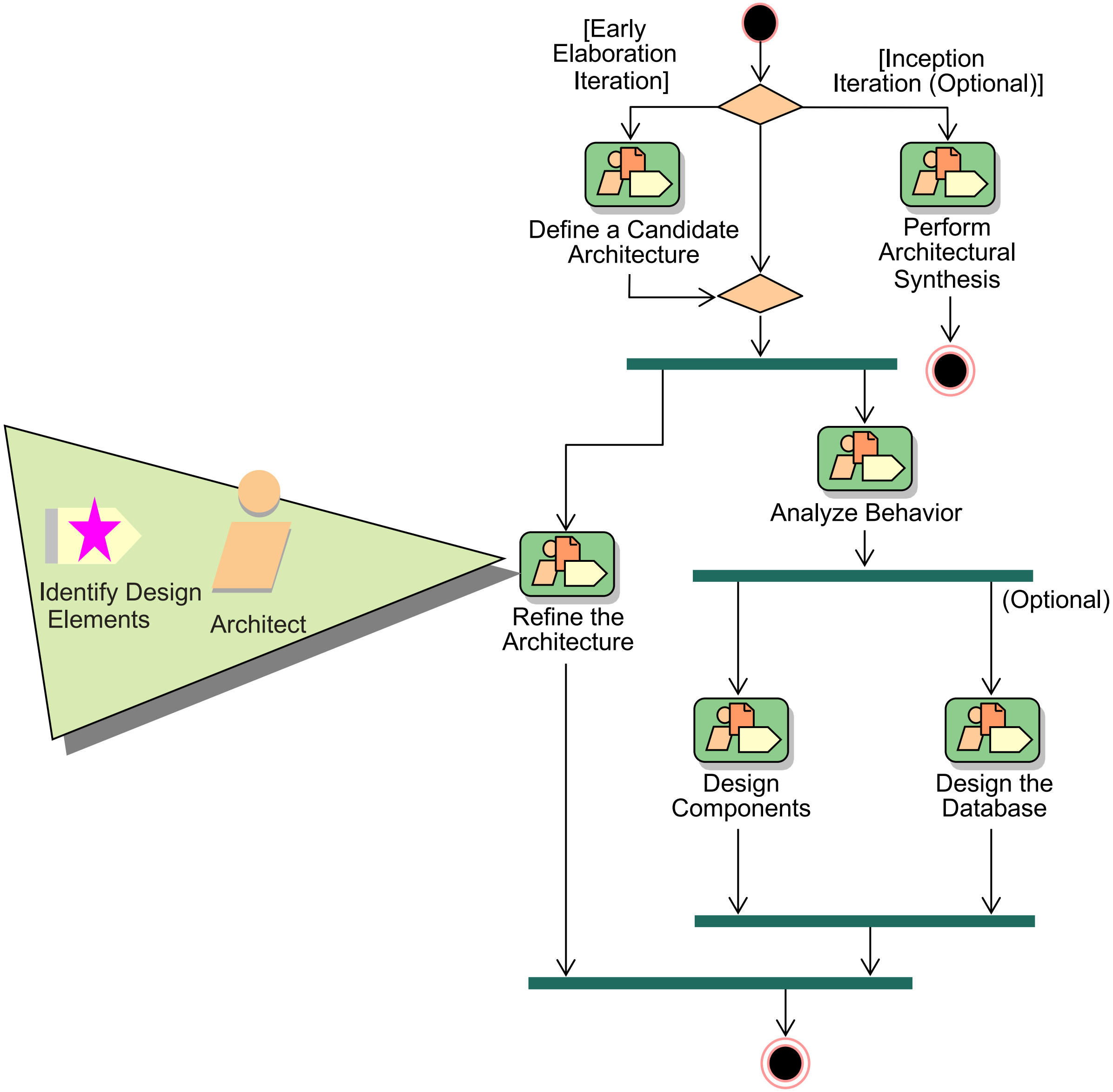
Software analysis and design

Module 11: Identify Design Elements

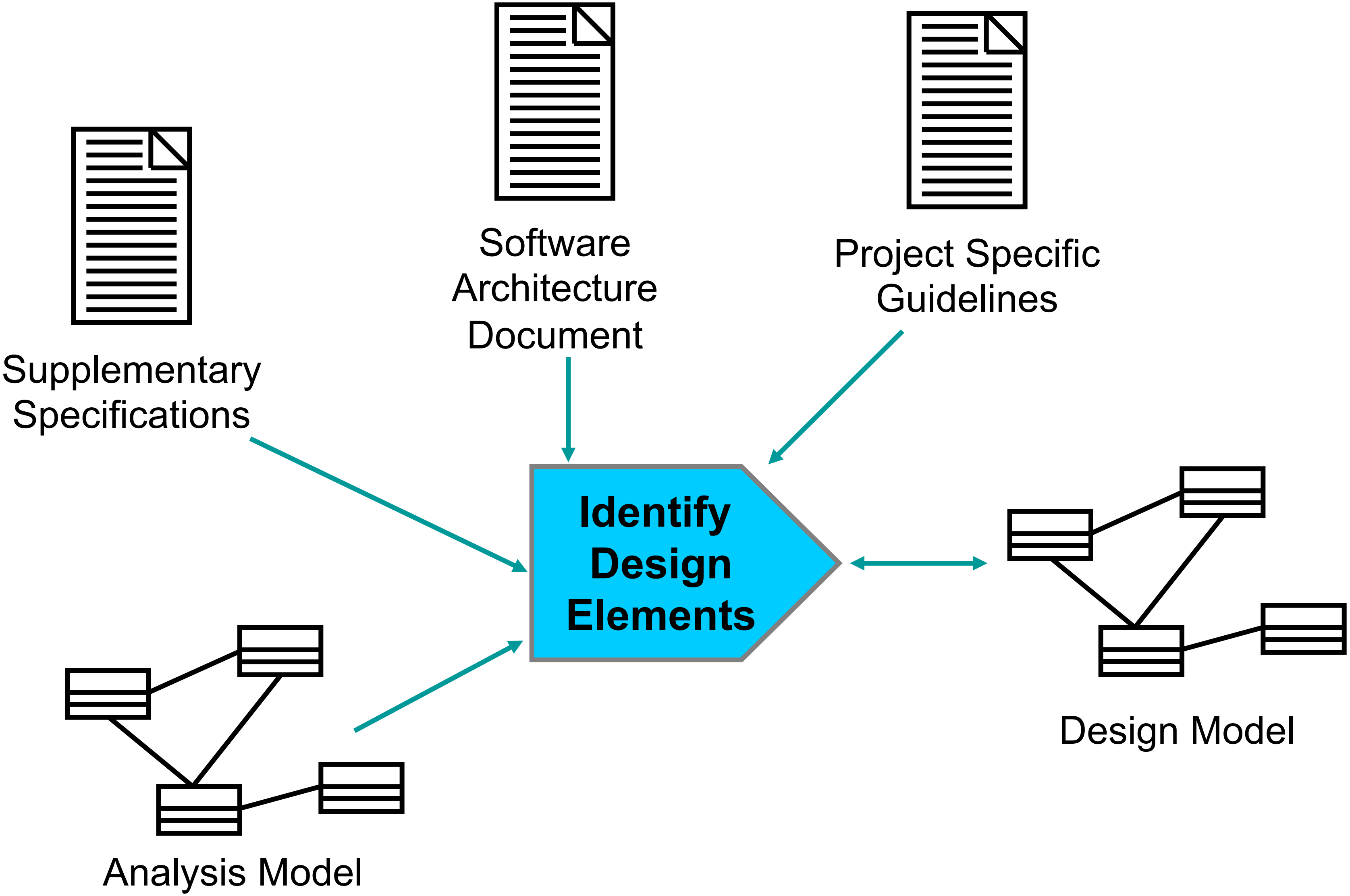
Objectives: Identify Design Elements

- Define the purpose of Identify Design Elements and demonstrate where in the lifecycle it is performed
- Analyze interactions of analysis classes and identify Design Model elements
  - Design classes
  - Subsystems
  - Subsystem interfaces

Identify Design Elements in Context


[Early Elaboration Iteration]
[Inception Iteration (Optional)]
Define a Candidate Architecture
Perform Architectural Synthesis
Analyze Behavior
Identify Design Elements
Architect
Refine the Architecture
(Optional)
Design Components
Design the Database

Identify Design Elements Overview



Supplementary Specifications

Software Architecture Document

Project Specific Guidelines

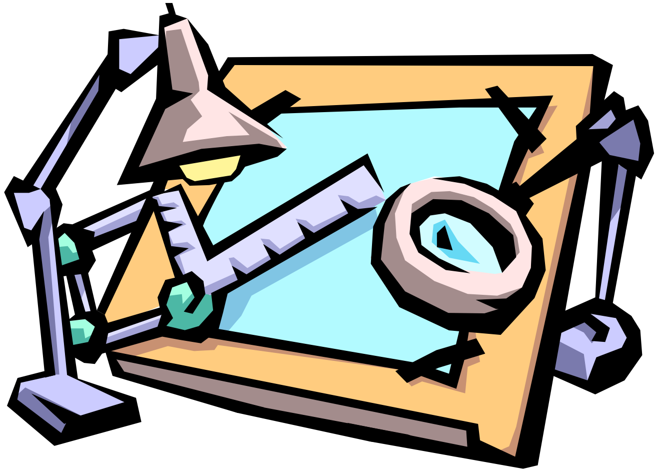
Identify Design Elements

Analysis Model

Design Model

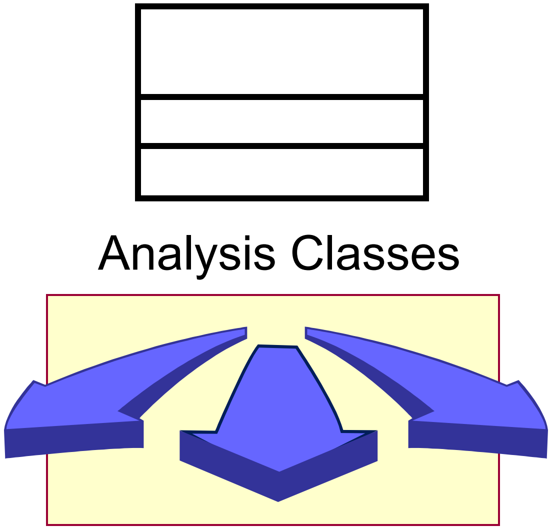
Identify Design Elements Steps

- Identify classes and subsystems
- Identify subsystem interfaces
- Update the organization of the Design Model
- Checkpoints

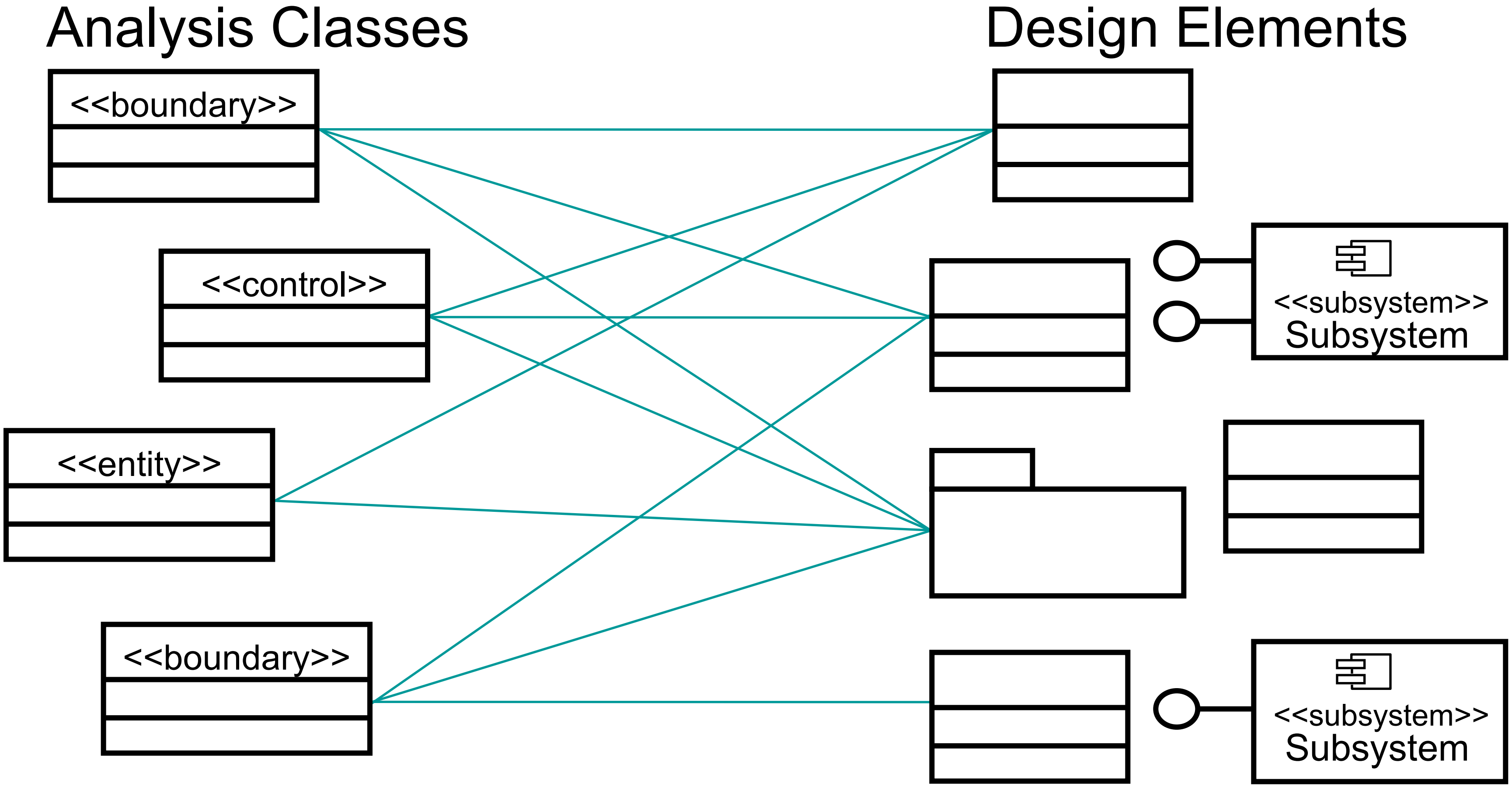


Identify Design Elements Steps

- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints



From Analysis Classes to Design Elements



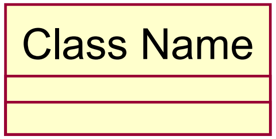
Many-to-Many Mapping

Identifying Design Classes

- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
- More complex analysis classes may
  - Split into multiple classes
  - Become a package
  - Become a subsystem (discussed later)
  - Any combination …

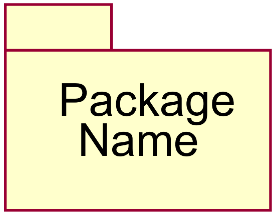
Review: Class and Package

- What is a class?
  - A description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics



- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain

  other model elements



Packaging Tips: Boundary Classes

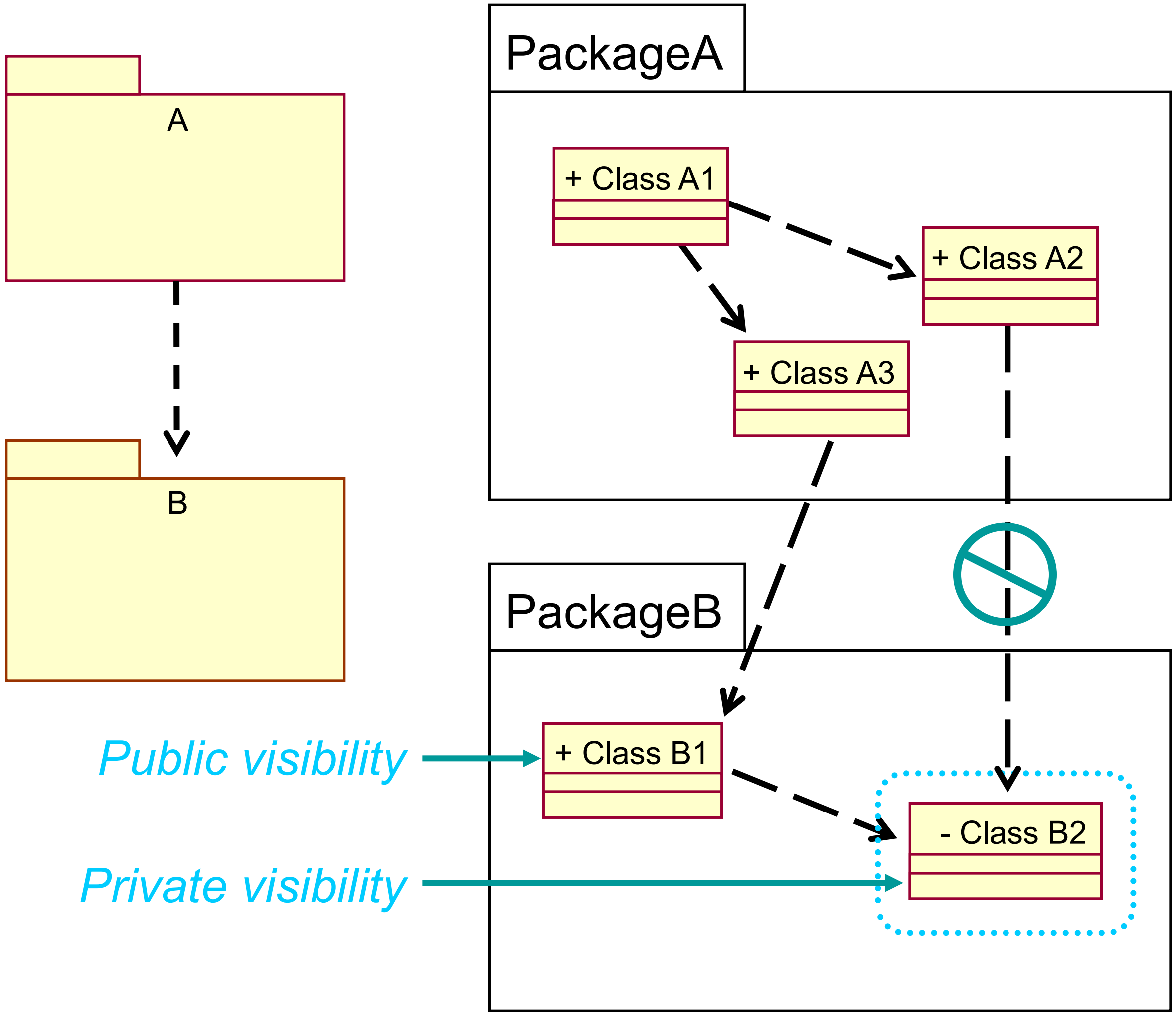
If it is **likely** the system interface will undergo considerable changes

Boundary classes placed in separate packages

If it is **unlikely** the system interface will undergo considerable changes

Boundary classes packaged with functionally related classes

Package Dependencies: Package Element Visibility

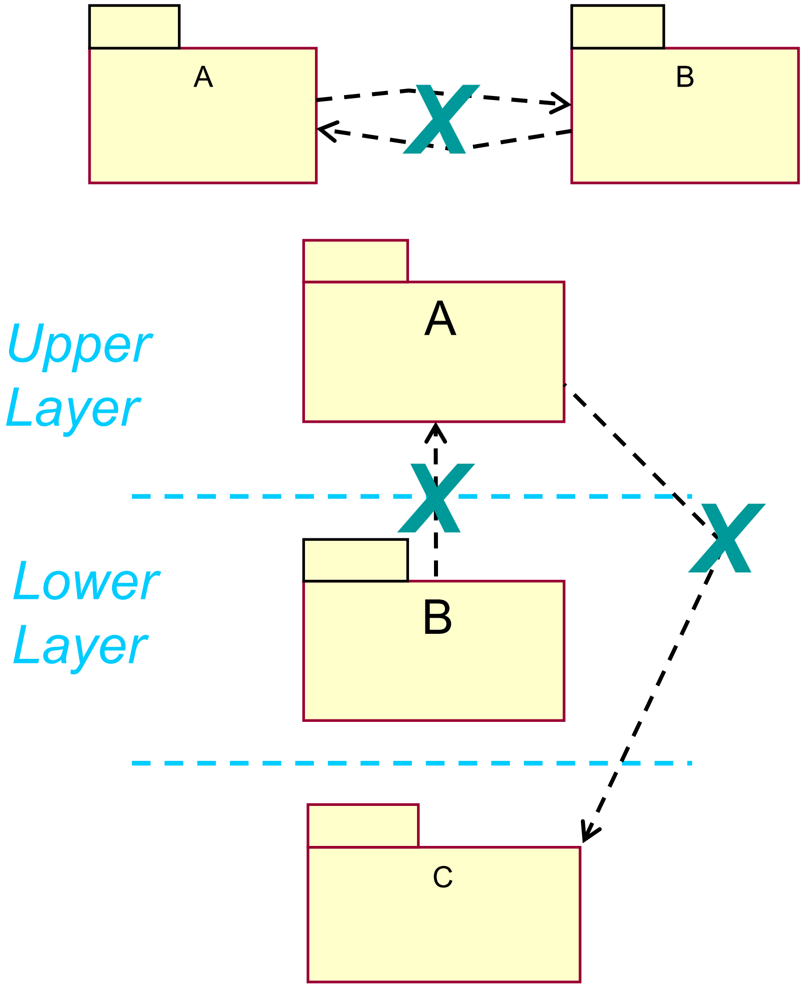


Only public classes can be referenced outside of the owning package

OO Principle: Encapsulation

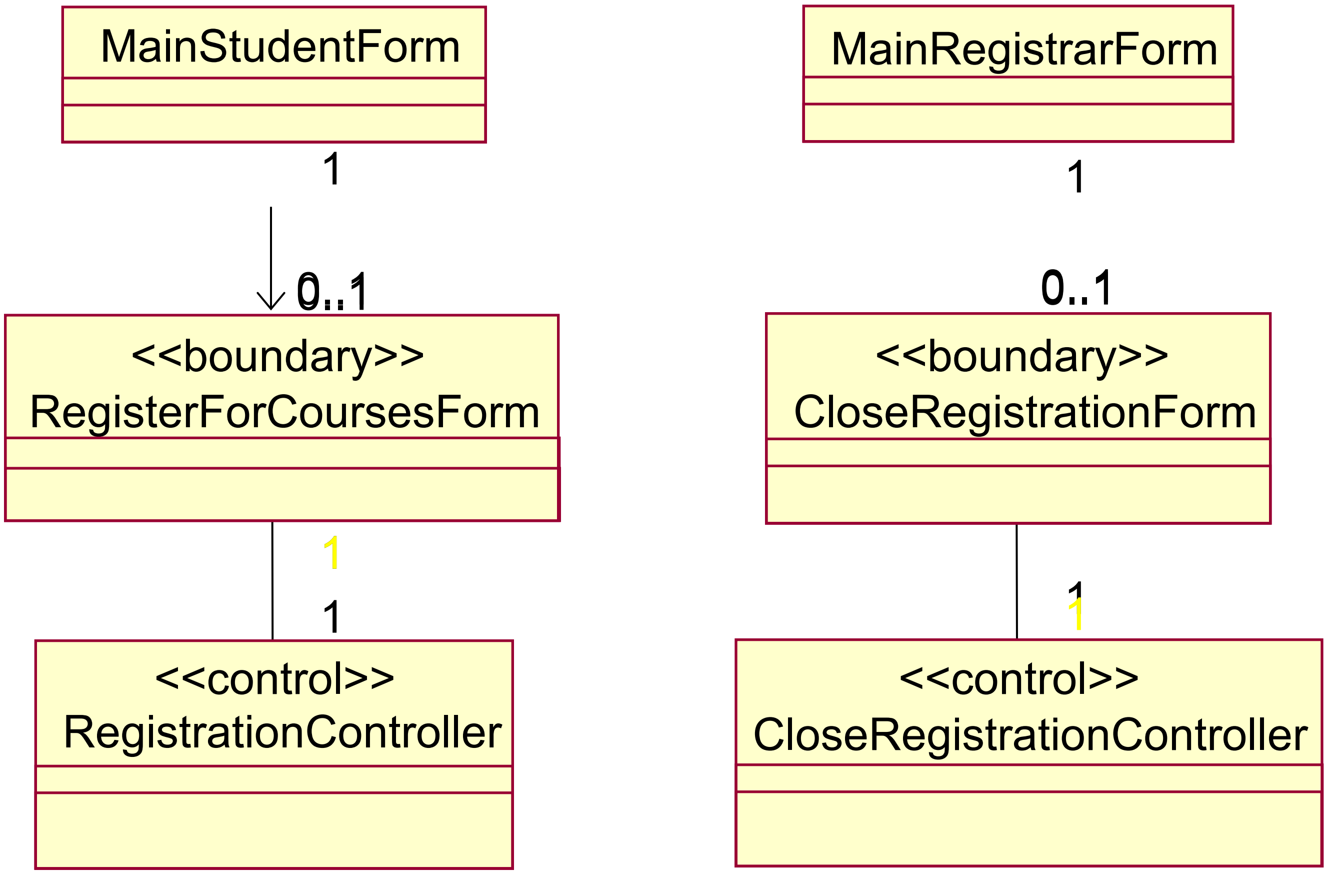
Package Coupling: Tips

- Packages should not be cross-coupled
- Packages in lower layers should not be dependent upon packages in upper layers
- In general, dependencies should not skip layers

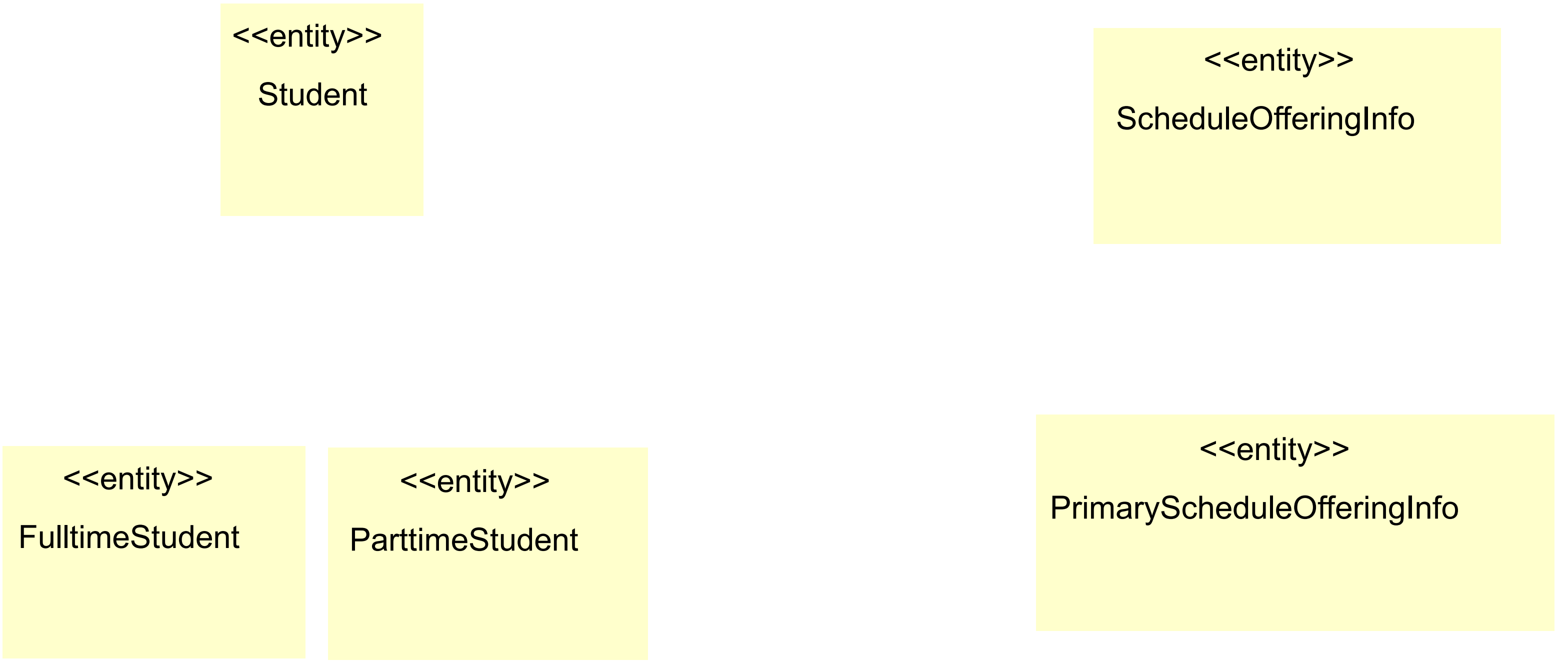


X = Coupling violation

Example: Registration Package



Example: University Artifacts Package: Generalization



<<entity>>
Student

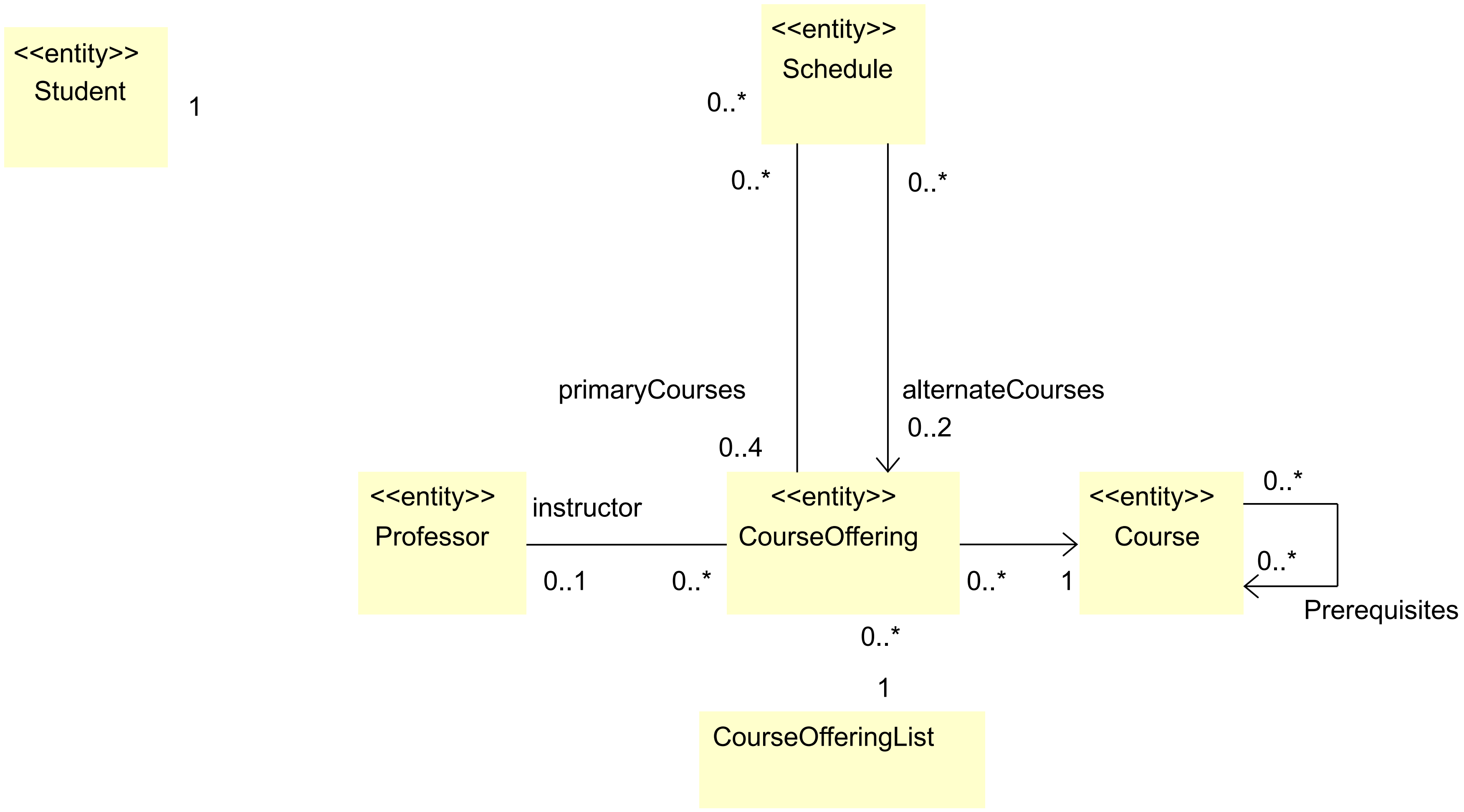
<<entity>>
ScheduleOfferingInfo

<<entity>>
FulltimeStudent

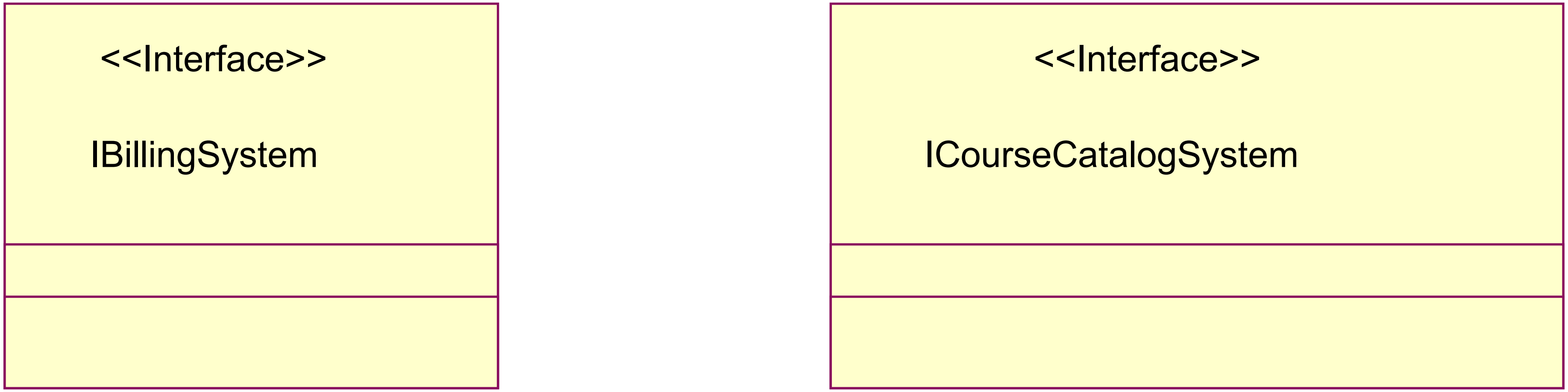
<<entity>>
ParttimeStudent

<<entity>>
PrimaryScheduleOfferingInfo

Example: University Artifacts Package: Associations



Example: External System Interfaces Package



<<Interface>>

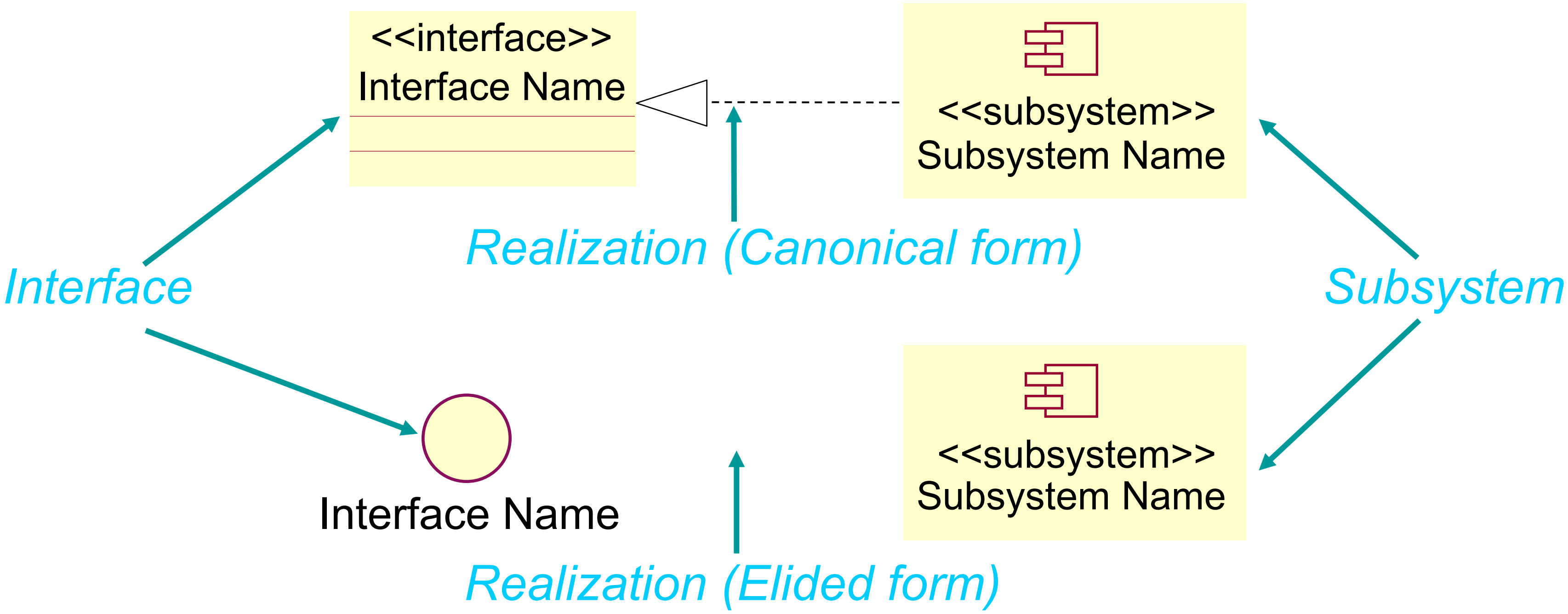
IBillingSystem

<<Interface>>

ICourseCatalogSystem

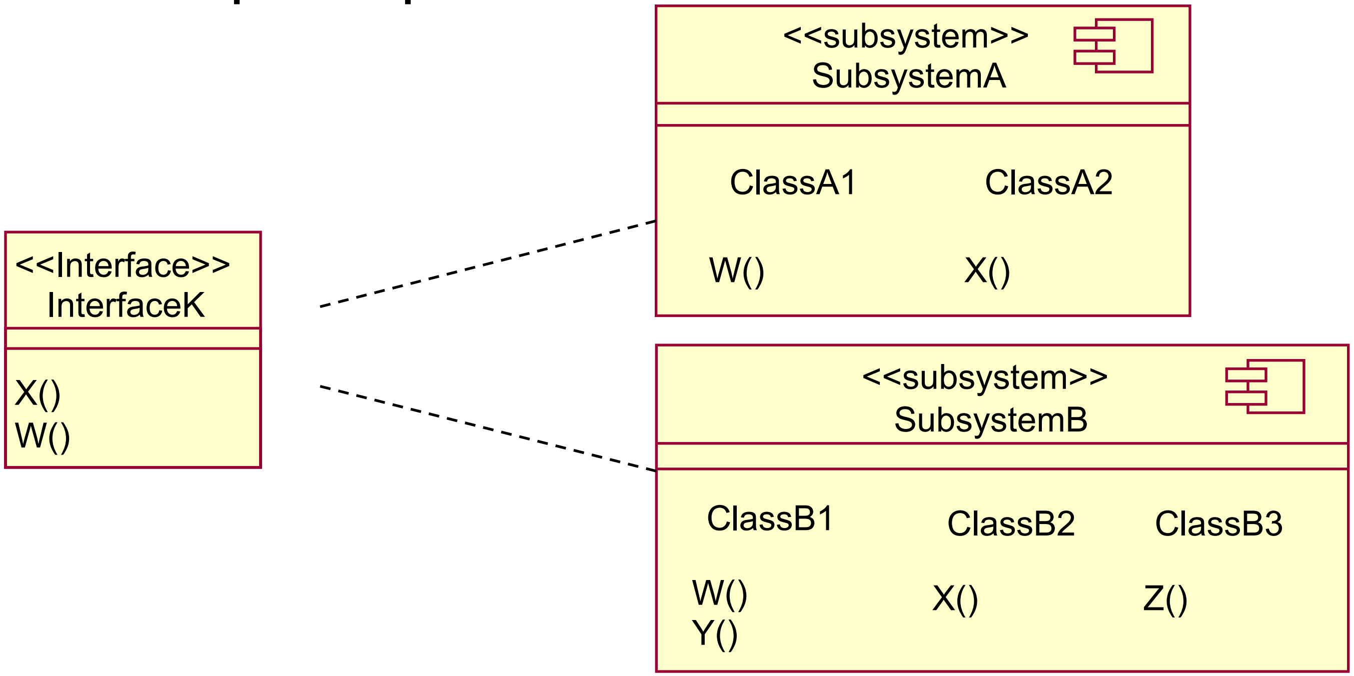
Review: Subsystems and Interfaces

- Realizes one or more interfaces that define its behavior



Subsystems and Interfaces (continued)

- Subsystems :
  - Completely encapsulate behavior
  - Represent an independent capability with clear interfaces (potential for reuse)
  - Model multiple implementation variants



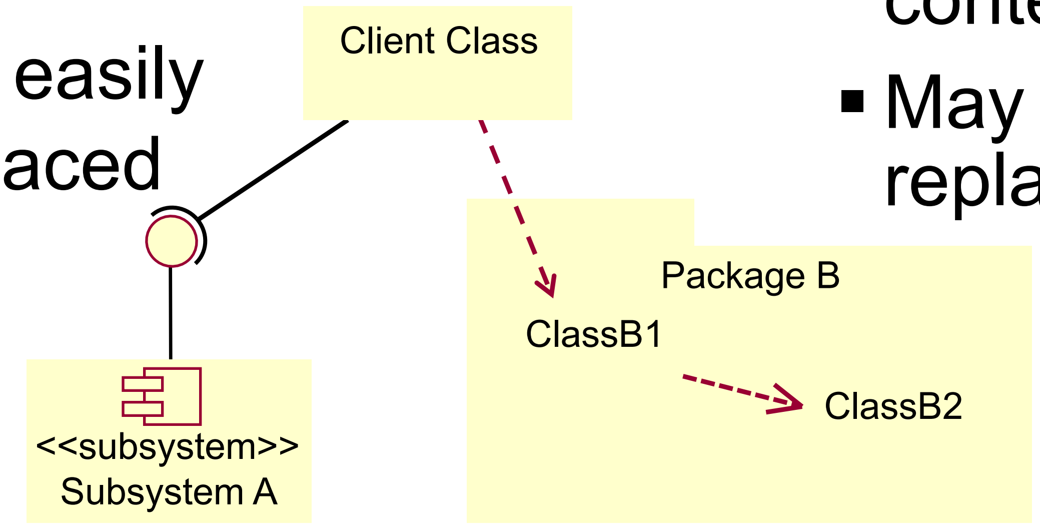
Packages versus Subsystems

Subsystems

- Provide behavior
- Completely encapsulate their contents
- Are easily replaced

Packages

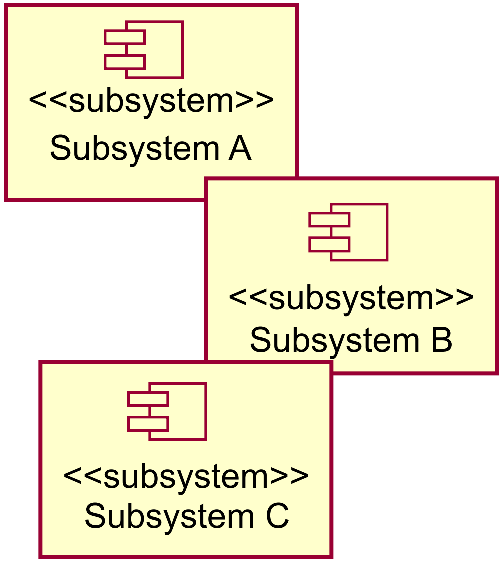
- Don’t provide behavior
- Don’t completely encapsulate their contents
- May not be easily replaced



Encapsulation is the key!

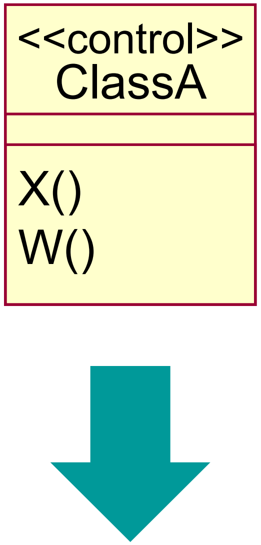
Candidate Subsystems

- Analysis classes which may evolve into subsystems:
  - Classes providing complex services and/or utilities
  - Boundary classes (user interfaces and external system interfaces)
- Existing products or external systems in the design (e.g., components):
  - Communication software
  - Database access support
  - Types and data structures
  - Common utilities
  - Application-specific products



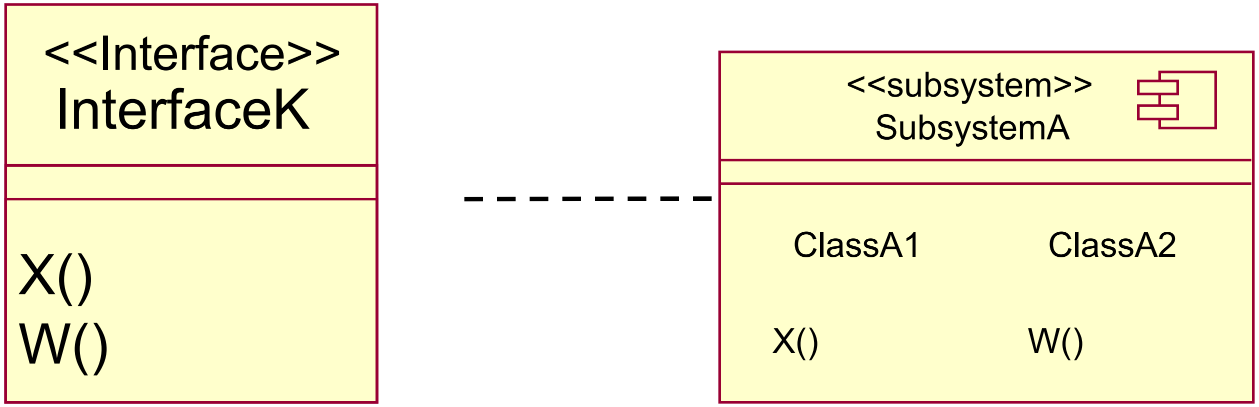
Identifying Subsystems

"Superman Class"



<<control>>
ClassA

X()
W()



<<Interface>>
InterfaceK

X()
W()

<<subsystem>>
SubsystemA

ClassA1

ClassA2

X()

W()

Identify Design Elements Steps

- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints

Identifying Interfaces

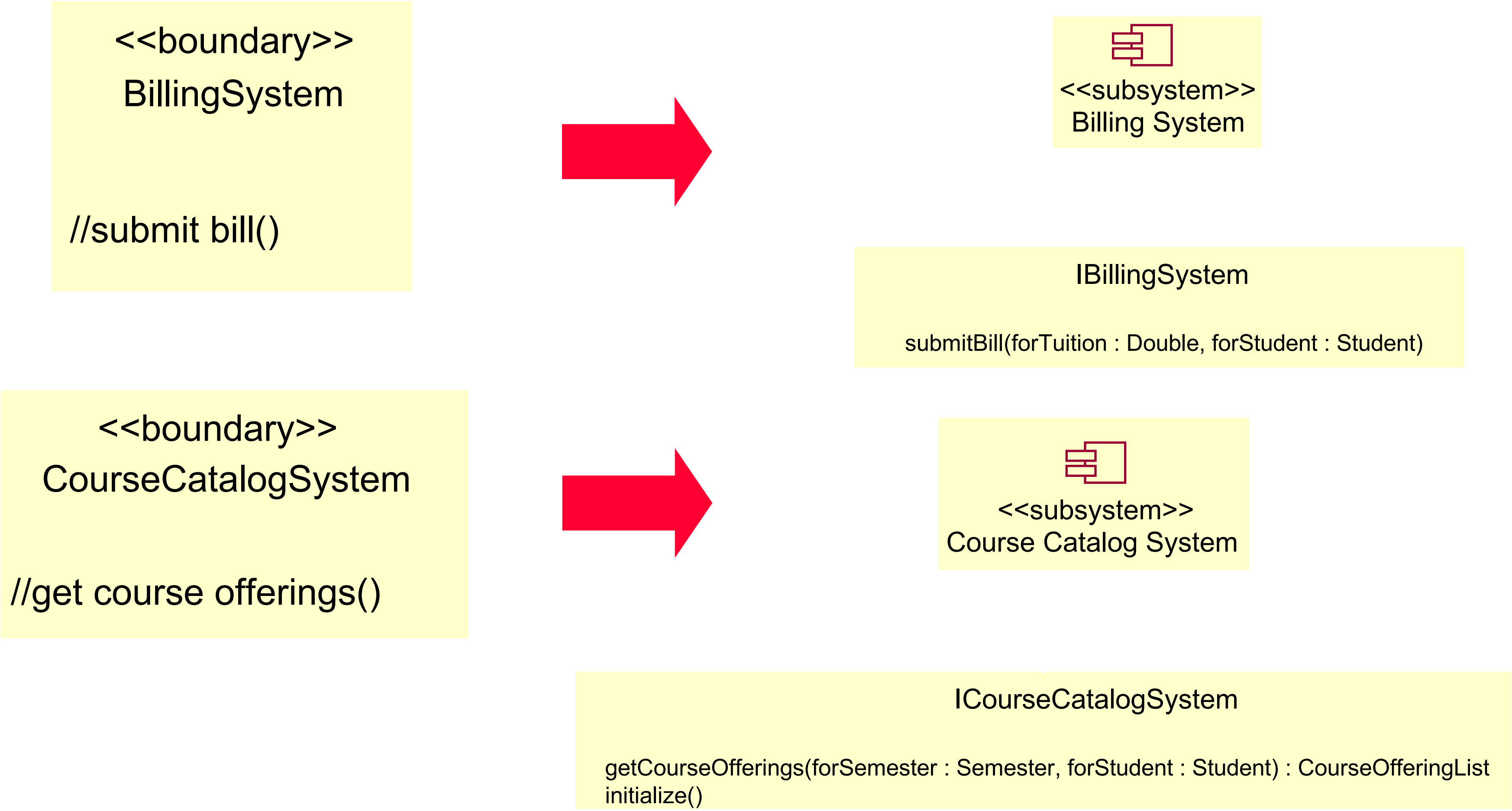
- Purpose
  - To identify the interfaces of the subsystems based on their responsibilities
- Steps
  - Identify a set of candidate interfaces for all subsystems.
  - Look for similarities between interfaces.
  - Define interface dependencies.
  - Map the interfaces to subsystems.
  - Define the behavior specified by the interfaces…
  - Package the interfaces

Stable, well-defined interfaces are key to a stable, resilient architecture.

Interface Guidelines

- Interface name
  - Reflects role in system
- Interface description
  - Conveys responsibilities
- Operation definition
  - Name should reflect operation result
  - Describes what operation does, all parameters and result
- Interface documentation
  - Package supporting info: sequence and state diagrams, test plans, etc.

Example: Design Subsystems and Interfaces



<<boundary>>
BillingSystem

//submit bill()

<<subsystem>>
Billing System

IBillingSystem

submitBill(forTuition : Double, forStudent : Student)

<<boundary>>
CourseCatalogSystem

//get course offerings()

<<subsystem>>
Course Catalog System

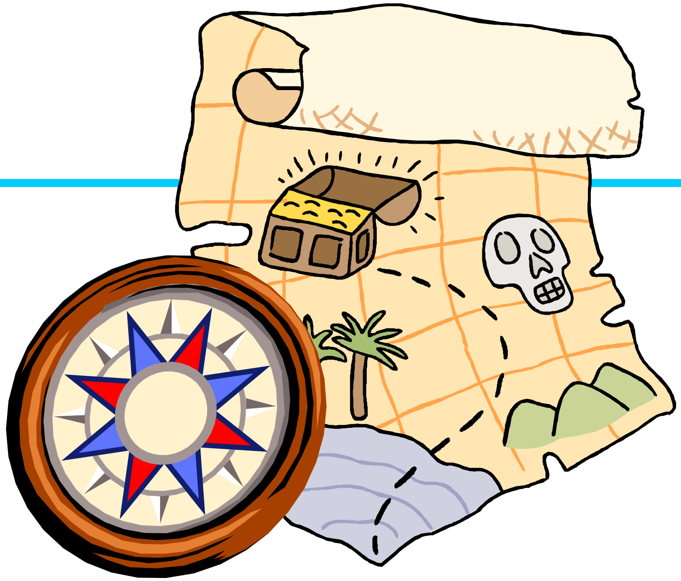
ICourseCatalogSystem

getCourseOfferings(forSemester : Semester, forStudent : Student) : CourseOfferingList
initialize()

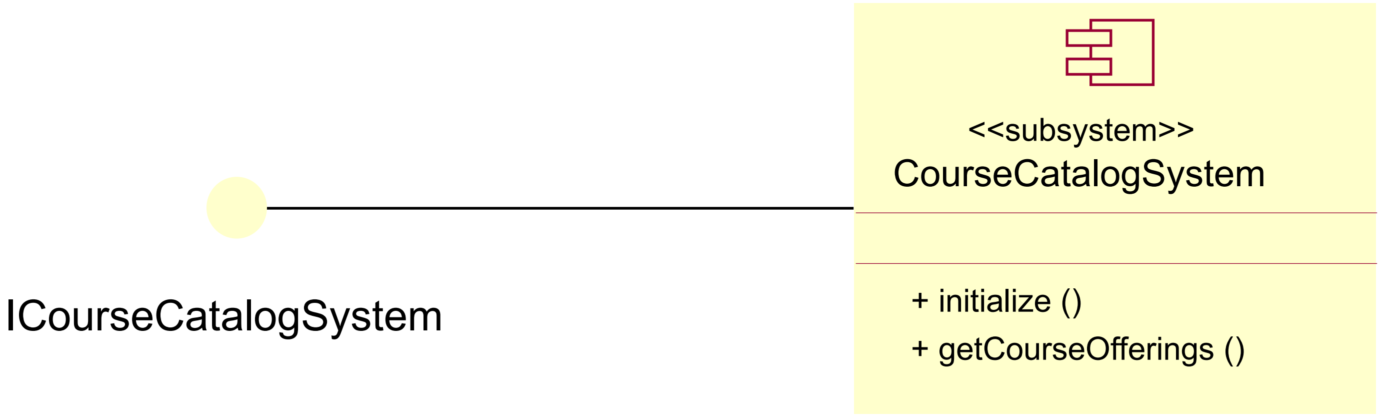
All other analysis classes map directly to design classes.

Example: Analysis-Class-To-Design-Element Map

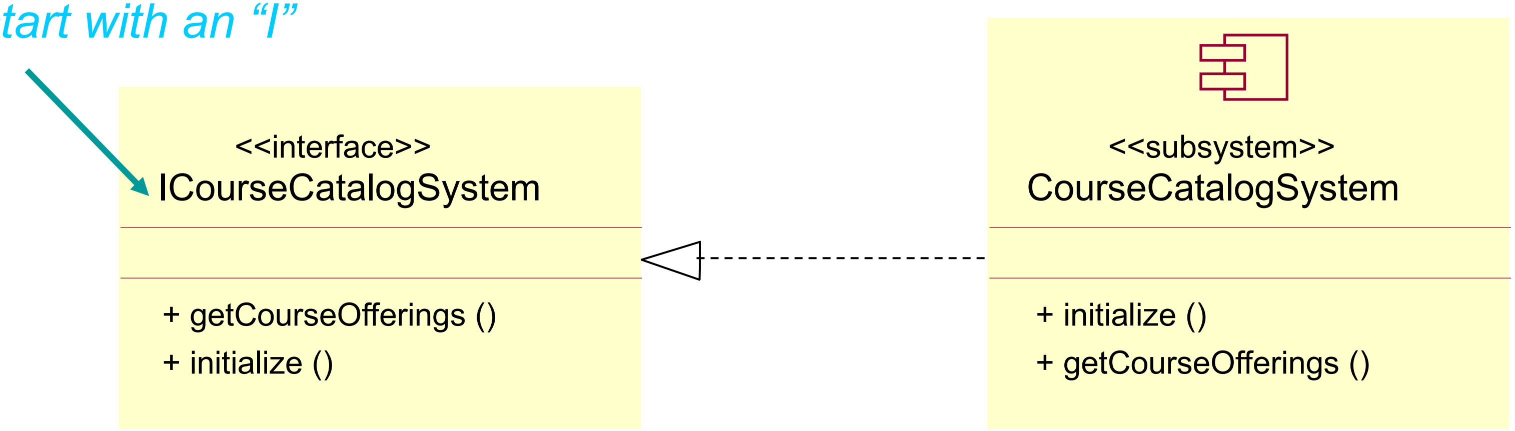
Analysis Class	**Design Element**
CourseCatalogSystem	CourseCatalogSystem Subsystem
BillingSystem	BillingSystem Subsystem
All other analysis classes map directly to design classes	



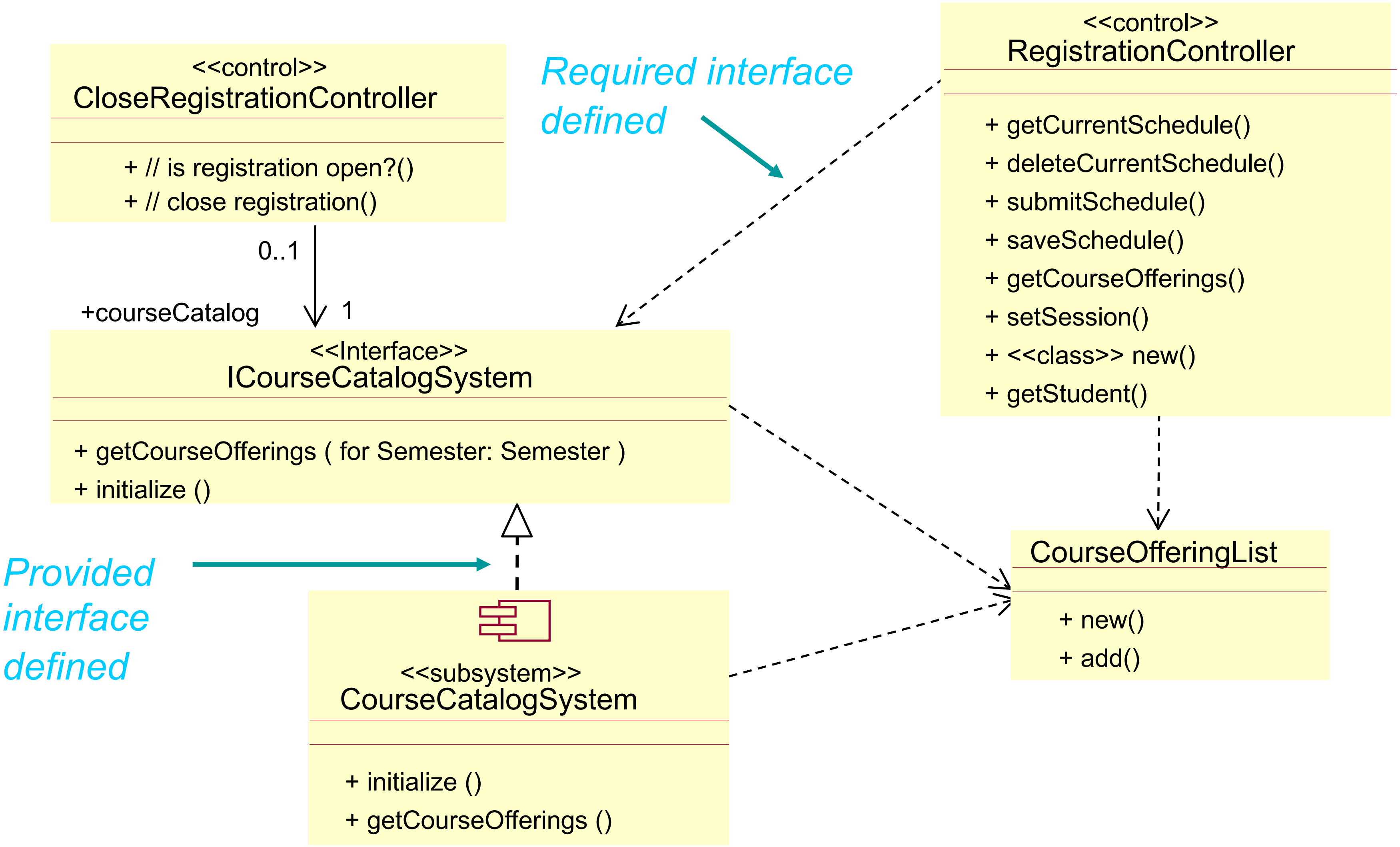
Modeling Convention: Subsystems and Interfaces



Interfaces start with an "I"



Example: Subsystem Context: CourseCatalogSystem



<<control>>
CloseRegistrationController

+ // is registration open?()
+ // close registration()

0..1

+courseCatalog

1

<<Interface>>
ICourseCatalogSystem

+ getCourseOfferings (for Semester: Semester)
+ initialize ()

Required interface defined

<<control>>
RegistrationController

+ getCurrentSchedule()
+ deleteCurrentSchedule()
+ submitSchedule()
+ saveSchedule()
+ getCourseOfferings()
+ setSession()
+ <<class>> new()
+ getStudent()

Provided interface defined

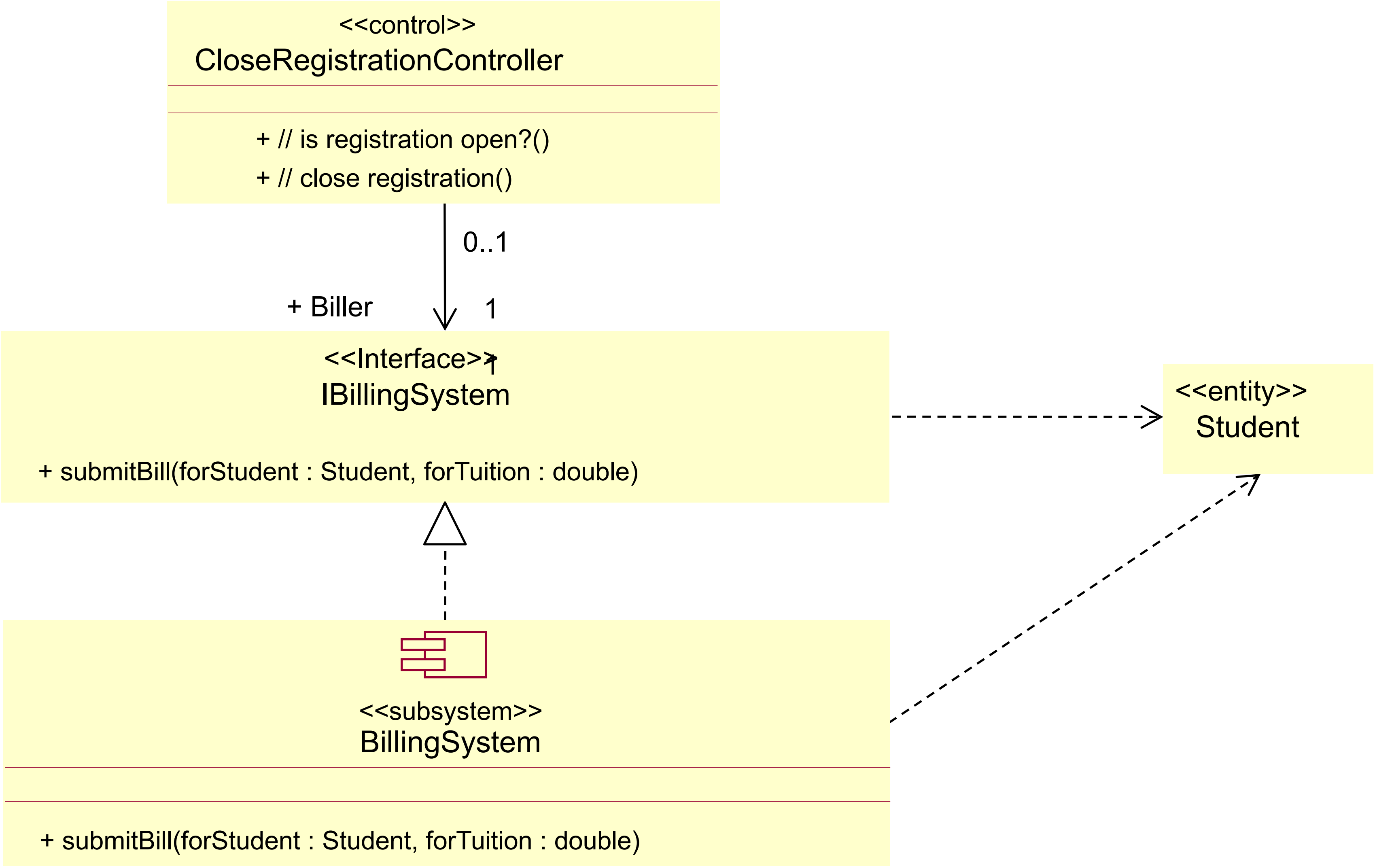
<<subsystem>>
CourseCatalogSystem

+ initialize ()
+ getCourseOfferings ()

CourseOfferingList

+ new()
+ add()

Example: Subsystem Context: Billing System



<<control>>
CloseRegistrationController

+ // is registration open?()
+ // close registration()

0..1

+ Biller

1

<<Interface>>
IBillingSystem

1

+ submitBill(forStudent : Student, forTuition : double)

<<entity>>
Student

<<subsystem>>
BillingSystem

+ submitBill(forStudent : Student, forTuition : double)

Identify Design Elements Steps

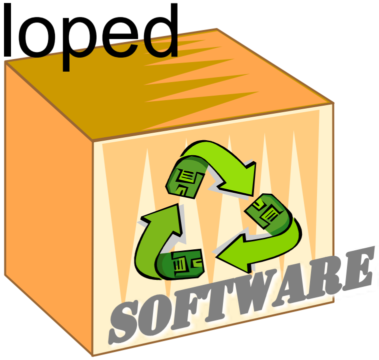
- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints

Identification of Reuse Opportunities

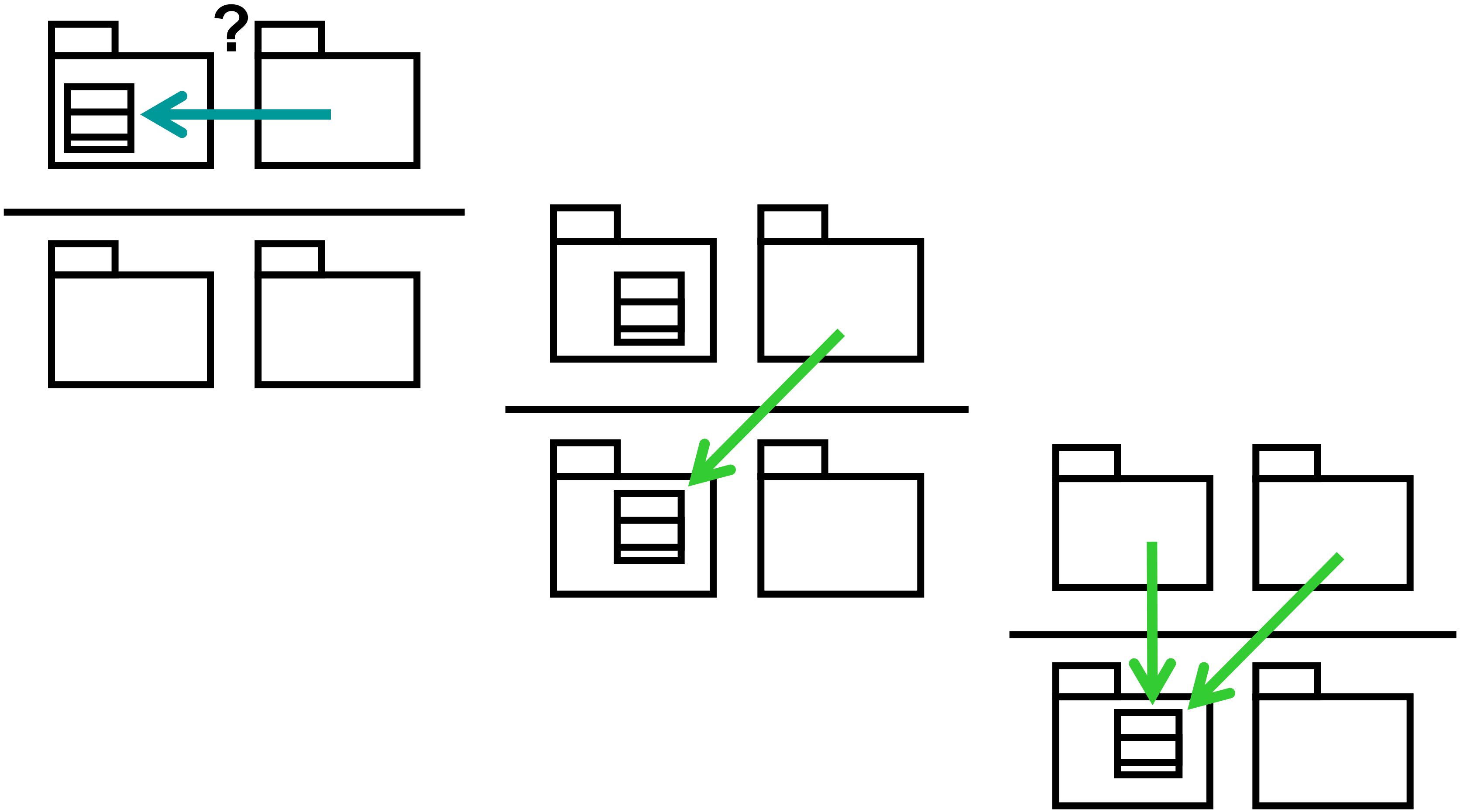
- Purpose
  - To identify where existing subsystems and/or components can be reused based on their interfaces.
- Steps
  - Look for similar interfaces
  - Modify new interfaces to improve the fit
  - Replace candidate interfaces with existing interfaces
  - Map the candidate subsystem to existing components

Possible Reuse Opportunities

- Internal to the system being developed
  - Recognized commonality across packages and subsystems
- External to the system being developed
  - Commercially available components
  - Components from a previously developed application
  - Reverse engineered components

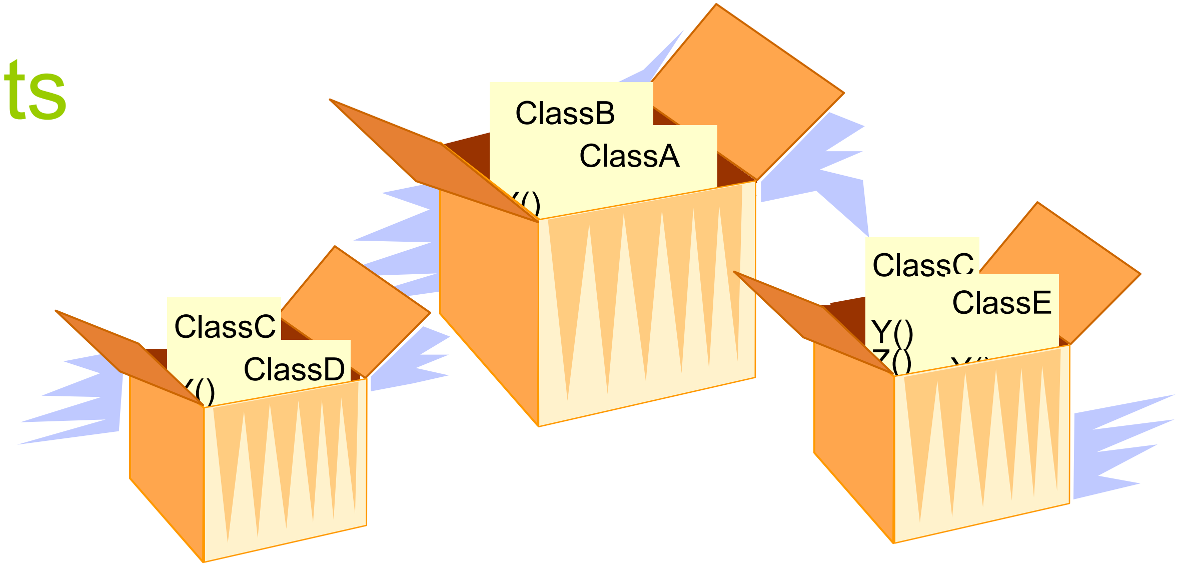


Reuse Opportunities Internal to System

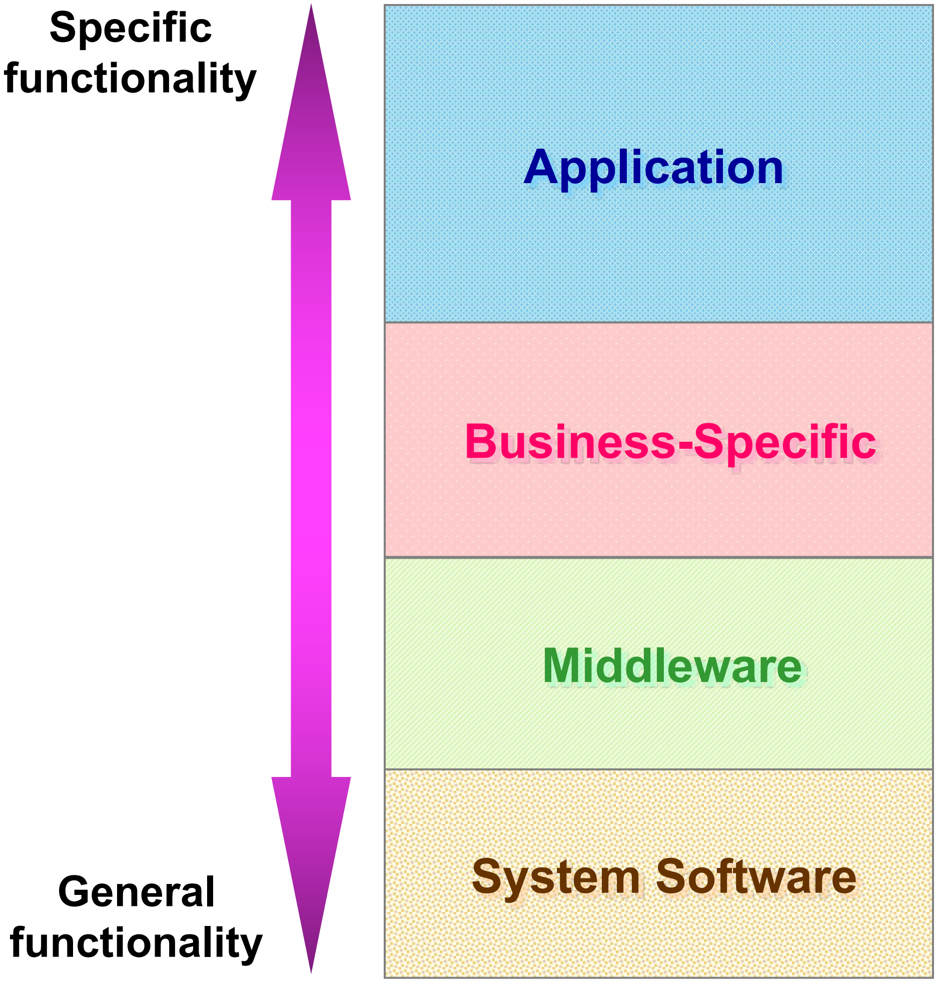


Identify Design Elements Steps

- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints



Review: Typical Layering Approach



Distinct application subsystems that make up an application — contains the value adding software developed by the organization.

Business specific — contains a number of reusable subsystems specific to the type of business.

Middleware — offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

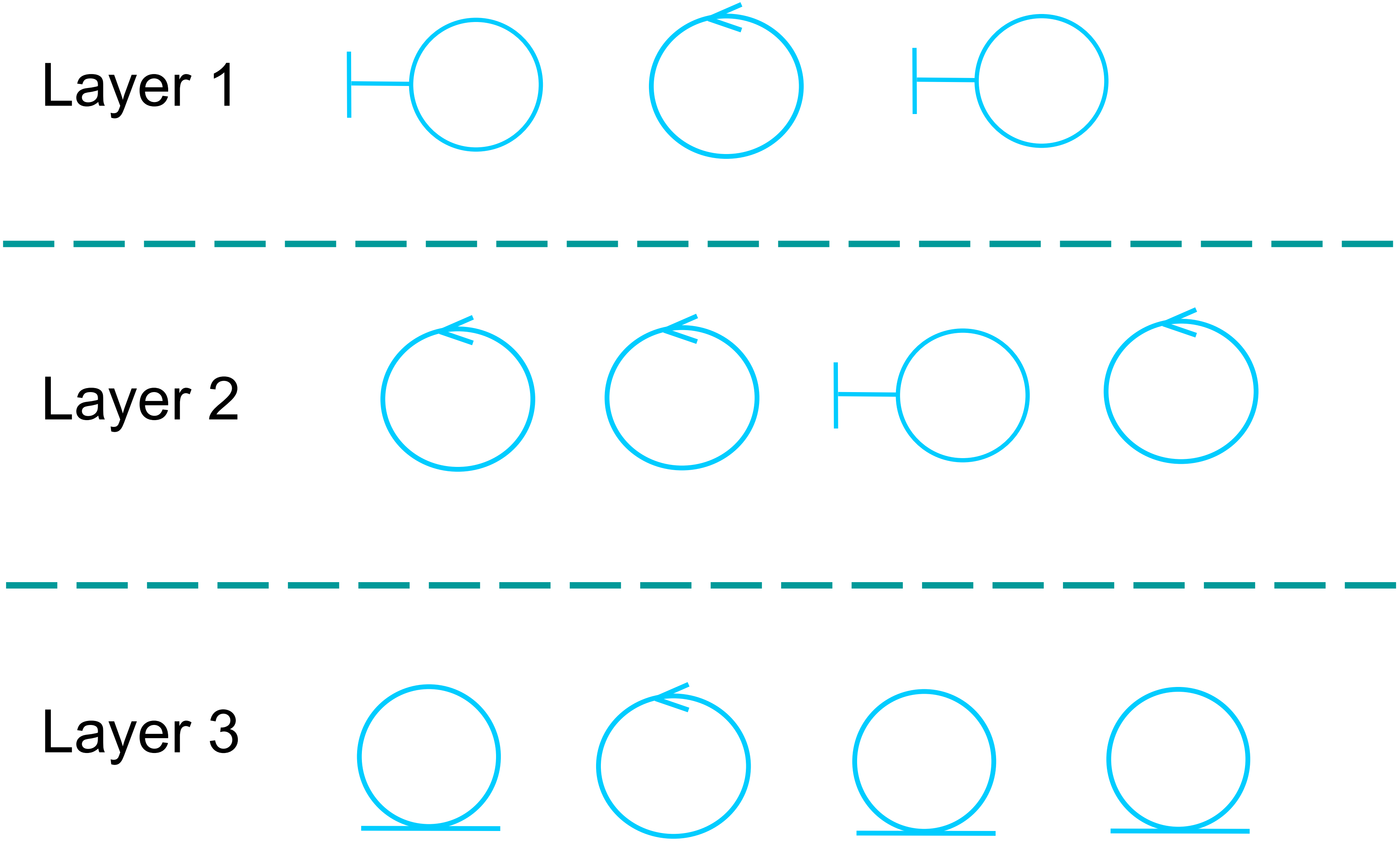
System software — contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers, and so on.

Layering Considerations

- Visibility
  - Dependencies only within current layer and below
- Volatility
  - Upper layers affected by requirements changes
  - Lower layers affected by environment changes
- Generality
  - More abstract model elements in lower layers
- Number of layers
  - Small system: 3-4 layers
  - Complex system: 5-7 layers

Goal is to reduce coupling and to ease maintenance effort.

Design Elements and the Architecture

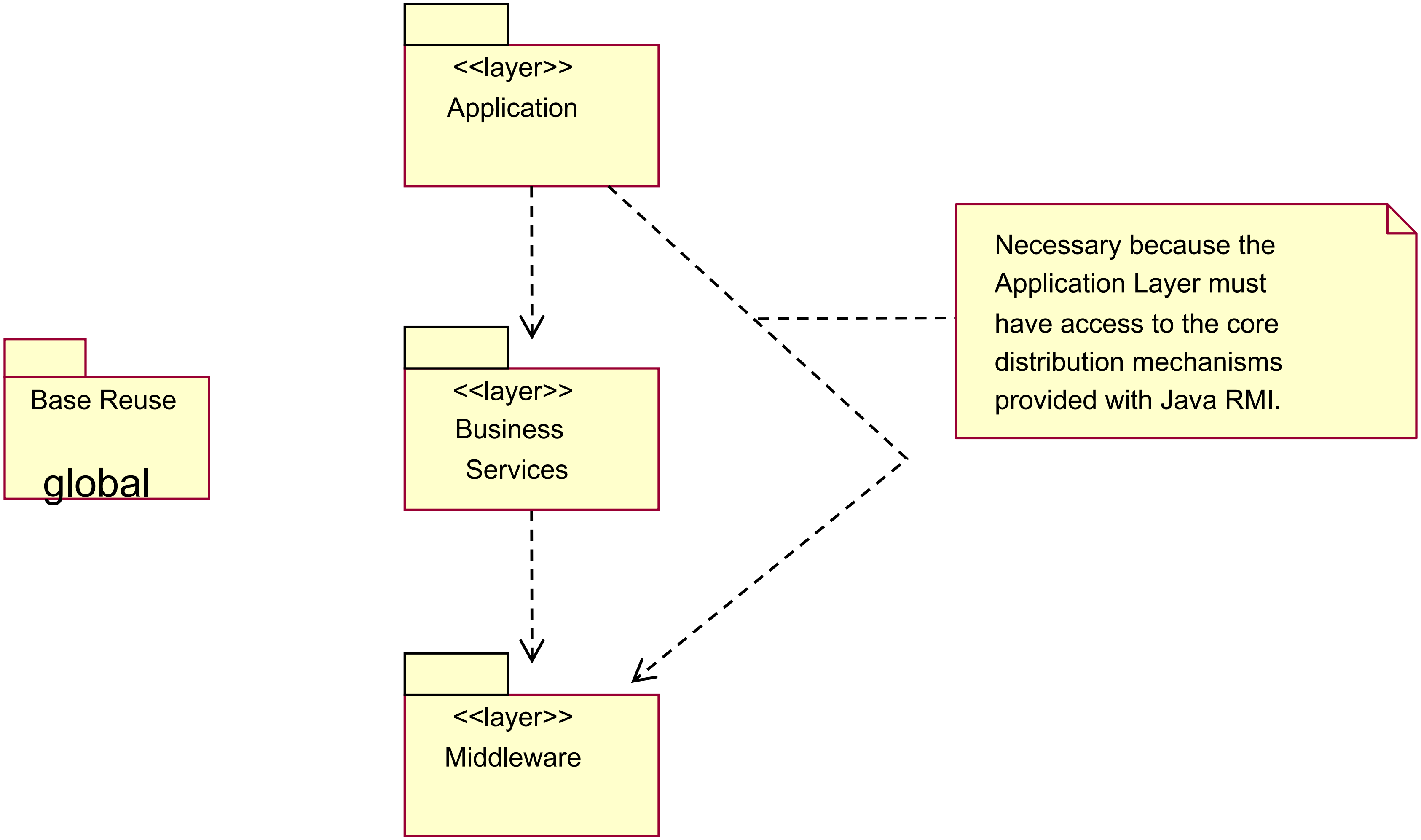


Layer 1

Layer 2

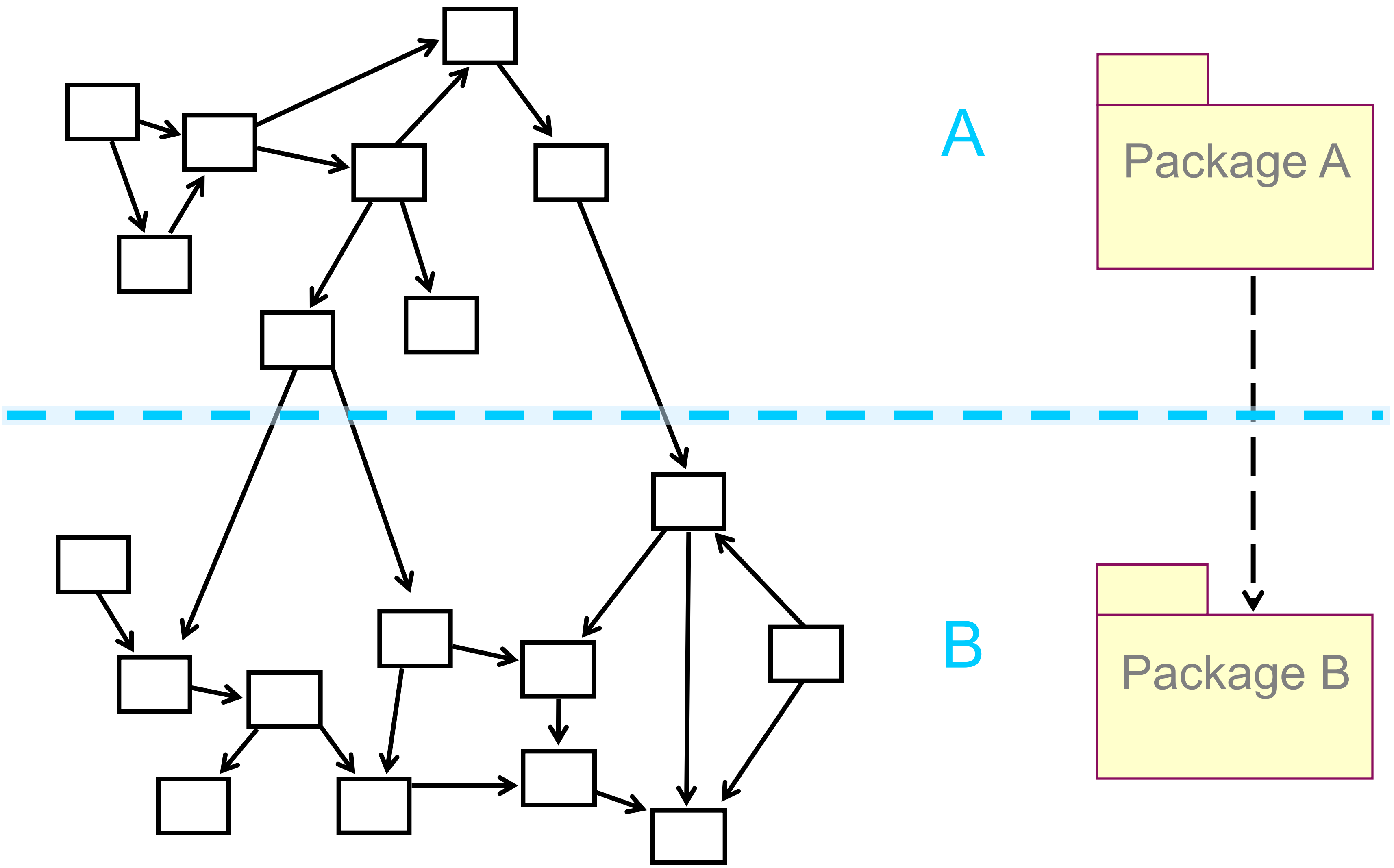
Layer 3

Example: Architectural Layers

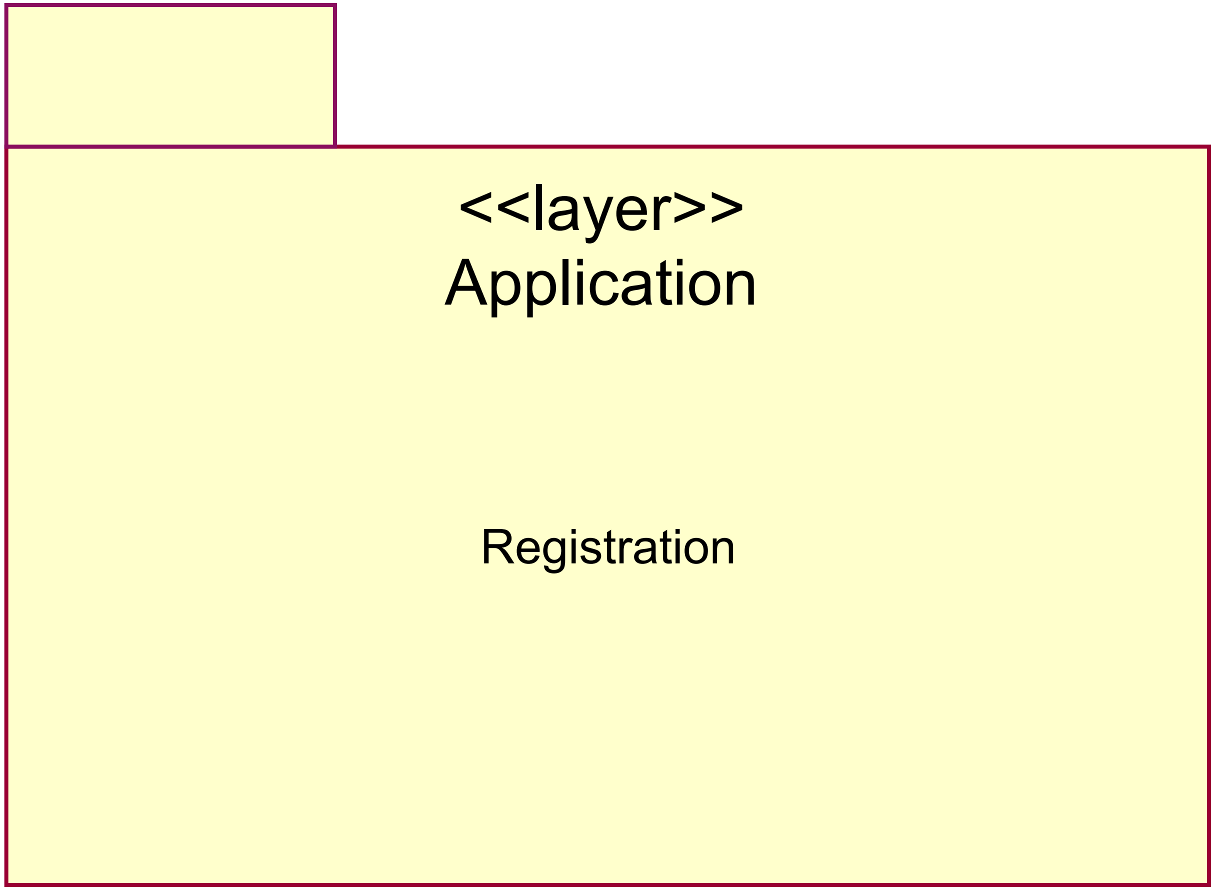


Necessary because the Application Layer must have access to the core distribution mechanisms provided with Java RMI.

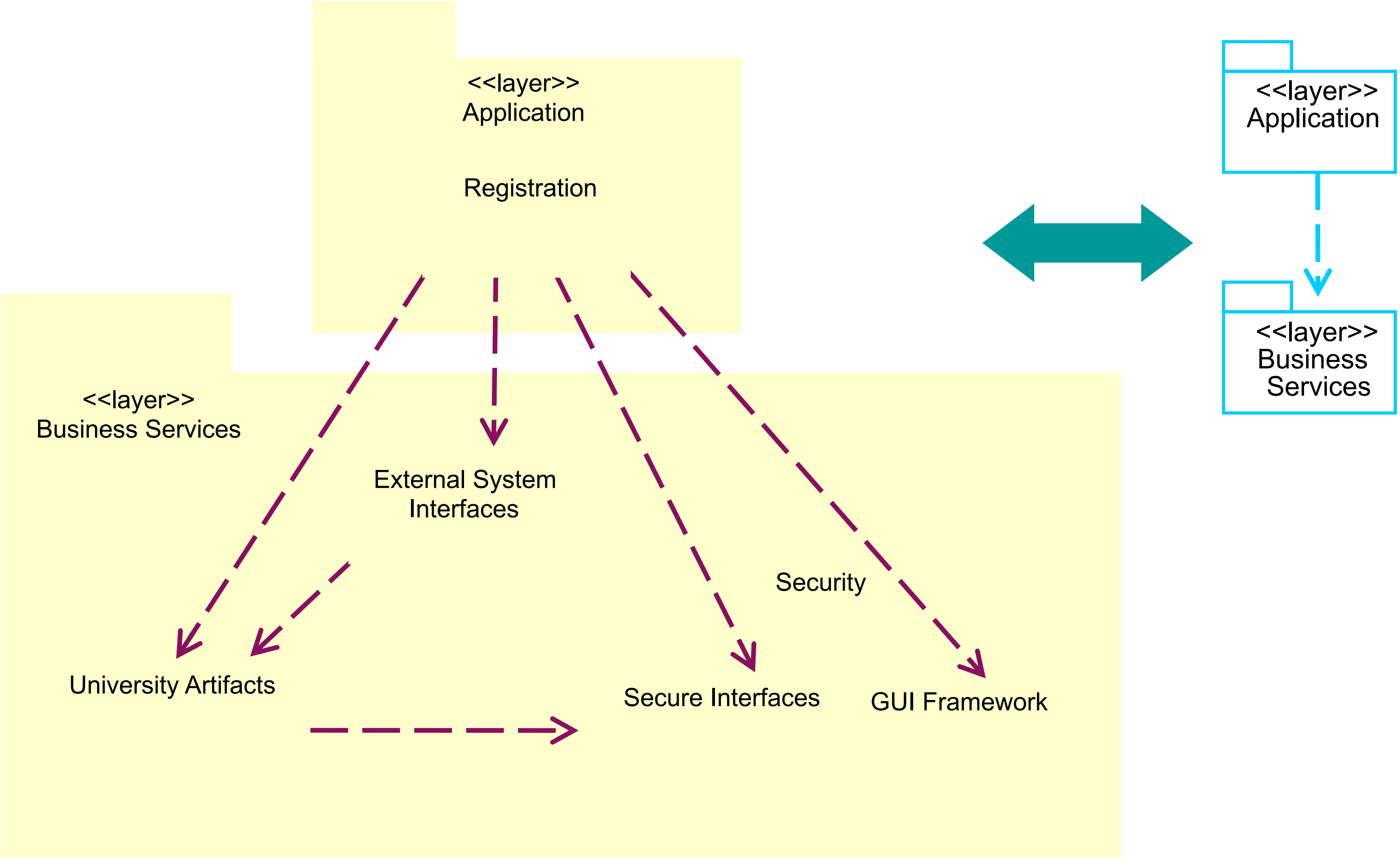
Example: Partitioning



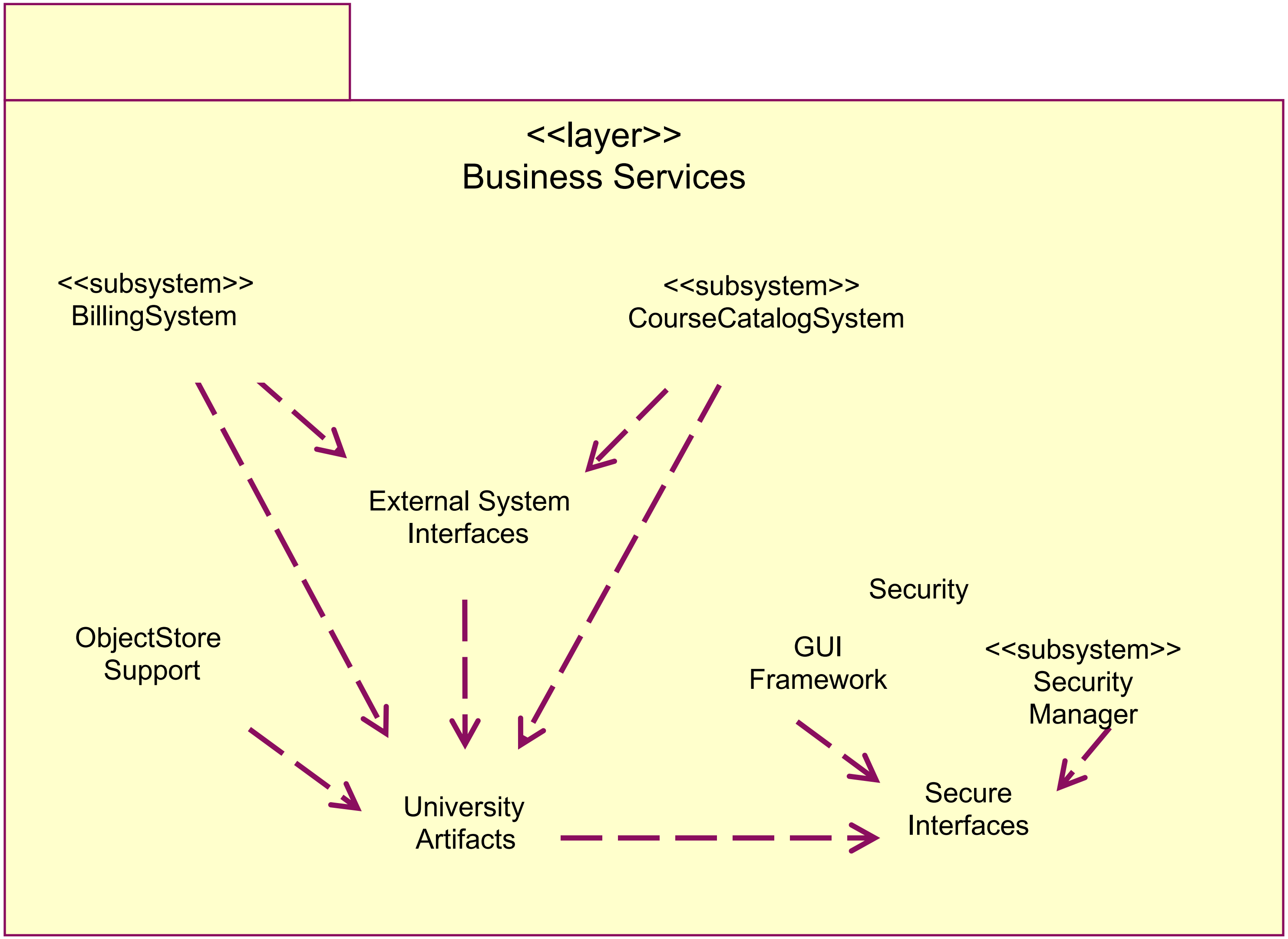
Example: Application Layer



Example: Application Layer Context



Example: Business Services Layer



<<layer>>
Business Services

<<subsystem>>
BillingSystem

<<subsystem>>
CourseCatalogSystem

External System Interfaces

Security

ObjectStore Support

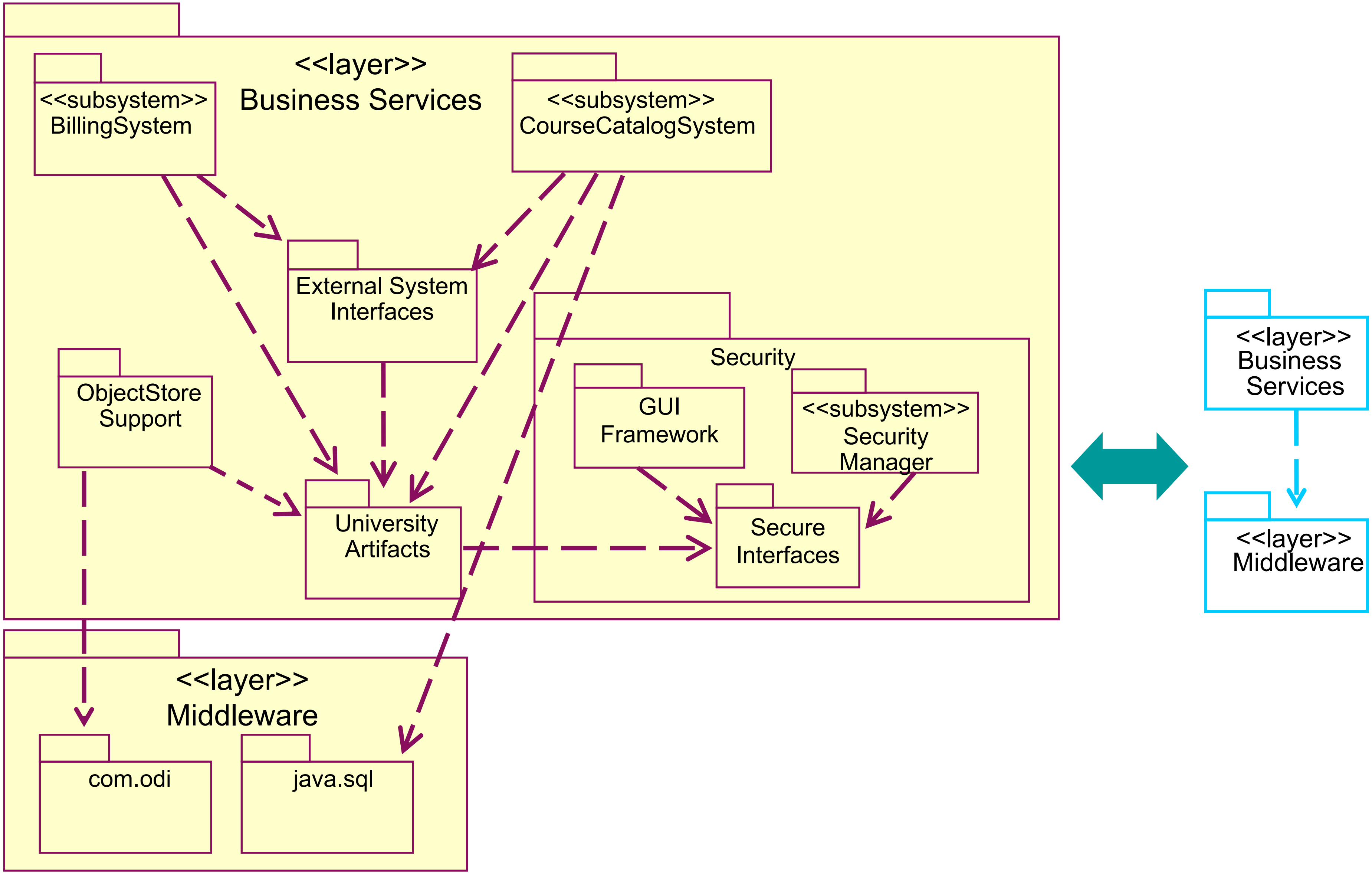
GUI Framework

<<subsystem>>
Security Manager

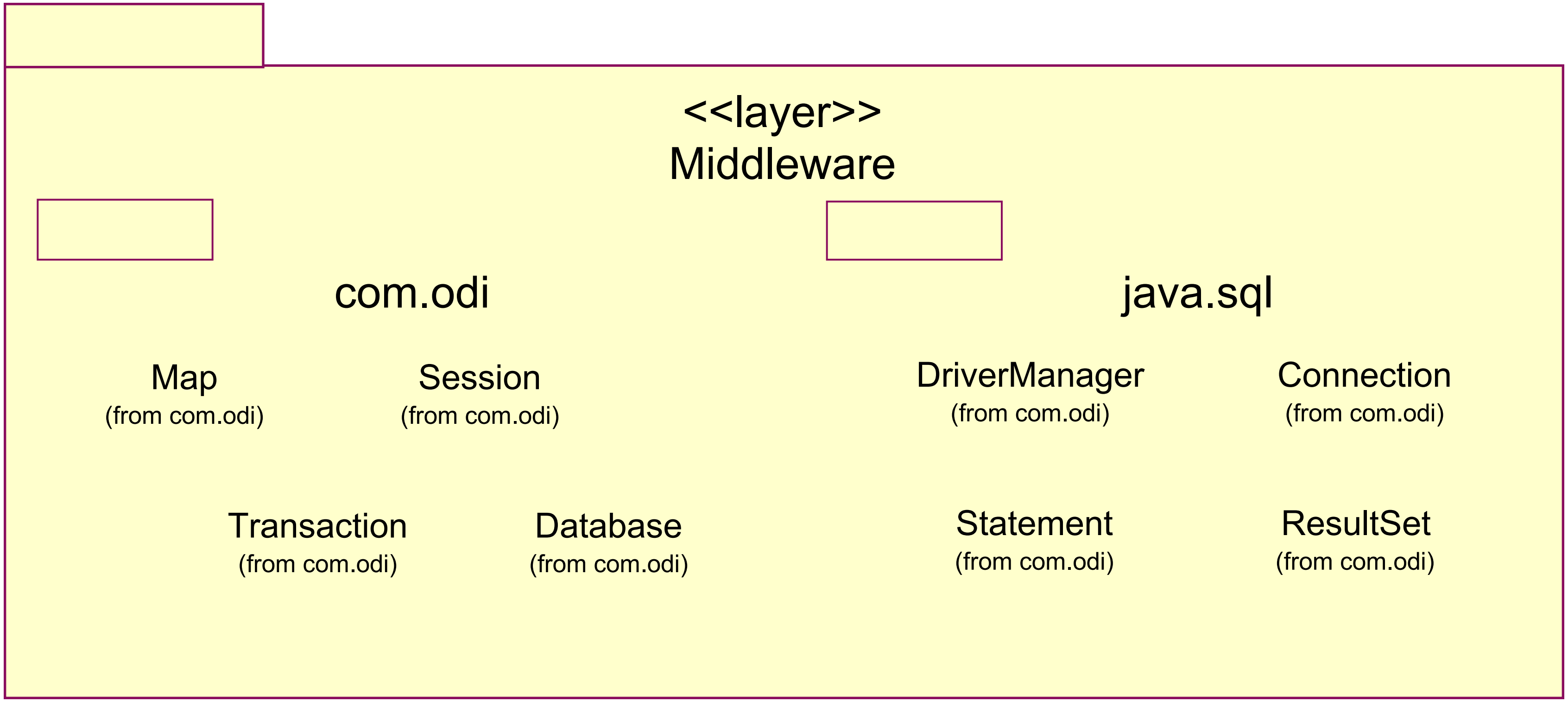
University Artifacts

Secure Interfaces

Example: Business Services Layer Context



Example: Middleware Layer



<<layer>>
Middleware

com.odi

Map
(from com.odi)

Session
(from com.odi)

Transaction
(from com.odi)

Database
(from com.odi)

java.sql

DriverManager
(from com.odi)

Connection
(from com.odi)

Statement
(from com.odi)

ResultSet
(from com.odi)

Identify Design Elements Steps

- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints

Checkpoints

- General
  - Does it provide a comprehensive picture of the services of different packages?
  - Can you find similar structural solutions that can be used more widely in the problem domain?
- Layers
  - Are there more than seven layers?
- Subsystems
  - Is subsystem partitioning done in a logically consistent way across the entire model?

Checkpoints (continued)

- Packages
  - Are the names of the packages descriptive?
  - Does the package description match with the responsibilities of contained classes?
  - Do the package dependencies correspond to the relationships between the contained classes?
  - Do the classes contained in a package belong there according to the criteria for the package division?
  - Are there classes or collaborations of classes within a package that can be separated into an independent package?
  - Is the ratio between the number of packages and the number of classes appropriate?

Checkpoints (continued)

- Classes
  - Does the name of each class clearly reflect the role it plays?
  - Is the class cohesive (i.e., are all parts functionally coupled)?
  - Are all class elements needed by the use-case realizations?
  - Do the role names of the aggregations and associations accurately describe the relationship?
  - Are the multiplicities of the relationships correct?

Review: Identify Design Elements

- What is the purpose of Identify Design Elements?
- What is an interface?
- What is a subsystem? How does it differ from a package?
- What is a subsystem used for, and how do you identify them?
- What are some layering and partitioning considerations?