Part 1: Define the Boundary of the Course Registration Context

**Purpose**

The **Course Registration Context** manages all course registration operations for a student in a specific term, including course selection, course drop, prerequisite/quota checks, and advisor approval. This context focuses on determining the courses a student will take for the current term and the processes leading up to final approval by the advisor.

**Model (Example Entities and Concepts)**

* **CourseSelection**: Tracks the courses selected by a student for the current term, including prerequisites and quota status.
* **RegistrationApproval**: Finalizes course registrations by transitioning them to confirmed status with advisor approval.
* **PrerequisiteInfo**: Contains information about the prerequisites for each course.
* **QuotaInfo**: Maintains the current quota status of courses.

**Contextual Boundaries**

* **Exclusions**:
  + This context does not manage details like the day or time a course is scheduled, or the venue for the classes (these are handled by the **Course Scheduling Context**).
  + Decisions such as which instructor teaches a course or when it will be offered in a semester are managed by the **Faculty Management Context**.
  + Student demographic data and academic history (e.g., graduation status, GPA) are stored in the **Student Information Management Context** and are accessed only when necessary.
* **Scope**:  
  The **Course Registration Context** is responsible only for the registration process within the current term, such as selecting, dropping courses, prerequisite/quota validation, and obtaining approval.

**Key Interactions**

1. **Faculty Management Context**: Provides information about course prerequisites and credit details. However, the **Course Registration Context** uses this information only during the registration process. Course creation or semester scheduling belongs to the Faculty Management Context.
2. **Course Scheduling Context**: Maintains the weekly schedule, timing, and venue details for courses. The **Course Registration Context** may reference this data (e.g., to check for time conflicts), but the creation and management of this information are outside its responsibility.
3. **Student Information Management Context**: Stores student identity and general academic data. The **Course Registration Context** consults this information to verify prerequisites but does not modify or manage it.

**Use Cases**

**1. Add Course (Course Selection)**

**Actor**: Student  
**Goal**: Allow students to select one or more courses from the available options for the current term and add them to their registration cart.

**Main Success Scenario**:

* The student logs into the system.
* The student views the list of courses available for the term (based on data from the Faculty Management Context).
* The student selects one or more courses.
* The system validates the courses for prerequisites and quota (handled by the Course Registration Context).
* If valid, the courses are added to the student's registration cart.

**Why is this part of the Course Registration Context?**  
The process of selecting courses is a core part of the registration workflow. The context focuses on enabling the student to answer "Which courses will I take?" without being concerned with faculty-related decisions or scheduling details.

**2. Drop Course**

**Actor**: Student  
**Goal**: Allow students to remove a course from their registration cart or list of unconfirmed courses.

**Main Success Scenario**:

* The student views their current registration cart.
* The student selects the course to be dropped.
* The system removes the course from the cart.

**Why is this part of the Course Registration Context?**  
Dropping courses is an essential part of managing the registration process. It involves updating the student's registration status and does not impact the Faculty Management or Course Scheduling Contexts.

**3. Course Registration Approval**

**Actors**: Student, Advisor  
**Goal**: Finalize course registrations by obtaining advisor approval.

**Main Success Scenario**:

* The student submits their final course selections to the advisor.
* The advisor reviews and approves the course selections.
* Approved courses are marked as officially registered.

**Why is this part of the Course Registration Context?**  
The approval process directly finalizes the registration workflow. The **Faculty Management Context** provides reference data, but the responsibility for handling approvals lies solely within the **Course Registration Context**.

**4. Prerequisite and Quota Validation**

**Actor**: System (internal logic of the Course Registration Context)  
**Goal**: Validate that the courses a student wants to register for meet prerequisite and quota requirements.

**Main Success Scenario**:

* When a student selects a course, the system checks prerequisites and quota.
* If valid, the course is added to the registration cart; otherwise, an error is shown.

**Why is this part of the Course Registration Context?**  
While prerequisites and quotas are defined by the Faculty Management Context, applying these rules to a student's registration is the responsibility of the **Course Registration Context**.

**5. View Registration Status**

**Actors**: Student, Advisor  
**Goal**: Allow the student or advisor to view the current registration status for the term, including confirmed and pending courses.

**Main Success Scenario**:

* The actor logs into the system.
* The actor views the student's registration status.

**Why is this part of the Course Registration Context?**  
This use case provides feedback on the registration process, a direct outcome of the **Course Registration Context** operations.

**Summary**

**(a) Separation of Contexts:**

1. **Faculty Management Context**: Defines courses, prerequisites, and credit rules but does not handle course selection or registration.
2. **Course Scheduling Context**: Manages schedules, times, and venues for courses but does not oversee registration.
3. **Course Registration Context**: Focuses solely on managing the registration process for the current term.

**(b) Why These Use Cases Belong Here:**

Each use case revolves around the student’s registration workflow, from course selection to final approval. Other contexts provide necessary data or rules, but the responsibility for execution and validation lies within the **Course Registration Context**.

Part 2: Apply Tactical Design

1)

**1a. Entities**

**1. StudentRegistration (Student Course Registration)**

• **Purpose:**  
Represents the courses a student wishes to take in a specific semester and the status of this registration process.

• **Attributes:**

* **registrationId** (UUID or similar unique identifier): A system-internal unique identifier for this entity.
* **studentId** (String or appropriate type): Indicates which student the registration belongs to.
* **semester** (Semester - Value Object): Specifies the semester for which the registration is made.
* **chosenCourses** (List<ChosenCourse>): A list of courses the student has selected. Each list item is another entity, the ChosenCourse object.
* **status** (RegistrationStatus - e.g., ENUM): Indicates the stage of the registration process (DRAFT, SUBMITTED, APPROVED, REJECTED).

• **Behavior (Methods):**

* **addCourse(chosenCourse: ChosenCourse): void**  
  Adds a new course to the registration basket if the current rules (prerequisites, quotas, semester restrictions) are satisfied.
* **removeCourse(courseId: String): void**  
  Removes a course from the registration if the registration is not yet approved or completed.
* **submitForApproval(): void**  
  Changes the registration status to SUBMITTED, initiating the advisor approval process. After this stage, adding or removing courses may be restricted.
* **approveRegistration(): void**  
  Changes the registration status to APPROVED as a result of advisor approval.
* **rejectRegistration(reason: String): void**  
  Rejects the registration with the advisor's reason, setting the status to REJECTED.

• **Why Entity?:**  
The StudentRegistration entity has its own identity (registrationId) and its state can change over time. Therefore, it is modeled as an entity.

**2. ChosenCourse (Selected Course)**

• **Purpose:**  
Represents each course that a student has selected within the registration.

• **Attributes:**

* **courseId** (String): Identifier of the course.
* **courseName** (String): Name of the course.
* **credits** (int): Credit information of the course.
* **prerequisiteSatisfied** (boolean): Indicates whether the student has satisfied the prerequisites for the course.
* **quotaAvailable** (boolean): Indicates whether there is available quota in the course.

• **Behavior (Methods):**

* **markPrerequisiteSatisfied(): void**  
  Sets the prerequisiteSatisfied attribute if prerequisite checks are passed.
* **markQuotaAvailable(): void**  
  Sets the quotaAvailable attribute if the quota is found to be available.

• **Why Entity?:**  
Although ChosenCourse carries its own identifier (courseId), it is part of the StudentRegistration aggregate. Each selected course maintains its own state (e.g., whether prerequisites are met), making it necessary to model it as a separate entity within the aggregate.

**1b. Aggregates**

**1. Student Aggregate**

* **Aggregate Root**: Student
* **Boundaries**:
  + The Student Aggregate takes the Student entity as its root and includes all entities and value objects directly related to the student's personal information and enrollment details.
  + **Included Objects**:
    - Student (root entity)
    - Address (Value Object)
    - Contact Information (Value Object)
    - Enrollment Records (Entity or Value Object, depending on the design)
    - Student Profile (Value Object)
* **Consistency Rules**:
  + Changes to a student’s personal information (address, contact details, etc.) must only be performed through the Student Aggregate Root.
  + Enrollment status and associated records must maintain consistency within the aggregate boundary. For example, changes to a student’s enrollment status should align with all related records.

**2. Course Aggregate**

* **Aggregate Root**: Course
* **Boundaries**:
  + The Course Aggregate takes the Course entity as its root and encompasses all entities and value objects related to course details.
  + **Included Objects**:
    - Course (root entity)
    - Schedule (Value Object)
    - Prerequisites (Value Object or Entity)
    - Instructor Assignments (Entity)
    - Course Materials (Value Object)
* **Consistency Rules**:
  + Changes to course details (e.g., schedule, prerequisites) must only be made through the Course Aggregate Root.
  + Course capacity and enrollment limits must be managed consistently within the aggregate boundary. For instance, the maximum number of students in a course must not be exceeded.

**3. Registration Aggregate**

* **Aggregate Root**: Registration
* **Boundaries**:
  + The Registration Aggregate takes the Registration entity as its root and includes all entities and value objects related to the registration process.
  + **Included Objects**:
    - Registration (root entity)
    - Payment Information (Value Object)
    - Registration Status (Value Object)
    - Registration History (Entity or Value Object)
* **Consistency Rules**:
  + Registration processes (e.g., course registration, withdrawal) must only be managed through the Registration Aggregate Root.
  + Ensuring that a student can only register for a course if all prerequisites (e.g., course capacity, student eligibility) are met must be handled consistently within the aggregate boundary.

**4. Instructor Aggregate**

* **Aggregate Root**: Instructor
* **Boundaries**:
  + The Instructor Aggregate takes the Instructor entity as its root and encompasses all entities and value objects related to the instructor.
  + **Included Objects**:
    - Instructor (root entity)
    - Contact Information (Value Object)
    - Assigned Courses (Entity)
    - Instructor Profile (Value Object)
* **Consistency Rules**:
  + Instructor information and assigned courses must only be managed through the Instructor Aggregate Root.
  + Rules such as preventing time conflicts between courses assigned to the same instructor must be enforced within the aggregate boundary.

**Managing Consistency Between Aggregates**  
Aggregates are designed to be independent and loosely coupled. However, when business rules require coordination across multiple aggregates, domain services or domain events can be used. In the Course Registration Context, this can be achieved as follows:

* **Domain Service**: Used to check a student's course registration limits.
* **Domain Events**: For example, an event can be published when a course reaches its capacity to prevent new registrations.

**Example: Registration Process**

* **Student Registering for a Course**:
  + A request is made to create a registration through the Registration Aggregate.
  + The Registration Aggregate interacts with the Student Aggregate to verify the student’s eligibility.
  + The Registration Aggregate interacts with the Course Aggregate to check course capacity.
  + If all checks are successful, the Registration Aggregate creates the registration and updates its status.
* **Ensuring Consistency**:
  + All operations on the Registration Aggregate must be performed through the Aggregate Root.
  + Changes made during the process must be atomic, meaning either all changes are successfully applied, or none are applied

1c. Value Objects

**1c. Value Objects for the Course Registration Context**

1. **Semester**
   * **Purpose:** Represents a specific term or semester for which the student is registering.
   * **Attributes:**
     1. year (int): The year of the semester (e.g., 2024).
     2. term (enum): The term of the semester (e.g., SPRING, FALL, SUMMER).
   * **Behavior (Methods):**
     1. isCurrentSemester(): boolean  
        Checks if the semester is the current active semester.
     2. toString(): String  
        Formats the semester as a readable string (e.g., "Spring 2024").
   * **Why Value Object?**  
     The Semester value object is immutable and represents descriptive information without identity.
2. **CoursePrerequisites**
   * **Purpose:** Encapsulates the prerequisites required for a course.
   * **Attributes:**
     1. requiredCourses (List<String>): List of course IDs that must be completed before enrolling.
     2. minimumGPA (double): Minimum GPA required for enrollment.
   * **Behavior (Methods):**
     1. areSatisfied(completedCourses: List<String>, studentGPA: double): boolean  
        Checks if the prerequisites are satisfied based on the student's completed courses and GPA.
   * **Why Value Object?**  
     Prerequisites are descriptive data with no independent identity, making them a perfect candidate for a value object.

* **QuotaStatus**
  + **Purpose:** Represents the current quota status of a course.
  + **Attributes:**
    - maxQuota (int): The maximum number of students allowed.
    - currentEnrollment (int): The number of students currently enrolled.
  + **Behavior (Methods):**
    - isAvailable(): boolean  
      Checks if there are open spots in the course quota.
    - getRemainingSeats(): int  
      Returns the number of remaining seats in the course.
  + **Why Value Object?**  
    The quota information is a measurement of the course capacity at a specific moment, not an entity.
* **RegistrationStatus**
  + **Purpose:** Represents the status of a course registration.
  + **Attributes:**
    - status (enum): The current registration status (e.g., DRAFT, SUBMITTED, APPROVED, REJECTED).
  + **Behavior (Methods):**
    - canEdit(): boolean  
      Checks if the registration can be modified based on its status.
    - isFinalized(): boolean  
      Returns true if the status is APPROVED or REJECTED.
  + **Why Value Object?**  
    Status is descriptive and immutable, representing a concept without identity.

**5 . CourseDetails**

* + **Purpose:** Provides detailed information about a course.
  + **Attributes:**
    - courseId (String): Unique identifier for the course.
    - courseName (String): The name of the course.
    - credits (int): The number of credits the course offers.
  + **Behavior (Methods):**
    - toString(): String  
      Formats the course details as a readable string.
  + **Why Value Object?**  
    Course details are purely descriptive and do not represent a unique entity in the context of registration.

**Summary of Why These Are Value Objects**

* **Immutability:** All value objects represent immutable concepts that describe properties or relationships in the domain.
* **No Identity:** None of these objects require a unique identifier or exist independently.
* **Reusability:** They can be easily reused across different contexts or aggregates without modification.

These value objects complement the entities and aggregates, ensuring the domain model adheres to the principles of Domain-Driven Design (DDD).

1d. Domain Objects:

**1. Course Registration Validation Service**

* **Purpose:** Validates the eligibility of a student to register for a specific course based on prerequisites, quotas, and term-specific rules.
* **Responsibilities:**
  + Check if the student satisfies all prerequisites for the course using the student's academic history and the course's prerequisite information.
  + Verify if there is sufficient quota available for the course.
  + Ensure the student does not exceed their credit limit for the term.
* **Methods:**
  + validatePrerequisites(studentId: String, courseId: String): boolean
    - Checks if the student has completed all prerequisite courses.
  + validateQuota(courseId: String): boolean
    - Ensures the course quota has not been exceeded.
  + validateCreditLimit(studentId: String, selectedCourses: List<CourseDetails>): boolean
    - Verifies that the total credits do not exceed the student's allowed credit limit for the term.
* **Why a Domain Service?**  
  This logic involves multiple aggregates (Student, Course, Registration) and crosses aggregate boundaries.

**2. Advisor Approval Service**

* **Purpose:** Manages the advisor's review and approval process for course registrations.
* **Responsibilities:**
  + Facilitate the advisor's ability to review and approve or reject a student's registration request.
  + Notify the student of the advisor's decision.
* **Methods:**
  + approveRegistration(registrationId: String, advisorId: String): void
    - Marks the student's registration as approved after validation.
  + rejectRegistration(registrationId: String, advisorId: String, reason: String): void
    - Rejects the registration with a given reason.
  + notifyStudent(studentId: String, decision: String): void
    - Sends a notification to the student about the decision.
* **Why a Domain Service?**  
  The advisor approval process is not tied to a single aggregate but involves both Student and Registration aggregates.

**3. Conflict Detection Service**

* **Purpose:** Identifies and resolves scheduling conflicts in a student's course selections.
* **Responsibilities:**
  + Compare the weekly schedules of the selected courses to detect overlaps.
  + Suggest alternatives if conflicts are detected.
* **Methods:**
  + checkForConflicts(selectedCourses: List<CourseDetails>): List<ConflictInfo>
    - Returns a list of detected conflicts.
  + suggestAlternatives(conflictedCourses: List<String>): List<CourseDetails>
    - Provides alternative courses that do not conflict with the existing schedule.
* **Why a Domain Service?**  
  Conflict detection involves data from multiple contexts, such as CourseScheduling and Registration.

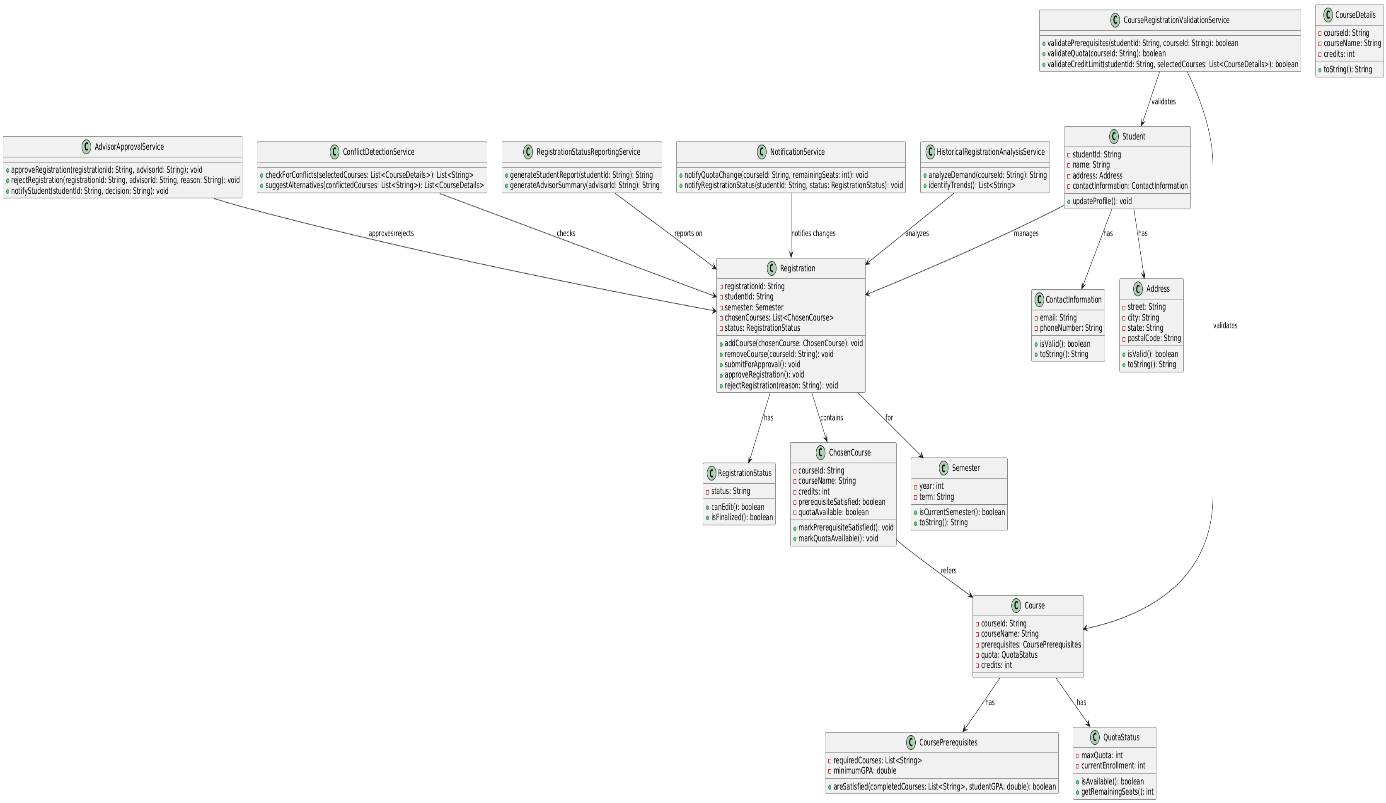
**4. Notification Service**

* **Purpose:** Handles sending notifications related to course registration processes (e.g., registration approval, rejection, quota updates).
* **Responsibilities:**
  + Notify students about changes in registration status.
  + Inform advisors of pending registration approvals.
  + Update students when quota statuses change.
* **Methods:**
  + notifyQuotaChange(courseId: String, remainingSeats: int): void
    - Notifies students on the waiting list about available seats.
  + notifyRegistrationStatus(studentId: String, status: RegistrationStatus): void
    - Sends a notification about the registration status change.
* **Why a Domain Service?**  
  Notifications are cross-cutting concerns that involve multiple aggregates and external systems.

**Why Use Domain Services?**

* **Separation of Concerns:** Keeps the entities and aggregates focused on their core responsibilities.
* **Reusability:** Domain services can be reused across different parts of the domain.
* **Cross-Aggregate Logic:** Provides a mechanism to handle operations that span multiple aggregates without tightly coupling them.

1. UML Class diagram:



NOTE: Provided seperated file as “Uml Class Diagram.png”

Part 3: Design the Use Case: Drop a Class

**1. Tactical Pattern Objects and Their Responsibilities**

**Entities:**

* **Student** (part of the Student Aggregate): Represents the student, holding their personal details and their enrollment records.
* **Course** (part of the Course Aggregate): Represents the course details, including capacity, schedule, and prerequisites.
* **Registration** (part of the Registration Aggregate): Represents the registration record connecting a student with a course for a specific term.

**Value Objects:**

1. **Enrollment Records**: Stores information about the courses the student is enrolled in (within the Student Aggregate).
2. **Registration Status**: Indicates whether a registration is confirmed, dropped, or pending (within the Registration Aggregate).

**Aggregates:**

1. **Student Aggregate**: Manages the student entity and associated enrollment records.
2. **Course Aggregate**: Manages course details, including the list of students enrolled in the course.
3. **Registration Aggregate**: Manages the registration process, including registration history and status.

**Domain Services:**

* **Drop Class Service**: Coordinates the interaction between aggregates (e.g., checking the drop period, removing the student from the course, and updating registration status).

**2. Collaboration to Fulfill the Use Case**

**Precondition:**

* The **Drop Class Service** verifies that:
  + The student is currently registered for the course (via the Student Aggregate or Registration Aggregate).
  + The drop period is active (e.g., validated through a domain rule or external configuration).

**Steps:**

**Step 1: View the List of Registered Courses**

* The system queries the Student Aggregate to retrieve the **Enrollment Records**.
* The Enrollment Records value object contains a list of courses the student is currently enrolled in.
* The list is displayed to the student.

**Step 2: Select a Course to Drop**

* The student selects a course from the displayed list.
* The system validates the selected course:
  + Checks if the student is currently registered for the selected course (via the Student Aggregate or Registration Aggregate).
  + Ensures the drop period is active.

**Step 3: Confirm the Drop**

* The student confirms the drop action.
* The **Drop Class Service** coordinates the following actions:
  + **Student Aggregate**: Updates the Enrollment Records by removing the selected course.
  + **Course Aggregate**: Updates the course details to decrement the number of enrolled students.
  + **Registration Aggregate**: Updates the Registration Status to reflect that the course has been dropped and logs the drop event in the Registration History.

**Postcondition:**

* The selected course is removed from the student’s enrollment records.
* The course's capacity is incremented in the Course Aggregate.
* The registration status is updated to "dropped" in the Registration Aggregate.

**3. Interaction Diagram**

**Actors:**

* **Student**
* **System** (mediates between aggregates and services)

**Interactions:**

**Retrieve Registered Courses**:

The system queries the Student Aggregate to fetch the Enrollment Records.

The system displays the list of registered courses to the student.

**Select a Course to Drop**:

The system validates the selected course using the Student Aggregate and Registration Aggregate.

The system checks the active drop period.

**Confirm Drop**:

The system invokes the **Drop Class Service**, which:

Updates the Enrollment Records in the Student Aggregate.

Updates the course capacity in the Course Aggregate.

Updates the registration status in the Registration Aggregate.

**4. Example Implementation in Pseudocode**

**Drop Class Service**

java

Copy code

class DropClassService {

public void dropClass(StudentId studentId, CourseId courseId) {

// Fetch aggregates

StudentAggregate student = studentRepository.findById(studentId);

CourseAggregate course = courseRepository.findById(courseId);

RegistrationAggregate registration = registrationRepository.findByStudentAndCourse(studentId, courseId);

// Preconditions

if (!registration.isActive()) {

throw new IllegalStateException("Registration not active for this course");

}

if (!isDropPeriodActive()) {

throw new IllegalStateException("Drop period is not active");

}

// Update aggregates

student.removeCourseFromEnrollment(courseId);

course.decrementEnrollment();

registration.markAsDropped();

// Persist changes

studentRepository.save(student);

courseRepository.save(course);

registrationRepository.save(registration);

}

private boolean isDropPeriodActive() {

// Logic to check if the drop period is active

return true; // Example stub

}

}

**5. Key Design Notes**

**Single Responsibility of Aggregates**:  
Each aggregate handles updates only within its boundary:

The Student Aggregate updates the enrollment records.

The Course Aggregate manages course capacity.

The Registration Aggregate ensures registration status is updated.

**Loose Coupling with Domain Services**:  
The **Drop Class Service** acts as the coordinator without tightly coupling aggregates.

**Atomic Transactions**:  
All changes must be performed atomically to ensure consistency. If any operation fails, no changes are applied. This can be implemented using a Unit of Work pattern or transactional mechanisms.

This design ensures that the "Drop a Class" use case is cleanly implemented, adhering to DDD principles.

2.Use GRASP Patterns for responsibility assignment

**Key GRASP Patterns Used**

**a. Controller**

**Role**: Orchestrates the interaction between the user and the system by delegating work to other objects.

**Assignment**: A RegistrationController acts as the controller for this use case.

**Justification**:

The controller receives input from the student (e.g., selecting and confirming the course to drop).

It delegates tasks such as retrieving data from aggregates, invoking the domain service, and coordinating the overall process.

**b. Information Expert**

**Role**: Assigns responsibility to the class with the most relevant information to fulfill the task.

**Assignments**:

**Student Aggregate**:

Responsible for managing the list of courses a student is registered for.

Has the necessary knowledge to check if a student is enrolled in the selected course.

Justification: The Student Aggregate contains the Enrollment Records value object, making it the best candidate for maintaining and validating a student’s enrollment data.

**Course Aggregate**:

Responsible for updating course capacity and validating constraints related to course capacity.

Justification: The Course Aggregate contains course-specific information such as the list of enrolled students and capacity details.

**Registration Aggregate**:

Responsible for managing the registration status and history for a student-course relationship.

Justification: The Registration Aggregate is the expert in tracking the state of registrations (active, dropped, pending) and maintaining consistency across changes.

**c. Creator**

**Role**: Assigns the responsibility of creating an object to the class that has the information required to initialize it.

**Assignments**:

**Drop Class Service**:

Creates the necessary changes to the Student, Course, and Registration aggregates as part of the process.

Justification: The service coordinates the interaction and knows the overall workflow for dropping a class.

**Registration Aggregate**:

Responsible for updating the registration status to "dropped."

Justification: The Registration Aggregate already tracks the state of the registration and has the authority to update it.

**d. High Cohesion**

**Role**: Ensures that responsibilities are assigned to promote focused functionality in objects, avoiding overburdening any one object.

**Assignments**:

**Drop Class Service**:

Acts as the coordinator for interactions across aggregates.

Justification: This service centralizes the "drop a class" workflow, preventing any one aggregate or controller from becoming overloaded with responsibilities.

**Student Aggregate**:

Only manages enrollment records and leaves validation and course capacity checks to other aggregates.

Justification: This separation keeps the Student Aggregate cohesive and focused on the student’s perspective of enrollment.

**e. Low Coupling**

**Role**: Reduces dependency between objects to make the system more maintainable and flexible.

**Assignments**:

**Drop Class Service**:

Mediates interactions between the Student, Course, and Registration aggregates.

Justification: By centralizing cross-aggregate logic in the service, we minimize direct coupling between aggregates. For example, the Student Aggregate does not need to directly interact with the Course Aggregate.

**3. Applying the Port and Adapter Pattern**

**1. Core Components**

**Domain Layer** (Core): Contains the business logic, domain services, and aggregates (Student, Course, Registration) relevant to the "Drop a Class" use case.

It operates independently of any external systems or frameworks.

Interfaces (Ports) are defined in this layer to express what the domain needs from external systems (e.g., repositories, event publishers).

**Application Layer**: Orchestrates the flow of data between the user interface and the domain layer. This layer contains the DropClassService, which invokes domain logic and interfaces with the infrastructure via Ports.

**Infrastructure Layer**: Implements the external system concerns, such as:

Repositories for persistence.

Event publishers for domain events.

Controllers for HTTP endpoints or UI interactions.

These are connected to the domain layer through Adapters.

**2. Ports**

Ports are interfaces that represent the domain layer's expectations from external systems. They allow the domain to remain independent of infrastructure by defining contracts.

**Inbound Port**: Interfaces that allow external systems (e.g., UI) to interact with the application. Example:

**DropClassUseCase**: A port defining the methods for dropping a class (e.g., dropClass(studentId, courseId)).

**Outbound Port**: Interfaces through which the domain interacts with external systems. Examples:

**StudentRepository**: Provides methods to retrieve and persist StudentAggregate.

**CourseRepository**: Provides methods to retrieve and persist CourseAggregate.

**RegistrationRepository**: Provides methods to retrieve and persist RegistrationAggregate.

**EventPublisher**: Allows publishing domain events (e.g., a capacity update when a course is dropped).

**3. Adapters**

Adapters implement the Ports and handle the interaction with infrastructure.

**Inbound Adapter**: Adapts user interactions (UI, CLI, REST API) into requests for the application.

Example: A RegistrationController receives a request to drop a course and calls the DropClassUseCase interface.

**Outbound Adapter**: Implements outbound ports to interact with external systems like databases or message brokers.

Example:

**JPAStudentRepository**: Implements StudentRepository using a JPA-based database.

**KafkaEventPublisher**: Implements EventPublisher using Kafka for event-driven communication.

**b. Benefits of the Port and Adapter Pattern**

**1. Independence of the Domain Model**

**Decoupled Domain Logic**:

The domain layer does not depend on infrastructure concerns such as persistence or UI frameworks. It only relies on Ports (interfaces), which define the required functionality.

For example, whether the StudentRepository uses JPA, MongoDB, or an in-memory database, the domain logic remains unaffected.

**Reusability**:

The domain model can be reused in different contexts (e.g., CLI, web, mobile applications) since it is not tied to any specific technology stack.

**2. Improved Maintainability**

**Easier to Test**:

By mocking Ports, the domain logic can be unit tested without involving infrastructure components like databases or message queues.

For example, the DropClassService can be tested by providing mocked StudentRepository and CourseRepository implementations.

**Clear Separation of Concerns**:

Each layer has distinct responsibilities, making the system easier to understand and modify.

**3. Enhanced Flexibility**

**Swappable Infrastructure**:

The implementation of adapters can be changed independently of the domain logic.

For example, if the persistence technology is switched from JPA to MongoDB, only the StudentRepository adapter needs to be updated.

**Support for Multiple Interfaces**:

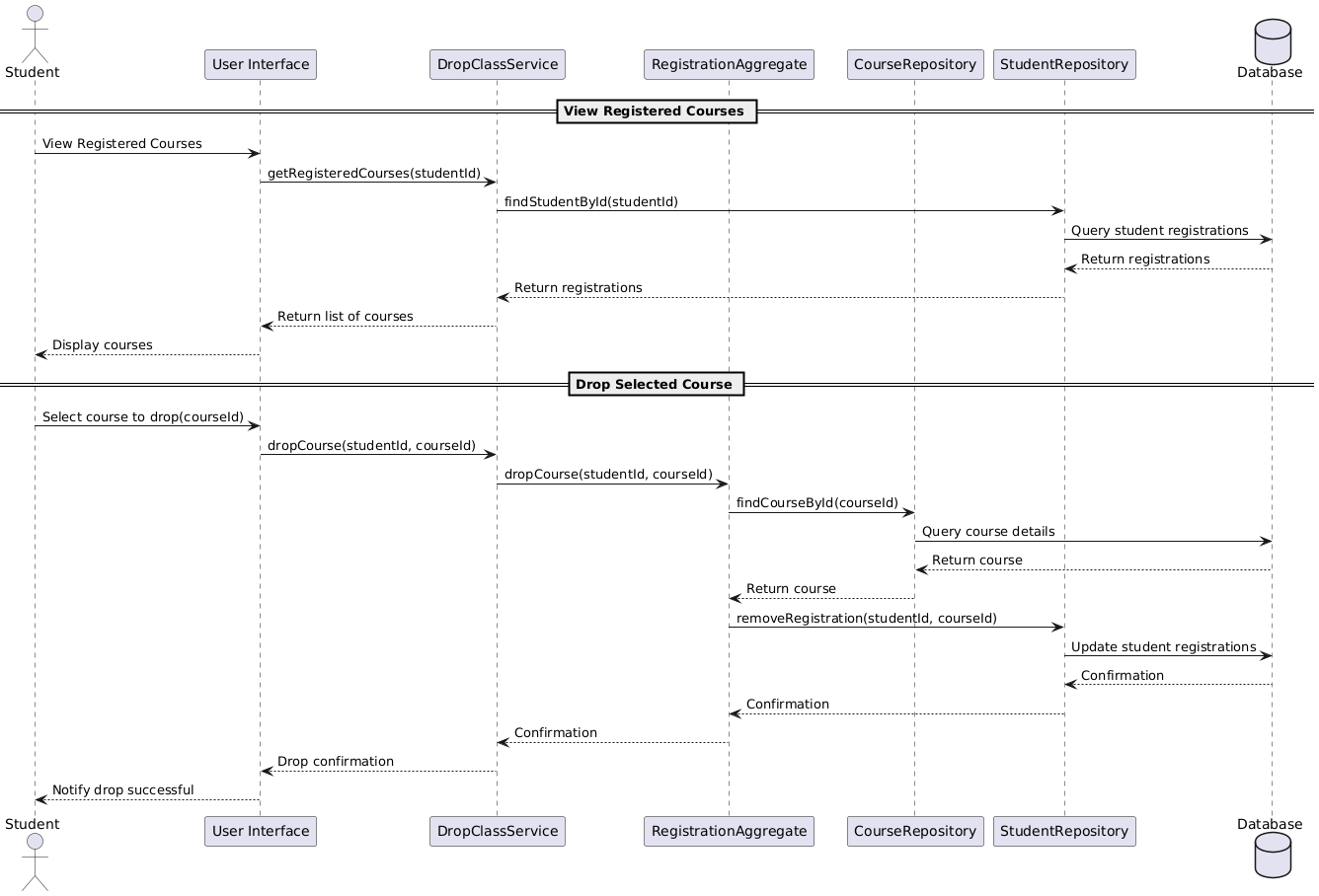
Different inbound adapters (e.g., REST API, CLI) can be added without modifying the domain layer. For example, a REST API for dropping a class can coexist with a batch processing job.

**4. Scalability with Domain Events**

**Event-Driven Extensions**:

Outbound adapters like EventPublisher can notify other systems when a course capacity changes. These notifications do not require changes in the domain logic, making it easier to scale functionality.

4) UML Sequence diagram



Additional Notes:

**a. How the Design Makes the Domain Model Independent from Infrastructure**

The **Port and Adapter Pattern** (Hexagonal Architecture) achieves independence by introducing interfaces (Ports) in the domain and implementing those interfaces (Adapters) in the infrastructure layer. This approach decouples the domain logic from specific technologies, frameworks, or external systems.

**Key Features of the Design:**

**Inbound Ports:**

Interfaces like DropClassUseCase define the expected input and behavior for interacting with the domain model.

These ports allow various types of user interfaces (e.g., REST APIs, CLI, or GUI) to interact with the system without affecting the domain logic.

**Outbound Ports:**

Interfaces such as StudentRepository, CourseRepository, and EventPublisher define the domain’s expectations from external systems like databases and event buses.

These ports allow infrastructure details (e.g., whether the system uses JPA, MongoDB, or Kafka) to remain isolated from the core logic.

**Encapsulation of Business Logic:**

The DropClassService resides in the application layer and uses domain services, aggregates, and repositories via Ports. This ensures that business rules are centralized and not scattered across various components.

**Implementation in Adapters:**

Infrastructure-specific implementations, such as JPAStudentRepository or KafkaEventPublisher, are plugged into the outbound ports, leaving the domain model unaffected by changes in infrastructure.

**b. Ensuring Flexibility for Future Changes**

**Easily Swappable Infrastructure**:

Since the domain interacts with infrastructure via Ports, changing the underlying database or messaging system involves only updating the adapter implementation, not the domain logic.

**Support for Multiple Interfaces**:

The system can accommodate additional inbound adapters (e.g., batch processing or mobile APIs) without modifying the domain logic.

**Scalability through Event-Driven Design**:

Domain events can be published using an adapter implementing the EventPublisher interface. Adding new event subscribers or switching to a different event broker (e.g., from Kafka to RabbitMQ) is straightforward.

**Maintainability**:

The clear separation of concerns ensures that changes to one layer (e.g., infrastructure) do not ripple into other layers (e.g., the domain).

By using the Port and Adapter Pattern, the system is resilient to changes and easily extendable, fostering long-term flexibility and maintainability.

**4. GRASP Responsibility Assignment Justification**

**a. Responsibility Assignment for Key Objects**

**1. Controller: RegistrationController**

**GRASP Pattern**: **Controller**

**Responsibility**:

Handles input from the user and delegates the request to the appropriate application service (DropClassService).

**Justification**:

The controller centralizes interactions for the use case, adhering to the Single Responsibility Principle.

This prevents the service or domain layer from being overloaded with UI-related tasks.

**2. DropClassService**

**GRASP Pattern**: **Coordinator, High Cohesion**

**Responsibility**:

Coordinates the workflow for dropping a class by invoking methods on domain aggregates and repositories.

Encapsulates business rules for validating the drop action (e.g., checking the drop period and updating aggregates).

**Justification**:

The service focuses on orchestrating the use case, maintaining high cohesion by keeping the domain logic centralized and clear.

It avoids coupling the domain layer directly with infrastructure by using repositories and event publishers via Ports.

**3. Student Aggregate**

**GRASP Pattern**: **Information Expert**

**Responsibility**:

Manages the student’s enrollment records, including validation and updates during the drop process.

**Justification**:

The Student Aggregate owns the Enrollment Records value object, making it the best candidate for operations related to the student’s course registrations.

**4. Course Aggregate**

**GRASP Pattern**: **Information Expert**

**Responsibility**:

Manages the course’s capacity and enrollment list, ensuring that capacity rules are enforced when a student drops a class.

**Justification**:

The Course Aggregate has the necessary knowledge about course capacity and enrolled students, making it the natural owner of these responsibilities.

**5. Registration Aggregate**

**GRASP Pattern**: **Information Expert**

**Responsibility**:

Tracks the registration state (e.g., active, dropped) and ensures that the registration history is updated consistently.

**Justification**:

The Registration Aggregate is responsible for managing the lifecycle of a registration, making it the ideal choice for updating the registration status.

**6. Repositories (StudentRepository, CourseRepository, RegistrationRepository)**

**GRASP Pattern**: **Creator**

**Responsibility**:

Provide persistence for aggregates and create instances when required by the domain logic.

**Justification**:

Repositories are well-suited for managing aggregate lifecycle and providing abstractions for database access, ensuring that infrastructure concerns do not leak into the domain layer.

**7. EventPublisher**

**GRASP Pattern**: **Low Coupling**

**Responsibility**:

Publishes domain events, such as a course capacity update, to notify external systems.

**Justification**:

This ensures loose coupling between the domain logic and other systems that need to react to changes, like notifying the UI or triggering analytics.