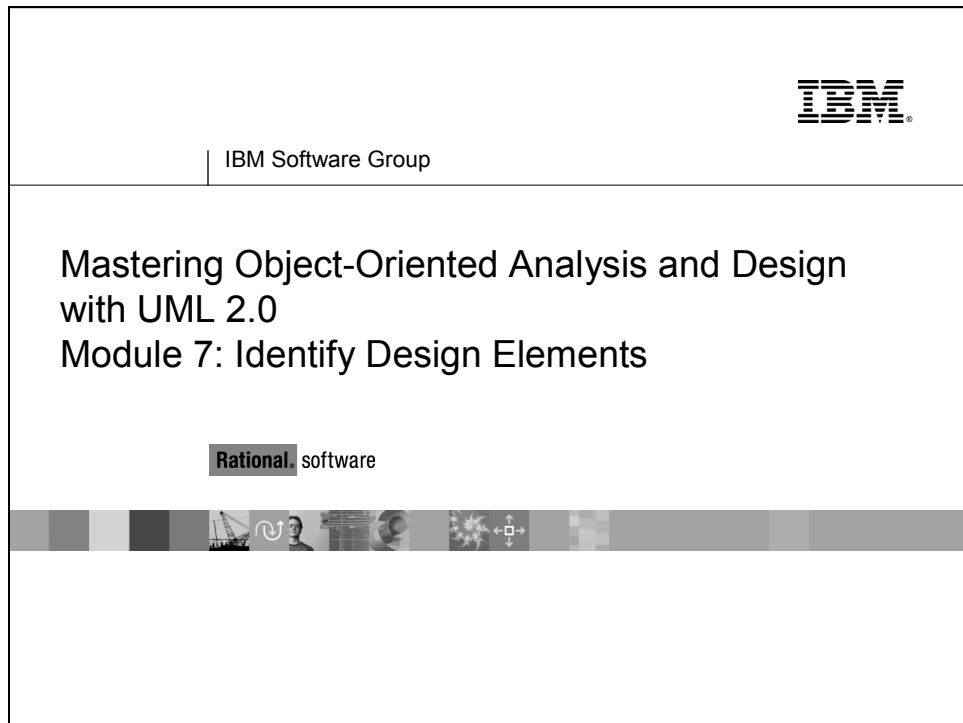


## ► ► ► Module 7 Identify Design Elements



### Topics

---

Identify Design Elements Overview.....	7-4
Identifying Design Classes.....	7-8
Identifying Interfaces .....	7-28
Identify Reuse Opportunities .....	7-36
Layering Considerations.....	7-41
Review.....	7-55

## Objectives: 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

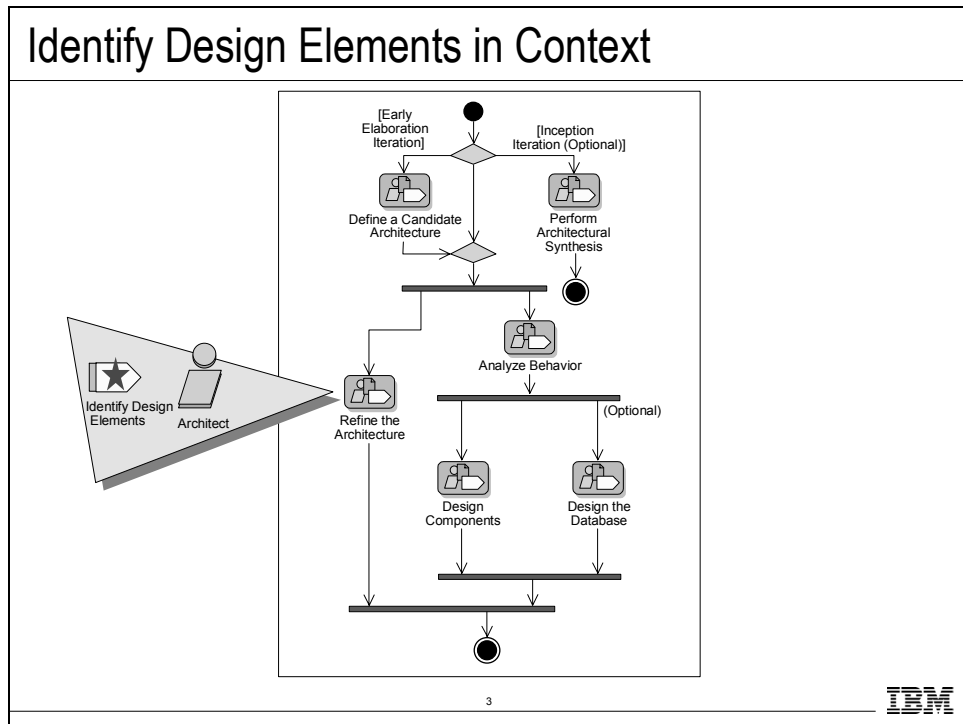
2



In this module, we will describe WHAT is performed in **Identify Design Elements**, but will not describe HOW to do it. Such a discussion is the purpose of an architecture course, which this course is not.

Understanding the rationale and considerations that support the architectural decisions is needed in order to understand the architecture, which is the framework in which designs must be developed.

## Identify Design Elements in Context



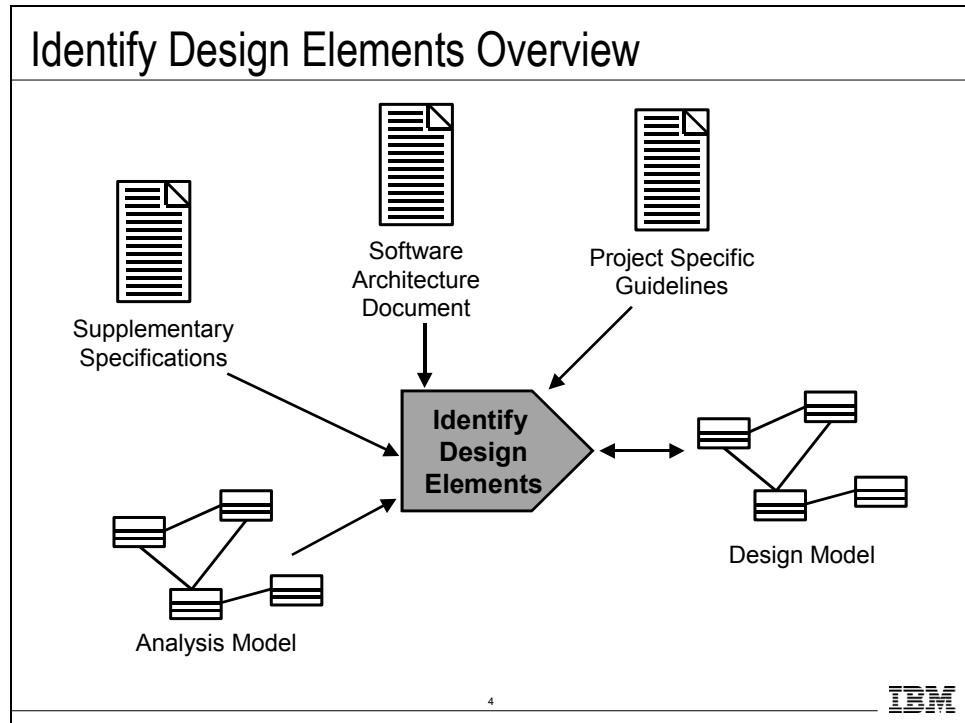
As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. **Identify Design Elements** is an activity in the Refine the Architecture workflow detail.

In Architectural Analysis, an initial attempt was made to define the layers of our system, concentrating on the upper layers. In Use-Case Analysis, you analyzed your requirements and allocated the responsibilities to analysis classes.

In **Identify Design Elements**, the analysis classes are refined into design elements (design classes and subsystems).

In Use-Case Analysis, you were concerned with the “what.” In the architecture activities, you are concerned with the “how” (for example, Design). Architecture is about making choices.

## Identify Design Elements Overview



The architect performs **Identify Design Elements**, once per iteration.

### Purpose

- To analyze interactions of analysis classes to identify Design Model elements

### Input Artifacts

- Supplementary Specifications
- Project Specific Guidelines
- Software Architecture Document
- Analysis Classes
- Analysis Model
- Design Model

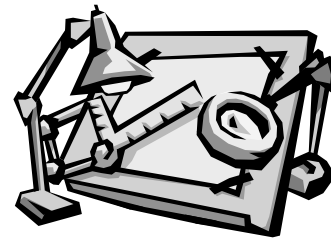
### Resulting Artifacts

- Design Model elements
  - Classes
  - Packages
  - Subsystems

## Identify Design Elements Steps

### Identify Design Elements Steps

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

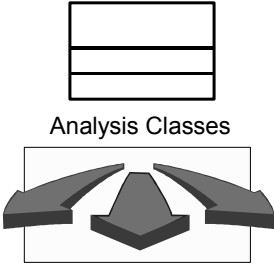


5

IBM

The above are the topics we will be discussing within the **Identify Design Elements** module. Unlike the designer activity modules, we will not be discussing each step of the activity, as the objective of this module is to understand the important **Identify Design Elements** concepts, not to learn HOW to create an architecture.

## Identify Design Elements Steps

Identify Design Elements Steps	
☆ ♦ Identify classes and subsystems	
♦ Identify subsystem interfaces	
♦ Identify reuse opportunities	
♦ Update the organization of the Design Model	
♦ Checkpoints	

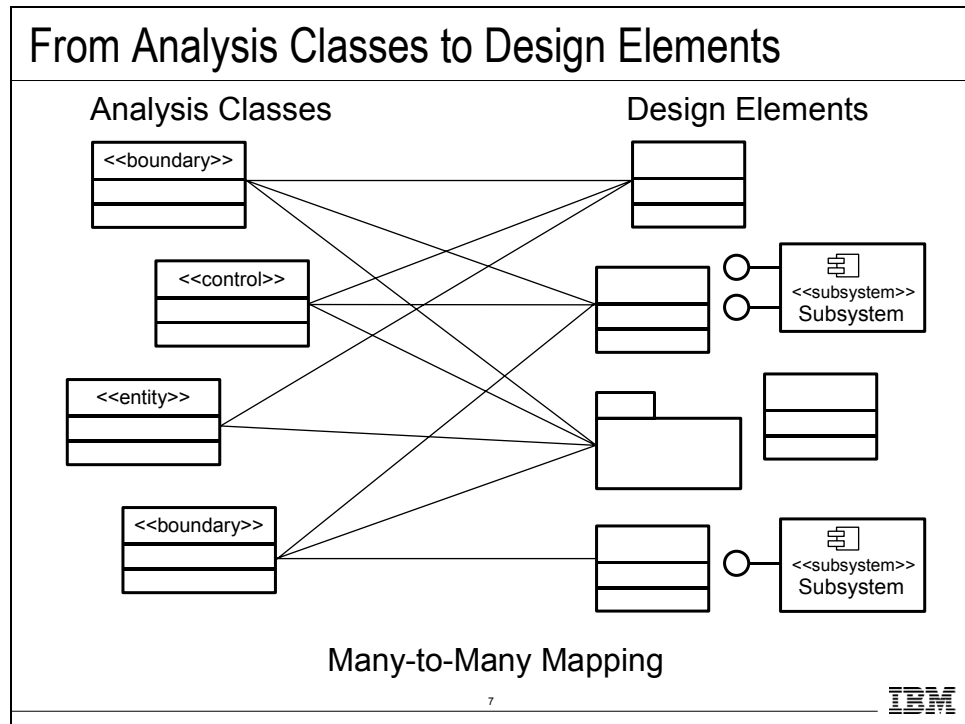
6

**IBM**

The purpose of Identify Classes and Subsystems is to refine the analysis classes into appropriate Design Model elements.

Remember, analysis classes will seldom retain their same structure through Design. Analysis classes may be expanded, collapsed, combined, or even deleted in Design.

## From Analysis Classes to Design Elements



**Identify Design Elements** is where the analysis classes identified during Use-Case Analysis are refined into design elements (for example, classes or subsystems). Analysis classes handle primarily functional requirements, and model objects from the "problem" domain; design elements handle nonfunctional requirements, and model objects from the "solution" domain.

It is in **Identify Design Elements** that you decide which analysis "classes" are really classes, which are subsystems (which must be further decomposed), and which are existing components and do not need to be "designed" at all.

Once the design classes and subsystems have been created, each must be given a name and a short description. The responsibilities of the original analysis classes should be transferred to the newly created subsystems. In addition, the identified design mechanisms should be linked to design elements.

## Identifying Design Classes

### 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 ...



8



If the analysis class is simple and already represents a single logical abstraction, then it can be directly mapped, one-to-one, to a design class. Typically, entity classes survive relatively intact into Design.

Throughout the design activities, analysis classes are refined into design elements (for example, design classes, packages, and subsystems). Some analysis classes may be split, joined, removed, or otherwise manipulated. In general, there is a many-to-many mapping between analysis classes and design elements. The possible mappings include the following.

- An analysis class can become:
  - One single class in the Design Model.
  - A part of a class in the Design Model.
  - An aggregate class in the Design Model (meaning that the parts in this aggregate may not be explicitly modeled in the Analysis Model.)
  - A group of classes that inherits from the same class in the Design Model.
  - A group of functionally related classes in the Design Model (for example, a package).
  - A subsystem in the Design Model.
  - A relationship in the Design Model.
- A relationship between analysis classes can become a class in the Design Model.
- Part of an analysis class can be realized by hardware, and not modeled in the Design Model at all.
- Any combination of the above.

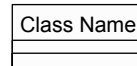


## Review: Class and Package

### Review: Class and Package

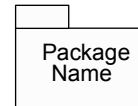
- ◆ What is a class?

- 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



9



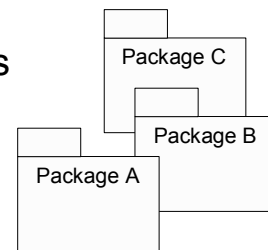
A **package** is a general purpose mechanism for organizing elements into groups. They provide the ability to organize the model under development. A package is represented as a tabbed folder.

Later in this module, we will contrast “vanilla” packages, as defined above, with subsystems.

## Group Design Classes in Packages

### 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



10



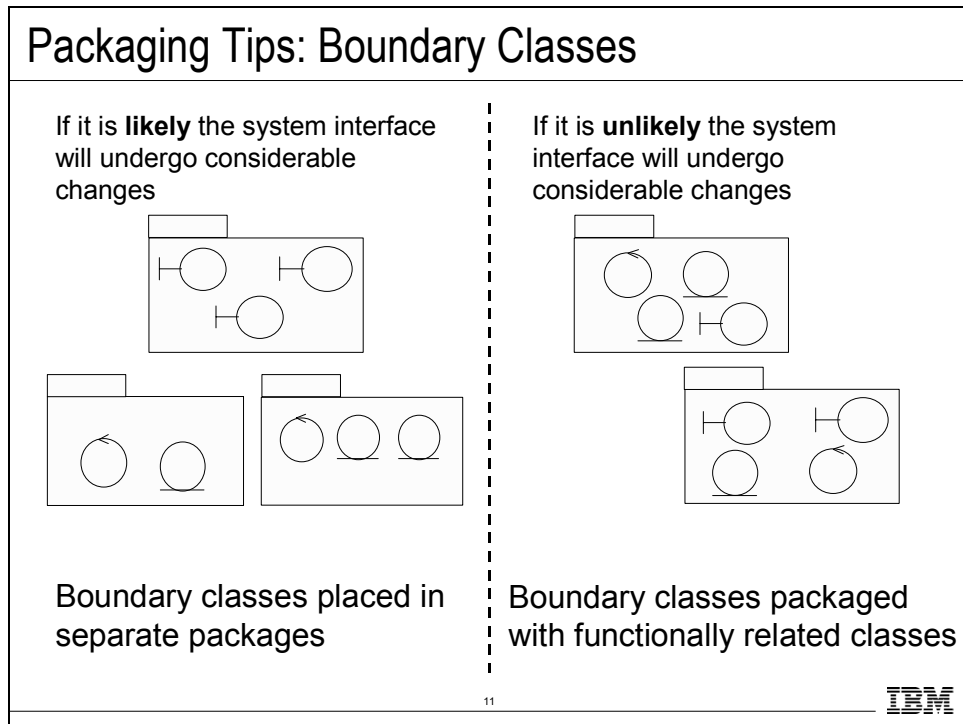
When identifying classes, you should group them into packages, for organizational and configuration management purposes.

The Design Model can be structured into smaller units to make it easier to understand. By grouping Design Model elements into packages and subsystems, then showing how those groupings relate to one another, it is easier to understand the overall structure of the model.

You might want to partition the Design Model for a number of reasons:

- You can use packages and subsystems as order, configuration, or delivery units when a system is finished.
- Allocation of resources and the competence of different development teams might require that the project be divided among different groups at different sites.
- Subsystems can be used to structure the Design Model in a way that reflects the user types. Many change requirements originate from users; subsystems ensure that changes from a particular user type will affect only the parts of the system that correspond to that user type.
- Subsystems are used to represent the existing products and services that the system uses.

## Packaging Tips: Boundary Classes



When the boundary classes are distributed to packages, there are two different strategies that can be applied. Which one to choose depends on whether or not the system interfaces are likely to change greatly in the future.

- If it is *likely* that the system interface will be replaced, or undergo considerable changes, the interface should be separated from the rest of the Design Model. When the user interface is changed, only these packages are affected. An example of such a major change is the switch from a line-oriented interface to a window-oriented interface.
- If no major interface changes are planned, changes to the system services should be the guiding principle, rather than changes to the interface. The boundary classes should then be placed together with the entity and control classes with which they are functionally related. This way, it will be easy to see what boundary classes are affected if a certain entity or control class is changed.

Mandatory boundary classes that are not functionally related to any entity or control classes, should be placed in separate packages, together with boundary classes that belong to the same interface.

If a boundary class is related to an optional service, group it in a separate subsystem with the classes that collaborate to provide the service. The subsystem will map onto an optional component that will be provided when the optional functionality is ordered.

## Packaging Tips: 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

12



A package should be identified for each group of classes that are functionally related. There are several practical criteria that can be applied when judging if two classes are functionally related. These are, in order of diminishing importance:

- If changes in one class' behavior and/or structure necessitate changes in another class, the two classes are functionally related.
- It is possible to find out if one class is functionally related to another by beginning with a class — for example, an entity class — and examining the impact of it being removed from the system. Any classes that become superfluous as a result of a class removal are somehow connected to the removed class. By superfluous, we mean that the class is only used by the removed class, or is itself dependent upon the removed class.
- Two objects can be functionally related if they interact with a large number of messages, or have an otherwise complicated 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 can be functionally related if they interact with, or are affected by changes in, the same actor. If two classes do not involve the same actor, they should not lie in the same package. The last rule can, of course, be ignored for more important reasons.

## Packaging Tips: Functionally Related Classes (continued)

### Packaging Tips: Functionally Related Classes (continued)

- ♦ 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

13

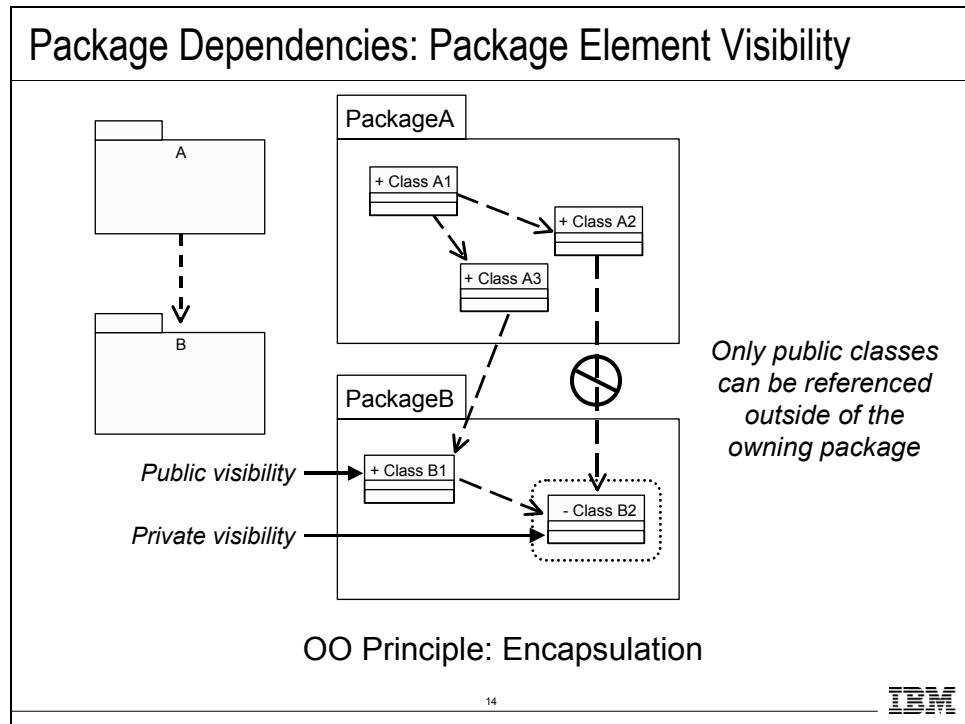


- Two classes can be functionally related if they have relationships between each other (associations, aggregations, and so on). Of course, this criterion cannot be followed mindlessly but can be used when no other criterion is applicable.
- A class can be functionally related to the class that creates instances of it.

These two criteria determine 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



In Architectural Analysis, we discussed package dependencies. Now let's look at package dependencies in more detail and see how visibility can be defined.

Visibility can be defined for package elements the same way it is defined for class attributes and operations. This visibility allows you to specify how other packages can access the elements that are owned by the package.

The visibility of a package element can be expressed by including a visibility symbol as a prefix to the package element name.

There are three types of visibility defined in the UML:

**Public:** Public classes can be accessed outside of the owning package. Visibility symbol: +.

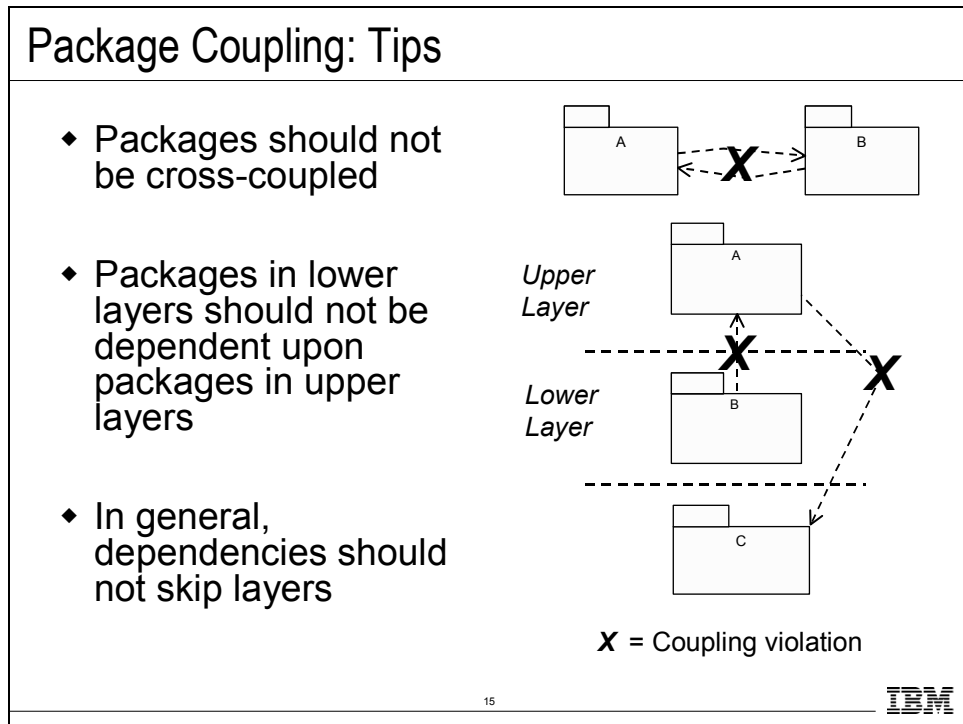
**Protected:** Protected classes can only be accessed by the owning package and any packages that inherit from the owning package. Visibility symbol: #.

**Private:** Private classes can only be accessed by classes within the owning package.  
Visibility symbol: -.

The public elements of a package constitute the package's interface. All dependencies on a package should be dependencies on public elements of the package.

Package visibility provides support for the OO principle of encapsulation.

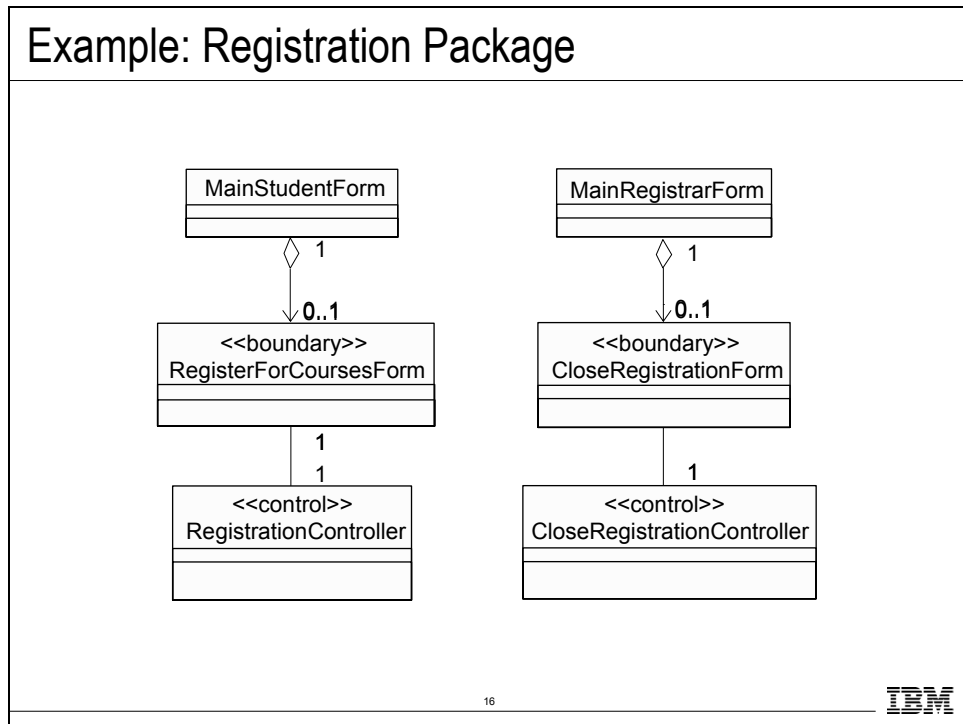
## Package Coupling: Tips



Package coupling is good and bad: Good, because coupling represents re-use, and bad, because coupling represents dependencies that make the system harder to change and evolve. Some general principles can be followed:

- Packages should not be cross-coupled (that is, co-dependent); for example, two packages should not be dependent on one another. In these cases, the packages need to be reorganized to remove the cross-dependencies.
- Packages in lower layers should not be dependent upon packages in upper layers. Packages should only be dependent upon packages in the same layer and in the next lower layer. In these cases, the functionality needs to be repartitioned. One solution is to state the dependencies in terms of interfaces, and organize the interfaces in the lower layer.
- In general, dependencies should not skip layers, unless the dependent behavior is common across all layers, and the alternative is to simply pass through operation invocations across layers.
- Packages should not depend on subsystems — only on other packages or on interfaces.

## Example: Registration Package



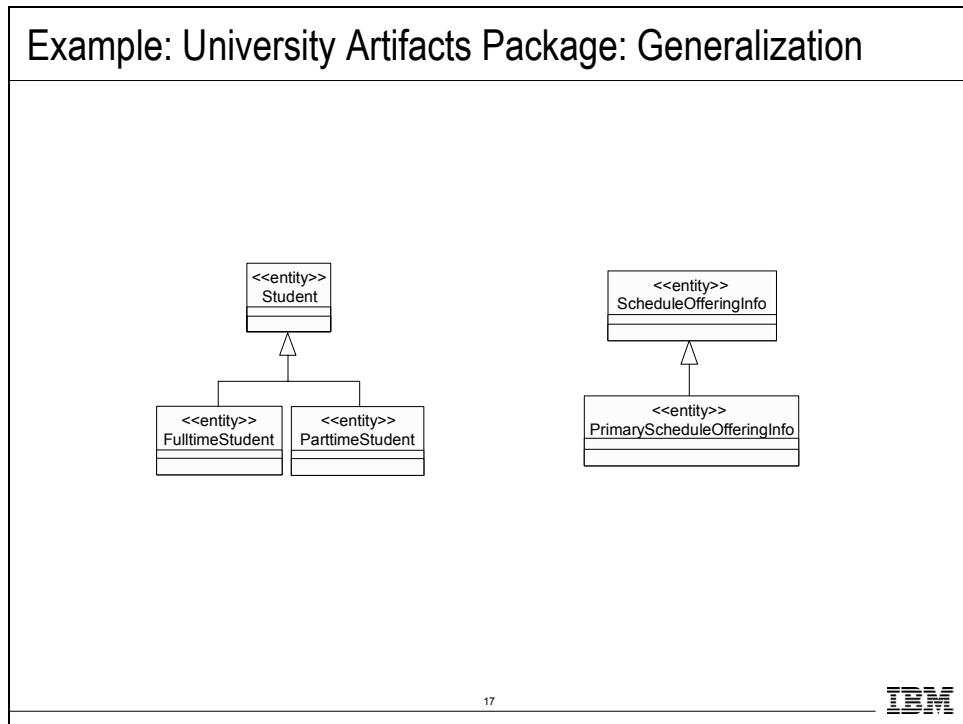
The next few slides describe the packaging decisions for the Course Registration System.

All classes specifically supporting registration were partitioned into the Registration package.

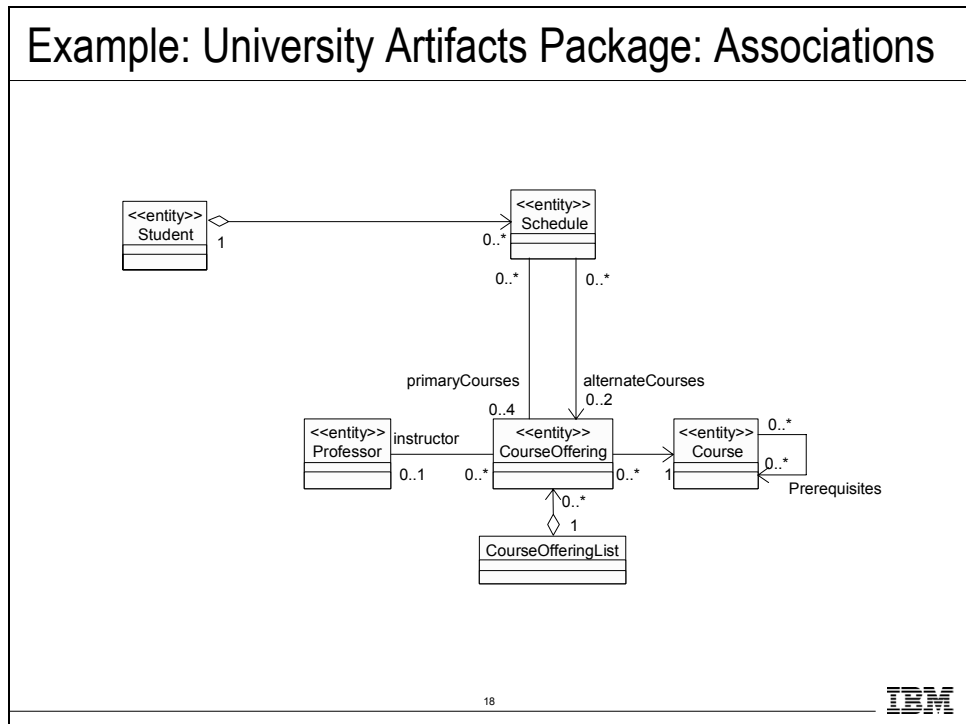
For simplicity on the above diagram, only the Student Registration package classes have been shown, and the operations and attributes are not displayed.



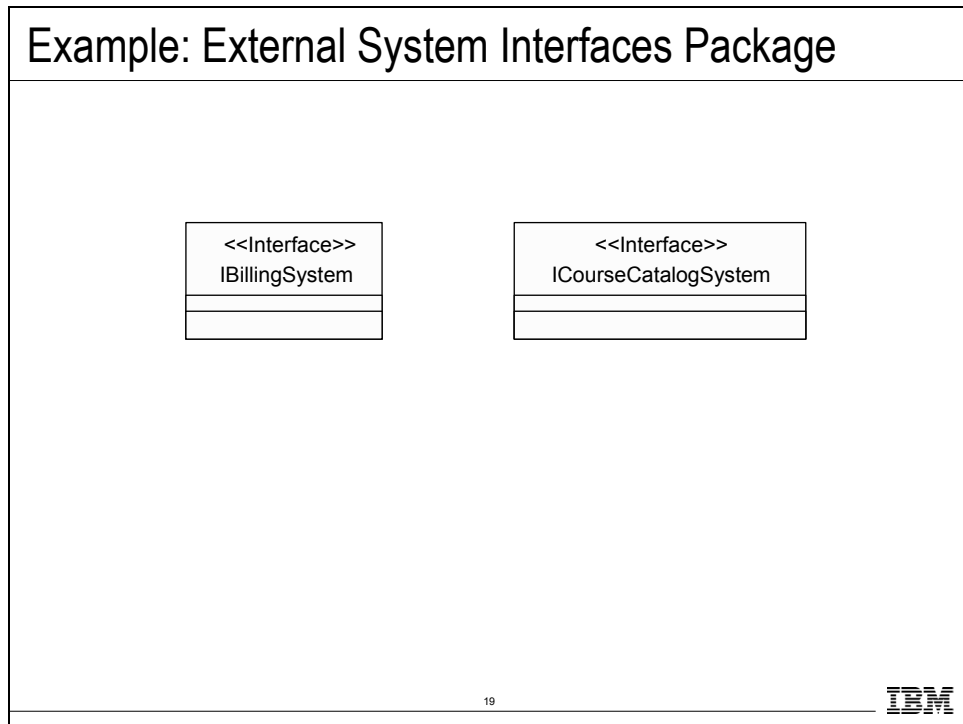
## Example: University Artifacts Package: Generalization



## Example: University Artifacts Package: Associations

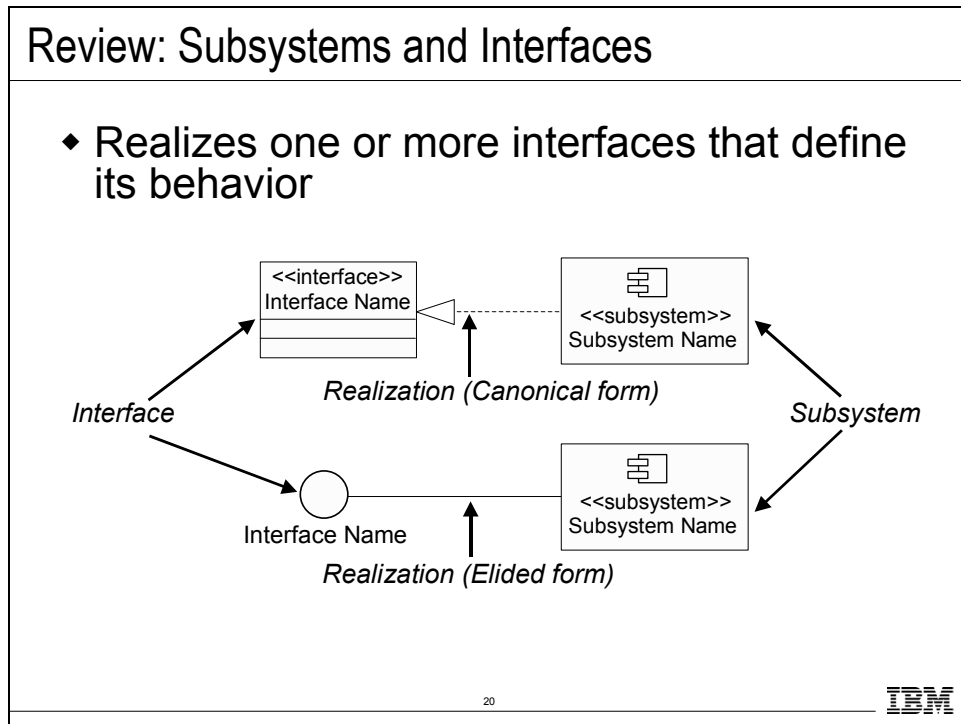


## Example: External System Interfaces Package



The external system access classes were partitioned into the External System Interfaces package. This is so that the external system interface classes can be configuration-managed independently from the subsystems that realize them. For simplicity, only the External System Interfaces package classes have been shown on the above diagram. The operations and attributes are not displayed.

## Review: Subsystems and Interfaces



A subsystem is a model element that has the semantics of a package, such that it can contain other model elements, and a class, such that it has behavior. A subsystem realizes one or more interfaces, which define the behavior it can perform.

A subsystem can be represented as a UML package (that is, a tabbed folder) with the «subsystem» stereotype.

An interface is a model element that defines a set of behaviors (a set of operations) offered by a classifier model element (specifically, a class, subsystem, or component). The relationship between interfaces and classifiers (subsystems) is not always one-to-one. An interface can be realized by multiple classifiers, and a classifier can realize multiple interfaces.

Realization is a semantic relationship between two classifiers. One classifier serves as the contract that the other classifier agrees to carry out.

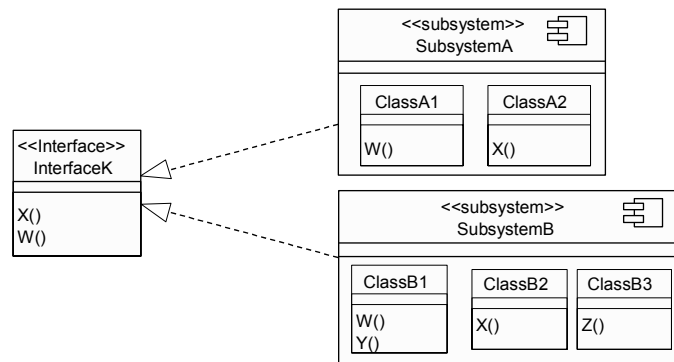
The realization relationship can be modeled as a dashed line with a hollow arrowhead pointing at the contract classifier (canonical form), or when combined with an interface, as a “ball” (elided form). Thus, in the above example, the two interface/subsystem pairs with the relation between them are synonymous.

Interfaces are a natural evolution from the public classes of a package (described on the previous slide) to abstractions outside the subsystem. Interfaces are pulled out of the subsystem like a kind of antenna, through which the subsystem can receive signals. All classes inside the subsystem are then private and not accessible from the outside.

## Subsystems and Interfaces (continued)

### Subsystems and Interfaces (continued)

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



21



A subsystem encapsulates its implementation behind one (or more) interfaces. Interfaces isolate the rest of the architecture from the details of the implementation. Operations defined for the interface are implemented by one or more elements contained within the subsystem.

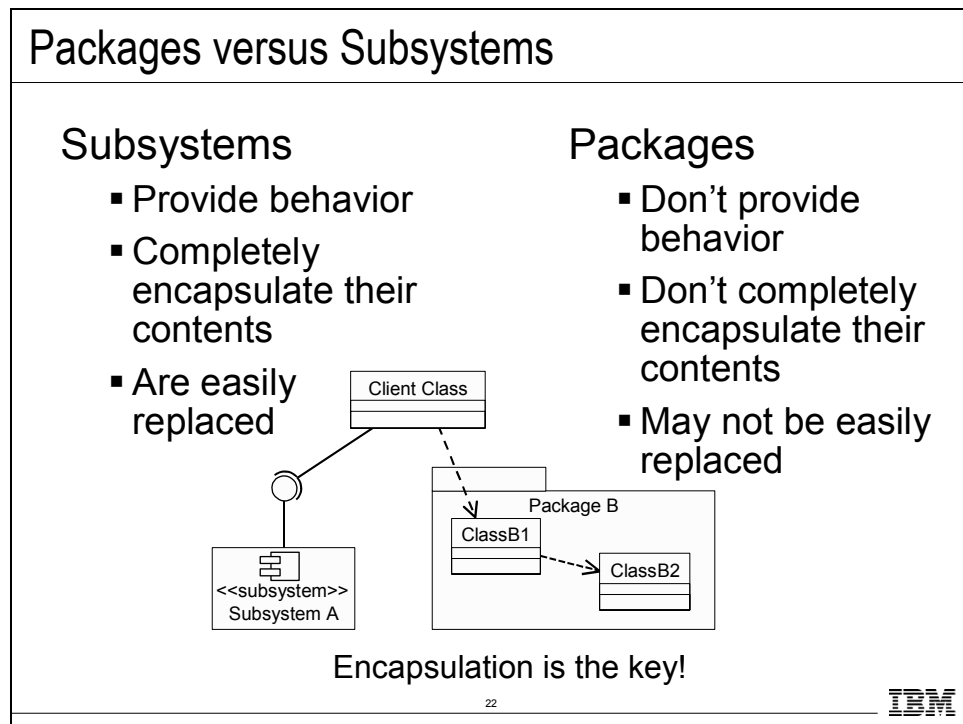
An interface is a pure specification. Interfaces provide the “family of behavior” that a classifier that implements the interface must support. Interfaces are separate things that have separate life spans from the elements that realize them. This separation of interface and implementation exemplifies the OO concepts of modularity and encapsulation, as well as polymorphism.

Note: Interfaces are not abstract classes. Abstract classes allow you to provide default behavior for some or all of their methods. Interfaces provide no default behavior.

As mentioned earlier, an interface can be realized by one or more subsystems. Any two subsystems that realize the same interfaces can be substituted for one another. The benefit of this is that, unlike a package, the contents and internal behaviors of a subsystem can change with complete freedom, so long as the subsystem's interfaces remain constant.

In the above example, InterfaceK defines the operations X() and Y(). Both SubsystemA and SubsystemB realize InterfaceK, which means that they provide the implementation for operations X() and Y(). Thus, SubsystemA and SubsystemB are completely “plug-and-playable” (that is, one can be replaced by the other without any impacts on clients of the subsystems).

## Packages versus Subsystems



Subsystems and packages are very alike, but are different in some essential ways. A subsystem provides interfaces by which the behavior it contains can be accessed. Packages provide no behavior; they are simply containers of things that have behavior. Packages help organize and control sets of classes that are needed in common, but which are not really subsystems. Packages are just used for model organization and configuration management.

Subsystems completely encapsulate their contents, providing behavior only through their interfaces. Dependencies on a subsystem are on its interface(s), not on specific subsystem contents. With packages, dependencies are on specific elements within the package.

With subsystems, the contents and internal behaviors of a subsystem can change with complete freedom as long as the subsystem's interfaces remain constant. With packages, it is impossible to substitute packages for one another unless they have the same public classes. The public classes and their public operations get frozen by the dependencies that external classes have on them. Thus, the designer is not free to eliminate these classes or change their behaviors if a better idea presents itself.

Note: Even when using packages, it is important that you hide the implementation from elements external to the package. All dependencies on a package should be on the public classes of the package. Public classes can be considered the interface of the package and should be managed as such (stabilized early).

## Subsystem Usage

### 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.

23



Subsystems provide a “replaceable design” element: Any two subsystems (or classes, for that matter) that realize the same interfaces are interchangeable.

Subsystems support multiple implementation variants. Subsystems can be used when modeling one of many implementation variants.

Subsystems can be used to represent components from the Implementation Model in the Design Model.

## Identifying Subsystems Hints

### 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.



24

IBM

**Object collaborations:** If the classes in a collaboration interact only with each other to produce a well-defined set of results, then encapsulate them within a subsystem.

**Optionality:** If collaborations model optional behavior, or features that may be removed, upgraded, or replaced with alternatives, then encapsulate them within a subsystem.

**User interface:** Create “horizontal” subsystems (boundary classes and related entity classes in separate subsystems) or “vertical” subsystems (related boundary and entity classes in the same subsystem), depending on the coupling of the user interface and entity classes.

**Actors:** Partition functionality used by two different actors, since each actor can independently change requirements.

**Class coupling and cohesion:** Organize highly coupled classes into subsystems, separating along the lines of weak coupling.

**Substitution:** Represent different service levels for a particular capability (for example, high, medium, and low availability) as a separate subsystem, that realizes the same interfaces.

**Distribution:** If particular functionality must reside on a particular node, ensure that the subsystem functionality maps onto a single node.

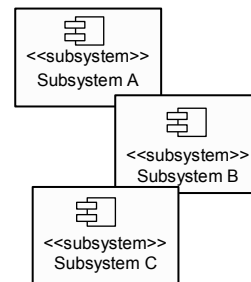
**Volatility:** You will want to encapsulate those chunks of your system that you expect to change.



## Candidate Subsystems

### 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



25



Examples of analysis classes that may evolve into subsystems include:

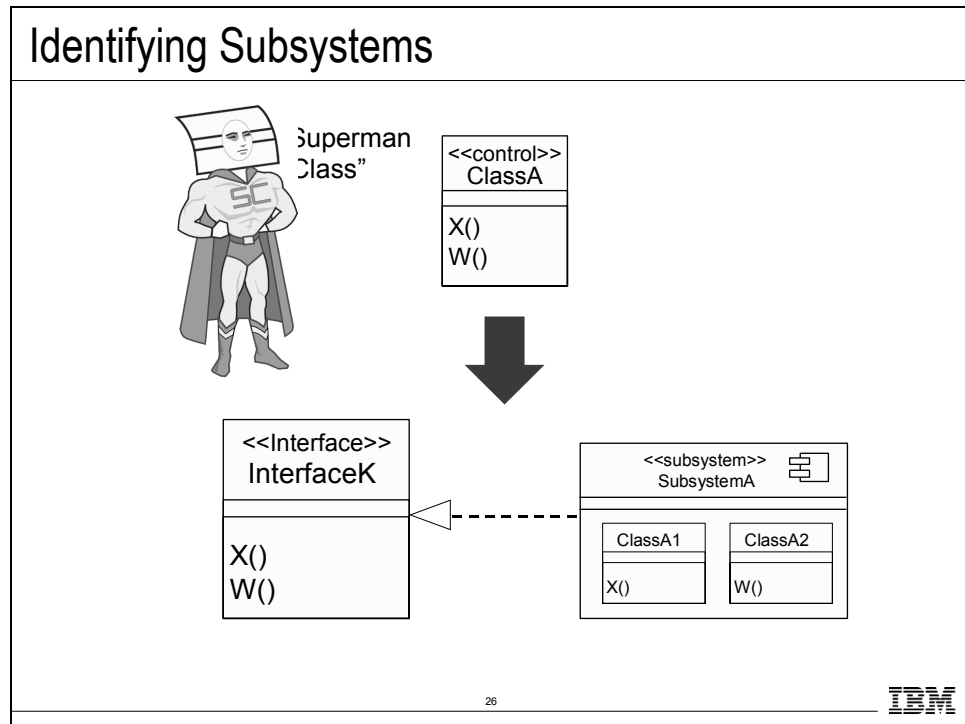
- Classes providing complex services and/or utilities. For example:
  - Credit or risk evaluation engines in financial applications
  - Rule-based evaluation engines in commercial applications
  - Security authorization services in most applications.
- Boundary classes, both for user interfaces and external system interfaces. If the interface(s) are simple and well-defined, a single class may be sufficient. Often, however, these interfaces are too complex to be represented using a single class. They often require complex collaborations of many classes. Moreover, these interfaces may be reusable across applications. As a result, a subsystem more appropriately models these interfaces in many cases.

Examples of products the system uses that you can represent by a subsystem include:

- Communication software (middle-ware, COM/CORBA support)
- Database access support (RDBMS mapping support)
- Types and data structures (stacks, lists, queues)
- Common utilities (math libraries)
- Application-specific products (billing system, scheduler)

Interfaces and subsystems provide the necessary decoupling between interface and implementation to model (in Design) what components do for Implementation.

## Identifying Subsystems



When the analysis class is complex, such that it appears to embody behaviors that cannot be the responsibility of a single class acting alone, or the responsibilities may need to be reused, the analysis class should be refined into a subsystem. This is a decision based largely on conjecture guided by experience. The actual representation may take a few iterations to stabilize.

As discussed earlier, the use of a subsystem allows the interface to be defined and stabilized, while leaving the design details of the interface implementation to remain hidden while its definition evolves.

The decision to make something a subsystem is often driven by the knowledge and experience of the architect. Since it tends to have a strong effect on the partitioning of the solution space, the decision needs to be made in the context of the whole model. It is the result of more detailed design knowledge, as well as the imposition of constraints imposed by the implementation environment.

When an analysis class is evolved into a subsystem, the responsibilities that were allocated to the "superman" analysis class are then allocated to the subsystem and an associated interface (that is, they are used to define the interface operations). The details of how that subsystem actually implements the responsibilities (that is, the interface operations) is deferred until Subsystem Design.

## Identify Design Elements Steps

### Identify Design Elements Steps

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

27



Interfaces define a set of operations that are realized by some classifier. In the Design Model, interfaces are principally used to define the interfaces for subsystems. This is not to say that they cannot be used for classes as well. But for a single class it is usually sufficient to define public operations on the class. These operators, in effect, define its “interface.”

Interfaces are important for subsystems because they allow the separation of the declaration of behavior (the interface) from the realization of behavior (the specific classes within the subsystem that realize the interface). This de-coupling provides us with a way to increase the independence of development teams working on different parts of the system, while retaining precise definitions of the “contracts” between these different parts.

## Identifying Interfaces

---

### 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.

28



Once the subsystems are identified, their interfaces need to be identified.

**Identify candidate interfaces.** Organize the subsystem responsibilities into groups of cohesive, related responsibilities. These groupings define the initial, first-cut set of interfaces for the subsystem. To start with, identify an operation for each responsibility, complete with parameters and return values.

**Look for similarities between interfaces.** Look for similar names, similar responsibilities, and similar operations. Extract common operations into a new interface. Be sure to look at existing interfaces as well, re-using them where possible.

**Define interface dependencies.** Add dependency relationships from the interface to all classes and/or interfaces that appear in the interface operation signatures.

**Map the interfaces to subsystems.** Create realization associations from the subsystem to the interface(s) it realizes.

**Define the behavior specified by the interfaces.** If the operations on the interface must be invoked in a particular order, define a state machine that illustrates the publicly visible (or inferred) states that any design element that realizes the interface must support.

**Package the interfaces.** Interfaces can be managed and controlled independently of the subsystems themselves. Partition the interfaces according to their responsibilities.

## Interface Guidelines

### 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.



29



**Interface name:** Name the interface to reflect the role it plays in the system. The name should be short — one-to-two words. It is not necessary to include the word "interface" in the name; it is implied by the type of model element (that is, interface).

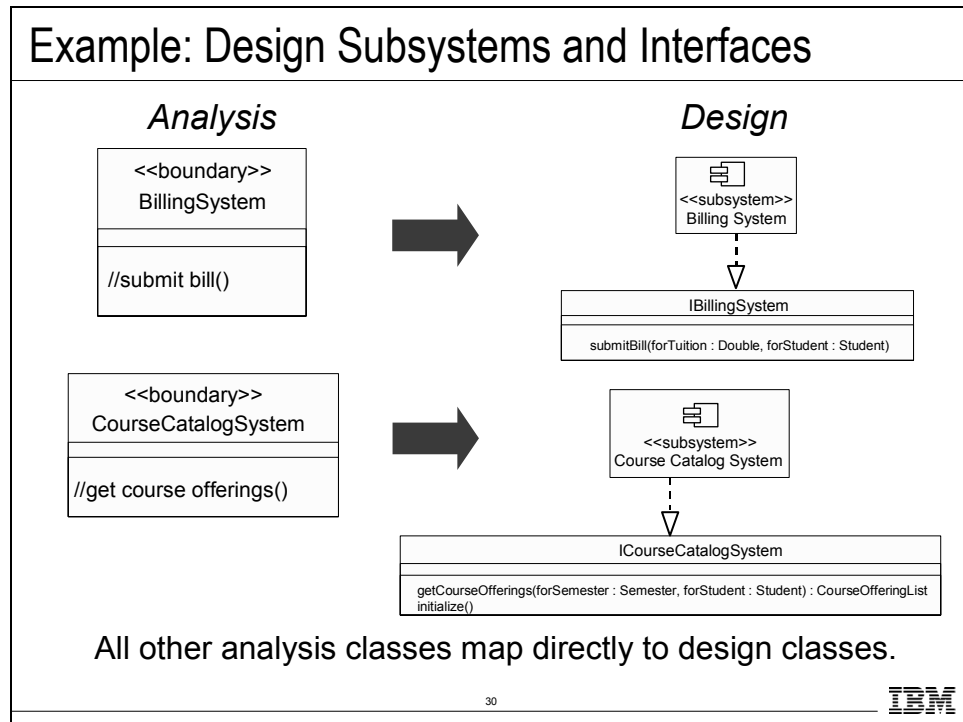
**Interface description:** The description should convey the responsibilities of the interface. The description should be several sentences long, up to a short paragraph. The description should not simply restate the name of the interface. Instead, it should illuminate the role the interface plays in the system.

**Operation definition:** Each interface should provide a unique and well-defined set of operations. Operation names should reflect the result of the operation. When an operation sets or gets information, including "set" or "get" in the name of the operation is redundant. Give the operation the same name as the property of the model element that is being set or retrieved. Example: name() returns the name of the object; name(aString) sets the name of the object to aString.

The description of the operation should describe what the operation does, including any key algorithms, and what value it returns. Name the parameters of the operation to indicate what information is being passed to the operation. Identify the type of the parameter.

**Interface documentation:** The behavior defined by the interface is specified as a set of operations.

## Example: Design Subsystems and Interfaces



During Use-Case Analysis, we modeled two boundary classes, the `BillingSystem` and the `CourseCatalogSystem`, whose responsibilities were to cover the details of the interfaces to the external systems. It was decided by the architects of the Course Registration System that the interactions to support external system access will be more complex than can be implemented in a single class. Thus, subsystems were identified to encapsulate these responsibilities and provide interfaces that give the external systems access. The above diagram includes these subsystems, as well as their interfaces.

The `BillingSystem` subsystem provides an interface to the external billing system. It is used to submit a bill when registration ends and students have been registered in courses.

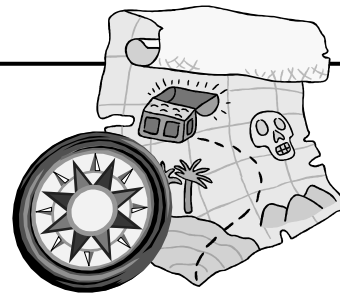
The `CourseCatalogSystem` subsystem encapsulates all the work involved for communicating to the legacy Course Catalog System. The system provides access to the unabridged catalog of all courses and course offerings provided by the university, including those from previous semesters.

These are subsystems rather than packages because a simple interface to their complex internal behaviors can be created. Also, by using a subsystem with an explicit and stable interface, the particulars of the external systems to be used (in this case, the `Billing System` and the legacy `Course Catalog System`) could be changed at a later date with no impact on the rest of the system.

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

### 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	

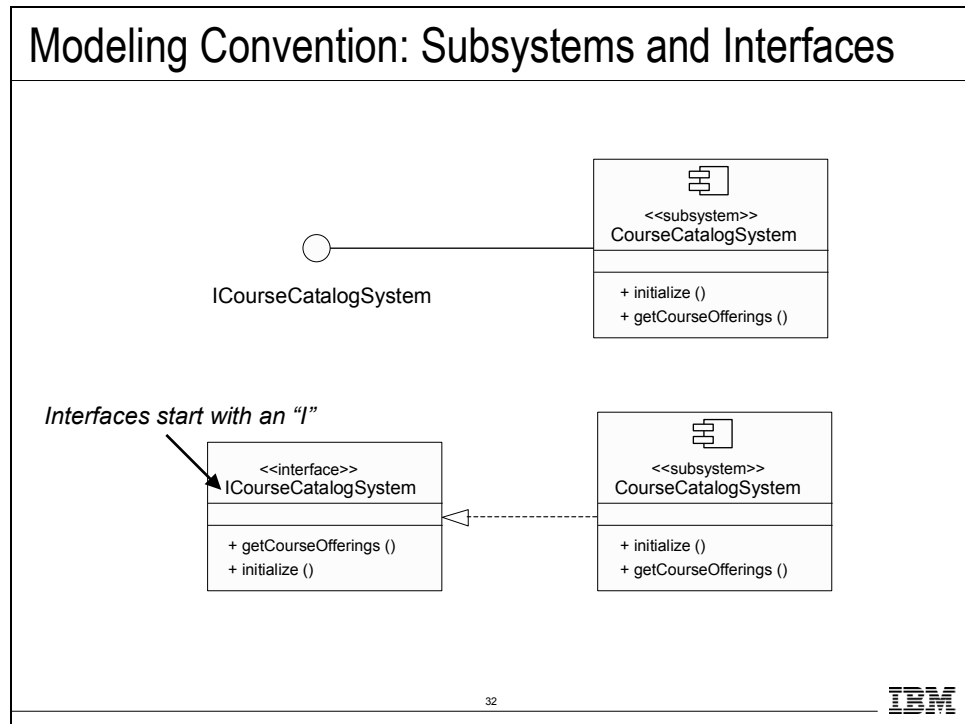


31

IBM

The above table is an example of how you could document the mapping of analysis classes to design elements. This may be refined as the Design process continues.

## Modeling Convention: Subsystems and Interfaces



For this course, we will represent subsystems as three items in the model:

- A `<<subsystem>>` package (that is, package with a stereotype of `<<subsystem>>`),
- A `<<subsystem proxy>>` class (that is, class with a stereotype of `<<subsystem proxy>>`)
- A subsystem interface (class with a stereotype of `<<interface>>`). The interface names will start with an 'I'.

The `<<subsystem>>` package provides a container for the elements that comprise the subsystem, the interaction diagrams that describe how the subsystem elements collaborate to implement the operations of the interfaces the subsystem realizes, and other diagrams that clarify the subsystem elements. The subsystem realizes the interface.

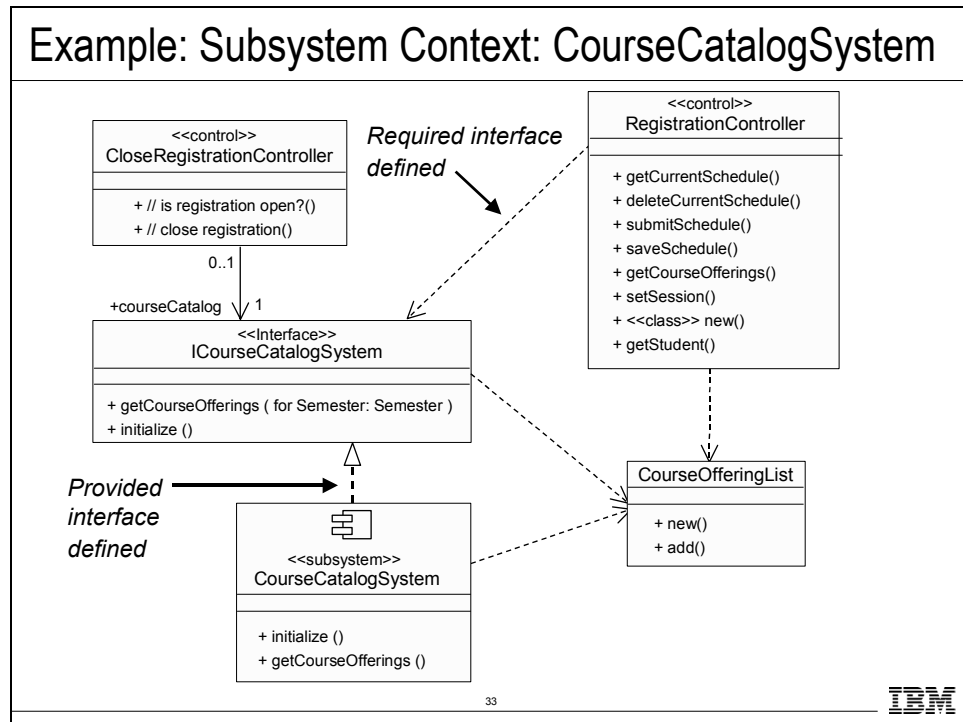
The `<<subsystem proxy>>` class realizes the interface(s) (as a proxy) and will orchestrate the implementation of the subsystem interface(s) operations. (This will be discussed further in the module on Subsystem Design.)

Such conventions make the consistent modeling of subsystems easier. We will see later in the course how we utilize this convention in representing subsystems in our diagrams.

Remember, interfaces are EXTERNAL to the subsystem package.



## Example: Subsystem Context: CourseCatalogSystem



The above is a context diagram for the CourseCatalogSystem subsystem.

A subsystem context class diagram should contain the subsystem, interface(s), associated realizes relationship(s), and any subsystem relationships (both to/from the subsystem and from/to other design elements).

The **ICourseCatalogSystem** interface is dependent on the **CourseOfferingList** class as the **CourseOfferingList** class appears in the signature of one its operations.

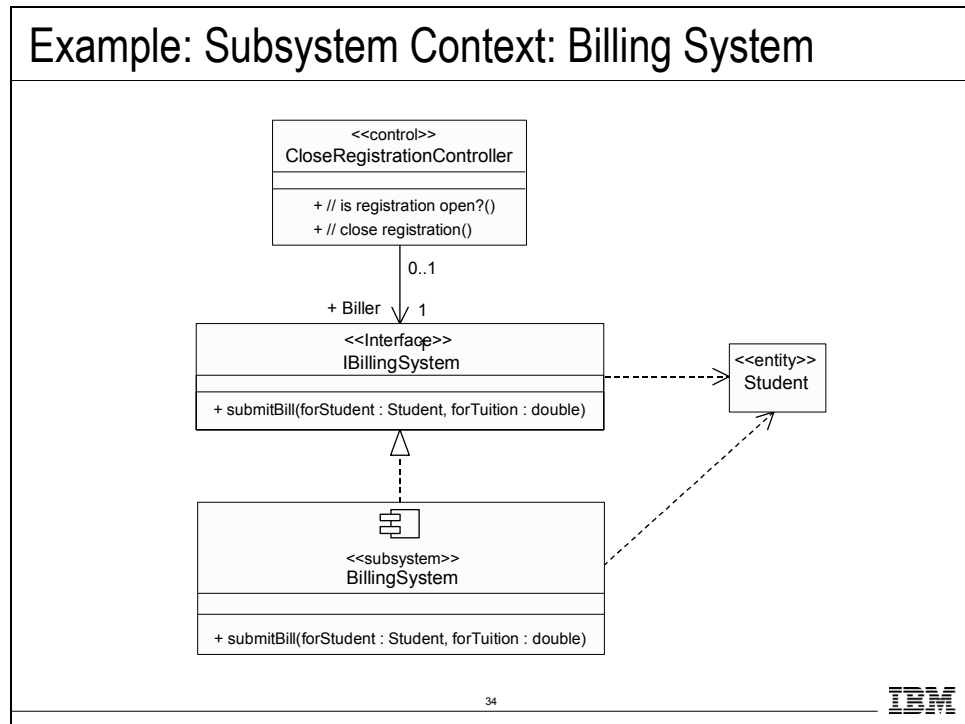
The **RegistrationController** and the **CloseRegistrationController** classes are dependent on the **ICourseCatalogSystem** interface to obtain the list of **CourseOfferings** being offered for a particular semester.

Notice the use of the modeling conventions, especially the subsystem proxy class and the realization relationship.

In **Identify Design Elements**, the interfaces are completely defined, including their signatures. This is important, as these interfaces will serve as synchronization points that enable parallel development.

There is not a Semester class. It is envisioned that this will be some type of enumeration in the implementation.

## Example: Subsystem Context: Billing System



The above is a context diagram for the BillingSystem subsystem.

A subsystem context class diagram should contain the subsystem, interface(s), associated realizes relationship(s), and any subsystem relationships (both to/from the subsystem and from/to other design elements).

The IBillingSystem interface is dependent on the student class as the student class appears in the signature of one its operations. (Stay tuned for dependencies in the Class Design module.)

The CloseRegistrationController is dependent on the IBillingSystem interface to bill the student for the courses he or she is enrolled in.

Notice the use of the modeling conventions, especially the subsystem proxy class and the realization relationship.

## Identify Design Elements Steps

### Identify Design Elements Steps

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



35



The identification of reuse opportunities is an important architectural step. It will help you to determine which subsystems and packages to develop, which ones to reuse, and which ones to buy. It is desirable to reuse existing components, wherever possible. Such reuse allows you to leverage existing successful implementations, eliminating (or greatly reducing) the resources needed to develop and test the component.

The identification of reuse opportunities is a unification effort, since it is determined if “things” that have been identified can be satisfied by what already exists.

The most effective reuse identification occurs after there is an understanding of the required behavior of the system and after some initial partitioning of the design elements has occurred.

On the next three slides, you will see the possible sources of reuse and how such reuse could be incorporated into the architecture.

## Identify Reuse Opportunities

### 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

36



Look for existing subsystems or components that offer similar interfaces. Compare each identified interface to the interfaces provided by existing subsystems or components. There usually will not be an exact match, but approximate matches can be found. Look first for similar behavior and returned values, then consider parameters.

Modify the newly identified interfaces to improve the fit. There may be opportunities to make minor changes to a candidate interface that will improve its conformance to the existing interface. Simple changes include rearranging or adding parameters to the candidate interface. They also include factoring the interface by splitting it into several interfaces, one or more of which match those of the existing component, with the "new" behaviors located in a separate interface.

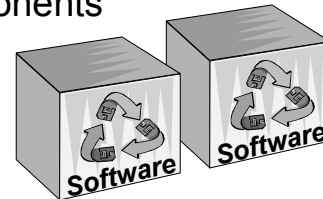
Replace candidate interfaces with existing interfaces where exact matches occur. After simplification and factoring, if there is an exact match to an existing interface, eliminate the candidate interface and simply use the existing interface.

Map the candidate subsystem to existing components. Look at existing components and the set of candidate subsystems. Factor the subsystems so that existing components are used wherever possible to satisfy the required behavior of the system. Where a candidate subsystem can be realized by an existing component, create traceability between the subsystem and the component. Note in the description for this subsystem that the behavior is satisfied by the associated existing component. In mapping subsystems onto components, consider the design mechanisms associated with the subsystem. Performance or security requirements may disqualify a component from reuse despite an otherwise perfect match between operation signatures.

## Possible Reuse Opportunities

### 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



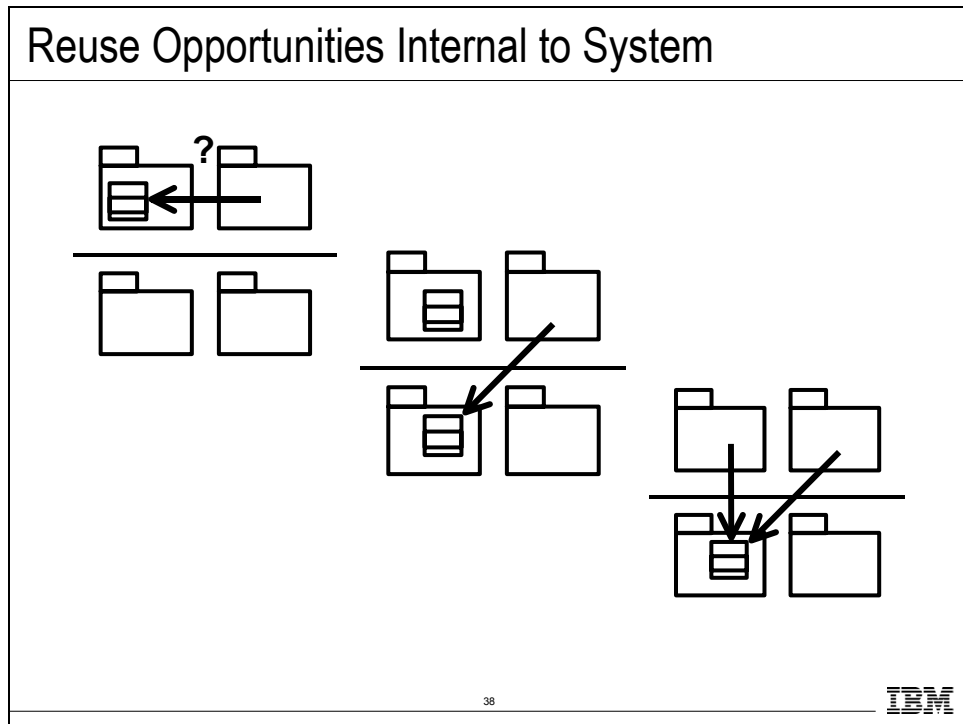
37



In organizations that build similar systems, there is often a set of common components that provide many of the architectural mechanisms needed for a new system. There may also be components available in the marketplace that also fill the need for architectural mechanisms. Existing components should be examined to determine their suitability and compatibility within the software architecture. These components can be reverse engineered into the Design Model for examination and incorporation. You can then use design elements as proxies for the components. This allows you to show how what has been bought and what needs to be built interact (a necessary requirement unless you are buying all your components and only doing assembly).

The advent of commercially successful component infrastructures such as CORBA, the Internet, ActiveX and JavaBeans triggers a whole industry of off-the-shelf components for various domains, allowing you to buy and integrate components rather than developing them all in-house. Reusable components provide common solutions to a wide range of common problems and may be larger than just collections of utilities or class libraries; they form the basis of reuse within an organization, increasing overall software productivity and quality.

## Reuse Opportunities Internal to System



Reuse can be discovered within the current software system. There may be design elements that are needed by more than one subsystem or package. The above slide is meant to visually represent what happens as reuse is discovered:

- Build the first application and some general parts.
- Build the second application, and you find that some parts of the first can be reused, but the parts were not designed to be reused. (Some are reused any way, with chaos as result). It is recognized that something is being used by one element that other elements may find useful (first diagram).
- Take the candidate reusable design elements (classes, packages, or subsystems) and make them reusable. This may involve changing their names, making them more general, improving their documentation, moving them to a common functionality layer, etc. Essentially, the candidate reusable entity is “pushed down” to a lower layer in the architecture so other elements may access it. Initially, only the original client of the element may use it (second diagram).
- Now can you use them for the new application.
- When the first application is upgraded, the old versions of the elements can be replaced with the reusable ones (third diagram).

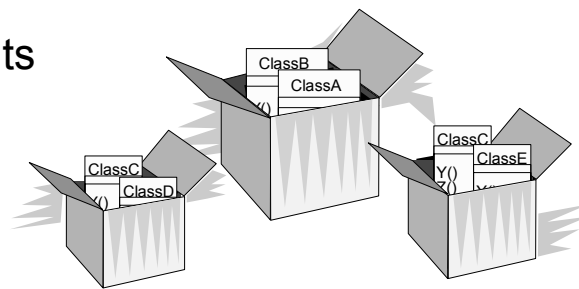
It is important for the architect to look within and across the design elements for common behavior (for example, common collaborations), pulling it out where possible. This is a reuse scavenging activity that falls under the **Identify Design Elements** umbrella.

*Note: To see an example of reuse of a Security mechanism, refer to the Security Mechanism tab in the Additional Information Appendix.*

## Identify Design Elements Steps

### Identify Design Elements Steps

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



39

IBM

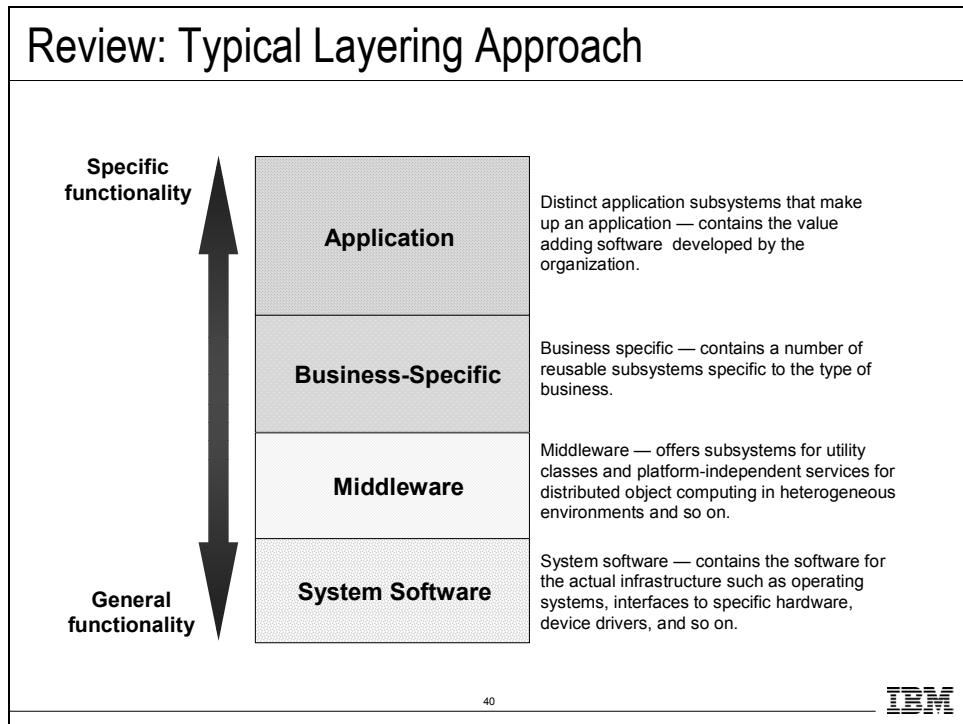
As new elements have been added to the Design Model, re-packaging the elements of the Design Model is often necessary. Repackaging achieves several objectives: It reduces coupling between packages and improves cohesion within packages in the Design Model. The ultimate goal is to allow different packages (and subsystems) to be designed and developed independently of one another by separate individuals or teams. While complete independence is probably impossible to achieve, loose coupling between packages tends to improve the ease of development of large or complex systems.

As new model elements are added to the system, existing packages may grow too large to be managed by a single team: The package must be split into several packages which are highly cohesive within the package but loosely coupled between the packages. Doing this may be difficult — some elements may be difficult to place in one specific package because they are used by elements of both packages. There are two possible solutions:

- Split the element into several objects, one in each package (this works where the element has several 'personalities,' or sets of somewhat disjoint responsibilities)
- Move the element into a package in a lower layer, where all higher layer elements might depend upon it equally.

You will discuss modeling the lower layers in this step.

## Review: Typical Layering Approach



This is a repeat of the slide first introduced in Architectural Analysis. It is included here as a review.

During Architectural Analysis, the focus was on the upper-level layers (that is, the application and business-specific layers). During **Identify Design Elements**, the focus is on the lower-level layers.

The layering principles originally described for packages also apply to subsystems.



## Layering Considerations

### 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.

41



Layering provides a logical partitioning of packages into layers with certain rules concerning the relationships between layers. Restricting inter-layer and inter-package dependencies makes the system more loosely coupled and easier to maintain. Failure to restrict dependencies causes architectural degradation, and makes the system brittle and difficult to maintain.

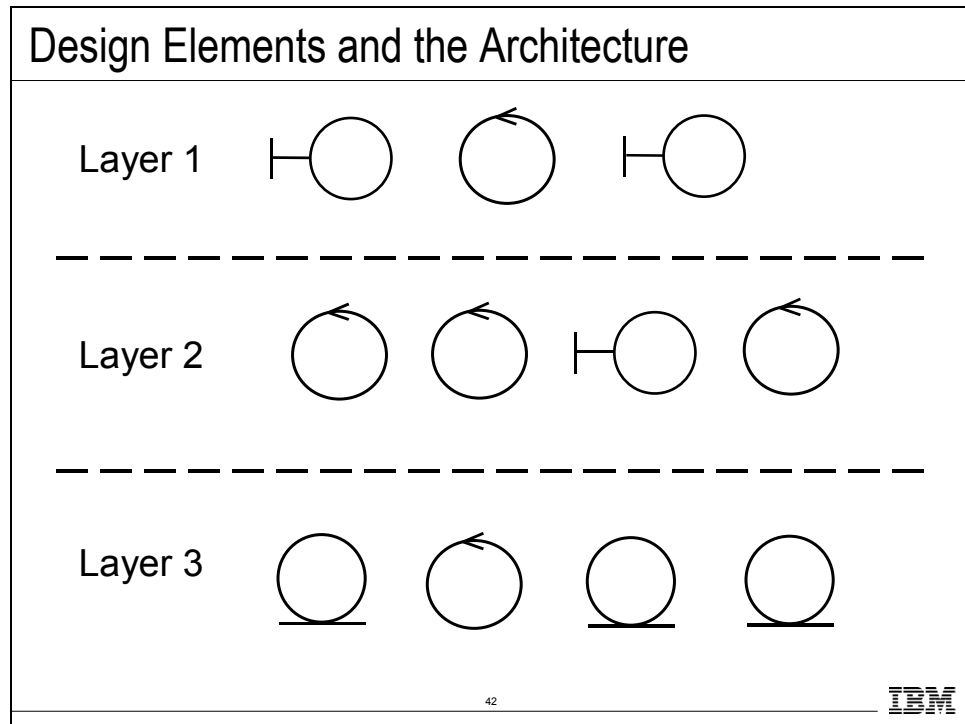
**Visibility:** Elements should only depend on elements in the same layer and the next lower layer. Exceptions include cases where packages need direct access to lower-layer services (for example, primitive services needed throughout the system, such as printing, sending messages, and so forth). There is little value in restricting messages to lower layers if the solution is to effectively implement call pass-throughs in the intermediate layers.

**Volatility:** In the highest layers, put elements that vary when user requirements change. In the lowest layers, put elements that vary when the implementation platform (hardware, language, operating system, database, and so forth) changes. Sandwiched in the middle, put elements that are generally applicable across wide ranges of systems and implementation environments. Add layers when additional partitions within these broad categories help to organize the model.

**Generality:** Abstract model elements tend to be placed lower in the model, where they can be reused. If not implementation-specific, they tend to gravitate toward the middle layers.

**Number of Layers:** For a small system, three layers are sufficient. For a complex system, five-to-seven layers are usually sufficient. For any degree of complexity, more than ten layers should be viewed with suspicion that increases with the number of layers.

## Design Elements and the Architecture



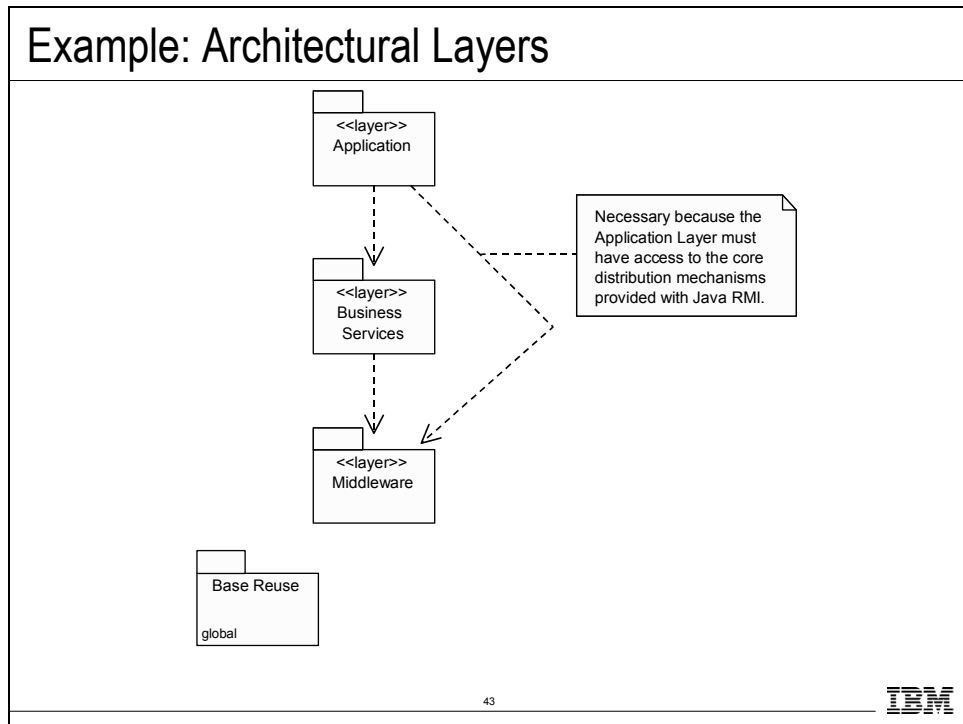
The identified design elements need to be allocated to specific layers in the architecture.

The above diagram is not meant to recommend or suggest specific layers. What is meant is that layers are *not* “just by stereotype.” You can have boundary classes in the Business Services layer (for example, outgoing interfaces), control classes in the Middleware layer (for example, transaction manager), and so on.

Even though all three stereotypes can appear in any layer, there are general trends that may help guide a novice designer. Most boundary classes tend to appear at the top, most control classes tend to appear in the services layer where control across entities is required, and most entities appear towards the bottom layers.

An experienced designer makes packages of classes that work together to provide a service (that is, he or she groups pieces of the system together that work closely to support some high-level capability). This leads to cohesive, reusable packages.

## Example: Architectural Layers



The layers defined in this activity are built on the architectural layers originally defined in Architectural Analysis.

The Middleware layer and the Base Reuse package were added in this activity.

The Middleware layer provides utilities and platform-independent services.

The Base Reuse package contains some common, generic reusable design elements and patterns.

The contents of each of these layers is described later in this module.

## Partitioning Considerations

### Partitioning Considerations

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

Try to avoid cyclic dependencies.

44



**Coupling and cohesion:** Design elements with tight coupling/cohesion (for example, lots of relationships and communication) should be placed in the same partition.

**User organization:** This occurs when an existing Enterprise Model has a strongly organizationally partitioned structure. Usually affects the top layers (application-specific services).

**Competence and/or skill areas:** In the middle and lower layers, specialization in skills should be considered during the development and support of complex infrastructure technology (for example, network, distribution, database, and/or communication management; process control). In the upper layers, the specialization of skills should be considered in the problem domain (for example, telecommunication call management, securities trading, insurance claims processing, and air traffic control).

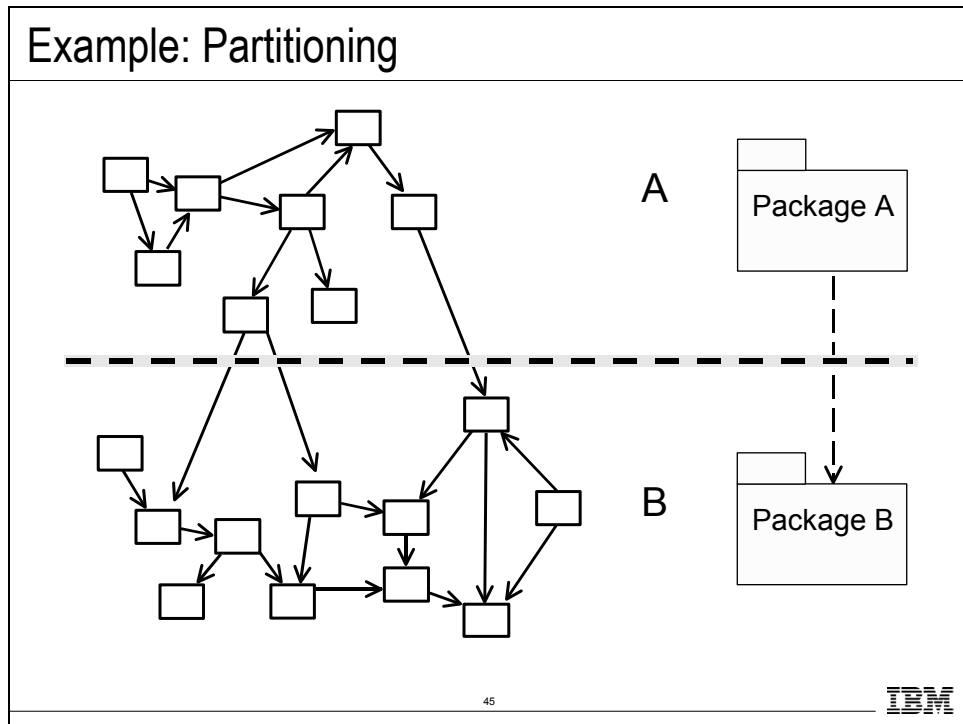
**System distribution:** This helps to visualize the network communication, which will occur as the system executes.

**Secrecy areas:** Functionality requiring special security clearance must be partitioned into subsystems that will be developed independently, with the interfaces to the secrecy areas the only visible aspect of these subsystems.

**Variability areas:** Functionality that is likely to be optional, and thereby delivered only in some variants of the system, should be organized into subsystems.

When partitioning, try to avoid circular dependencies, as they make it impossible to reuse one package without the other.

## Example: Partitioning

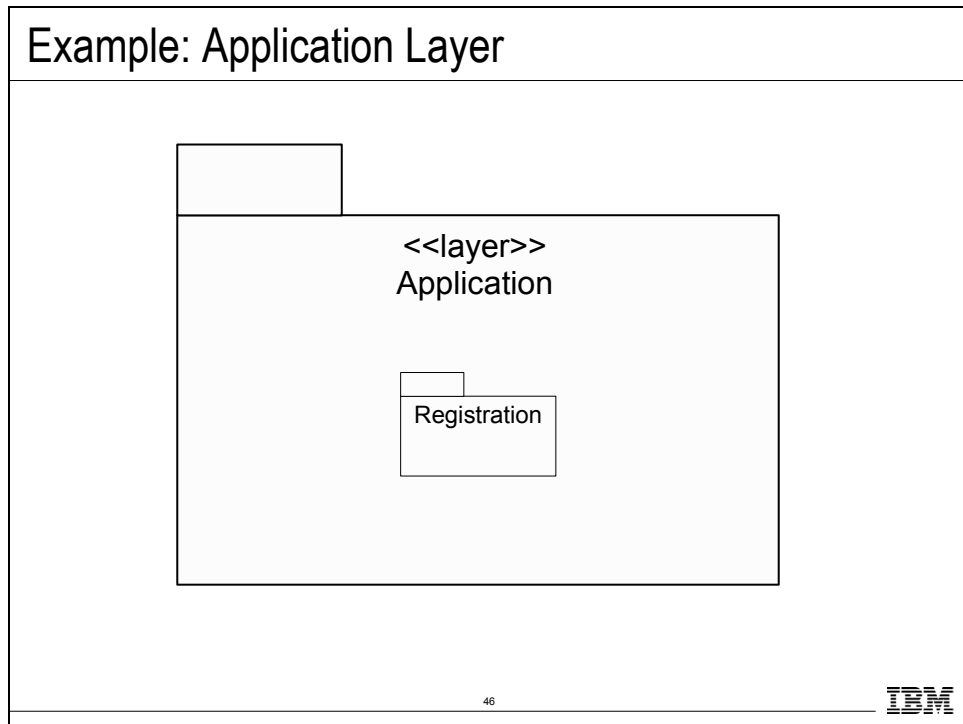


The above example shows a partitioning of design elements. The diagram on the left shows some classes divided into two partitions, A and B. The diagram on the right shows the resulting packages.

Notice the following:

- Maximum coupling and cohesion within packages versus minimal coupling between packages (left diagram).
- The dependencies between packages (right diagram) reflect/support the dependencies between the classes contained within the packages (left diagram).

## Example: Application Layer



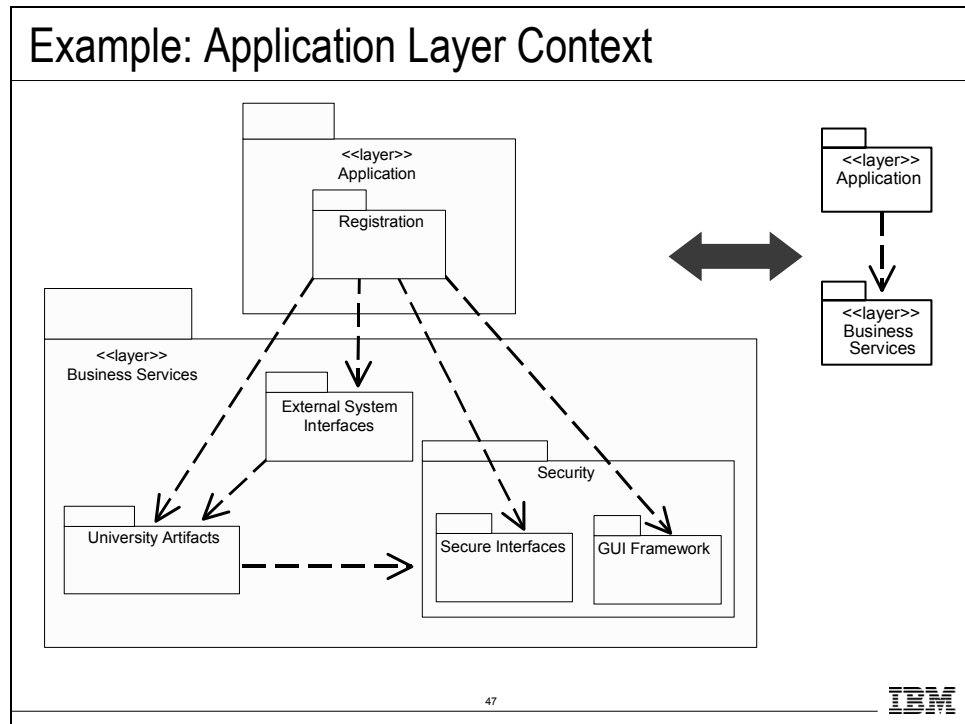
The next few slides describe the layering decisions for the Course Registration System.

The Application layer contains application-specific design elements.

The Registration package that was previously defined has been allocated to the Application layer.

In the Course Registration example, we have been concentrating on the Student Registration application. As additional applications are defined within the Course Registration System, additional packages could be added to the Application layer — for example, Student Evaluation.

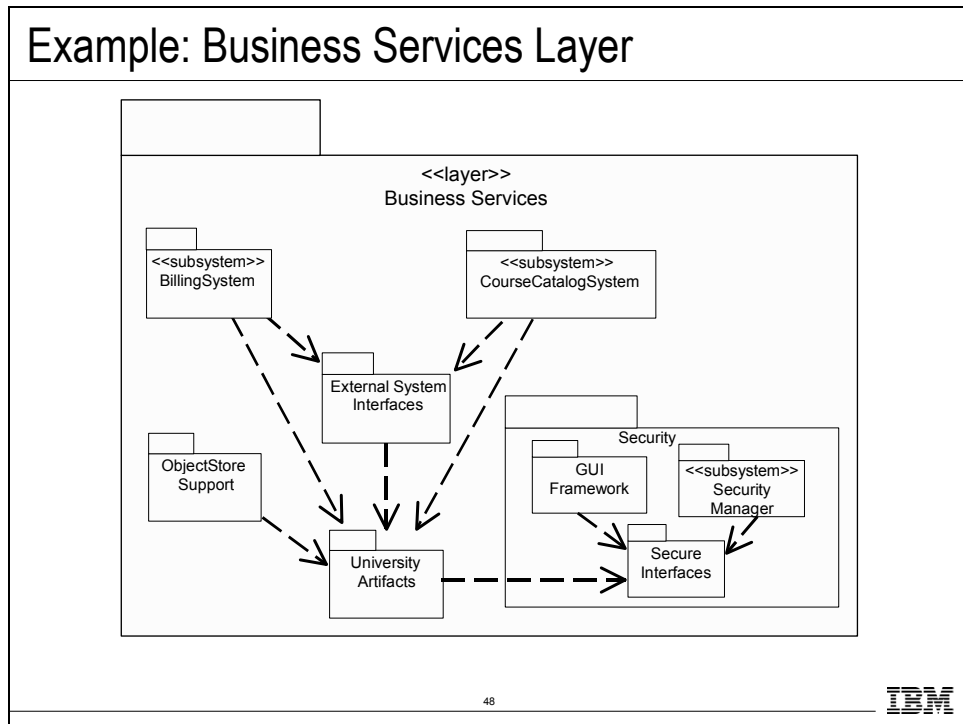
## Example: Application Layer Context



The Registration package depends on the University Artifacts package for the core abstractions, the External System Interface packages for the external system interfaces, and the GUI Framework and Security Interfaces packages for the security framework.

Notice how the package dependencies are consistent with the layer dependencies.

## Example: Business Services Layer

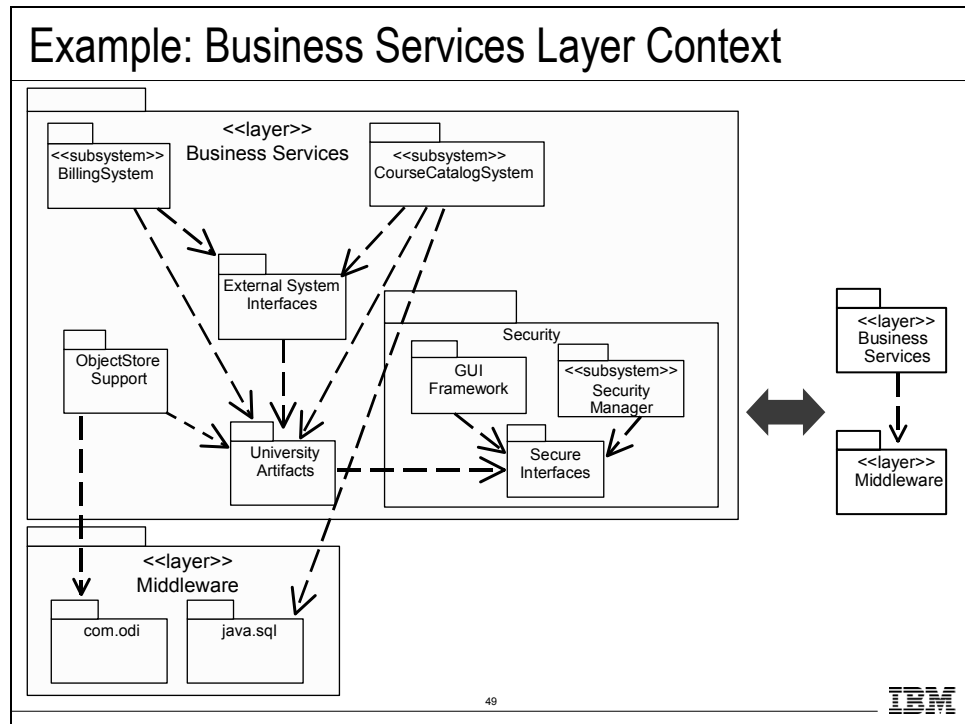


The Business Services layer contains business-specific elements that are used in several applications.

The external system access subsystems, their interfaces, the key abstractions package, the OODBMS persistency support package, and the security package were placed in the Business Services layer. It is anticipated that these packages might be required by multiple applications.



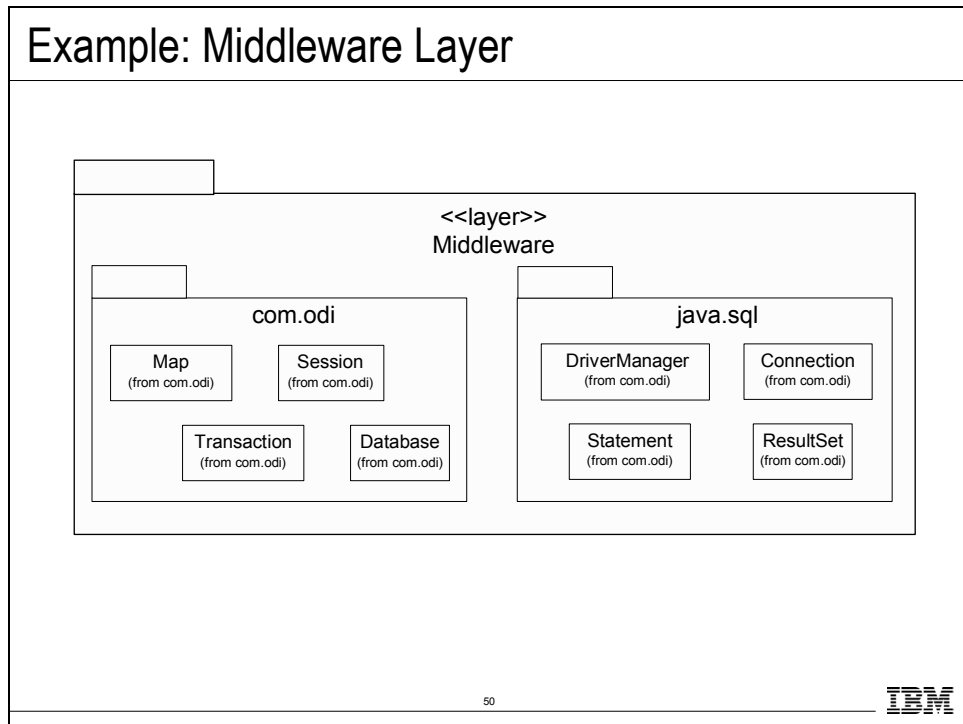
## Example: Business Services Layer Context



The above diagram models the context of the Business Services layer. It models the relationships that the Business Services layer has with the other layers in our architecture, and shows how the design element dependencies are consistent with the layer dependencies.

Packages within the Business Services layer require access to the `com.odi` and `java.sql` packages in order to utilize the core persistency mechanisms.

## Example: Middleware Layer



The Middleware layer provides utilities and platform-independent services.

All of these packages are commercially available, `com.odi` from ObjectStore and `java.sql` with most standard Java IDEs.

## Identify Design Elements Steps

### Identify Design Elements Steps

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



51



This is where the quality of the architecture modeled up to this point is assessed against some very specific criteria.

In this module, we will concentrate on those checkpoints that the designer is most concerned with. The architect should do a much more detailed review of the **Identify Design Elements** results and correct any problems before the project moves on to the next activity. We will not cover those checkpoints, as they are out of scope of this course. (Remember, this is not an architecture course.)

## Checkpoints

Checkpoints	
<ul style="list-style-type: none"><li>♦ General<ul style="list-style-type: none"><li>▪ Does it provide a comprehensive picture of the services of different packages?</li><li>▪ Can you find similar structural solutions that can be used more widely in the problem domain?</li></ul></li><li>♦ Layers<ul style="list-style-type: none"><li>▪ Are there more than seven layers?</li></ul></li><li>♦ Subsystems<ul style="list-style-type: none"><li>▪ Is subsystem partitioning done in a logically consistent way across the entire model?</li></ul></li></ul>	
52	

The next few slides contain the key things a designer would look for when assessing the results of **Identify Design Elements**. As stated earlier, an architect would have a more detailed list.

A well-structured architecture:

- Encompasses a set of classes, typically organized into multiple hierarchies.
- Provides a set of collaborations that specify how those classes cooperate to provide various system functions.

## Checkpoints (continued)

### 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?



53

**IBM**

If the classes contained in a package do not belong there according to the criteria for the package division then move them to other packages or create more packages.

If the classes in the packages are not related functionally, then move some of the classes to other packages or create more packages.

As a guideline regarding an appropriate ratio between the number of packages and the number of classes, 5 packages and 1,000 classes is probably a sign that something is wrong.

## Checkpoints (continued)

### 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?



54

IBM

A well-structured class:

- Provides a crisp abstraction of some thing drawn from the vocabulary of the problem domain or the solution domain.
- Embodies a small, well-defined set of responsibilities, and carries them all out very well.
- Provides a clear separation of the abstraction's behavior and its implementation.
- Is understandable and simple yet extensible and adaptable.

A sampling of some class checkpoints are listed above. A more detailed set of checkpoints for classes will be discussed in the Class Design module.

## Review

---

### 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?

55



## Exercise: Identify Design Elements

### Exercise: Identify Design Elements

#### ◆ Given the following:

##### ▪ The analysis classes and their relationships

- Payroll Exercise Solution, Use-Case Analysis, Part 2 (use-case realization VOPCs)

##### ▪ The layers, packages, and their dependencies

- Exercise Workbook: Payroll Architectural Handbook, Logical View, Architectural Layers and their Dependencies



56



The goal of this exercise is to identify design elements (subsystems, interfaces, design classes) and then identify the location of the design elements in the architecture.

Use the results of the Use-Case Analysis exercise as input for this activity.

References to the givens:

- The analysis classes and their relationships: Payroll Exercise Solution, Use-Case Analysis, Part 2 (use-case realization VOPCs).
- The architectural layers and their dependencies: Exercise Workbook: *Payroll Architectural Handbook*, Logical View, Architectural Layers and their Dependencies.



## Exercise: Identify Design Elements (continued)

### Exercise: Identify Design Elements (continued)

- ♦ 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)



57

IBM

Review the previously defined analysis classes, and look for potential subsystems, keeping in mind the subsystem usage suggestions discussed in this module. For each subsystem you choose, be prepared to justify the choice.

Remember to complete the signatures for the identified interfaces.

Note: Subsystems might be depended on by other design elements. The dependencies on a subsystem should be on the subsystem interface(s). It is important to understand the context in which the subsystem exists.

Once subsystems have been identified and their interfaces defined, it is important to document the mapping from the initial analysis classes to these design elements as this will provide analysis-to-design traceability.

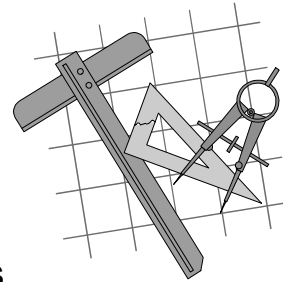
Up to this point, the analysis classes were not mapped to a specific position in the architecture. (They were not mapped to packages.) It is during **Identify Design Elements** that we must allocate the design elements (subsystems and classes) to an appropriate place in the architecture (that is, a package and/or layer). This involves partitioning, as we discussed earlier in the module.

When partitioning the design elements, make sure that the dependencies between the packages and layers support the dependencies needed between the model elements contained in the packages and layers.

## Exercise: Identify Design Elements

### 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



58

IBM

The interface realization class diagram should contain the subsystem proxy class, the interface(s) the subsystem realizes, the associated realization relationship(s), and any subsystem relationships (both to and from the subsystem to other design elements). Since the objective of this exercise is NOT to do the design of the subsystems, in most cases this diagram will only contain relationships TO the subsystem INTERFACE.

Be sure to use the conventions defined in the course. (For example, use the recommended stereotypes.)

The produced table should have two columns: “Design Element” and “Owning Package.” The “owning” package is the UML package that directly contains the design element.

Refer to the following slide if needed:

- Subsystem context diagrams: Slides 33-34

## Exercise: Review

### 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?



59

IBM

After completing a model, it is important to step back and review your work. Some helpful questions are the following:

- Has the use case behavior been successfully represented in the model? In other words, is the flow of events the same in the specifications as it is in the model?
- Has there been any significant behavior that was added? Removed? Changed? The model should reflect the intent of the use-case specifications.
- Is each stereotype behaving properly? Are actors only interfacing with boundary classes? Are control classes controlling the use-case flow of events only? Are any classes doing operations on data (attributes) that are not owned by that class?

