CS 319 Object-Oriented Software Engineering

Deliverable 5

Final Class Diagram / Design Patterns

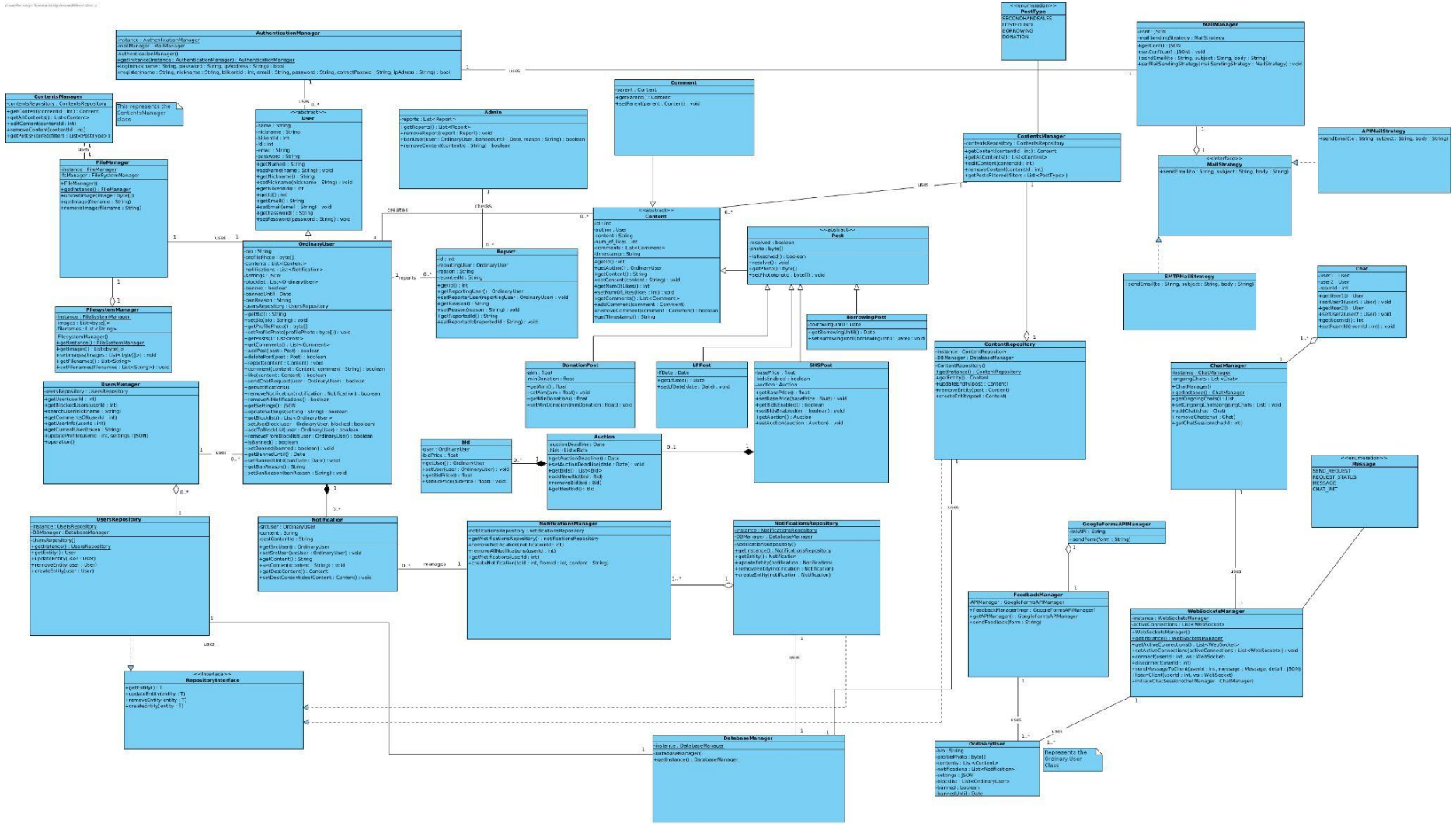
S2T8

Bilginer Oral	Ercan Bahri Nazlıoğlu	Alper Biçer	Mohamed Amine Boukattaya	Eren Karakaş
22103163	21903151	22003097	22301131	22002722

1. Final Object Design **3**
2. Design Patterns **4**
2.1 Singleton Design Pattern **4**
2.2 Repository Design Pattern **6**
2.3 Strategy Design Pattern **7**

1. Final Object Design

Following is the revised object design of the system, exercising the appropriate design patterns:



2. Design Patterns

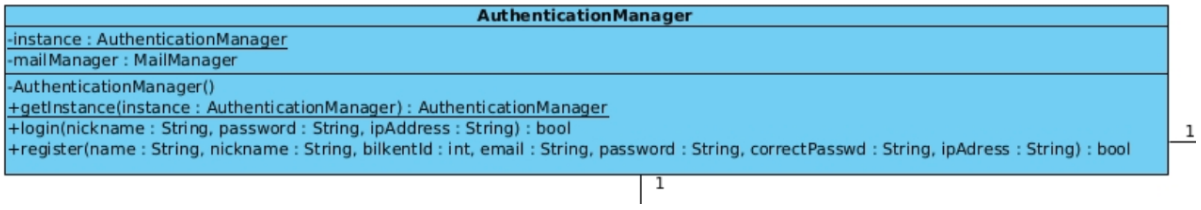
2.1 Singleton Design Pattern

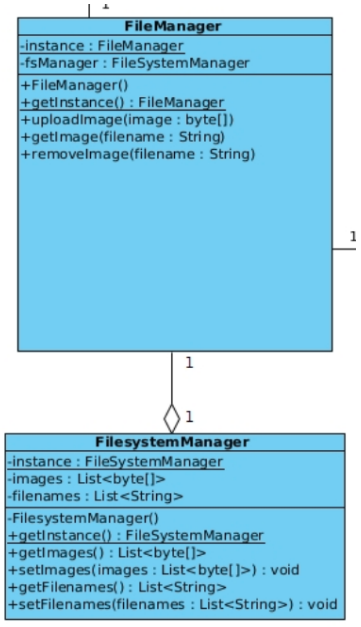
We have opted to use the singleton design pattern for the following classes:

- Authentication Manager
- FileManager
- FileSystemManager
- UsersRepository
- NotificationsRepository
- ContentsRepository
- ChatManager
- WebSocketsManager
- DatabaseManager

These classes will have only one instance throughout the uptime of the system. This choice provides several advantages in terms of resource utilization, consistency, and centralized control.

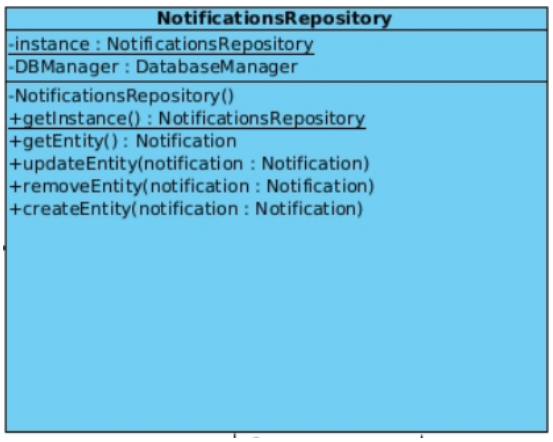
Authentication Manager:
The Authentication Manager uses the singleton pattern by ensuring that only one instance exists at any given time. This guarantees a consistent authentication state across the entire application, preventing potential conflicts and security vulnerabilities.

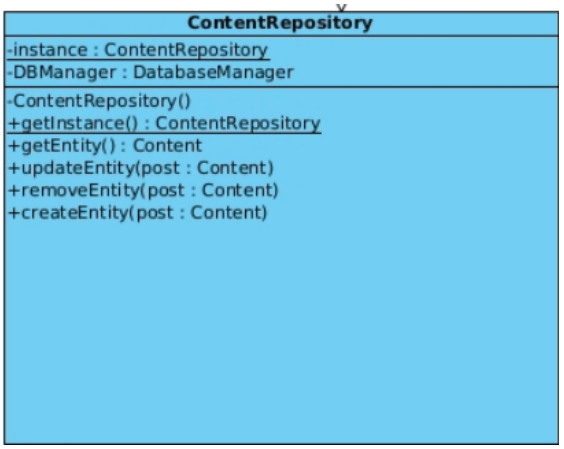


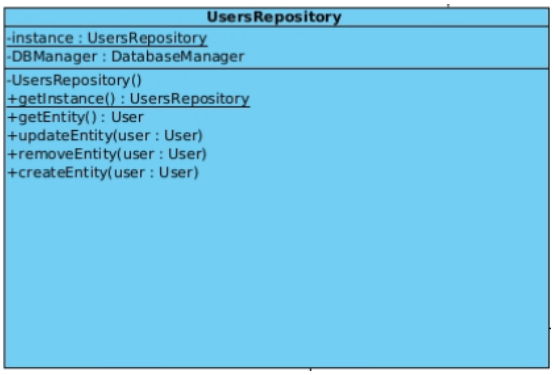


FileManager and FileSystemManager:
Singleton instances of the FileManager and FileSystemManager are used to control file operations and interactions with the file system. This ensures the prevention of concurrency issues and reduces resource utilization.

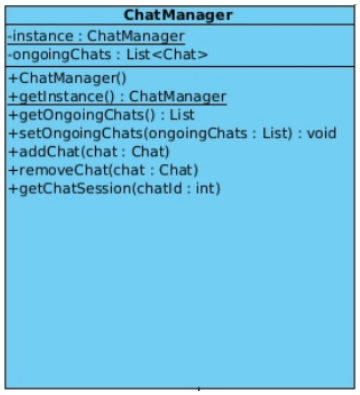
UsersRepository, NotificationsRepository, and ContentsRepository:
These repository classes play an important role in handling user data, notifications, and content storage/access. By implementing the singleton pattern, a single, shared instance guarantees data consistency.

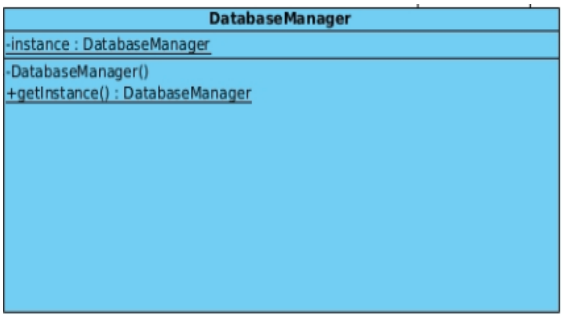






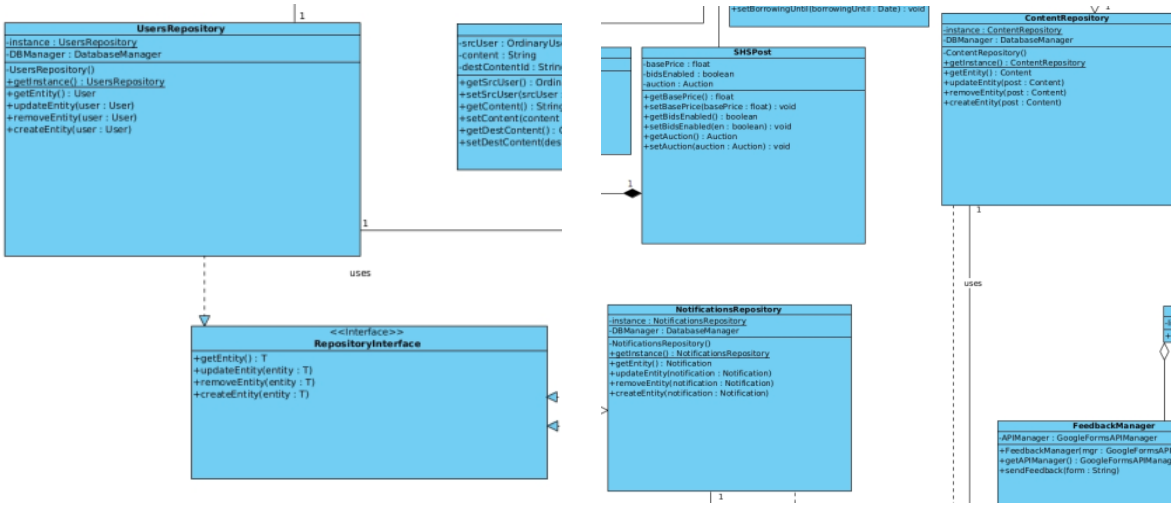
ChatManager and WebSocketsManager:
For real-time communication and WebSocket handling, the singleton pattern is applied to the ChatManager and WebSocketsManager. This approach ensures that there is only one instance managing the communication channels, preventing conflicts, and providing a centralized point.





DatabaseManager:
The DatabaseManager, responsible for database interactions, is designed as a singleton to maintain a single connection throughout the server's lifecycle. This optimizes resource utilization and ensures that database transactions are efficiently and securely managed, rejecting multiple accesses simultaneously.

2.2 Repository Design Pattern

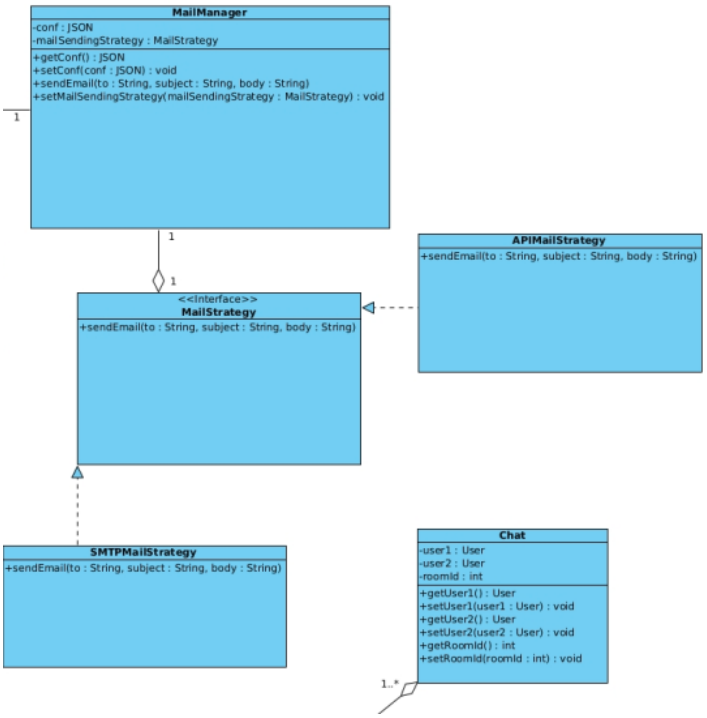


In the architecture of our application, we have strategically employed the repository design pattern to manage data access and storage for key entities within the system. This design pattern is particularly employed in the UsersRepository, ContentsRepository, and NotificationsRepository classes, all of which implement the RepositoryInterface to ensure a unified and consistent approach to handling common functionalities related to data management. These classes also use the DatabaseManager class (which has only a singleton object) in order to maintain database accesses.

The repository design pattern abstracts away the complexities of data access, allowing the rest of the application to interact with data through a high-level interface. Furthermore, this abstraction makes it easier to adapt to changes in the storage system or migrate to a different database technology.

By implementing a shared interface, our repository classes enable code reusability across different entities, offering a more maintainable codebase. Any changes or optimizations to common data access methods can be applied universally through the interface, reducing redundancy and the likelihood of introducing errors during updates. This centralization enhances the organization of our codebase, making it clear where and how data operations are performed.

2.3 Strategy Design Pattern



In our application, we have decided to make use of the strategy design pattern to handle various email-sending algorithms effectively. For this purpose, we introduced the MailStrategy interface, which serves as a blueprint for implementing different email-sending strategies.

Two concrete classes have been developed to implement the MailStrategy interface: SMTPMailStrategy and APIMailStrategy. Each of these classes provides its own algorithm for sending emails to users. This modular approach allows us to extend and customize email-sending behaviors in the future easily.

The main component responsible for managing emails is the MailManager class. This class relies on a MailStrategy instance to determine which strategy to employ for sending emails. By decoupling the strategy implementation from the core email management logic, we aim to improve the overall flexibility and maintainability of our software.