COMP 3111
SOFTWARE ENGINEERING

LECTURE 16
SYSTEM ANALYSIS AND DESIGN

LEARNING OBJECTIVES

1. Understand the purpose and importance of system analysis and design in software development.

2. Know the major activities that take place during system analysis and design.

3. Know how to realize design goals and deal with the implementation environment.

4. Know what are architectural patterns and design patterns and when to use them.

SYSTEM ANALYSIS AND DESIGN OUTLINE

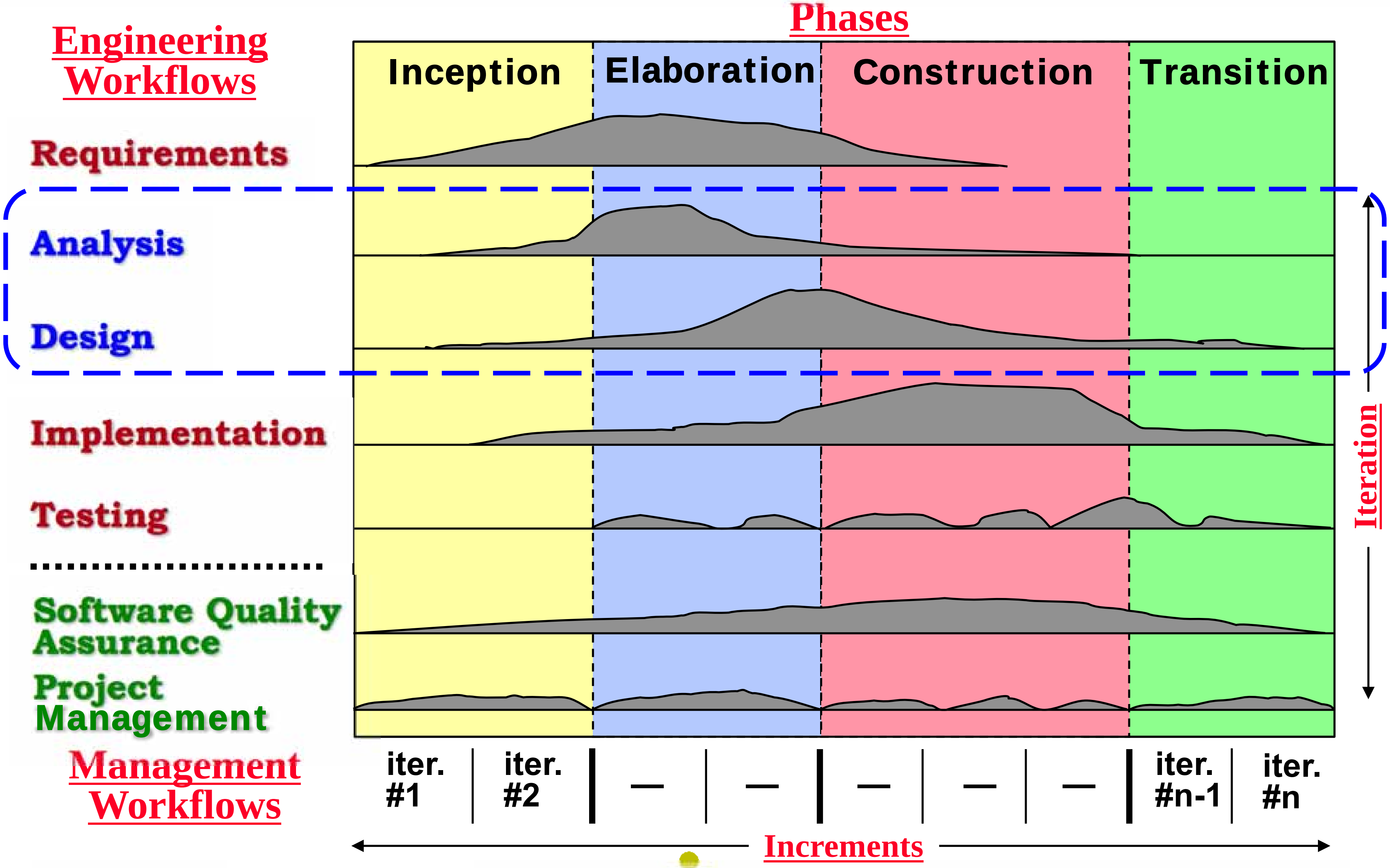
System Analysis and Design Overview

- Life Cycle Role
- The Purpose and Importance of System Analysis and Design
- Realizing Design Goals
- Dealing with the Implementation Environment

System Analysis and Design Activities

- Architectural Analysis and Design
- Use–case Analysis
- Class Design
- Object Behaviour Analysis: State Machine Diagrams
- Design Patterns
- Anti Patterns

SYSTEM ANALYSIS AND DESIGN LIFE CYCLE ROLE



THE PURPOSE OF SYSTEM ANALYSIS AND DESIGN

System analysis and design **structures the use-case model into a form that is both robust and maintainable, adapts it to the implementation environment and prepares it for implementation.**

Objective: sound and stable architecture; blueprint for implementation

The system is *gradually structured* into basic “parts” with well-defined relationships among them in two stages:

1. **building the system skeleton** in which the parts come together to produce a stable basis.

   ☞ **The *skeleton* is provided by the software architecture.**

2. **growing the system** in which the parts aggregate around the skeleton to produce a finer system structure.

   ☞ **The *growing* is accomplished by the analysis and design workflows.**

☞ **Technical solutions are applied from the solution domain.**

THE IMPORTANCE OF SYSTEM ANALYSIS AND DESIGN

Why not go straight to implementation?

- Since the use-case model is not sufficiently formal, we need to:
  - complete specification to remove any ambiguities and redundancies.
  - describe requirements more precisely using developer's language.
  - structure requirements for easier understanding and maintenance.

- We also need to consider issues previously ignored, such as how to deal with:
  - nonfunctional requirements.
  - implementation environment.

It gives us a more precise and detailed understanding of the requirements *before* we start to implement.

USE-CASE MODEL	ANALYSIS & DESIGN MODEL
Uses client's language	Uses developer's language
External view of the system	Internal view of the system
Use cases structure the external view	Classes and subsystems structure the internal view
Used primarily as a contract between client/developer to agree on system functionality	Used primarily by developers to understand how to implement the system
May contain ambiguities, redundancies, inconsistencies, etc.	Should not contain ambiguities, redundancies, inconsistencies, etc.
Captures system functionality	Outlines system implementation

REALIZING DESIGN GOALS

We need to decide how design goals will be realized.

Design goals come mainly from nonfunctional requirements (i.e., system qualities that we try to "optimize").

☞ **Usually only a small subset of the design goals can be effectively realized simultaneously.**

THEREFORE we need to prioritize design goals and possibly develop trade-offs against each other as well as against managerial goals (e.g., schedule, budget).

Examples: space vs. speed; delivery time vs. functionality, etc.

REALIZING DESIGN GOALS:
IDENTIFYING DESIRABLE SYSTEM QUALITIES

Design goals guide the decisions made by developers, especially when implementation trade-offs are needed.

Dependability
- robustness
- reliability
- availability
- fault tolerance
- security
- safety

Maintenance
- extensibility
- modifiability
- adaptability
- portability
- readability
- traceability

Cost
- development
- deployment
- upgrade
- maintenance
- administration

Performance
- response time
- throughput
- memory

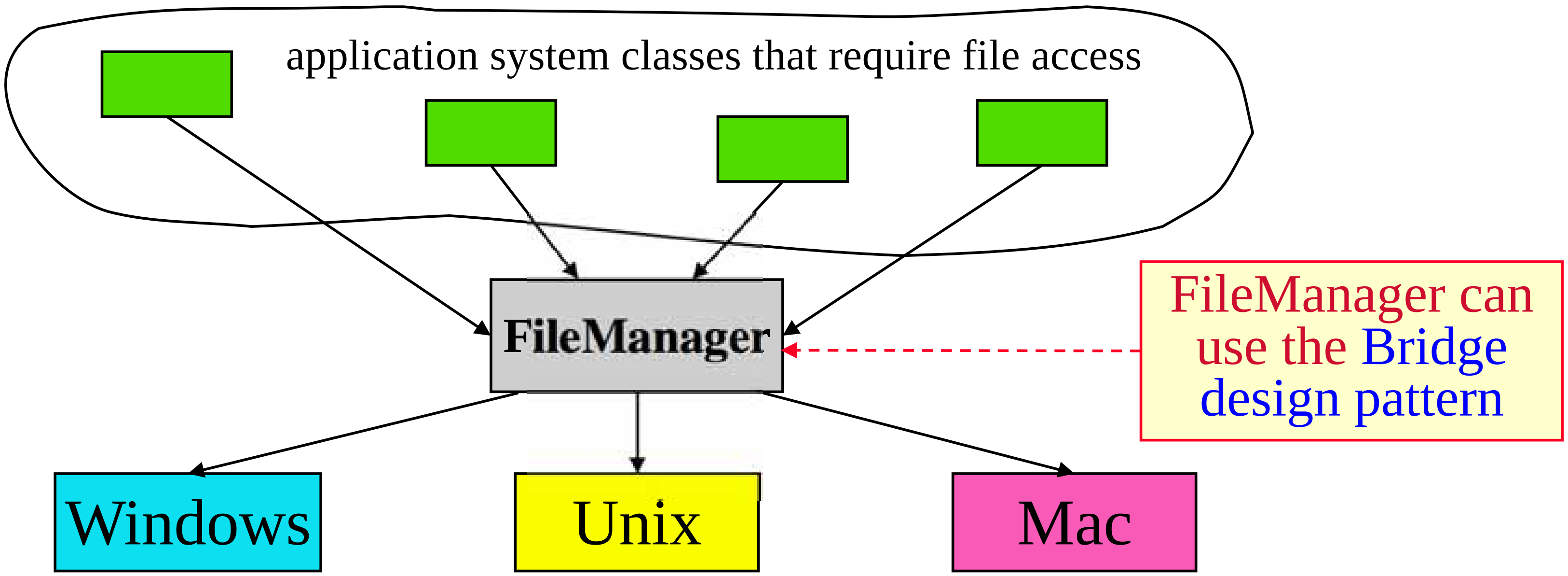
End user
- utility
- usability

7.2 7.4.1

DEALING WITH THE IMPLEMENTATION ENVIRONMENT

General design strategy for technical issues:
Encapsulate and isolate the implementation environment.

Create "bridge" classes that represent occurrences of components in the implementation environment.



FileManager can use the Bridge design pattern

☞ **We often need to define many additional classes to deal with the implementation environment so as to facilitate maintenance.**

ARCHITECTURAL ANALYSIS AND DESIGN

- Architectural analysis and design focuses on
  - understanding how a system should be organized.
  - designing the overall structure of the system.
- The main structural components and the relationships between them are identified in terms of subsystems.

  ☞A subsystem encapsulates its contents and provide *services* (operations) to other subsystems via an interface.

  ☞A subsystem provides information hiding.
- Nonfunctional requirements are highly dependent on the system architecture.

Architectural analysis and design is an essential tool for managing complexity.

ARCHITECTURAL ANALYSIS AND DESIGN: LAYERS AND PARTITIONS

- **Subsystem layers**

1. closed layered — Each layer can depend only on the layer immediately below it.

   ☞Lower coupling (but more overhead).

2. open layered — Each layer can depend on any layer below it.

   ☞Higher coupling (but less overhead).

Usually there are 3 to 5 subsystem layers in practice.

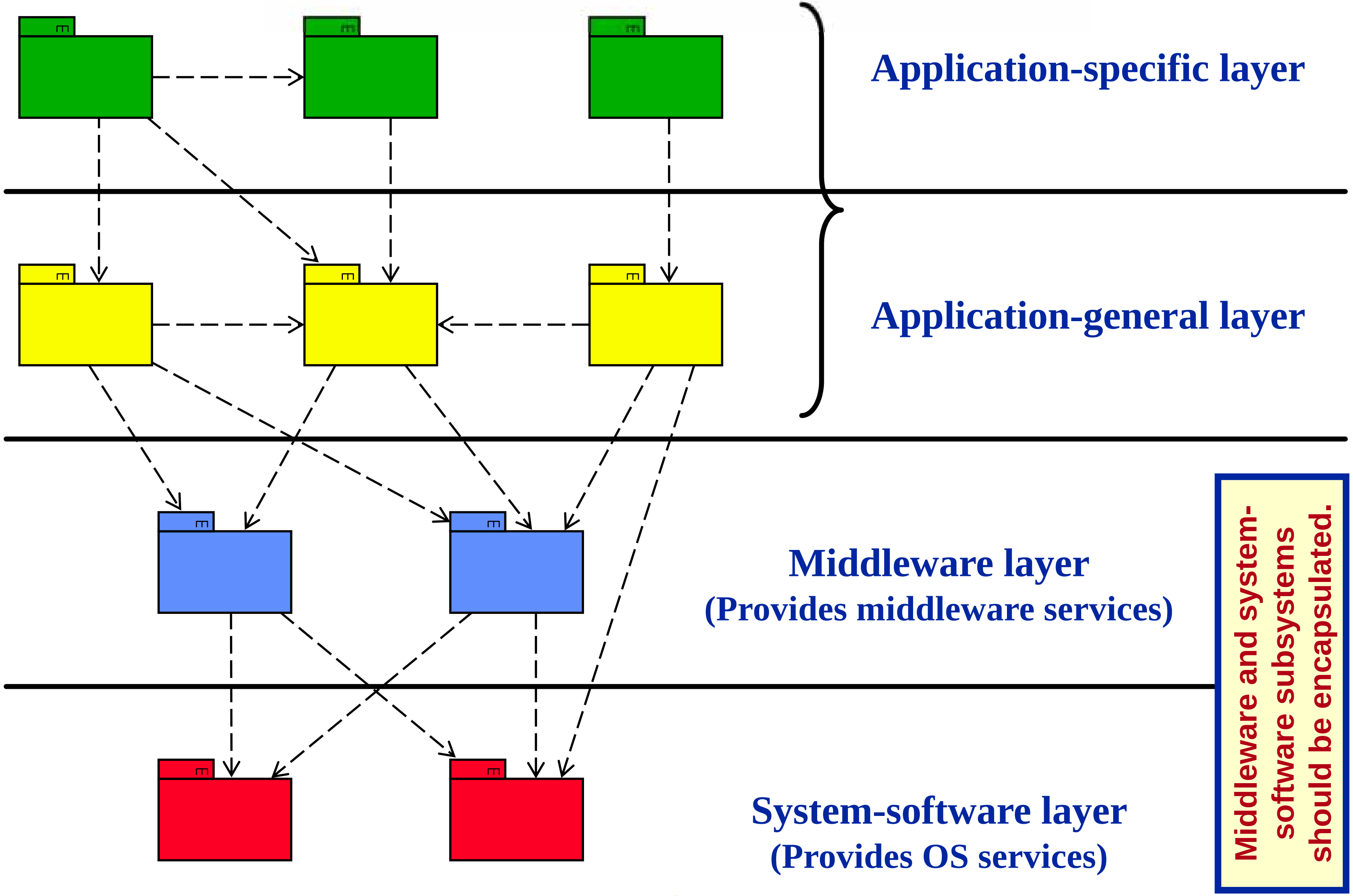
- **Subsystem partitions**

  – Partitions organize subsystems in one layer into different services.

  ☞This results in peer to peer services within a layer.

Partitioning and layering produce a subsystem hierarchy.

ARCHITECTURAL ANALYSIS AND DESIGN: LAYERS AND PARTITIONS



Application-specific layer

Application-general layer

Middleware layer
(Provides middleware services)

System-software layer
(Provides OS services)

Middleware and system-software subsystems should be encapsulated.

ARCHITECTURAL ANALYSIS AND DESIGN: ARCHITECTURE DECISIONS

- Based primarily on the nonfunctional requirements.

Performance

- Use a small number of subsystems to localize critical functions.

Security

- Use a layered architecture with most critical assets in the innermost layer.

Safety

-

Availability

- Include redundant subsystems to allow replacement or update without stopping the system.

Maintainability

-

ARCHITECTURAL PATTERNS

An *architectural pattern (style)* specifies how to organize and connect a set of software components.

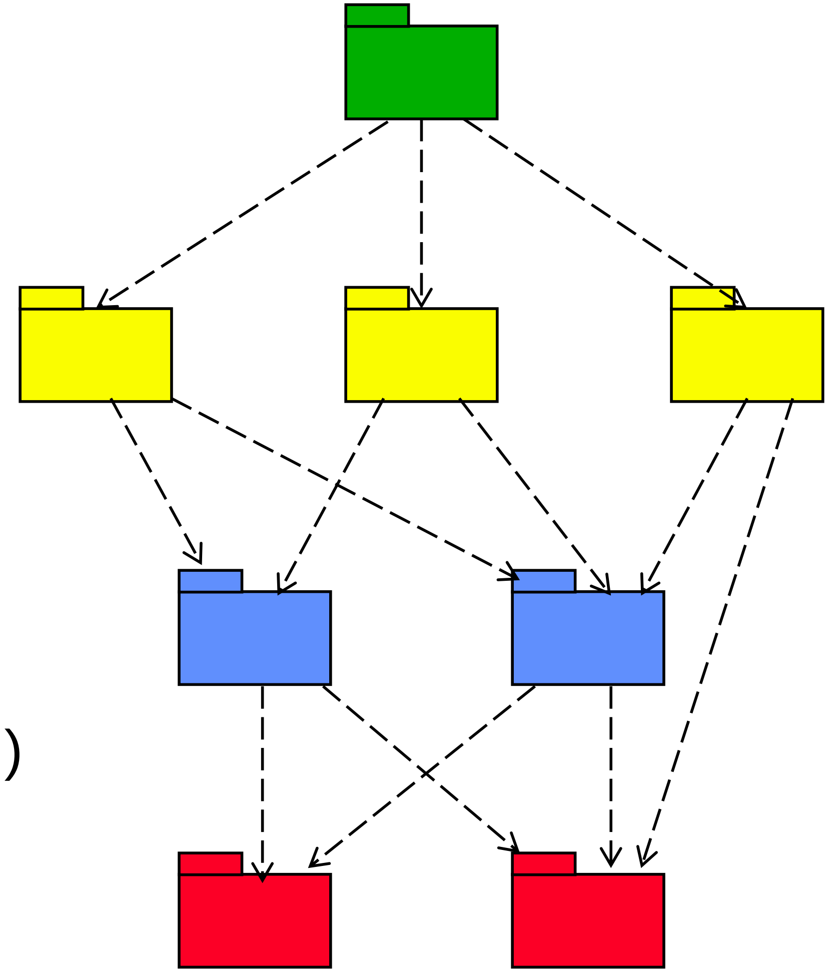
- It is a blueprint for system organization that defines the software architecture of a system.

- It includes the specification of:
  - system decomposition
  - global control flow
  - error handling policies
  - inter module/component communication protocols

- Most large software systems use several architectural patterns in different parts of the system.

ARCHITECTURAL PATTERN: MULTI-LAYER

Goal: **To build the software in layers so that each layer communicates only with the layers below it.**



Top layer: user interface

Middle layers: application functions

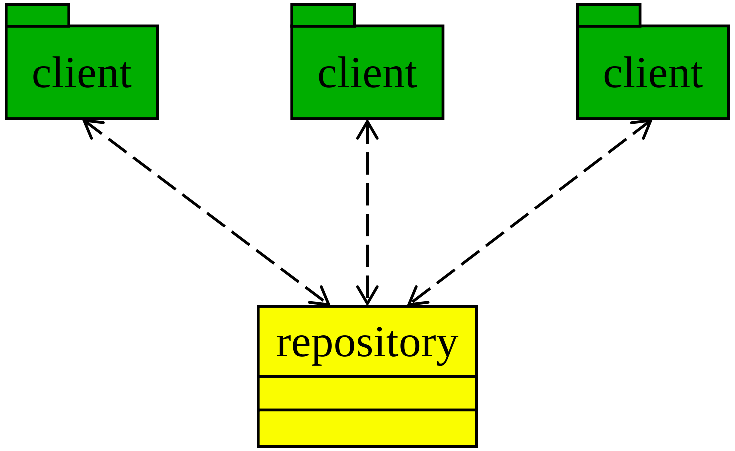
Bottom layers: common services (e.g., data storage, communication, etc.)

☞ Each layer has a well-defined API defining the services it provides.

Often combined with other patterns.

ARCHITECTURAL PATTERN: REPOSITORY

Goal: To centralize the management of data.



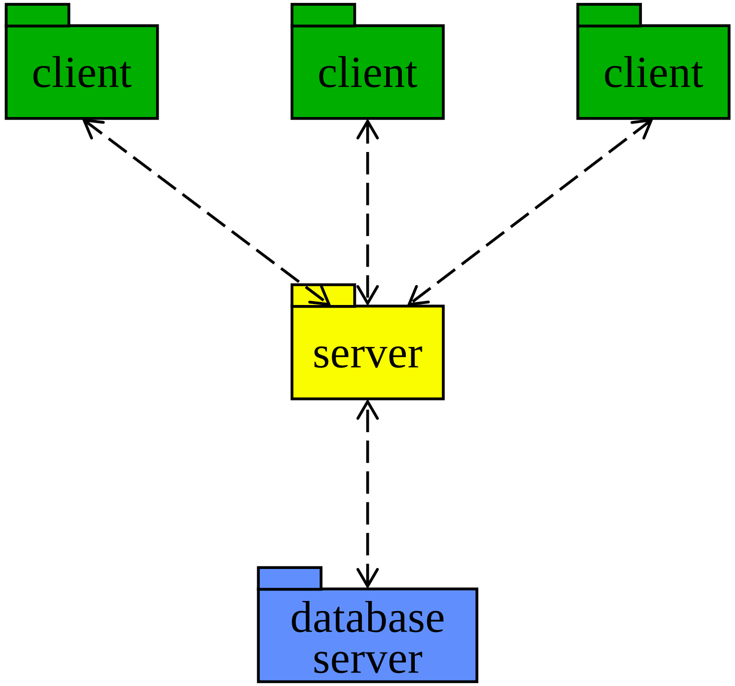
- Control flow can be dictated by:
  - repository via triggers on the data (e.g., change to data).
  - subsystems using locks to synchronize.

- Possible drawbacks:
  - the repository can be a performance and modifiability bottleneck.
  - there is high coupling between the repository and subsystems.

A typical pattern for applications requiring data management.

ARCHITECTURAL PATTERN: CLIENT-SERVER

Goal: To separate and distribute the system's functionality.



three-tier

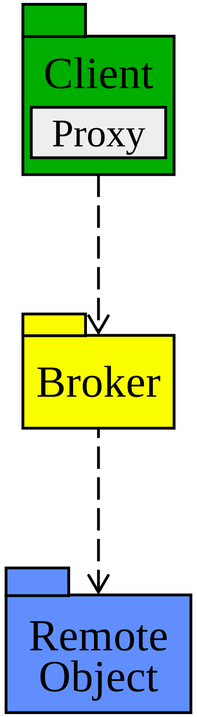
The server communicates with both a client (usually via the Internet) and with a database server (usually via an intranet).

peer-to-peer

Each subsystem can be both a server and a client. Control flow within each subsystem is independent from others except to synchronize requests.

ARCHITECTURAL PATTERN: BROKER

Goal: **To distribute aspects of the system transparently to different nodes.**



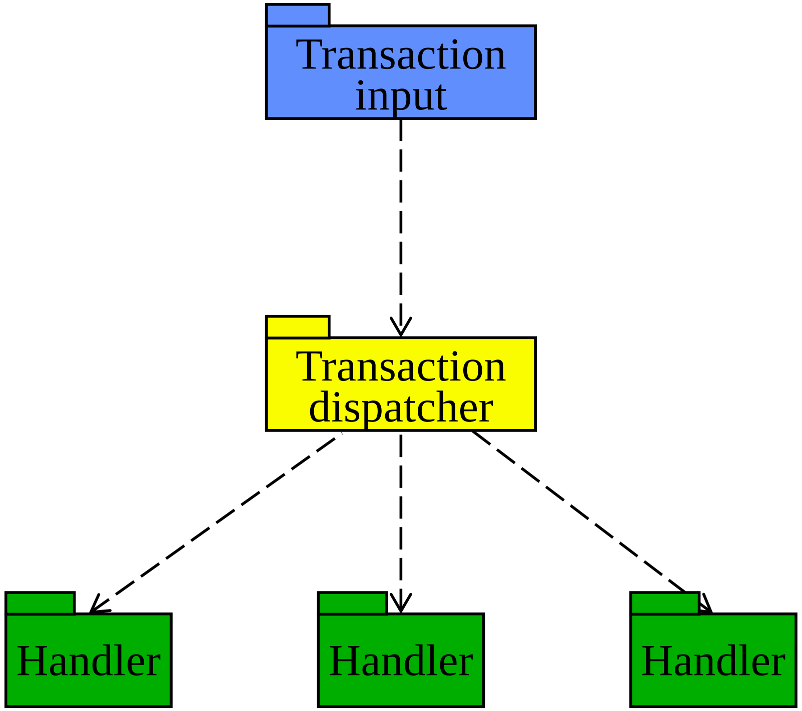
Examples:

- CORBA - Common Object Request Broker Architecture
- Microsoft COM - Common Object Model

The Proxy design pattern can be used to implement the broker architecture.

ARCHITECTURAL PATTERN: TRANSACTION PROCESSING

Goal: **To direct input to specialized subsystems (handlers).**

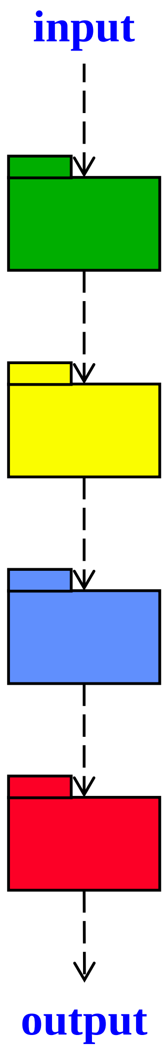


- A transaction dispatcher decides what to do with each input transaction, dispatching a procedure call or message to a handler that will handle the transaction.

Example: A database engine.

ARCHITECTURAL PATTERN: PIPE-AND-FILTER

Goal: To provide flexibility, modifiability and reusability of subsystems.

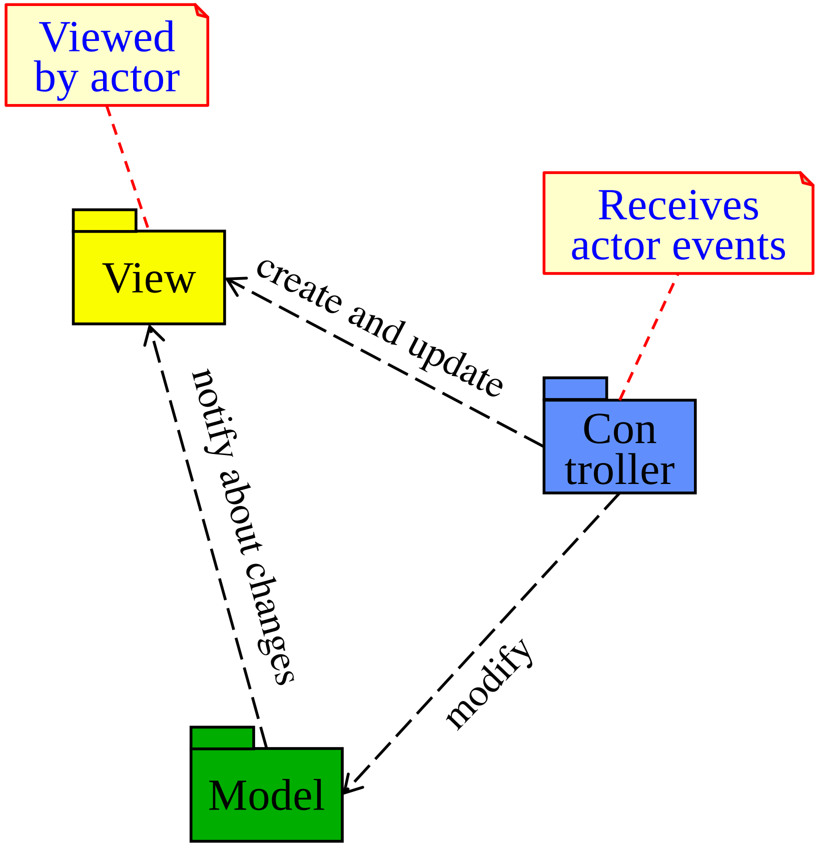


- A stream of data is passed through a series of filters (subsystems), called a pipeline, each of which transforms it in some way.
- Each filter is executed concurrently.

ARCHITECTURAL PATTERN: MODEL-VIEW-CONTROLLER (MVC)

Goal: To separate the user interface layer from other parts of the system.

- The model contains the classes (data) to be viewed and manipulated.
- The view contains objects used to render the data from the model and also the various controls with which the user can interact.
- The controller contains the objects that control and handle the user's interaction with the view and the model.



The Observer design pattern is normally used to separate the model from the view.

USE-CASE ANALYSIS: ANALYSIS CLASSES

An ***analysis class*** is an **abstraction** of one or several classes in the final system implementation.

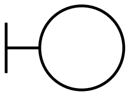
The design phase usually has many—up to 5 times—more classes!

☞ **We focus on handling *only* functional requirements.**

- Class descriptions are conceptual, not implementation oriented:
  - attribute types are conceptual, *not* programming language types.
  - behavior is defined by textual descriptions of *responsibilities.*
  - relationships are conceptual, *not* implementation oriented.

- Analysis classes are one of the three following stereotypes:
  - boundary
  - entity
  - control

ANALYSIS CLASS: BOUNDARY CLASS



boundary class name

«boundary»
boundary
class name

A *boundary class* models the interaction between the system and its actors.

- Represents an *abstraction* of UI elements (windows, forms, panes, etc.) or devices (printer interfaces, sensors, terminals, etc.).

- Interacts with *actors* outside the system as well as with *objects* inside the system.

- The description should be at a fairly high conceptual level.

DESCRIBE WHAT *NOT* HOW!

(Do not describe every button, menu item, etc. of a UI!).

Encapsulates and isolates changes in the system's interface.

IDENTIFYING BOUNDARY CLASSES

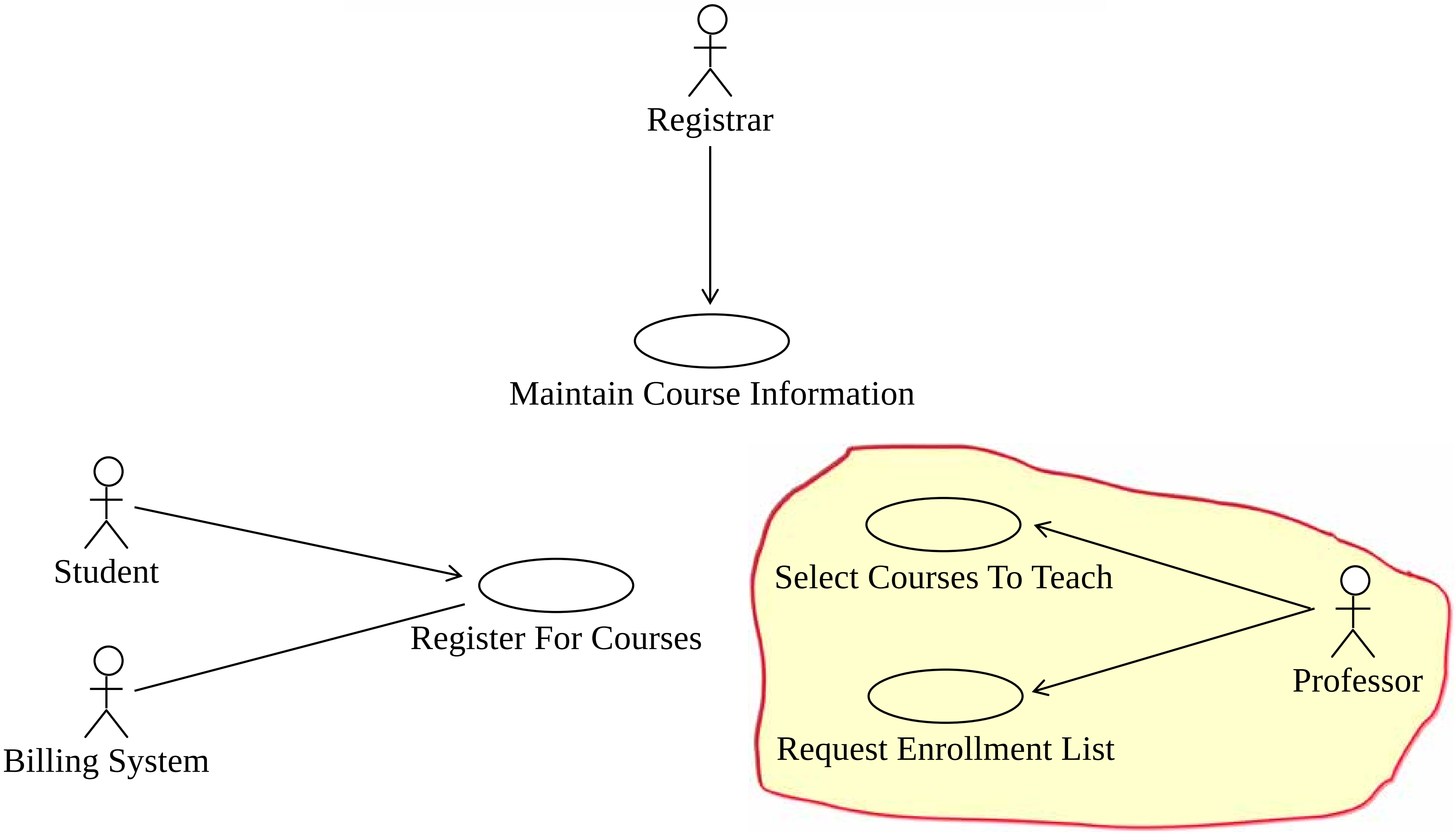
Usually found from use case descriptions → start from the actors:

- Identify forms and windows needed to enter data into the system.
- Identify notices and messages the systems uses to respond.
- Do not model the visual aspects of the interface.
- Always use the user's terms for describing interfaces.

Specification strategy

- Initially, identify one boundary class instance (i.e., object) for each actor/use case pair.
  - ☞ For human actors, it represents the *primary UI window* with which the actor interacts with the system.
  - ☞ For external system actors, it represents the *communication interface* to the external system.

ASU: USE-CASE MODEL



Registrar

Maintain Course Information

Student

Register For Courses

Billing System

Select Courses To Teach

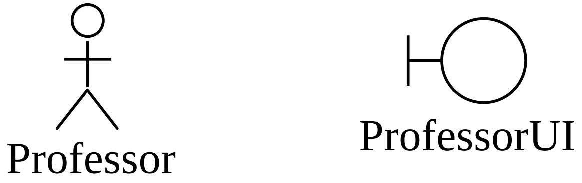
Professor

Request Enrollment List

ASU BOUNDARY CLASSES: PROFESSOR ACTOR

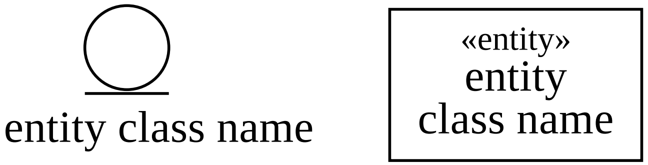
Initial specification

Identify one boundary class (object) for each actor/use case pair.
Try to reuse the same boundary object for an actor, if possible.



5.3.2 5.4.1

ANALYSIS CLASS: ENTITY CLASS



An *entity class* models information that is long-lived and often persistent.

- Represents concepts such as an individual, a real-life object or a real-life event.

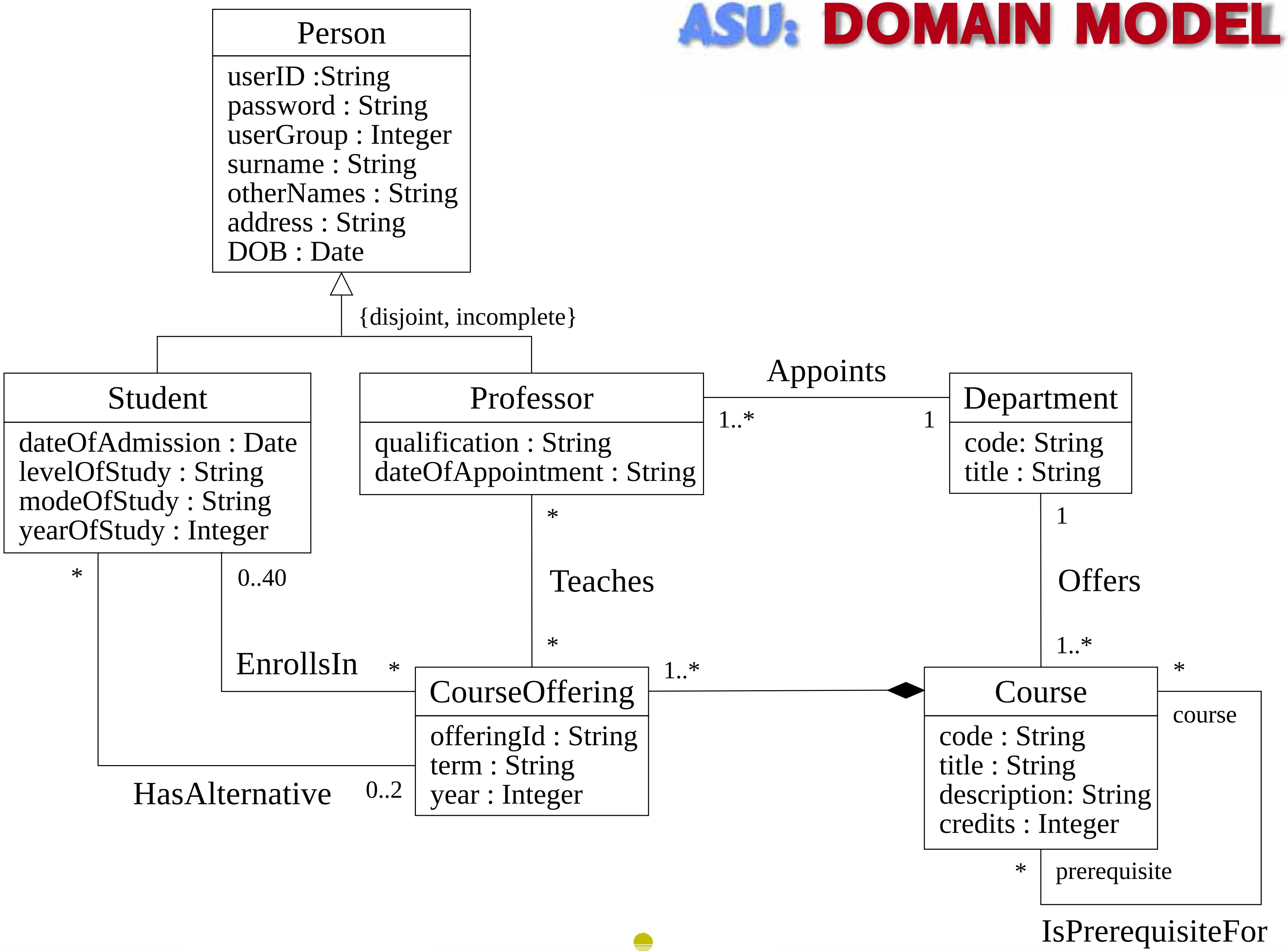
☞ **An entity object will likely get its data from a database**
(but, it actually represents a dynamic object in the analysis model).

Encapsulates and isolates changes to the information it represents.

Specification strategy

- Examine the scenarios of a use-case to determine what domain model classes are needed to carry out the use case.

ASU: DOMAIN MODEL



Person
userID :String
password : String
userGroup : Integer
surname : String
otherNames : String
address : String
DOB : Date

Student
dateOfAdmission : Date
levelOfStudy : String
modeOfStudy : String
yearOfStudy : Integer

Professor
qualification : String
dateOfAppointment : String

Department
code: String
title : String

CourseOffering
offeringId : String
term : String
year : Integer

Course
code : String
title : String
description: String
credits : Integer

ASU USE CASE: SELECT COURSES TO TEACH

Flow of events — *Create Schedule* scenario

1. The use case begins when the Professor actor chooses to select the courses he wants to teach.

In which class(es) are these attributes?

2. The system displays the interface for selecting the courses to teach.
3. The Professor indicates the term and year in which he would like to teach courses.

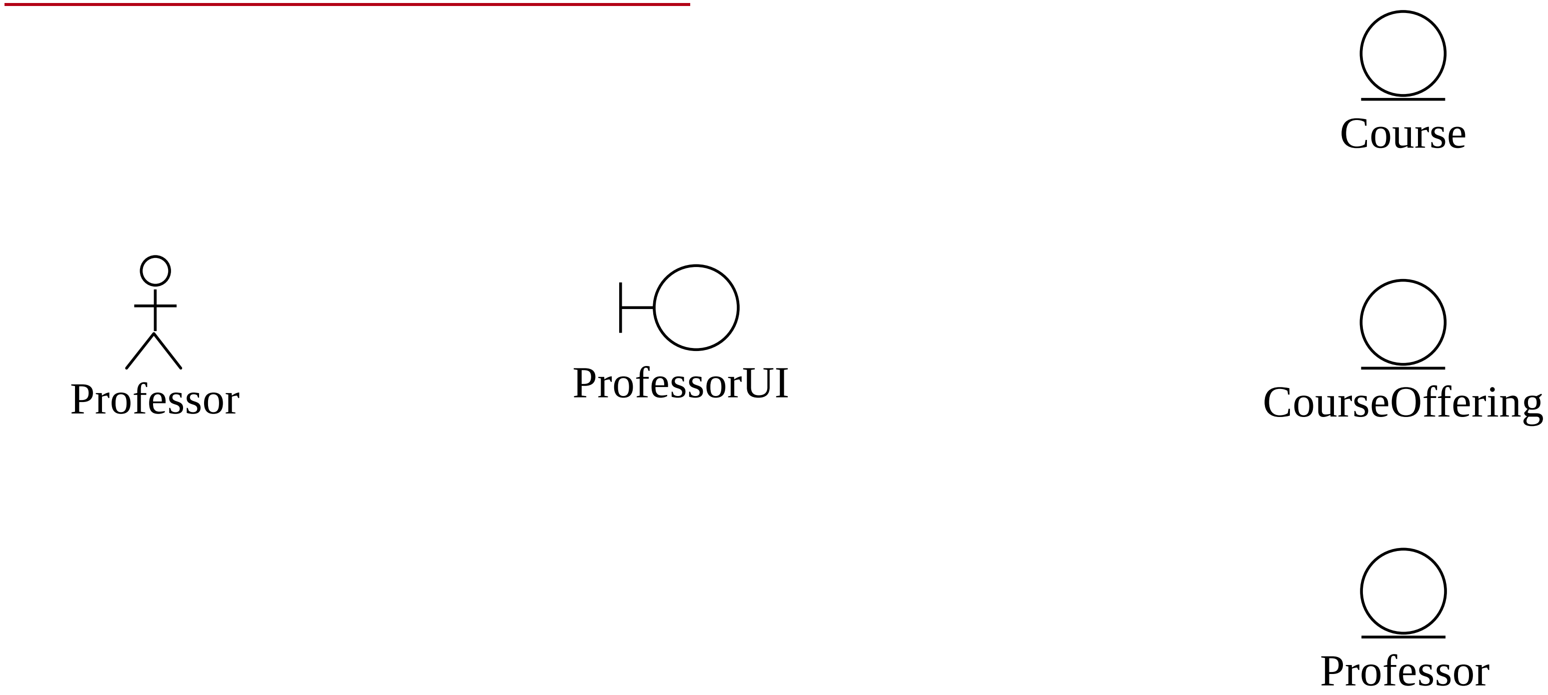
Which class(es) contain this information?

4. The system retrieves and displays the available course information for the given term if the final date for changes has not passed.
5. The Professor selects the courses that he would like to teach.
6. The Professor confirms the selection.
7. The system creates the Professor's teaching schedule in the database.
8. The system notifies the Professor that the teaching schedule has been created.
9. The use case ends.

Which class(es) are needed to represent this?

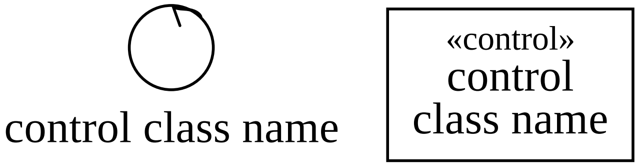
ASU ENTITY CLASSES: SELECT COURSES TO TEACH

Create Schedule scenario



5.3.2 5.4.3

ANALYSIS CLASS: CONTROL CLASS



A *control class* models coordination, sequencing, transactions and control behaviour (for one or more use cases).

- A control class instance (object) usually does not have a correspondence in the application domain.

  ☞ A control object provides the "glue" that ties other classes together into one use-case realization.

- A control class instance (i.e., object) is used to represent:
  - control related to a specific use case.
  - business logic (e.g., complex derivations and calculations).

Encapsulates and isolates changes to control/business logic.

ANALYSIS CLASS: CONTROL CLASS (CONT'D)

Specification strategy

- Initially, assign one control object for each actor/use case pair; this object is responsible for the flow of events in the use case.
  - Complicated behavior may need several control objects.
  - Can also combine/eliminate some control objects.

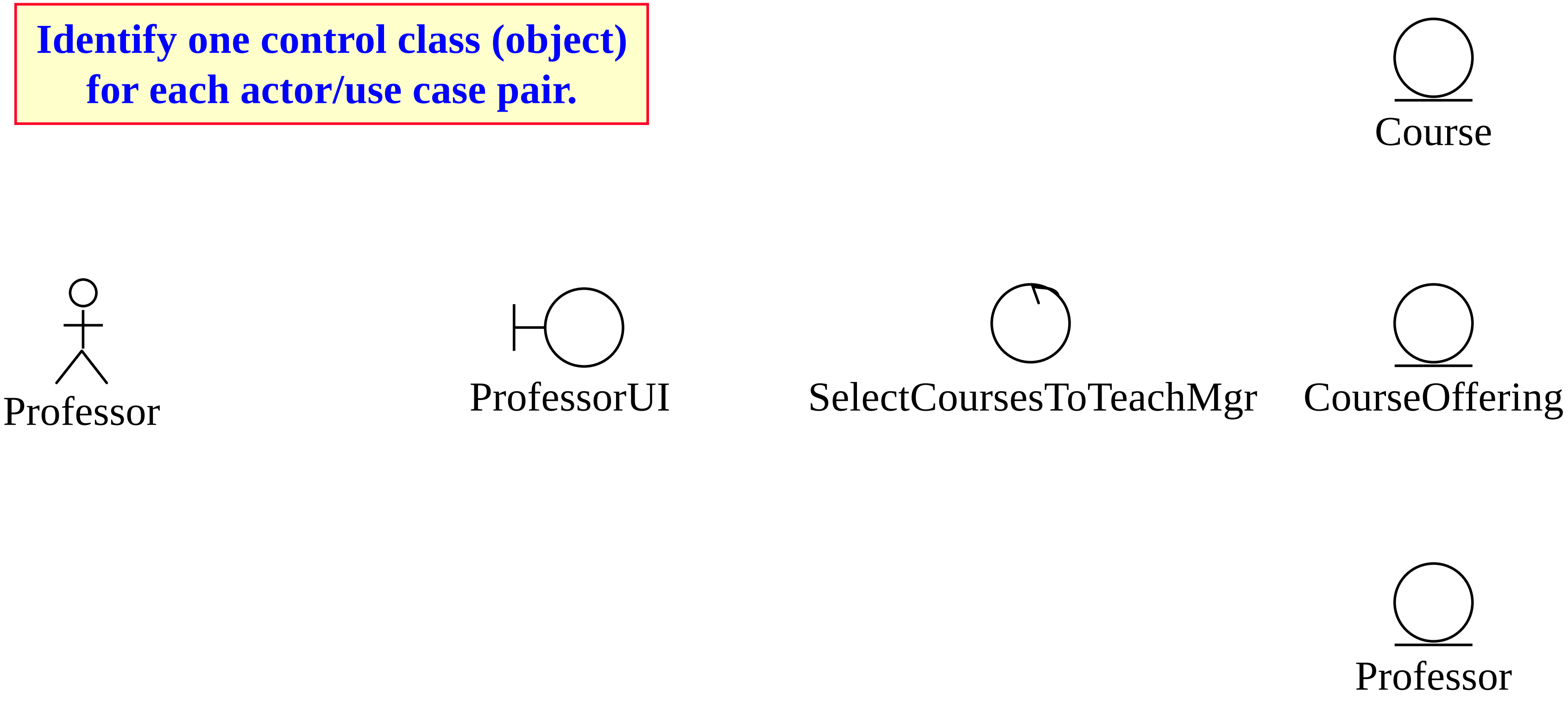
Handle later!

☞ **A control object should only be tied to at most one actor!**

WHY?

ASU CONTROL CLASSES:
SELECT COURSES TO TEACH

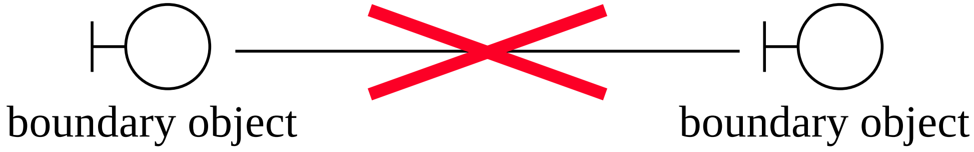
Identify one control class (object) for each actor/use case pair.



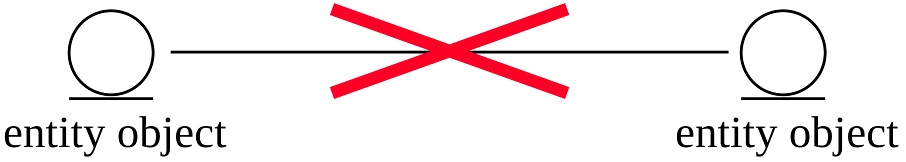
How should we "connect" the boundary, control and entity objects (i.e., which objects should be allowed to interact with each other)?

ANALYSIS OBJECT INTERACTION: BEST PRACTICES

- Actors can interact *only* with boundary objects.
- Boundary objects can interact *only* with control objects and actors (*initially*).

☞ 

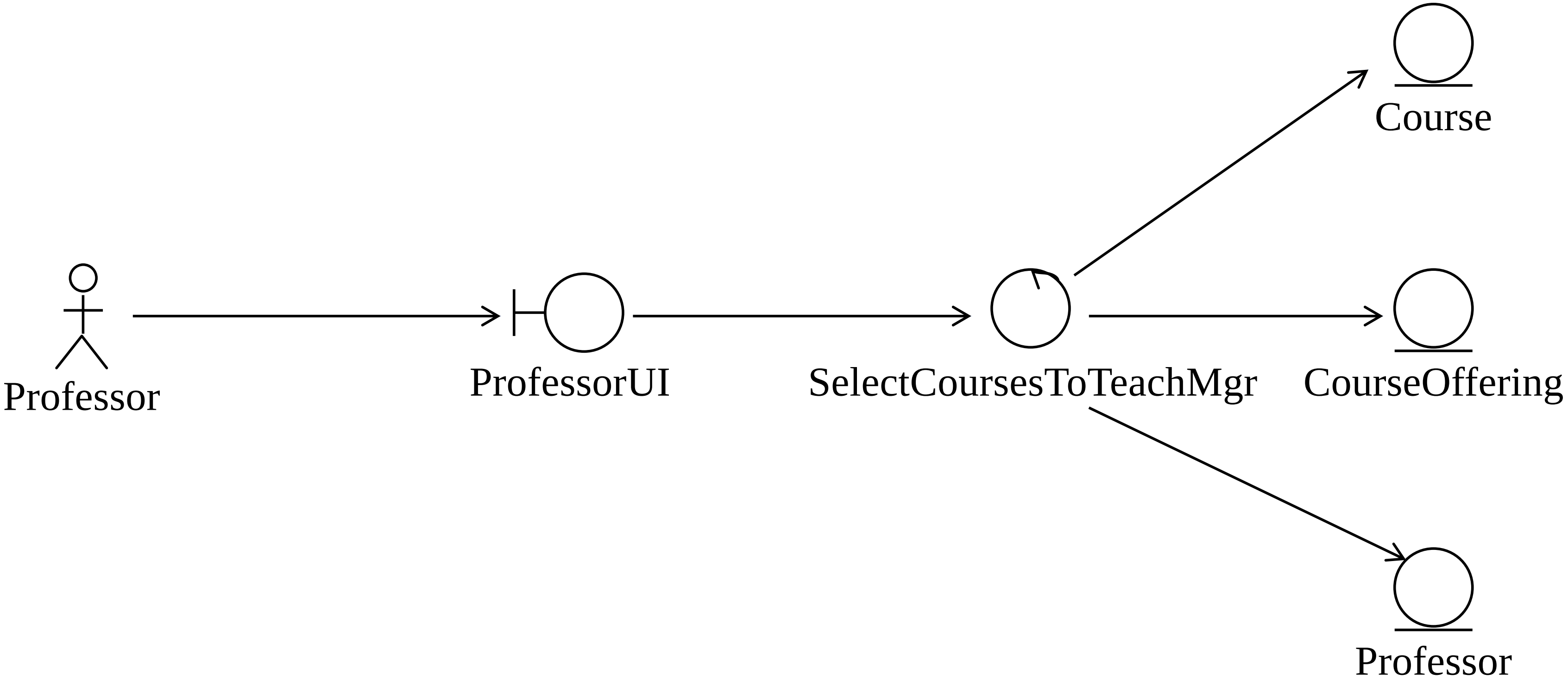
- Entity objects can interact *only* with control objects (*initially*).

☞ 

- Control objects can interact with boundary, entity and *other control objects*.

These best practices result in a well structured and maintainable system.

ASU CONTROL CLASSES:
SELECT COURSES TO TEACH



How should we “connect” the boundary, control and entity objects (i.e., which objects should be allowed to interact with each other)?

USE CASE PARTITIONING

Each use case is entirely divided up into a collaboration of:

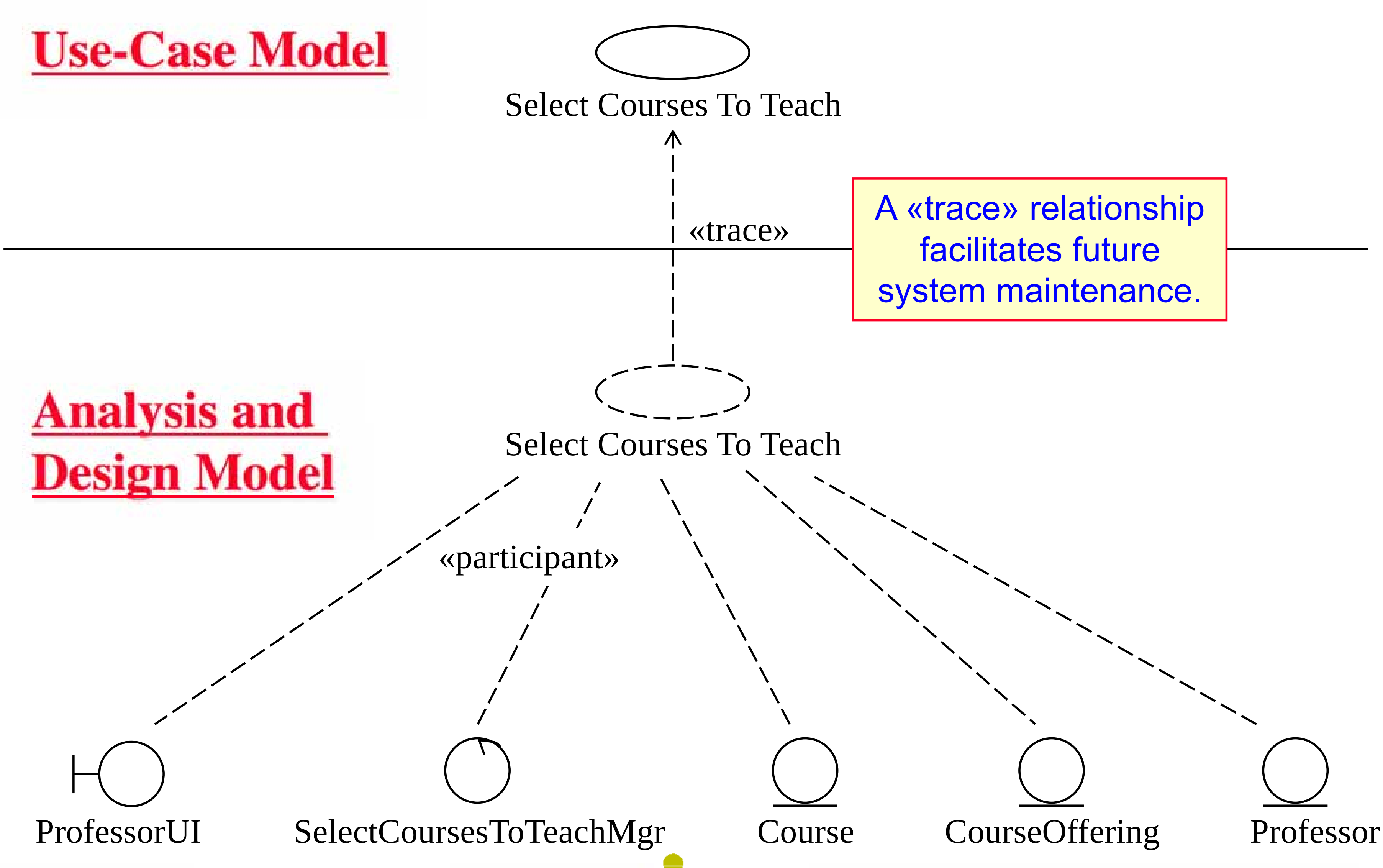
- **boundary classes** → functionalities directly dependent on the system's environment.
- **entity classes** → functionalities dealing with storage and handling of information.
- **control classes** → functionalities specific to one or a few use cases and not naturally placed in boundary or entity classes.

Goal: To achieve localization of changes so as to result in a stable system.

In practice we need to make many judgments about where to place the functionality.

(Design patterns can help guide the decisions.)

ASU: USE CASE PARTITIONING EXAMPLE



A «trace» relationship facilitates future system maintenance.

CLASS DESIGN

A ***design class*** is a class whose specifications has been completed to such a degree that it **can be implemented.**

We need to implement problem domain (analysis) classes using solution domain (implementation technology) classes.

Boundary class → determined by the use of specific UI technologies,

Entity class → determined by the use of data management technologies,

Control class → determined by issues related to:

- distribution → Do we need a separate design class at each node?
- performance → Do we merge with boundary class?
- transactions → Do we need transaction management?

CLASS DESIGN: ACTIVITIES

- **Select reusable components**
  - class libraries and design patterns.
- **Complete the specification**
  - **Add:** attributes; associations; operations; types; visibility; constraints.

☞ **Identify: active classes**.

- **Restructure the design model**
  - add associations; increase reuse with inheritance.
- **Optimize the design model**
  - revise access paths; collapse classes; cache or delay computations.

Design classes are described using the syntax of the programming language.

CLASS DESIGN: ACTIVE CLASS

- An active class has its own thread of control, is usually a boundary or control class and is *shown with a thicker border* in a class diagram.

- An active class can be identified by considering:

  ☞ The performance, throughput and availability requirements of different actors as they interact with the system,

  e.g., a need for fast (real-time) response might be managed by a dedicated active object for taking input and providing output.

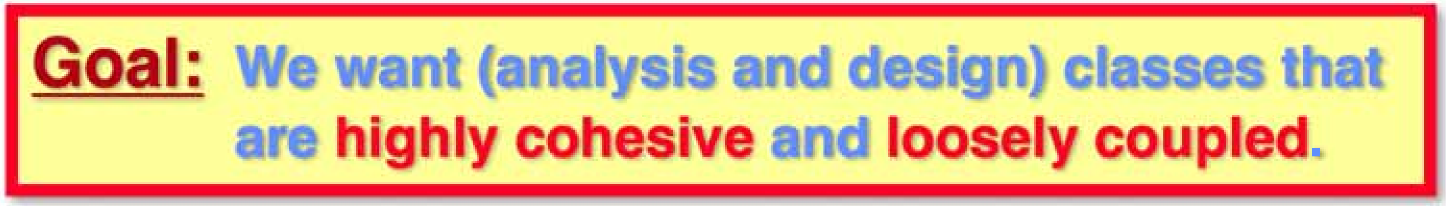
  ☞ The system's distribution onto nodes — active objects are needed to support distribution onto several nodes,

  e.g., one active object per node and separate active objects to handle inter-node communication.

  ☞ Other requirements:

  e.g., system startup and termination, liveness, deadlock avoidance, starvation avoidance, reconfiguration of nodes, connection capacity, etc. → An active object is needed to initiate/monitor activity.

COHESION AND COUPLING



Goal: We want (analysis and design) classes that are highly cohesive and loosely coupled.

cohesion - A measure of the number of functionally different things a class has to do.

☞ **A class is most cohesive when it does only one thing.**

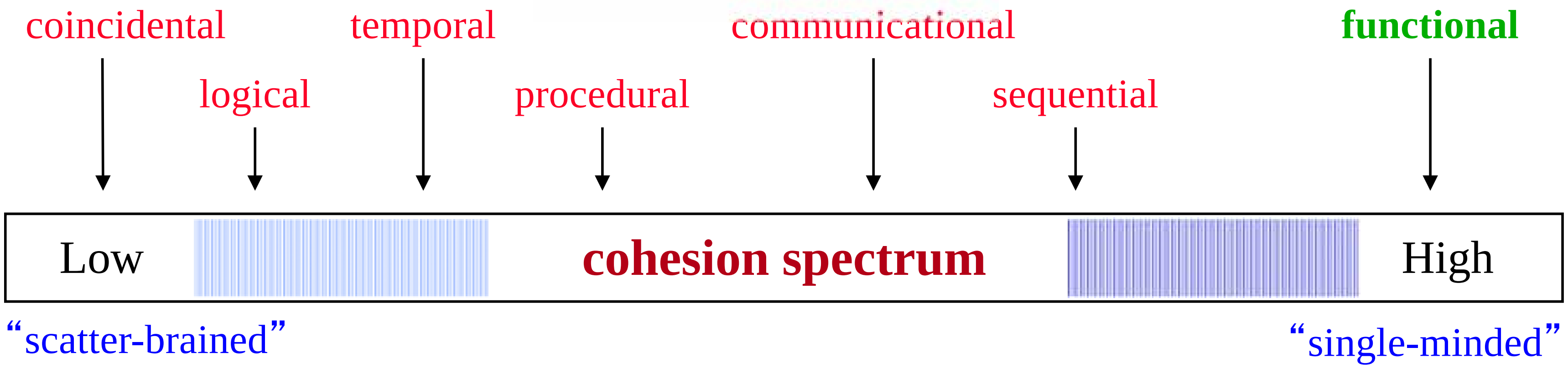
coupling - A measure of the number and types of dependencies a class has with other classes.

☞ **A class has the lowest coupling when it has minimal dependencies with other classes.**

There is a trade-off between cohesion and coupling.

(We should follow the 7 ± 2 heuristic for coupling.)

COHESION



WORST **coincidental** - The class does not achieve any definable function.

logical - The class does several similar, but slightly different things.

temporal - The class contains various operations that happen to be executed at the same time.

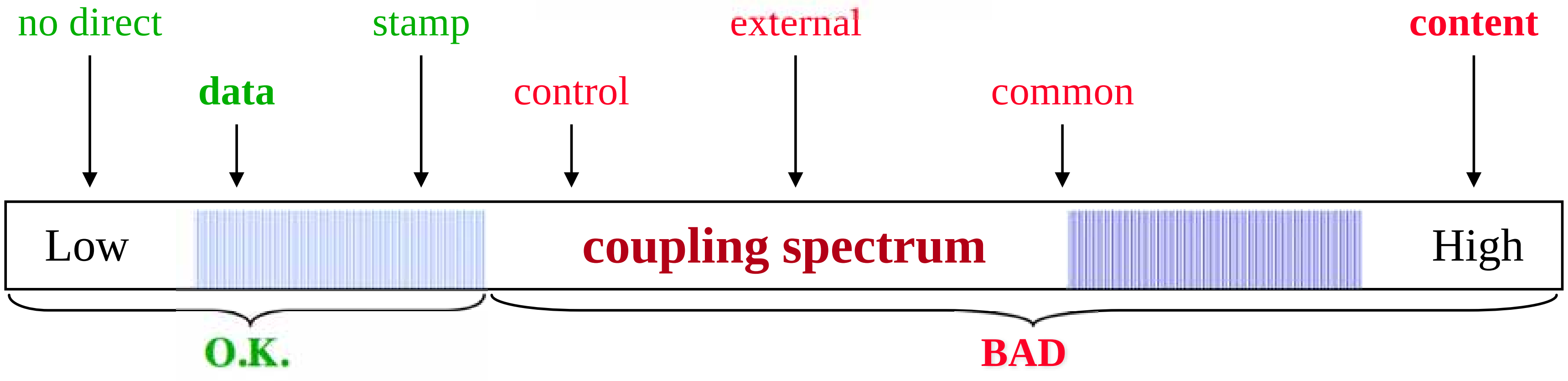
procedural - The class operations must be executed in a pre-specified order.

communicational - All class operations operate on the same data stream or data structure.

sequential - The output from one class is input to the another class.

BEST **functional** - The class has one and only one identifiable function.

COUPLING



no direct - The classes are not related.

data - The classes take as input only simple data (e.g., a parameter list). ***BEST!***

stamp - The classes pass a portion of a data structure as an argument.

control - The classes pass control information (e.g., via a flag or switch).

external -The classes are tied to an environment external to the system.

common - Several classes reference a global common data area.

content - One class makes use of data or control information maintained within another class. **AVOID!**

SOLID DESIGN PRINCIPLES

S – The Single Responsibility Principle

Each class should have a single responsibility.

O – The Open Closed Principle

A class should be open for extension but closed for modification.

L – The Liskov Substitution Principle

A super class must be able to use objects of subclasses.

I – Interface Segregation Principle

Do not force clients to depend upon interfaces that they do not use.

D – Dependency Inversion Principle

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.