Part 1: Use Case Definition for 'Register for a Class'

Use-Case Description: Register for a Class

Name:

Register for a Class

Primary Actor:

Student

Goal:

Allow a student to register for a class offering during the active registration period.

Scope:

Course Registration System

Preconditions:

System State:

- The registration period must be active.
- Course offerings must be available and published.
- The course must not have reached its maximum capacity.

User State:

- The student must be authenticated in the system.
- The student must have sufficient credits available for registration.

Postconditions:

Successful Execution:

- The student is enrolled in the selected course offering.
- The course enrollment count is updated.
- A confirmation notification is sent to the student.

Unsuccessful Execution:

- Relevant error messages are displayed for issues such as unmet prerequisites, capacity limitations, or expired registration periods.
- No changes are made to the student or course offering states.

Domain Rules:

Rule 1: Prerequisites

- A student must complete all prerequisite courses before enrolling in a specific course offering.

Rule 2: Enrollment Capacity

- Each course offering has a maximum capacity. Once this capacity is reached, further enrollments are blocked unless a waitlist is enabled.

Rule 3: Active Registration Period

- Registration is only allowed during the officially designated registration period.

Rule 4: Student Eligibility

- The student must have sufficient credits to complete the registration process.

Explanation of Rule Integration:

- Main Success Scenario: Step 2 explicitly checks Rules 1, 2, and 4 during the validation process.
- Alternate Flows: Alternate Flow 3 explicitly handles Rule 3, validating the registration period.

Part 2: Tactical Design Documentation

a. Definitions of Tactical Pattern Objects

a.1. Entities

1. Student (Part of the Student Aggregate)
    - Attributes:
        - studentId (Unique Identifier)
        - name
        - email
        - status
    - Behaviors:

        - isEligibleForRegistration() – Validates if the student meets eligibility requirements.
        - registerForClass(Enrollment enrollment) – Adds a course to the student's active enrollments.
2. Enrollment (Part of the Student Aggregate)
    - Attributes:
        - enrollmentId (Unique Identifier)
        - courseOfferingId
        - status (e.g., Enrolled, Waitlisted, Dropped)
    - Behaviors:
        - markAsWaitlisted() – Updates the status to waitlisted.
        - markAsEnrolled() – Updates the status to enrolled.
3. Course Offering (Part of the Course Aggregate)
    - Attributes:
        - courseOfferingId (Unique Identifier)
        - courseId
        - semester
        - capacity (Maximum enrollments)
        - enrolledCount (Current number of students enrolled)
    - Behaviors:
        - canAcceptEnrollment() – Checks if the course offering has available capacity.
        - incrementEnrollmentCount() – Updates the count of enrolled students.

a.2. Aggregates

1. Student Aggregate
    - Root Entity: Student
    - Other Entities: Enrollment
    - Boundary Justification:
      Ensures consistency in the student's registration process, such as validating eligibility and managing their active enrollments.
2. Course Aggregate
    - Root Entity: Course Offering
    - Boundary Justification:
      Maintains integrity of course offerings, ensuring enrollment limits and prerequisite enforcement are consistently applied.

a.3. Domain Services

1. RegistrationService
    - Purpose: Handles behaviors involving multiple aggregates, such as registering a student for a class.
    - Responsibilities:
        - Validate prerequisites using the Course Aggregate.
        - Verify student eligibility using the Student Aggregate.

        - Update Course Offering and Enrollment entities accordingly.
    - Methods:
        - registerStudentForClass(studentId, courseOfferingId)

a.4. Value Objects

1. Prerequisite
    - Attributes:
        - courseId
    - Description: Represents a single prerequisite course that must be completed before enrolling.
2. Semester
    - Attributes:
        - startDate
        - endDate
    - Description: Represents the semester during which the course offering is available.

a.5. Domain Events

1. ClassRegistered
    - Attributes:
        - studentId
        - courseOfferingId
        - timestamp
    - Trigger: Raised when a student successfully registers for a course offering.
2. ClassWaitlisted
    - Attributes:
        - studentId
        - courseOfferingId
        - timestamp
    - Trigger: Raised when a student is added to the waitlist for a course offering.

b. Justification for Aggregate Boundaries

b.1. Student Aggregate

- Ensures all interactions with the student (e.g., registration, eligibility validation) occur in a controlled and consistent manner.
- The Enrollment entity is part of this aggregate as it directly represents the student's participation in courses and must remain consistent with their state.

b.2. Course Aggregate

- Encapsulates the Course Offering entity to manage enrollment counts and prerequisite validations.
- Ensures consistency by controlling access to course-specific rules, such as capacity checks.

b.3. Consistency Justification

- The boundaries align with the domain rules specified in Part 1:
  - Prerequisites: Enforced via the Course Aggregate.
  - Enrollment Capacity: Managed directly by the Course Offering entity.
  - Student Eligibility: Ensured within the Student Aggregate.
- The RegistrationService acts as a mediator for cross-aggregate interactions, preventing tight coupling and adhering to DDD principles.

Part 3: Design the Use Case:

1. UML Sequence Diagram

a)

Key Components in the Diagram:

1. Actors & Layers:
   - Actor: Student (initiates the registration request).
   - Application Layer: RegisterForClassController receives and forwards the request to the domain layer.
   - Domain Model Layer: Handles business logic through RegistrationService, Student Aggregate, and Course Aggregate.
   - Infrastructure Layer: Uses repositories and the Message Bus for persistence and communication.
2. Flow Steps:
   - Step 1: Student sends a registration request to the controller.
   - Step 2: The controller calls RegistrationService.registerStudentForClass.
   - Step 3: The service:
     - Retrieves the Student and CourseOffering aggregates from the repositories.
     - Validates domain rules (e.g., prerequisites, capacity, eligibility).
     - Updates aggregates (Enrollment and CourseOffering state changes).
   - Step 4: Aggregates are saved atomically using the Unit of Work pattern.
   - Step 5: A ClassRegisteredEvent is published to the Message Bus for eventual consistency.
3. Domain Rules Applied:
   - Prerequisites: Checked in the CourseAggregate.
   - Enrollment Capacity: Managed within CourseOffering.
   - Eligibility: Verified within StudentAggregate.

b)

1. Interaction Between Layers:
    - The sequence diagram will demonstrate interactions between:
        - Application Layer: Handles user requests and coordinates domain layer actions.
        - Domain Model Layer: Contains domain logic and enforces rules.
        - Infrastructure Layer: Manages persistence and communication between bounded contexts.
2. Roles of Domain Objects:
    - Student Aggregate: Validates the student's eligibility and manages their enrollment.
    - Course Aggregate: Ensures course-specific rules (e.g., capacity checks).
    - Domain Services: Coordinates actions across aggregates.
3. Port and Adapter Pattern:
    - The controller and repositories act as adapters, ensuring the domain logic remains decoupled from external frameworks or technologies.
4. GRASP Patterns:
    - Controller: RegisterForClassController is responsible for handling the use case logic.
    - Information Expert: RegistrationService determines which domain objects hold necessary data for validation.
    - Creator: RegistrationService creates a new Enrollment entity.
5. Unit of Work:
    - Ensures atomicity by grouping operations into a single transaction in the RegistrationService.
6. Eventual Consistency:
    - If the ClassRegistered event needs to trigger actions in other bounded contexts (e.g., notifying the billing system), the event is published via a Message Bus.

2. Eventual Consistency

Message Bus Integration:

- The ClassRegisteredEvent triggers workflows in other contexts, such as:
    - Notification System: Sends a confirmation to the student.
- By using the Message Bus, these workflows remain decoupled from the core registration logic, enhancing scalability.

3.Explanation of Design Decisions

i)

The Port and Adapter Pattern (also known as Hexagonal Architecture) is used to decouple the domain logic from external systems like databases or message buses. In this design:

1. Ports:
    - Application Layer Ports: RegisterForClassController acts as the input port for handling student registration requests.
    - Domain Services: Expose interfaces that define the domain logic without relying on specific infrastructure implementations.
2. Adapters:
    - Infrastructure Adapters: Repositories (Database Repository) and the Message Bus handle the actual persistence and communication logic.
    - These adapters implement the interfaces defined in the domain, ensuring the domain layer does not depend on infrastructure-specific details.

By adhering to this pattern:

- The domain model remains pure and focused solely on business rules and logic.
- Infrastructure components can be easily swapped or updated (e.g., changing the database technology) without impacting the domain layer.

ii)

The GRASP (General Responsibility Assignment Software Patterns) principles guide responsibility assignment in this use case:

1. Controller:
    - The RegisterForClassController mediates between the user interface (Student) and the domain logic.
    - It ensures the domain layer is not concerned with handling user requests directly.
2. Information Expert:
    - The RegistrationService is the expert in orchestrating the registration process since it has access to both Student Aggregate and Course Aggregate.
    - Each aggregate (e.g., Student, Course) is responsible for validating rules specific to its context.
3. Creator:
    - The RegistrationService creates a new Enrollment entity as it holds the necessary information (student and course data).
4. High Cohesion & Low Coupling:
    - Responsibilities are distributed across layers and components, reducing dependencies and maintaining modularity.

iii)

The Unit of Work Pattern ensures that all operations related to the registration process are executed atomically. This pattern is applied in the RegistrationService during the registration process:

1. Scope:
    - The retrieval, validation, updates to aggregates, and persistence actions are grouped as a single unit.
2. Atomicity:
    - If any operation (e.g., capacity check, eligibility validation) fails, the entire process is rolled back, leaving the system in a consistent state.
3. Implementation:
    - The repositories handle persistence within the unit of work. Once all validations and updates are complete, the unit of work commits the changes to the database.

Summary of Design Rationale

1. Port and Adapter Pattern ensures clear separation of concerns and adaptability to infrastructure changes.
2. GRASP Patterns promote clarity in responsibility assignment, ensuring each component performs a well-defined role.
3. Unit of Work Pattern guarantees consistency and reliability in the registration process, particularly in a distributed or concurrent environment.

4.Write Code of the Application Service Class:

a) -

b)

i. Coordinate between the layers

i.Repositories

Explanation:
Repositories serve as an abstraction over the data persistence layer. They provide methods to retrieve, save, and update domain objects (e.g., entities or aggregates) without exposing the underlying database operations.

Interaction in the Code:

- Fetching domain objects:
  The service retrieves the Student and CourseOffering objects using their respective repositories:

  ```
  student = self.student_repository.get_by_id(student_id)
  course_offering = self.course_offering_repository.get_by_id(course_offering_id)
  ```

- Saving updated objects:
  After modifying the state of the student or course offering (e.g., registering a student, updating a waitlist), the updated objects are saved back to the repositories:

  ```
  self.student_repository.save(student)
  self.course_offering_repository.save(course_offering)
  ```

Purpose:
The application service delegates persistence-related tasks to repositories, ensuring that it remains focused on coordination and not database operations, adhering to the Repository Pattern.

ii. Domain Objects (Aggregates, Entities, and Value Objects)

Explanation:
The domain objects encapsulate the business logic and rules of the application. The service interacts with them to perform validations and execute domain-specific behaviors.

- Entities: Represent uniquely identifiable objects, such as Student and CourseOffering. These entities maintain their state and lifecycle.
- Aggregates: Groups of entities that form a consistent boundary for operations. For example, a CourseOffering can be considered an aggregate, with its enrollment and waitlist forming part of its boundary.
- Value Objects: Represent immutable data without identity, such as a Semester in the provided code.

Interaction in the Code:

- Validating Business Rules:
  The service calls methods on domain objects to check eligibility and prerequisites:

  ```
  if not student.is_eligible_for_registration():
      return RegistrationResult.failure("Student is not eligible for registration")

  if not course_offering.validate_prerequisites(student.completed_courses):
      return RegistrationResult.failure("Prerequisites not met")
  ```

- Executing Domain Behaviors:
  The service invokes methods to modify the state of domain objects, like registering a student or adding them to a waitlist:

enrollment = student.register_for_class(course_offering_id, semester)
course_offering.add_student(student_id)

Purpose:
This interaction ensures that the business logic remains in the domain objects, while the service orchestrates workflows. This aligns with the Domain-Driven Design (DDD) principle of keeping logic within aggregates and entities.

iii. External Systems Through Adapters

Explanation:
Adapters provide a bridge between the application and external systems, ensuring that communication is abstracted and the application remains decoupled. These could include messaging systems, external APIs, or other services.

Interaction in the Code:

- The message_bus serves as an adapter to publish events (e.g., StudentRegisteredEvent or StudentWaitlistedEvent) to external systems:

  self.message_bus.publish(registration_event)

  This might notify other systems like:

  - Email or notification services: To inform students or administrators about registration or waitlist status.
  - Analytics services: To log registration events for reporting purposes.
  - Other microservices: To trigger actions in dependent systems.

Purpose:
The adapter pattern ensures that the application service interacts with external systems in a loosely coupled manner, improving maintainability and scalability. If the external system changes, only the adapter needs modification, not the application service itself.

Explanation of How the Application Service Operates(for deliverables)

The RegisterStudentForClassService application service is a central component in the registration workflow, designed to manage and coordinate all the necessary steps to successfully register a student for a class. It serves as a mediator that bridges the gap between different layers of the application architecture, ensuring a smooth flow of information and operations. Specifically, it interacts with repositories to handle data persistence and retrieval, with domain objects to enforce business logic and rules, and with external systems to publish domain events or notify other services. By encapsulating this workflow, the service ensures that the overall process remains cohesive, modular, and easy to maintain, adhering to the principles of separation of concerns and single responsibility.

5.Design Report

i)

1. Repository Pattern:
    - StudentRepository and CourseRepository abstract the storage and retrieval logic of domain objects.
    - This ensures the application service is not tied to specific data access logic, promoting flexibility and testability.
2. Aggregate Pattern:
    - Student and CourseOffering are treated as aggregates, encapsulating their internal consistency rules (e.g., enrolling a student in a course).
3. Domain Event Pattern:
    - ClassRegisteredEvent indicates the successful completion of the registration process.
    - This event decouples domain logic from external actions, such as notifications or logging.
4. Service Layer Pattern:
    - RegisterForClassService acts as the application service layer, coordinating the registration process, interacting with repositories and domain logic, and publishing domain events.

ii)

1. Domain Events:
    - Events like ClassRegisteredEvent facilitate eventual consistency by enabling other subsystems (e.g., notifications) to asynchronously react to changes.
    - In distributed systems, tools like RabbitMQ or Kafka can handle event propagation and maintain consistency.

iii)

1. Maintainability:
    - Clear separation of concerns between the application service, domain objects, repositories, and event publishers makes the codebase easier to extend and maintain.
    - For example, repository implementations can change (e.g., switching databases) without affecting other layers.
2. Scalability:
    - New features, such as additional domain logic or event types, can be added with minimal changes to the existing code.
    - The event-driven design supports future integrations with external systems.
3. Separation of Concerns:
    - Validation, domain logic, and persistence responsibilities are distinct, ensuring clean code and minimizing coupling.
    - Business rules remain within the domain layer, isolated from application logic.

5.Explanation of the Code

1.Repositories

- The student_repository and course_offering_repository are used to fetch Student and CourseOffering objects by their unique identifiers:

  ```
  student = self.student_repository.get_by_id(student_id)
  course_offering = self.course_offering_repository.get_by_id(course_offering_id)
  ```

- These repositories act as abstractions over the persistence layer, allowing the service to interact with the underlying database without directly coupling to database operations.
- After making modifications to the domain objects (e.g., registering a student or updating the course capacity), the repositories save the changes:

  ```
  self.student_repository.save(student)
  self.course_offering_repository.save(course_offering)
  ```

2. Domain Objects

- The application service interacts with domain objects (entities like Student and CourseOffering) to enforce business rules:
  - Checks if the student is eligible for registration:

    ```
    if not student.is_eligible_for_registration():
        return RegistrationResult.failure("Student is not eligible for registration")
    ```

  - Ensures that course prerequisites are met:

    ```
    if not course_offering.validate_prerequisites(student.completed_courses):
        return RegistrationResult.failure("Prerequisites not met")
    ```

  - Executes operations like enrolling the student or adding them to a waitlist:

    ```
    enrollment = student.register_for_class(course_offering_id, semester)
    course_offering.add_student(student_id)
    ```

- By delegating logic to the domain objects, the service ensures that operations like checking prerequisites or updating capacity remain consistent and encapsulated within the domain layer.

3. Domain Service (if applicable)

- Although your code does not explicitly use a domain service, its functionality is embedded within the domain objects themselves. For example:
    - The Student and CourseOffering objects handle business logic directly, such as is_eligible_for_registration or can_accept_enrollment.
- In a more traditional setup, a domain service could encapsulate complex operations that involve multiple entities, like checking prerequisites across courses or managing waitlist priorities.

4.Unit of Work

- The unit_of_work is used to manage transactions, ensuring that all operations are treated as a single atomic unit:
    - Beginning a Transaction: Before making any changes to the database, a transaction is started:

      self.unit_of_work.begin_transaction()

    - Rolling Back: If an error occurs during registration, all changes are reverted to ensure the database remains consistent:

      self.unit_of_work.rollback()

    - Committing Changes: On successful completion of the registration process, the changes are saved to the database:

      self.unit_of_work.commit()

5.Event Bus

- The message_bus is responsible for publishing domain events to notify external systems:
    - After successful registration, a StudentRegisteredEvent is published:

      self.message_bus.publish(registration_event)

    - If the student is added to a waitlist, a StudentWaitlistedEvent is published:

      self.message_bus.publish(waitlist_event)

- These events enable communication with external systems (e.g., sending email notifications, updating analytics systems) while keeping the service decoupled from these external components.
- By relying on asynchronous event processing, the service can remain lightweight and focused on the registration workflow.