Introduction to Design Principles

CS6.401 Software Engineering

Karthik Vaidhyanathan
https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY
HYDERABAD

SERC
Software Engineering Research Centre

GRASP: Low Coupling

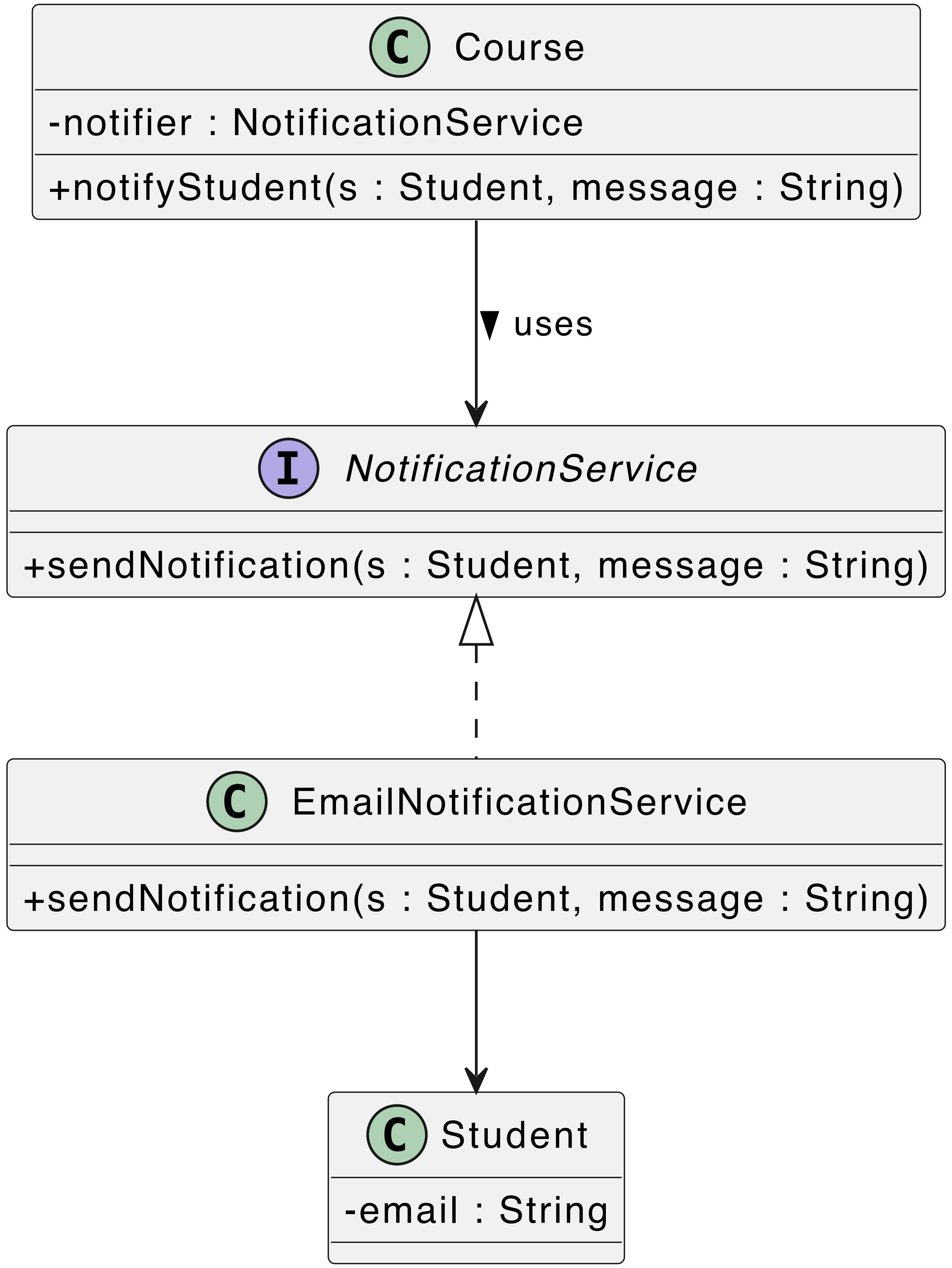
Course need to send an Email to notify

- One way is that course can use the Email class directly to notify

- If Email class changes or new notification needs to be incorporated - Course has to change

- Interface provides a good abstraction

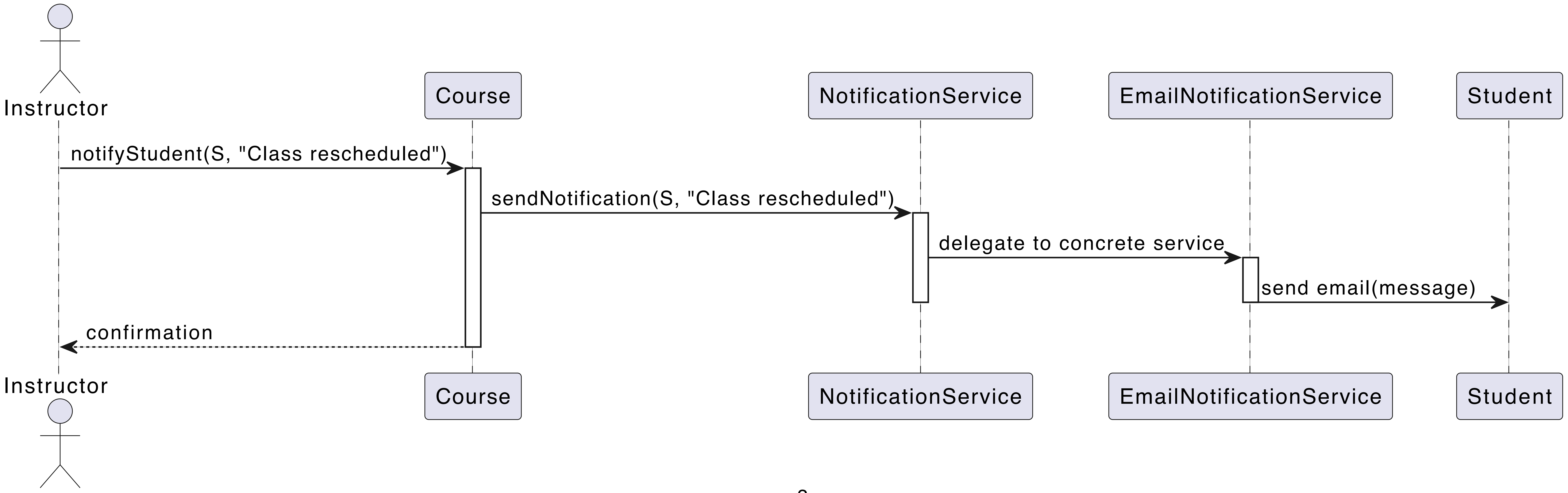
- Always reduce dependency in concrete class


C Course
-notifier : NotificationService
+notifyStudent(s : Student, message : String)
▼ uses
I NotificationService
+sendNotification(s : Student, message : String)
C EmailNotificationService
+sendNotification(s : Student, message : String)
C Student
-email : String

GRASP: Low Coupling

Course need to send an Email to notify

- One way is that course can use the Email class directly to notify
- If Email class changes or new notification needs to be incorporated - Course has to change
- Interface provides a good abstraction
- Always reduce dependency in concrete class



GRASP: Low Coupling

Low Coupling means assigning responsibilities so that classes and components depend as little as possible on one another

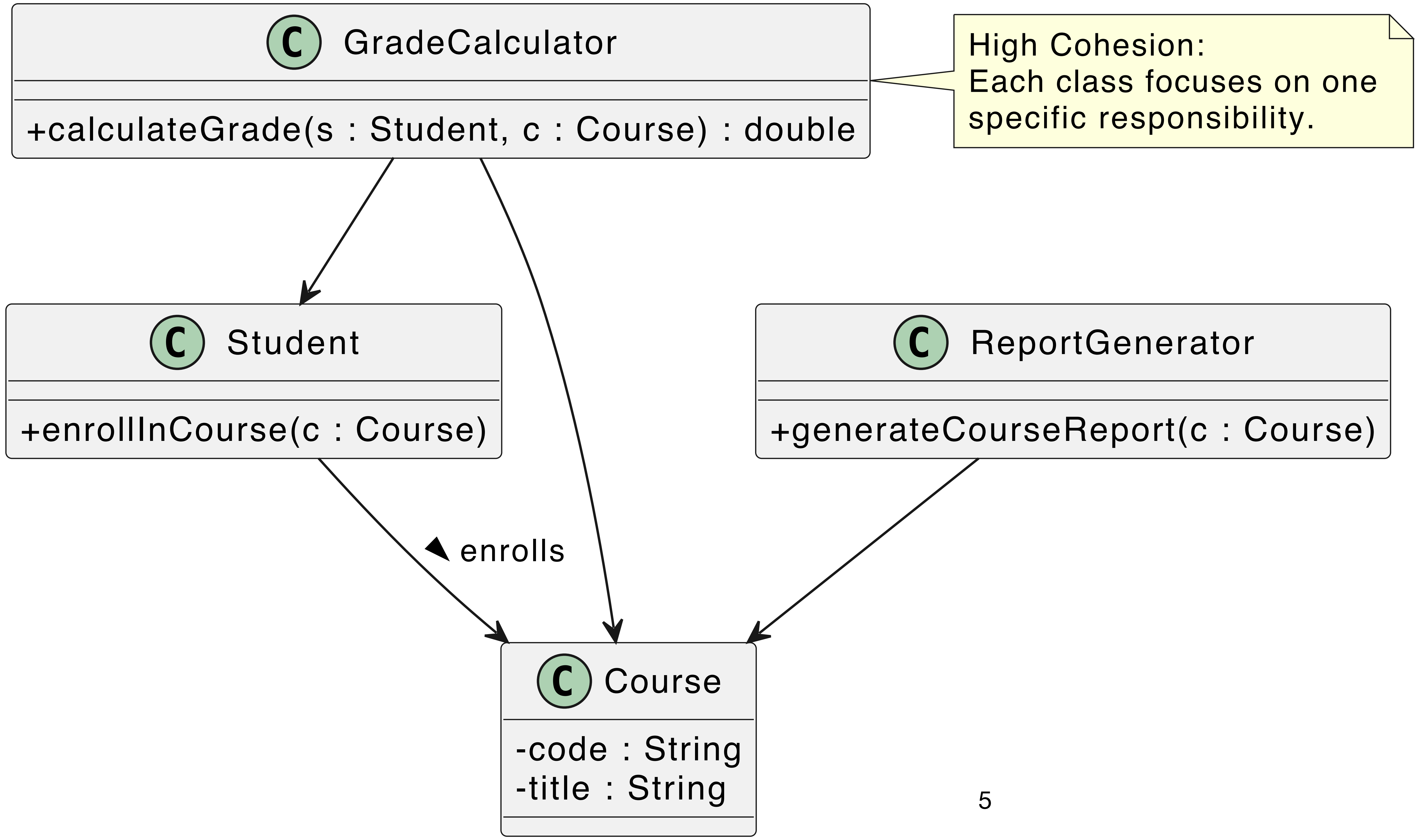
- Coupling is the degree of dependancy between classes
- High coupling - change in one can impact all dependant ones!
- Design with goal to minimise the impact of a change
  - Assign responsibilities such that to reduce coupling
  - Given two alternatives, chose the one that minimizes coupling

GRASP: High Cohesion

Adding Grade and Report generation functionality

- Student class itself can have all functionalities - High cohesion

- Separate responsibility such that each does exactly one

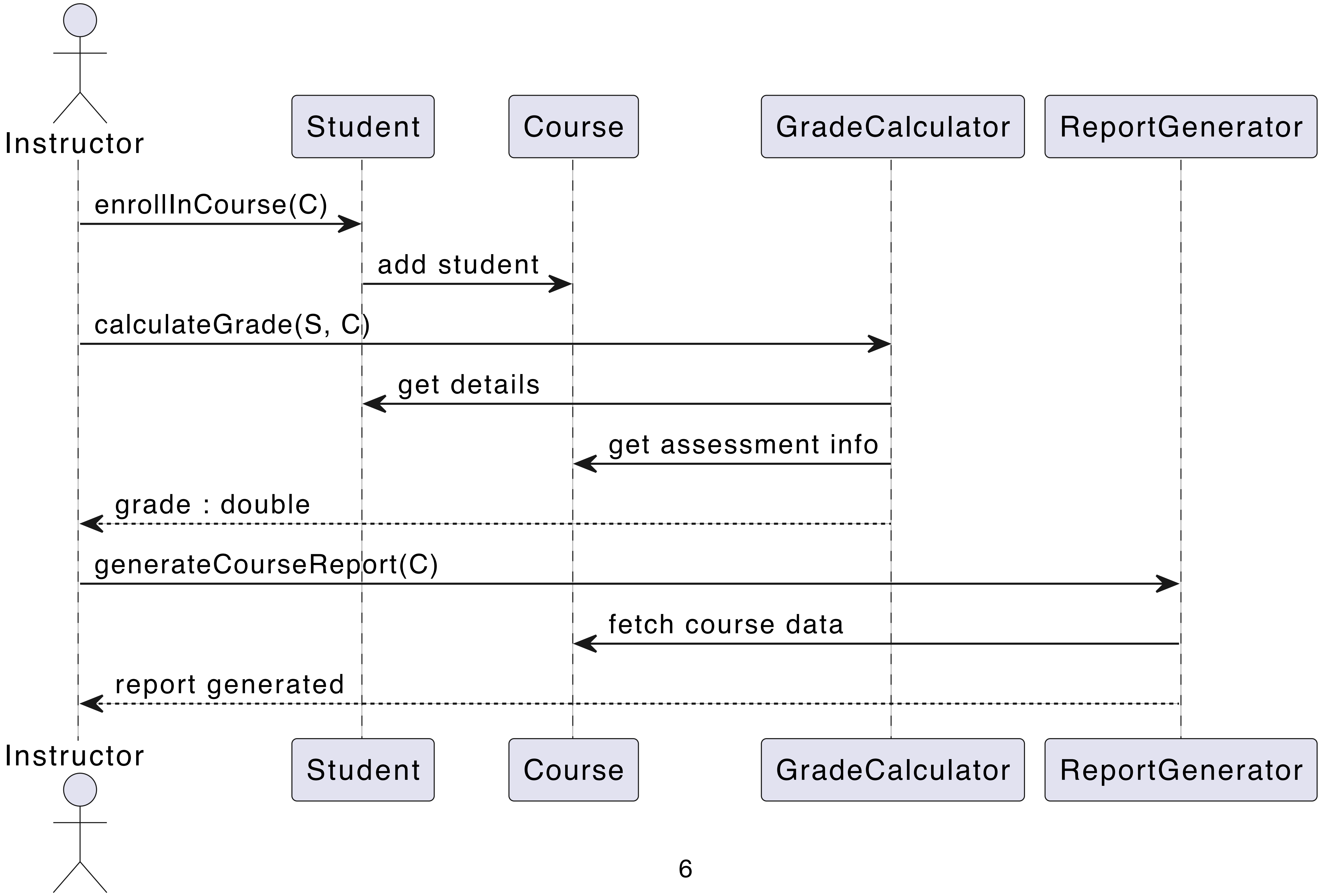

C GradeCalculator
+calculateGrade(s : Student, c : Course) : double
High Cohesion:
Each class focuses on one
specific responsibility.
C Student
+enrollInCourse(c : Course)
C ReportGenerator
+generateCourseReport(c : Course)
◀ enrolls
C Course
-code : String
-title : String

GRASP: High Cohesion

Adding Grade and Report generation functionality

- Student class itself can have all functionalities - High cohesion

- Separate responsibility such that each does exactly one



GRASP: High Cohesion

Assigning responsibilities so that classes have closely related and focused functions => each class does one thing well.

- Do one thing and do it well!
- Give end-to-end responsibility to one class
- Reduce communication
- Low cohesion comes with lot of issues
    - Complex, bulky classes
    - Harder to debug and makes it difficult to reuse

GRASP: Protected Variation

Identify points of predicted change or instability in a system and protect other parts of the system from those variations through stable interfaces or abstractions.

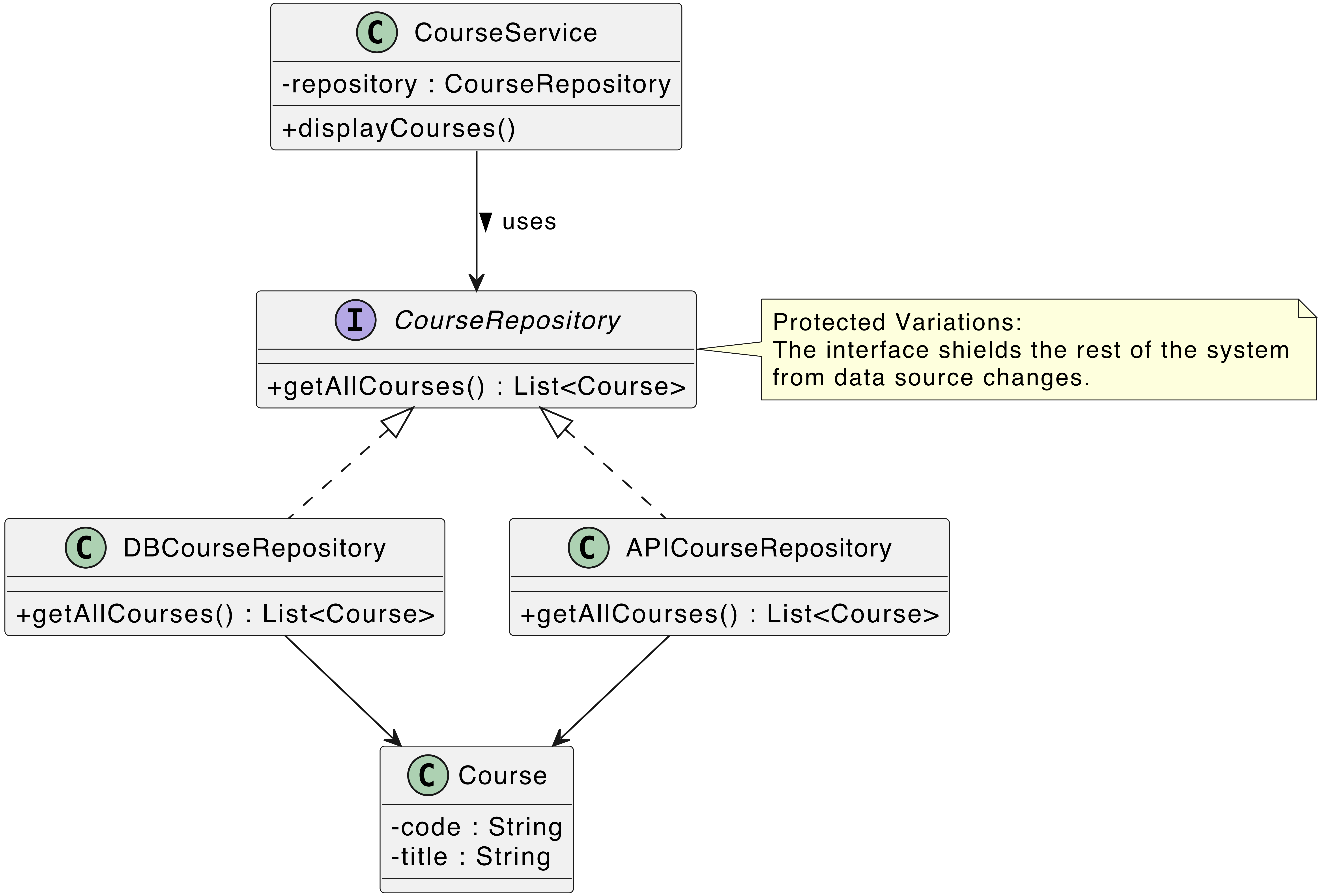
How to protect part of a class from changes in part of another class?

- Related to ensuring low coupling
- Code of a part of class B is protected from changes in code of part A
- Introduce interface around the unstable part of the codebase
- Systems evolve: technologies, databases, or APIs may change
- Anticipate where change may happen, protect that through:
    - Interfaces
    - Adapter classes in between
    - …

Protected Variation

System receives course data from data sources

- Assume we tie it to one data source: database
- It could also leverage more source later like external APIs some file import => Modify the class
- Create interface to abstract



CourseService
-repository : CourseRepository
+displayCourses()
uses
CourseRepository
+getAllCourses() : List<Course>
DBCourseRepository
+getAllCourses() : List<Course>
APICourseRepository
+getAllCourses() : List<Course>
Course
-code : String
-title : String

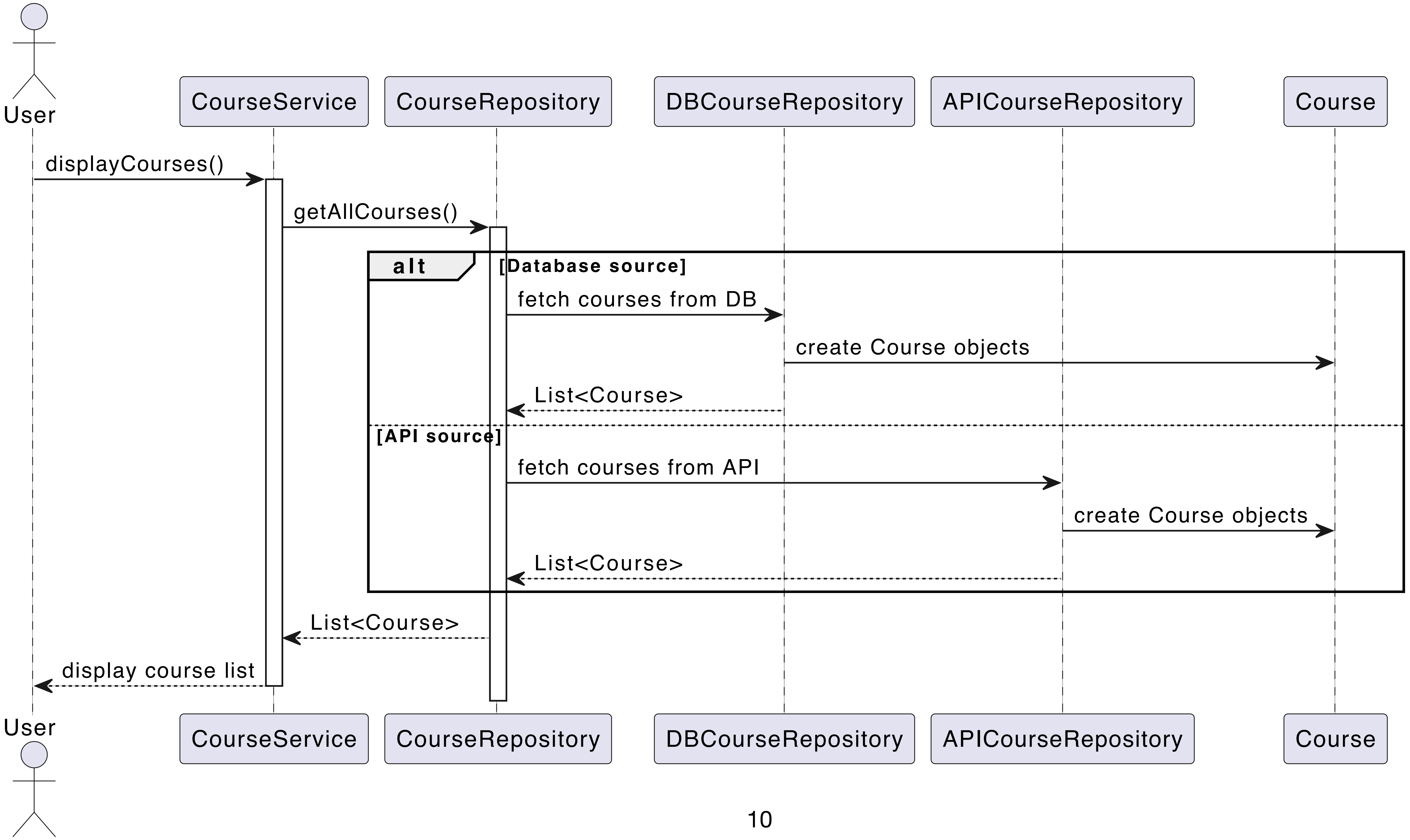
Protected Variations:
The interface shields the rest of the system
from data source changes.

Protected Variation

System receives course data from data sources

- It could also leverage more source later like external APIs, some file import => Modify the class

- Create interface to abstract



Indirection

Introduces an intermediate object to mediate between other components or services, decoupling them and reducing direct dependencies.

How to ensure one class can communicate to another class without knowing it well?

- Another principle/pattern to reduce coupling
- Introduce a new class between two classes A and B
- Changes in A or B doesn't affect each other. The intermediary absorbs the impact
- Introduces a class as opposed to protected variation

.

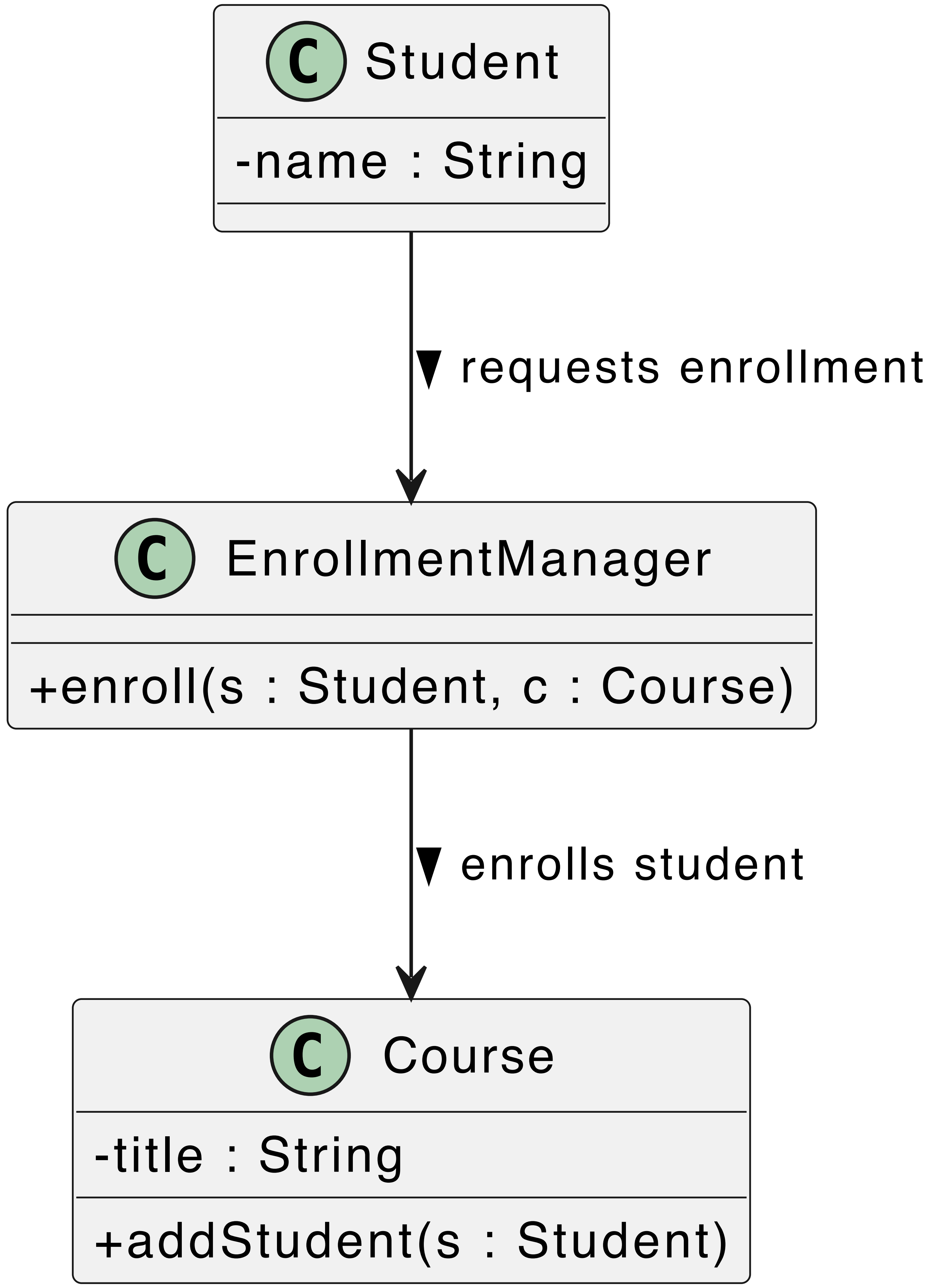
Indirection

Student wants to enroll in a course

- One way is to have an enrolment functionality in the course or in student

- Creation of direct dependency between classes is not always good!

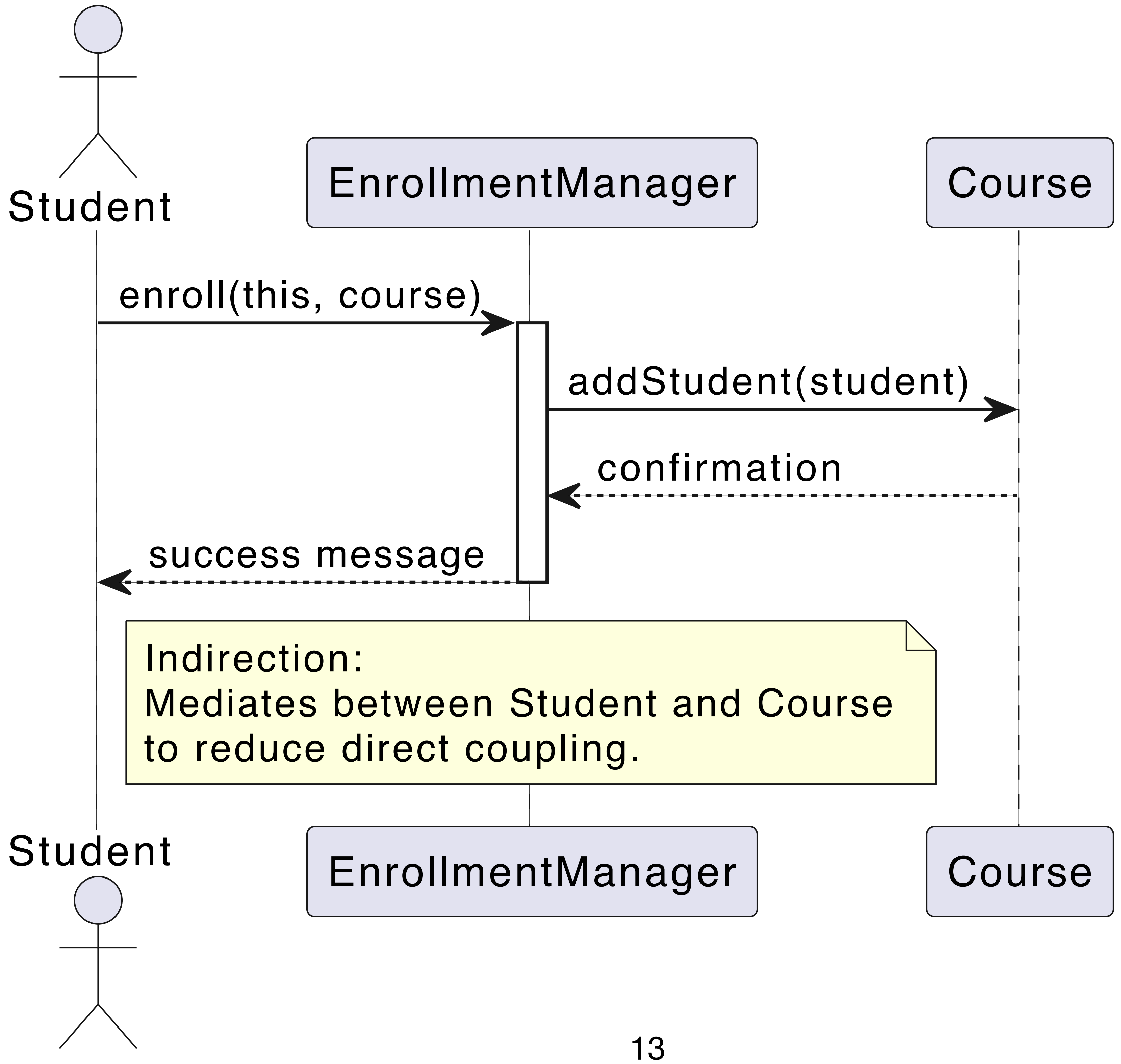
- Create an intermediate class that manages this functionality



Indirection

Student wants to enroll in a course

- One way is to have an enrolment functionality in the course or in student
- Creation of direct dependency between classes is not always good!
- Create an intermediate class that manages this functionality



Polymorphism

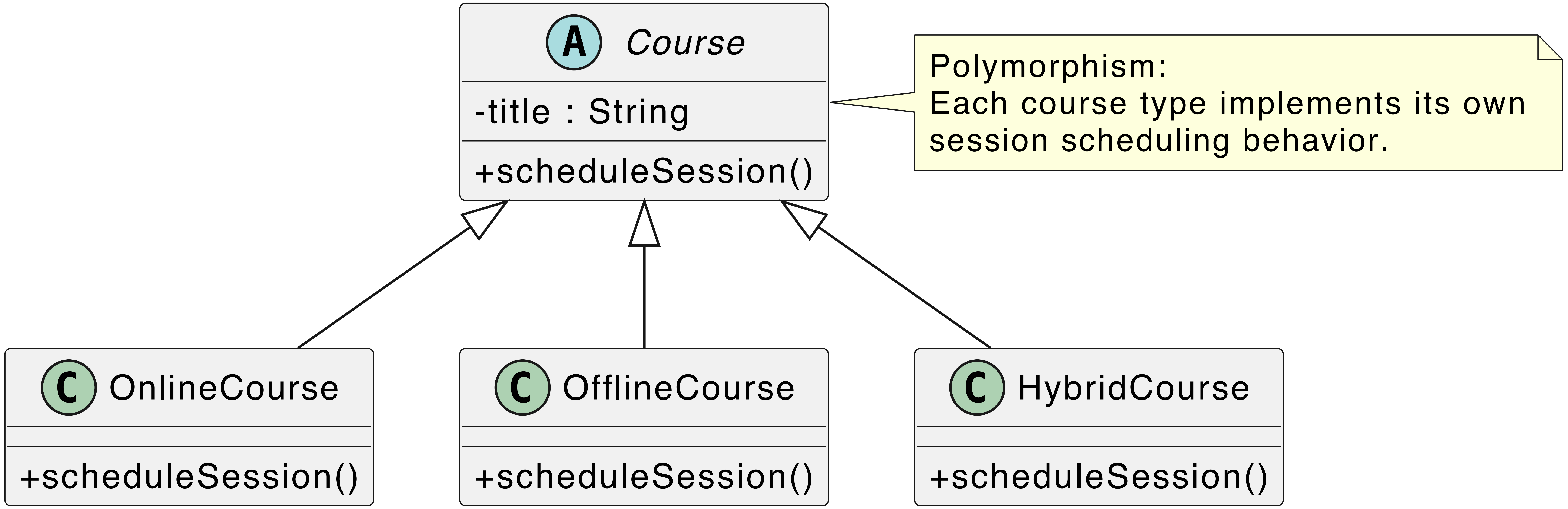
It assigns responsibility for behavior that varies by type to the types themselves, avoiding type-checking conditionals.

- How to decouple clients from different ways of accomplishing a single task?
  - Contributes to low coupling
  - Several ways to accomplish a task or a functionality
  - Achieved through interfaces, overloading methods of super classes
- Replace conditional logic on type with polymorphic calls

Polymorphism

Many ways to deliver a course and each has some rules

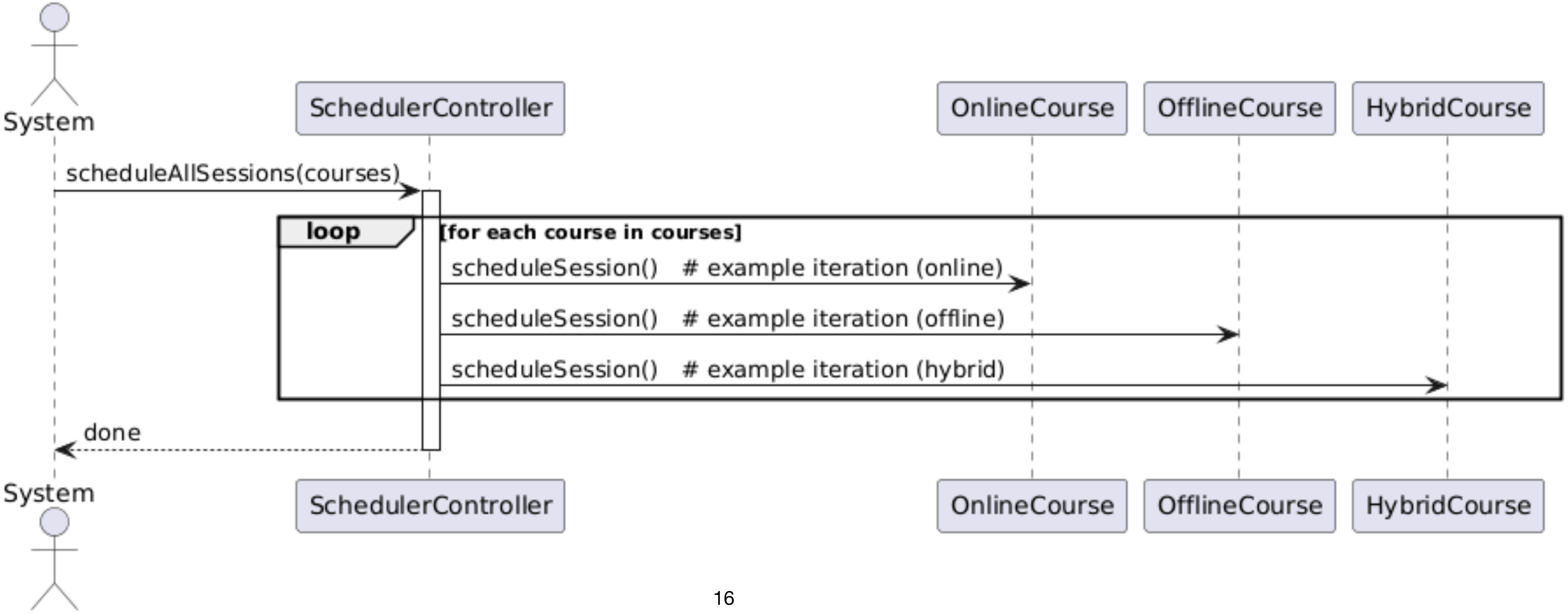
- Three types of course delivery: Online, Offline, Hybrid
- Scheduling for each course has to be done differently
- Create one schedule session functionality with if conditions: Anti-pattern
  - For any change or any addition, the if condition has to be modified



Polymorphism

Many ways to deliver a course and each has some rules

- Three types of course delivery: Online, Offline, Hybrid
- Scheduling for each course has to be done differently
- Create one schedule session functionality with if conditions: Anti-pattern
    - For any change or any addition, the if condition has to be modified


System
SchedulerController
OnlineCourse
OfflineCourse
HybridCourse
scheduleAllSessions(courses)
loop
[for each course in courses]
scheduleSession() # example iteration (online)
scheduleSession() # example iteration (offline)
scheduleSession() # example iteration (hybrid)
done
System
SchedulerController
OnlineCourse
OfflineCourse
HybridCourse

Pure Fabrication

Create a non-domain (“made up”) class to take on responsibilities when doing so improves cohesion, reduces coupling, or enables reuse, even if that class doesn’t map to a real-world concept.

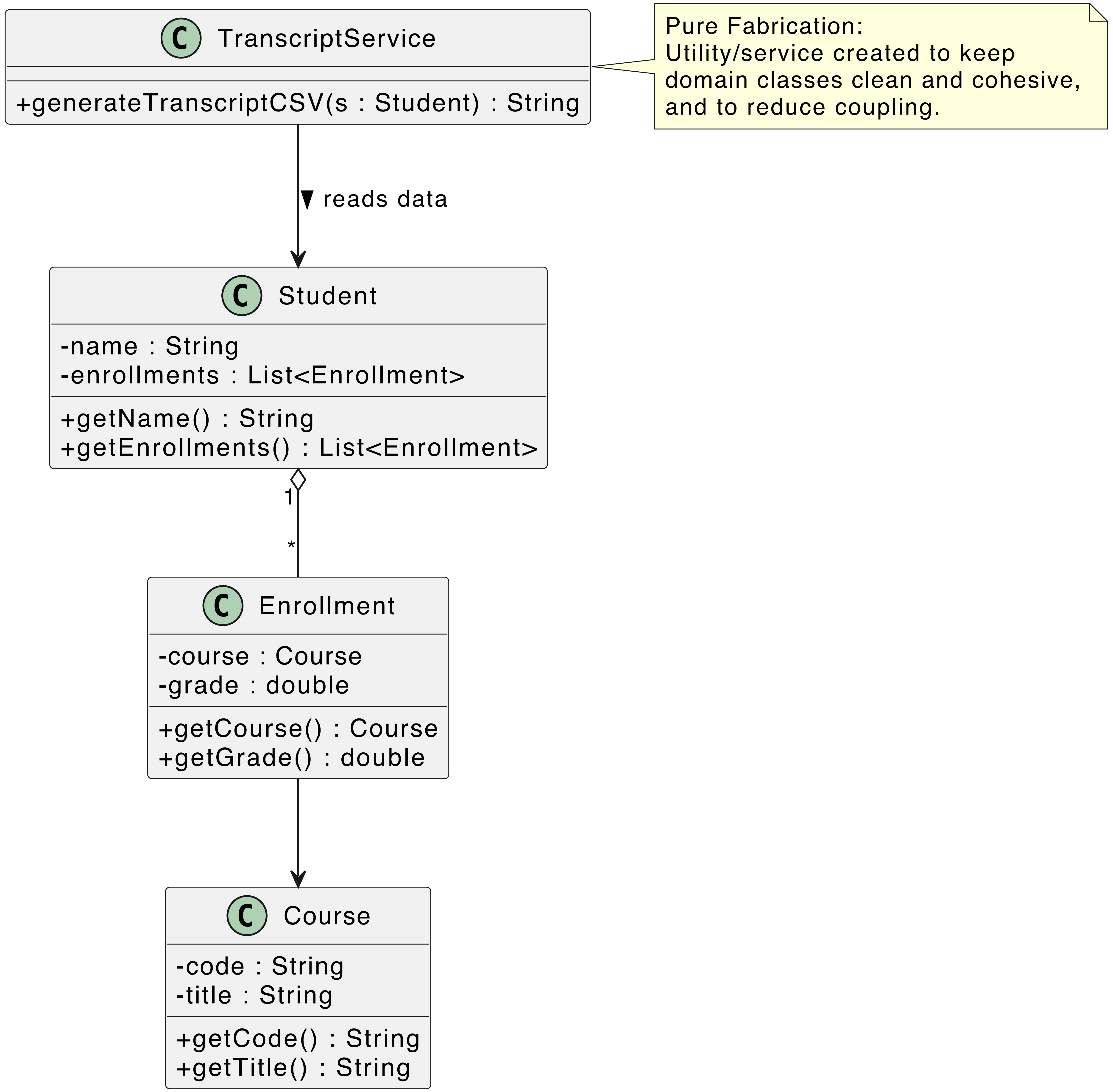
Whom to assign responsibility to when it does not fit into either of the classes?

- Promotes cohesion and reduces coupling
- Sometimes responsibility needs to be assigned but does not fit in one class
- Create a new class (which does not map to domain) for the responsibility

Pure Fabrication

Generate Transcript

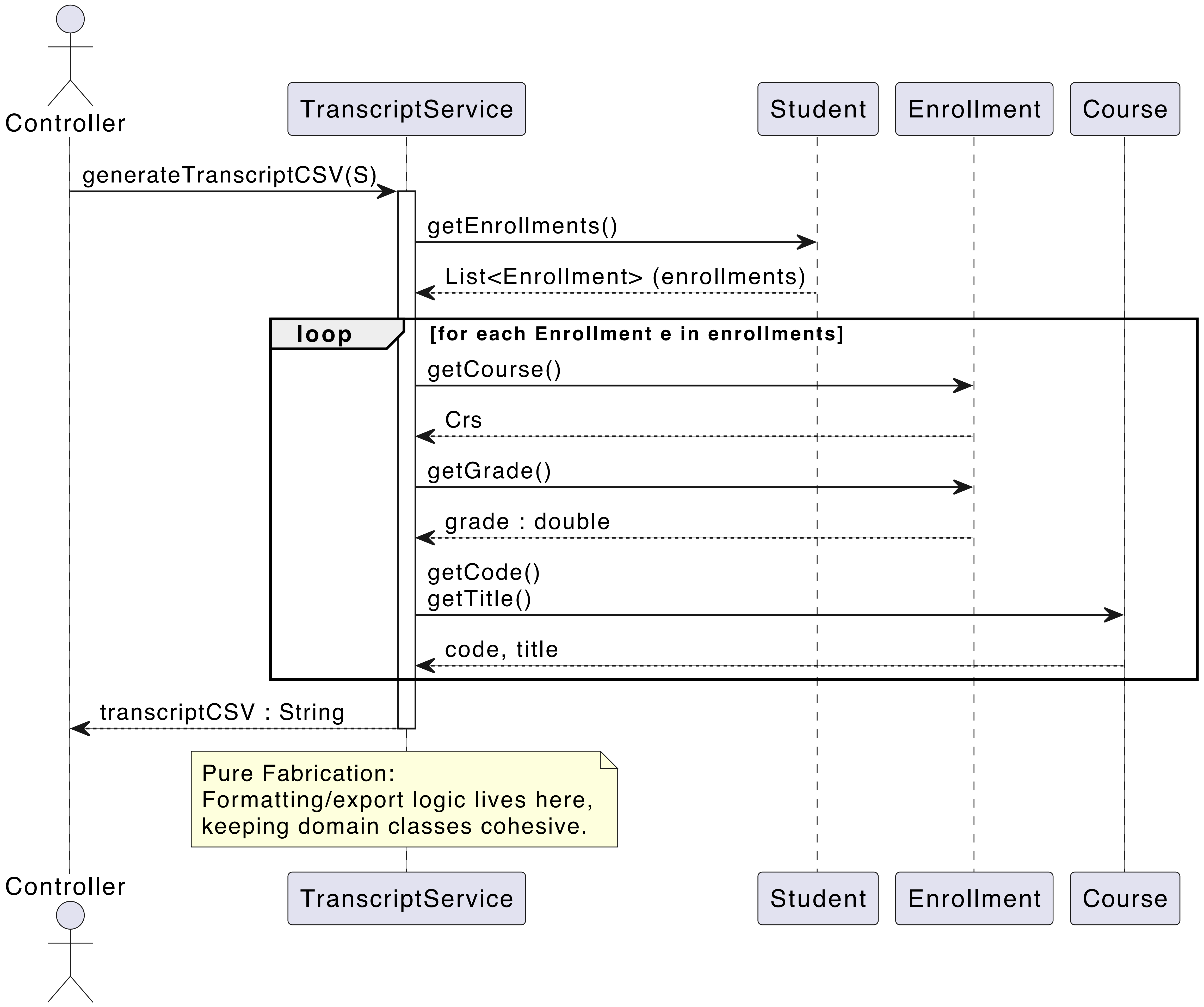
- Transcript generation functionality can be kept inside the student class
  - It may need to support more formats
  - Lot of changes may happen
- Create a new class that only has to handle this responsibility
  - This may not always work!



Pure Fabrication

Generate Transcript

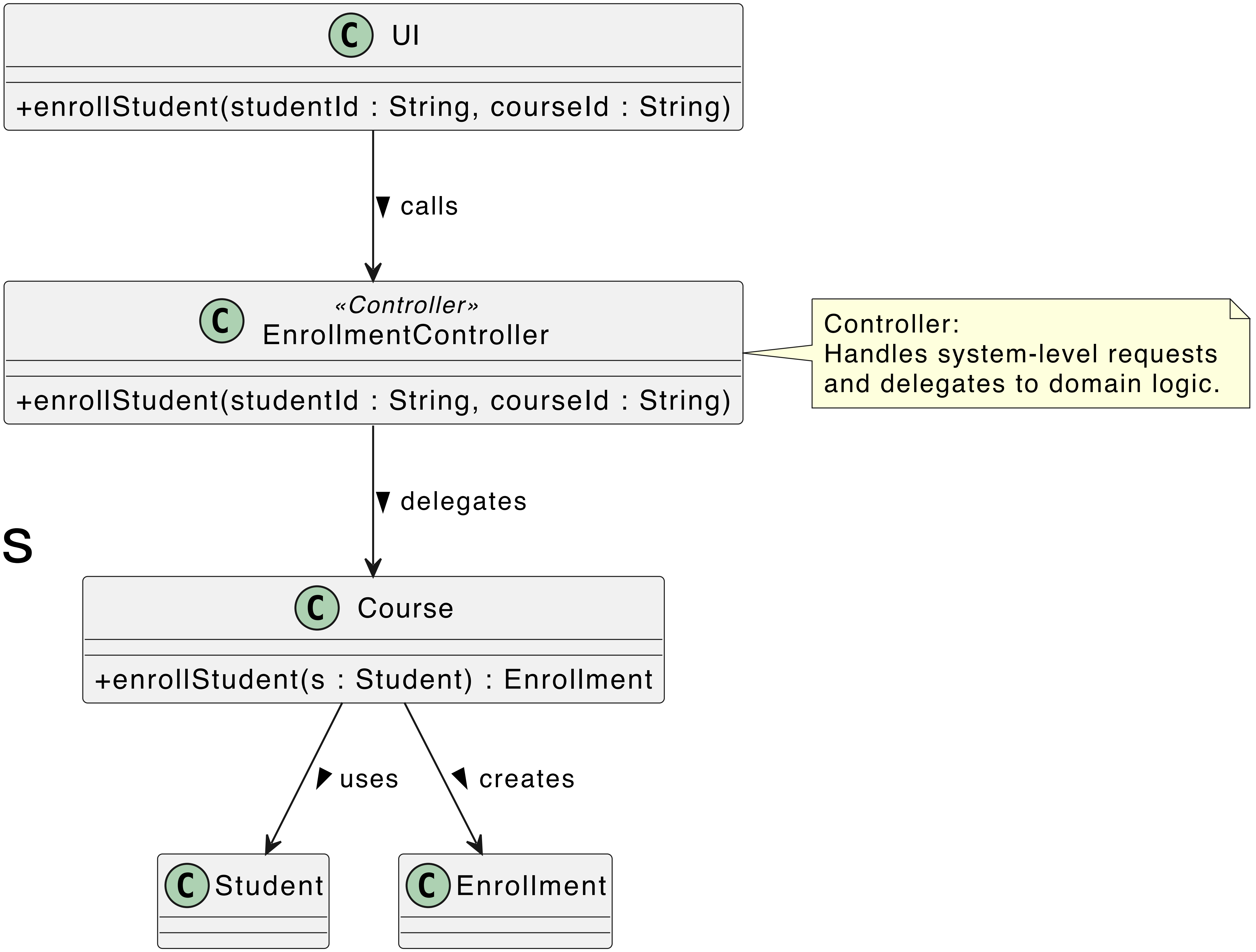
- Transcript generation functionality can be kept inside the student class
    - It may need to support more formats
    - Lot of changes may happen
- Create a new class that only has to handle this responsibility
    - This may not always work!



Controller

Assign the responsibility of handling a system operation (from the UI or external interface) to a Controller object

- Subtype of pure fabrication
- Very common in UI applications
- Separate concerns between two classes
- Does not map to any domain object
- Eg: Enroll student to a course
  - Enrolment controller can handle!



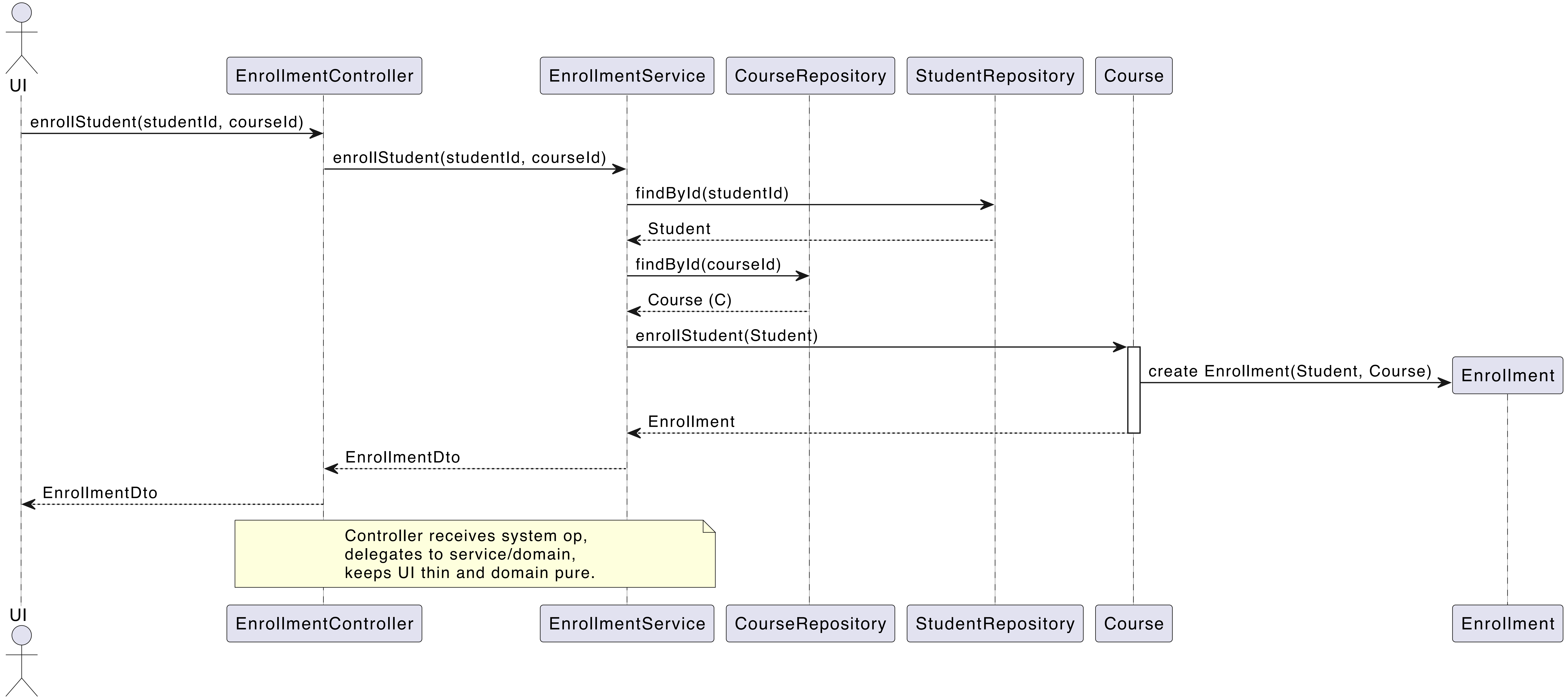
C UI
+enrollStudent(studentId : String, courseId : String)
calls
«Controller»
C EnrollmentController
+enrollStudent(studentId : String, courseId : String)
delegates
C Course
+enrollStudent(s : Student) : Enrollment
uses
creates
C Student
C Enrollment

Controller:
Handles system-level requests
and delegates to domain logic.

Controller

Assign the responsibility of handling a system operation (from the UI or external interface) to a Controller object

- Very common in UI applications
- Eg: Enroll student to a course
    - Enrolment controller can handle!



Controller receives system op,
delegates to service/domain,
keeps UI thin and domain pure.

SOLID Design Principle

S: Single Responsibility Principle

Handle one responsibility and do it well (High Cohesion, Information expert,..)

O: Open for extension, closed for modification

No need to modify classes for changes (polymorphism, protected variations)

L: Liskov Substitution Principle

Subtypes should be replaceable without breaking behaviours (polymorphism)

I: Interface Segregation Principle

Don’t depend on unused methods (low coupling, controller)

D: Dependency Inversion Principle

Depend on abstractions and not implementations (low coupling, indirection, protection variation)

.

Some Takeaways

- Who gets what responsibility
- Reduce coupling and high cohesion
- Use abstractions, interfaces, polymorphism when necessary
- Think about separation of concerns
- Ultimately its also about simplicity and understandability
- Design principles are not rules -> its more guidelines to make system design effective and efficient.

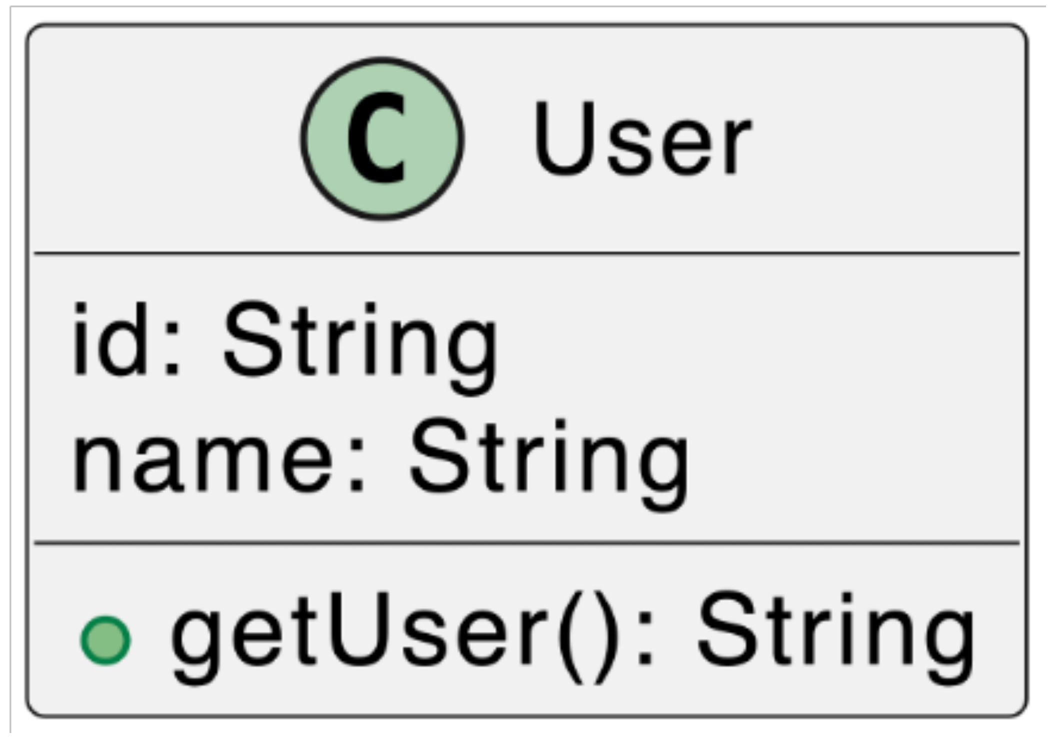
Patterns, Patterns Everywhere…

- We have a natural tendency to look for patterns in anything and everything
- Pattern of grades for courses
- Patterns of buildings
- Pattern of questions in question papers
- Climate patterns (rainfall, summer, …)
- …

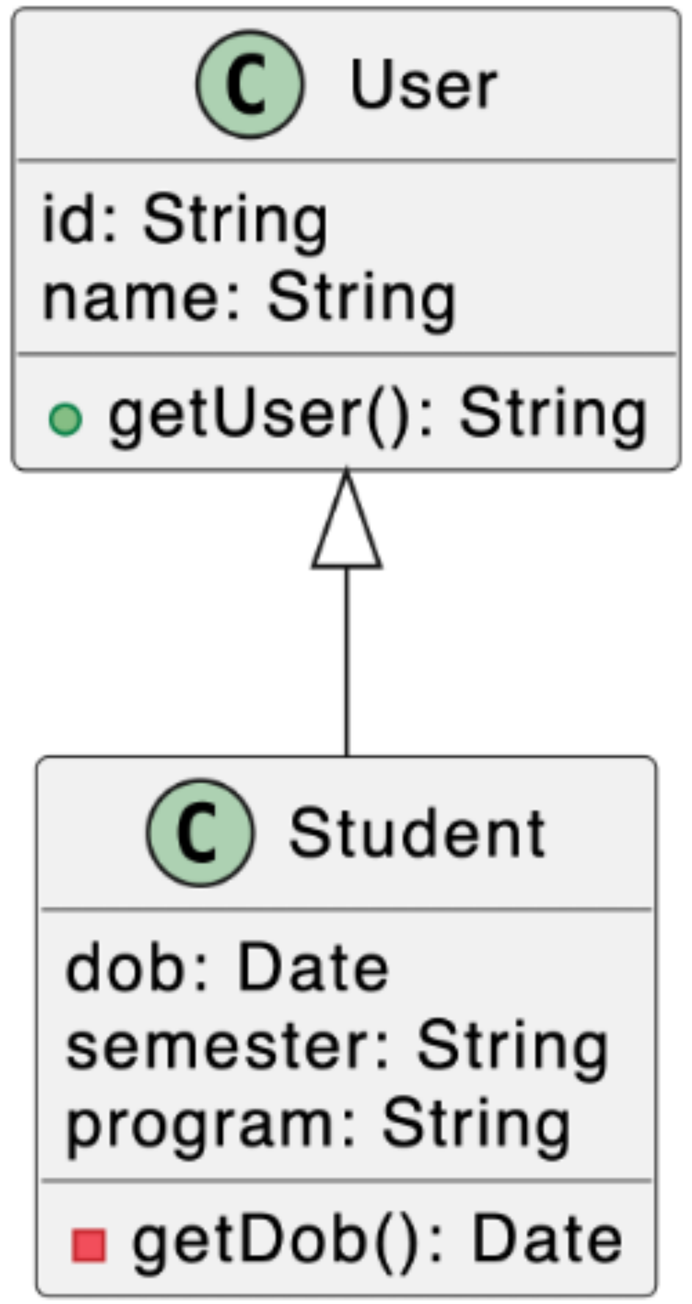
What about Software?

Many patterns to design and build software systems

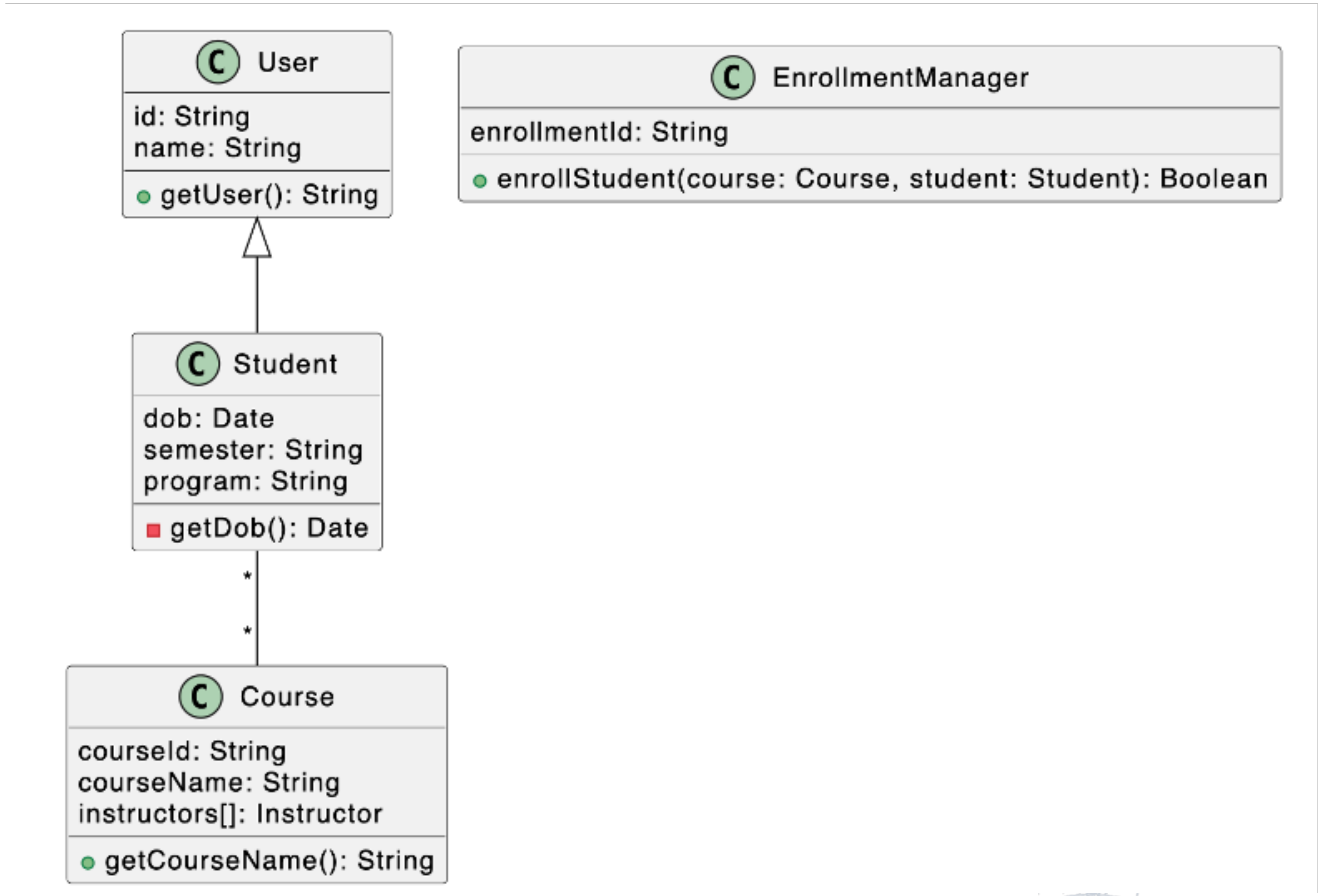
- Architectural Patterns [Higher Level]
- Design Patterns [Lower level]



Extract classes



Relations



Assign Responsibilities

Design Patterns

Each Pattern describes a problem which **occurs over and over again** in our **environment** and then **describes the core of the solution** to that problem, in such a way that you can **use this solution a million times over**, without ever doing it the same way twice -- Christopher Alexander

Patterns captures {Context, Problem, Solution}

What are some of the patterns that you can think of?

Design Patterns

- Principles, relationships and techniques for creating **reusable** OO design
- Identifies participating objects, their roles, responsibilities and relationships
- **Not about** Linked Lists, hash tables, etc.
- The are low level structures inside classes
- **Not about** complex domain specific design or design of subsystems
- Domain specific design is more at high level – Architectural level

Elements of Design Pattern

Mainly divided into three based on the purpose they serve:

- Creational, Structural and Behavioural

Each category has a purpose, a set of patterns that work in a different scope:

- Class or object

There are a total of 23 classic patterns: **Gang of Four (GOF) patterns**

The famous book Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

Classification of Design Patterns

Creational

Class - Defer creation to subclasses

Object – Defer creation to another object

Structural

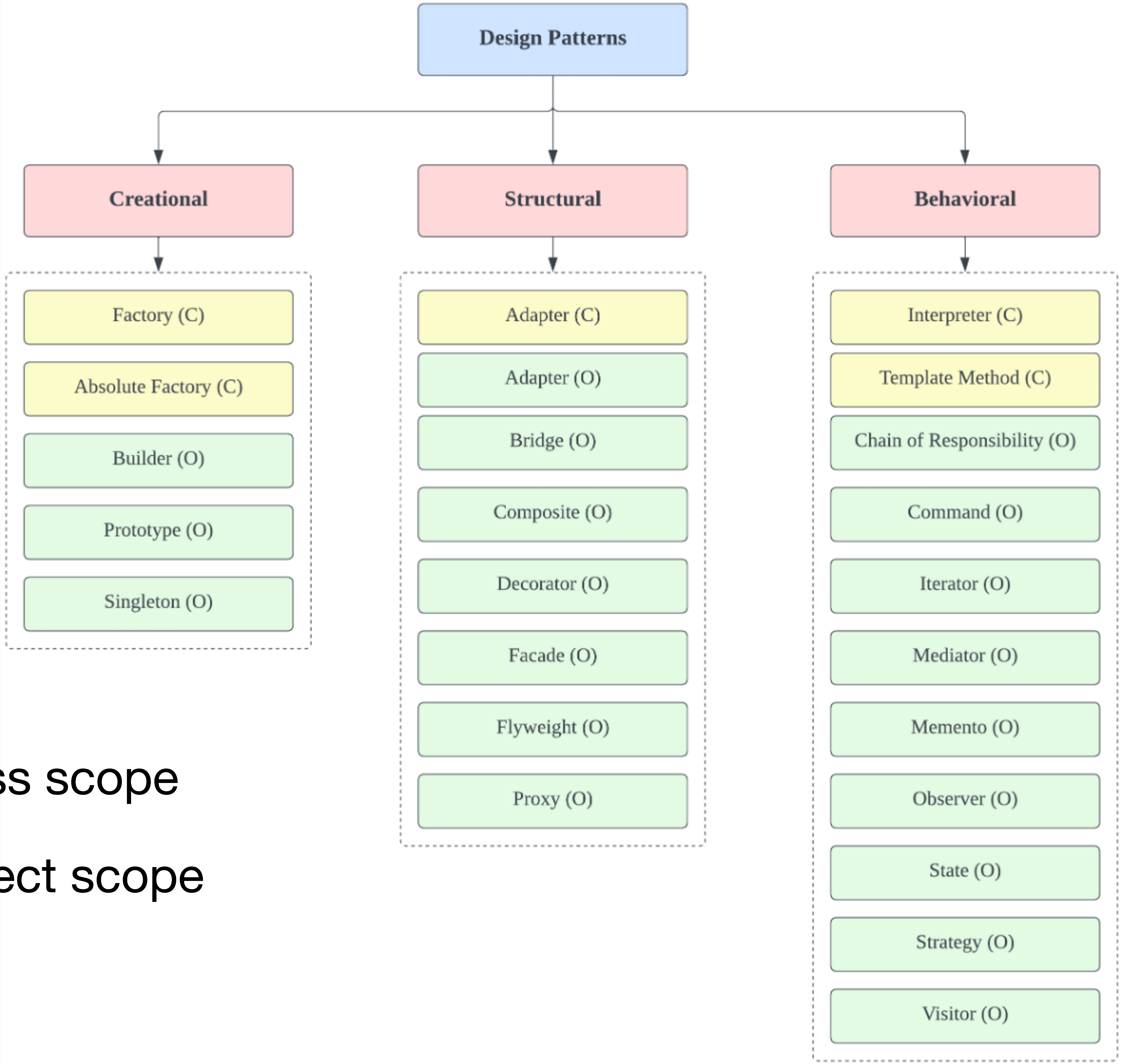
Class – Structure via inheritance

Object – Structure via Composition

Behavioral

Class – algorithms/control via inheritance

Object – algorithms/control via object groups



Design Patterns

Creational
- Factory (C)
- Absolute Factory (C)
- Builder (O)
- Prototype (O)
- Singleton (O)

Structural
- Adapter (C)
- Adapter (O)
- Bridge (O)
- Composite (O)
- Decorator (O)
- Facade (O)
- Flyweight (O)
- Proxy (O)

Behavioral
- Interpreter (C)
- Template Method (C)
- Chain of Responsibility (O)
- Command (O)
- Iterator (O)
- Mediator (O)
- Memento (O)
- Observer (O)
- State (O)
- Strategy (O)
- Visitor (O)

C - Class scope

O - Object scope

Four Elements of a Pattern

Pattern Name: Handle to describe a design problem

Problem: When to apply the pattern, preconditions, special relationships, etc.

Solution: Elements that make up the design, relationships and collaborations

Not a particular solution but an abstract representation with potentials

Consequences: Results and trade-off of applying a given pattern

Perform cost-benefit analysis

Each Pattern is described in detail following a standard template

Thank you

Email: karthik.vaidhyanathan@iiit.ac.in

X: @karthi_ishere