**CS 752 Software Architecture and Design Practices**

**Course Project – PART A**

**Software Design**

| **Group Number** | CS752-2024-09 |
| --- | --- |
| **BRD Module Code** | OAES-002 |
| **BRD Module Name** | Assessment Instrument Management |
| **Group Members** | |
| **Roll Number** | **Student Name** |
| MT2023083 | H Anarghya |
| MT2023161 | Nishtha Paul |
| MT2023195 | Nisha Rathod |

## 

## **Class Diagram**

## 

## **Sequence Diagrams**

### Create Assessment Instrument

### 

### Review Assessment Instrument

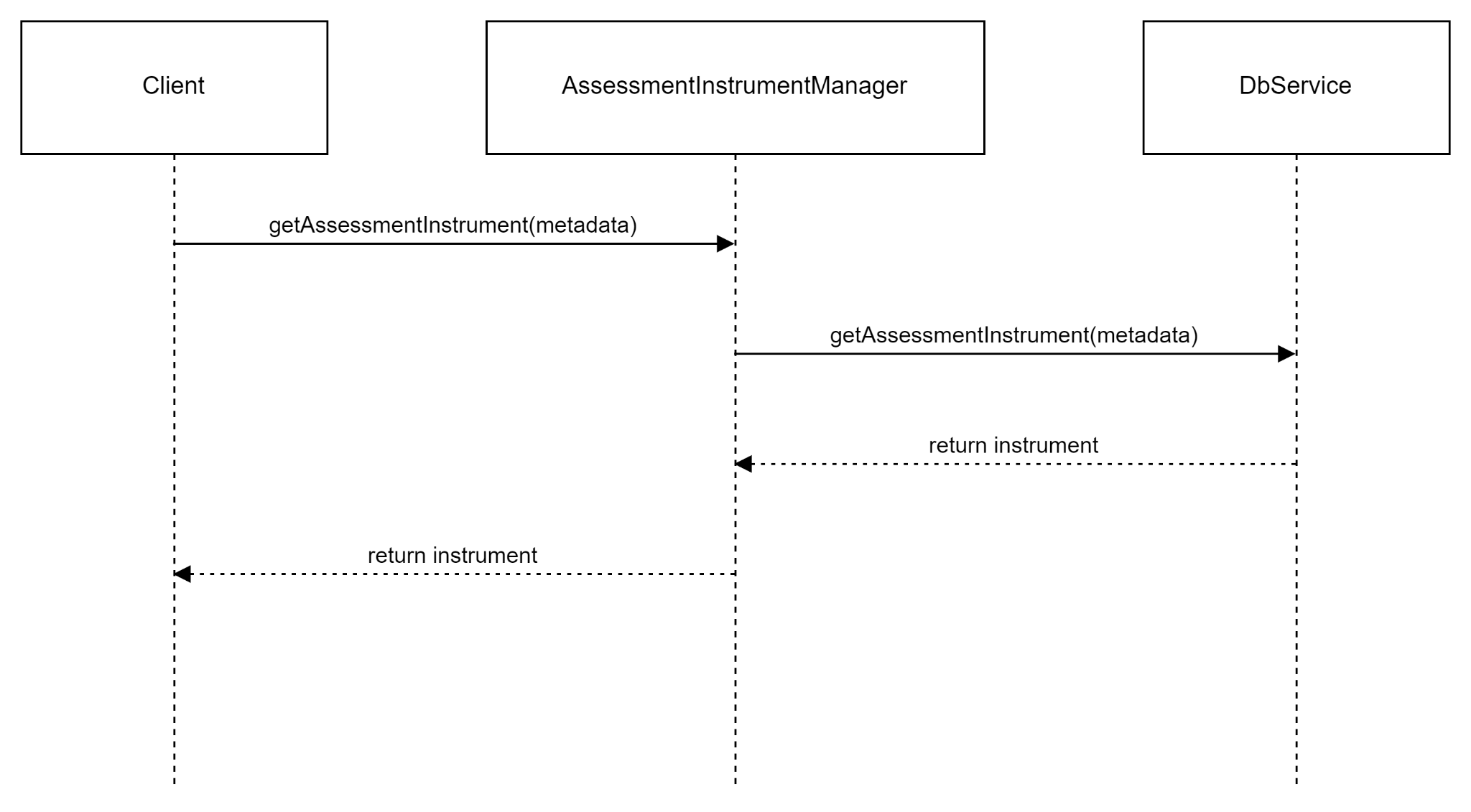
### Approve Assessment Instrument

### Add Item

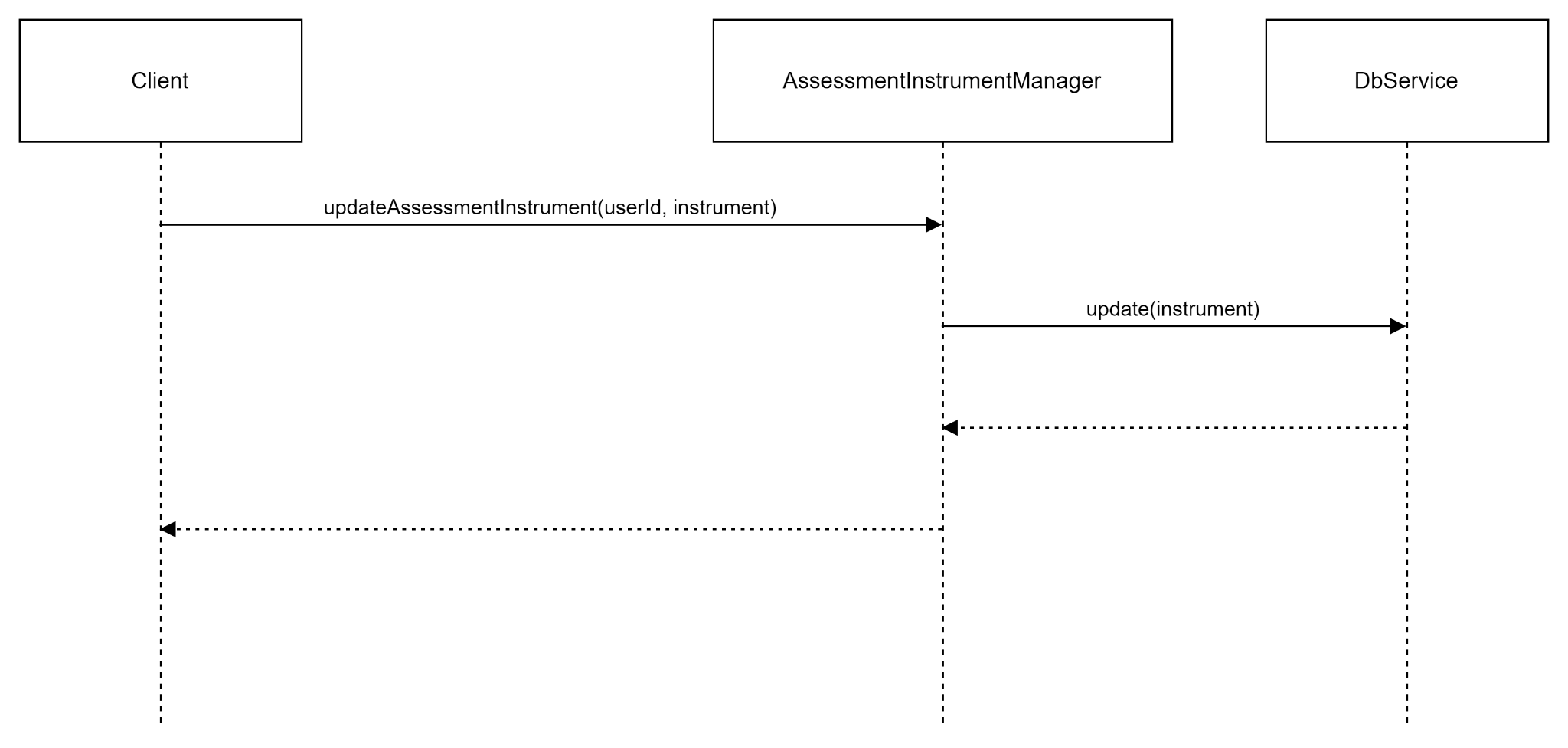
### 

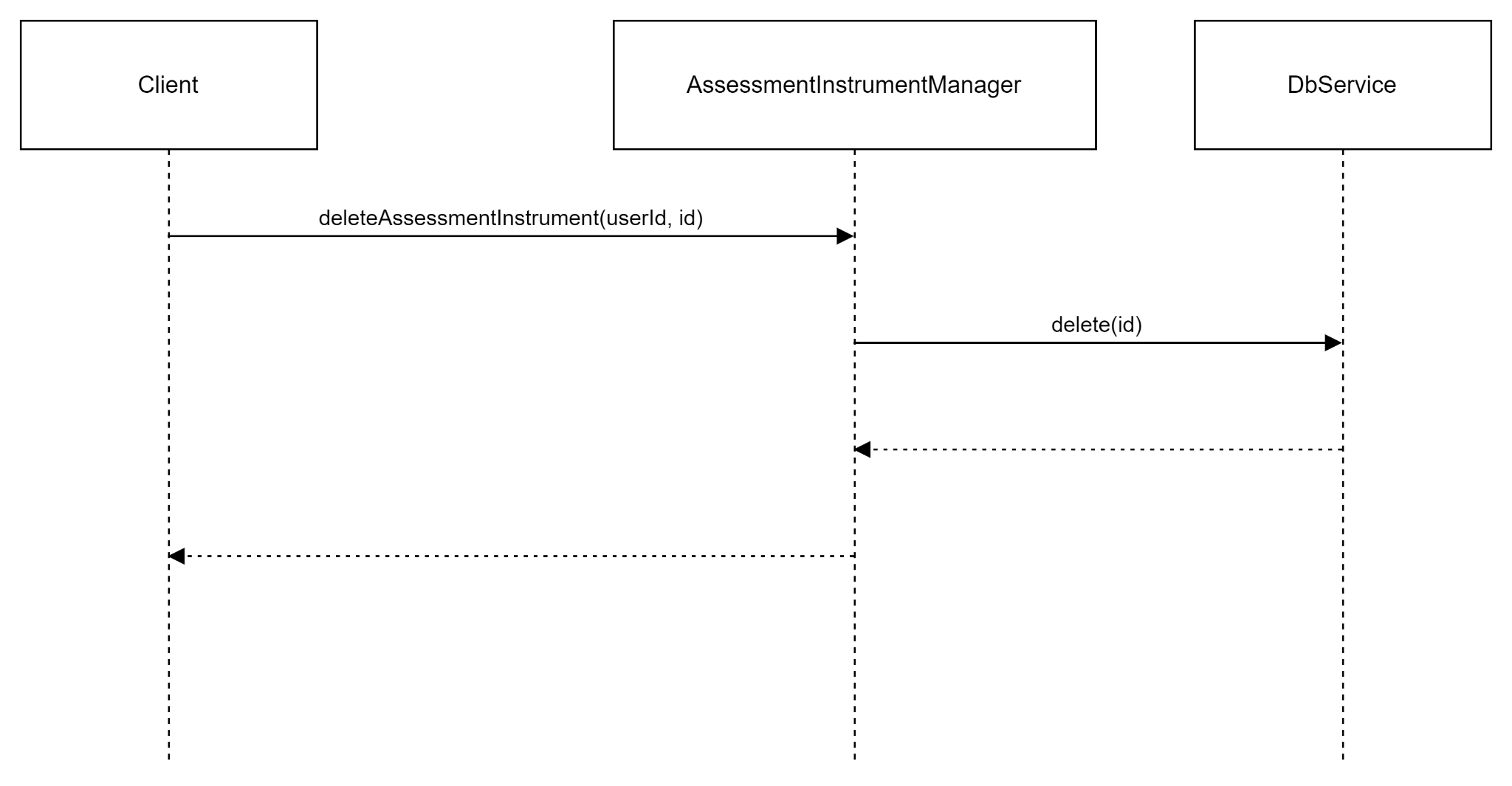
### Remove Item

### Get Assessment Instrument



### Update and Delete Assessment Instrument





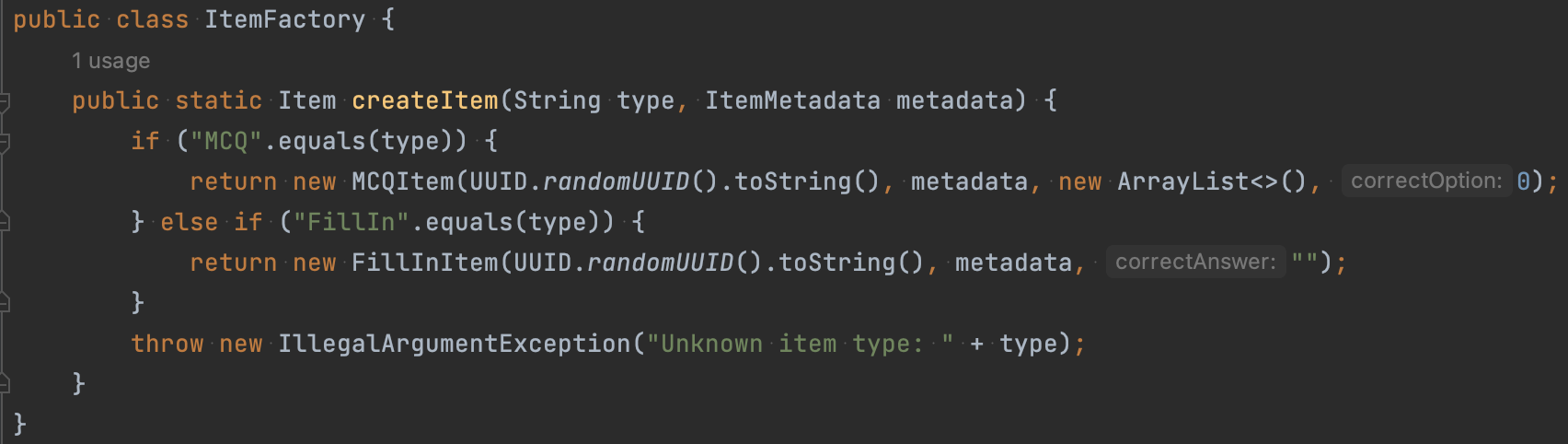
## 

## **Design Patterns**

**1. Factory Method Pattern**

- Used in: *ItemFactory* class

- Justification: The Factory Method pattern is used to create different types of Item objects (MCQ and Fill-In) without specifying their exact classes. This allows for easy extension to new item types in the future without modifying existing code.



**2. State Pattern**

- Used in: *AssessmentInstrumentState* interface and its implementations (*DraftState*, *ReviewState*, *ApprovedState*)

- Justification: The State pattern is used to manage the different states of an assessment instrument (Draft, Review, Approved) and their transitions. This encapsulates state-specific behavior and makes it easier to add new states in the future.



## 

## **Key Design Principles**

1. Single Responsibility Principle (SRP)

- Application: Each class has a single, well-defined responsibility.

- Justification: The *AssessmentInstrumentManager* is responsible for managing assessment instruments, ItemFactory for creating items, and each state class (*DraftState*, *ReviewState*, *ApprovedState*) is responsible for managing behavior in that specific state. It doesn't handle item creation or database operations directly, instead delegating these responsibilities to other classes (*ItemBankManager* and *DbService*)

- Example: The MCQItem and FillInItem classes are responsible only for their specific item types, adhering to SRP.

2. Open/Closed Principle (OCP)

- Application: The design is open for extension but closed for modification.

- Justification: The use of the State pattern (DraftState, ReviewState, ApprovedState) allows new states to be added without modifying existing code. Also, New item types can be added by implementing the Item interface without modifying existing code.

- Example: If a new state like "PublishedState" is needed, it can be added without changing the AssessmentInstrument class. Adding a new item type (e.g., EssayItem) would only require creating a new class implementing Item and adding a case in the ItemFactory, without changing existing item classes.

3. Interface Segregation Principle (ISP)

- Application: Clients are not forced to depend on interfaces they do not use.

- Justification: The Item interface is kept minimal, and specific item types add their own methods as needed. Instead of having a single, large interface, the design separates concerns into different interfaces like *AssessmentInstrumentState*, *Item*, *DbService* and *ItemBankManager*. This allows clients to depend only on the methods they need.

5. Dependency Inversion Principle (DIP)

- Application: High-level modules do not depend on low-level modules; both depend on abstractions.

- Justification: The AssessmentInstrumentManager depends on interfaces rather than concrete implementations.

- Example: The manager uses DbService and ItemBankManager interfaces, allowing for different implementations to be injected without changing the manager's code. This promotes loose coupling and easier testing.

7. Don't Repeat Yourself (DRY)

- Application: Code duplication is minimized throughout the design.

- Justification: Instead of repeating state-specific logic throughout the AssessmentInstrument class, the state pattern centralizes this logic in the respective state classes. This avoids duplicating code for state transitions and state-specific behaviors.

8. Program to an Interface, not to an implementation

- Application: The code consistently uses interfaces for defining contracts between components.

- Justification: The use of interfaces like AssessmentInstrumentState, Item, DbService, and ItemBankManager allows for flexibility in implementations and easier mocking for unit tests.

9. Composition over Inheritance

- Application: The design favors composition for creating complex objects.

- Justification: AssessmentInstrument is composed of metadata and a list of items, rather than inheriting from a base class. This provides more flexibility in object construction and avoids the pitfalls of deep inheritance hierarchies.

This design adheres to these principles to create a flexible, maintainable, and extensible system for managing assessment instruments. The use of design patterns and adherence to these principles allows for easy addition of new features and modifications to existing functionality without significant changes to the core structure of the system.