Here is the article translated into English in **Markdown** format with all the necessary details, including the categories of patterns in the catalog:

Enterprise Design Patterns

Introduction

In the development of enterprise applications, **design patterns** are fundamental to providing proven, reusable solutions to common architectural problems. These patterns help create more maintainable, scalable, and understandable systems by following best practices in software development.

This article focuses on the **Domain Model Pattern**, a widely used pattern in enterprise applications. This pattern helps structure business logic efficiently, keeping business rules and system objects well-defined.

Project Context

The project focuses on the **management of courses and student enrollment**, where students can enroll in courses as long as they meet certain requirements, such as prerequisites, credit limits, and schedule availability.

The application allows students to enroll in various activities or courses, ensuring that all business rules, such as prerequisites, enrollment limits, and schedule validation, are respected.

Design Patterns: **Domain Model**

The **Domain Model** is a design pattern used to represent business objects within an application, encapsulating business logic within these objects. This pattern helps maintain business rules and the relationships between objects directly within the domain model.

What is the Domain Model Pattern?

The **Domain Model** organizes and represents key concepts within the domain and the relationships between them, allowing business rules to be managed within domain objects rather than external components like controllers or databases.

This pattern is useful when there are complex business rules that need to be centrally managed within the domain objects, rather than relying on external services or databases.

Categories of Patterns in the Catalog

Below are the categories of design patterns relevant in the **Enterprise Application Architecture Patterns Catalog**. These categories help classify the patterns based on the application area they address.

Business Logic Patterns

- **Transaction Script**: This pattern organizes business logic in scripts that handle transactions. Each operation in the application is implemented as a method where business rules are executed.
- **Domain Model**: This is the pattern we are using in the example. Here, business logic is organized within domain objects, making it easier to create complex and scalable applications.
- **Table Module**: This pattern uses a single class to handle business logic for a table in the database. It is useful when dealing with a highly normalized database structure.

Data Source Architectural Patterns

- **Row Data Gateway**: This pattern is used when a database row represents an object in the application. Each row has a "gateway" that manages access to it.
- **Table Data Gateway**: Similar to the previous pattern, but instead of accessing an individual row, the entire table of data is accessed.
- **Active Record**: In this pattern, the domain object is also responsible for persisting its own state to the database. Each object knows how to store and retrieve its data.
- **Data Mapper**: Separates business logic from data access. Domain objects are not directly connected to the database; instead, they are mapped through a mapper responsible for persistence.

Object-Relational Behavioral Patterns

- **Unit of Work**: This pattern manages a set of database operations in a consistent manner, ensuring that all changes are made in a single transaction.
- **Identity Map**: Ensures that each object is represented by a single instance during a session, avoiding duplication and issues with object state.
- **Lazy Load**: Loads data only when it is needed, optimizing application performance by avoiding unnecessary data loading.

Service Layer Patterns

- **Service Layer**: Defines a service layer that groups related business operations. It is used to organize and simplify business logic in large applications.
- **Remote Facade**: Similar to the Service Layer, but focused on distributed applications, where a "facade" provides a unified interface to interact with remote services.

Distribution Patterns

- **Data Transfer Object (DTO)**: This pattern is used to transfer data between layers or applications without exposing business logic, optimizing data transfer.

UI Patterns

- **Model-View-Controller (MVC)**: A pattern that separates the application into three components: the model (data), the view (user interface), and the controller (application logic), making the application easier to manage and scale.

- **Model-View-Presenter (MVP)**: Similar to MVC, but with a difference in the interaction between the view and the presenter, where the presenter has more control over the user interface.

Session State Patterns

- **Client Session State**: This pattern manages session state on the client side, maintaining necessary data throughout the user's session.
- **Server Session State**: Similar to the above, but the state is maintained on the server, offering better session control, though with higher server resource consumption.

Key Code

Below are the key parts of the code that implement the **Domain Model Pattern** for a course enrollment system:

Class Definitions

1. **Course Class**: Represents a course, with attributes like code, name, credits, and schedules.

```python
class Course:
    def __init__(self, code, name, credits, semester, prerequisites=None, schedules=None):
        self.code = code
        self.name = name
        self.credits = credits
        self.semester = semester
        self.prerequisites = prerequisites or []
        self.schedules = schedules or []

    def __str__(self):
        return f"{self.code} - {self.name} ({self.credits} credits)"
```

2. **Student Class**: Represents a student, with attributes such as name, approved courses, and active enrollments.

```python
class Student:
    def __init__(self, name):
        self.name = name
        self.approved_courses = []  # Approved courses
        self.enrollments = []  # Active enrollments

    def total_credits(self):
        return sum(enrollment.course.credits for enrollment in self.enrollments)

    def can_enroll_in(self, course: Course):
        if any(enrollment.course == course for enrollment in self.enrollments):
            return False, "Already enrolled in this course."
        for prereq in course.prerequisites:
            if prereq not in self.approved_courses:
                return False, f"Missing prerequisite: {prereq.code}"
        if self.total_credits() + course.credits > 24:
            return False, "Exceeds credit limit."
        return True, "Enrollment valid."

    def enroll_in(self, course: Course):
        can_enroll, message = self.can_enroll_in(course)
        if can_enroll:
            self.enrollments.append(course)
            print(f"✅ {self.name} enrolled in: {course}")
        else:
            print(f"❌ Could not enroll in {course}: {message}")
```

3. **Schedule Class**: Represents a course schedule and checks for scheduling conflicts with other courses.

```
class Schedule:
    def __init__(self, day, start: time, end: time):
        self.day = day  # Day of the week
        self.start = start  # Start time
        self.end = end  # End time

    def overlaps_with(self, other):
        if self.day != other.day:
            return False
        return not (self.end <= other.start or self.start >= other.end)
```

Enrollment Simulation

In this example, the student tries to enroll in multiple courses. Below is the code to simulate this process:

```
# Create course instances
prog1 = Course("INF101", "Programming I", 4, semester=1, schedules=[
    Schedule("Monday", time(8, 0), time(10, 0))
])
prog2 = Course("INF201", "Programming II", 4, semester=2, prerequisites=[prog1],
schedules=[
    Schedule("Tuesday", time(10, 0), time(12, 0))
])
mat1 = Course("MAT101", "Mathematics I", 6, semester=1, schedules=[
    Schedule("Monday", time(9, 30), time(11, 30))
])

# Create a student instance
student = Student("John Doe")

# Try to enroll the student in courses
student.enroll_in(prog1)  # Should succeed
student.enroll_in(prog2)  # Should fail due to prerequisite
student.enroll_in(mat1)  # Should fail due to schedule conflict
```

Expected Output

The output from running the code is as follows:

```
✅ John Doe enrolled in: INF101 - Programming I (4 credits)
❌ Could not enroll in INF201 - Programming II (4 credits): Missing prerequisite:
INF101
❌ Could not enroll in MAT101 - Mathematics I (6 credits): Schedule conflict with
another course.
❌ Could not enroll in MAT101 - Mathematics I (6 credits): Schedule conflict with
another course.
```

Explanation of the Output:

1. **Programming I (INF101)**: The student, **John Doe**, successfully enrolls in the "Programming I" course, as there are no restrictions preventing it.

2. **Programming II (INF201)**: Enrollment fails because **John Doe** has not passed "Programming I" (prerequisite), thus preventing the enrollment.

3. **Mathematics I (MAT101)**: Enrollment fails due to a **schedule conflict** with the "Programming I" course, as both are scheduled for Monday morning.

This result shows how the system correctly manages business rules, validating prerequisites, credit limits, and course schedules.

Conclusion

The **Domain Model Pattern** has been successfully implemented to manage student enrollment in courses, ensuring that all business rules regarding prerequisites, credit limits, and schedule conflicts are properly respected.