UE21CS352B:
Object Oriented Analysis and Design with Java

Online Freelance Marketplace

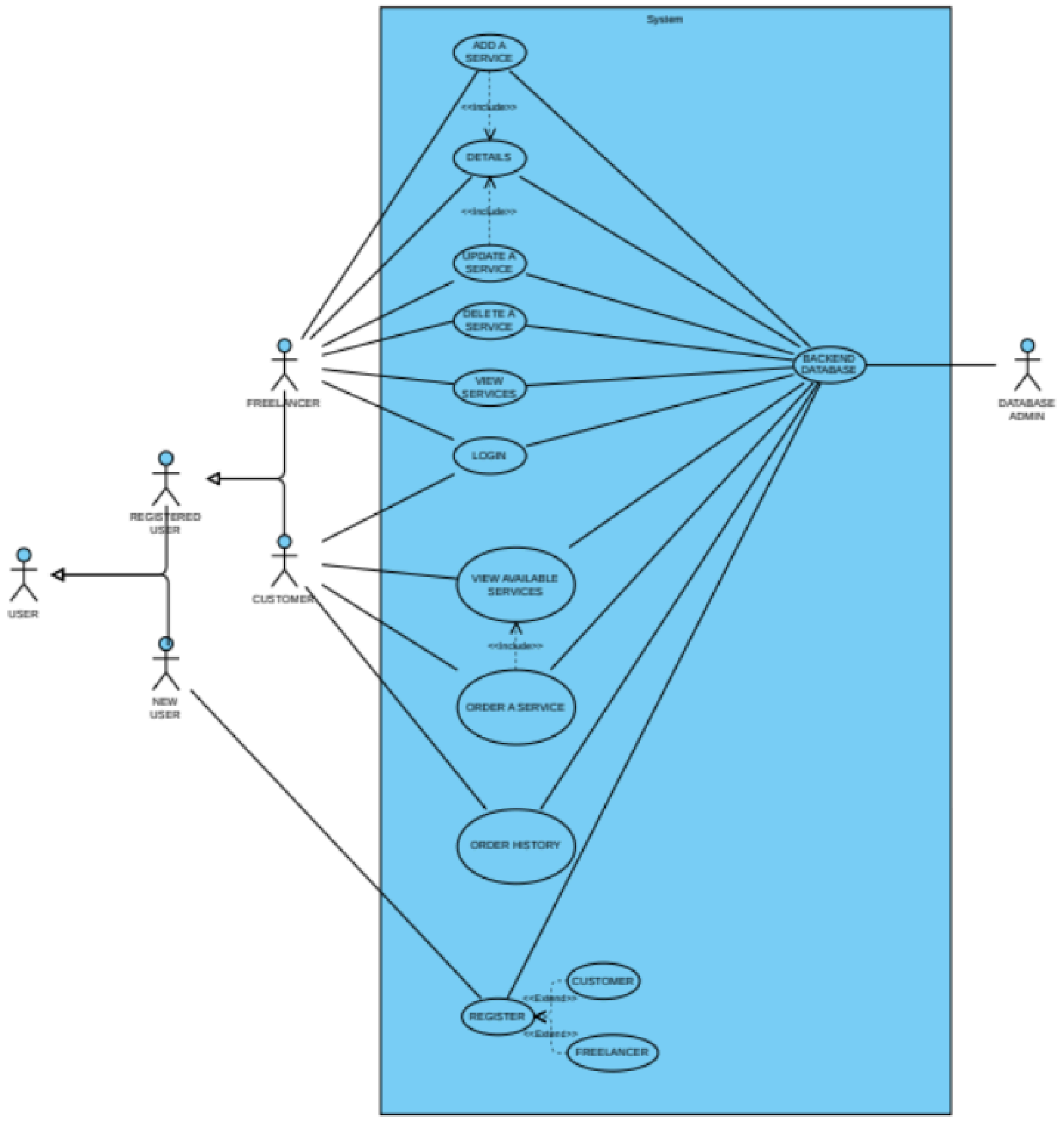
Project Report

K Ganesh Vaidyanathan
PES1UG21CS253
K Siddharth Rao
PES1UG21CS265
M S Varun
PES1UG21CS309
Kousthub R Menon
PES1UG21CS284

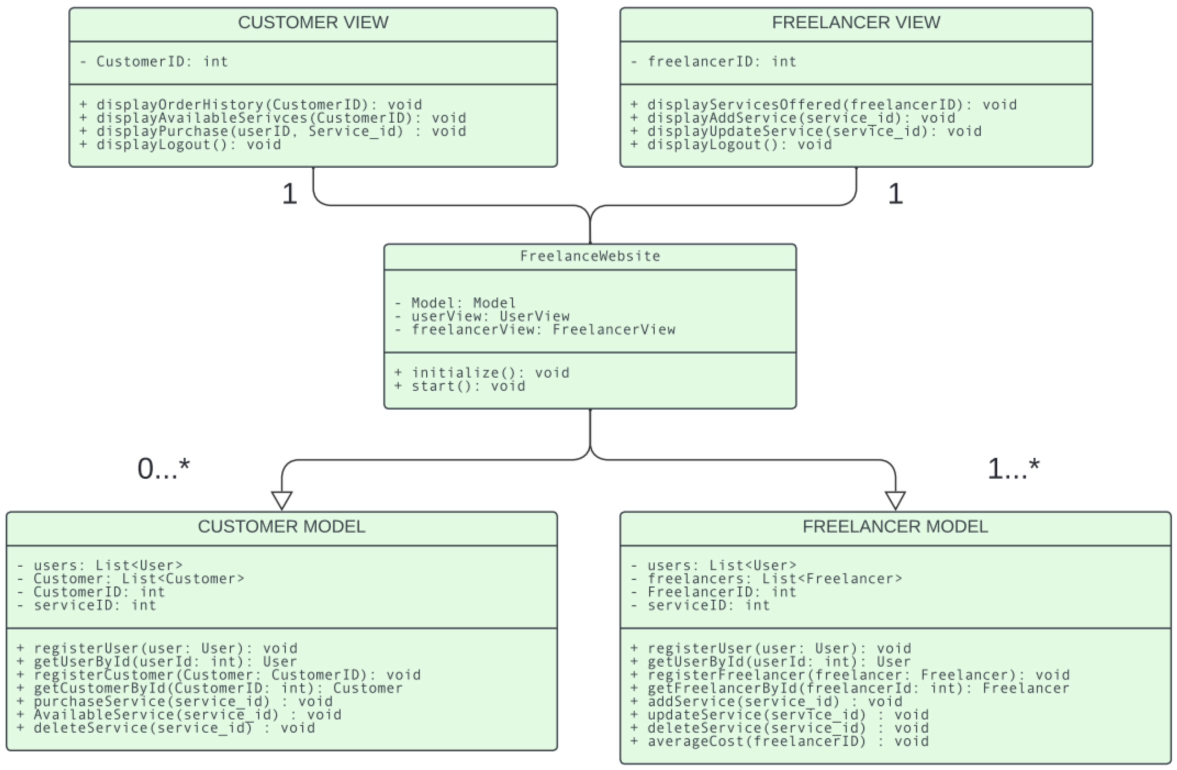
Mentor: Prof. Bhargavi Mokashi

Table of Contents

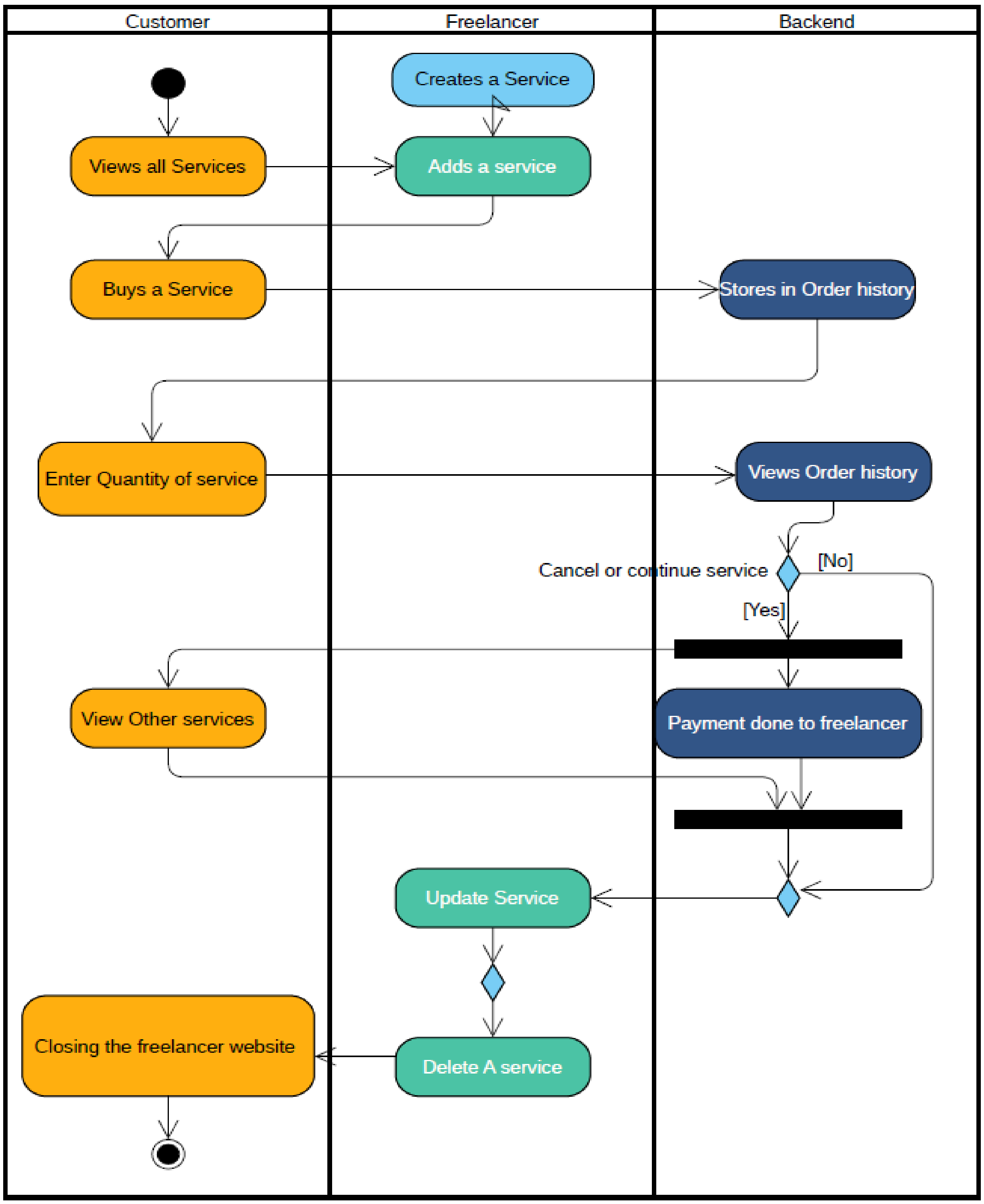
1. Use Case Diagram



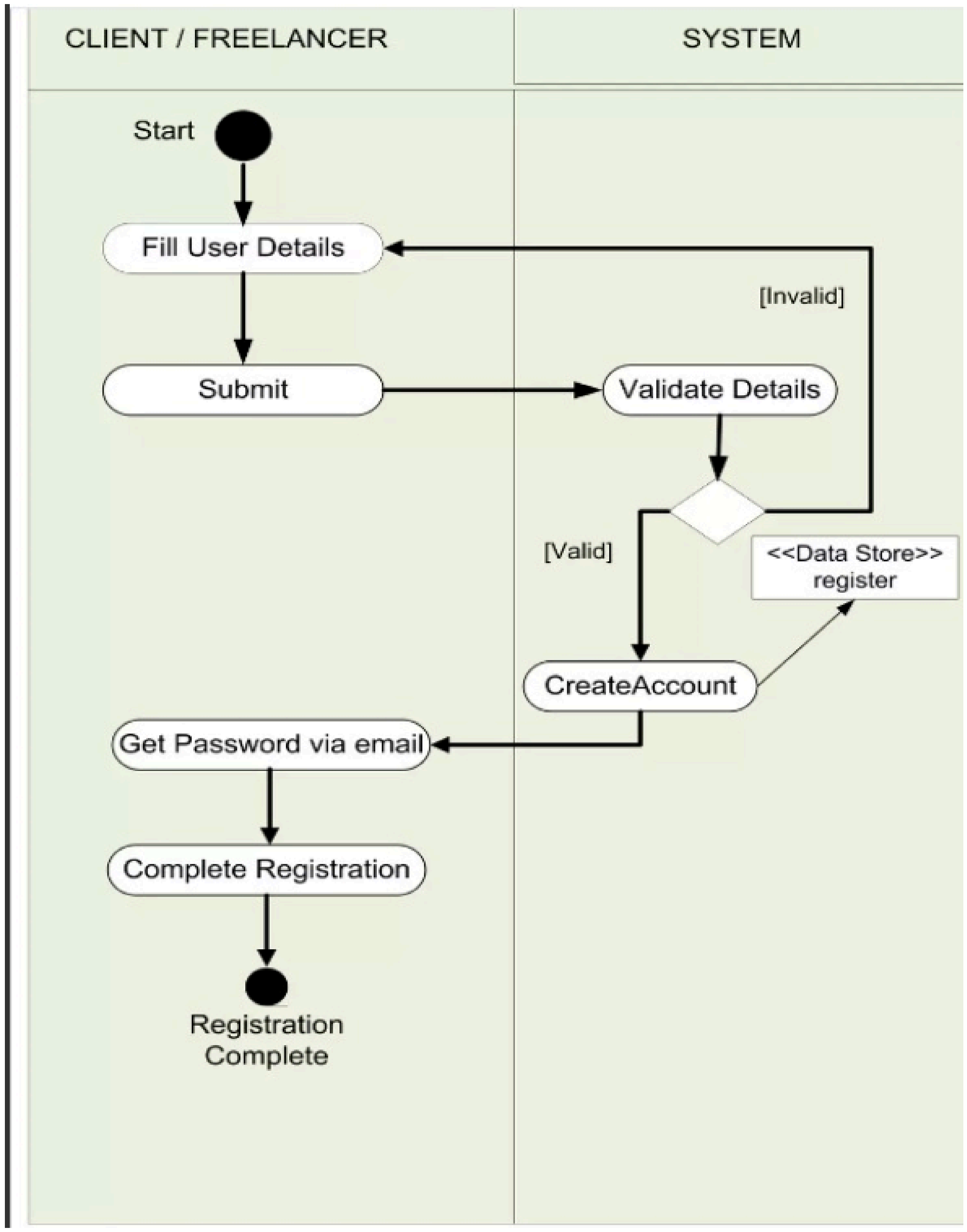
2. Class Diagram


CUSTOMER VIEW
- CustomerID: int
+ displayOrderHistory(CustomerID): void
+ displayAvailableSerivces(CustomerID): void
+ displayPurchase(userID, Service_id) : void
+ displayLogout(): void
FREELANCER VIEW
- freelancerID: int
+ displayServicesOffered(freelancerID): void
+ displayAddService(service_id): void
+ displayUpdateService(service_id): void
+ displayLogout(): void
1
1
FreelanceWebsite
- Model: Model
- userView: UserView
- freelancerView: FreelancerView
+ initialize(): void
+ start(): void
0...*
1...*
CUSTOMER MODEL
- users: List<User>
- Customer: List<Customer>
- CustomerID: int
- serviceID: int
+ registerUser(user: User): void
+ getUserById(userId: int): User
+ registerCustomer(Customer: CustomerID): void
+ getCustomerById(CustomerID: int): Customer
+ purchaseService(service_id) : void
+ AvailableService(service_id) : void
+ deleteService(service_id) : void
FREELANCER MODEL
- users: List<User>
- freelancers: List<Freelancer>
- FreelancerID: int
- serviceID: int
+ registerUser(user: User): void
+ getUserById(userId: int): User
+ registerFreelancer(freelancer: Freelancer): void
+ getFreelancerById(freelancerId: int): Freelancer
+ addService(service_id) : void
+ updateService(service_id) : void
+ deleteService(service_id) : void
+ averageCost(freelancerID) : void

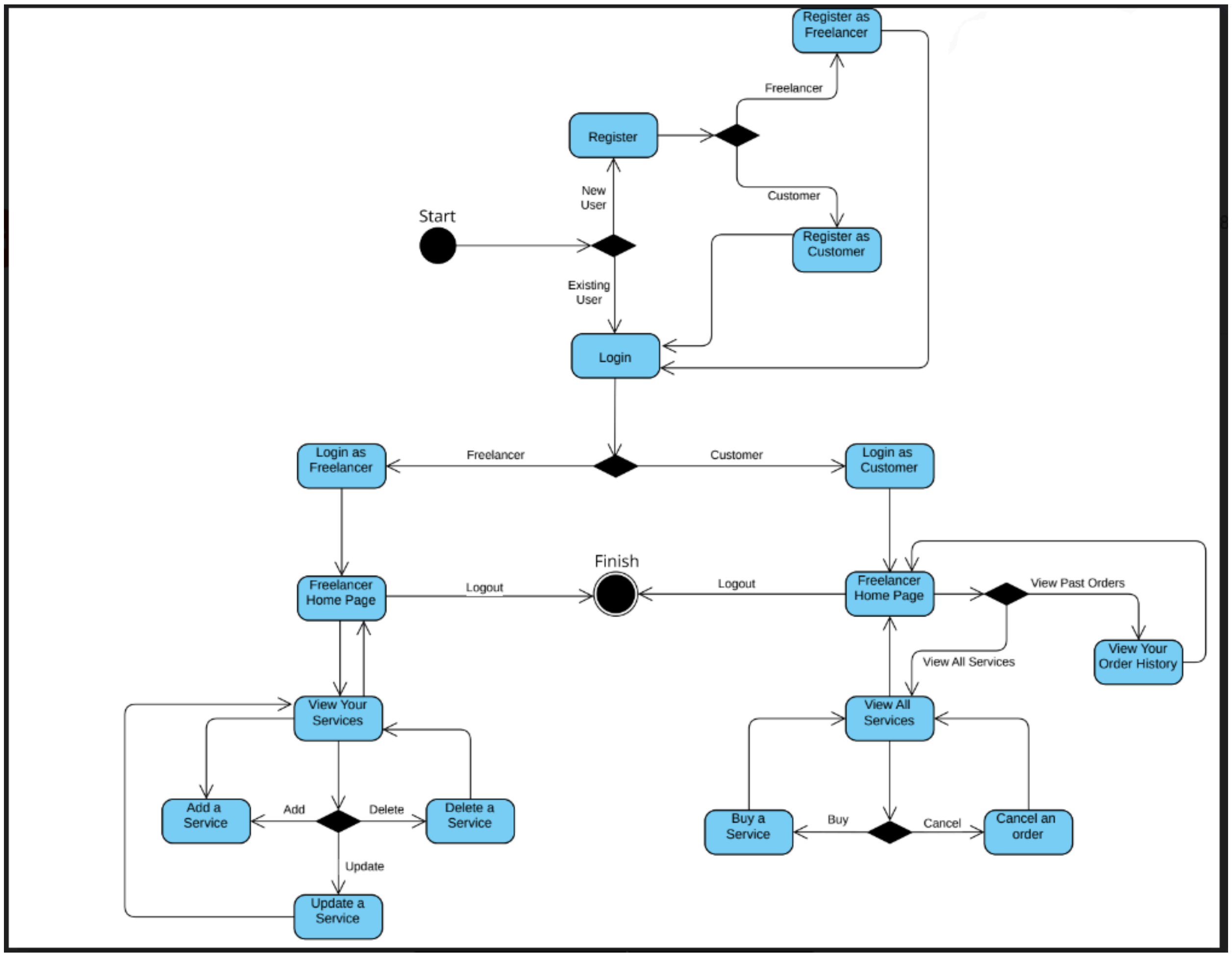
3. Service Operations(Major Use Case) - Activity Diagram



4. Customer/Freelancer Registration (Minor Use Case) - Activity Diagram



5. State Diagram



Factory Pattern: We employ the factory pattern to streamline the creation of services offered by freelancers, ensuring a structured approach to service instantiation and management within our system.

Data Transfer Object: By utilizing the DTO pattern, we centralize service data storage, facilitating efficient transfer and manipulation of information across various components of our application as needed.

Adapter Pattern: Leveraging Spring Boot's JPA repository classes, we seamlessly convert Java code into database entities and relationships, enabling compatibility with diverse backend databases like H2, MySQL, or PostgreSQL.

Singleton Pattern: Through the implementation of a singleton, we ensure the existence of only one instance of the "factory" product, promoting resource efficiency and consistent access within our system.

Single Responsibility Principle (SRP): Our system adheres to SRP by organizing controllers into distinct functionalities such as login, registration, purchase, and operations for both freelancers and customers, promoting clear code structure and maintainability.

High Cohesion: We design our system components with high cohesion, ensuring that each module or class focuses on a specific task or responsibility, leading to better code readability, reusability, and easier maintenance.

Low Coupling: By minimizing interdependencies between modules or classes, we achieve low coupling, enhancing the flexibility, scalability, and testability of our system, while reducing the risk of unintended side effects during modifications or updates.

Creator Principle: we emphasize the importance of encapsulating object creation logic, promoting code clarity, flexibility, and ease of extension, thereby facilitating the evolution of our system over time.

Controller Principle: we delegate control logic to specialized controller components, ensuring a clear separation of concerns and promoting modularization, which enhances code organization and simplifies maintenance.