Deliverable 2 - Report

—

Singleton Design Pattern:

1. AuthenticationManager

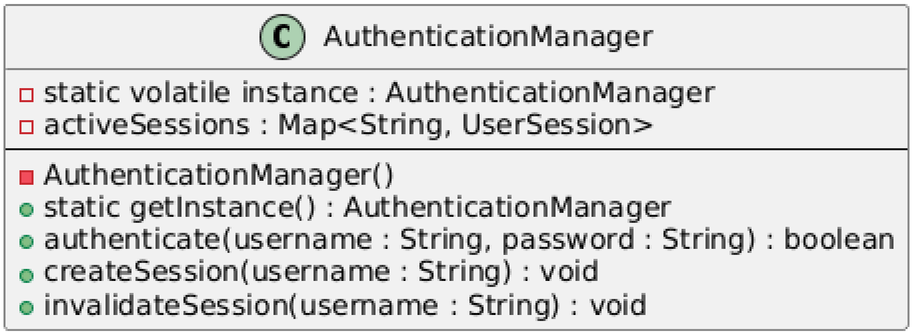
a. Located in the AccountManagement module in the Domain Layer in the Package Diagram.

b. Reason: The authentication service should maintain a single instance to manage all user authentications across the system. This ensures consistent security practices and session management. **Multiple Instances could lead to:**

   i. Conflicting session management & Inconsistent login/logout processes

   ii. Unnecessary memory consumption & Potential race conditions in session tracking

c. UML:



```
(C) AuthenticationManager
□ static volatile instance : AuthenticationManager
□ activeSessions : Map<String, UserSession>
■ AuthenticationManager()
● static getInstance() : AuthenticationManager
● authenticate(username : String, password : String) : boolean
● createSession(username : String) : void
● invalidateSession(username : String) : void
```

2. AuditDB Access Controller

a. Located in the Data Access Layer in the Package Diagram.

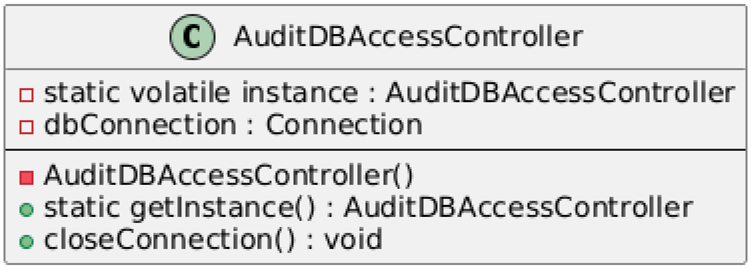
b. Reason: Audit logging must be centralized and consistent across the entire application (e.g., login attempts, medical record modifications). A Singleton ensures all system actions are logged through the same controller, maintaining a complete and consistent audit trail for regulatory compliance (HIPAA standards). **Multiple Instances could:**

   i. Generate conflicting or duplicate entries.

   ii. Performance overhead due to multiple database connection attempts.

   iii. Difficulty in analysis.

c. UML:


C AuditDBAccessController
□ static volatile instance : AuditDBAccessController
□ dbConnection : Connection
■ AuditDBAccessController()
● static getInstance() : AuditDBAccessController
● closeConnection() : void

Factory Method Design Pattern:

1. DatabaseConnectionFactory

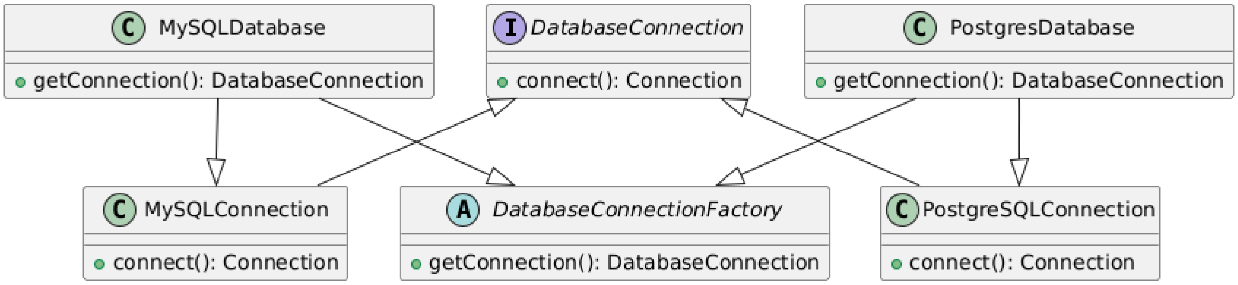
d. Located in the Data Access Layer in the Package Diagram.

e. Reason: The application requires a centralized mechanism for creating database connections dynamically. Using the Factory Method ensures:

   i. Encapsulation of database connection logic.

   ii. Flexibility to support multiple database types in the future.

f. UML:


C MySQLDatabase
+ getConnection(): DatabaseConnection
I DatabaseConnection
+ connect(): Connection
C PostgresDatabase
+ getConnection(): DatabaseConnection
C MySQLConnection
+ connect(): Connection
A DatabaseConnectionFactory
+ getConnection(): DatabaseConnection
C PostgreSQLConnection
+ connect(): Connection