Tutorial Sheet 5

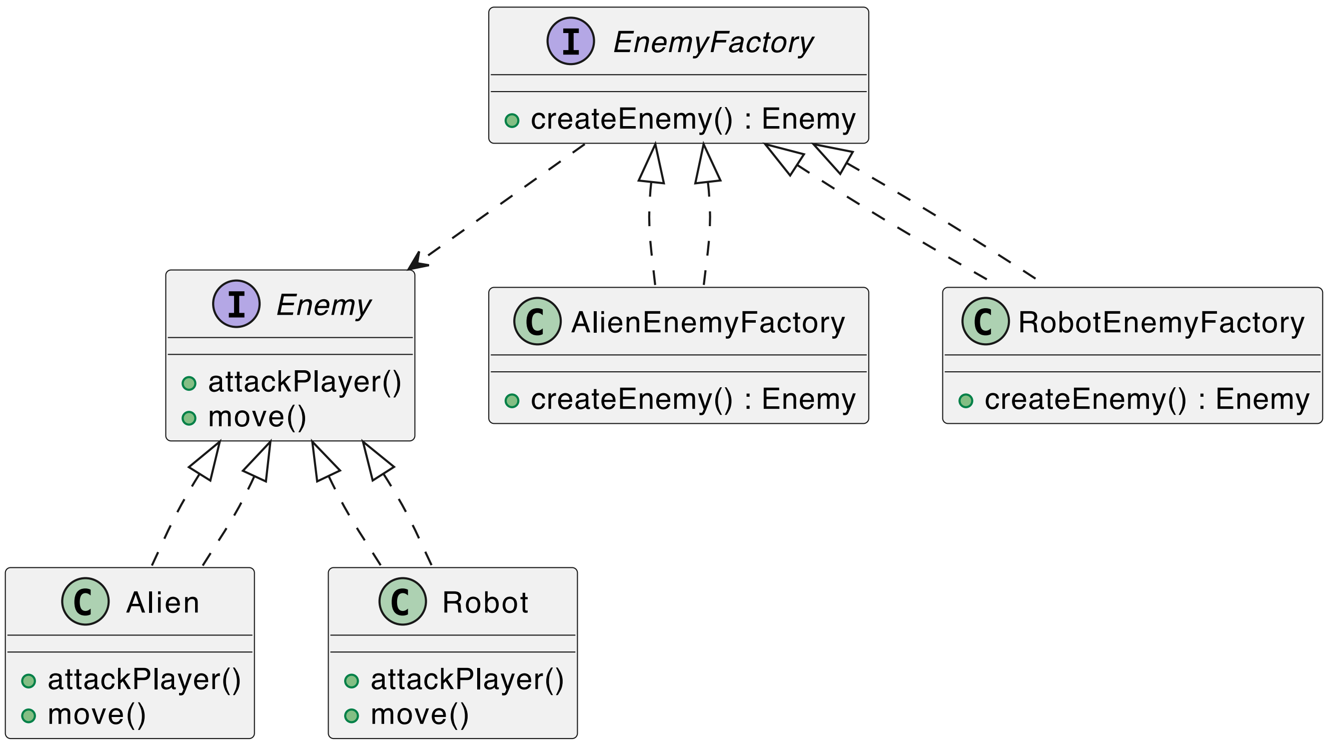
Important: Some of the diagrams may not have rendered perfectly well (there may be multiple arrows), if that is the case, please consult the diagrams from the lecture notes.

1. **Problem Statement: Creating different types of enemies in a game**

a. The problem with the current approach is that it violates the Open/Closed Principle. Whenever a new enemy type needs to be added, the `createEnemy` method in the `EnemyFactory` class needs to be modified, which is a violation of the Open/Closed Principle.

b. The Factory Method design pattern can solve this problem by providing an interface for creating objects (enemies), but allowing subclasses to decide which class to instantiate.

c. UML Diagram:

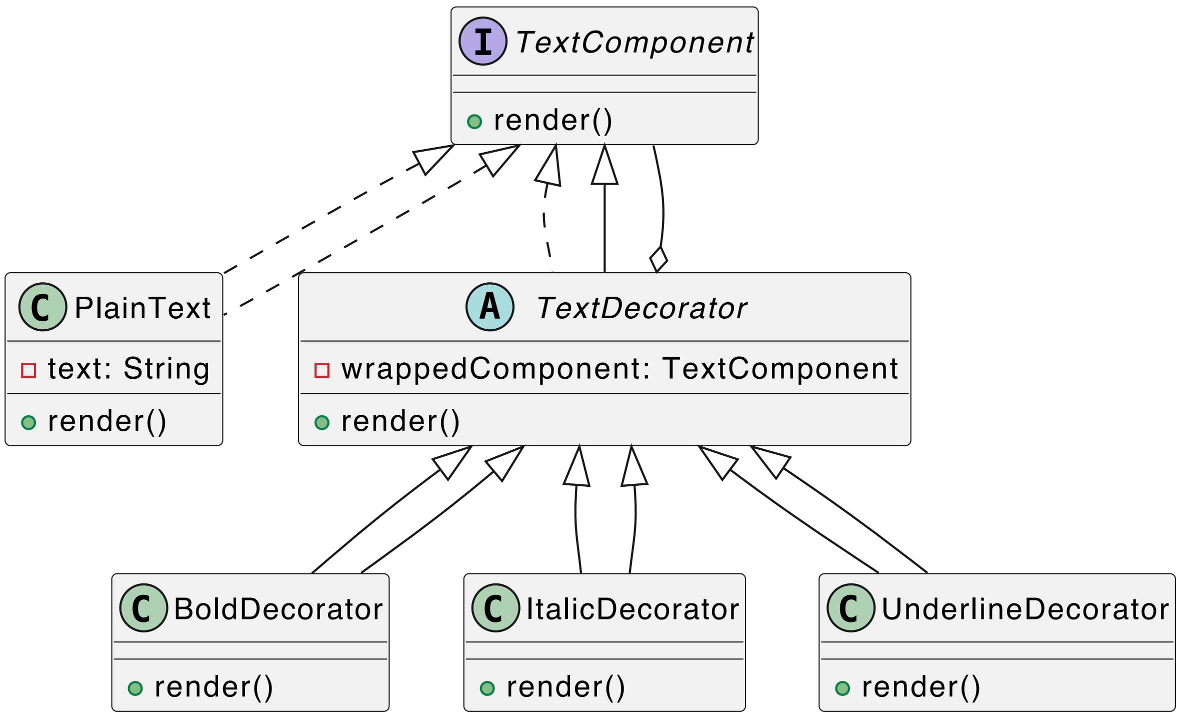


2. **Problem Statement: Text formatting in a text editor application**

a. The issue with the current implementation is that it doesn’t support dynamic composition of formatting options. If we need to create a new formatting option that combines existing options, we would need to create a new class for that specific combination.

b. The Decorator design pattern can solve this problem by allowing dynamic composition of formatting options at runtime.

c. UML Diagram:

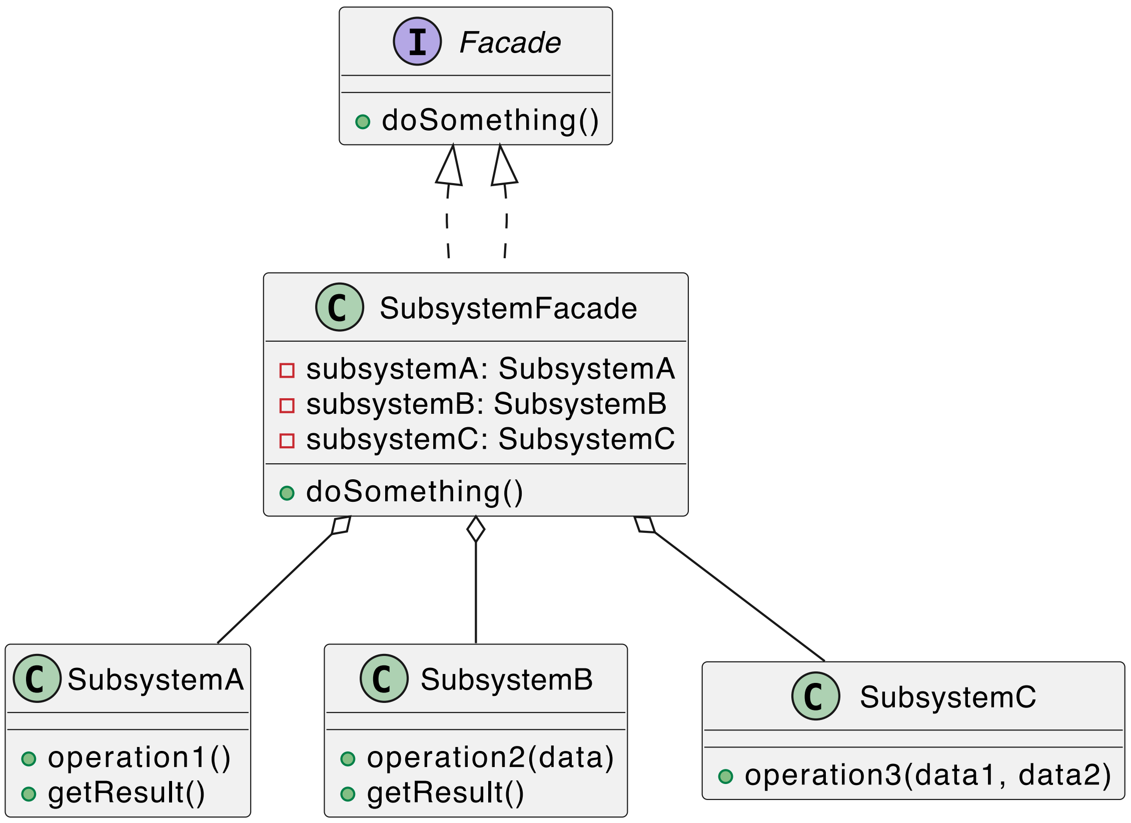


3. **Problem Statement: Complex interactions between subsystems**

a. The problem with the current approach is that the client code is tightly coupled with the implementation details of the subsystems and their interactions. This makes the code difficult to understand, maintain, and extend.

b. The Facade design pattern can solve this problem by providing a simplified interface that hides the complexity of the subsystem interactions from the client code.

c. UML Diagram:



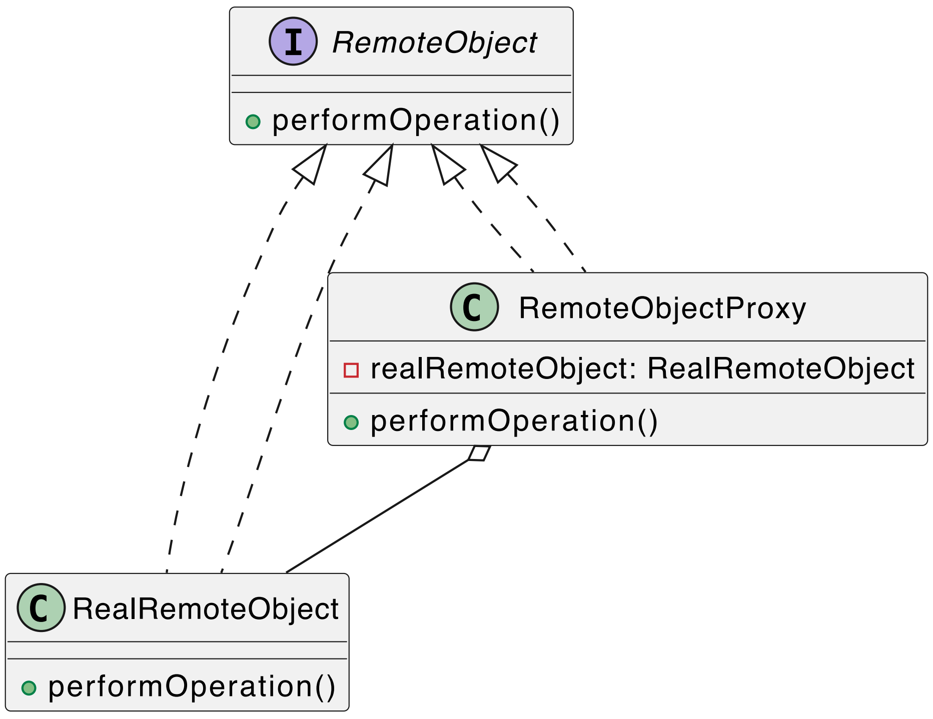
4. **Problem Statement: Accessing remote objects or resources**

a. The potential issues with the current approach are:

- Performance: Accessing remote objects or resources directly can lead to performance issues, especially in distributed systems or over slow networks.
- Availability: If the remote object or resource is not available, the client code will fail.

b. The Proxy design pattern can solve this problem by providing a local representation or placeholder for the remote object or resource.

c. UML Diagram:

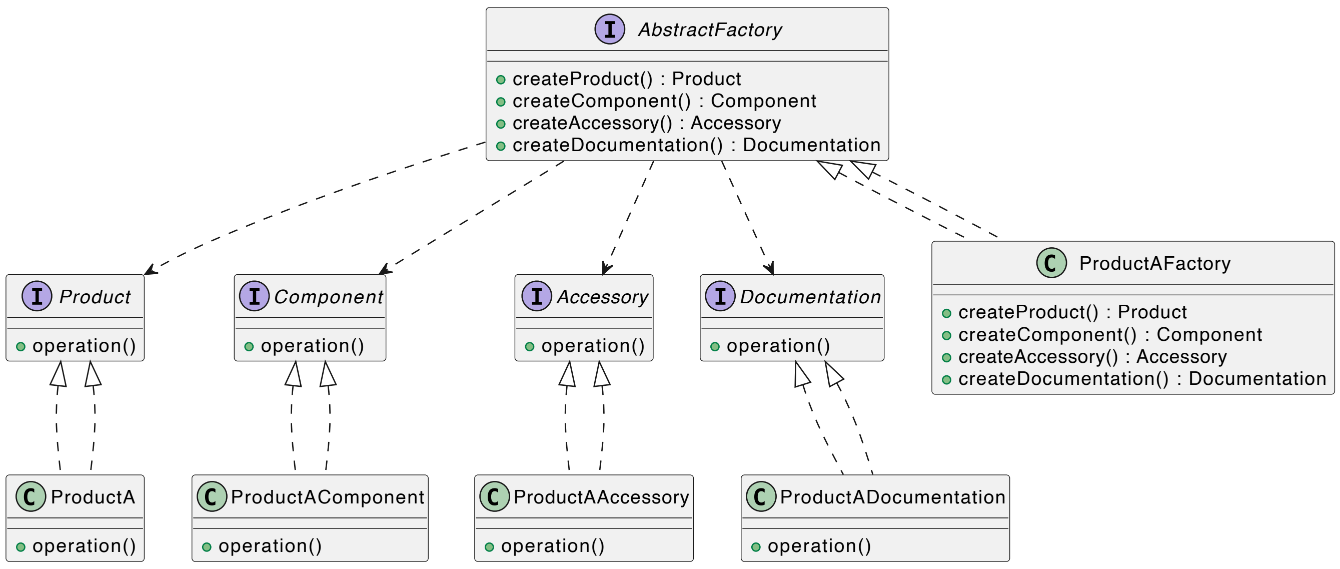


5. **Problem Statement: Creating families of related objects**

a. The potential issue with the current approach is that it violates the Open/Closed Principle. Whenever a new product type needs to be added, multiple classes need to be created or modified (e.g., `ProductC`, `ProductCComponent`, `ProductCAccessory`, `ProductCDocumentation`).

b. The Abstract Factory design pattern can solve this problem by providing an interface for creating families of related objects without specifying their concrete classes.

c. UML Diagram:

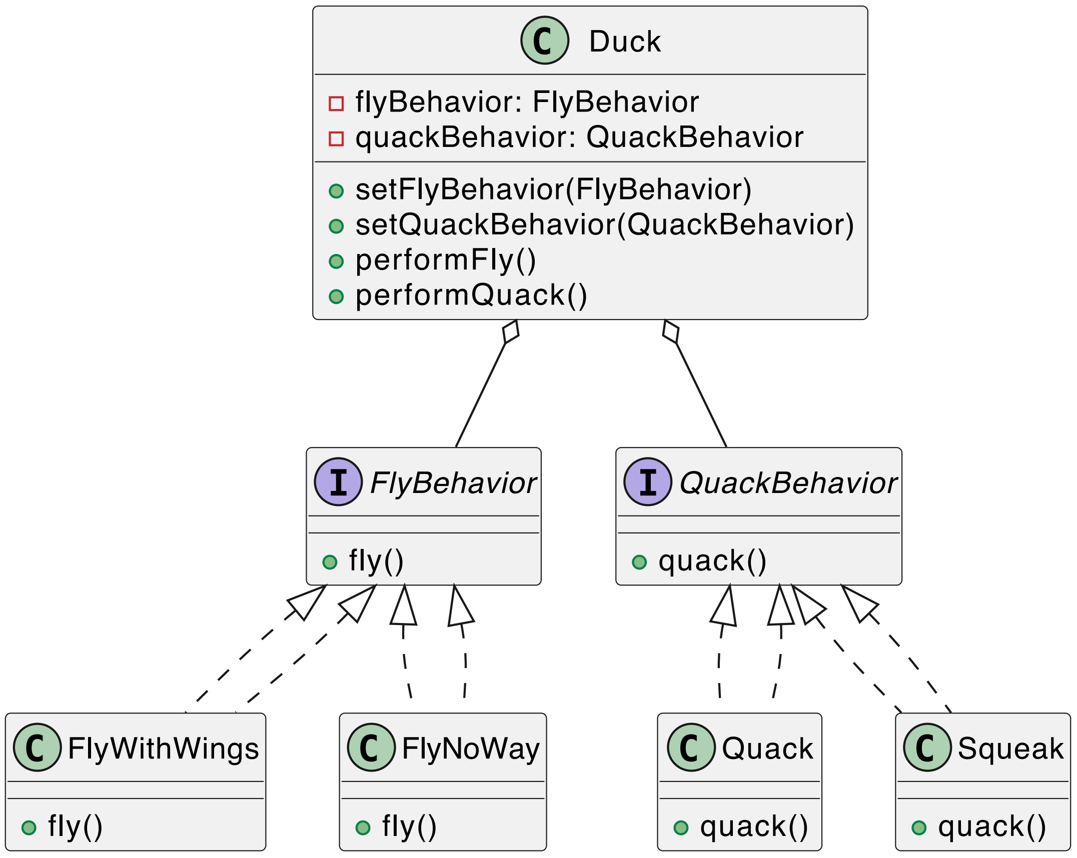

AbstractFactory
+ createProduct() : Product
+ createComponent() : Component
+ createAccessory() : Accessory
+ createDocumentation() : Documentation
Product
+ operation()
Component
+ operation()
Accessory
+ operation()
Documentation
+ operation()
ProductAFactory
+ createProduct() : Product
+ createComponent() : Component
+ createAccessory() : Accessory
+ createDocumentation() : Documentation
ProductA
+ operation()
ProductAComponent
+ operation()
ProductAAccessory
+ operation()
ProductADocumentation
+ operation()

6. **Problem Statement: Implementing different duck behaviors**

a. The potential issue with the current approach is that it violates the Single Responsibility Principle. The `Duck` class is responsible for both the quacking and flying behaviors, which can lead to inflexible and rigid code when new behaviors need to be added or modified.

b. The Strategy design pattern can solve this problem by separating the behavior implementations from the core object (Duck) and allowing them to be easily interchangeable.

c. UML Diagram:

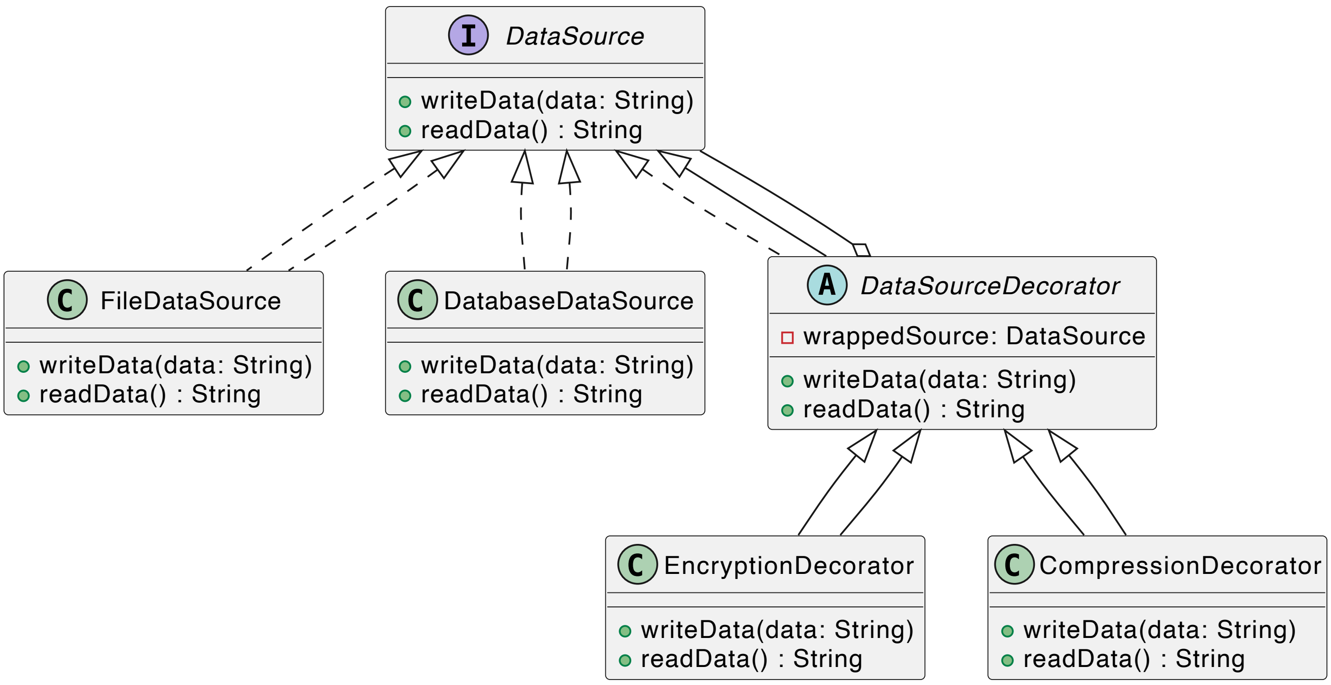


7. **Problem Statement: Managing weather data sources**

a. The potential issue with the current approach is that it violates the Open/Closed Principle. If we need to add new features like encryption or compression to the data sources, we would need to modify the existing `FileDataSource` and `DatabaseDataSource` classes, which can lead to code duplication and increased complexity.

b. The Decorator design pattern can solve this problem by allowing the addition of new features to the data sources dynamically without modifying their existing code.

c. UML Diagram:

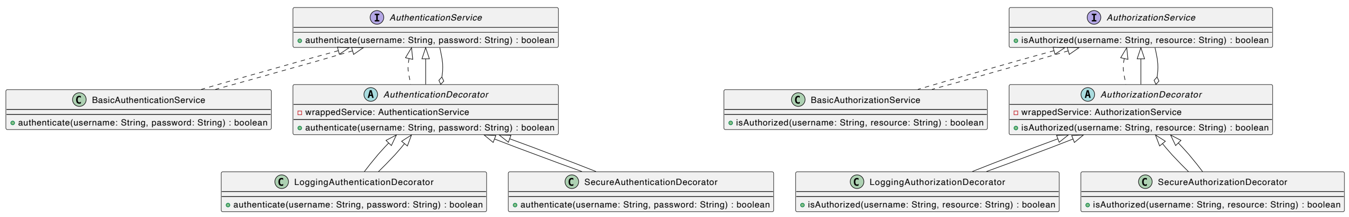


8. **Problem Statement: Managing user authentication and authorization**

a. The potential issue with the current approach is that it violates the Open/Closed Principle. If we need to add additional security checks or logging mechanisms, we would need to modify the existing `BasicAuthenticationService` and `BasicAuthorizationService` classes, which can lead to code duplication and increased complexity.

b. The Decorator design pattern can solve this problem by allowing the addition of new features to the authentication and authorization services dynamically without modifying their existing code.

c. UML Diagram:


AuthenticationService
authenticate(username: String, password: String) : boolean
BasicAuthenticationService
authenticate(username: String, password: String) : boolean
AuthenticationDecorator
wrappedService: AuthenticationService
authenticate(username: String, password: String) : boolean
LoggingAuthenticationDecorator
authenticate(username: String, password: String) : boolean
SecureAuthenticationDecorator
authenticate(username: String, password: String) : boolean
AuthorizationService
isAuthorized(username: String, resource: String) : boolean
BasicAuthorizationService
isAuthorized(username: String, resource: String) : boolean
AuthorizationDecorator
wrappedService: AuthorizationService
isAuthorized(username: String, resource: String) : boolean
LoggingAuthorizationDecorator
isAuthorized(username: String, resource: String) : boolean
SecureAuthorizationDecorator
isAuthorized(username: String, resource: String) : boolean

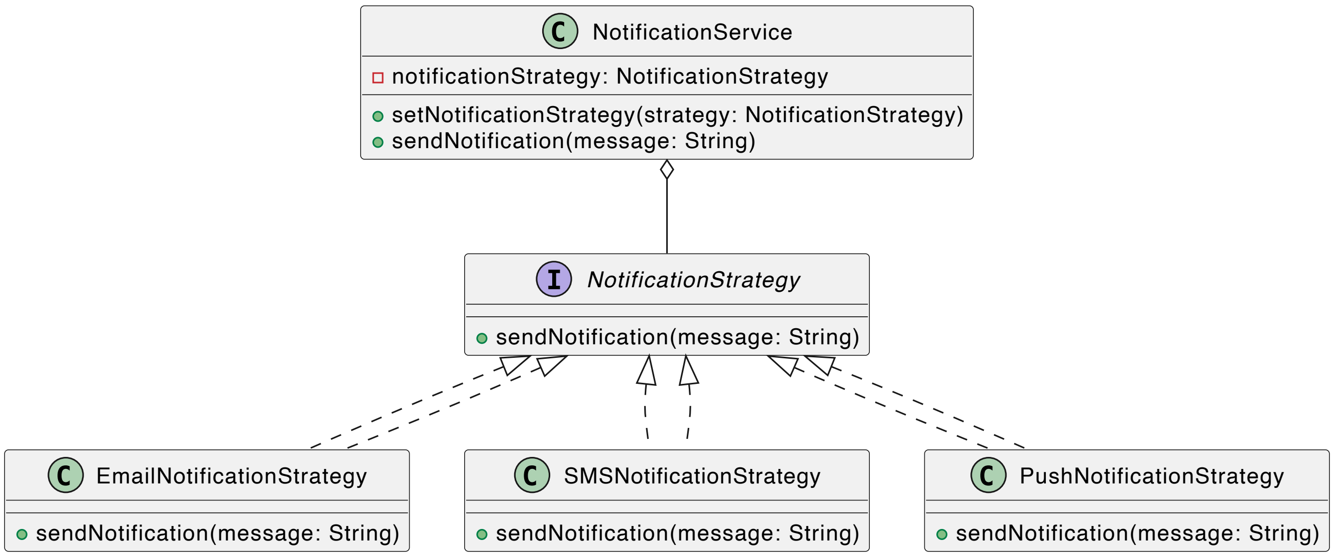
9. **Problem Statement: Managing different types of notifications**

a. The potential issue with the current approach is that it violates the

Open/Closed Principle. If we need to add a new notification channel, we would need to create a new class implementing the `NotificationService` interface, which can lead to code duplication and increased complexity.

b. The Strategy design pattern can solve this problem by separating the different notification strategies from the core `NotificationService` and allowing them to be easily interchangeable.

c. UML Diagram:


NotificationService
notificationStrategy: NotificationStrategy
setNotificationStrategy(strategy: NotificationStrategy)
sendNotification(message: String)
NotificationStrategy
sendNotification(message: String)
EmailNotificationStrategy
sendNotification(message: String)
SMSNotificationStrategy
sendNotification(message: String)
PushNotificationStrategy
sendNotification(message: String)