Behavioral Design Patterns

1. **Observer Pattern (User Engagement Notifications)**
    - **Use Case:** A personalized learning platform that sends notifications to users about new courses based on their interests and previous activity, fostering engagement and enhancing user experience.
    - **Code Snippet:**

```
class Observer:
    def update(self, course):
        pass

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

    def update(self, course):
        print(f"{self.name}, check out this new course: {course.title}!")

class Course:
    def __init__(self, title):
        self.title = title

class CourseNotifier:
    def __init__(self):
        self._observers = []

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

    def notify(self, course):
        for observer in self._observers:
            observer.update(course)

# Example usage
notifier = CourseNotifier()
user1 = User("Alice")
notifier.subscribe(user1)
notifier.notify(Course("Introduction to Math"))
```

2. **Strategy Pattern (Personalized Recommendation System)**

- **Use Case:** An e-commerce application that allows users to choose between different product recommendation strategies based on their shopping behavior (e.g., collaborative filtering or content-based filtering), ensuring they receive relevant suggestions.
- **Code Snippet:**

```
class RecommendationStrategy:
    def recommend(self, user_data):
        pass

class CollaborativeFiltering(RecommendationStrategy):
    def recommend(self, user_data):
        return "Recommended products based on user preferences."

class ContentBasedFiltering(RecommendationStrategy):
    def recommend(self, user_data):
        return "Recommended products based on product attributes."

class RecommendationContext:
    def __init__(self, strategy: RecommendationStrategy):
        self.strategy = strategy

    def execute_recommendation(self, user_data):
        return self.strategy.recommend(user_data)

# Example usage
user_data = {"interests": ["electronics", "books"]}
context = RecommendationContext(CollaborativeFiltering())
print(context.execute_recommendation(user_data))
```

Creational Design Patterns

3. **Factory Method Pattern (Product Customization)**
    - **Use Case:** A furniture company that allows customers to customize their furniture pieces by selecting materials and styles. The factory method ensures the correct product type is created based on user choices, enhancing satisfaction and brand loyalty.
    - **Code Snippet:**

```
class Furniture:
    def description(self):
        pass

class Chair(Furniture):
    def description(self):
        return "A custom chair."

class Table(Furniture):
    def description(self):
        return "A custom table."

class FurnitureFactory:
    @staticmethod
    def create_furniture(furniture_type):
        if furniture_type == "chair":
            return Chair()
        elif furniture_type == "table":
            return Table()
        else:
            raise ValueError("Unknown furniture type")

# Example usage
furniture = FurnitureFactory.create_furniture("table")
print(furniture.description())
```

4. **Singleton Pattern (User Session Management)**

- **Use Case:** A web application that uses a session manager to ensure only one instance of a user session exists at a time. This pattern enhances security and user experience by maintaining consistency across the application.
- **Code Snippet**

```
class SessionManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.active_sessions = {}
        return cls._instance

    def create_session(self, user_id):
```

```
        self.active_sessions[user_id] = "Session data"
        return f"Session created for {user_id}"

    def get_session(self, user_id):
        return self.active_sessions.get(user_id, "No active session")

# Example usage
session_manager = SessionManager()
print(session_manager.create_session("user123"))
print(session_manager.get_session("user123"))
```

Structural Design Patterns

5. **Adapter Pattern (Payment Gateway Integration)**
    - **Use Case:** An online marketplace that integrates multiple payment gateways to provide users with flexible payment options. The adapter pattern allows seamless interaction between the platform and various payment systems, enhancing user convenience.
    - **Code Snippet:**

```
class PayPal:
    def make_payment(self, amount):
        print(f"Processing ${amount} payment through PayPal.")

class Stripe:
    def make_payment(self, amount):
        print(f"Processing ${amount} payment through Stripe.")

class PaymentAdapter:
    def __init__(self, payment_gateway):
        self.payment_gateway = payment_gateway

    def pay(self, amount):
        self.payment_gateway.make_payment(amount)

# Example usage
paypal_adapter = PaymentAdapter(PayPal())
stripe_adapter = PaymentAdapter(Stripe())
paypal_adapter.pay(50)
stripe_adapter.pay(100)
```

6. **Decorator Pattern (Enhanced User Profiles)**

- **Use Case:** A social networking application where user profiles can be dynamically enhanced with additional features like profile pictures, bio, and interests, providing a richer user experience and allowing for personalization.
- **Code Snippet:**

```
class UserProfile:
    def get_profile(self):
        return "User profile"

class ProfilePictureDecorator:
    def __init__(self, profile):
        self.profile = profile

    def get_profile(self):
        return self.profile.get_profile() + " + Profile Picture"

class BioDecorator:
    def __init__(self, profile):
        self.profile = profile

    def get_profile(self):
        return self.profile.get_profile() + " + Bio"

# Example usage
profile = UserProfile()
decorated_profile = ProfilePictureDecorator(profile)
fully_decorated_profile = BioDecorator(decorated_profile)
print(fully_decorated_profile.get_profile())  # Output: User profile + Profile Picture + Bio
```