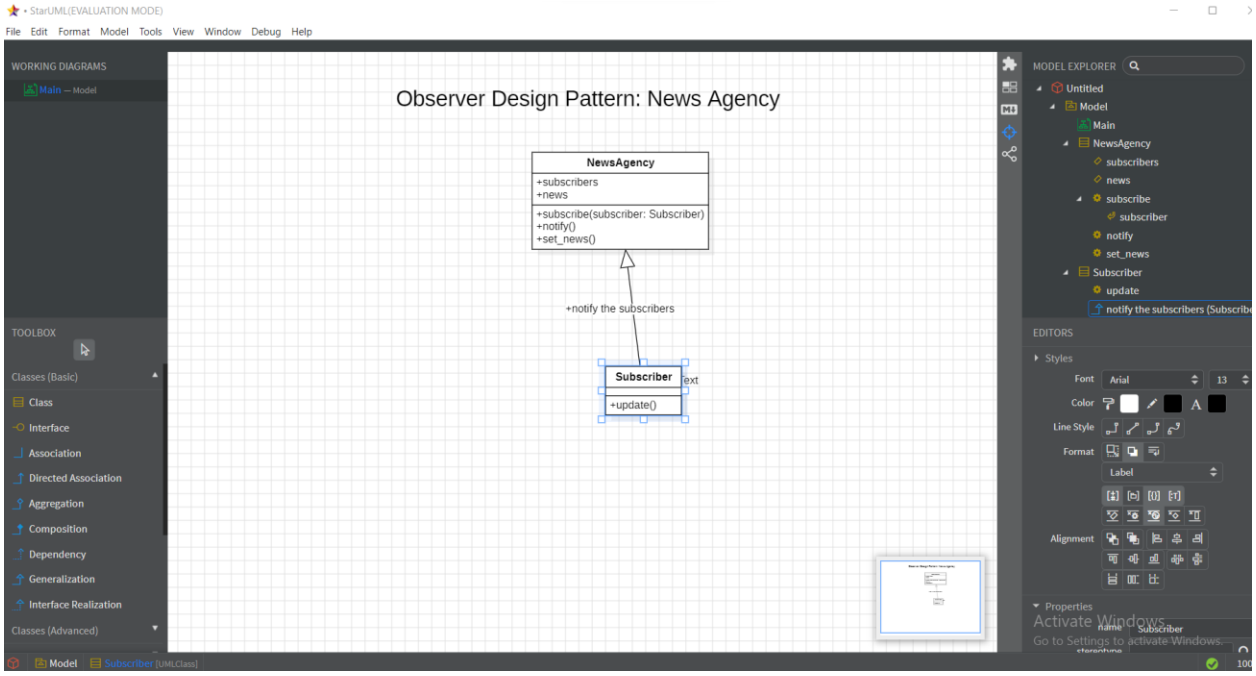
3 Client Code:

- In the main block, a `NewsAgency` instance is created, along with two `Subscriber` instances. The subscribers subscribe to the news agency and then the agency adds news, which triggers notifications to all subscribers.

Summary

- The **Observer Pattern** allows a subject to notify multiple observers about changes in its state, which is useful for implementing distributed event handling systems.
- In this example, the `NewsAgency` acts as the subject that maintains a list of subscribers (observers) and notifies them of any news updates.
- This pattern promotes loose coupling between the subject and its observers, making the system more flexible and maintainable.

UML Diagram



8Q. Design and Implement a Façade pattern for home theatre system?

ANS:

Code Implementation

```
class TV:
    def on(self): print("TV ON")
    def off(self): print("TV OFF")

class SoundSystem:
    def on(self): print("Sound ON")
    def off(self): print("Sound OFF")

class Lights:
    def dim(self): print("Lights DIM")
    def bright(self): print("Lights BRIGHT")

class HomeTheatreFacade:
    def __init__(self):
        self.tv = TV()
        self.sound = SoundSystem()
        self.lights = Lights()
```

```
    def watch_movie(self):
        self.lights.dim(); self.tv.on(); self.sound.on()

    def end_movie(self):
        self.tv.off(); self.sound.off(); self.lights.bright()

# Client Code
home_theatre = HomeTheatreFacade()
home_theatre.watch_movie()  # Start movie
home_theatre.end_movie()    # End movie
```

```
OUTPUT:
Lights DIM
TV ON
Sound ON
TV OFF
Sound OFF
Lights BRIGHT
```

Explained

1 Subsystem Classes:

- These classes represent the various components of the home theater system.
- Each class has methods to turn the component on or off (for `TV` and `SoundSystem`) or to dim or brighten the lights (for `Lights`).

2 Facade Class:

- The `HomeTheatreFacade` class provides a unified interface to the subsystem.
- The constructor initializes instances of `TV`, `SoundSystem`, and `Lights`.
- The `watch_movie` method dims the lights, turns on the TV, and activates the sound system.
- The `end_movie` method turns off the TV and sound system, and brightens the lights.

3 Client Code:

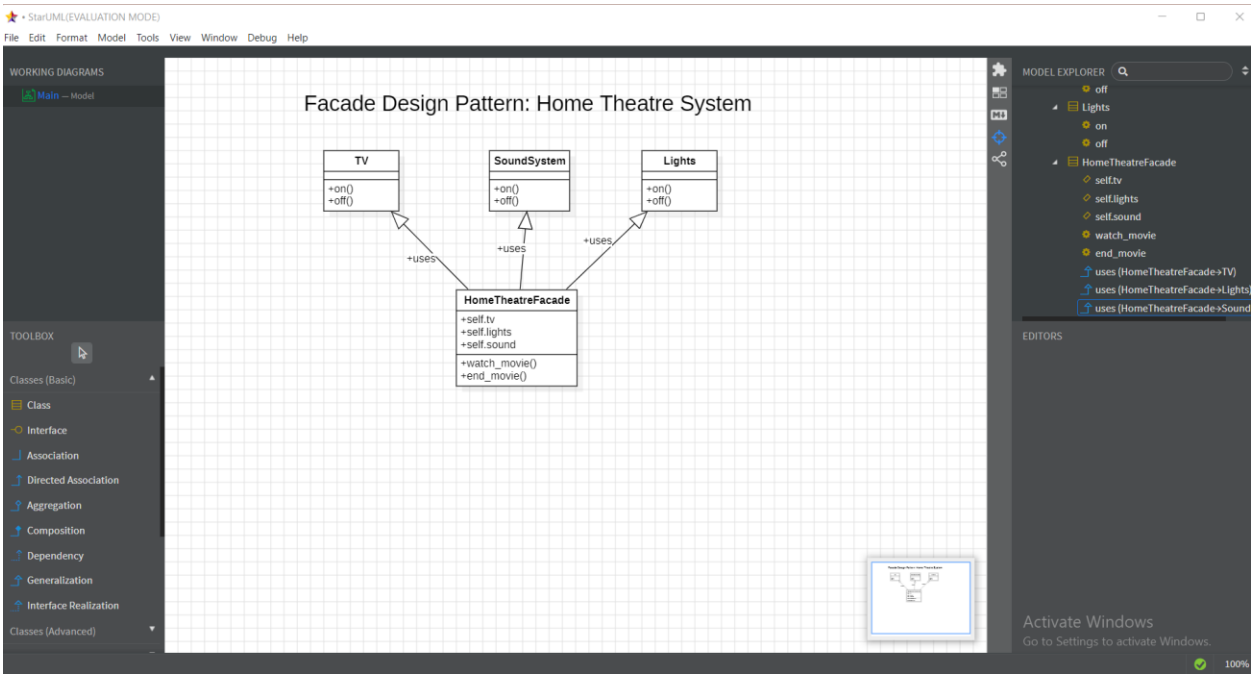
- This section demonstrates how to use the façade
- The client interacts only with the `HomeTheatreFacade`, making it simple to start and end a movie without needing to manage each component individually.

Summary

- The **Facade Pattern** simplifies the interface to a complex system by providing a single class that manages interactions between the system components.

- In this example, the `HomeTheatreFacade` class abstracts the complexity of controlling the `TV`, `SoundSystem`, and `Lights`, allowing the client to perform operations like watching or ending a movie with minimal effort. This pattern promotes code readability and usability.

UML Diagram



9 Q.Design and Implement a Template Method for Document Processing (word, pdf, excel)?

ANS:

```
class DocumentProcessor:
    def process(self):
        self.open_file(); self.read_data(); print("File closed")

class WordProcessor(DocumentProcessor):
    def open_file(self): print("Word opened")
    def read_data(self): print("Reading Word")

class PDFProcessor(DocumentProcessor):
    def open_file(self): print("PDF opened")
    def read_data(self): print("Reading PDF")
```

```
# Client Code
WordProcessor().process()
PDFProcessor().process()

OUTPUT:
Word opened
Reading Word
File closed
PDF opened
Reading PDF
File closed
```

1 **Abstract Class**:

- The `DocumentProcessor` class defines the template method `process()` and the common steps involved in processing a document.
- In the `process()` method, the steps for processing a document are defined, but the specific implementations for opening and reading files are delegated to subclasses through the methods `open_file()` and `read_data()`.

2 **Concrete Classes**:

- The `WordProcessor` and `PDFProcessor` classes provide specific implementations for the steps defined in `DocumentProcessor`.

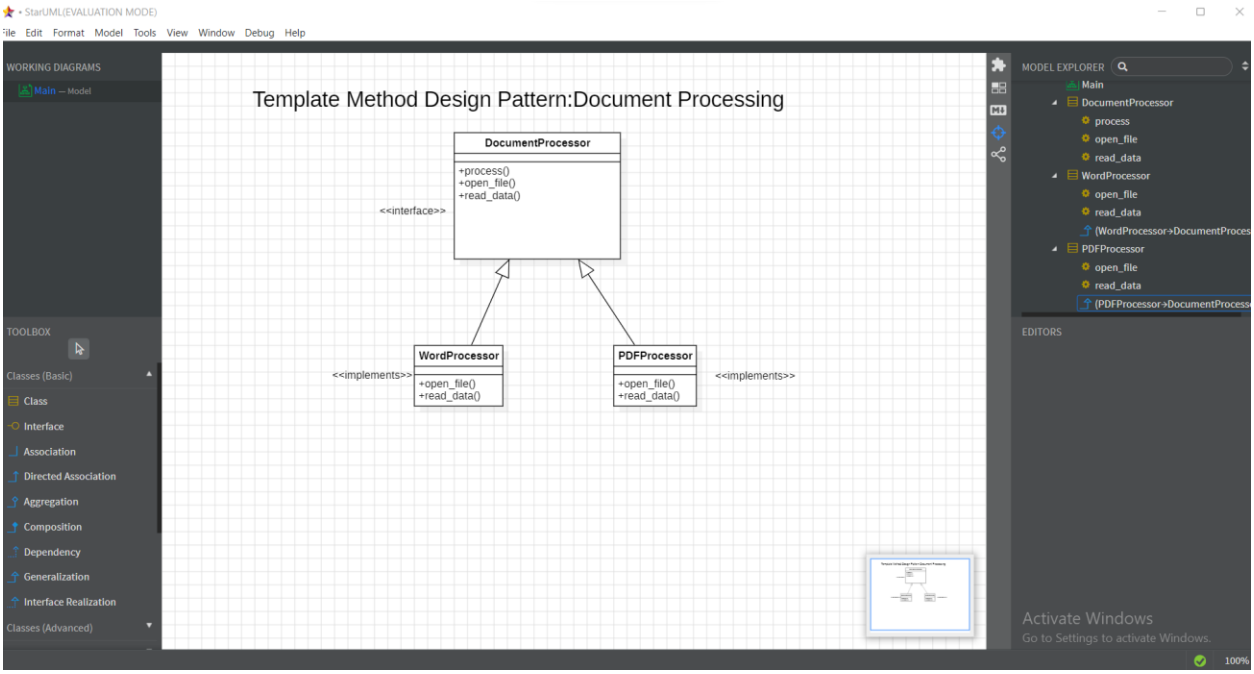
3 **Client Code**:

- The client code demonstrates how to use the template method with both concrete classes
- When you call `process()` on a `WordProcessor` instance, it will open a Word document and read its data. The same applies to the `PDFProcessor`, which processes a PDF file.

Summary

- The **Template Method Pattern** allows you to define a common process (in this case, processing documents) while allowing subclasses to specify the details of certain steps.
- The `DocumentProcessor` class provides the structure for the algorithm, while `WordProcessor` and `PDFProcessor` provide their specific implementations. This promotes code reuse and adherence to the **DRY** (Don't Repeat Yourself) principle.

UML Diagram



10 Q. Design and Implement weather monitoring system that notifies multiple display devices whenever the weather conditions change that follows the Observer Design Pattern?

AND

Code Implementation

```
class WeatherStation:
    def __init__(self):
        self.subscribers = []
        self.temperature = 0

    def subscribe(self, subscriber):
        self.subscribers.append(subscriber)

    def notify(self):
        for sub in self.subscribers:
            sub.update(self.temperature)

    def set_temperature(self, temp):
        self.temperature = temp
        self.notify()
```

```
class DisplayDevice:
    def update(self, temp):
        print(f"Temperature updated: {temp}°C")

# Client Code
station = WeatherStation()
display1 = DisplayDevice()
display2 = DisplayDevice()

station.subscribe(display1)
station.subscribe(display2)

station.set_temperature(25)  # Notify displays

OUTPUT:
Temperature updated: 25°C
Temperature updated: 25°C
Explained
```

1 **Subject Class**:

- The `WeatherStation` class acts as the subject that maintains a list of observers (subscribers) and notifies them of any changes (in this case, the temperature).
- The `subscribe()` method allows an observer to subscribe to updates, while the `notify()` method sends the current temperature to all subscribed observers. The `set_temperature()` method changes the temperature and triggers notifications.

2 Observer Class:

- The `DisplayDevice` class represents the observer that gets updated with the temperature.

3 Client Code:

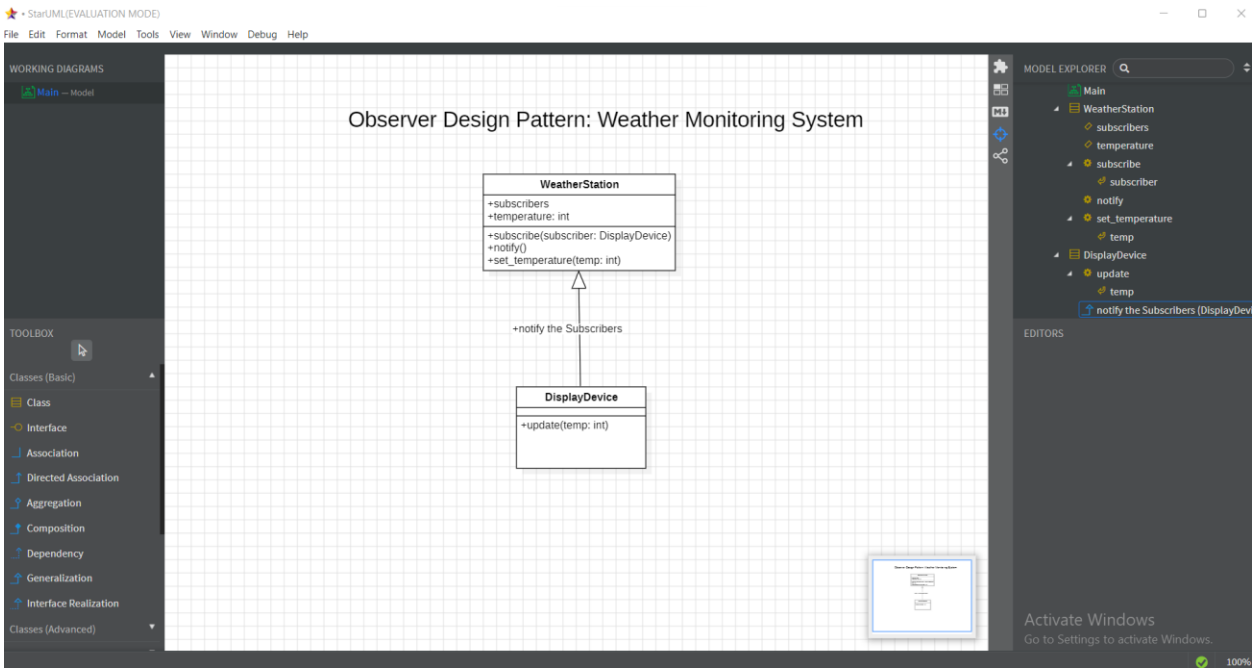
- The client code demonstrates how to use the observer pattern with the `WeatherStation` and `DisplayDevice`.
- When `set_temperature(25)` is called, both display devices are notified of the temperature change.

Summary

- The **Observer Design Pattern** is useful when you want to create a subscription model where multiple objects (observers) are notified about changes in another object (subject).

- In this implementation, the `WeatherStation` serves as the subject that tracks its state (temperature) and notifies all registered `DisplayDevice` instances whenever the temperature changes.

UML Diagram



11 Q. Design and Implement a Proxy pattern to control access to an object (e.g., a protected resource or remote service).?

ANS:

Code Implementation

```
class RealResource:
    def access_resource(self):
        return "Accessing protected resource."

class Proxy:
    def __init__(self, user):
        self.user = user
        self.resource = RealResource()

    def access(self):
        if self.user == "admin":
            return self.resource.access_resource()
        return "Access denied."
```

```
# Client Code
proxy1 = Proxy("admin")
proxy2 = Proxy("guest")

print(proxy1.access())  # Allowed
print(proxy2.access())  # Denied
```

OUTPUT

```
Accessing protected resource.
Access denied.
```

Explained

1 **Real Subject Class**:

- The `RealResource` class represents the actual resource that clients want to access. It has a method to access the resource.
- The `access_resource()` method simulates accessing the protected resource.

2 **Proxy Class**:

- The `Proxy` class controls access to the `RealResource` instance. It checks the user's permissions before allowing access to the resource.
- The `__init__()` method initializes the proxy with the user and the real resource.
- The `access()` method checks if the user has permission (i.e., if the user is "admin"). If so, it grants access; otherwise, it denies access.

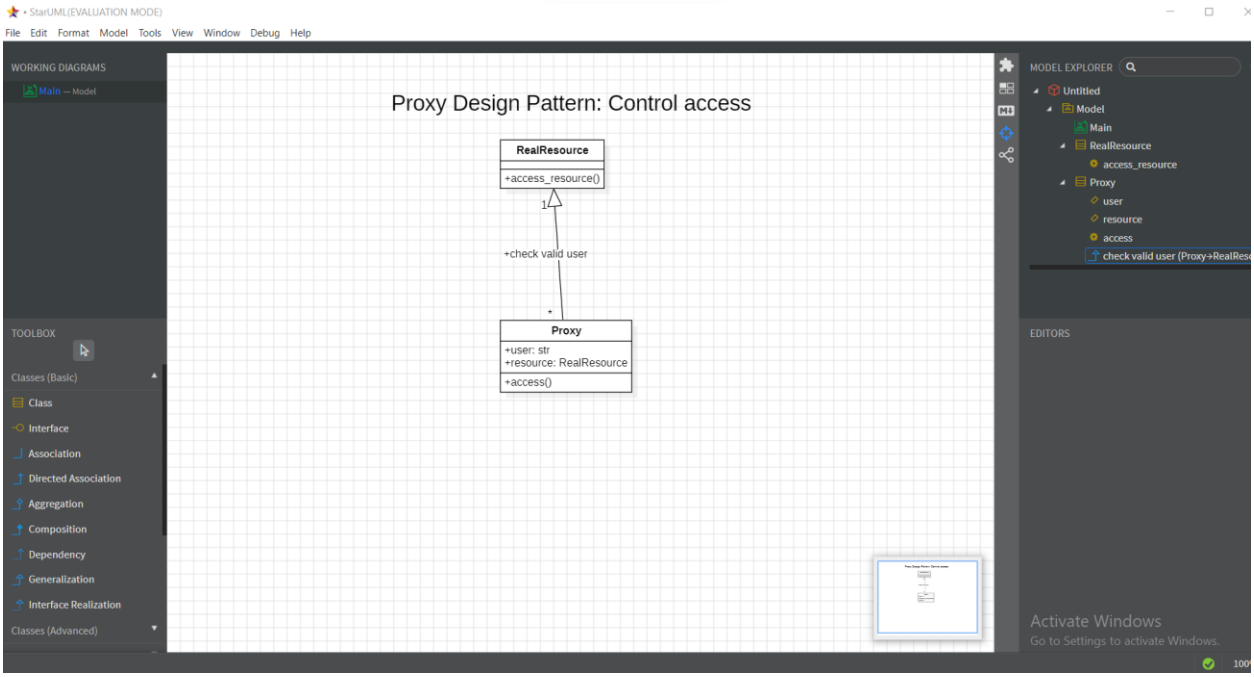
3 **Client Code**:

- The client code demonstrates how to use the `Proxy` to control access to the `RealResource`.
- Two `Proxy` instances are created: one for an "admin" user and one for a "guest" user.
- When calling `access()` on `proxy1`, it allows access to the resource; calling `access()` on `proxy2` denies access.

Summary

- The **Proxy Design Pattern** is useful when you want to provide a surrogate or placeholder for another object to control access to it.
- In this implementation, the `Proxy` checks the user's permissions before allowing access to the `RealResource`, effectively acting as a gatekeeper.

UML Diagram



12. Q. Design and Implement a Mediator pattern to manage communication between a set of objects (e.g., chat room with multiple participants).?

```
class ChatRoom:
    def __init__(self):
        self.participants = []

    def add(self, participant):
        self.participants.append(participant)
        participant.room = self

    def send(self, sender, msg):
        for p in self.participants:
            if p != sender: p.receive(msg, sender.name)

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

    def send(self, msg): self.room.send(self, msg)
    def receive(self, msg, sender): print(f"{sender}: {msg}")
```

```
# Client Code
room = ChatRoom()
alice = Participant("Alice")
bob = Participant("Bob")

room.add(alice)
room.add(bob)

alice.send("Hi Bob!")
OUTPUT:
Alice: Hi Bob!
```

Explained

1 **ChatRoom Class (Mediator)**:

- The `ChatRoom` class manages the participants and coordinates communication between them.
- **Attributes**:
  - `participants`: A list that holds all participants in the chat room.
- **Methods**:
  - `add(participant)`: Adds a participant to the room and assigns the room to the participant.
  - `send(sender, msg)`: Sends a message to all participants except the sender.

2 **Participant Class (Colleague)**:

- The `Participant` class represents users in the chat room.
- **Attributes**:
  - `name`: The name of the participant.
- **Methods**:
  - `send(msg)`: Sends a message through the chat room.
  - `receive(msg, sender)`: Receives a message from another participant and prints it.

3 **Client Code**:

- The client code demonstrates the interaction between participants through the chat room.

Summary

- The **Mediator Design Pattern** allows for the decoupling of components by using a mediator to facilitate communication between them.
- In this example, the `ChatRoom` class serves as the mediator that manages the interaction between `Participant` instances, making the system more organized and reducing dependencies among the components.

UML Diagram