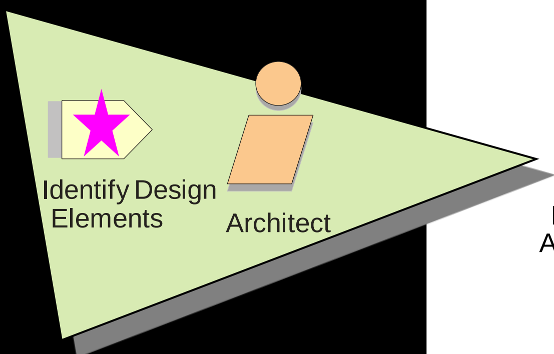
Object-Oriented Analysis and Design

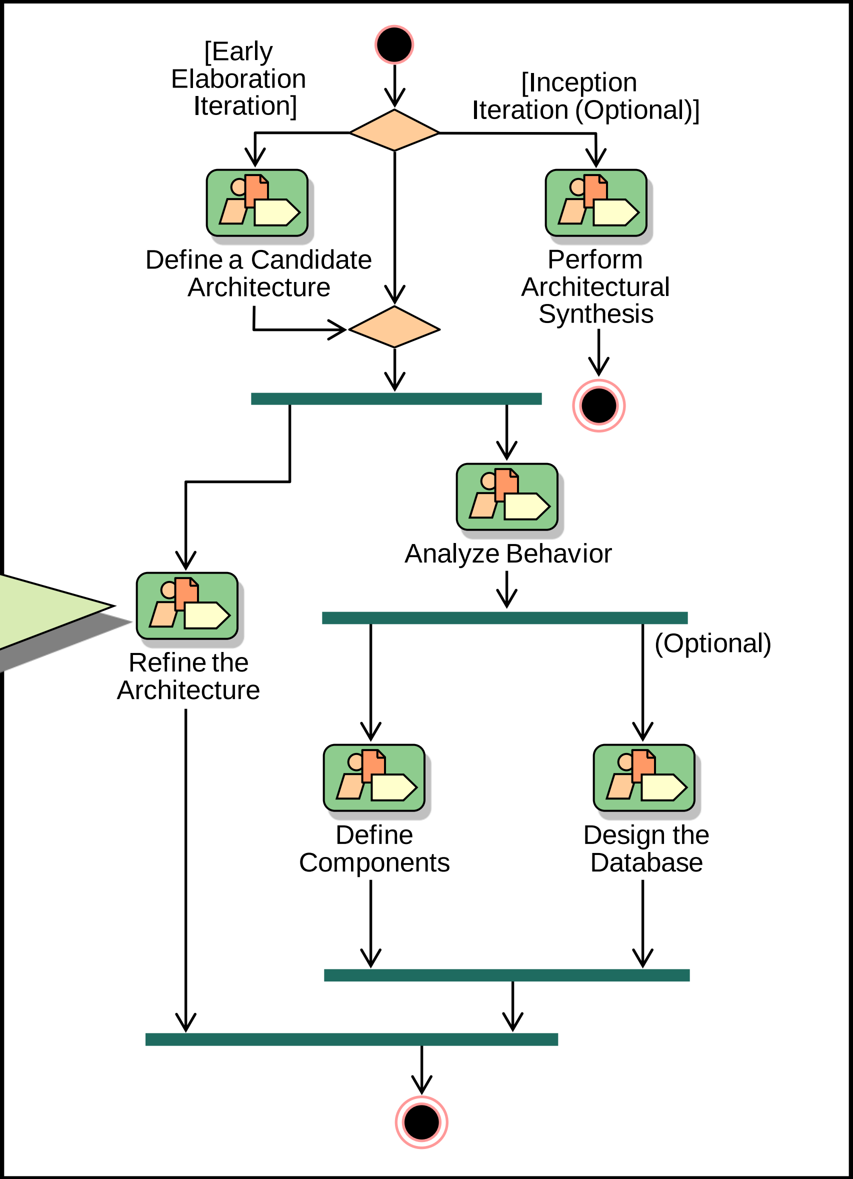
Lecture 7: Identify Design Elements

Objectives: Identify Design Elements

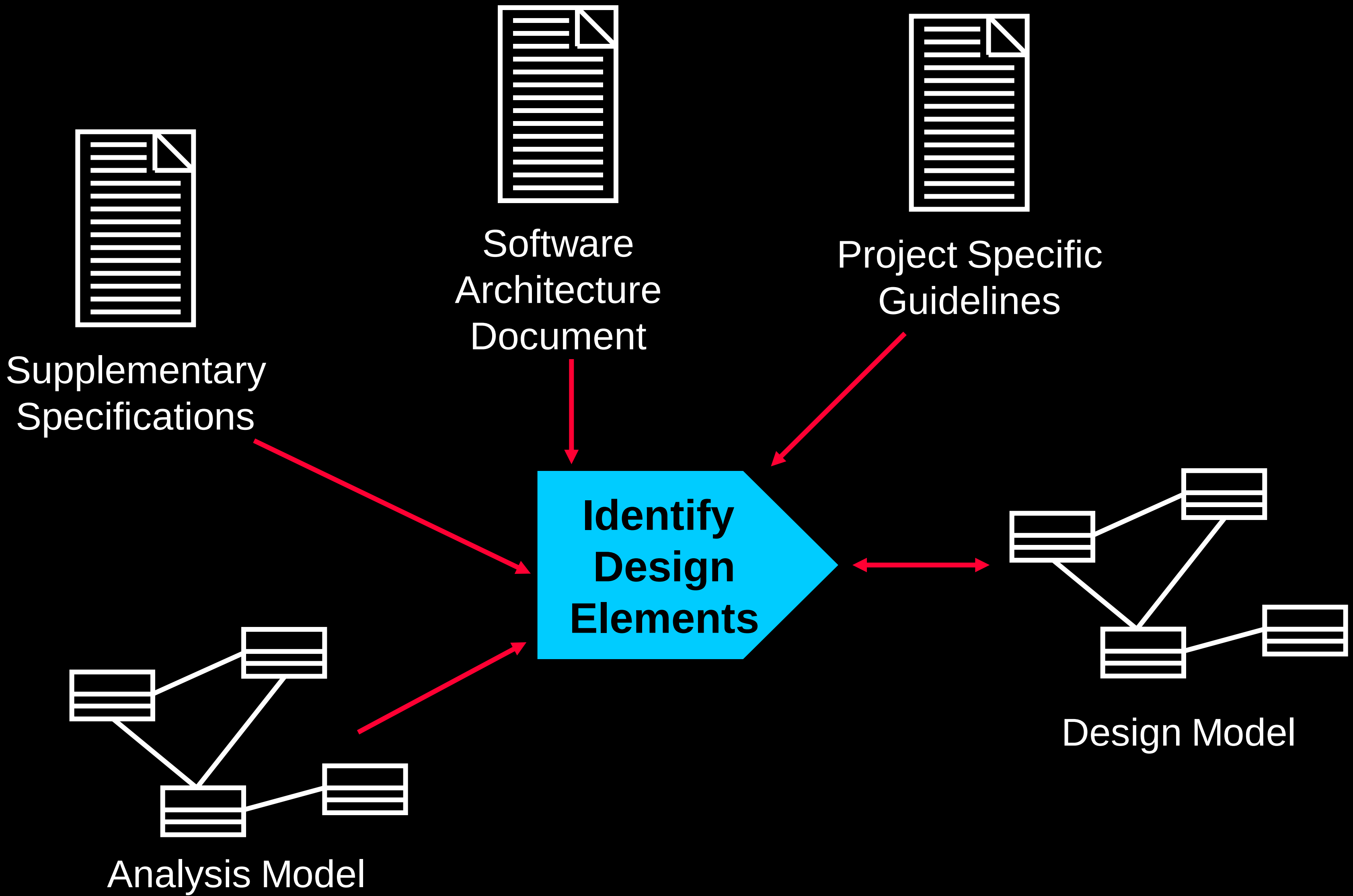
- Define the purpose of Identify Design Elements and demonstrate where in the lifecycle it is performed
- Analyze interactions of analysis classes and identify Design Model elements
  - Design classes
  - Subsystems
  - Subsystem interfaces

Identify Design Elements in Context





Identify Design Elements Overview

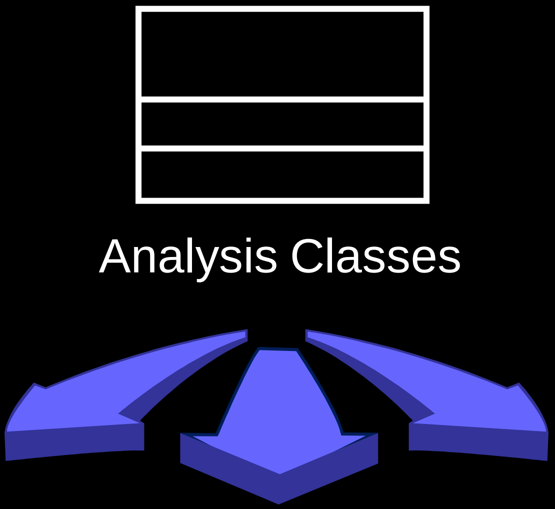


Identify Design Elements Steps

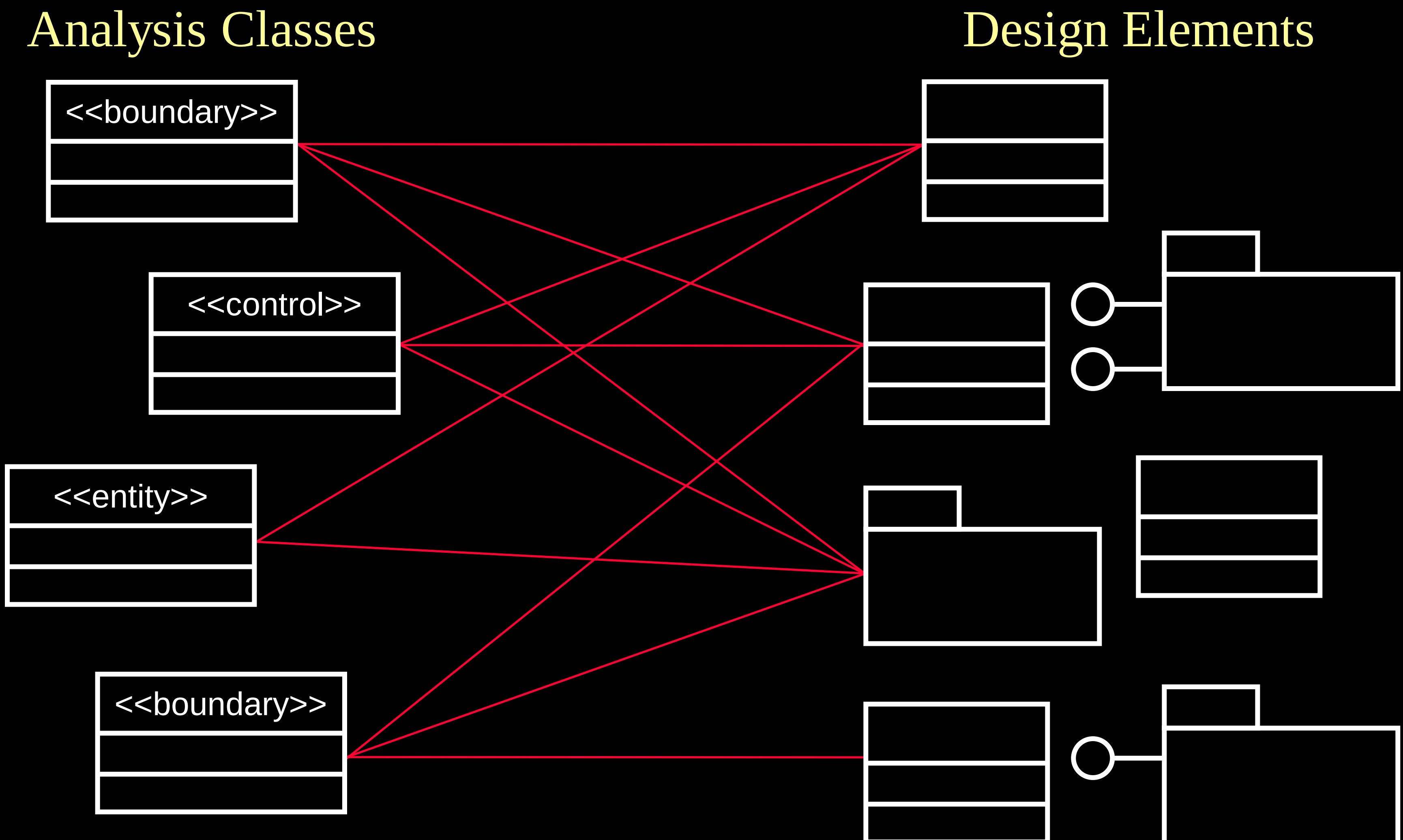
- Identify classes and subsystems
- Identify subsystem interfaces
- Update the organization of the Design Model
- Checkpoints

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



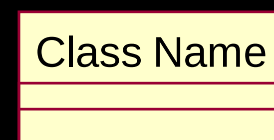
Many-to-Many Mapping

Identifying Design Classes

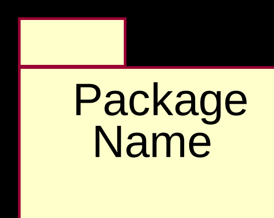
- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
- More complex analysis classes may
  - Split into multiple classes
  - Become a package
  - Become a subsystem (discussed later)
  - Any combination …

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

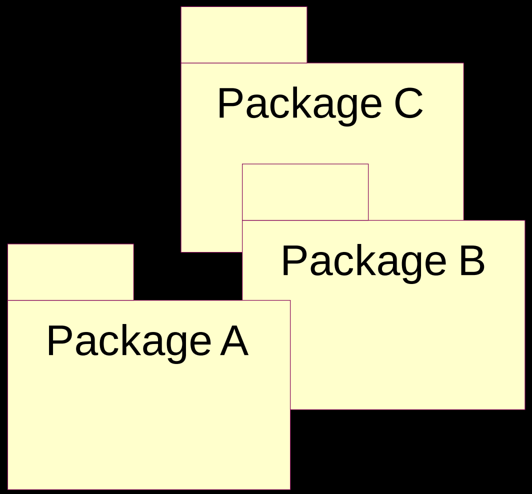


- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain other model elements



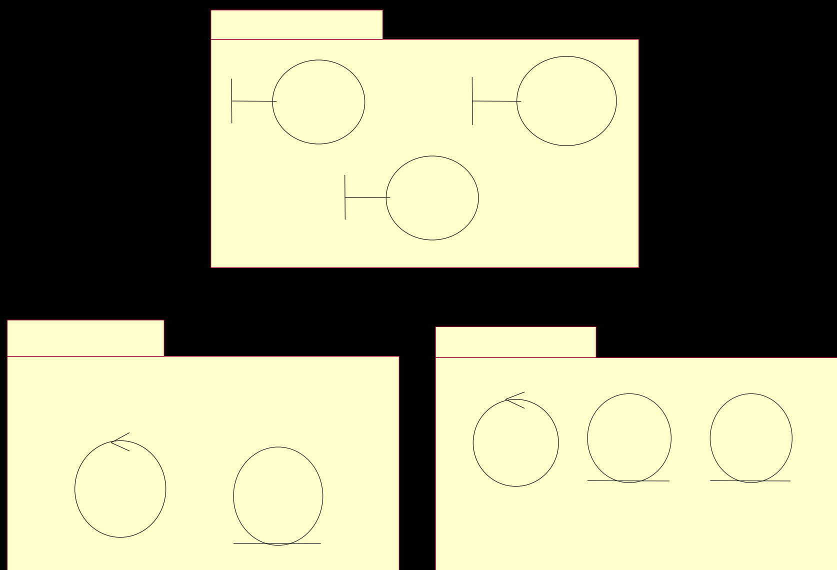
Group Design Classes in Packages

- You can base your packaging criteria on a number of different factors, including:
  - Configuration units
  - Allocation of resources among development teams
  - Reflect the user types
  - Represent the existing products and services the system uses



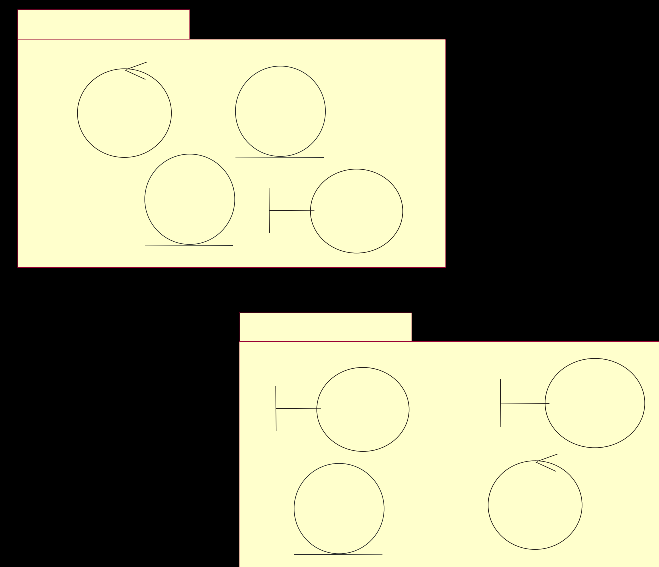
Packaging Tips: Boundary Classes

If it is <u>likely</u> the system interface will undergo considerable changes



Boundary classes placed in separate packages

If it is <u>unlikely</u> the system interface will undergo considerable changes



Boundary classes packaged with functionally related classes

Packaging Tips: Functionally Related Classes

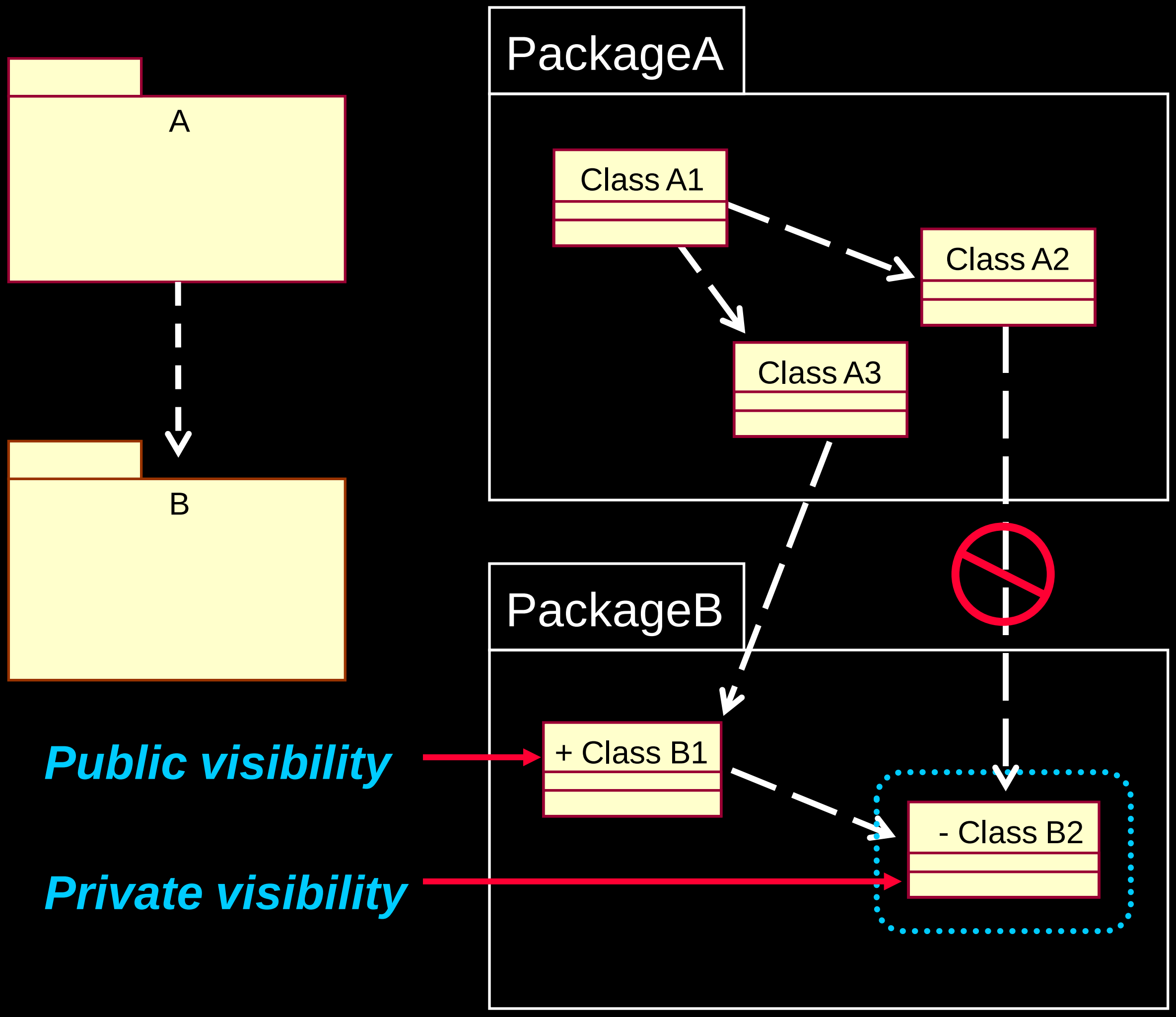
- Criteria for determining if classes are functionally related:
  - Changes in one class' behavior and/or structure necessitate changes in another class
  - Removal of one class impacts the other class
  - Two objects interact with a large number of messages or have a complex intercommunication
  - A boundary class can be functionally related to a particular entity class if the function of the boundary class is to present the entity class
  - Two classes interact with, or are affected by changes in the same actor

Continued…

Packaging Tips: Functionally Related Classes (cont.)

- Criteria for determining if classes are functionally related (continued):
  - Two classes have relationships between each other
  - One class creates instances of another class
- Criteria for determining when two classes should *NOT* be placed in the same package:
  - Two classes that are related to different actors should not be placed in the same package
  - An optional and a mandatory class should not be placed in the same package

Package Dependencies: Package Element Visibility

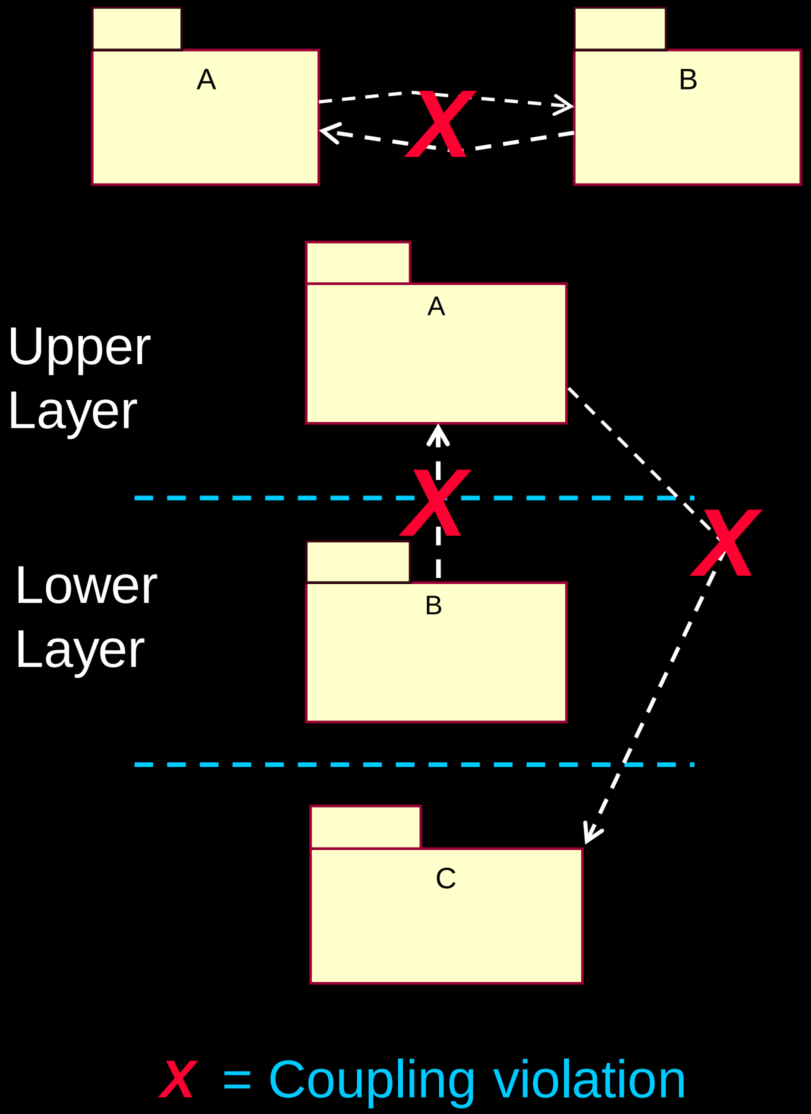


Only public classes can be referenced outside of the owning package

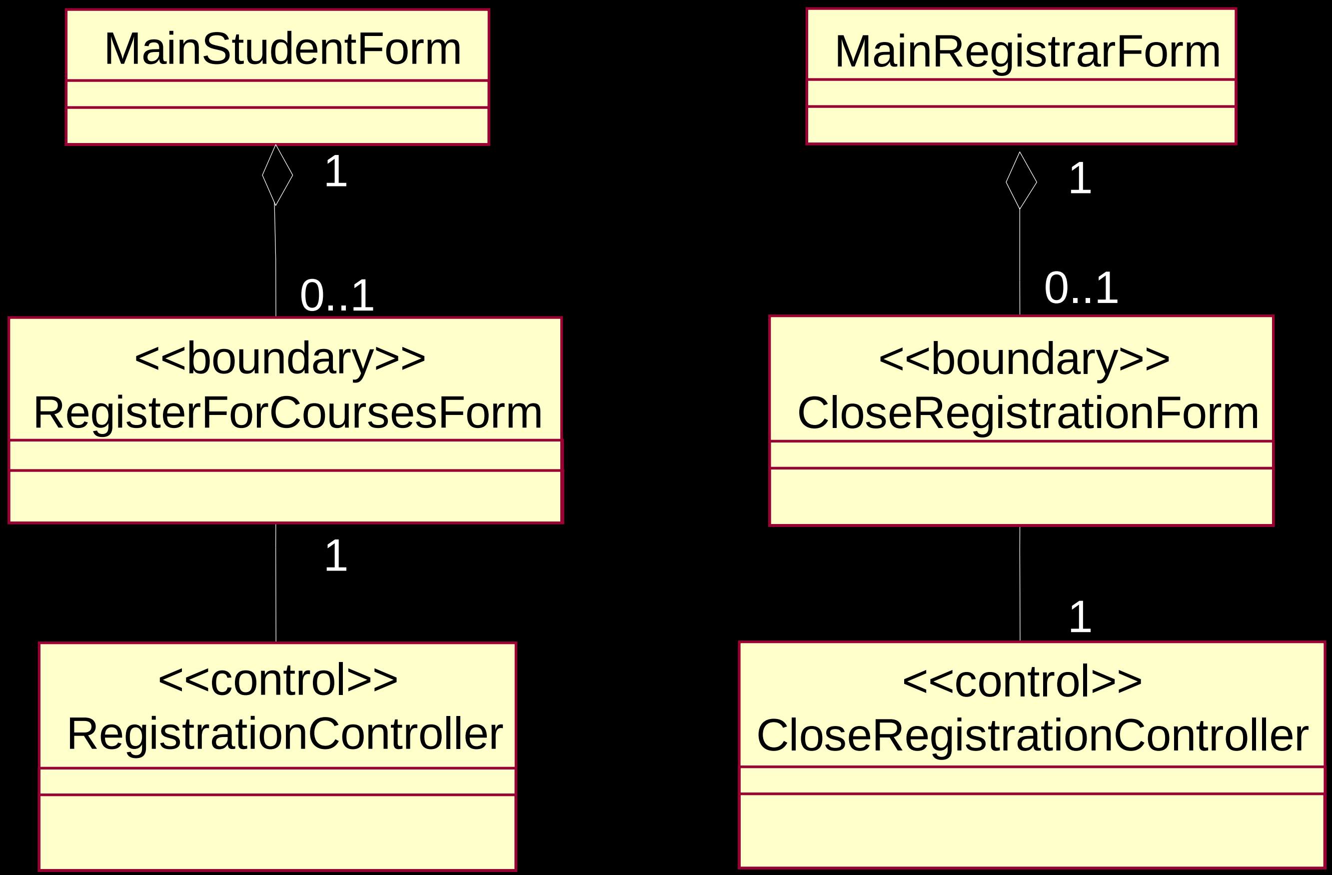
OO Principle: Encapsulation

Package Coupling: Tips

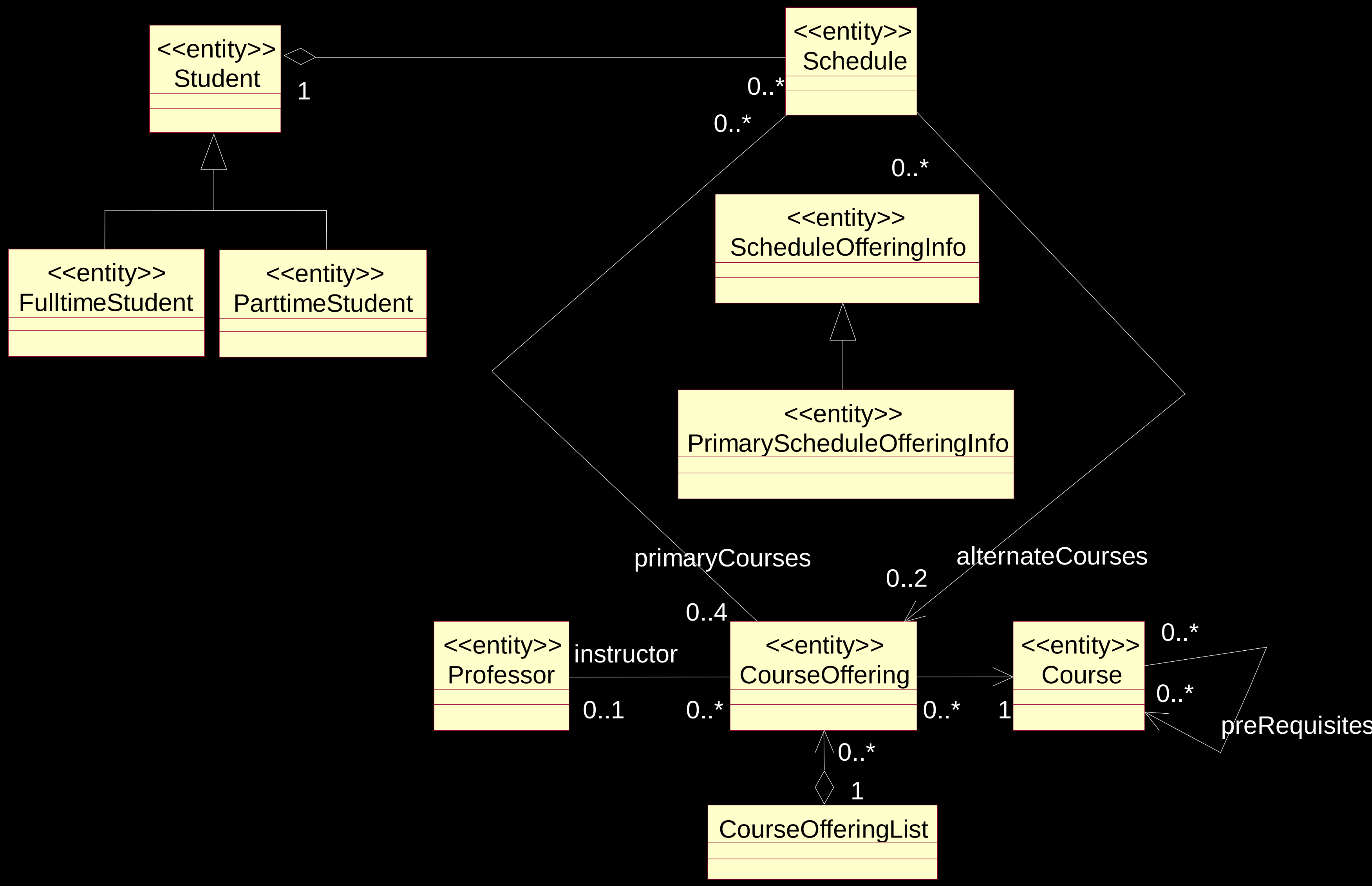
- Packages should not be cross-coupled
- Packages in lower layers should not be dependent upon packages in upper layers
- In general, dependencies should not skip layers



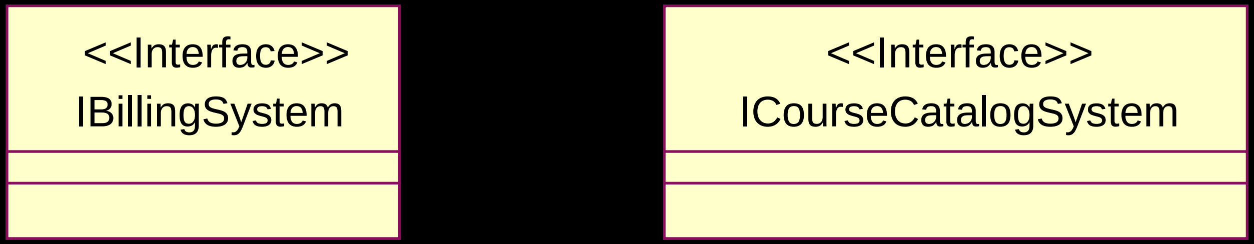
Example: Registration Package



Example: University Artifacts Package

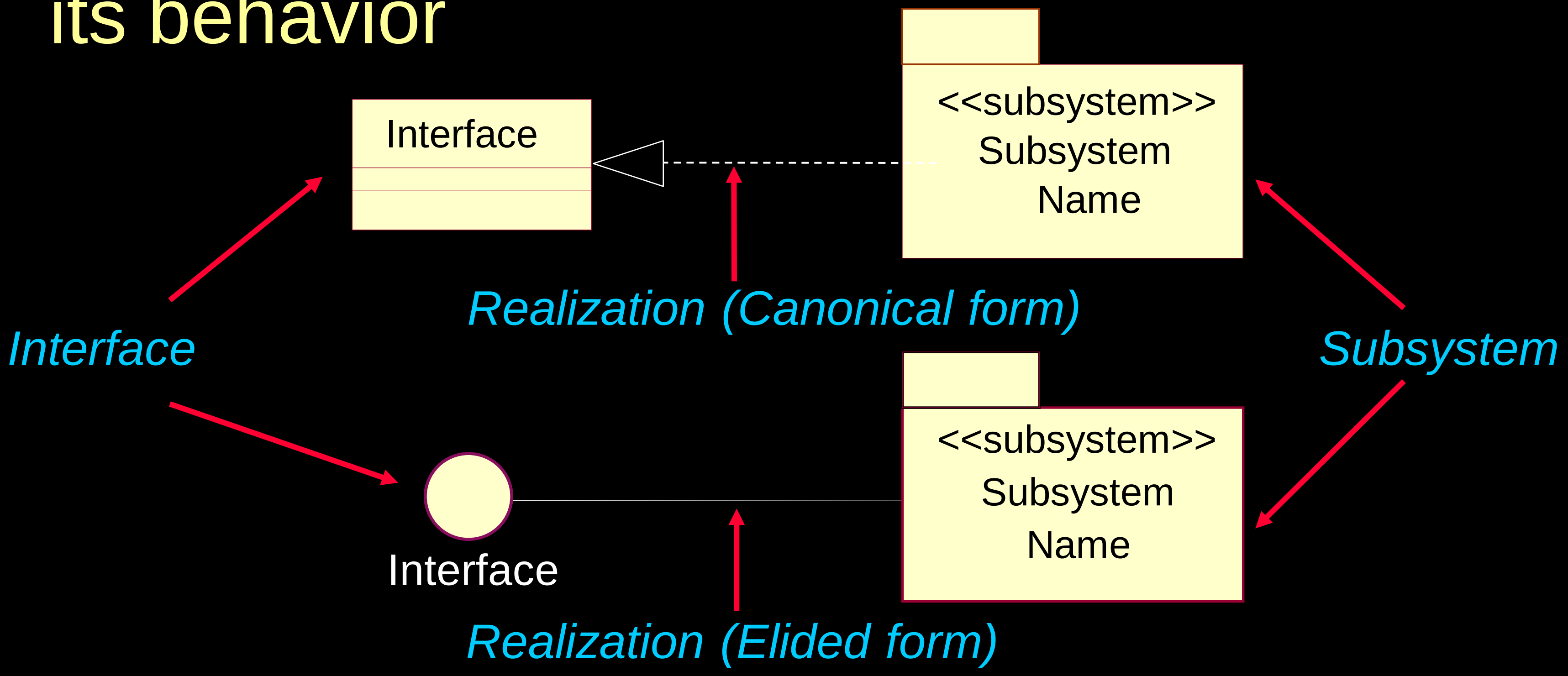


Example: External System Interfaces Package



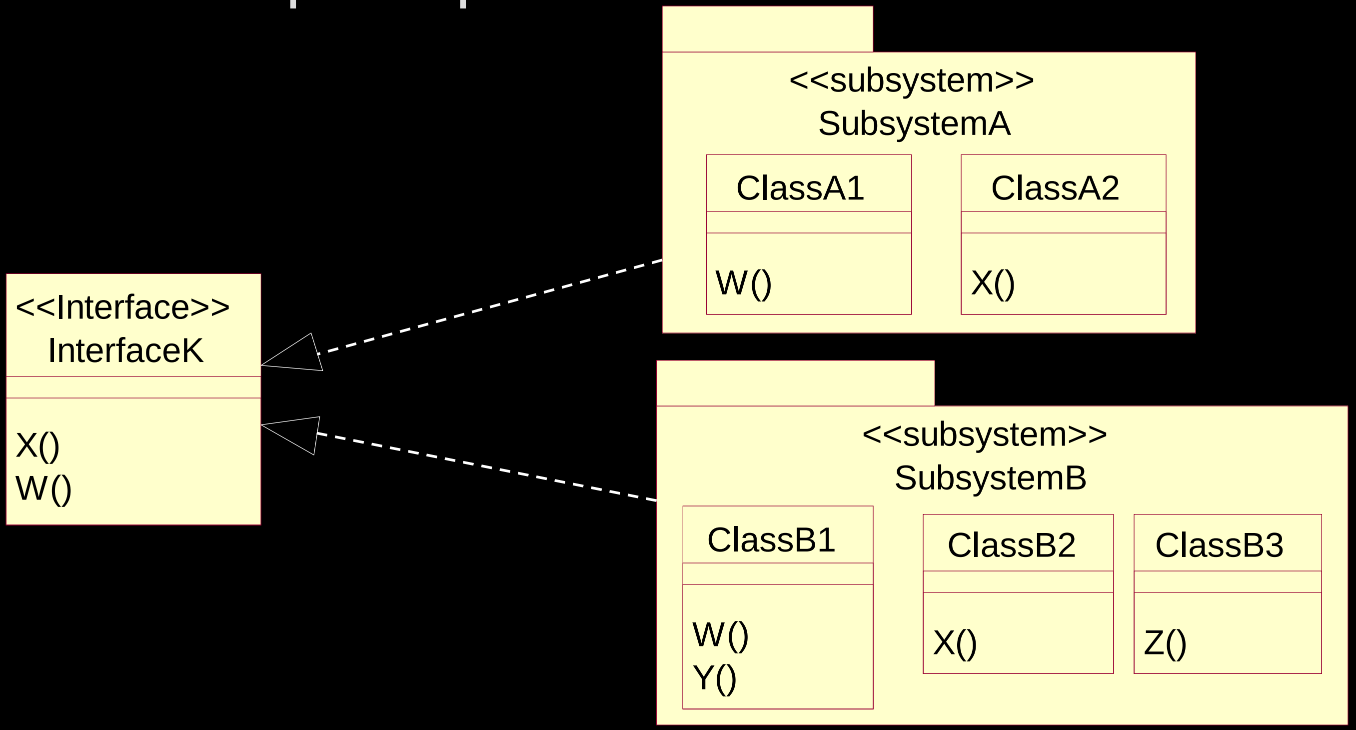
Review: Subsystems and Interfaces

- Are a “cross between” a package (can contain other model elements) and a class (has behavior)
- Realizes one or more interfaces that define its behavior



Subsystems and Interfaces (cont.)

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



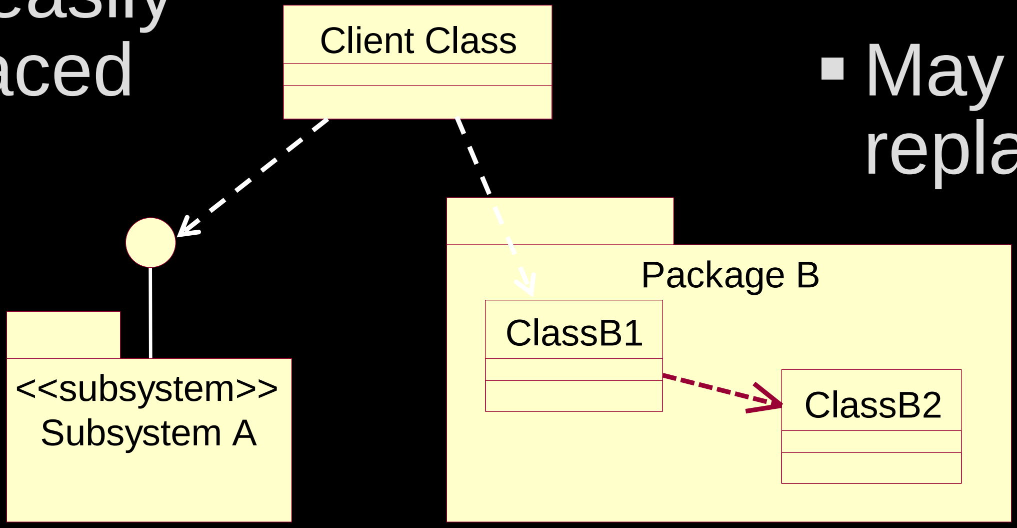
Packages versus Subsystems

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!

Subsystem Usage

- Subsystems can be used to partition the system into parts that can be independently:
  - ordered, configured, or delivered
  - developed, as long as the interfaces remain unchanged
  - deployed across a set of distributed computational nodes
  - changed without breaking other parts of the systems
- Subsystems can also be used to:
  - partition the system into units which can provide restricted security over key resources
  - represent existing products or external systems in the design (e.g. components)

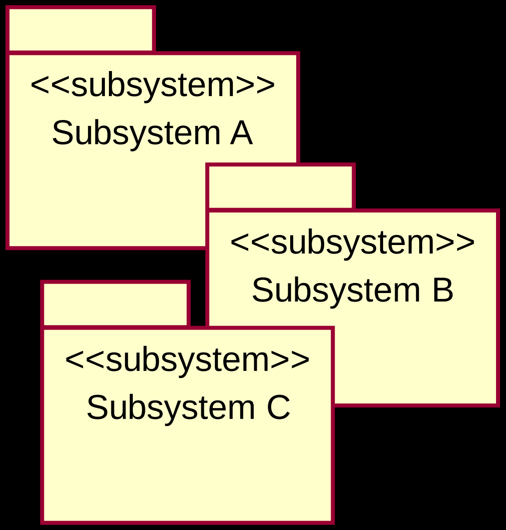
Subsystems raise the level of abstraction

Identifying Subsystems Hints

- Look at object collaborations.
- Look for optionality.
- Look to the user interface of the system.
- Look to the actors.
- Look for coupling and cohesion between classes.
- Look at substitution.
- Look at distribution.
- Look at volatility.

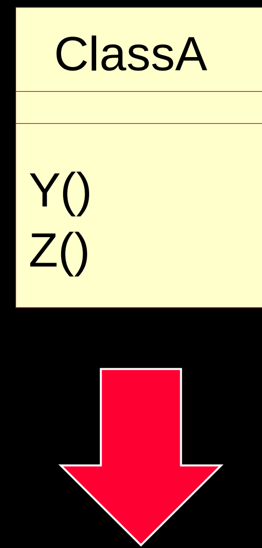
Candidate Subsystems

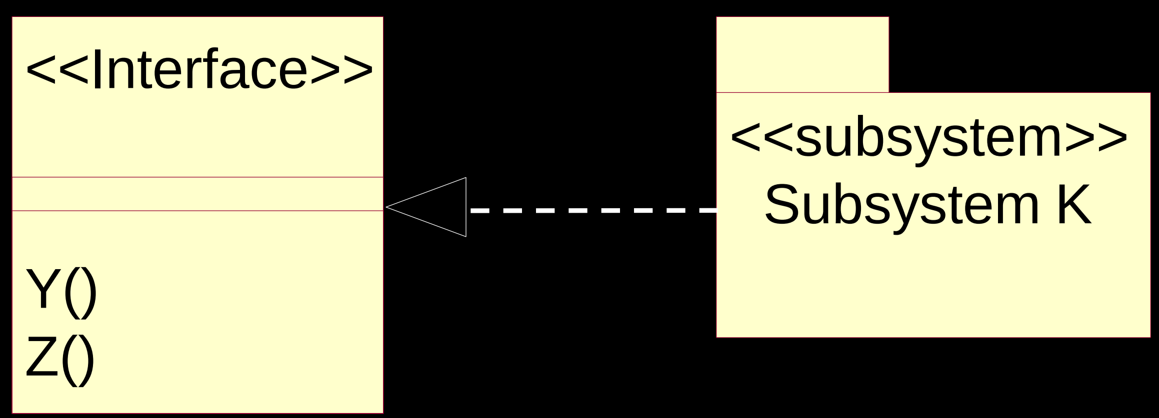
- Analysis classes which may evolve into subsystems:
  - Classes providing complex services and/or utilities
  - Boundary classes (user interfaces and external system interfaces)
- Existing products or external systems in the design (e.g., components):
  - Communication software
  - Database access support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



Identifying Subsystems

"Superman Class"





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.

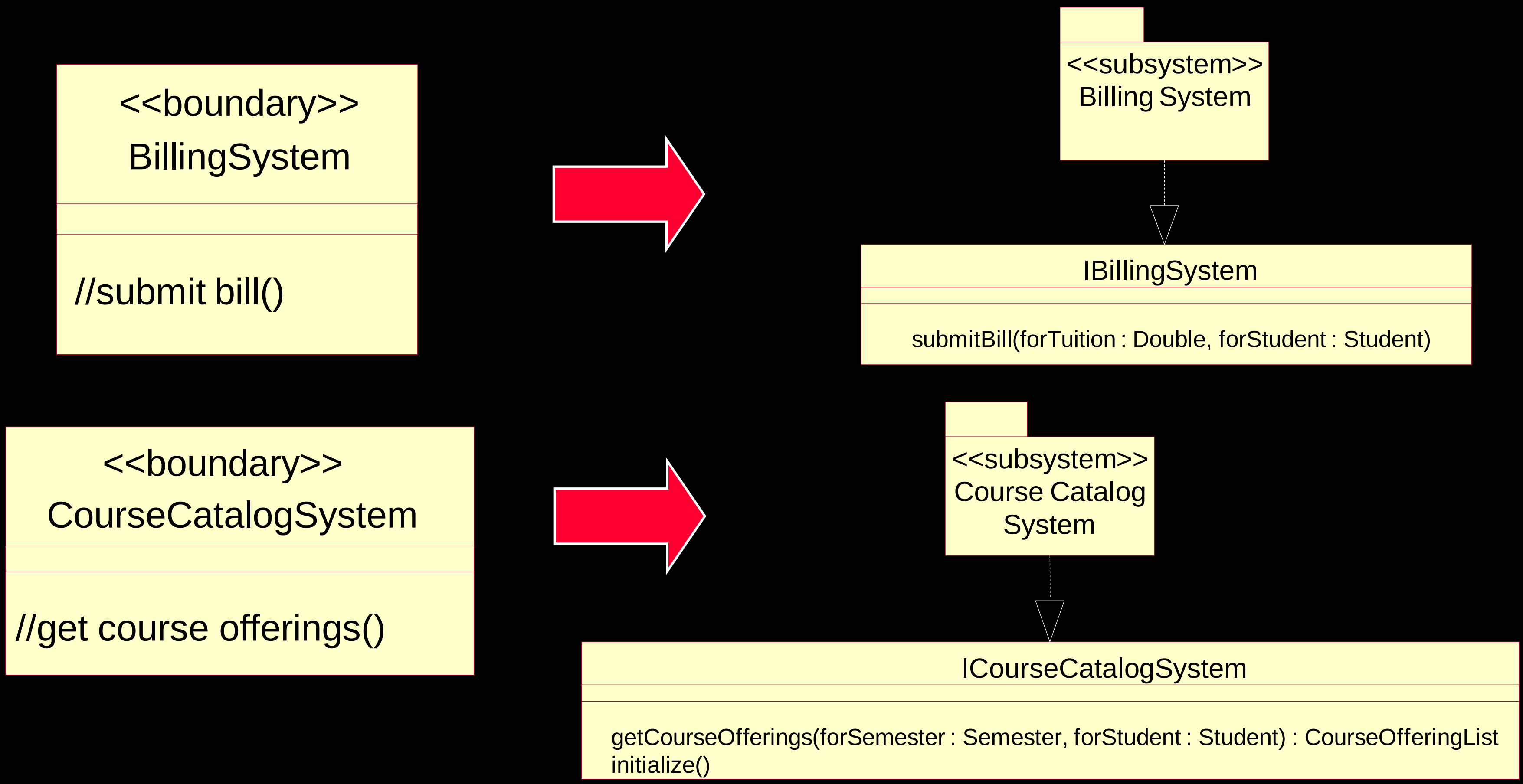
Stable, well-defined interfaces are key to a stable, resilient architecture.

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

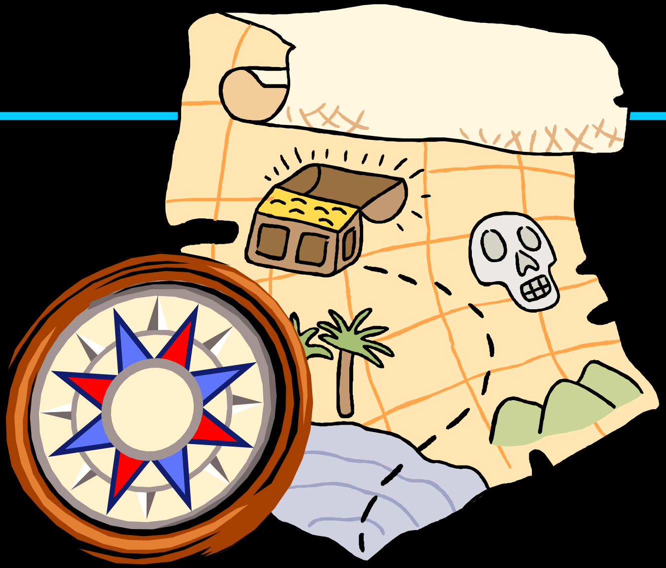
Analysis | Design



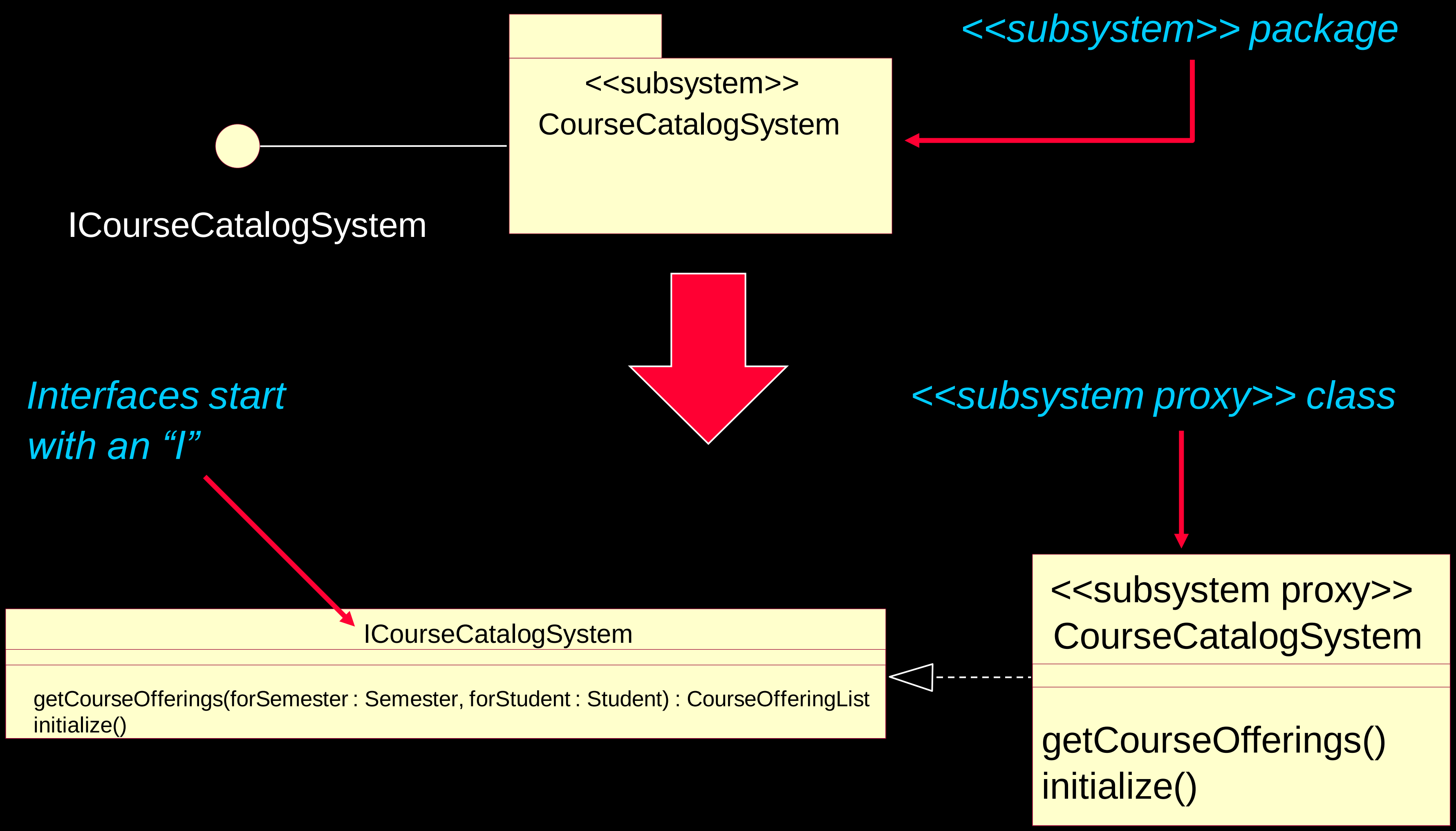
All other analysis classes map directly to design classes

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

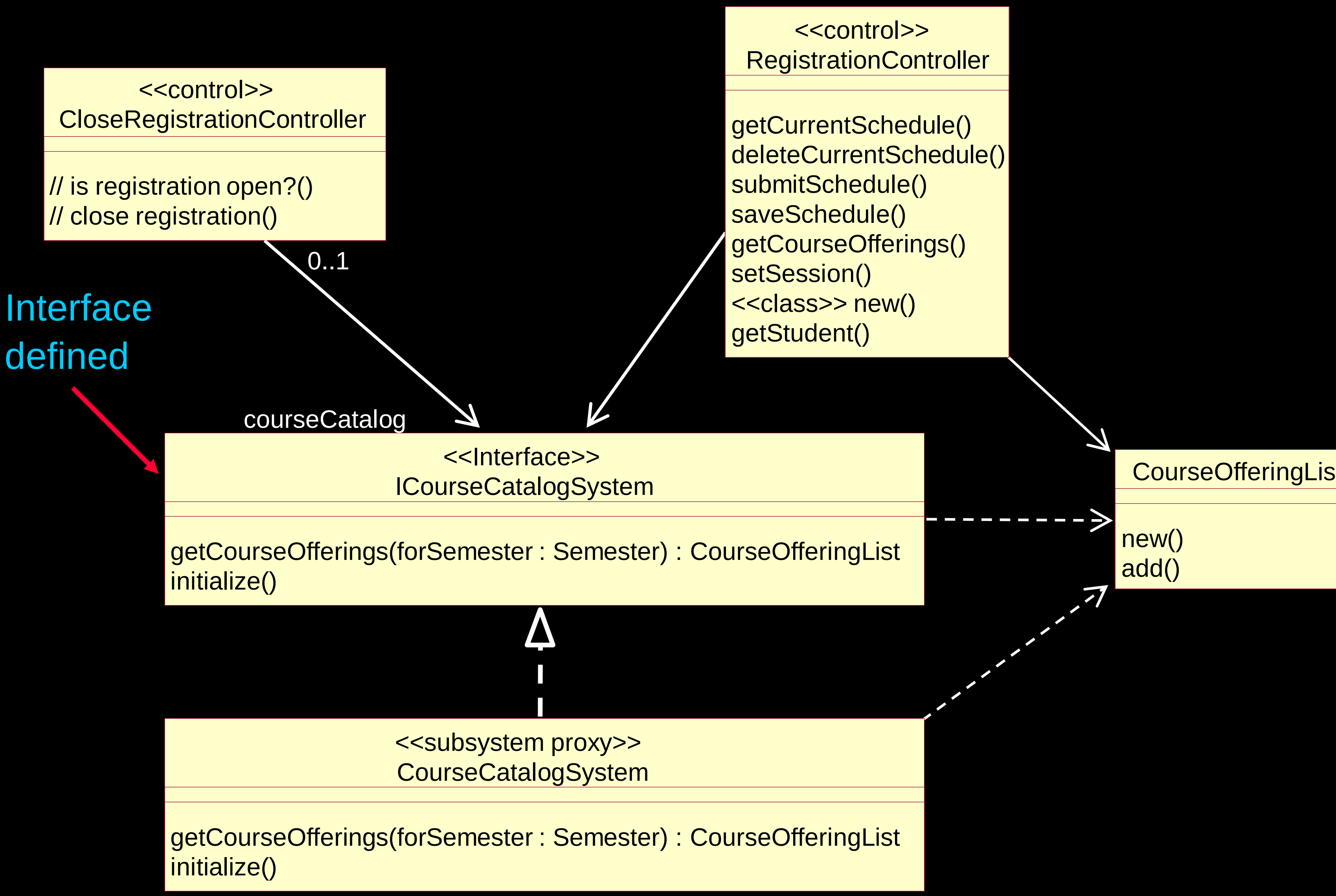
Analysis Class	Design Element
CourseCatalogSystem	CourseCatalogSystem Subsystem
BillingSystem	BillingSystem Subsystem
All other analysis classes map directly to design classes	



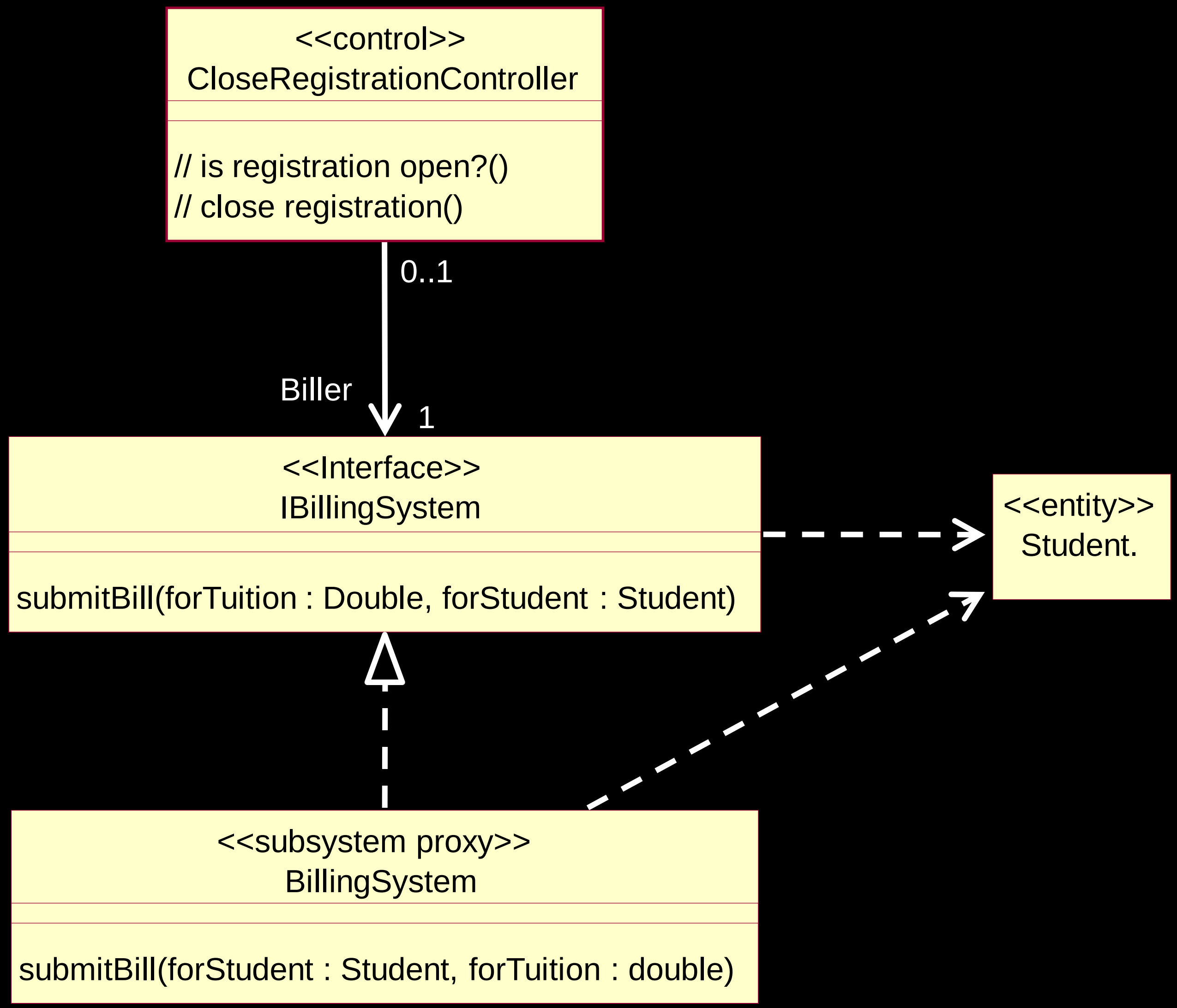
Modeling Convention: Subsystems and Interfaces



Example: Subsystem Context: CourseCatalogSystem



Example: Subsystem Context: Billing System



Identify Design Elements Steps

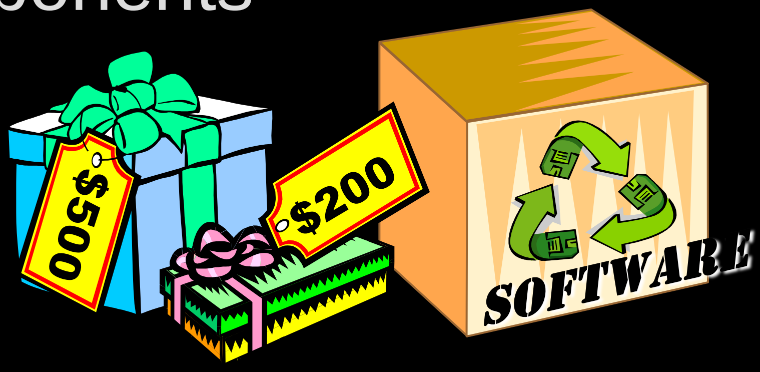
- Identify classes and subsystems
- Identify subsystem interfaces
- ★ Identify reuse opportunities
- Update the organization of the Design Model
- Checkpoints

Identification of Reuse Opportunities

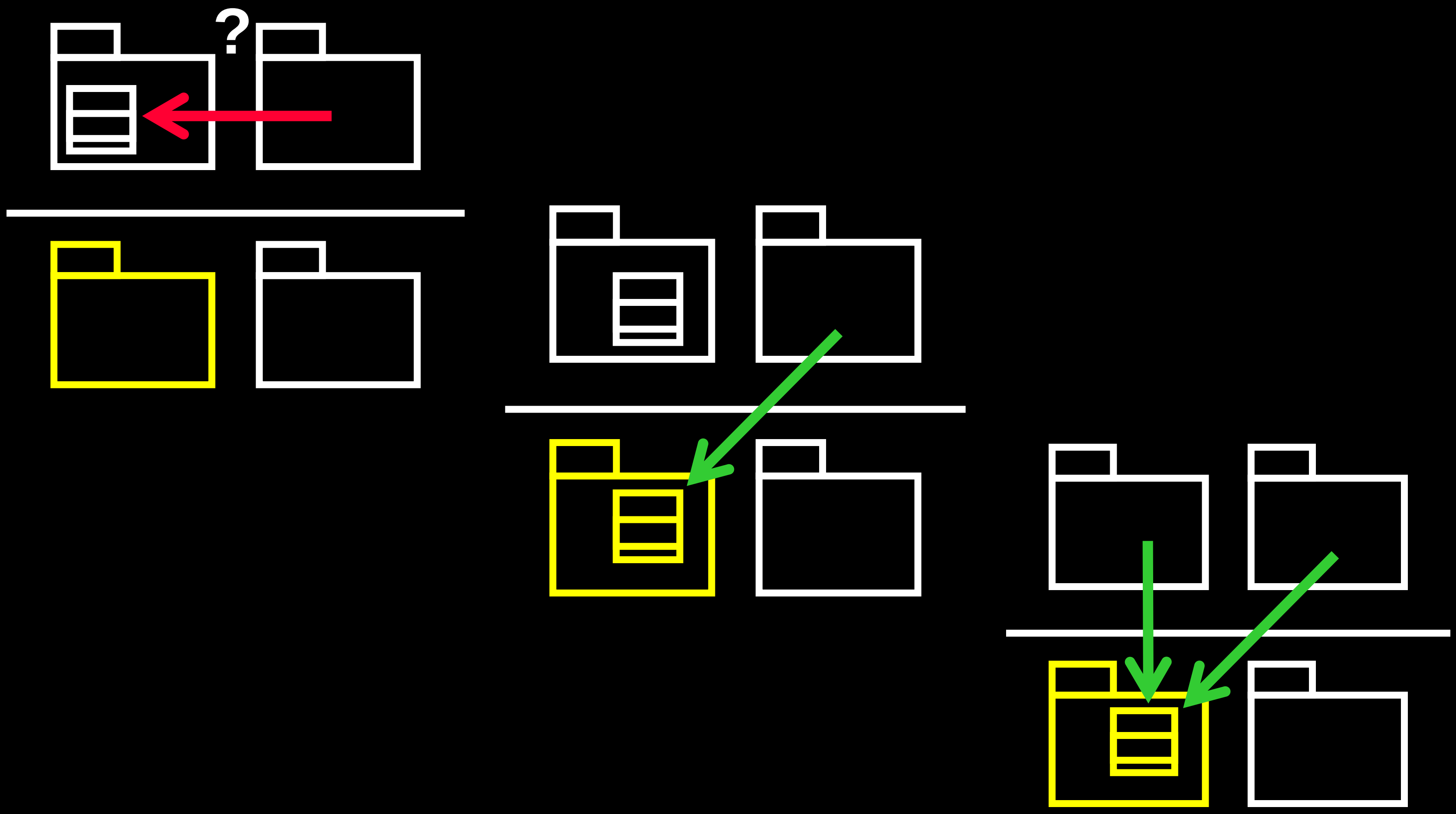
- Purpose
  - To identify where existing subsystems and/or components can be reused based on their interfaces.
- Steps
  - Look for similar interfaces
  - Modify new interfaces to improve the fit
  - Replace candidate interfaces with existing interfaces
  - Map the candidate subsystem to existing components

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

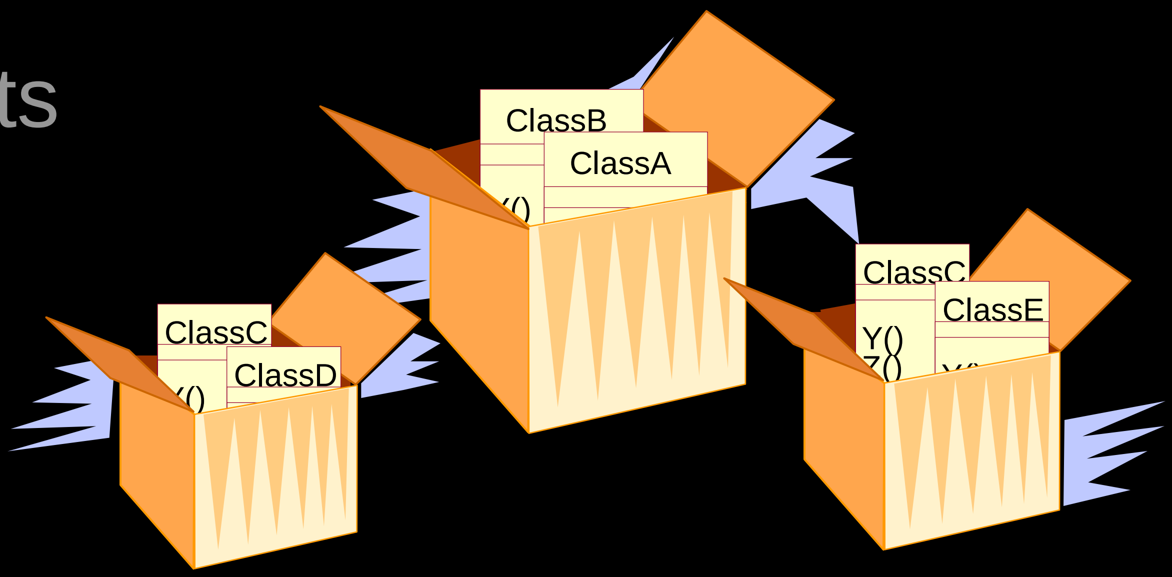


Reuse Opportunities Internal to System



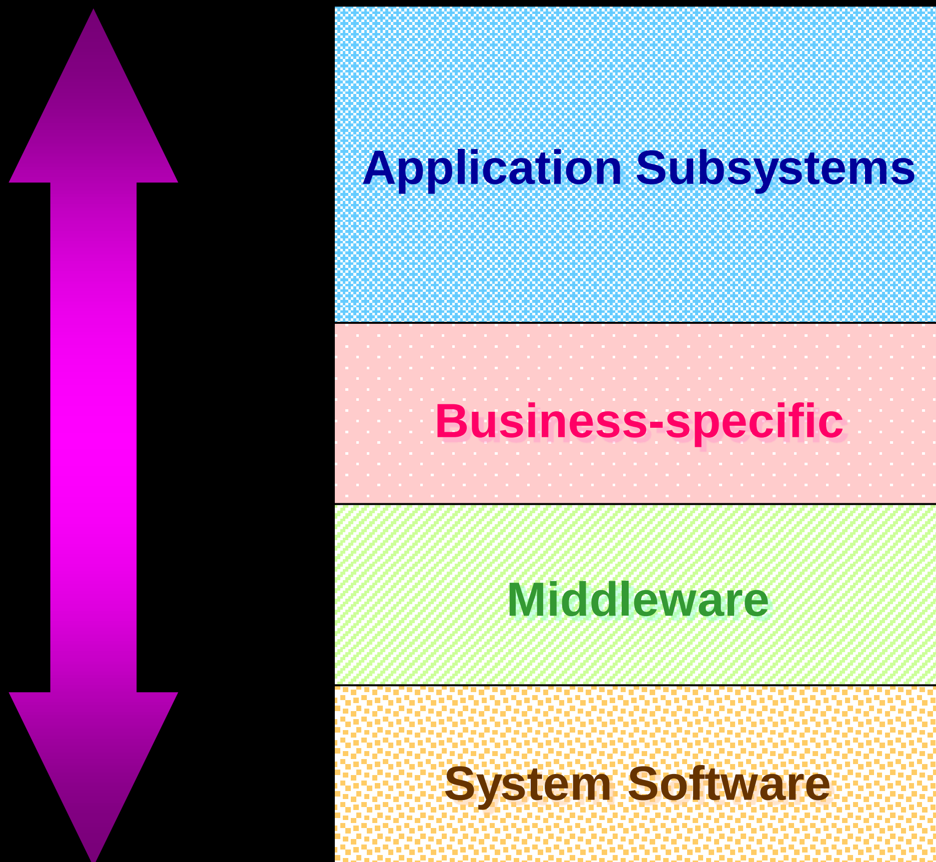
Identify Design Elements Steps

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



Review: Typical Layering Approach

Specific functionality



General functionality

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.

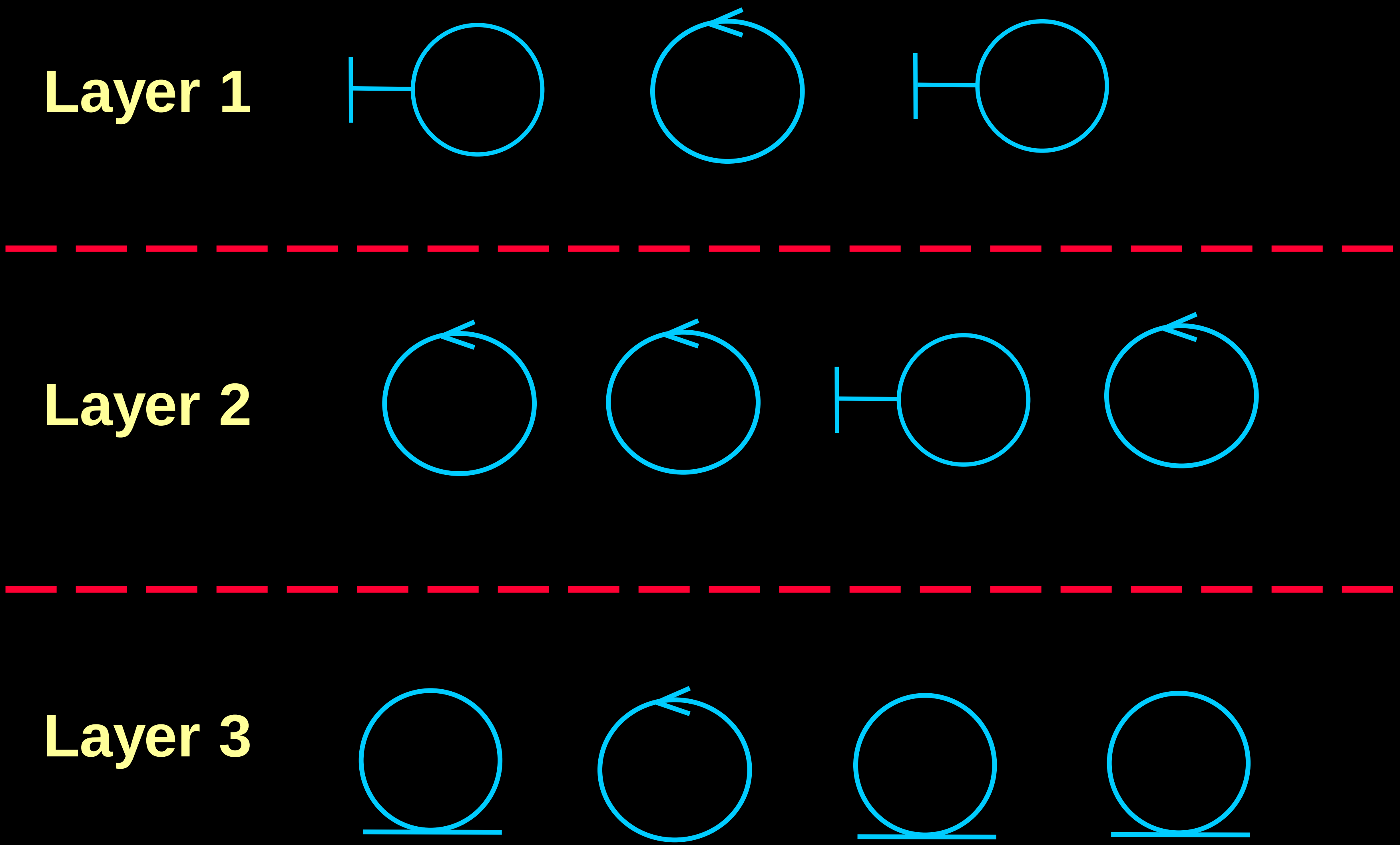
System software – contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

Layering Considerations

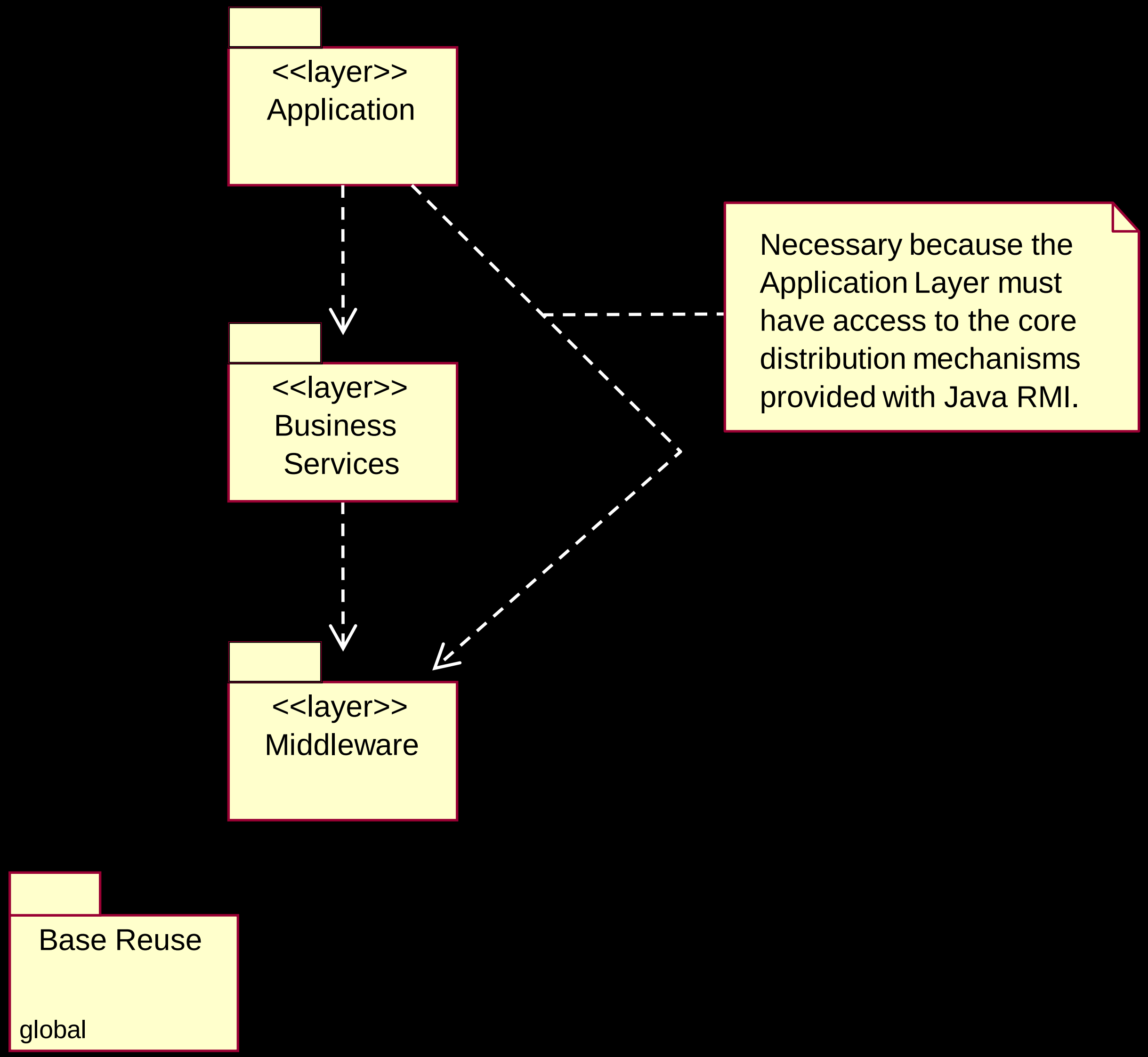
- Visibility
  - Dependencies only within current layer and below
- Volatility
  - Upper layers affected by requirements changes
  - Lower layers affected by environment changes
- Generality
  - More abstract model elements in lower layers
- Number of layers
  - Small system: 3-4 layers
  - Complex system: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

Goal is to reduce coupling and to ease maintenance effort.

Design Elements and the Architecture



Example: Architectural Layers

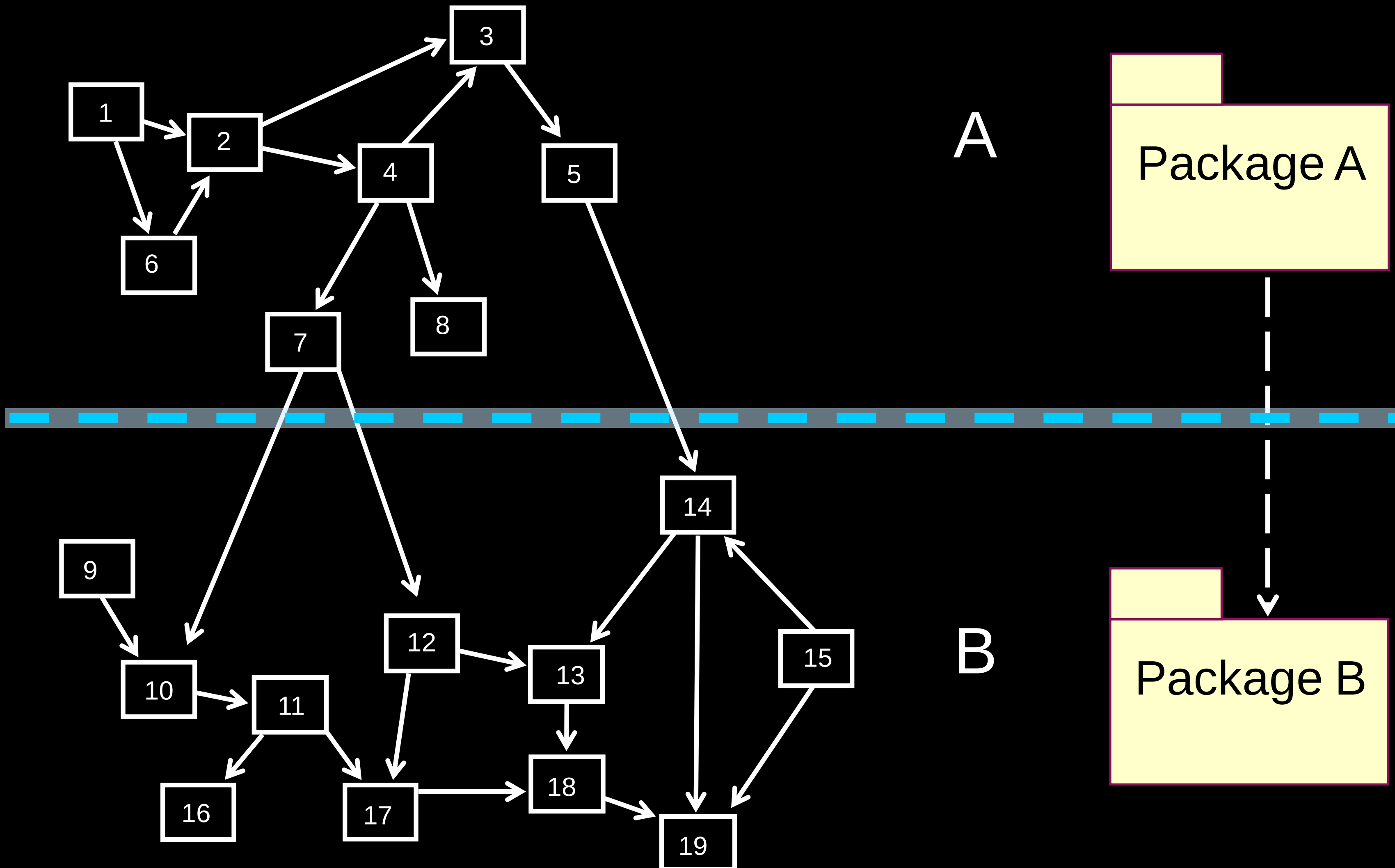


Partitioning Considerations

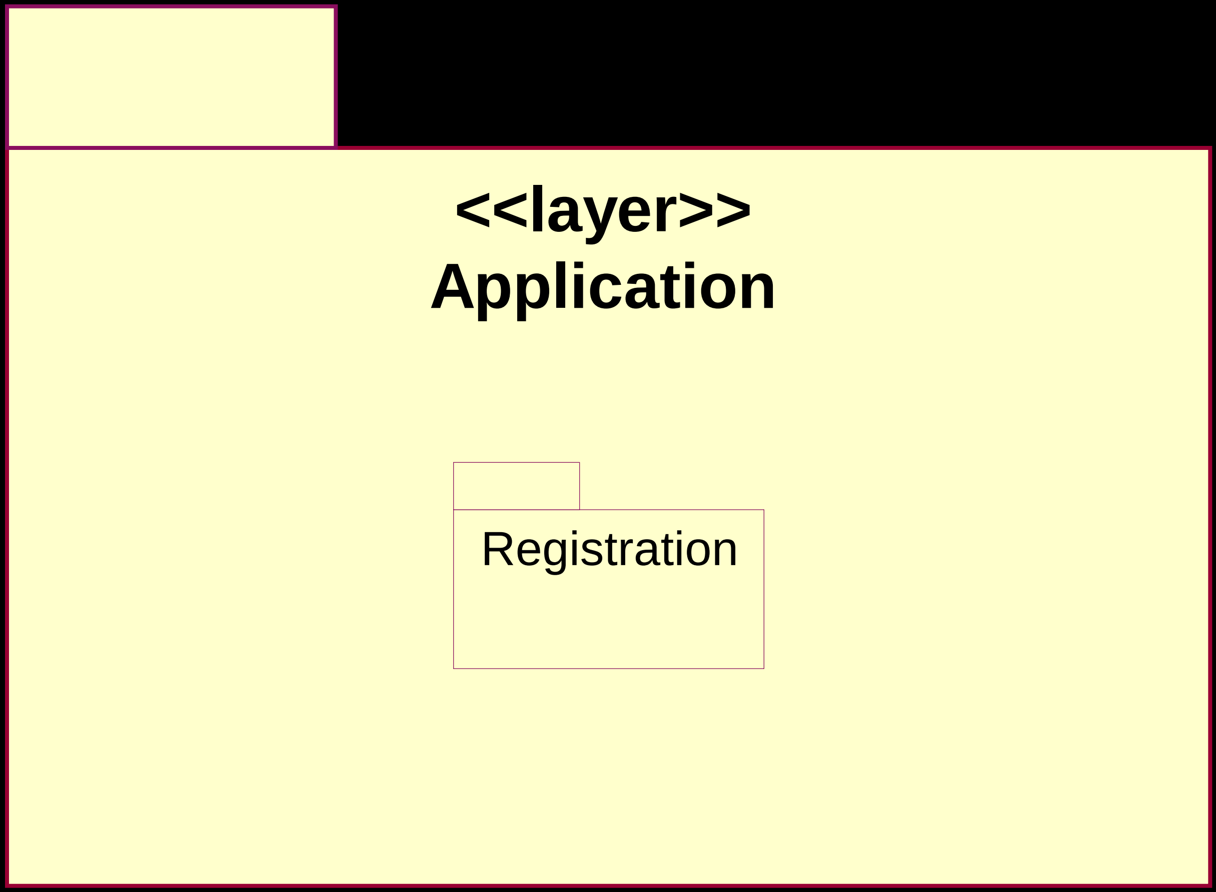
- Coupling and cohesion
- User organization
- Competency and/or skill areas
- System distribution
- Secrecy
- Variability

Try to avoid cyclic dependencies.

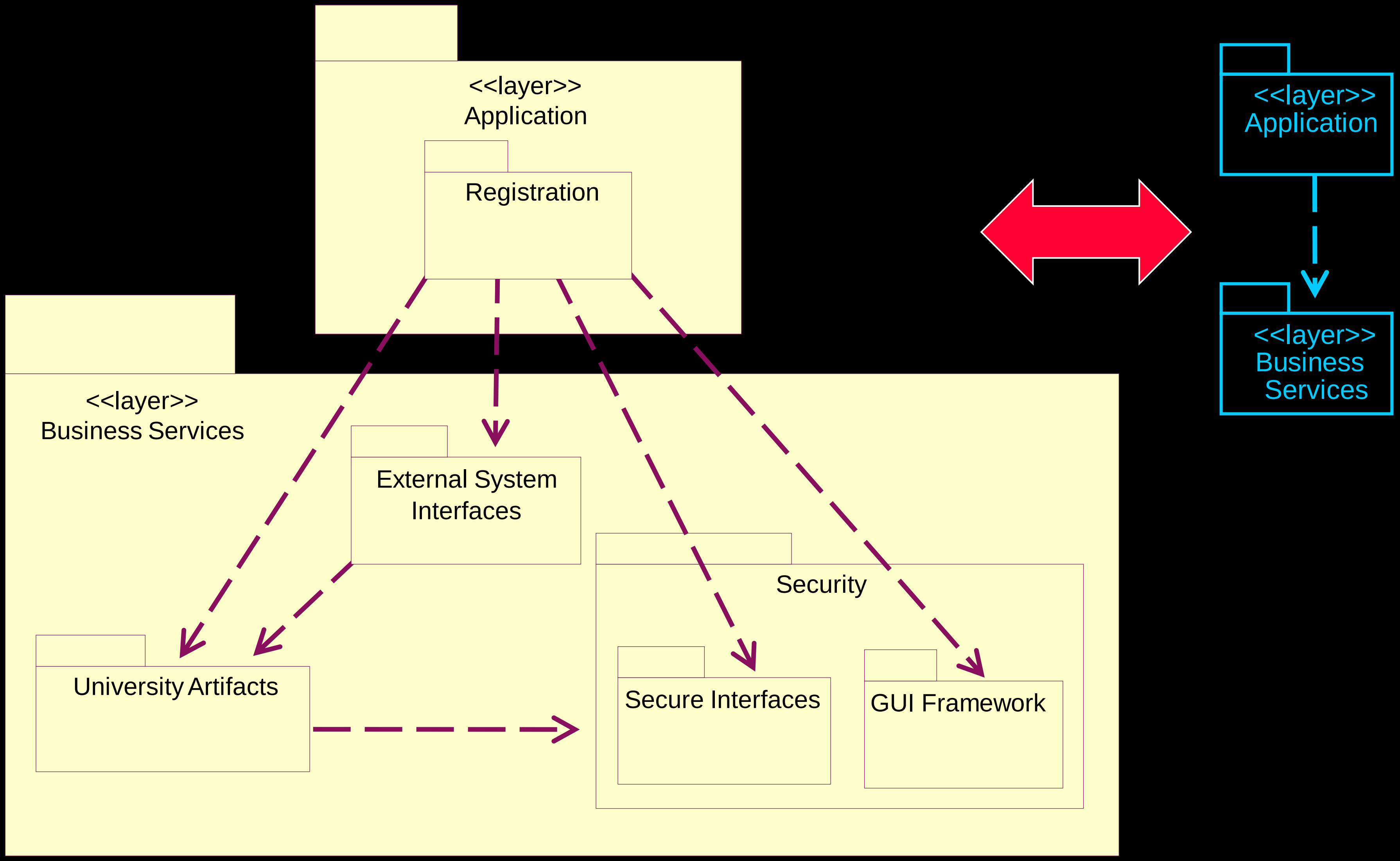
Example: Partitioning



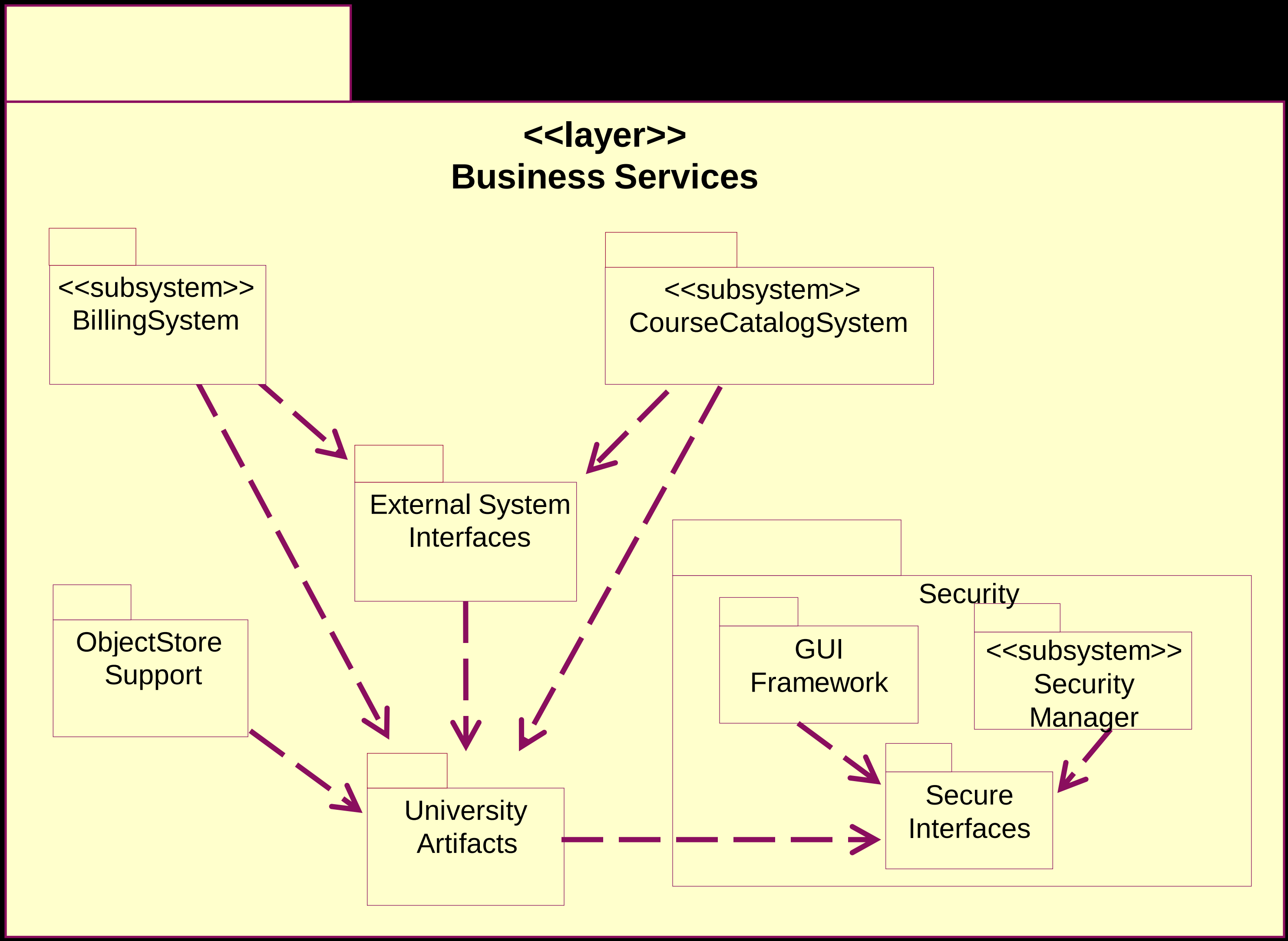
Example: Application Layer



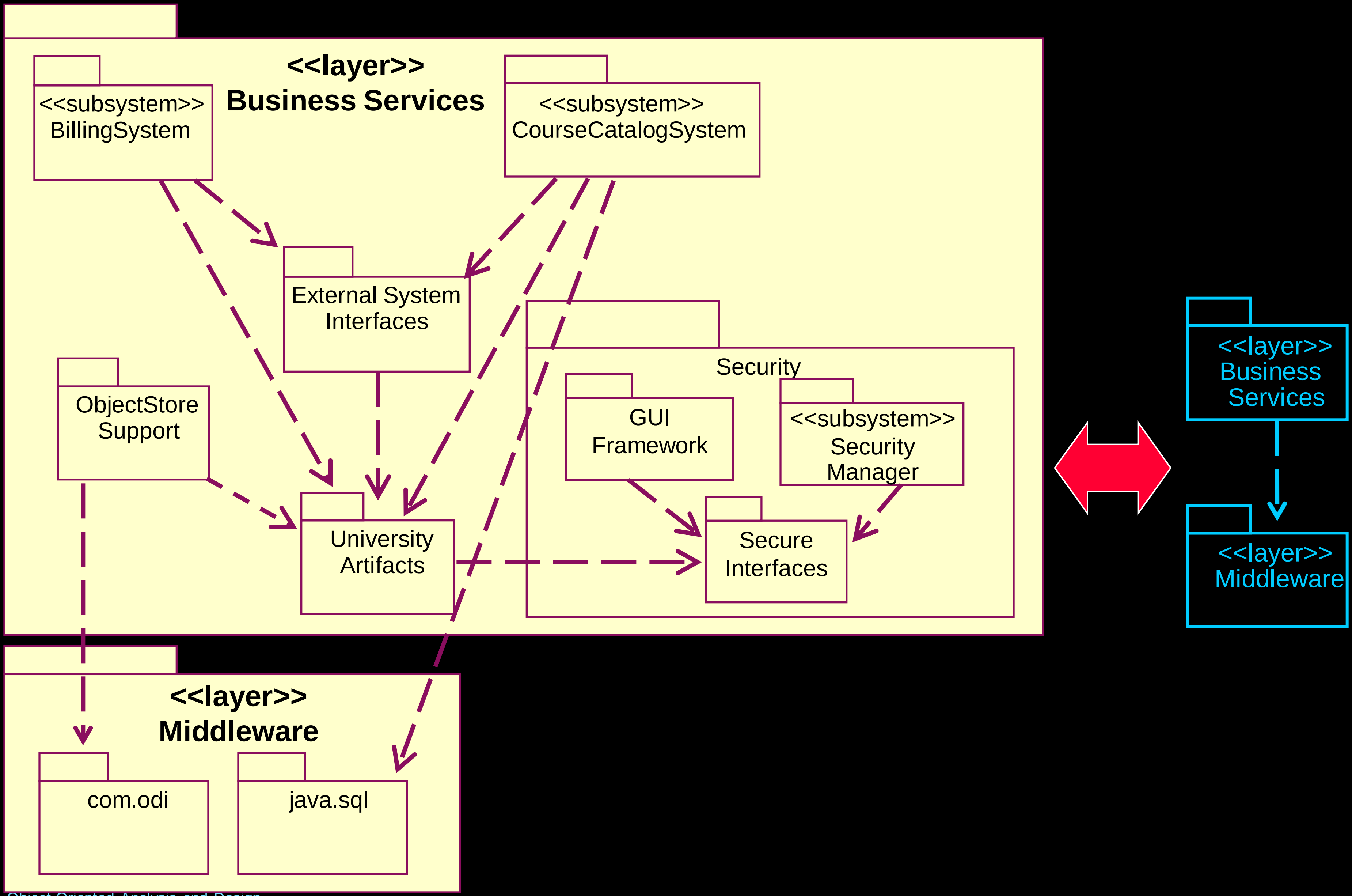
Example: Application Layer Context



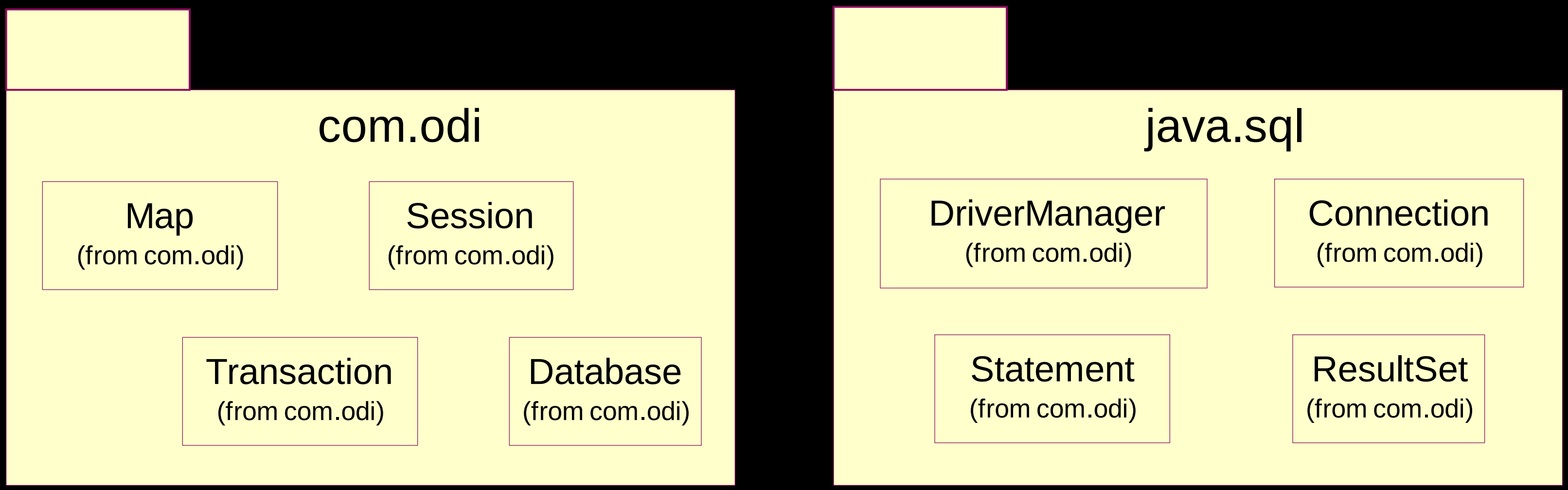
Example: Business Services Layer



Example: Business Services Layer Context



Example: Middleware Layer



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?

(continued)

Checkpoints (cont.)

- 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?

(continued)

Checkpoints (cont.)

- 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?

Exercise: Identify Design Elements

- Given the following:
  - The analysis classes and their relationships
  - The layers, packages, and their dependencies

(continued)

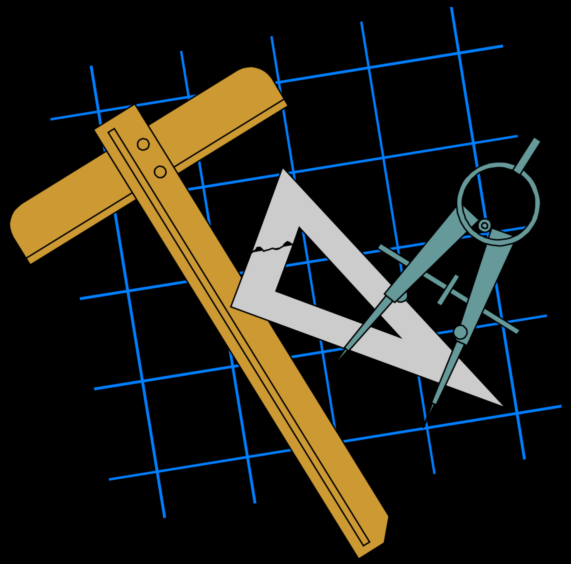
Exercise: Identify Design Elements (cont.)

- Identify the following:
  - Design classes, subsystems, their interfaces and their relationships with other design elements
  - Mapping from the analysis classes to the design elements
  - The location of the design elements (e.g. subsystems and their design classes) in the architecture (i.e., the package/layer that contains the design element)

(continued)

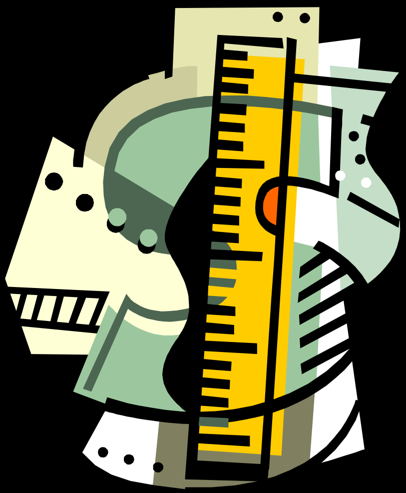
Exercise: Identify Design Elements

- Produce the following:
  - For each subsystem, an interface realization class diagram
  - Table mapping analysis classes to design elements
  - Table listing design elements and their “owning” package



Exercise: Review

- Compare your results with the rest of the class
  - What subsystem did you find? Is it partitioned logically? Does it realize an interface(s)? What analysis classes does it map to?
  - Do the package dependencies correspond to the relationships between the contained classes? Are the classes grouped logically?
  - Are there classes or collaborations of classes within a package that can be separated into an independent package?



Payroll System