EGE UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT
OBJECT ORIENTED ANALYSIS AND DESIGN

HOMEWORK-3

PREPARED BY

Team NO: 31

05210000261-Bahrihan Torpil

05210000238-Mehmet Ali Avcı

05210000260-Kutlu Çağan Akın

Part 1: Use Case – Register for a Class

1- DEFINING "REGISTER FOR A CLASS" USE CASE

Primary Actor

- **Student**: The individual attempting to register for a class.

Goal

- Successfully register the student for a selected course offering while ensuring all rules and constraints are satisfied.

Preconditions

1. The student is an active user in the system.
2. The course registration period is open.
3. The course offering has available seats.
4. The student meets the prerequisites for the selected course (if applicable).
5. The student has no financial or academic holds preventing registration.

Main Success Scenario

1. The student logs into the system and navigates to the course registration page.
2. The system displays a list of available course offerings for the current semester.
3. The student selects a course to register for.
4. The system validates:
    - The course has available seats.
    - The student meets the prerequisites for the course.
    - The student is eligible for registration.
5. If all validations pass:
    - The system creates an enrollment record linking the student to the course offering.
    - The course offering's available seats are reduced by one.
    - A confirmation message is displayed to the student.

Alternative Flow

- If the student wants to register for multiple courses:
    1. The process repeats for each course the student selects.
    2. The system ensures validation for every selected course individually.

Exception Flow

1. **E1**: No available seats in the course.
    - The system displays an error: "This course offering is full."
2. **E2**: Prerequisites not met.
    - The system displays an error: "You do not meet the prerequisites for this course."
3. **E3**: Academic or financial hold.
    - The system displays an error: "You have a hold preventing registration."

Postconditions

1. **If Successful**:
    - The student is added to the course offering.
    - The available seats in the course offering are reduced by one.
    - The enrollment record is created, associating the student with the course offering.
    - A domain event ClassRegistered is triggered for further notifications (e.g., sending confirmation emails).
2. **If Unsuccessful**:
    - The system displays an appropriate error message explaining why the registration could not be completed.
    - No changes are made to the system state.

2- DOMAIN RULES

Domain Rules:

1. **Capacity Rule:**
    - A student cannot register for a course offering if there are no available seats.
2. **Prerequisite Rule:**
    - A student can only register for a course if all prerequisite courses are completed.
3. **Eligibility Rule:**
    - A student must be eligible to register (no academic or financial holds).
4. **Unique Enrollment Rule:**
    - A student cannot register for the same course offering multiple times.

Part 2: Apply Tactical Design

Entities

1- **Student** (Root entity of the StudentAggregate)

- **Attributes**:
    - StudentID: A unique identifier for the student.
    - Name: The student's name.

- Email: The student's email address.
- Enrollments: A list of the student's course enrollments.

- **Behavior**:
  - RegisterForCourse(courseOfferingID: String, studentID: String): Registers the student for a course offering.
  - DropCourse(courseOfferingID: String, studentID: String): Removes the student from an existing course offering.
  - GetRegisteredCourses(studentID: String): Returns a list of all courses the student is currently enrolled in.
  - GetTotalCredits(): Calculates the total number of credits the student is enrolled in.

2-**Enrollment** (Belongs to both the StudentAggregate and the CourseOfferingAggregate)

- **Attributes**:
  - EnrollmentID: A unique identifier for the enrollment.
  - StudentID: The ID of the associated student.
  - CourseOfferingID: The ID of the associated course offering.
  - EnrollmentStatus: The enrollment status (e.g., "Active", "Dropped").
  - Grade: The grade assigned to the student (optional).

- **Behavior**:
  - MarkAsDropped(): Updates the enrollment status to "Dropped".

- AssignGrade(grade: String): Updates the student's grade.
- IsActive(): Returns true if the enrollment status is "Active".

3-**CourseOffering** (Root entity of the CourseOfferingAggregate)

- **Attributes**:
  - CourseOfferingID: A unique identifier for the course offering.
  - CourseID: The ID of the associated course.
  - Semester: The semester in which the course is offered.
  - AvailableSeats: The number of seats available for registration.
- **Behavior**:
  - DecreaseSeats(): Decreases the available seats count.
  - IncreaseSeats(): Increases the available seats count.
  - HasAvailableSeats(): Checks if there are any available seats.

4-**Course** (Root entity of the CourseAggregate)

- **Attributes**:
  - CourseID: A unique identifier for the course.
  - Title: The course title.
  - Credits: The number of credits for the course.
  - Department: The department offering the course.
  - Prerequisites: A list of courses that must be completed before enrolling in this course.
- **Behavior**:

- GetDetails(): Returns detailed information about the course.

5-**Faculty**

- **Attributes**:
  - Name: The name of the faculty.
  - ID: A unique identifier for the faculty.

Aggregates

1. **StudentAggregate**
   - **Root Entity**: Student.
   - **Boundaries**:
     - Encompasses the Student entity and its related Enrollments.
   - **Consistency Rules**:
     - A student can only be registered for a course once, ensured by the uniqueness of Enrollment.
2. **CourseOfferingAggregate**
   - **Root Entity**: CourseOffering.
   - **Boundaries**:
     - Includes CourseOffering and its associated Enrollments.
   - **Consistency Rules**:
     - The number of students registered must not exceed the course offering's maximum capacity.

3. **CourseAggregate**
    - **Root Entity**: Course.
    - **Boundaries**:
        - Only includes the Course entity.
    - **Consistency Rules**:
        - The course information must remain consistent with its associated course offerings.

Value Objects

1. **EnrollmentStatus**
    - **Attributes**:
        - Status: A string representing the enrollment status (e.g., "Active", "Dropped").
        - Timestamp: The timestamp indicating when the status was last updated.
    - **Behavior**:
        - IsDropped(): Boolean: Returns true if the status is "Dropped".
        - UpdateStatus(newStatus: String): EnrollmentStatus: Creates a new EnrollmentStatus object with the updated status and timestamp.
2. **CourseDetails**
    - **Attributes**:

        - Location: The location where the course will take place.
        - Schedule: Days and times of the course.
        - Syllabus: A document or description detailing the course content.
    - **Behavior**:
        - FormatDetails(): String: Returns a formatted string containing the course title, credits, and department.

Domain Services

1. **RegistrationService**
    - **Behavior**

a. **StudentRegistrationForCourse(studentID, courseOfferingID)**

    - **Purpose**: Handles the process of registering a student for a course offering.
    - **Steps**:
        1. Initiates a transaction using **Unit of Work**.
        2. Retrieves the student and course offering entities from their respective repositories.
        3. Validates business rules:
            - Ensures there are available seats in the course offering.
            - Checks if the student satisfies the prerequisites for the course.

        - Verifies that the student does not exceed their credit limit.

    4. Creates a new **Enrollment** object for the student and course offering.
    5. Updates the student and course offering entities with the new enrollment.
    6. Saves changes to the repositories via **Unit of Work**.
    7. Publishes a **ClassRegisteredEvent** using **Message Bus**.
    8. Commits the transaction.

b. **ValidatePrerequisites(student, courseOffering)**

- **Purpose**: Validates that the student satisfies the prerequisites and credit limits for the course offering.
- **Checks**:
    - Ensures the student's total credits do not exceed the maximum credit limit.
    - Ensures the student has completed all prerequisite courses.

Responsibilities

1. **Coordinates the Registration Process**:
    - Acts as the central service for orchestrating all steps involved in registering a student for a course.

- Delegates specific tasks (e.g., seat validation, prerequisite checks) to the appropriate domain objects or services.

2. **Ensures Business Rule Compliance**:
   - Validates that all business rules (e.g., seat availability, prerequisites, credit limits) are satisfied before completing the registration.
3. **Manages Data Consistency**:
   - Ensures that changes to the **StudentAggregate**, **CourseOfferingAggregate**, and **Enrollment** are consistent across all repositories.
4. **Utilizes Domain Events for Eventual Consistency**:
   - Publishes domain events (e.g., ClassRegisteredEvent) to notify other parts of the system about the registration.
5. **Manages Transactions Using Unit of Work**:
   - Ensures atomicity by wrapping all operations in a transaction, committing only if all steps are successful, or rolling back otherwise.

2. DropService

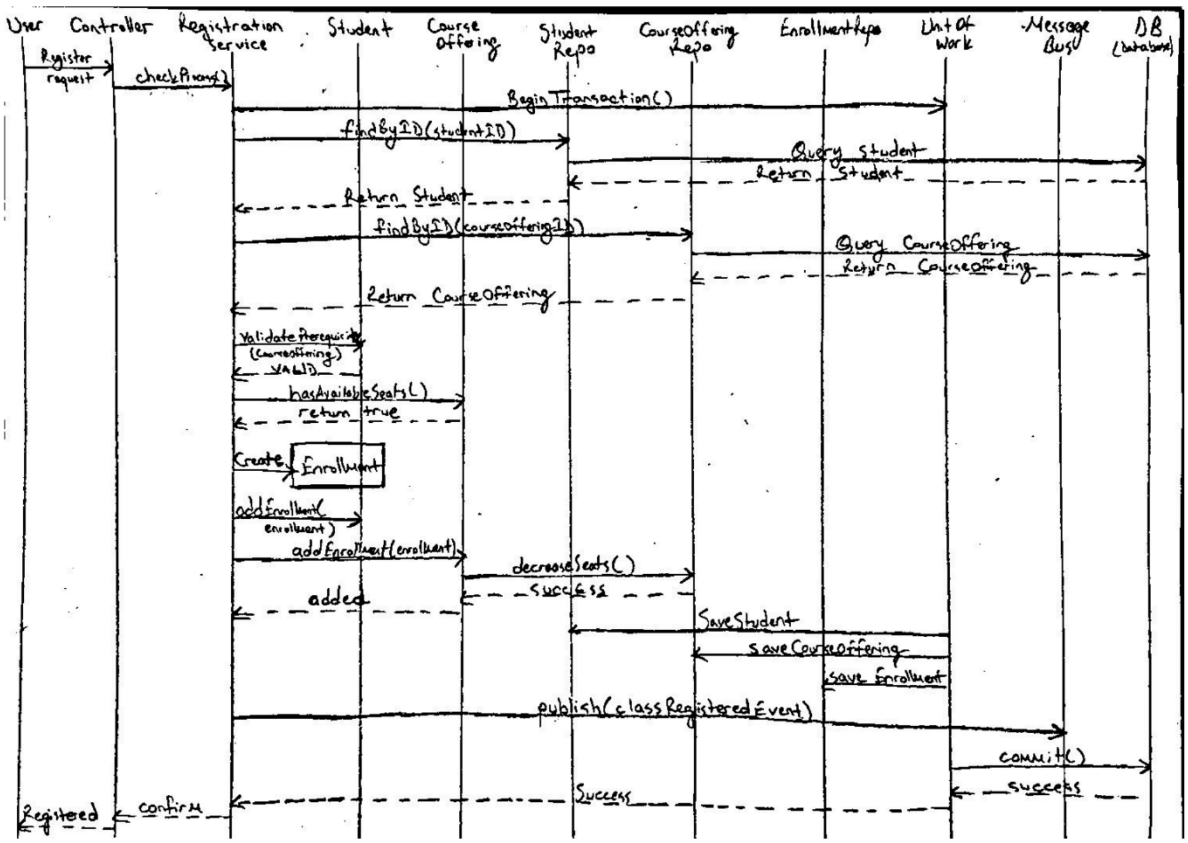
- **Behavior**:
  - viewRegisteredCourses(studentID): Retrieves the list of courses the student is currently enrolled in.
  - removeCourse(studentID, courseOfferingID): Handles the removal of the specified course for the given student.

        - removeStudent(courseOfferingID, studentID): Removes the student from the course offering.
        - updateEnrollment(courseID, studentID): Updates the enrollment record, marking the course as dropped.
    - **Responsibilities**:
        - Manages the course dropping process.
        - Ensures consistent updates between the StudentAggregate and CourseOfferingAggregate.

Part 3: Design the Use Case

---

1. UML Sequence Diagram



---

2. Eventual Consistency

- The **Message Bus** publishes a domain event called ClassRegisteredEvent.
- This domain event is asynchronously communicated to other independent systems (e.g., for notifications or analytics processing).

---

3. Design Decisions

i. Port and Adapter Pattern

The **Port and Adapter Pattern** ensures that the domain model is independent of infrastructure concerns by decoupling core business

logic from external systems. This is achieved through **Ports** (interfaces) and **Adapters** (implementations).

- **Ports**:
    - Act as abstractions for external systems or dependencies.
    - Examples in this use case:
        - **StudentRepository**, **CourseOfferingRepository**, and **EnrollmentRepository** as interfaces for data access.
        - **MessageBus** as an interface for publishing domain events.
- **Adapters**:
    - Provide concrete implementations for the Ports.
    - Examples:
        - **SQLStudentRepository**, **SQLCourseOfferingRepository**, and **SQLEnrollmentRepository** handle database interactions.
        - **MessageBus** manages event publishing.

By using this pattern:

1. The **domain model** is shielded from infrastructure changes, promoting flexibility.
2. Testing becomes easier, as mock implementations can replace adapters.
3. The system becomes more maintainable and extensible, allowing new adapters to be added without altering the core domain.

ii. GRASP Patterns

- Creator ensures that the RegistrationService and domain objects like CourseOffering are responsible for creating objects they aggregate or closely manage, such as Enrollment.
- Information Expert assigns validation logic to the classes (e.g., Student, CourseOffering) that have the required knowledge.
- Low Coupling ensures that dependencies are minimized using patterns like Unit of Work and Message Bus.
- Controller centralizes the workflow in RegistrationService, providing a single point of control for the use case.
- High Cohesion ensures that each class is focused on its domain responsibilities without overlapping.

iii. Unit of Work

- Manages all operations within a single transaction.
- Ensures rollback in case of failure, maintaining system consistency.

4. Application Service Code

```python
class RegistrationService:
    MAX_CREDIT_LIMIT = 18 #random

    def __init__(self, unit_of_work, message_bus):
        """
        :param unit_of_work: UnitOfWork object to manage repositories and transactions.
        :param message_bus: MessageBus object to handle domain events.
        """
        self.unit_of_work = unit_of_work
        self.message_bus = message_bus

    def RegisterStudentForCourse(self, student_id: str, course_offering_id: str):
        """Handles the process of adding a course for a student."""
        self.unit_of_work.begin_transaction() # Transaction begins
        try:
            # Retrieve student and course offering
            student = self.unit_of_work.student_repository.find_by_id(student_id)
            if not student:
                raise ValueError("Student not found")

            course_offering = self.unit_of_work.course_offering_repository.find_by_id(course_offering_id)
            if not course_offering:
                raise ValueError("Course offering not found")

            # Check available seats
            if not course_offering.has_available_seats():
                raise ValueError("No available seats")

            # Validate prerequisites
            self.validate_prerequisites(student, course_offering)

            # Create enrollment
            enrollment = Enrollment(student, course_offering)

            # Associate enrollment with student and course offering
            student.add_enrollment(enrollment)
            course_offering.add_enrollment(enrollment)
            course_offering.decrease_seats()
```

```python
            # Save changes using Unit of Work
            self.unit_of_work.student_repository.save(student)
            self.unit_of_work.course_offering_repository.save(course_offering)
            self.unit_of_work.enrollment_repository.save(enrollment)

            # Publish domain event
            self.message_bus.publish(ClassRegisteredEvent(student_id, course_offering_id))

            # Commit transaction
            self.unit_of_work.commit()
        except Exception as e:
            # Rollback on failure
            self.unit_of_work.rollback()
            raise e

    def validate_prerequisites(self, student, course_offering):
        """Checks if the student satisfies prerequisites for the course."""
        # Credit limit check
        current_credits = student.get_total_credits()
        course_credits = course_offering.course.credits
        if current_credits + course_credits > self.MAX_CREDIT_LIMIT:
            raise ValueError("Credit limit exceeded")

        # Prerequisites check
        prerequisites = course_offering.course.prerequisites
        for prerequisite in prerequisites:
            if not student.has_completed_course(prerequisite):
                raise ValueError(f"Prerequisite not satisfied: {prerequisite.title}")
```

5. Design Report

How Port and Adapter Ensures Independence

- The domain model is independent of repositories. StudentRepository and CourseOfferingRepository are connected via interfaces.

- This minimizes the impact of infrastructure changes on the domain model.

How GRASP Patterns Were Applied

- **Controller**: RegistrationService serves as the central orchestrator for the Add Course use case, managing interactions between domain objects, repositories, and infrastructure, while delegating tasks to appropriate domain objects.

- **Information Expert**:

- **Student**: Handles validations for prerequisites, credit limits, and enrollments, leveraging its knowledge of enrollments and completed courses.

- **CourseOffering**: Manages seat availability and enrollments, maintaining its state (e.g., available seats, semester).

- **Creator**: RegistrationService creates the **Enrollment** object as it has the required information (Student and CourseOffering) to establish the relationship.

- **Low Coupling**:

- **Unit of Work** handles transactions, decoupling domain objects from persistence.

- **Message Bus** supports asynchronous communication, keeping the domain model independent from external systems.

- **High Cohesion**:

- Each class focuses on its specific responsibilities:

    - **Student**: Enrollment logic and personal data.

- **CourseOffering**: Seat availability and enrollments.
- **RegistrationService**: Coordinates the registration workflow.

Unit of Work for Transaction Management

- All operations started with beginTransaction and concluded with either commit or rollback.
- Ensured atomicity and consistency of transactions.

Eventual Consistency

- ClassRegisteredEvent was published asynchronously, enabling independent systems to react without coupling.