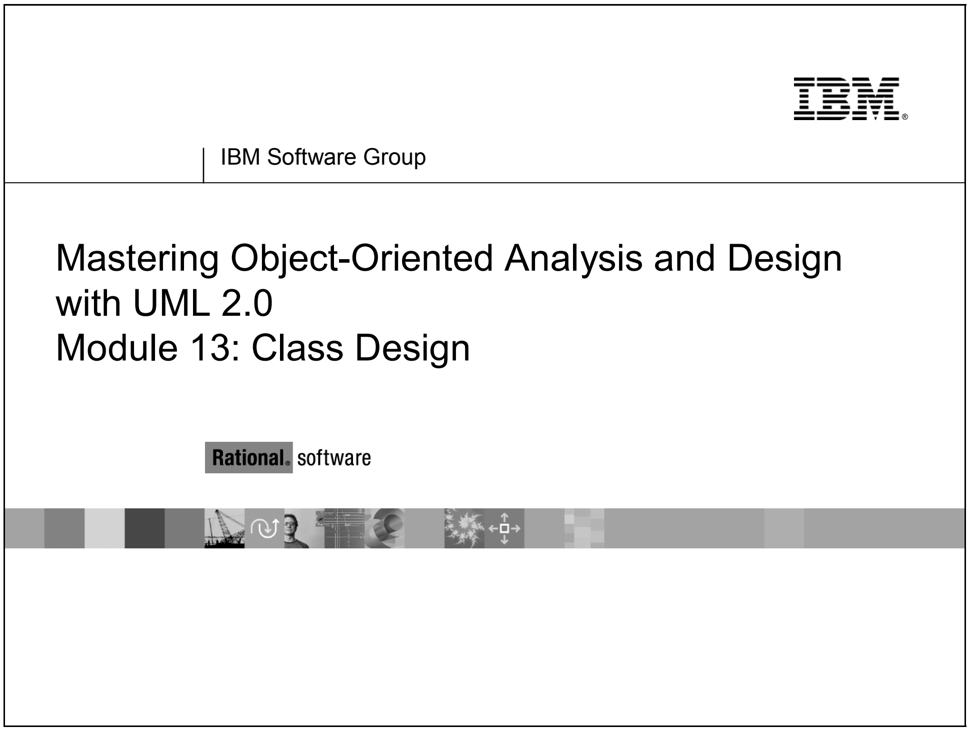
▸ ▸ ▸ Module 13 Class Design

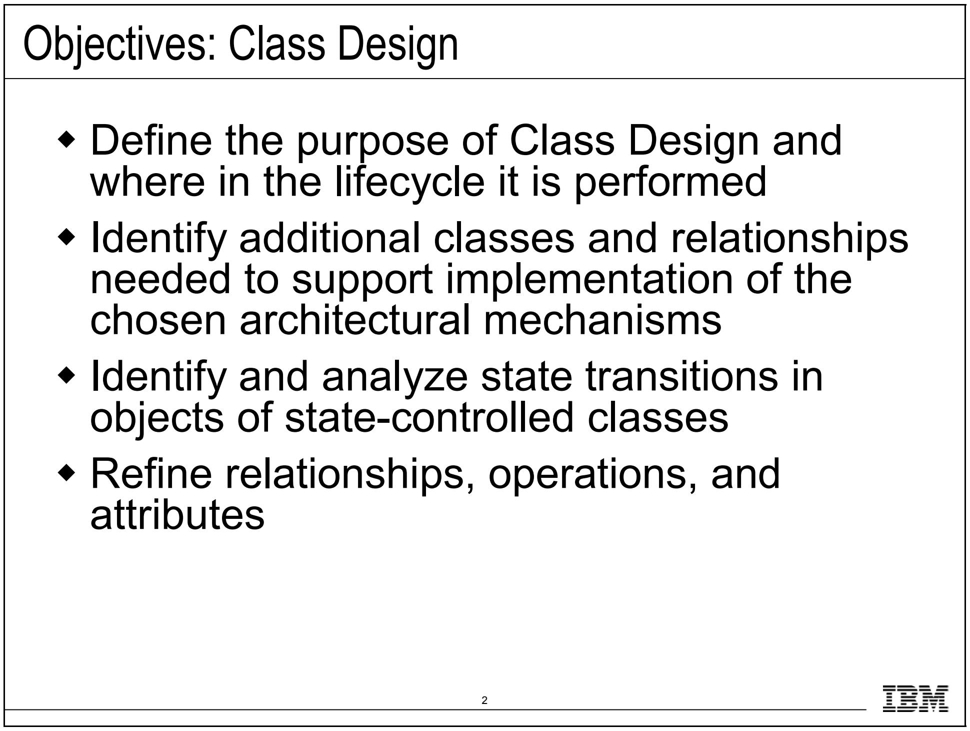


IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0
Module 13: Class Design

Topics

Objectives: Class Design

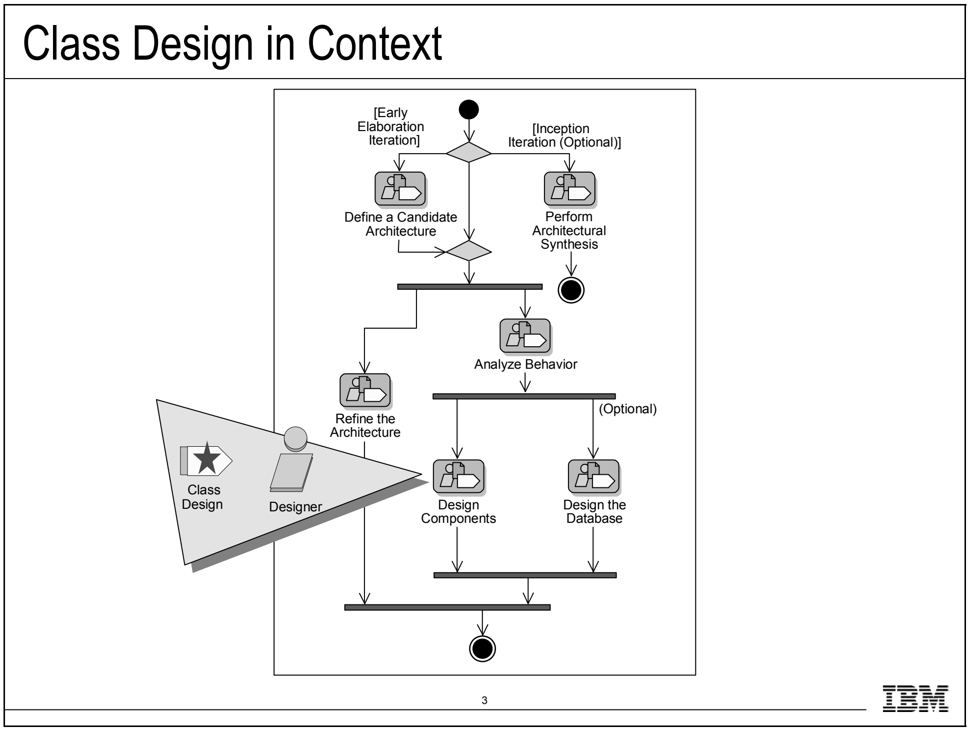


Objectives: Class Design

- Define the purpose of Class Design and where in the lifecycle it is performed
- Identify additional classes and relationships needed to support implementation of the chosen architectural mechanisms
- Identify and analyze state transitions in objects of state-controlled classes
- Refine relationships, operations, and attributes

In **Class Design**, the focus is on fleshing out the details of a particular class (for example, what operations and classes need to be added to support, and how do they collaborate to support, the responsibilities allocated to the class).

Class Design in Context



Class Design in Context

[Early Elaboration Iteration]
[Inception Iteration (Optional)]
Define a Candidate Architecture
Perform Architectural Synthesis
Analyze Behavior
Refine the Architecture
(Optional)
Class Design
Designer
Design Components
Design the Database
3
IBM

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.

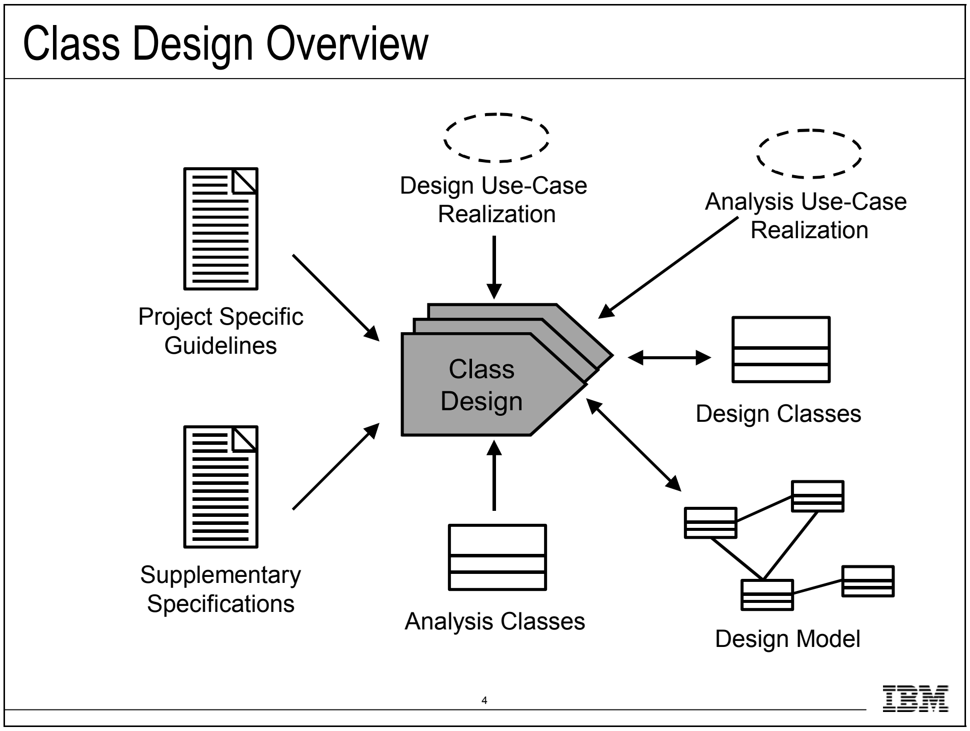
Identify Design Elements is where you decide what the infrastructure is. The infrastructure is the pieces of the architecture, if you will, and how they interact. Use-Case Design is where the responsibilities of the system are allocated to the pieces. Subsystem Design and **Class Design** are where you detail the specifics of the pieces.

During **Class Design**, you take into account the implementation and deployment environments.You may need to adjust the classes to the particular products in use, the programming languages, distribution, adaptation to physical constraints (for example, limited memory), performance, use of component environments such as COM or CORBA, and other implementation technologies.

There is frequent iteration between **Class Design**, Subsystem Design, and Use-Case Design.

Class Design is performed for each class in the current iteration.

Class Design Overview



Class Design Overview

Design Use-Case Realization

Analysis Use-Case Realization

Project Specific Guidelines

Class Design

Design Classes

Supplementary Specifications

Analysis Classes

Design Model

4

Purpose:

- Ensure that the classes provide the behavior the Use-Case Realizations require.
- Ensure that sufficient information is provided to implement the class.
- Handle non-functional requirements related to the class.
- Incorporate the design mechanisms used by the class.

Input Artifacts:

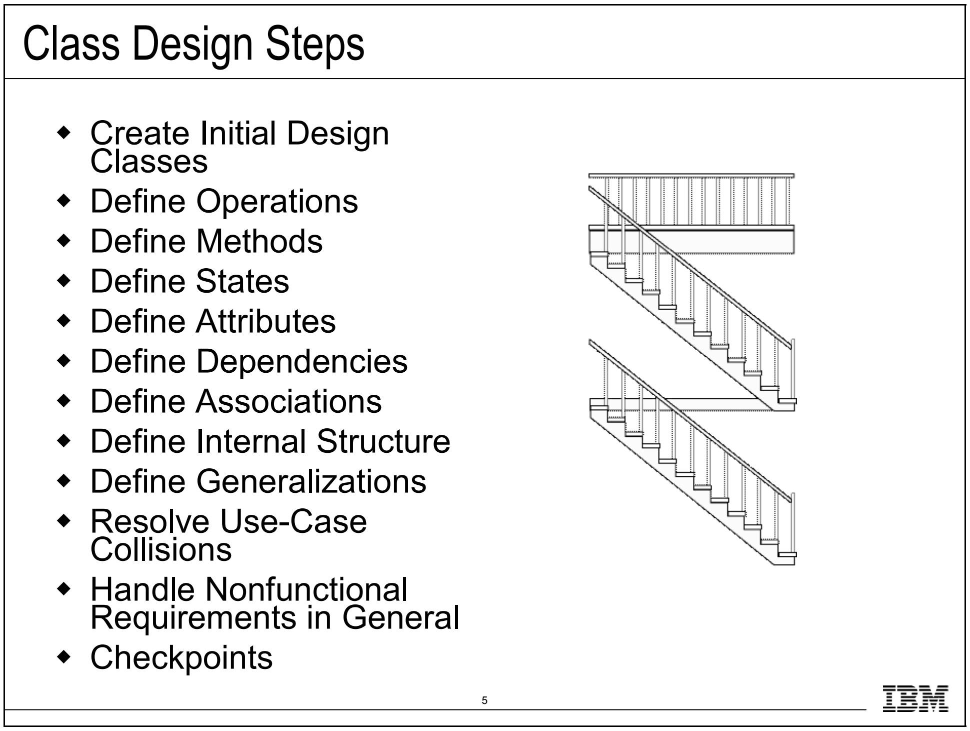
- Supplementary Specifications
- Project Specific Guidelines
- Analysis Classes
- Analysis use-case realization
- Design classes
- Design Model
- Design use-case realization

Resulting Artifacts:

- Design classes
- Design Model

The detailed steps performed during this activity are summarized on the next page, and described in the remainder of this module.

Class Design Steps



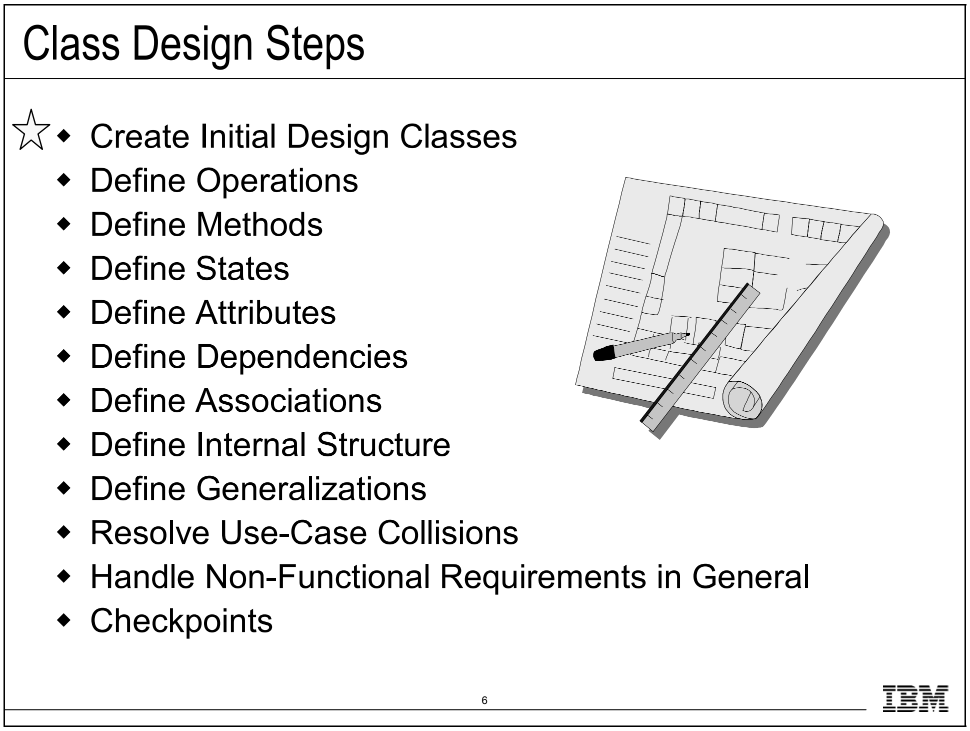
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Nonfunctional Requirements in General
- Checkpoints

5

The **Class Design** steps that you will address in this module are listed above.

Class Design Steps



Class Design Steps

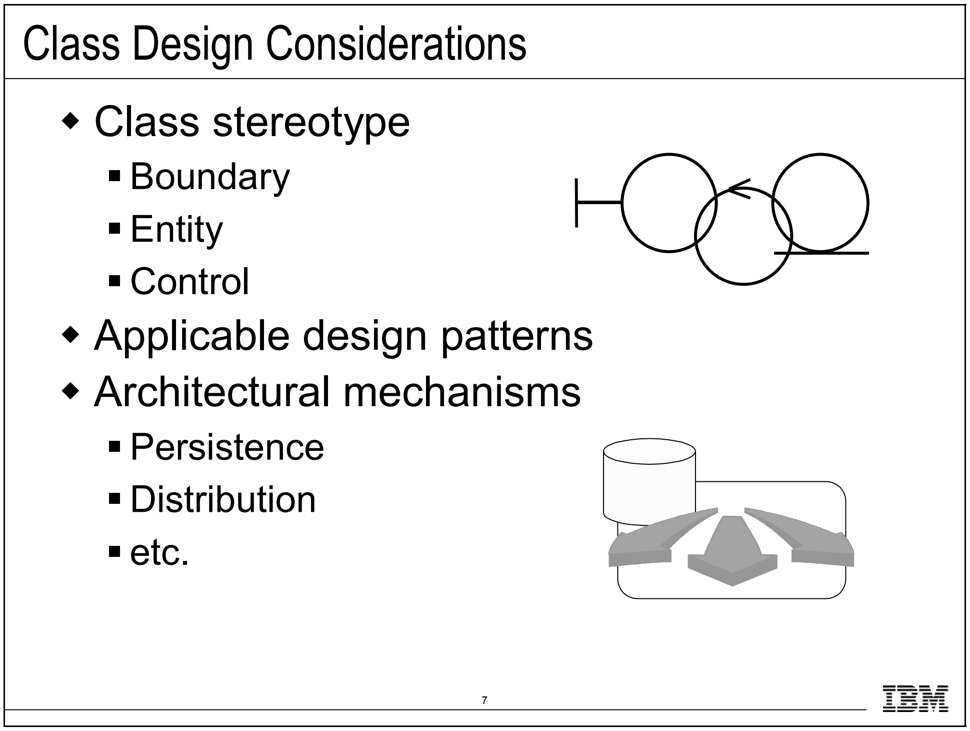
☆
- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

6

IBM

At this point, we create one or several (initial) design classes from the original design classes given as input to this activity. The design classes created in this step will be refined, adjusted, split and/or merged in the subsequent steps when assigned various "design" properties, such as operations, methods, and a state machine, describing how the class is designed.

Class Design Considerations



Class Design Considerations

- Class stereotype
  - Boundary
  - Entity
  - Control
- Applicable design patterns
- Architectural mechanisms
  - Persistence
  - Distribution
  - etc.

7

When performing **Class Design**, you need to consider:

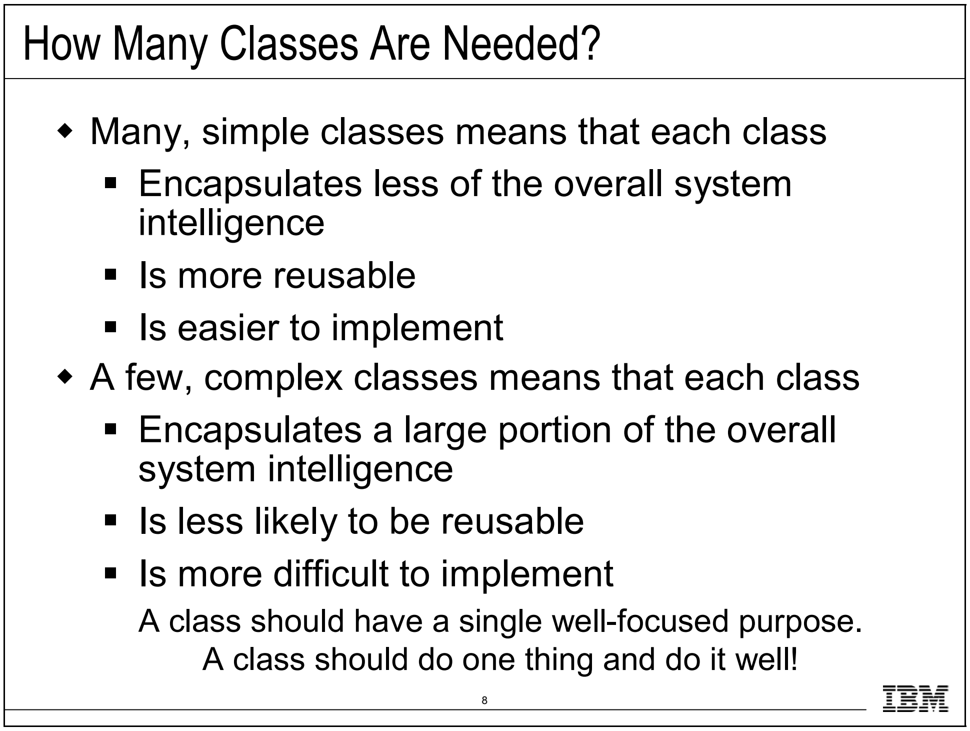
- How the classes that were identified in analysis as boundary, control, and entity classes will be realized in the implementation.
- How design patterns can be used to help solve implementation issues.
- How the architectural mechanisms will be realized in terms of the defined design classes.

Specific strategies can be used to design a class, depending on its original analysis stereotype (boundary, control, and entity). These stereotypes are most useful during Use-Case Analysis when identifying classes and allocating responsibility. At this point in design, you really no longer need to make the distinction — the purpose of the distinction was to get you to think about the roles objects play, and make sure that you separate behavior according to the forces that cause objects to change. Once you have considered these forces and have a good class decomposition, the distinction is no longer really useful.

Here the class is refined to incorporate the architectural mechanisms.

When you identify a new class, make an initial pass at its relationships. These are just the initial associations that tie the new classes into the existing class structure. These will be refined throughout **Class Design**.

How Many Classes Are Needed?



How Many Classes Are Needed?

- Many, simple classes means that each class
  - Encapsulates less of the overall system intelligence
  - Is more reusable
  - Is easier to implement
- A few, complex classes means that each class
  - Encapsulates a large portion of the overall system intelligence
  - Is less likely to be reusable
  - Is more difficult to implement

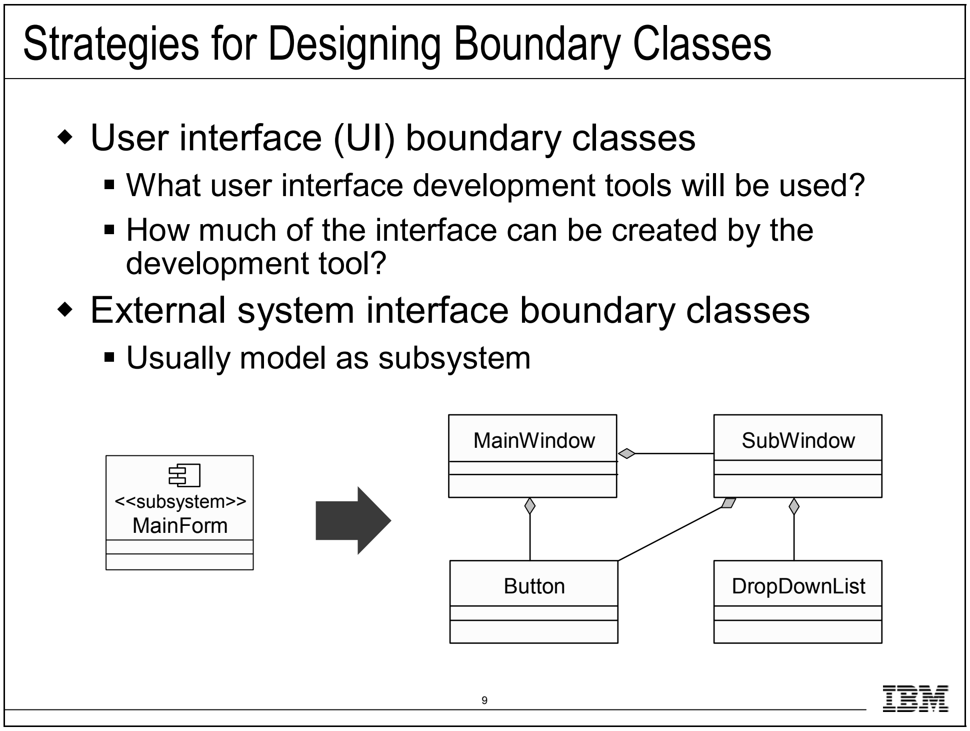
A class should have a single well-focused purpose. A class should do one thing and do it well!

These are some things you should think about as you define additional classes.

The proper size of a class depends heavily on the implementation environment. Classes should map directly to some phenomenon in the implementation language in such a way that the mapping results in good code.

With that said, you should design as if you had classes and encapsulation even if your implementation language does not support it. This will help keep the structure easy to understand and modify.

Strategies for Designing Boundary Classes



Strategies for Designing Boundary Classes

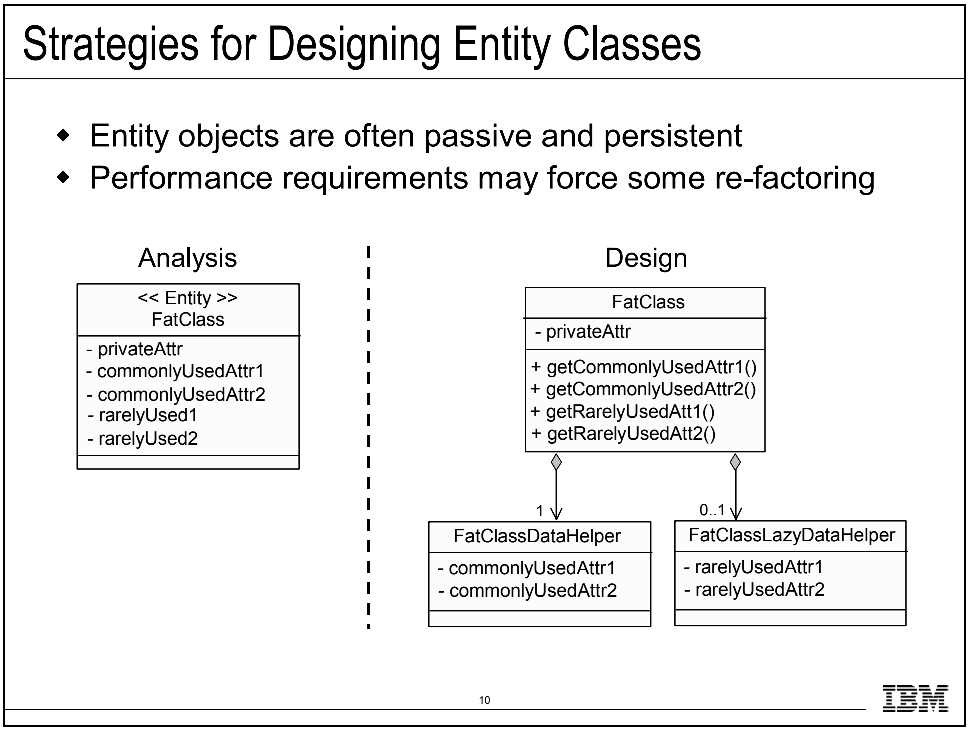
- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
- External system interface boundary classes
  - Usually model as subsystem

During Analysis, high-level boundary classes are identified. During Design, the design must be completed. You may need to create additional classes to support actual GUI and external system interactions.

Some mechanisms used to implement the UI can be architecturally significant. These should be identified and described by the architect in the Identify Design Mechanisms activity, and applied by the designer here in the **Class Design** activity. Design of user interface boundary classes will depend on the user interface development tools being used on the project. You only need design what the development environment will not automatically create for you. All GUI builders are different. Some create objects containing the information from the window. Others create data structures with the information. Any prototyping of the user interface done earlier is a good starting point for this phase of design.

Since interfaces to external systems usually have complex internal behavior, they are typically modeled as subsystems. If the interface is less complex, such as a pass through to an existing API, you may represent it with one or more design classes. In the latter case, use a single design class per protocol, interface, or API.

Strategies for Designing Entity Classes



Strategies for Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring

Analysis

<< Entity >>
FatClass
- privateAttr
- commonlyUsedAttr1
- commonlyUsedAttr2
- rarelyUsed1
- rarelyUsed2

Design

FatClass
- privateAttr
+ getCommonlyUsedAttr1()
+ getCommonlyUsedAttr2()
+ getRarelyUsedAtt1()
+ getRarelyUsedAtt2()

1

FatClassDataHelper
- commonlyUsedAttr1
- commonlyUsedAttr2

0..1

FatClassLazyDataHelper
- rarelyUsedAttr1
- rarelyUsedAttr2

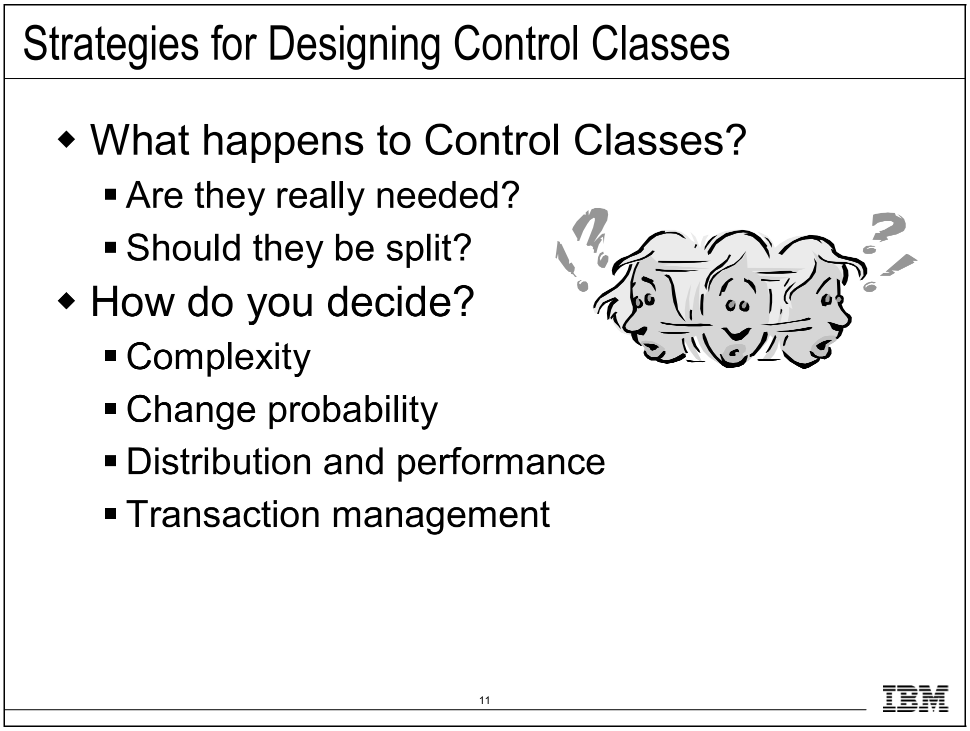
10

During Analysis, entity classes may have been identified and associated with the analysis mechanism for persistence, representing manipulated units of information. Performance considerations may force some re-factoring of persistent classes, causing changes to the Design Model that are discussed jointly between the database designer and the designer responsible for the class. The details of a database-based persistence mechanism are designed during Database Design, which is beyond the scope of this course.

Here we have a persistent class with five attributes. One attribute is not really persistent; it is used at runtime for bookkeeping. From examining the use cases, we know that two of the attributes are used frequently. Two other attributes are used less frequently. During Design, we decide that we'd like to retrieve the commonly used attributes right away, but retrieve the rarely used ones only if some client asks for them. We do not want to make a complex design for the client, so, from a data standpoint, we will consider the FatClass to be a proxy in front of two real persistent data classes. It will retrieve the FatClassDataHelper from the database when it is first retrieved. It will only retrieve the FatClassLazyDataHelper from the database in the rare occasion that a client asks for one of the rarely used attributes.

Such behind-the-scenes implementation is an important part of tuning the system from a data-oriented perspective while retaining a logical object-oriented view for clients to use.

Strategies for Designing Control Classes



Strategies for Designing Control Classes

- What happens to Control Classes?
  - Are they really needed?
  - Should they be split?
- How do you decide?
  - Complexity
  - Change probability
  - Distribution and performance
  - Transaction management

11

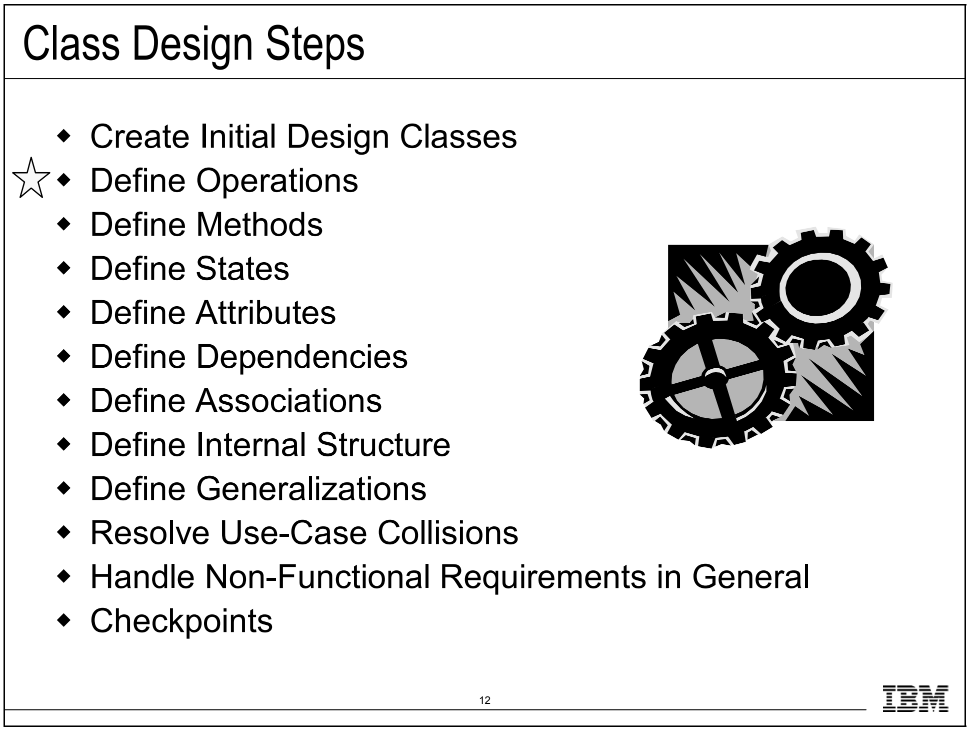
If control classes seem to be just "pass-throughs" from the boundary classes to the entity classes, they may be eliminated.

Control classes may become true design classes for any of the following reasons:

- They encapsulate significant control flow behavior.
- The behavior they encapsulate is likely to change.
- The behavior must be distributed across multiple processes and/or processors.
- The behavior they encapsulate requires some transaction management.

We saw in the Describe Distribution module how a single control class in Analysis became two classes in Design (a proxy and a remote). For our example, the control classes were needed in Design.

Define Operations



Class Design Steps

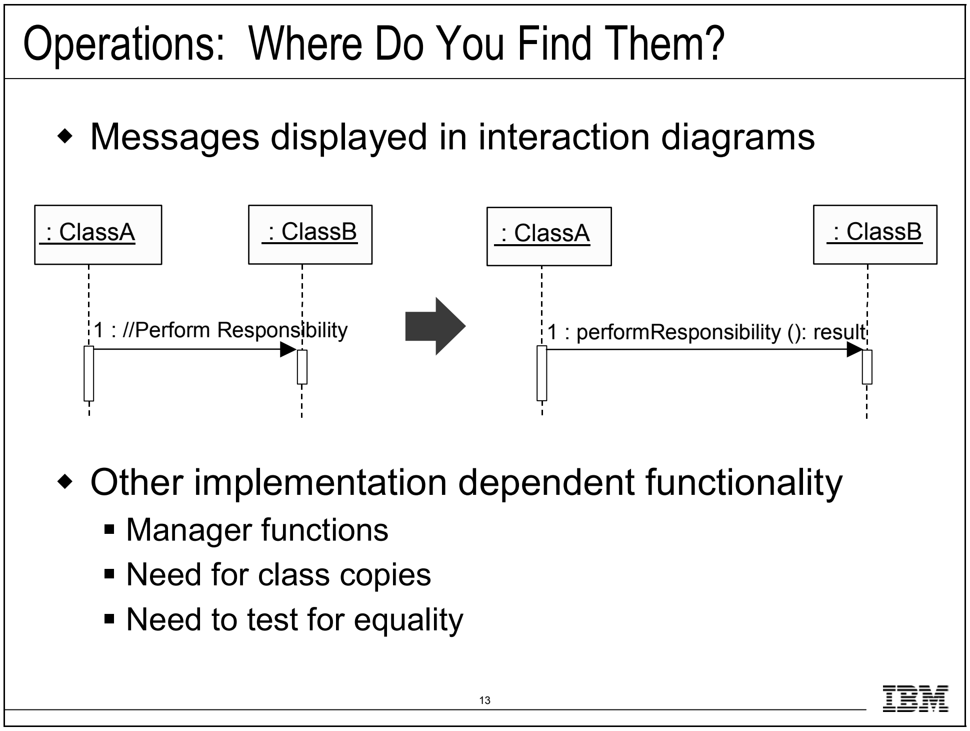
- Create Initial Design Classes
- ☆ Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

12

IBM

At this point, an initial set of design classes has been identified. Now we can turn our attention to finalizing the responsibilities that have been allocated to those classes.

Operations: Where Do You Find Them?



Operations: Where Do You Find Them?

- Messages displayed in interaction diagrams

: ClassA
: ClassB
1 : //Perform Responsibility
: ClassA
: ClassB
1 : performResponsibility (): result

- Other implementation dependent functionality
  - Manager functions
  - Need for class copies
  - Need to test for equality

13

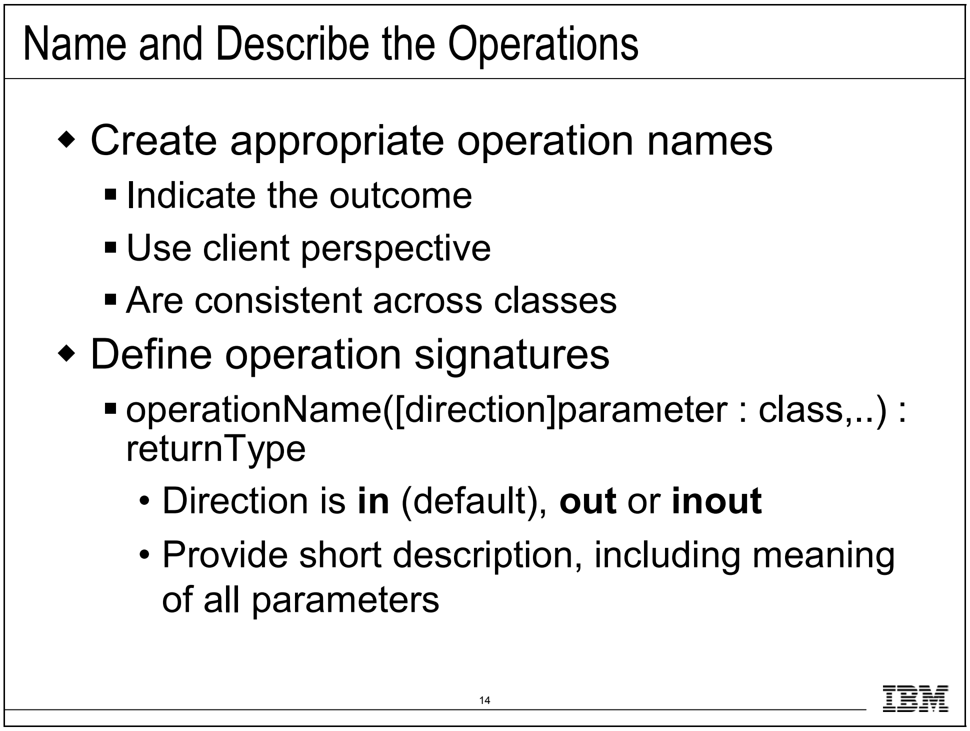
To identify operations on design classes:

- Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
- Study the Use-Case Realizations in the class participations to see how the operations are used by the Use-Case Realizations. Extend the operations, one Use-Case Realization at a time, refining the operations, their descriptions, return types and parameters. Each Use-Case Realization's requirements, as regards classes, are textually described in the Flow of Events of the Use-Case Realization.
- Study the use-case Special Requirements, to make sure that you do not miss implicit requirements on the operation that might be stated there.

Use-Case Realizations cannot provide enough information to identify all operations. To find the remaining operations, consider the following:

- Is there a way to initialize a new instance of the class, including connecting it to instances of other classes to which it is associated?
- Is there a need to test to see if two instances of the class are equivalent?
- Is there a need to create a copy of a class instance?
- Are any operations required on the class by mechanisms which they use? (Example: a “garbage collection” mechanism may require that an object be able to drop all of its references to all other objects in order for unused resources to be freed.)

Name and Describe the Operations



Name and Describe the Operations

- Create appropriate operation names
  - Indicate the outcome
  - Use client perspective
  - Are consistent across classes
- Define operation signatures
  - operationName([direction]parameter : class,..) : returnType
    - Direction is **in** (default), **out** or **inout**
    - Provide short description, including meaning of all parameters

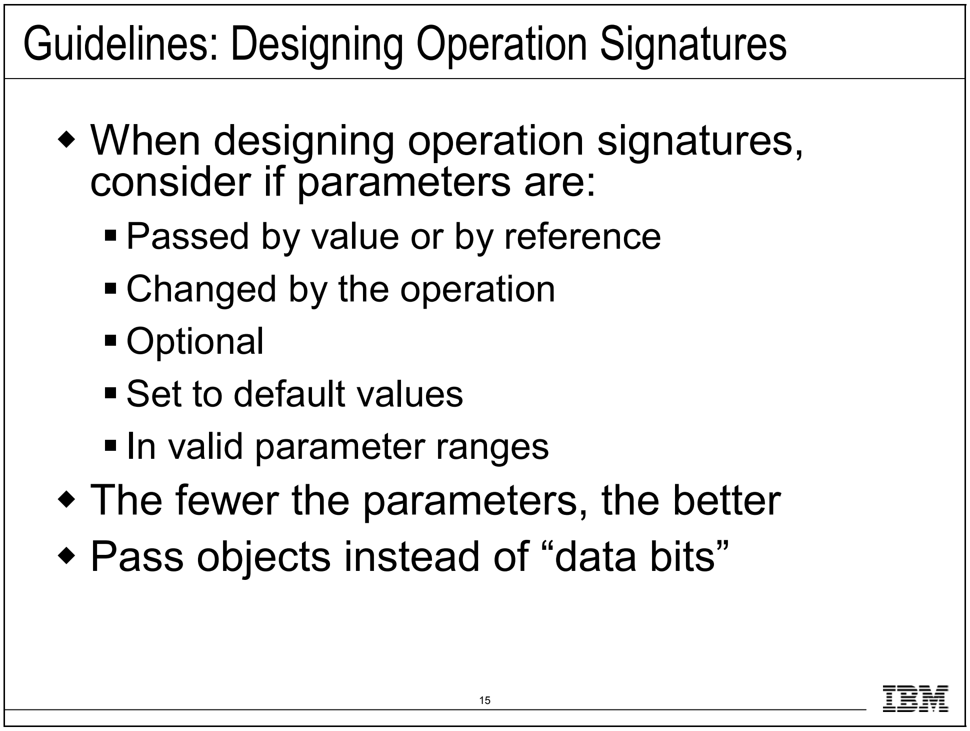
14

Operations should be named to indicate their outcome. For example, getBalance() versus calculateBalance(). One approach for naming operations that get and set properties is to simply name the operation the same name as the property. If there is a parameter, it sets the property; if not, it returns the current value.

You should name operations from the perspective of the client asking for a service to be performed by the class. For example, getBalance() versus receiveBalance(). The same applies to the operation descriptions. Descriptions should always be written from the operation USER's perspective. What service does the operation provide?

It is best to specify the operations and their parameters using implementation language syntax and semantics. This way the interfaces will already be specified in terms of the implementation language when coding starts.

Guidelines: Designing Operation Signatures



Guidelines: Designing Operation Signatures

- When designing operation signatures, consider if parameters are:
  - Passed by value or by reference
  - Changed by the operation
  - Optional
  - Set to default values
  - In valid parameter ranges
- The fewer the parameters, the better
- Pass objects instead of “data bits”

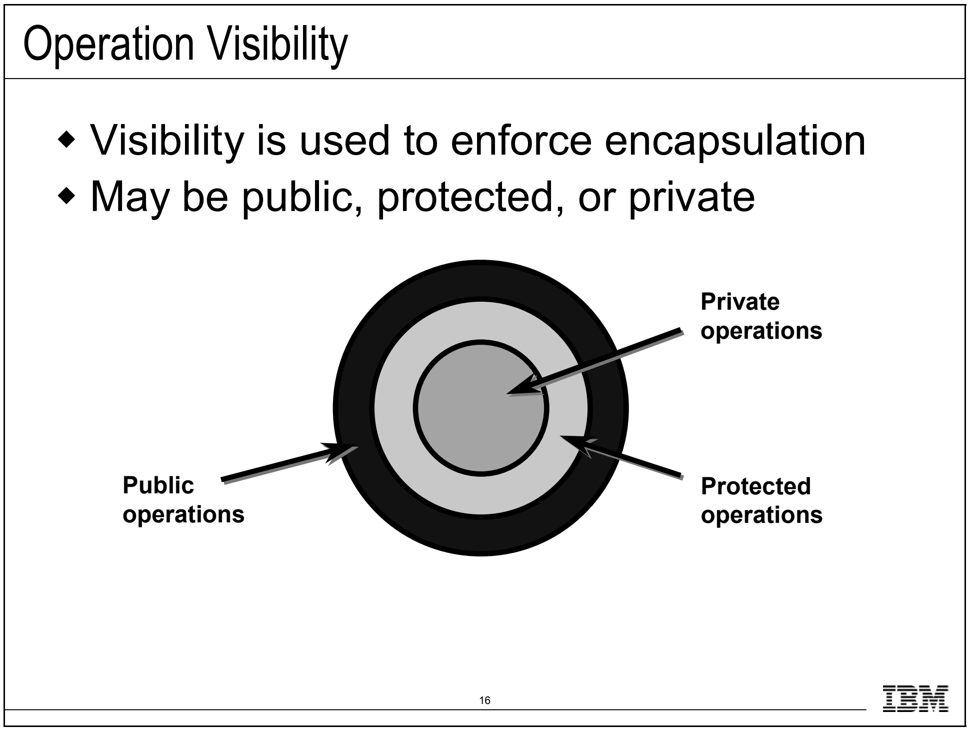
In addition to the short description of the parameter, be sure to include things like:

- Whether the parameter should be passed by value or by reference, and if by reference, is the parameter changed by the operation? By value means that the actual object is passed. By reference means that a pointer or reference to the object is passed.
  The signature of the operation defines the interface to objects of that class, and the parameters should therefore be designed to promote and define what that interface is. For example, if a parameter should never be changed by the operation, then design it so that it is not possible to change it — if the implementation environment supports that type of design.
- Whether parameters may be optional and/or have default values when no value is provided.
- Whether the parameter has ranges of valid values.

The fewer parameters you have, the better. Fewer parameters help to promote understandability, as well as maintainability. The more parameters clients need to understand, the more tightly coupled the objects are likely to be conceptually.

One of the strengths of OO is that you can manipulate very rich data structures, complete with associated behavior. Rather than pass around individual data fields (for example, StudentID), strive to pass around the actual object (for example, Student). Then the recipient has access to all the properties and behavior of that object.

Operation Visibility



Operation Visibility

- Visibility is used to enforce encapsulation
- May be public, protected, or private

Operation visibility is the realization of the key object-orientation principle of encapsulation.

Public members are accessible directly by any client.

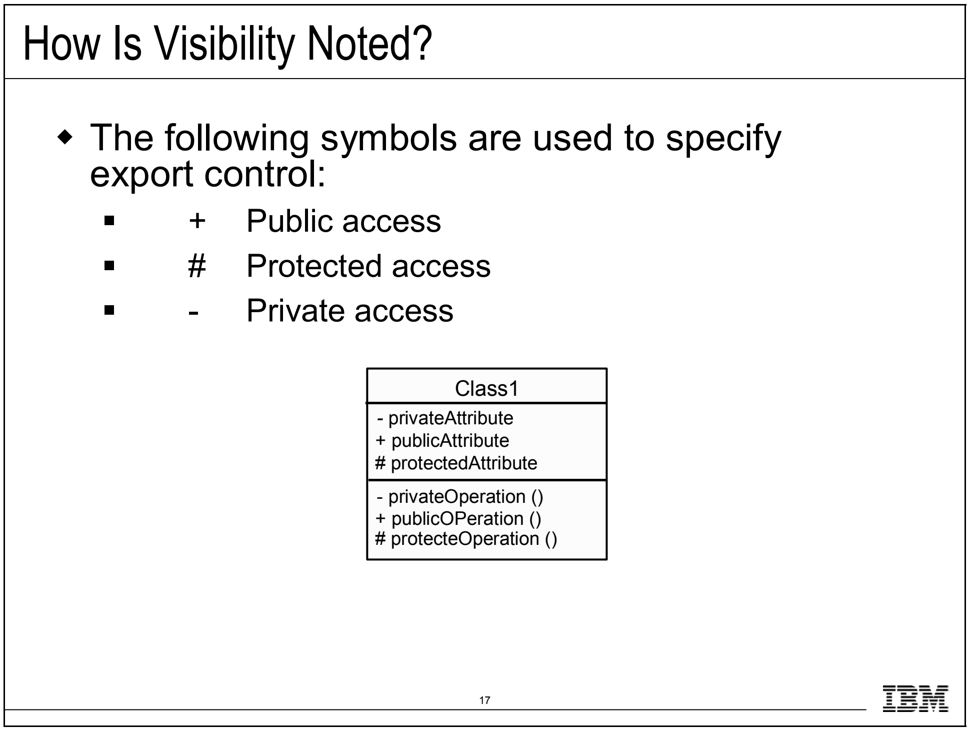
Protected members are directly accessible only by instances of subclasses.

Private members are directly accessible only by instances of the class to which they are defined.

How do you decide what visibility to use? Look at the Interaction diagrams on which the operation is referenced. If the message is from outside of the object, use public. If it is from a subclass, use protected. If it's from itself, use private. You should define the most restrictive visibility possible that will still accomplish the objectives of the class. Client access should be granted explicitly by the class and not taken forcibly.

Visibility applies to attributes as well as operations. Attributes are discussed later in this module.

How Is Visibility Noted?



How Is Visibility Noted?

- The following symbols are used to specify export control:
  - + Public access
  - # Protected access
  - - Private access

Class1

- privateAttribute
+ publicAttribute
# protectedAttribute

- privateOperation ()
+ publicOPeration ()
# protecteOperation ()

17

In the UML, you can specify the access clients have to attributes and operations.

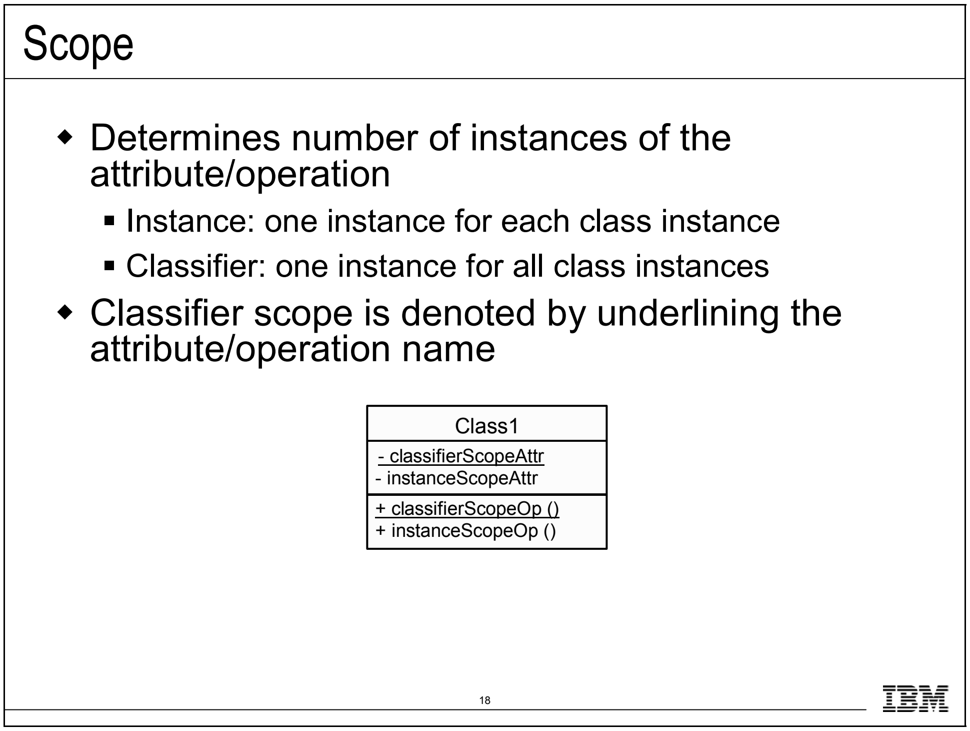
Export control is specified for attributes and operations by preceding the name of the member with the following symbols:

+ Public

# Protected

- Private

Scope



Scope

- Determines number of instances of the attribute/operation
  - Instance: one instance for each class instance
  - Classifier: one instance for all class instances
- Classifier scope is denoted by underlining the attribute/operation name

Class1

- classifierScopeAttr
- instanceScopeAttr

+ classifierScopeOp ()
+ instanceScopeOp ()

The owner scope of an attribute/operation determines whether or not the attribute/operation appears in each instance of the class (instance scoped), or if there is only one instance for all instances of the class (classifier scoped).

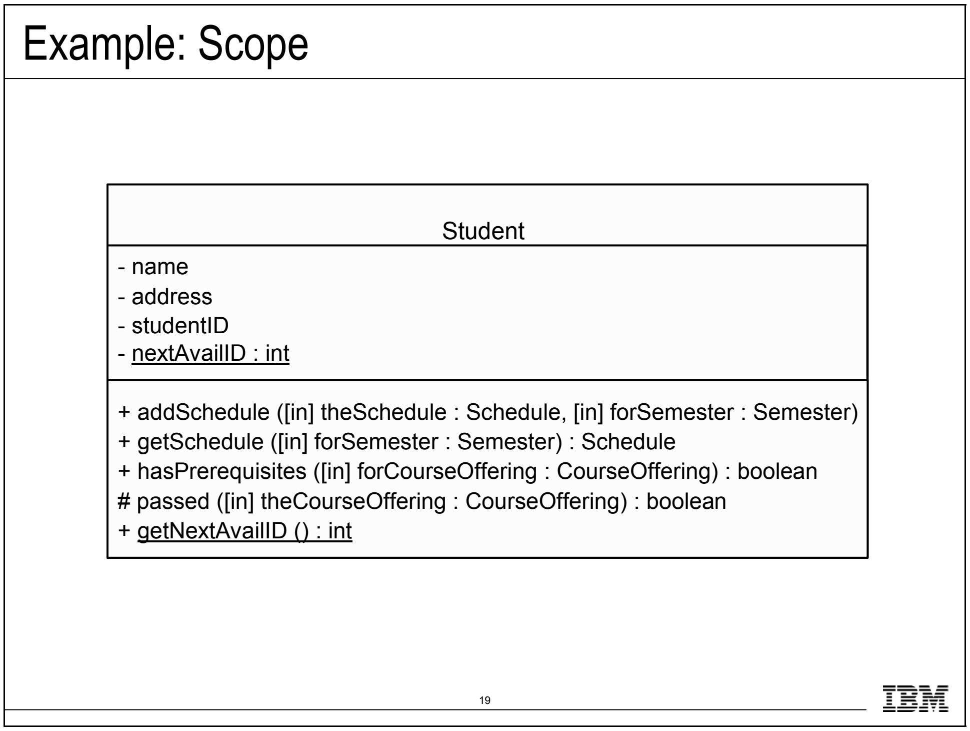
Classifier-scoped attributes and operations are denoted by underlining their names. Lack of an underline indicates instance-scoped attributes and operations.

Classifier-scoped attributes are shared among all instances of the classifier type.

In most cases, attributes and operations are instance scoped. However, if there is a need to have a single instance of an operation, say to generate a unique ID among class instances, or to create a class instance, the classifier scope operations can be used.

Classifier-scoped operations can only access classifier-scoped attributes.

Example: Scope



Example: Scope

Student

- name
- address
- studentID
- nextAvailID : int

+ addSchedule ([in] theSchedule : Schedule, [in] forSemester : Semester)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] theCourseOffering : CourseOffering) : boolean
+ getNextAvailID () : int

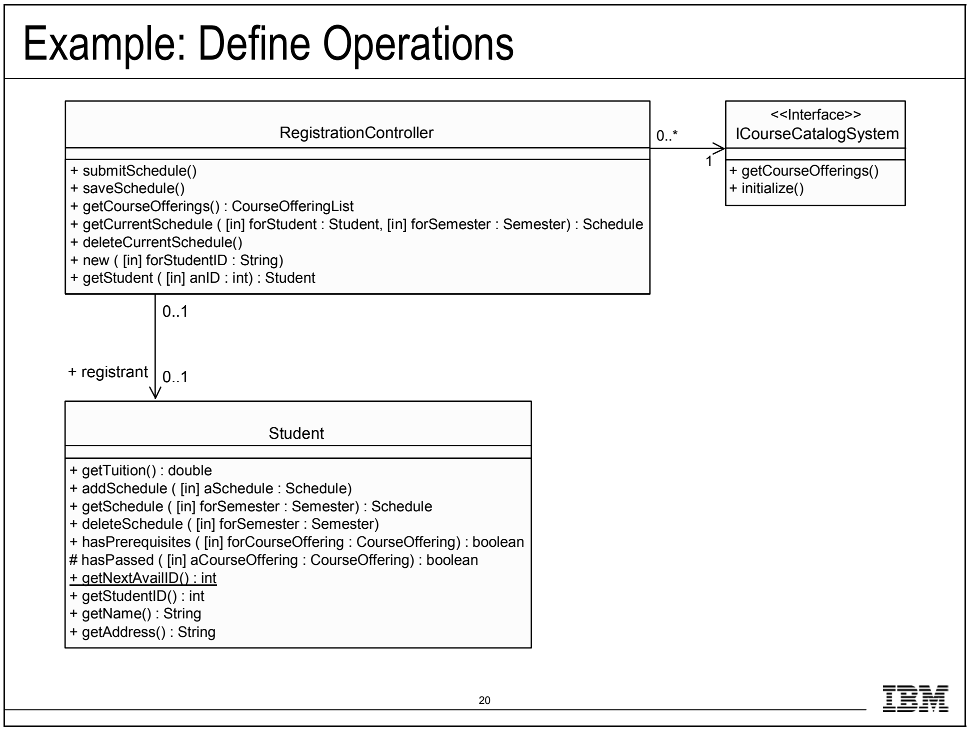
19

In the above example, there is a single classifier-scoped attribute, nextAvailID, and a single classifier-scoped operation, getNextAvailID(). These classifier-scoped class features support the generation of a unique ID for each Student.

Each Student instance has its own unique StudentID, whereas, there is only one nextAvailID for all Student instances.

The getNextAvailID() classifier-scoped operation can only access the classifier scope attribute nextAvailID.

Example: Define Operations



Example: Define Operations

RegistrationController

+ submitSchedule()
+ saveSchedule()
+ getCourseOfferings() : CourseOfferingList
+ getCurrentSchedule ([in] forStudent : Student, [in] forSemester : Semester) : Schedule
+ deleteCurrentSchedule()
+ new ([in] forStudentID : String)
+ getStudent ([in] anID : int) : Student

<<Interface>>
ICourseCatalogSystem

+ getCourseOfferings()
+ initialize()

Student

+ getTuition() : double
+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ deleteSchedule ([in] forSemester : Semester)
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# hasPassed ([in] aCourseOffering : CourseOffering) : boolean
+ getNextAvailID() : int
+ getStudentID() : int
+ getName() : String
+ getAddress() : String

20

This example is a portion of the VOPC for the Register for Courses Use-Case Realization.

Notice the <<class>> operations.

Those operations marked with a '+' are public and can be invoked by clients of the class.

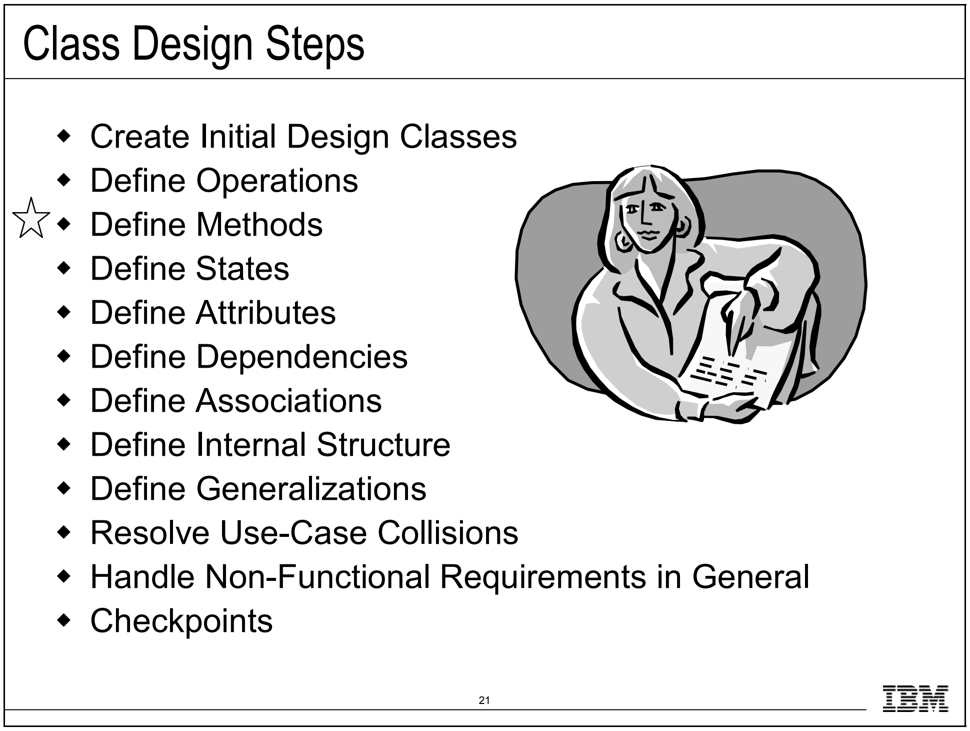
Those operations marked with a # are protected and can only be invoked by the defining class and any subclasses. These operations usually correspond to reflexive operations on the interaction diagrams.

The dependency from the Student class to the CourseOfferingClass was added to support the inclusion of the CourseOffering as a parameter to operations within the Student class.

Semester is included as the type for several parameters in the Student operations. For the Course Registration System, it is considered to be an abstract data type that has no significant behavior and thus is not modeled as a separate class.

Note: The attribute compartment has been suppressed in the above diagram.

Define Methods



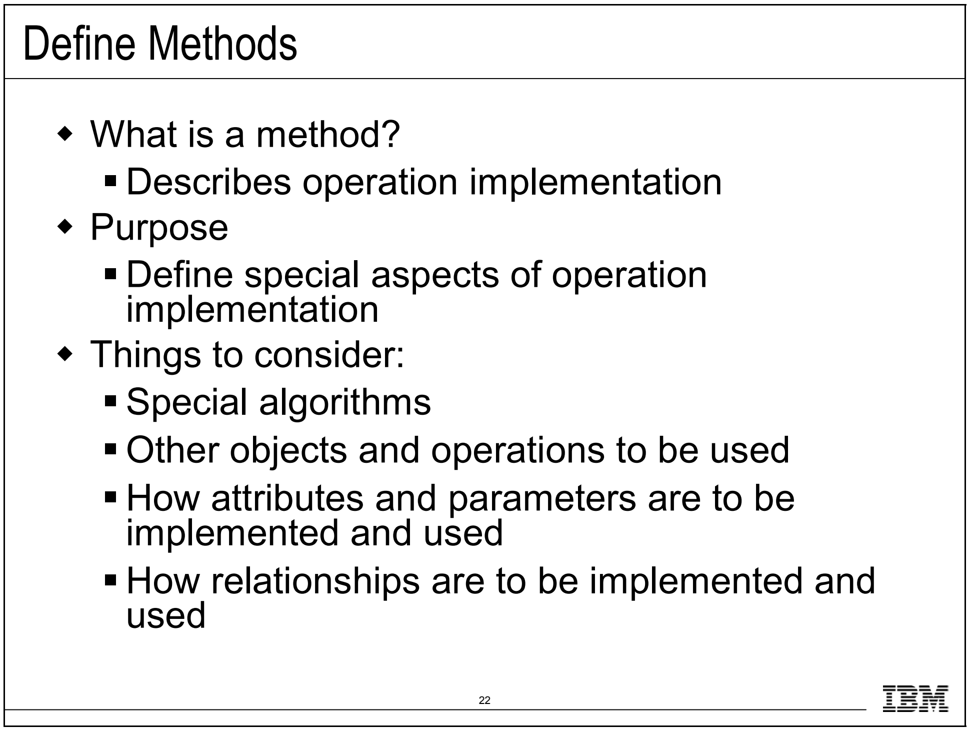
Class Design Steps

- Create Initial Design Classes
- Define Operations
- ☆ Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

21

Once the operations have been defined, some additional information may need to be documented for the class implementers.

Define Methods



Define Methods

- What is a method?
  - Describes operation implementation
- Purpose
  - Define special aspects of operation implementation
- Things to consider:
  - Special algorithms
  - Other objects and operations to be used
  - How attributes and parameters are to be implemented and used
  - How relationships are to be implemented and used

A method specifies the implementation of an operation. It describes *how* the operation works, not just *what* it does.

The method, if described, should discuss:

- How operations are to be implemented.
- How attributes are to be implemented and used to implement operations.
- How relationships are to be implemented and used to implement operations.

The requirements will naturally vary from case to case. However, the method specifications for a class should always state:

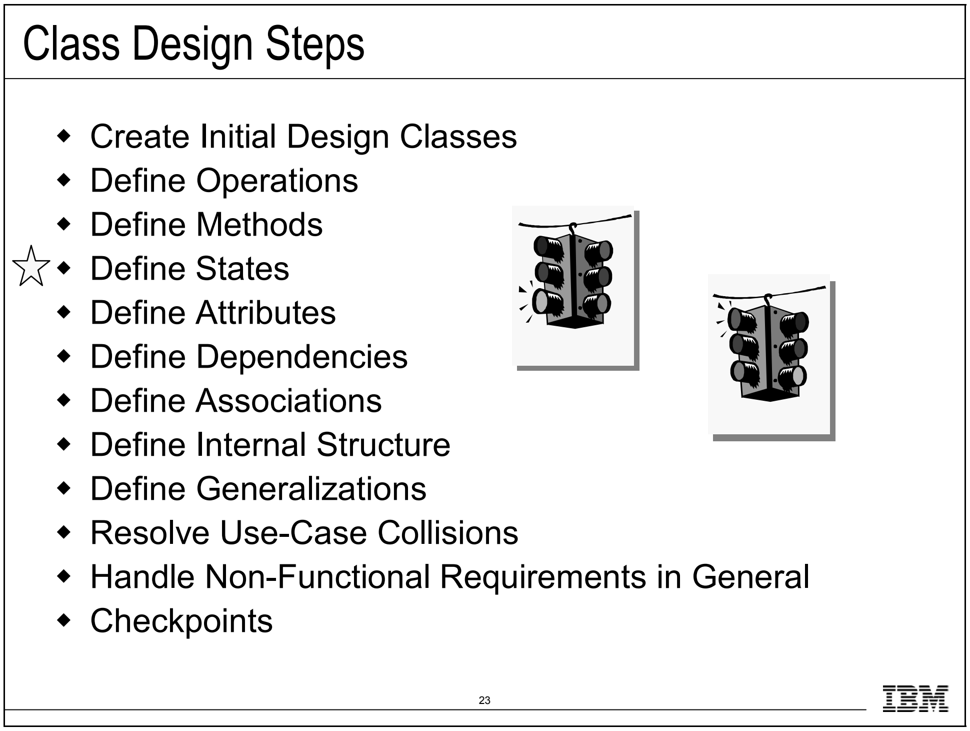
- What is to be done according to the requirements.
- What other objects and operations are to be used.

More specific requirements may concern:

- How parameters are to be implemented.
- Any special algorithms to be used.

In many cases, where the behavior required by the operation is sufficiently defined by the operation name, description and parameters, the methods are implemented directly in the programming language. Where the implementation of an operation requires use of a specific algorithm, or requires more information than is presented in the operation's description, a separate method description is required.

Define States



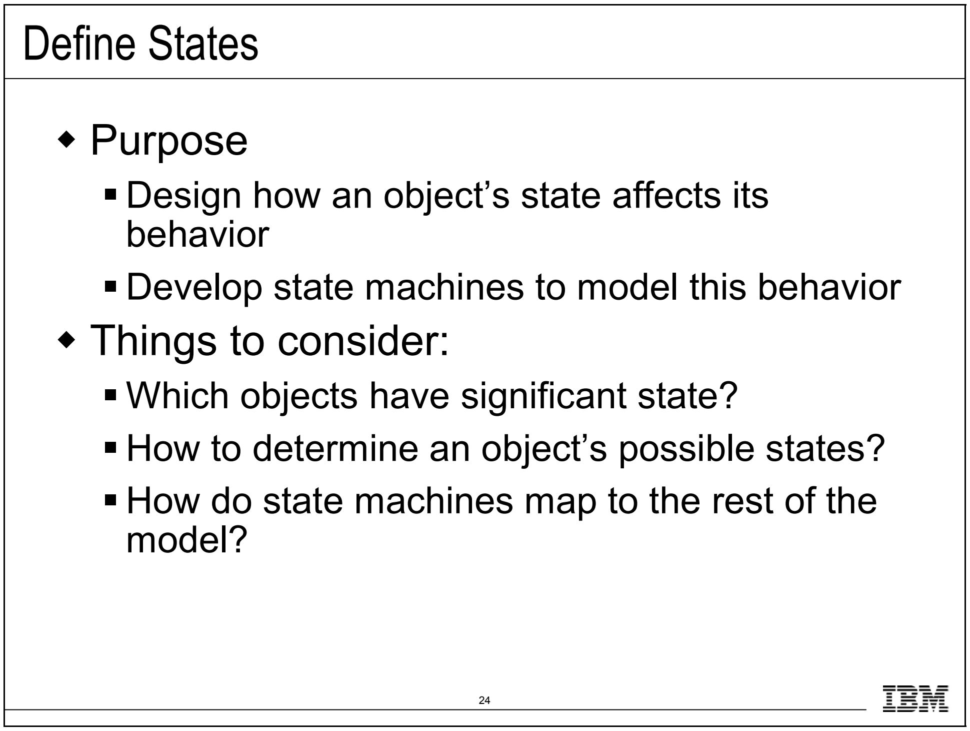
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- ☆ Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Now that we have a pretty clear understanding of the functionality provided by each of the classes, we can determine which of these classes have significant state behavior and model that behavior.

Note: State machines diagrams are specific to a class and are important during detailed design, so they are presented here in **Class Design.** However, state machines may be developed at any point in the software development lifecycle.

Define States



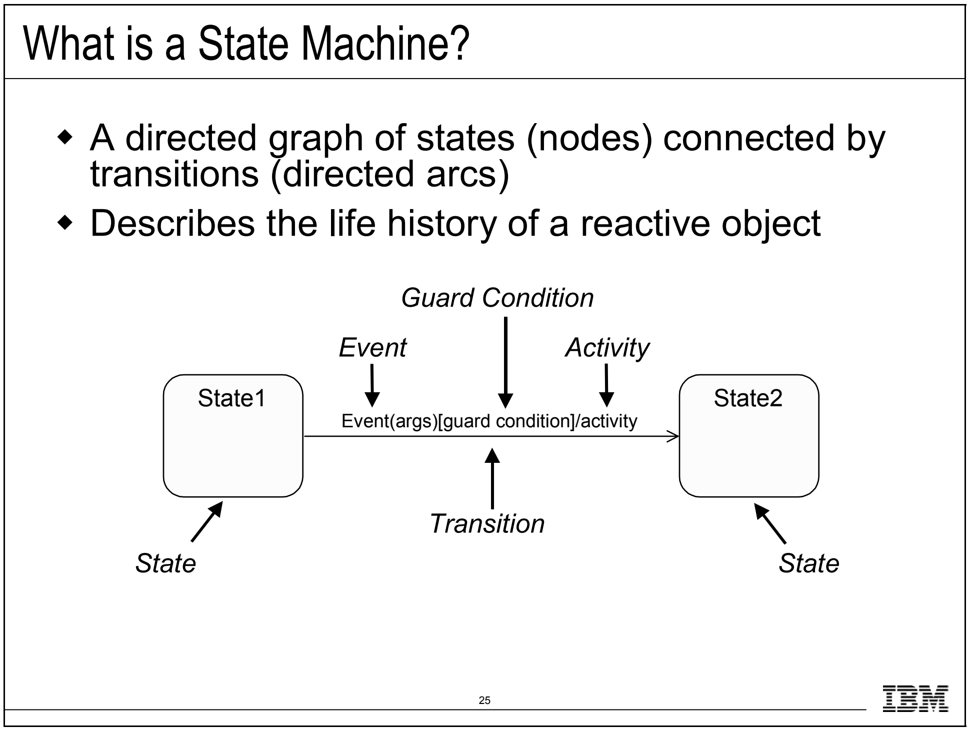
Define States

- Purpose
  - Design how an object's state affects its behavior
  - Develop state machines to model this behavior
- Things to consider:
  - Which objects have significant state?
  - How to determine an object's possible states?
  - How do state machines map to the rest of the model?

The state an object resides in is a computational state, and is defined by the stimuli the object can receive and what operations can be performed as a result. An object that can reside in many computational states is state-controlled.

For each class exhibiting state-controlled behavior, describe the relations between an object's states and an object's operations.

What is a State Machine?



What is a State Machine?

- A directed graph of states (nodes) connected by transitions (directed arcs)
- Describes the life history of a reactive object

Guard Condition

Event

Activity

State1

Event(args)[guard condition]/activity

State2

Transition

State

State

A state machine is a tool for describing:

- States the object can assume.
- Events that cause an object to transition from state to state.
- Significant activities and actions that occur as a result.

A state machine is a diagram used to show the life history of a given class, the events that cause a transition from one state to another, and the actions that result from a state change. State machines emphasize the event-ordered behavior of a class instance.

The state space of a given class is the enumeration of all the possible states of an object.

A **state** is a condition in the life of an object. The state of an object determines its response to different events.

An **event** is a specific occurrence (in time and space) of a stimulus that can trigger a state transition.

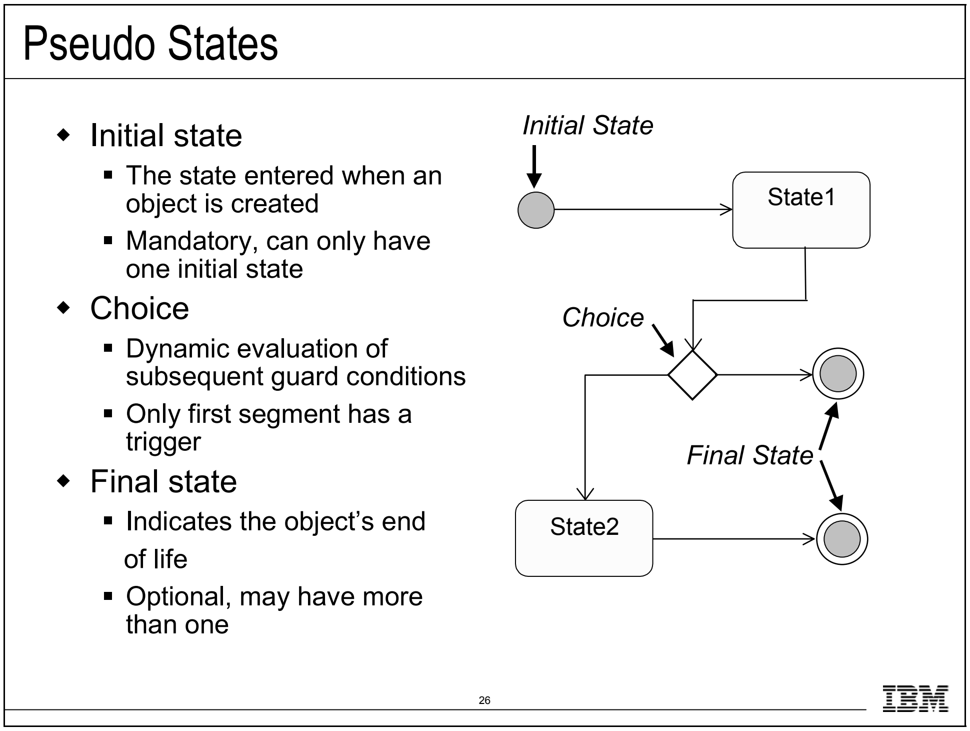
A **transition** is a change from an originating state to a successor state as a result of some stimulus. The successor state could possibly be the originating state. A transition may take place in response to an event, and can be labeled with an event.

A **guard condition** is a Boolean expression of attribute values that allows a transition only if the condition is true.

An **action** is an atomic execution that results in a change in state, or the return of a value.

An **activity** is a non-atomic execution within a state machine.

Pseudo States



Pseudo States

- Initial state
  - The state entered when an object is created
  - Mandatory, can only have one initial state
- Choice
  - Dynamic evaluation of subsequent guard conditions
  - Only first segment has a trigger
- Final state
  - Indicates the object's end of life
  - Optional, may have more than one

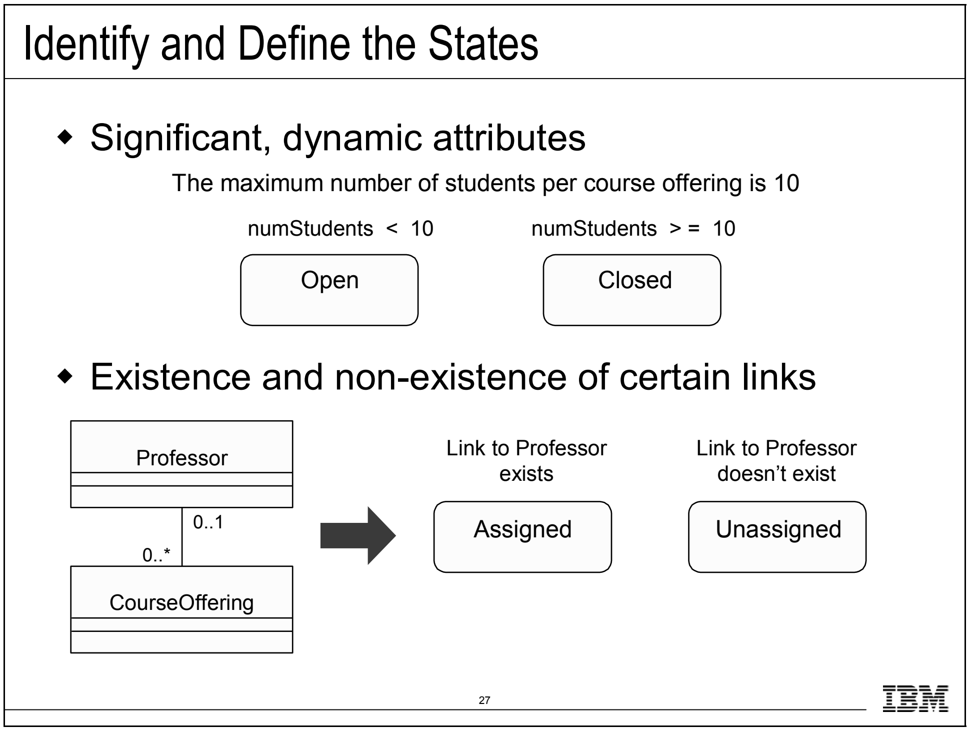
The initial state is the state entered when an object is created

- An initial state is mandatory.
- Only one initial state is permitted.
- The initial state is represented as a solid circle.

A final state indicates the end of life for an object

- A final state is optional.
- More than one final state may exist.
- A final state is indicated by a bull's eye.

Identify and Define the States



Identify and Define the States

- Significant, dynamic attributes

The maximum number of students per course offering is 10

numStudents < 10

Open

numStudents > = 10

Closed

- Existence and non-existence of certain links

Professor

0..1

0..*

CourseOffering

Link to Professor exists

Assigned

Link to Professor doesn’t exist

Unassigned

27

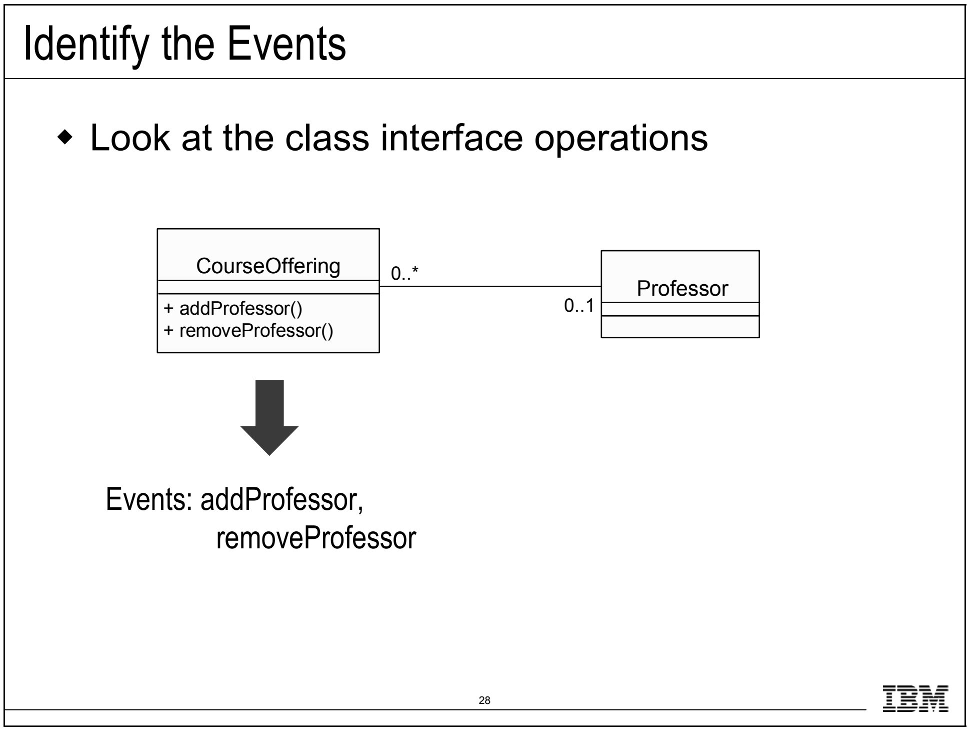
The states of an object can be found by looking at its class’ attributes and relationships with other classes.

Do not forget to establish the initial and final states for the object. If there are pre-conditions or post-conditions of the initial and final states, define those as well.

It is important not only to identify the different states, but also to explicitly define what it means to be in a particular state.

The above example demonstrates two states of a CourseOffering class instance. A CourseOffering instance may have a Professor assigned to teach it or not (hence the multiplicity of 0..1 on the Professor end of the CourseOffering-Professor association). If a Professor has been assigned to the CourseOffering instance, the state of the CourseOffering instance is “Assigned.” If a Professor has not been assigned to the CourseOffering instance, the state of the CourseOffering instance is “Unassigned.”

Identify the Events



Identify the Events

- Look at the class interface operations

CourseOffering

+ addProfessor()
+ removeProfessor()

0..*

0..1

Professor

Events: addProfessor,
removeProfessor

Determine the events to which the object responds. These can be found in the object's interfaces or protocols.

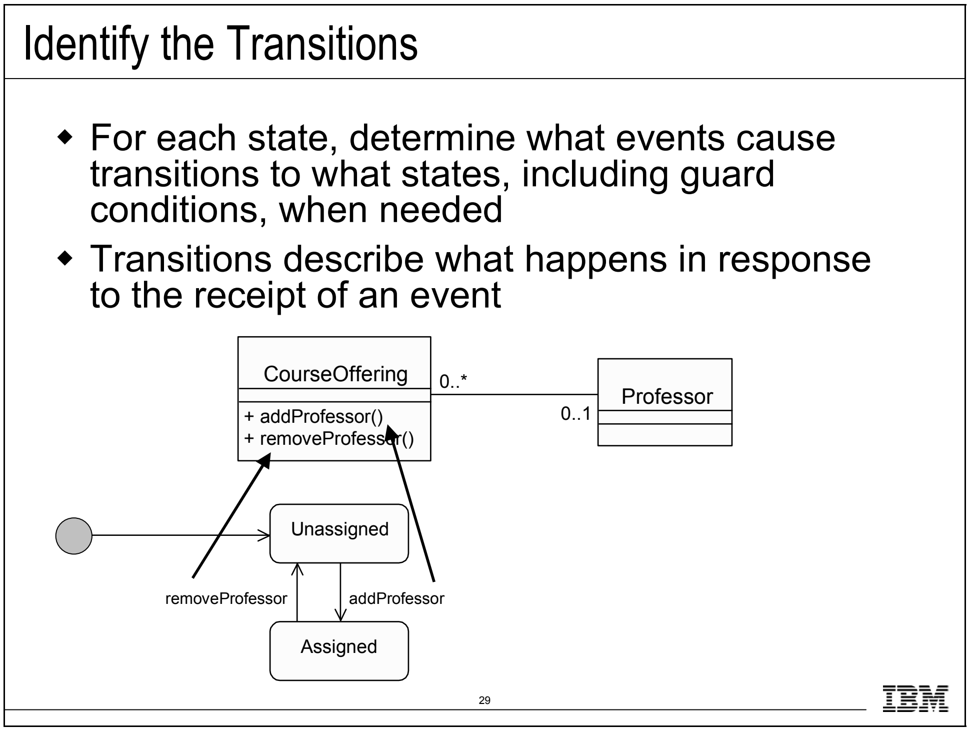
The class must respond to all messages coming into the class instances on all of the interaction diagrams. These messages should correspond to operations on the associated classes. Thus, looking at the class interface operations provides an excellent source for the events the class instance must respond to.

In the above example, two of the CourseOffering operations are addProfessor() and removeProfessor. Each of these operations can be considered an event that a CourseOffering instance must respond to.

Note: A subset of the CourseOffering behavior is shown above.

Events are not operations, although they often map 1 to 1. Remember: Events and operations are not the same thing.

Identify the Transitions



Identify the Transitions

- For each state, determine what events cause transitions to what states, including guard conditions, when needed
- Transitions describe what happens in response to the receipt of an event

CourseOffering

+ addProfessor()
+ removeProfessor()

0..*

0..1

Professor

Unassigned

removeProfessor

addProfessor

Assigned

29

From the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding these transitions.

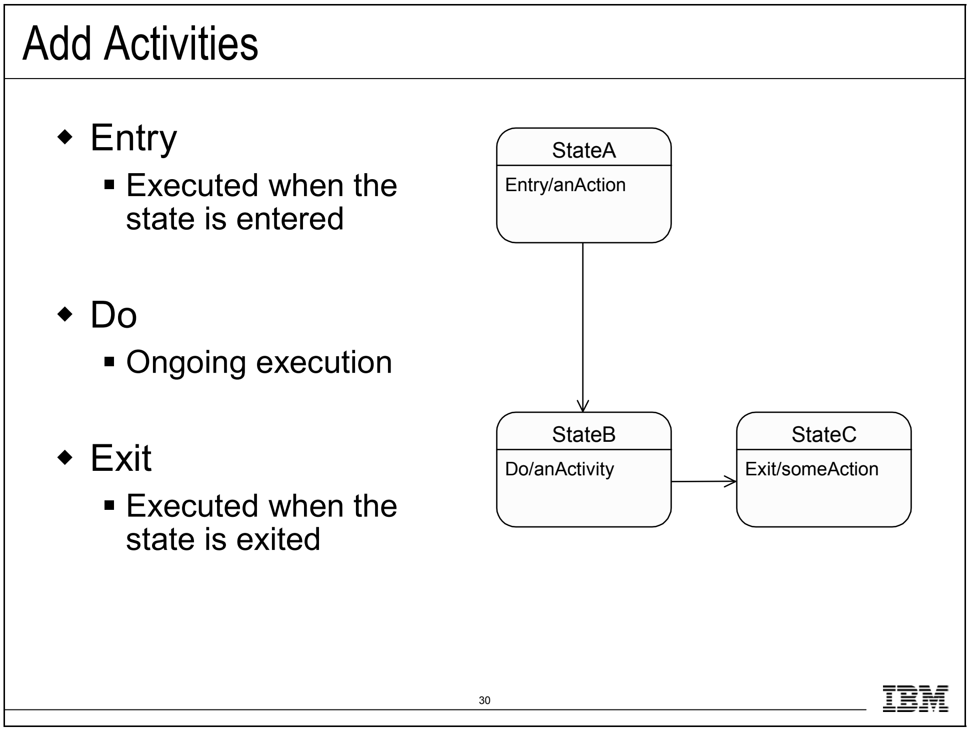
If there are multiple automatic transitions, each transition needs a guard condition. The conditions must be mutually exclusive.

Each state transition event can be associated with an operation. Depending on the object's state, the operation may have a different behavior. The transition events describe how this occurs.

In the above example, when a CourseOffering instance is Unassigned (meaning a Professor has not been assigned to teach it yet), and the instance receives an addProfessor event, the CourseOffering instance transitions into the Assigned state. Conversely, when a CourseOffering instance is Assigned (meaning a Professor has been assigned to teach it), and the instance receives a “removeProfessor” event, the CourseOffering instance transitions into the Unassigned state.

Note: A subset of the CourseOffering behavior is shown above.

Add Activities



Add Activities

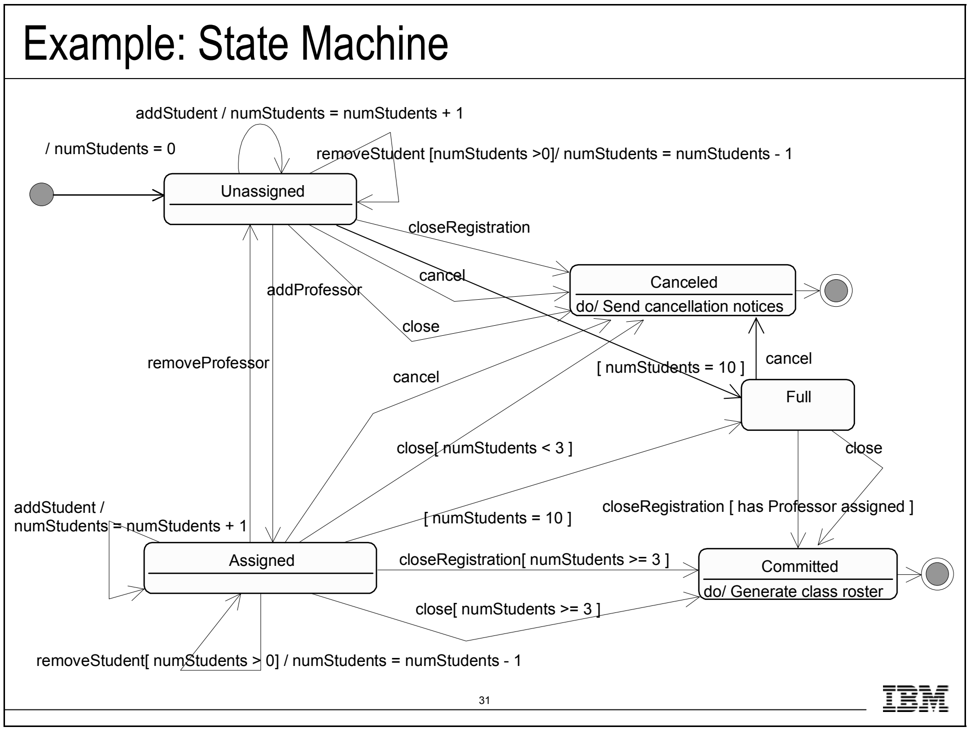
- Entry
  - Executed when the state is entered
- Do
  - Ongoing execution
- Exit
  - Executed when the state is exited

Entry Activity – Executed when the state is entered, after any activity associated with an incoming transition and before any internal Do activity.

Do Activity – An ongoing activity that is executed as long as the state is active.

Exit Activity – Executed when the state is exited, after completion of any internal Do activity and before any activity associated with an outgoing transition.

Example: State Machine



Example: State Machine

addStudent / numStudents = numStudents + 1
/ numStudents = 0
removeStudent [numStudents >0]/ numStudents = numStudents - 1
Unassigned
closeRegistration
cancel
Canceled
do/ Send cancellation notices
addProfessor
close
removeProfessor
cancel
[numStudents = 10]
cancel
Full
close[numStudents < 3]
close
addStudent /
numStudents = numStudents + 1
[numStudents = 10]
closeRegistration [has Professor assigned]
Assigned
closeRegistration[numStudents >= 3]
Committed
do/ Generate class roster
close[numStudents >= 3]
removeStudent[numStudents > 0] / numStudents = numStudents - 1
31

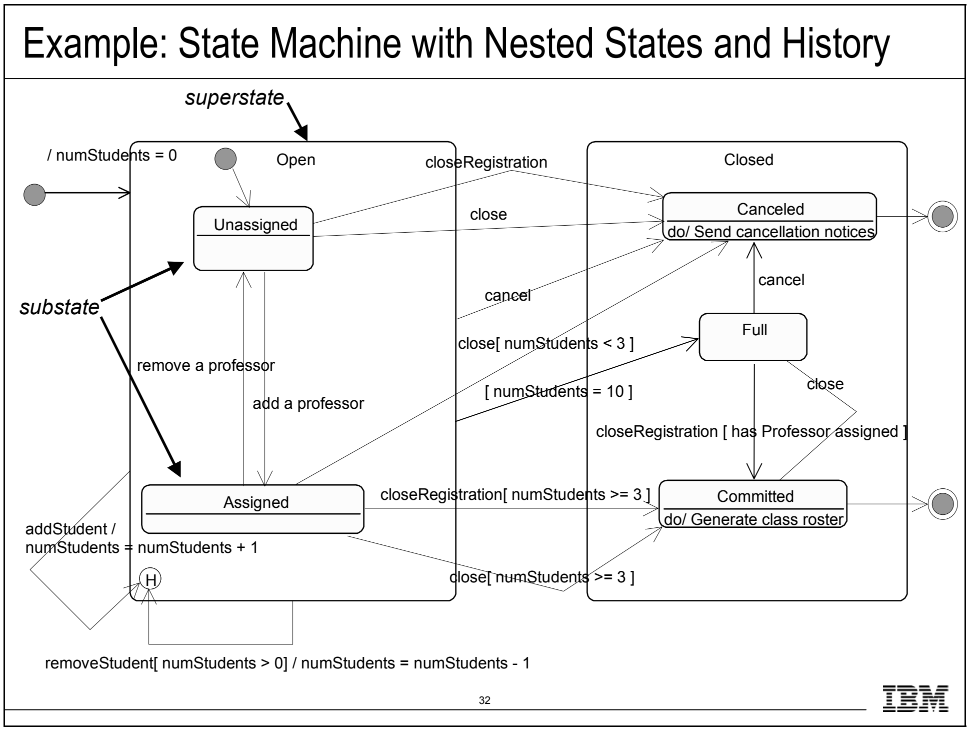
The above is a state machine for the CourseOffering class. Here are some things to note about it:

- A student can be added or removed from the course offering when the course offering is in the assigned or unassigned state.
- The closing of a course occurs in two phases:

Close registration is where course offerings are committed if they have a professor and enough students or canceled if there is no professor. At this point, the course offering is not closed due to low enrollment because during schedule leveling, students may be added or removed from the course offering. Leveling is where it is verified that the maximum number of selected course offerings have been committed by selecting alternates, where necessary.

Close is where the final status of a course offering is determined. If there is a professor and there are enough students, the course offering is committed. If not, the course offering is canceled. This is meant to occur after leveling.

Example: State Machine with Nested States and History



Example: State Machine with Nested States and History

superstate
/ numStudents = 0
Open
closeRegistration
Closed
Canceled
do/ Send cancellation notices
Unassigned
close
substate
cancel
cancel
Full
close[numStudents < 3]
remove a professor
[numStudents = 10]
close
add a professor
closeRegistration [has Professor assigned]
Assigned
closeRegistration[numStudents >= 3]
Committed
do/ Generate class roster
addStudent /
numStudents = numStudents + 1
H
close[numStudents >= 3]
removeStudent[numStudents > 0] / numStudents = numStudents - 1
32
IBM

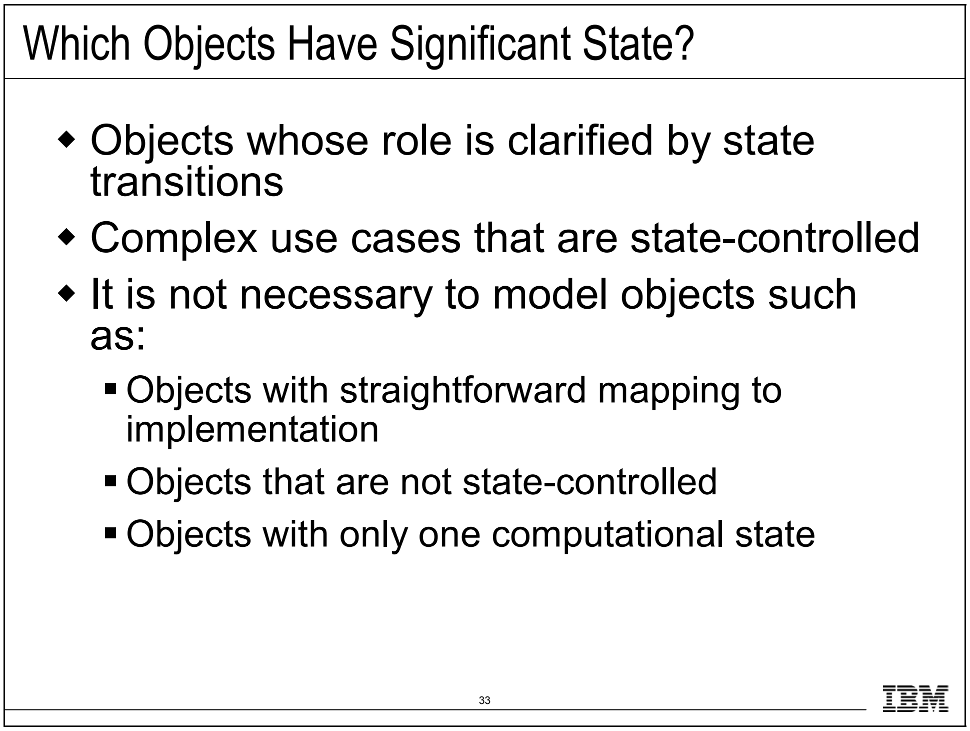
State machines can become unmanageably large and complex. Nested states may be used to simplify complex diagrams. A **superstate** is a state that encloses nested states called **substates**. Common transitions of the substates are represented at the level of the superstate. Any number of levels of nesting is permitted.

The **history** indicator supports the modeling of "memorized" states. A history indicator (circled H) is placed in the state region whose last executed state needs to be "remembered" after an activity has been interrupted. The use of the history feature indicates that upon return to a superstate, the most recently visited substate will be entered.

The history indicator is a "pseudo-state," similar to the stop state. States that make use of history must have their transition arrows going to the history pseudo-state. Transitions to the history indicator cause the last state that was executed in the enclosing state region to be resumed. Transitions to the outer boundary of the state region cause a transition back to the start state of the region. If the history feature is not used, the initial substate will always be entered when the superstate is entered.

The above is the CourseOffering class state machine where nested states with history have been used to simplify the diagram. The Open superstate was created to leverage the common transitions of the Unassigned and Assigned states. The use of the history indicator demonstrates that when the common transitions occur, the CourseOffering returns to the substate that it was in when the event was received.

Which Objects Have Significant State?



Which Objects Have Significant State?

- Objects whose role is clarified by state transitions
- Complex use cases that are state-controlled
- It is not necessary to model objects such as:
  - Objects with straightforward mapping to implementation
  - Objects that are not state-controlled
  - Objects with only one computational state

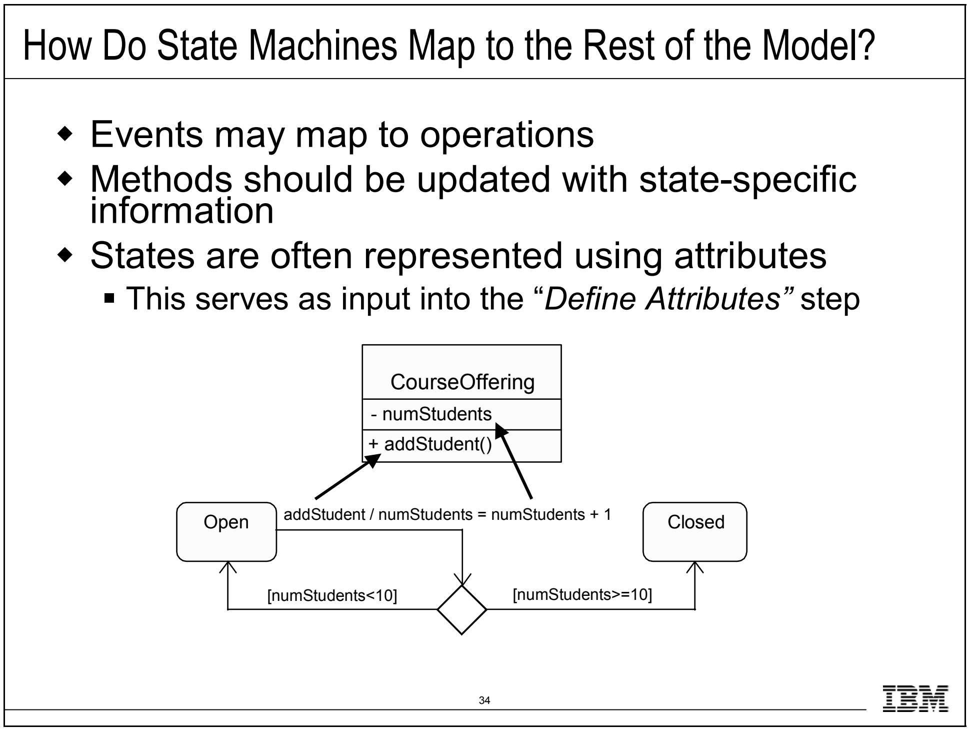
33

The state an object resides in is a computational state. It is defined by the stimuli the object can receive and what operations can be performed as a result. An object that can reside in many computational states is state-controlled.

The complexity of an object depends on:

- The number of different states.
- The number of different events it reacts to.
- The number of different actions it performs that depend on its state.
- The degree of interaction with its environment (other objects).
- The complexity of conditional, repetitive transitions.

How Do State Machines Map to the Rest of the Model?



How Do State Machines Map to the Rest of the Model?

- Events may map to operations
- Methods should be updated with state-specific information
- States are often represented using attributes
  - This serves as input into the “*Define Attributes”* step

Some state transition events can be associated with an operation. Depending on the object's state, the operation may have a different behavior. The transition events describe how this occurs.

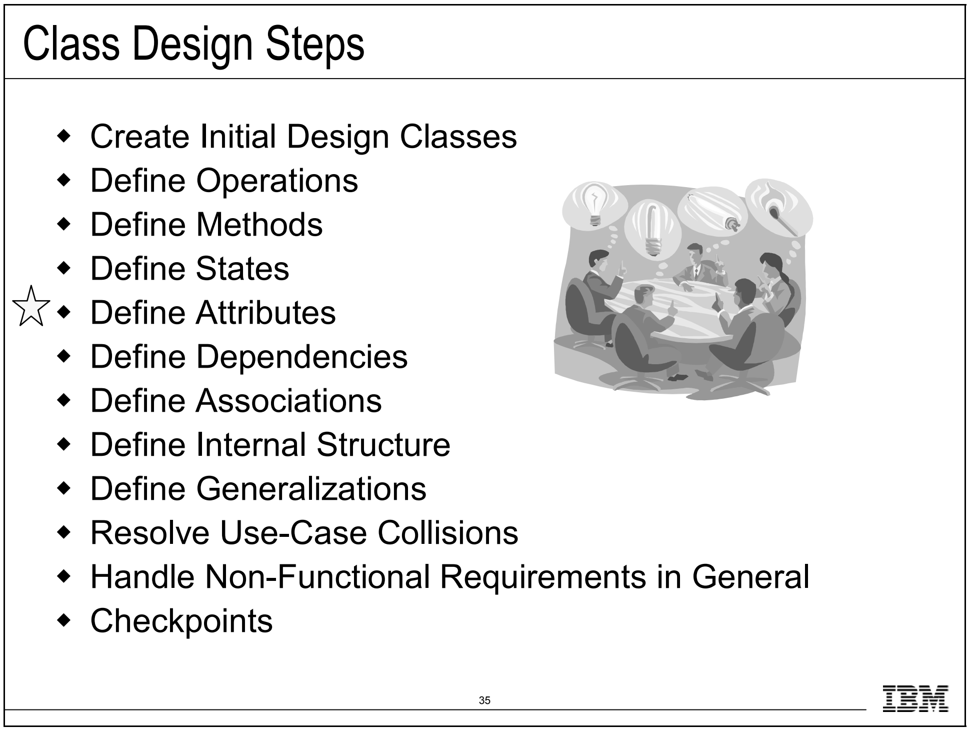
The method description for the associated operation should be updated with the state-specific information, indicating what the operation should do for each relevant state.

Operation calls are not the only source of events. In the UML, you can model four different kinds of events:

- Signals
- Calls
- Passing of time
- Change in state

States are often represented using attributes. The state machines serve as input into the attribute identification step.

Define Attributes

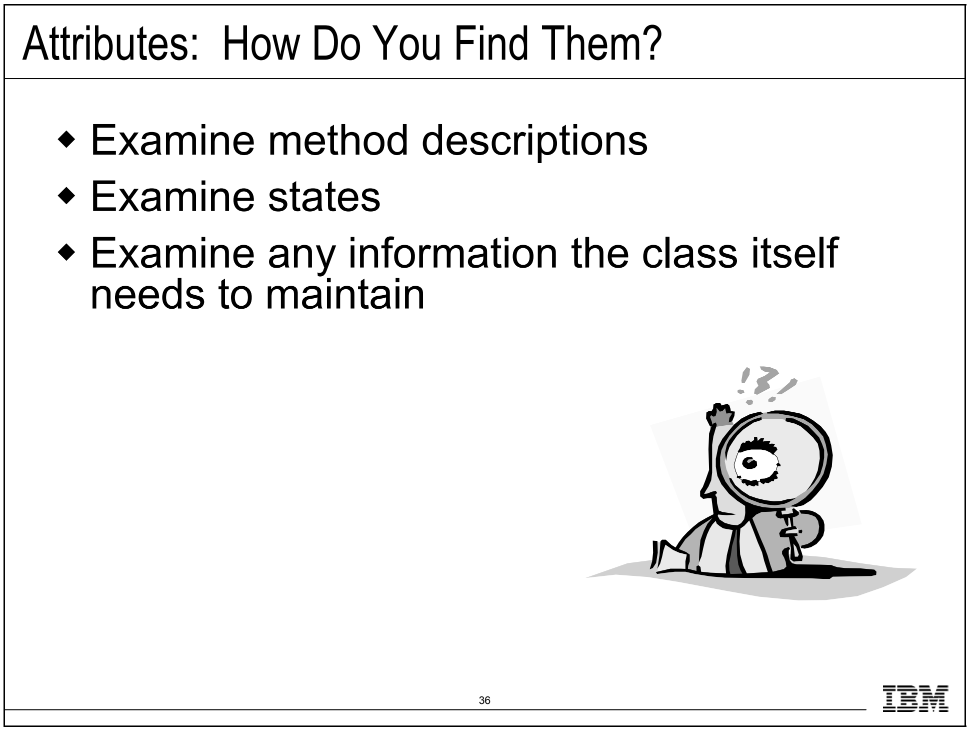


Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- ☆ Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

At this point, we have a pretty good understanding of the design classes, their functionality, and their states. All of this information affects what attributes are defined.

Attributes: How Do You Find Them?



Attributes: How Do You Find Them?

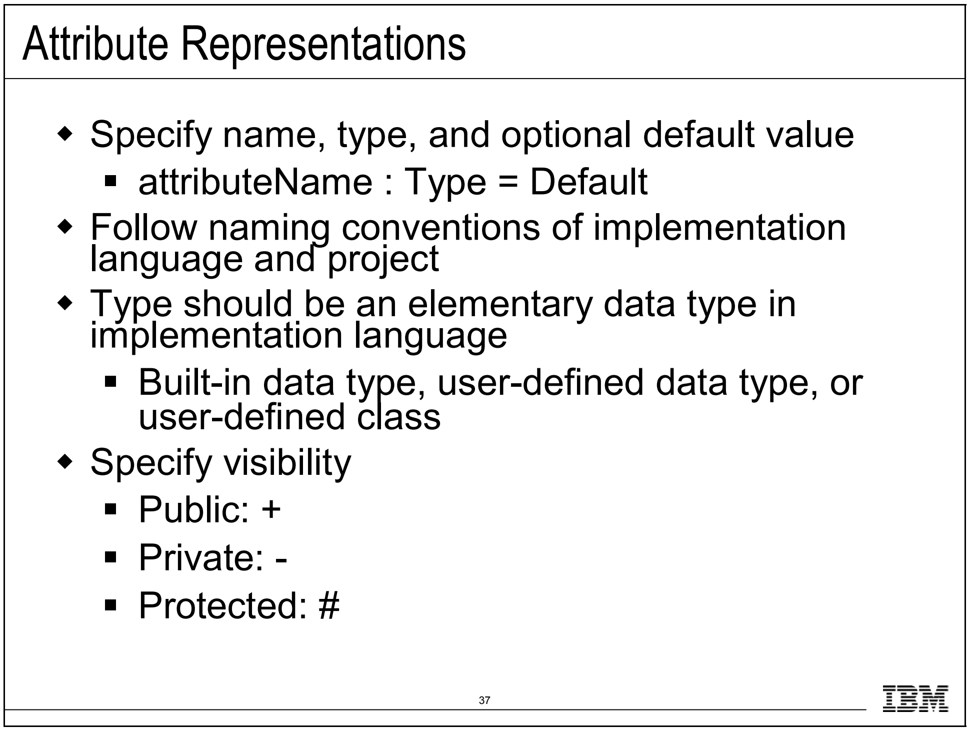
- Examine method descriptions
- Examine states
- Examine any information the class itself needs to maintain

Attributes the class needs to carry out its operations and the states of the class are identified during the definition of methods. Attributes provide information storage for the class instance and are often used to represent the state of the class instance. Any information the class itself maintains is done through its attributes.

You may need to add additional attributes to support the implementation and establish any new relationships that the attributes require.

Check to make sure all attributes are needed. Attributes should be justified — it is easy for attributes to be added early in the process and survive long after they are no longer needed due to shortsightedness. Extra attributes, multiplied by thousands or millions of instances, can have a large effect on the performance and storage requirements of the system.

Attribute Representations



Attribute Representations

- Specify name, type, and optional default value
  - attributeName : Type = Default
- Follow naming conventions of implementation language and project
- Type should be an elementary data type in implementation language
  - Built-in data type, user-defined data type, or user-defined class
- Specify visibility
  - Public: +
  - Private: -
  - Protected: #

37

IBM

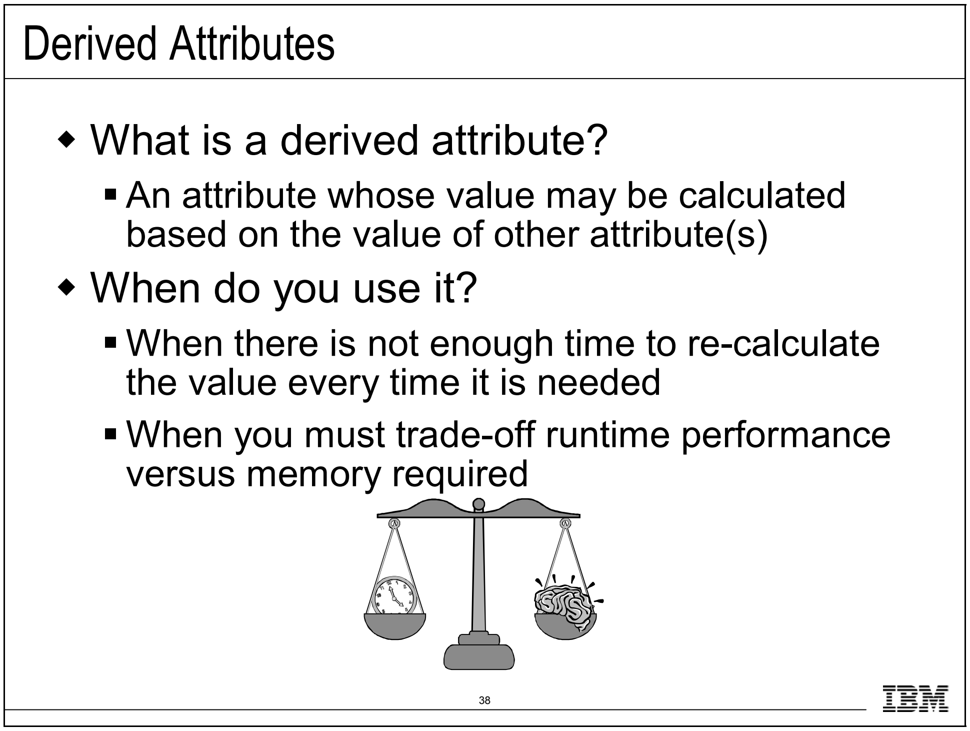
In analysis, it was sufficient to specify the attribute name only, unless the representation was a requirement that was to constrain the designer.

During Design, for each attribute, define the following:

- Its **name**, which should follow the naming conventions of both the implementation language and the project.
- Its **type**, which will be an elementary data type supported by the implementation language.
- Its **default or initial value**, to which it is initialized when new instances of the class are created.
- Its **visibility**, which will take one of the following values:
  - **Public**: The attribute is visible both inside and outside the package containing the class.
  - **Protected**: The attribute is visible only to the class itself, to its subclasses, or to **friends** of the class (language dependent)
  - **Private**: The attribute is visible only to the class itself and to **friends** of the class.

For persistent classes, also include whether the attribute is persistent (the default) or transient. Even though the class itself may be persistent, not all attributes of the class need to be persistent.

Derived Attributes

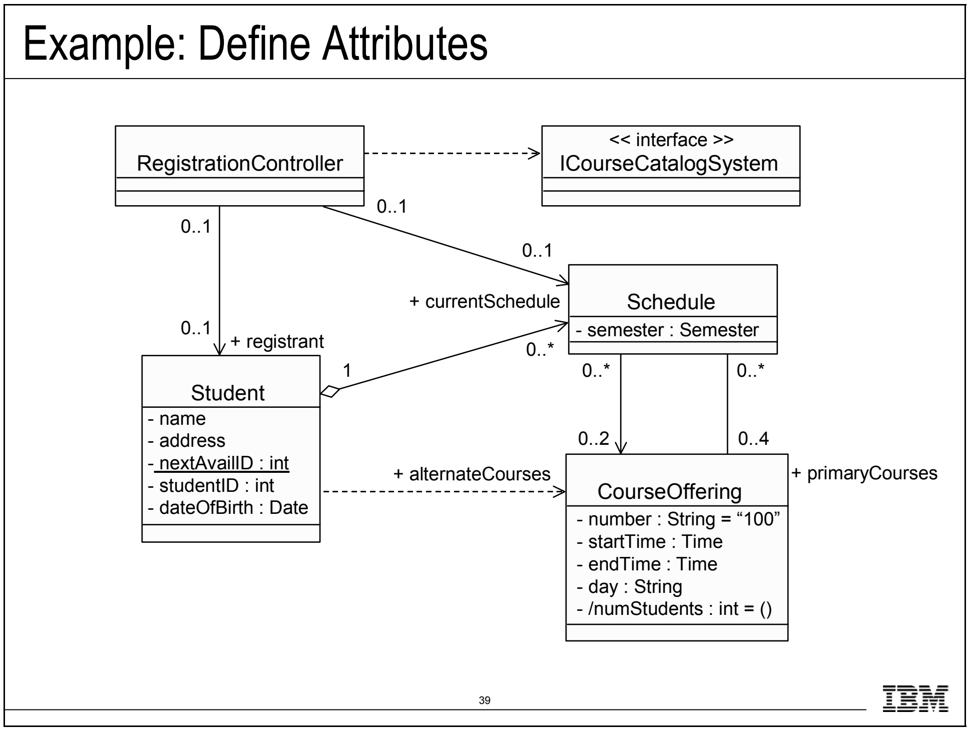


Derived Attributes

- What is a derived attribute?
  - An attribute whose value may be calculated based on the value of other attribute(s)
- When do you use it?
  - When there is not enough time to re-calculate the value every time it is needed
  - When you must trade-off runtime performance versus memory required

Derived attributes and operations indicate a constraint between values. They can be used to describe a dependency between attribute values that must be maintained by the class. They do not necessarily mean that the attribute value is always calculated from the other attributes.

Example: Define Attributes



Example: Define Attributes

The above example is a portion of the VOPC for the Register for Courses Use-Case Realization. It is the same diagram that was provided earlier in the Define Operations section, except now the operation compartment has been suppressed.

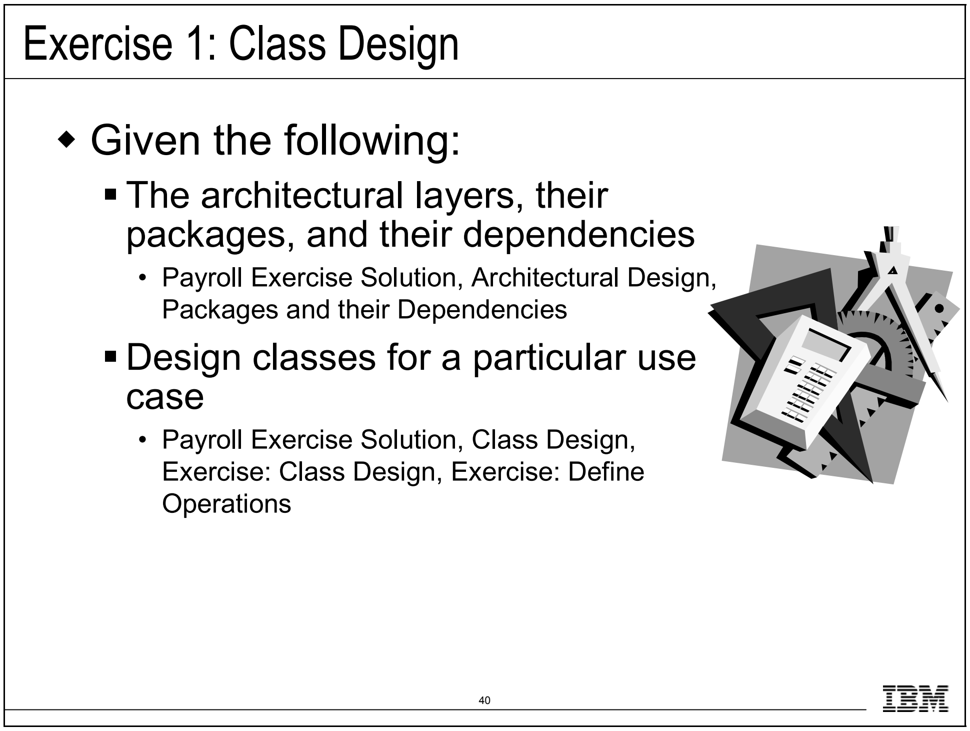
Notice the <<class >> attribute.

In this example, all attributes are private (marked with a "-") to support encapsulation.

The number of students currently enrolled in the CourseOffering is represented by the derived attribute, numStudents, which has a default value of 0.

Semester, Time, and Date are included as the type for some of the attributes. For the Course Registration System, they are considered to be abstract data types that have no significant behavior and thus are not modeled as separate classes.

Exercise 1: Class Design



Exercise 1: Class Design

- Given the following:
  - The architectural layers, their packages, and their dependencies
    - Payroll Exercise Solution, Architectural Design, Packages and their Dependencies
  - Design classes for a particular use case
    - Payroll Exercise Solution, Class Design, Exercise: Class Design, Exercise: Define Operations

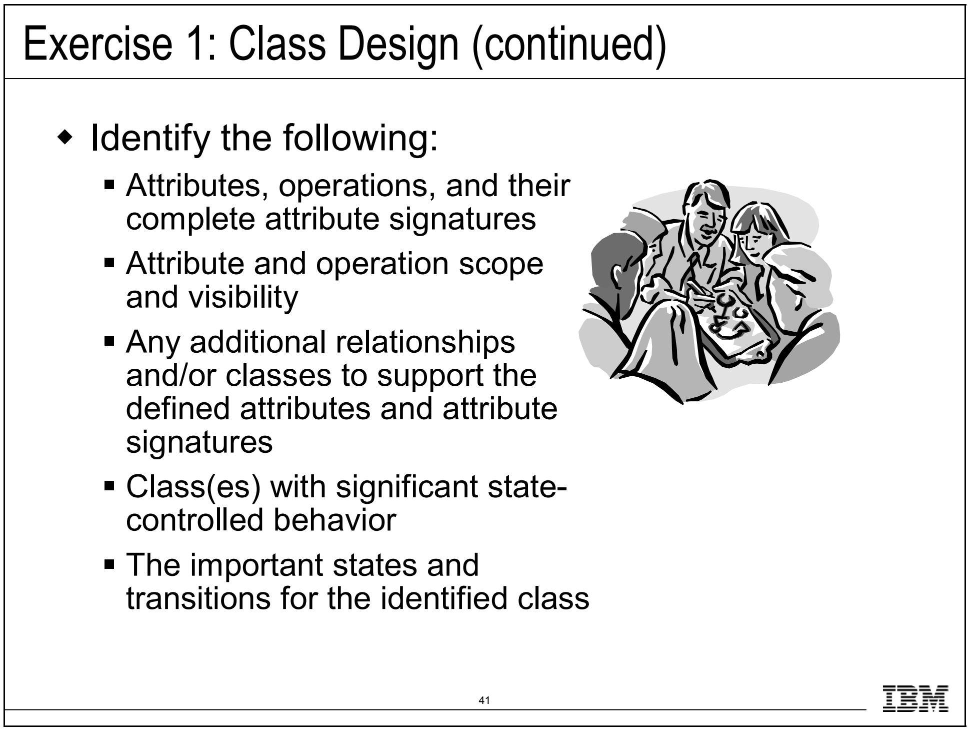
40

The goal of this exercise is to design the attributes and operations of design classes for a use case. This exercise will also require the students to model state-controlled behavior of a design class in a state machine.

References to the givens on the slide:

- **The architectural layers, their packages, and their dependencies:** Payroll Exercise Solution, Architectural Design, Packages and their Dependencies section
- **Design classes for a particular use case:** Payroll Exercise Solution, Class Design, Exercise: Class Design, Exercise: Define Operations section.

Exercise 1: Class Design (continued)



Exercise 1: Class Design (continued)

- Identify the following:
  - Attributes, operations, and their complete attribute signatures
  - Attribute and operation scope and visibility
  - Any additional relationships and/or classes to support the defined attributes and attribute signatures
  - Class(es) with significant state-controlled behavior
  - The important states and transitions for the identified class

A complete operation signature includes the operation name, operation parameters, and operation return value. This may drive the identification of new classes and relationships.

Operation visibility may be public, private, or protected.

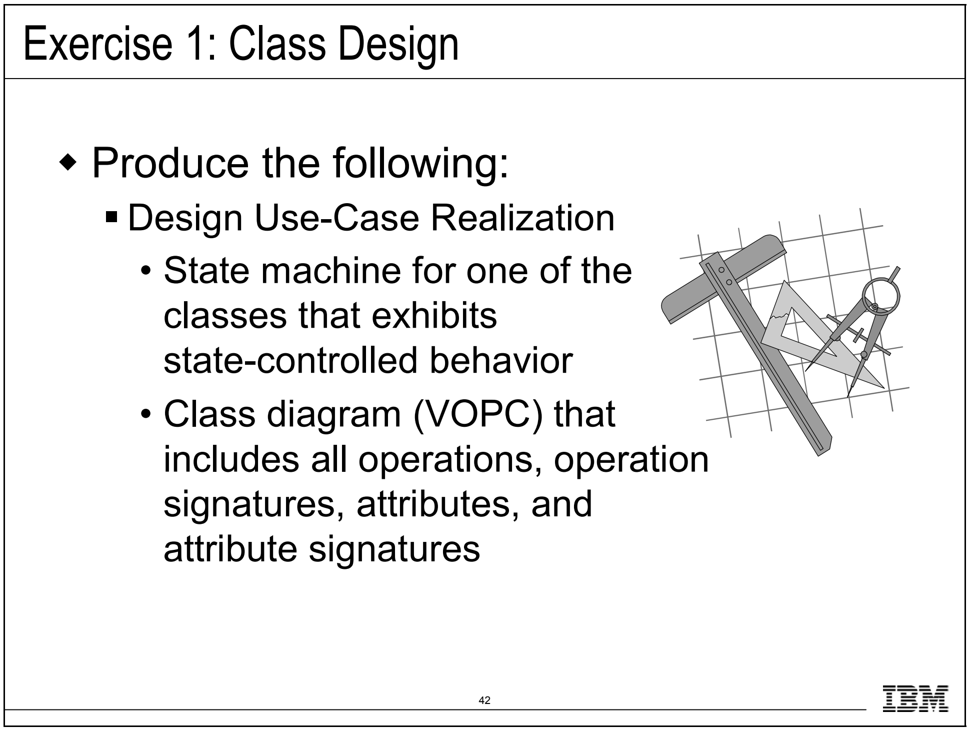
A complete attribute signature includes the attribute name, attribute type, and attribute default value (optional). This may drive the identification of new classes and relationships.

Attribute visibility may be public, private, or protected.

When adding relationships and/or classes, the package/subsystem interrelationships may need to be refined. Such a refinement can be approved only by the architecture team.

The produced state machine should include states, transitions, events, and guard conditions.

Exercise 1: Class Design



Exercise 1: Class Design

- Produce the following:
  - Design Use-Case Realization
    - State machine for one of the classes that exhibits state-controlled behavior
    - Class diagram (VOPC) that includes all operations, operation signatures, attributes, and attribute signatures

42

One of the goals of this exercise is to model state-controlled behavior of a design class in a state machine.

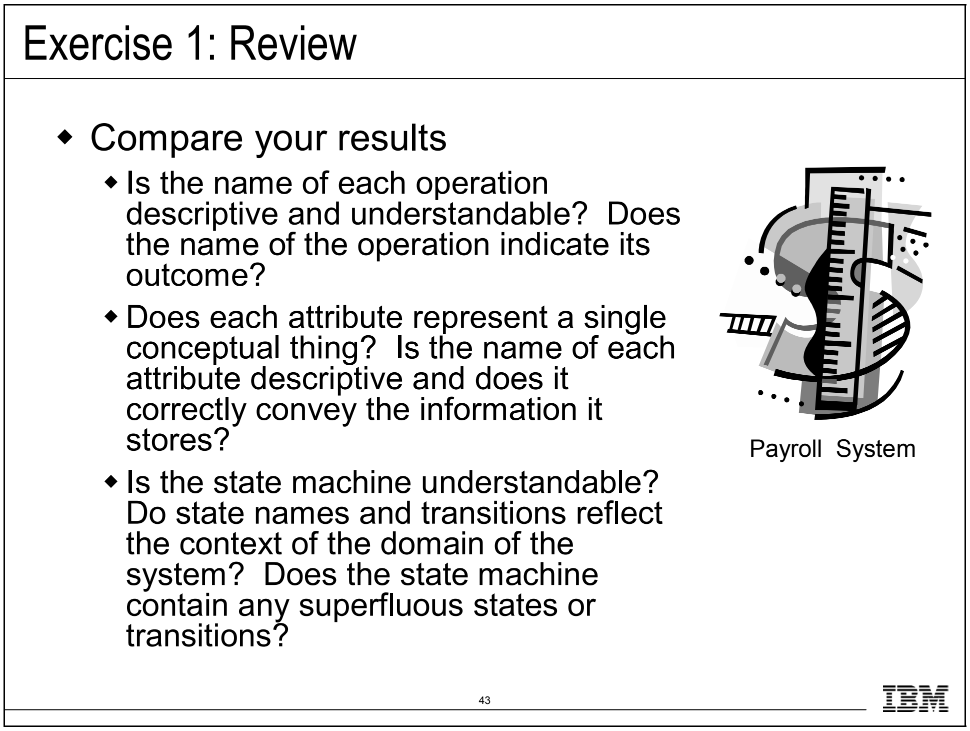
The produced state machine should include states, transitions, events, and guard conditions.

A complete operation signature includes the operation name, operation parameters, and operation return value. This may drive the identification of new classes and relationships.

References to sample diagrams within the course that are similar to what should be produced are:

- **Operations**: p.13-20
- **State machines**: p. 13-32
- **Attributes**: p. 13-39

Exercise 1: Review



Exercise 1: Review

- Compare your results
  - Is the name of each operation descriptive and understandable? Does the name of the operation indicate its outcome?
  - Does each attribute represent a single conceptual thing? Is the name of each attribute descriptive and does it correctly convey the information it stores?
  - Is the state machine understandable? Do state names and transitions reflect the context of the domain of the system? Does the state machine contain any superfluous states or transitions?

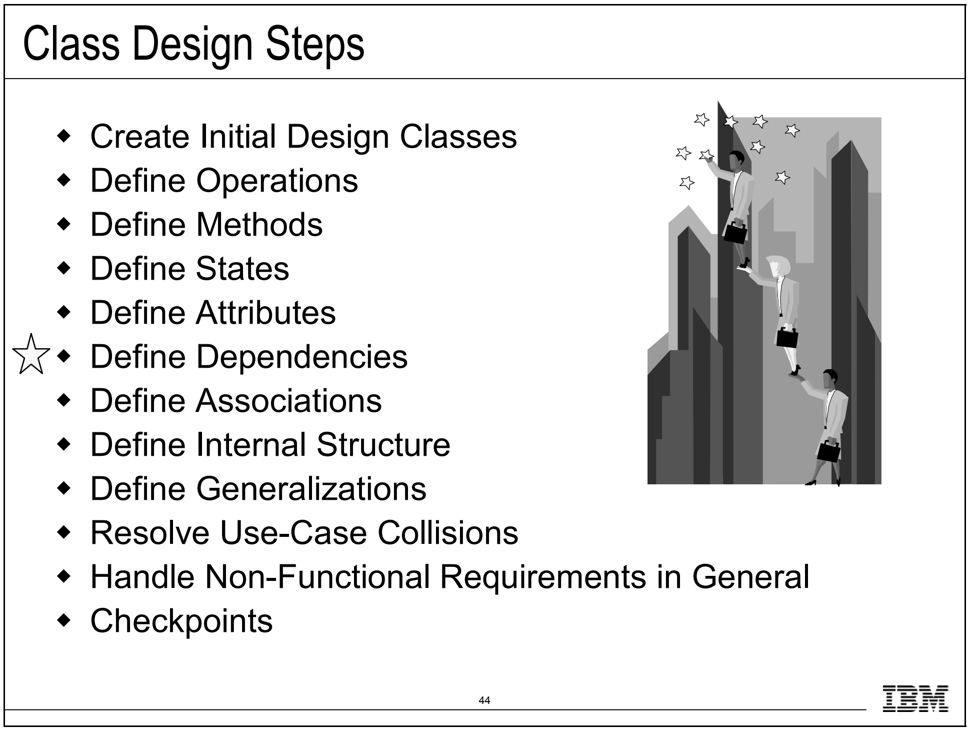
Payroll System

43

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

- Is the name of each operation descriptive and understandable? The operation should be named from the perspective of the user, so the name of the operation should indicate its outcome.
- Does each attribute represent a single conceptual thing? Is the name of each attribute descriptive and does it correctly convey the information it stores?
- Is the state machine understandable? Do the state names and transitions accurately reflect the domain of the system? The state machine should accurately reflect the lifetime of an object.
- Does the state machine contain any superfluous states or transitions? States and transitions should reflect the object's attributes, operations, and relationships. Do not read anything extra into the lifetime of the object that cannot be supported by the class structure.

Define Dependencies

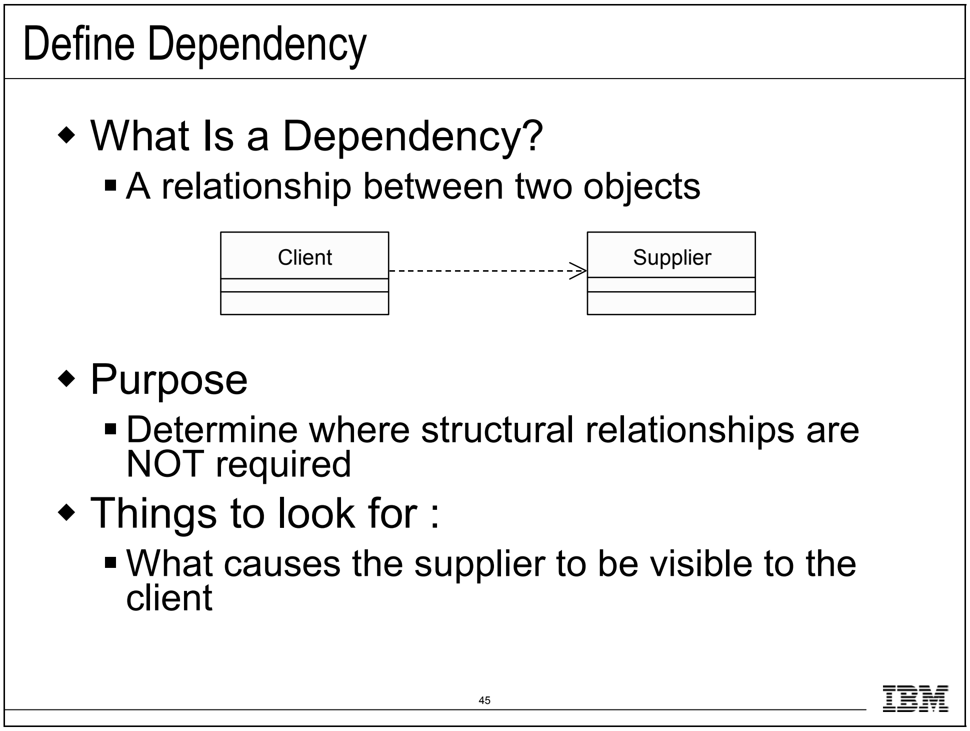


Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- ☆ Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Prior to Design, many of the relationships were modeled as associations. Now, with the added knowledge you have gained throughout Design, you are in a position to refine some of those relationships into dependencies.

Define Dependency



Define Dependency

- What Is a Dependency?
  - A relationship between two objects

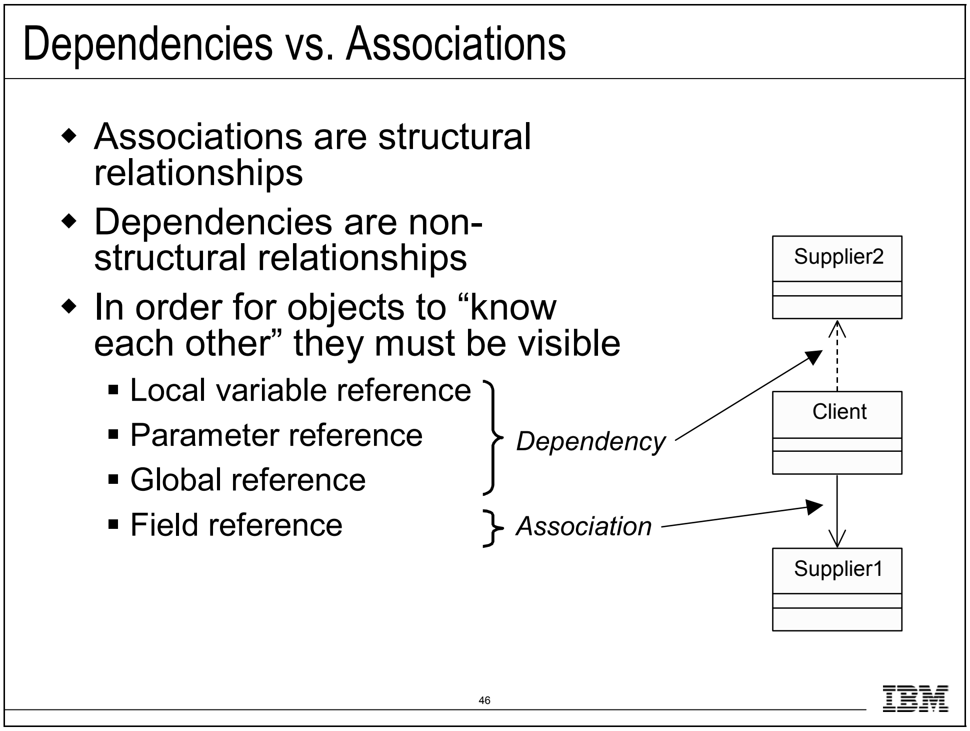
Client
Supplier

- Purpose
  - Determine where structural relationships are NOT required
- Things to look for :
  - What causes the supplier to be visible to the client

During Analysis, we assumed that all relationships were structural (associations or aggregations). In Design, we must decide what type of communication pathway is required.

A dependency relationship denotes a semantic relationship between model elements, where a change in the supplier may cause a change in the client.

Dependencies vs. Associations



Dependencies vs. Associations

- Associations are structural relationships
- Dependencies are non-structural relationships
- In order for objects to "know each other" they must be visible
  - Local variable reference
  - Parameter reference
  - Global reference
  - Field reference

Dependency

Association

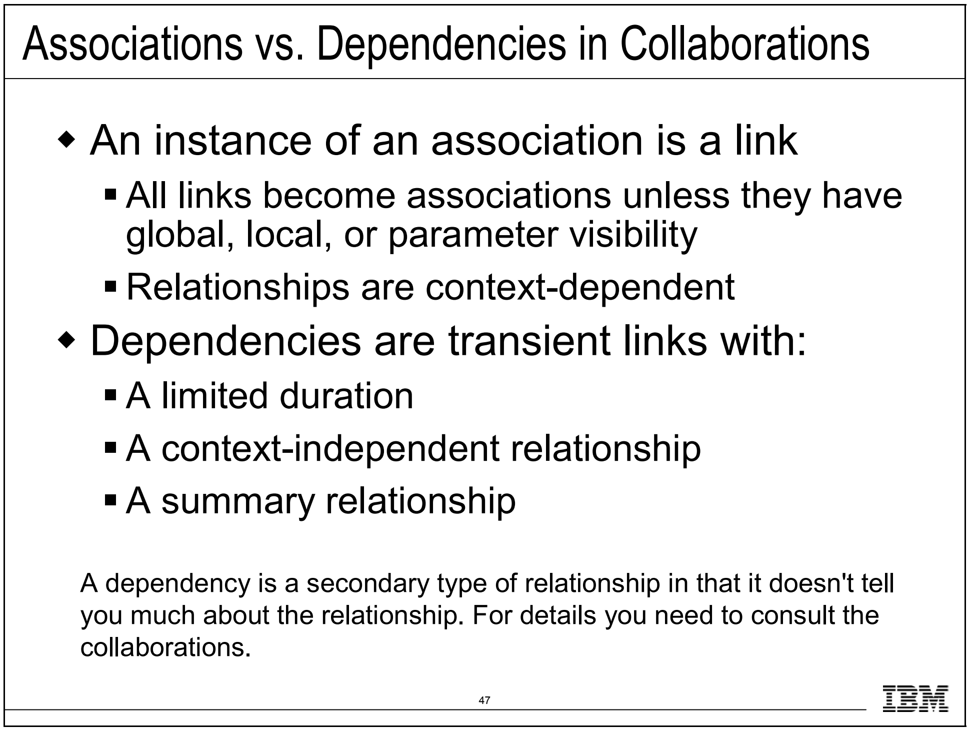
There are four options for creating a communication pathway to a supplier object:

- **Global**: The supplier object is a global object.
- **Parameter**: The supplier object is a parameter to, or the return class of, an operation in the client object.
- **Local**: The supplier object is declared locally (that is, created temporarily during execution of an operation).
- **Field**: The supplier object is a data member in the client object.

A dependency is a type of communication pathway that is a transient type of relationship. These occur when the visibility is global, parameter, or local.

Look at each association relationship and determine whether it should remain an association or become a dependency. Associations and aggregations are structural relationships (field visibility). Association relationships are realized by variables that exist in the data member section of the class definition. Any other relationships (global, local, and parameter visibility) are dependency relationships.

Associations vs. Dependencies in Collaborations



Associations vs. Dependencies in Collaborations

- An instance of an association is a link
  - All links become associations unless they have global, local, or parameter visibility
  - Relationships are context-dependent
- Dependencies are transient links with:
  - A limited duration
  - A context-independent relationship
  - A summary relationship

A dependency is a secondary type of relationship in that it doesn't tell you much about the relationship. For details you need to consult the collaborations.

47

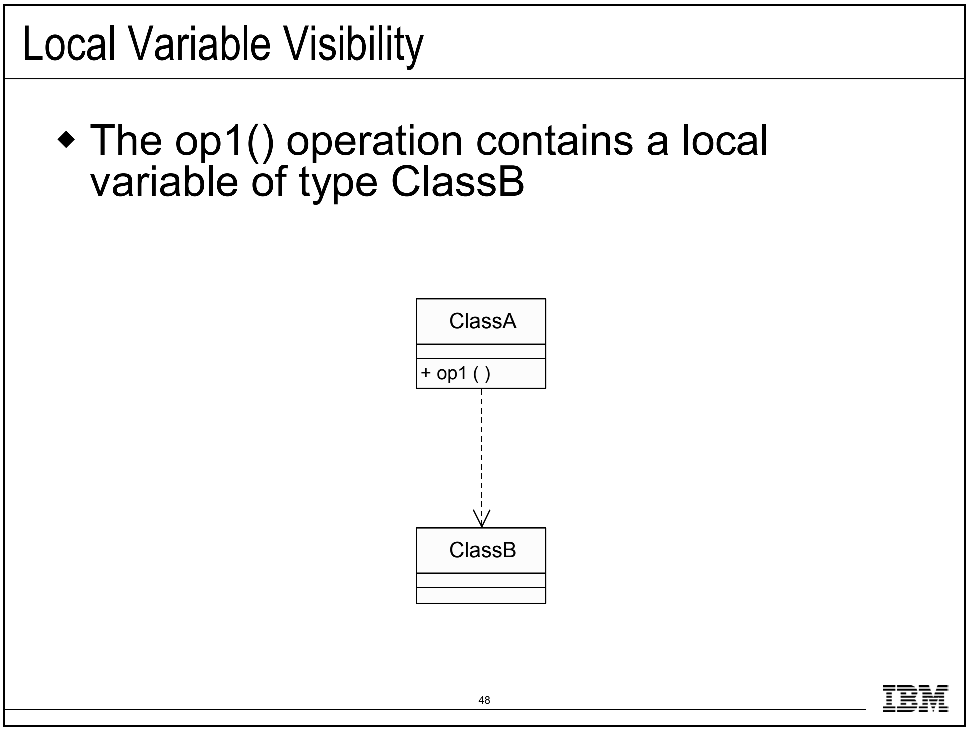
According to the UML 2 Specification, a link is an instance of an association. Specifically, an association declares a connection (link) between instances of the associated classifiers (classes). A dependency relationship indicates that the implementation or functioning of one or more elements requires the presence of one or more other elements.

Note that links modeled as parameter, global, or local are transient links. They exist only for a limited duration and in the specific context of the collaboration; in that sense, they are instances of the association role in the collaboration. However, the relationship in a class model (independent of context) should be, as stated above, a dependency. In *The UML Reference Manual,* the definition of a transient link is: "It is possible to model all such links as associations, but then the conditions on the associations must be stated very broadly, and they lose much of their precision in constraining combinations of objects." In this situation, the modeling of a dependency is less important than the modeling of the relationship in the collaboration, because the dependency does not describe the relationship completely, only that it exists.

If you believe that a link in a collaboration is a transient link, it indicates that the context-independent relationship between the classes is a dependency. Dependency is a “summary” relationship — for details you have to consult the behavioral model.

Context is defined as "a view of a set of related modeling elements for a particular purpose."

Local Variable Visibility

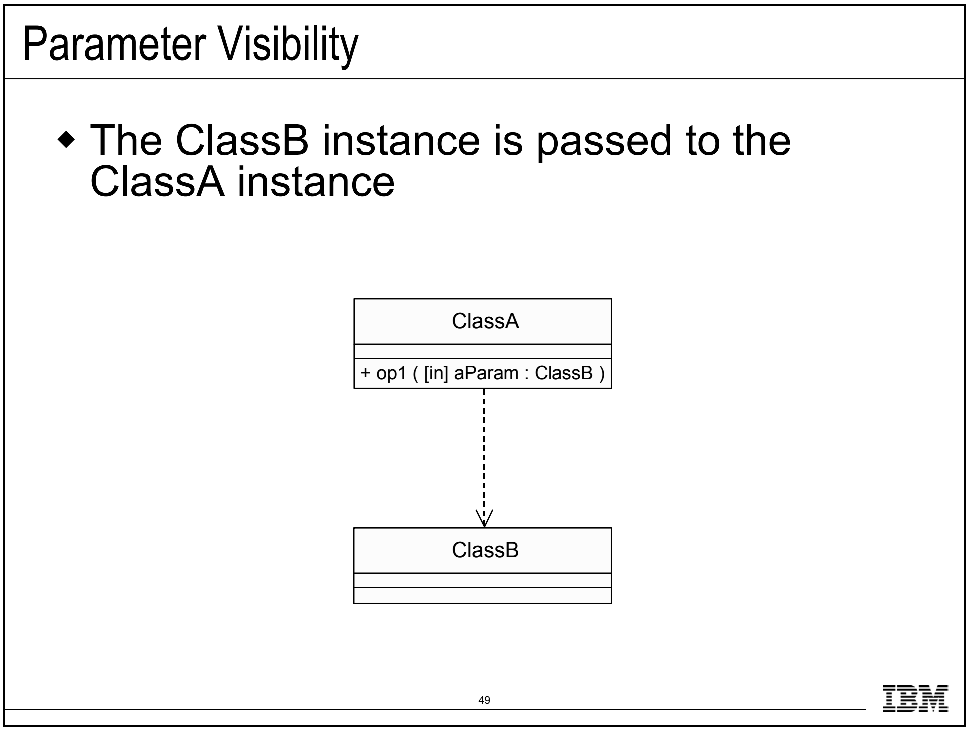


Local Variable Visibility

- The op1() operation contains a local variable of type ClassB

Is the receiver a temporary object created and destroyed during the operation itself? If so, establish a **dependency** between the sender and receiver classes in a class diagram containing the two classes.

Parameter Visibility

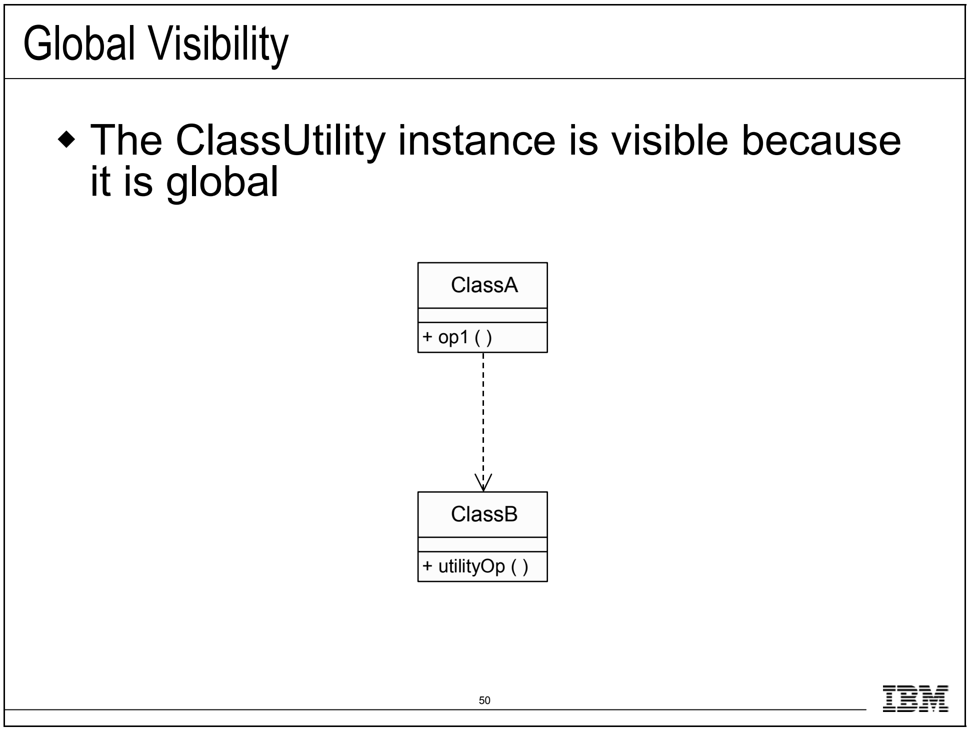


Parameter Visibility

- The ClassB instance is passed to the ClassA instance

Is the reference to the receiver passed as a parameter to the operation? If so, establish a **dependency** between the sender and receiver classes in a Class diagram containing the two classes.

Global Visibility

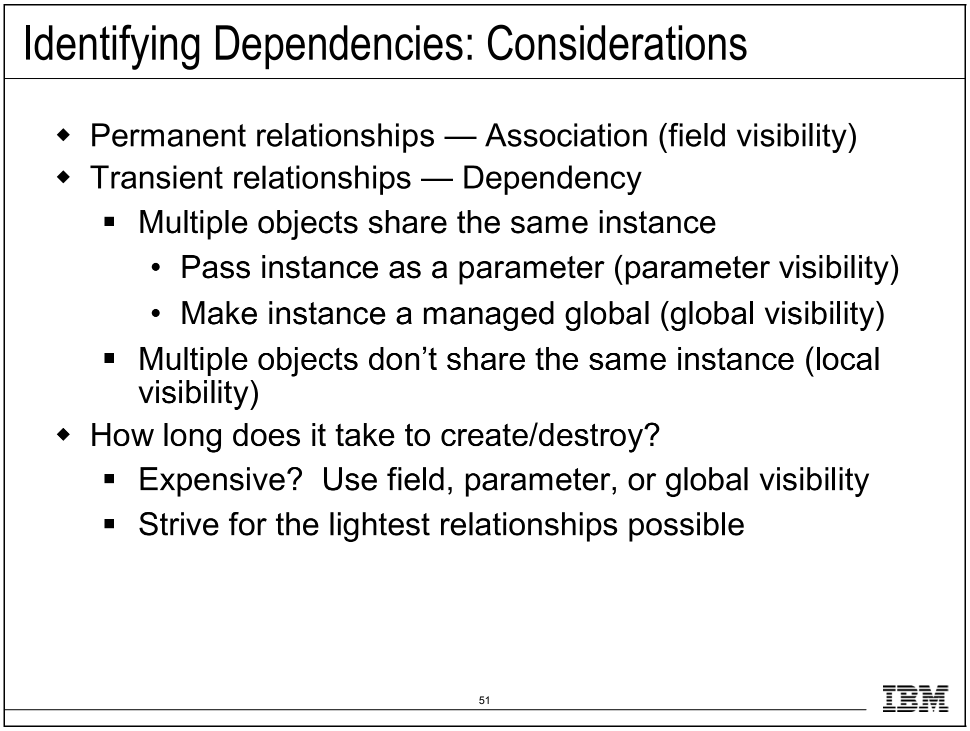


Global Visibility

- The ClassUtility instance is visible because it is global

Is the receiver a “global?” If so, establish a **dependency** between the sender and receiver classes in a class diagram containing the two classes.

Identifying Dependencies: Considerations



Identifying Dependencies: Considerations

- Permanent relationships — Association (field visibility)
- Transient relationships — Dependency
  - Multiple objects share the same instance
    - Pass instance as a parameter (parameter visibility)
    - Make instance a managed global (global visibility)
  - Multiple objects don’t share the same instance (local visibility)
- How long does it take to create/destroy?
  - Expensive? Use field, parameter, or global visibility
  - Strive for the lightest relationships possible

Strive for real-world relationships. Semantic, structural relationships should be associations.

Strive for the lightest relationships possible. Dependency is the cheapest to keep, easiest to use, and benefits from encapsulation.

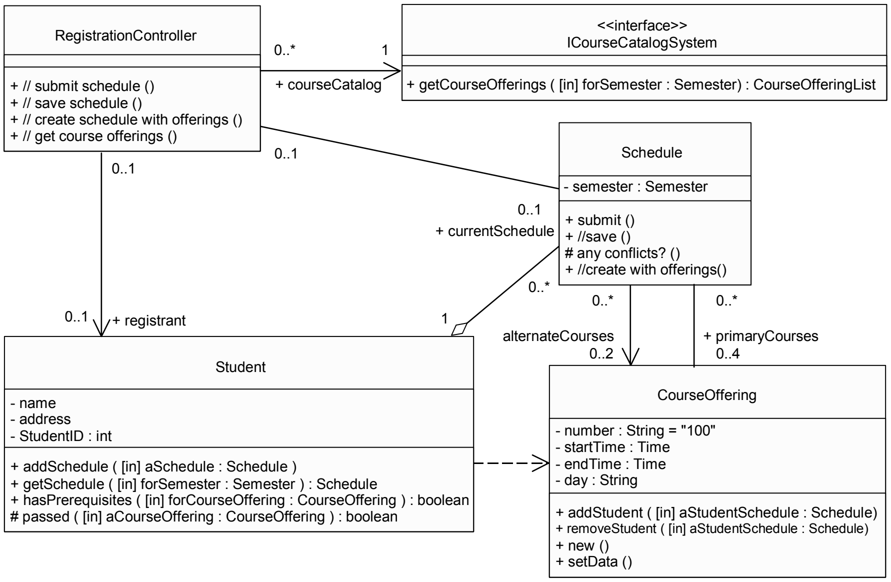
Is the relationship transient? Will you need this relationship again and again, or do you just need it to do some work and then throw it away? If its needed again and again, that is, if a thing appears to remain related to another thing even across the execution of one or more operations, then it is likely an association and should benefit from field visibility. Otherwise, the relationship may be transient and can have local, parameter, or global visibility.

Is the same instance shared across objects? If many objects at runtime need it again and again and they share the same instance, maybe you should pass it around as a parameter. If there is only one instance in existence in the whole process, set it up as a managed global (for example, Singleton design pattern). If the same instance is not shared, then having a local copy will suffice (local visibility).

How long does the instance take to create and destroy? Is it expensive to connect and disconnect every time you need it? If so, you would want to give the object field, parameter, or global visibility.

Example: Define Dependencies (before)

Example: Define Dependencies (before)



RegistrationController

+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()

<<interface>>
ICourseCatalogSystem

+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList

Schedule

- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()

Student

- name
- address
- StudentID : int

+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean

CourseOffering

- number : String = "100"
- startTime : Time
- endTime : Time
- day : String

+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()

This class diagram is a subset of the View of Participating Classes (VOPC) from the Register for Courses use case.

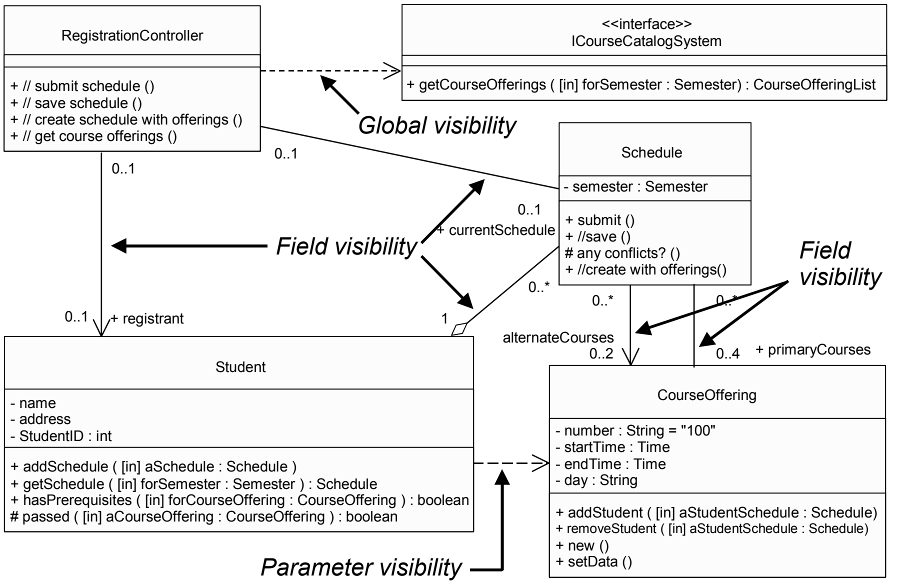
Up to this point, most of the relationships that we have defined have been associations (and aggregations). Now we will see how some of these associations/aggregations are refined into dependencies.

The dependency shown above from Schedule to CourseOffering was previously defined in the Define Operations section to support the Schedule operation signatures.

Note: All association/aggregation relationships should be examined to see if they should be dependencies. We have just chosen to work with a VOPC subset for this example.

Example: Define Dependencies (after)

Example: Define Dependencies (after)



RegistrationController

+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()

<<interface>>
ICourseCatalogSystem

+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList

Schedule

- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()

Student

- name
- address
- StudentID : int

+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean

CourseOffering

- number : String = "100"
- startTime : Time
- endTime : Time
- day : String

+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()

Global visibility

Field visibility

Field visibility

Parameter visibility

This example demonstrates how an association on the class diagram provided on the previous slide has been refined into a dependency relationship.

During a registration session, the RegistrationController works with one Student, the registrant (the Student that is registering), and one Schedule, the current Schedule (the Student's Schedule for the current semester). These instances need to be accessed by more than one of the RegistrationController's operations, so field visibility is chosen from RegistrationController to Student and from RegistrationController to Schedule. Thus, the relationships remain associations.

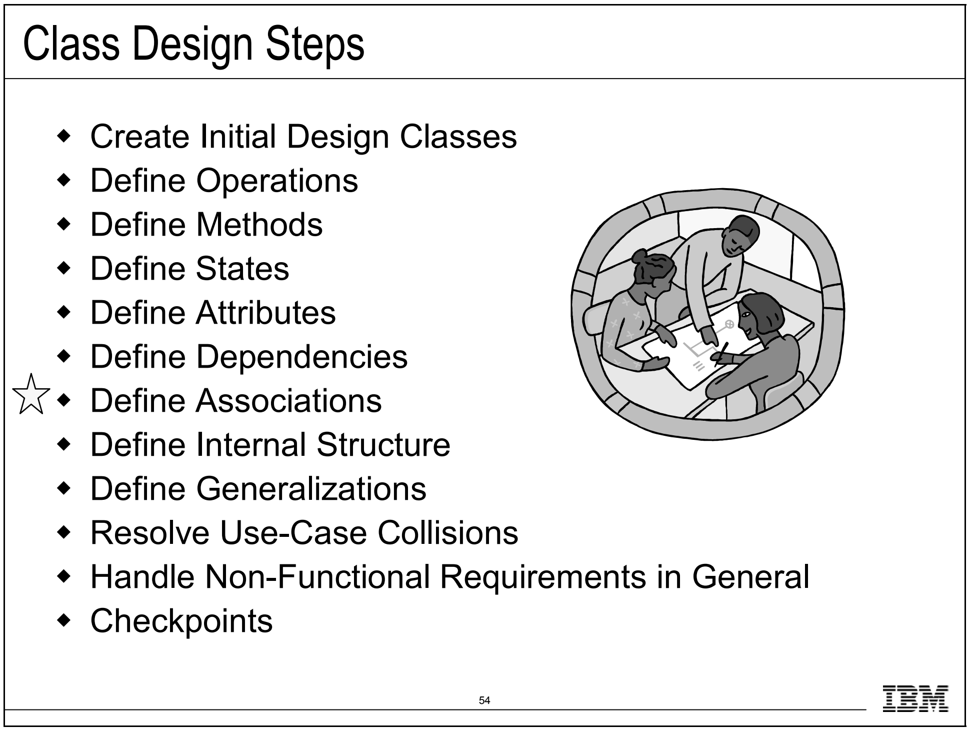
A Student manages its own Schedules, so field visibility is chosen from Student to Schedule and the relationship remains an aggregation.

CourseOfferings are part of the semantics of what defines a Schedule (a Schedule is the courses a that a Student has selected for a semester). Thus, field visibility is chosen from Schedule to CourseOffering and the relationships remain associations.

The Student class has several operations where CourseOffering appears in the parameter list. Thus, parameter visibility is chosen from Student to CourseOffering. This relationship was actually defined earlier in the Define Operations section.

It is envisioned that the course Catalog System may need to be accessed by multiple clients in the system, so global visibility was chosen, and the relationship becomes a dependency. This is the only change that was made to the relationships shown on the previous slide.

Define Associations



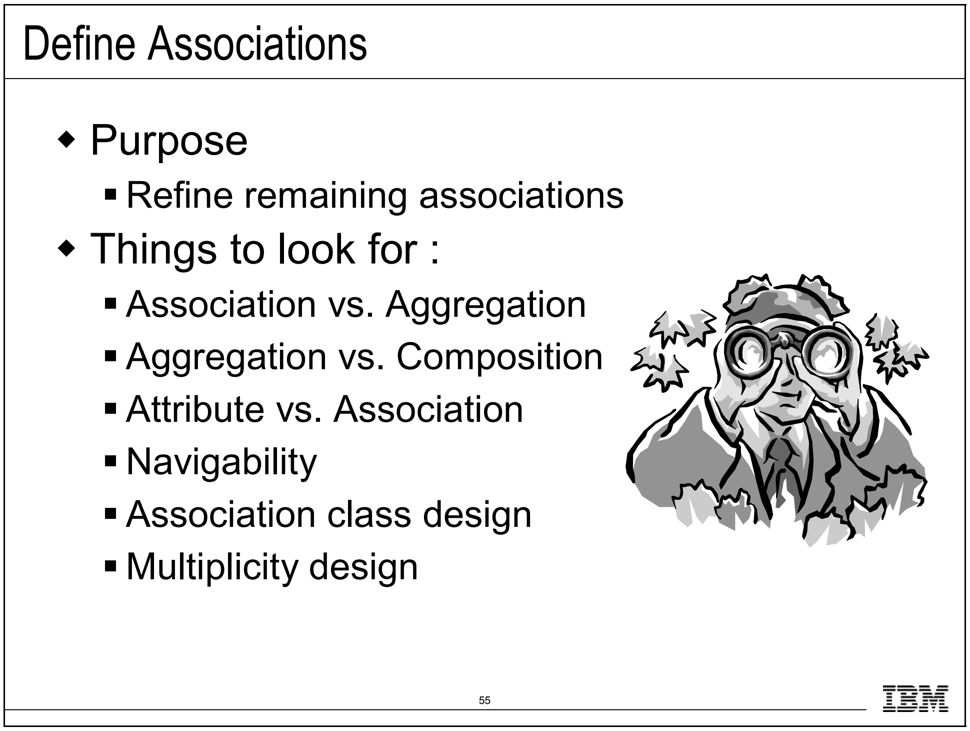
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- ☆ Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

54

We now are going to turn our attention to the remaining class associations, adding some Design-level refinements that drive the implementation.

Define Associations



Define Associations

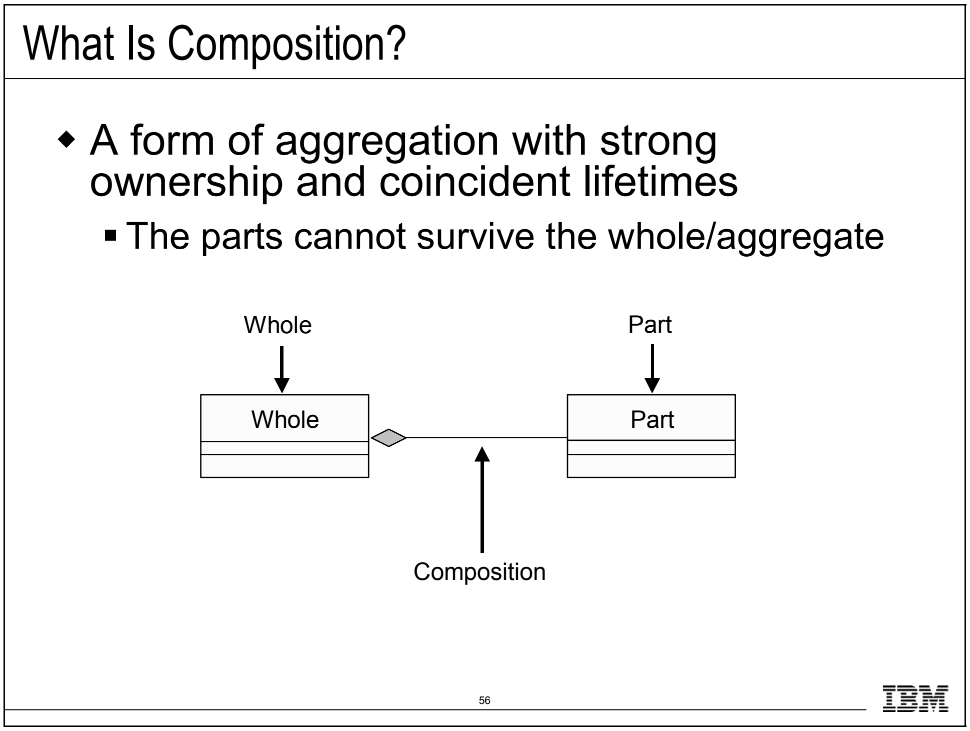
- Purpose
  - Refine remaining associations
- Things to look for :
  - Association vs. Aggregation
  - Aggregation vs. Composition
  - Attribute vs. Association
  - Navigability
  - Association class design
  - Multiplicity design

At this point, we have identified which class relationships should be dependencies. Now it is time to design the details of the remaining associations.

Also, additional associations may need to be defined to support the method descriptions defined earlier. Remember, to communicate between their instances, classes must have relationships with each other.

We will discuss each of the “things to look for” topics on subsequent slides, except for “Association vs. Aggregation,” which was discussed in the Use-Case Analysis module.

What Is Composition?



What Is Composition?

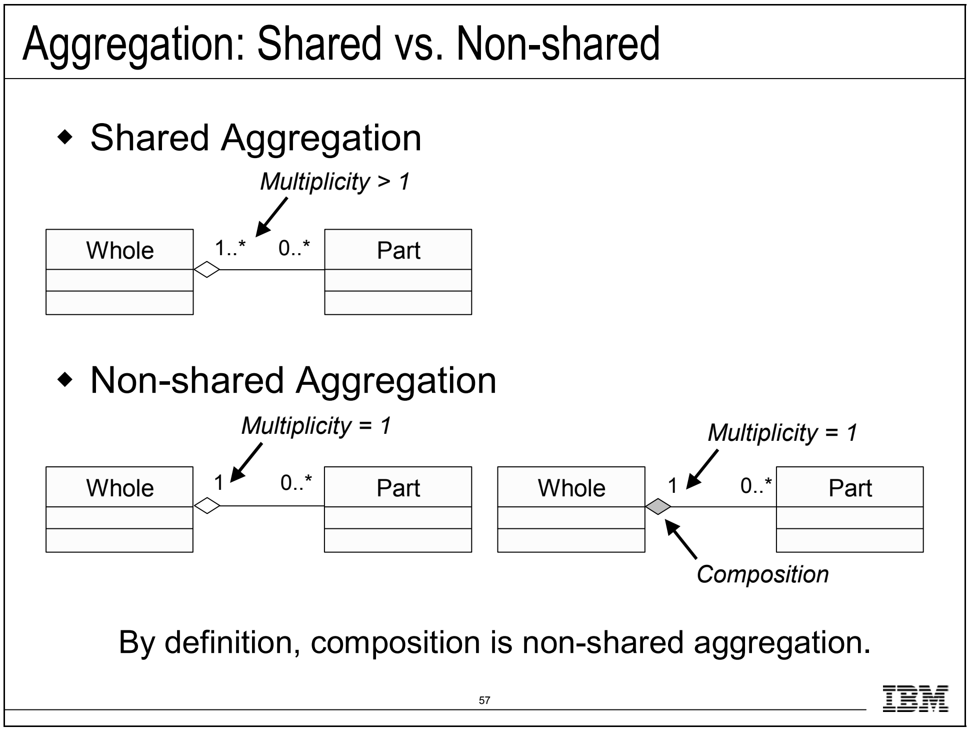
- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate

Composition is a form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate. The whole “owns” the part and is responsible for the creation and destruction of the part. The part is removed when the whole is removed. The part may be removed (by the whole) before the whole is removed.

A solid filled diamond is attached to the end of an association path (on the “whole side”) to indicate composition.

In some cases, composition can be identified as early as Analysis, but more often it is not until Design that such decisions can be made confidently. That is why composition is introduced here rather than in Use-Case Analysis.

Aggregation: Shared vs. Non-shared



Aggregation: Shared vs. Non-shared

- Shared Aggregation

Multiplicity > 1

Whole 1..* 0..* Part

- Non-shared Aggregation

Multiplicity = 1

Whole 1 0..* Part

Multiplicity = 1

Whole 1 0..* Part

Composition

By definition, composition is non-shared aggregation.

57

IBM

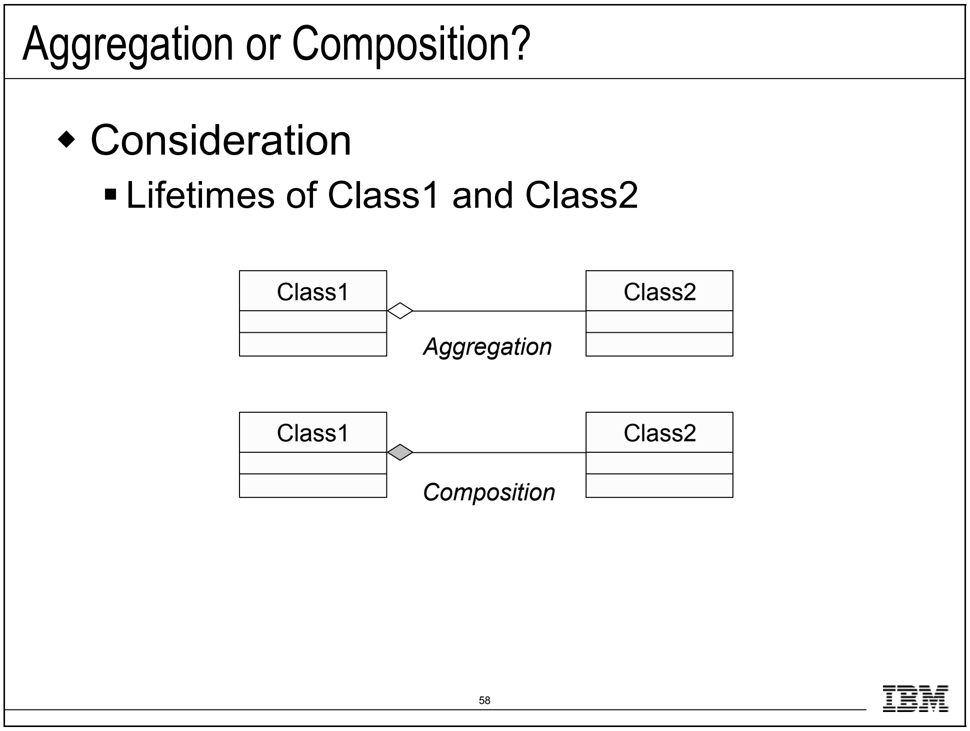
For composition, the multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is also unchangeable; that is, once established its links cannot be changed. Parts with multiplicity having a lower bound of 0 can be created after the aggregate itself, but once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

Non-shared aggregation does not necessarily imply composition.

An aggregation relationship that has a multiplicity greater than one established for the aggregate is called “shared,” and destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph, or a tree with many roots.

An example of shared aggregation may be between a University class and a Student class (the University being the “whole” and the Students being the “parts”). With regards to registration, a Student does not make sense outside the context of a University, but a Student may be enrolled in classes in multiple Universities.

Aggregation or Composition?



Aggregation or Composition?

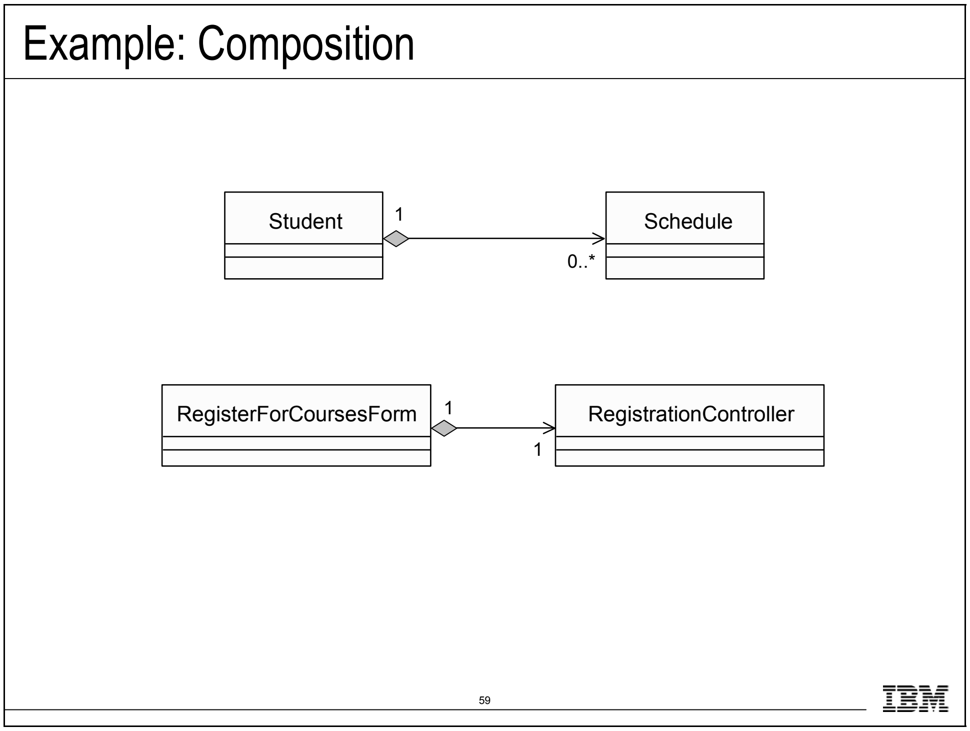
- Consideration
  - Lifetimes of Class1 and Class2

The use of association versus aggregation was discussed in the Use-Case Analysis module. Here we discuss the use of “vanilla” aggregation versus composition.

Composition should be used over "plain" aggregation when there is a strong interdependency between the aggregate and the parts, where the definition of the aggregate is incomplete without the parts. For example, it does not make sense to have an Order if there is nothing being ordered.

Composition should be used when the whole and part must have coincident lifetimes. Selection of aggregation or composition will determine how object creation and deletion are designed.

Example: Composition



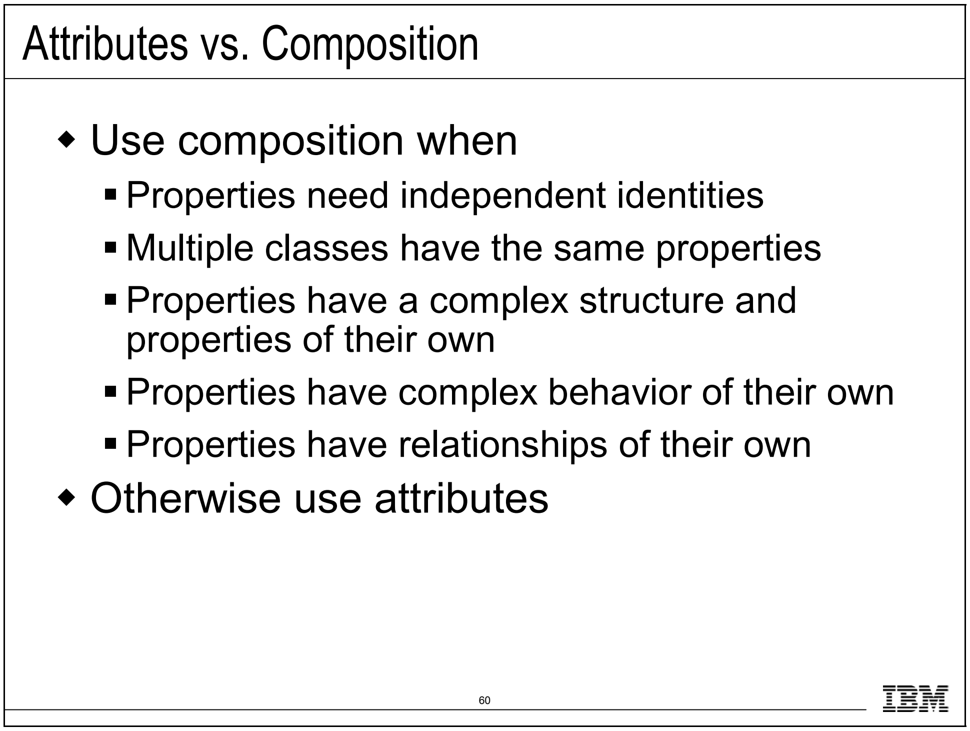
Example: Composition

This slide demonstrates two examples of composition.

The top graphic demonstrates how a previous aggregation relationship has been refined into a composition relationship.The relationship from Student to Schedule is modeled as a composition because if you got rid of the Student, you would get rid of any Schedules for that Student.

The bottom graphic demonstrates how a previous association relationship has been refined into a composition relationship. It was decided that an instance of a RegistrationController would NEVER exist outside the context of a particular Register For Courses Student session. Thus, since the RegisterForCoursesForm represents a particular Register For Courses session, a RegistrationController would NEVER exist outside of the context of a particular RegisterForCoursesForm. When a MaintainScheduleForm is created, an instance of RegistrationController should always be created. When MaintainScheduleForm is deleted, the instance of the RegistrationController should always be deleted. Thus, because they now have coincident lifetimes, composition is used instead of an association.

Attributes vs. Composition



Attributes vs. Composition

- Use composition when
  - Properties need independent identities
  - Multiple classes have the same properties
  - Properties have a complex structure and properties of their own
  - Properties have complex behavior of their own
  - Properties have relationships of their own
- Otherwise use attributes

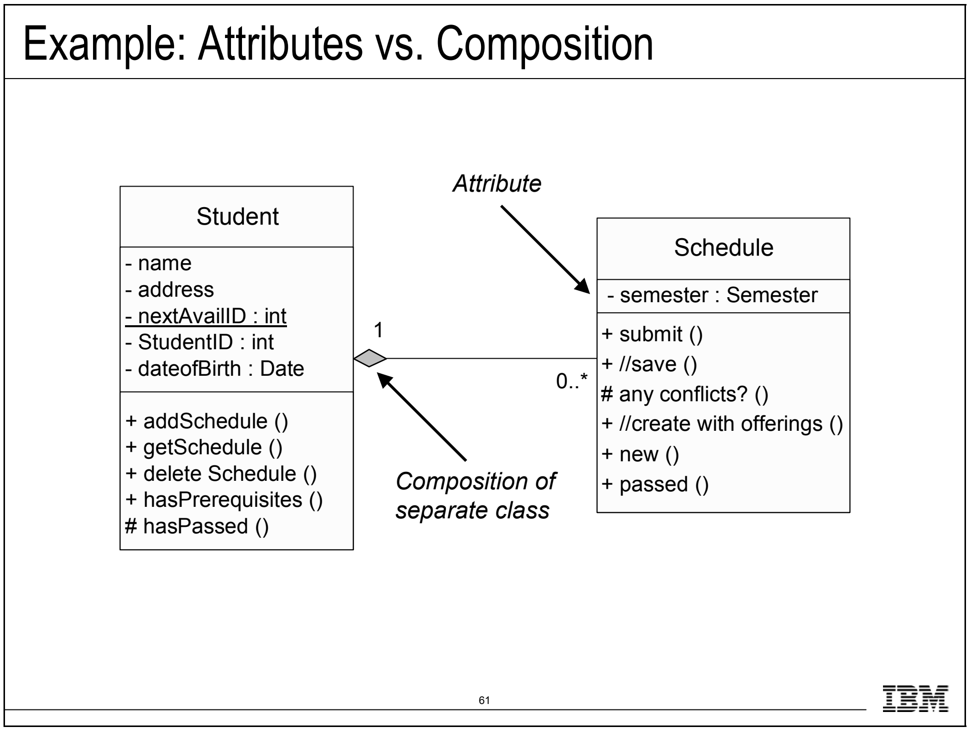
A property of a class is something that the class knows about. You can model a class property as a class (with a composition relationship), or as a set of attributes of the class. In other words, you can use composition to model an attribute.

The decision whether to use a class and composition, or a set of attributes, depends on the following:

- Do the properties need to have independent identity, such that they can be referenced from a number of objects? If so, use a class and composition.
- Do a number of classes need to have the same properties? If so, use a class and composition.
- Do the properties have a complex structure, properties, and behavior of their own? If so, use a class (or classes) and composition.
- Otherwise, use attributes.

The decision of whether to use attributes or a composition association to a separate class may also be determined based on the degree of coupling between the concepts being represented. When the concepts being modeled are tightly connected, use attributes. When the concepts are likely to change independently, use composition.

Example: Attributes vs. Composition



Example: Attributes vs. Composition

Student

- name
- address
- nextAvailID : int
- StudentID : int
- dateofBirth : Date

+ addSchedule ()
+ getSchedule ()
+ delete Schedule ()
+ hasPrerequisites ()
# hasPassed ()

Schedule

- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings ()
+ new ()
+ passed ()

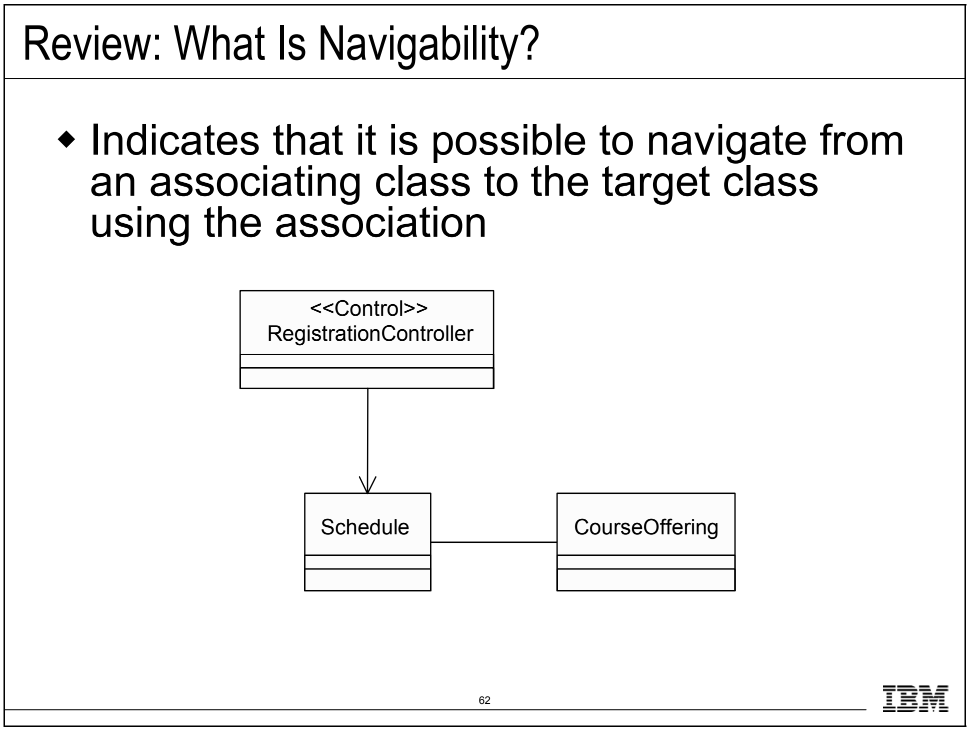
Attribute

Composition of separate class

In this example, Semester is not a property that requires independent identity, nor does it have a complex structure. Thus, it was modeled as an attribute of Schedule.

On the other hand, the relationship from Student to Schedule is modeled as a composition rather than an attribute because Schedule has a complex structure and properties of its own.

Review: What Is Navigability?



Review: What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association

The navigability property on a role indicates that it is possible to navigate from an associating class to the target class using the association. This may be implemented in a number of ways: by direct object references, by associative arrays, hash-tables, or any other implementation technique that allows one object to reference another.

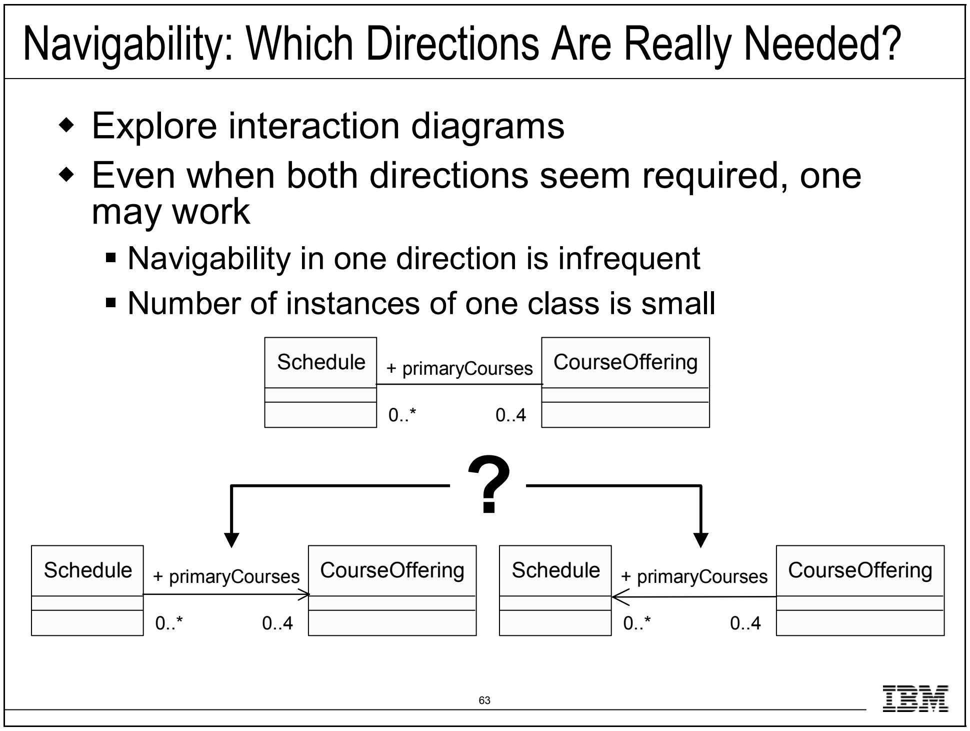
Navigability is indicated by an open arrow placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true (associations are bi-directional by default).

In the course registration example, the association between the Schedule and the Course Offering is navigable in both directions. That is, a Schedule must know the Course Offering assigned to the Schedule and the Course Offering must know the Schedules it has been placed in.

When no arrowheads are shown, the association is assumed to be navigable in both directions.

In the case of the associations between Schedule and Registration Controller**,** the Registration Controller must know its Schedules, but the Schedules have no knowledge of the Registration Controllers (or other classes, since many things have addresses) associated with the address. As a result, the navigability property of the Registration Controller end of the association is turned off.

Navigability: Which Directions Are Really Needed?



Navigability: Which Directions Are Really Needed?

- Explore interaction diagrams
- Even when both directions seem required, one may work
  - Navigability in one direction is infrequent
  - Number of instances of one class is small

During Use-Case Analysis, associations (and aggregations) may have been assumed to be bi-directional (that is, communication may occur in both directions).

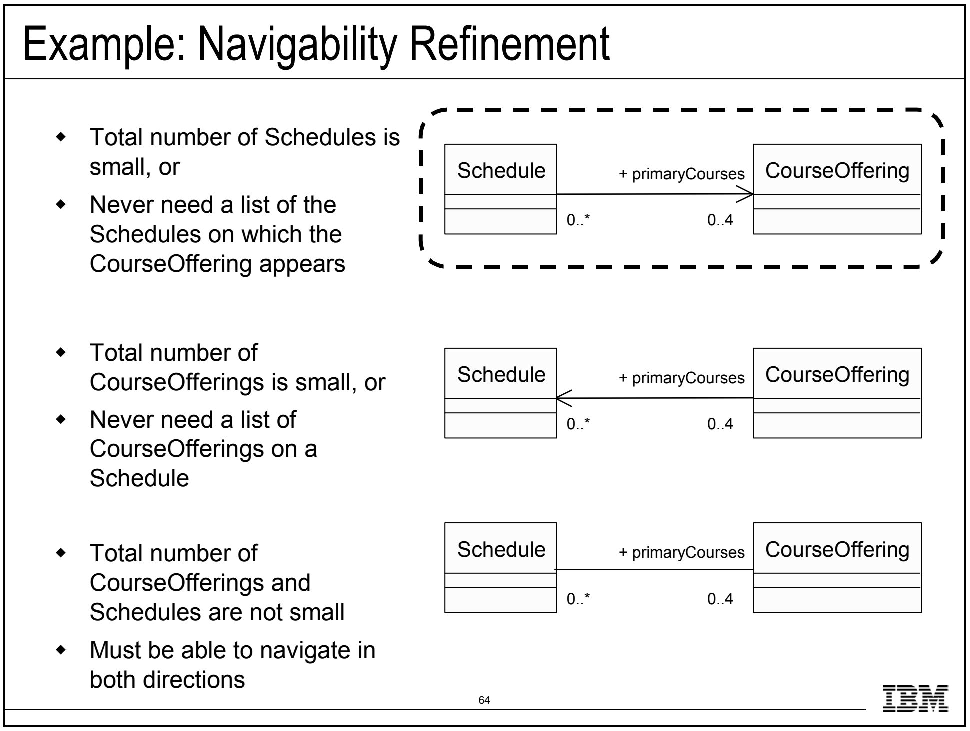
During **Class Design**, the association's navigability needs to be refined so that only the required communication gets implemented.

Navigation is readdressed in **Class Design** because we now understand the responsibilities and collaborations of the classes better than we did in Use-Case Analysis. We also want to refine the relationships between classes.Two-way relationships are more difficult and expensive to implement than one-way relationships. Thus, one of the goals in **Class Design** is the reduction of two-way (bi-directional) relationships into one-way (unidirectional) relationships.

The need for navigation is revealed by the use cases and scenarios. The navigability defined in the class model must support the message structure designed in the interaction diagrams.

In some circumstances even if two-way navigation is required, a one-way association may suffice. For example, you can use one-way association if navigation in one of the directions is very infrequent and does not have stringent performance requirements, or the number of instances of one of the classes is very small.

Example: Navigability Refinement



Example: Navigability Refinement

- Total number of Schedules is small, or
- Never need a list of the Schedules on which the CourseOffering appears

- Total number of CourseOfferings is small, or
- Never need a list of CourseOfferings on a Schedule

- Total number of CourseOfferings and Schedules are not small
- Must be able to navigate in both directions

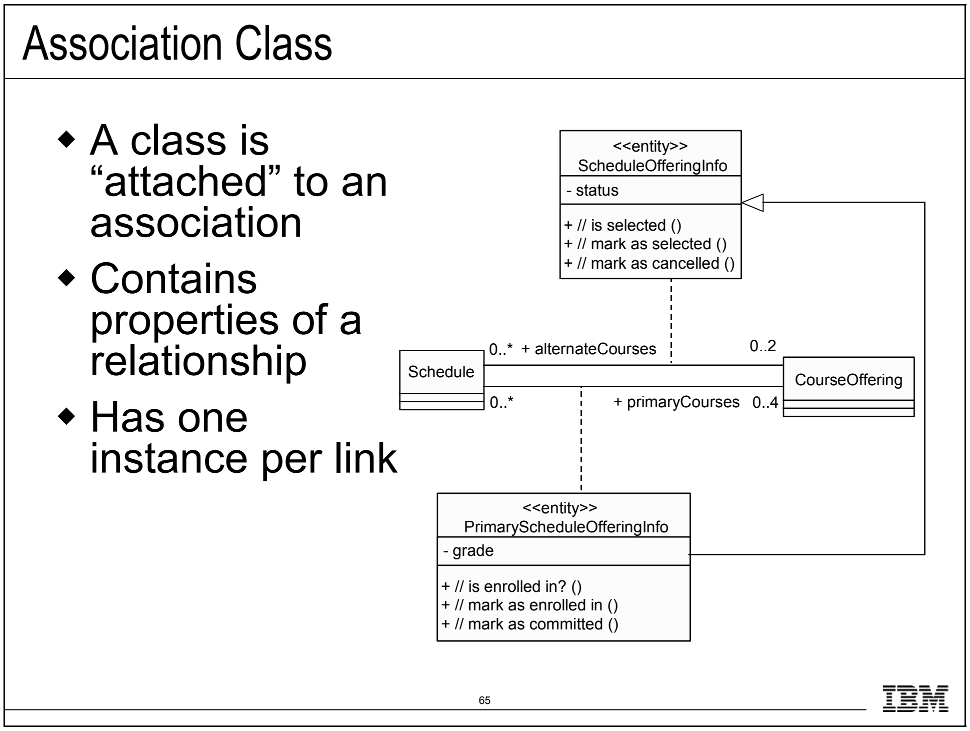
In Situation 1: Implement only the Schedule-to-CourseOffering direction. If navigation is required in the other direction (that is, if a list of Schedules on which the CourseOffering appears is needed), it is implemented by searching all of the Schedule instances and checking the CourseOfferings that appear on them.

In Situation 2: Implement only the CourseOffering-to-Schedule direction. If navigation is required in the other direction (that is, if a list of CourseOfferings on a Schedule is needed), it is implemented by searching all of the CourseOffering instances and checking the Schedules on which they appear.

For the example in this course, the first option (navigation from Schedule to the Primary CourseOfferings) was chosen.

Note: This navigation decision holds true for the navigability from Schedule to the alternate CourseOfferings, as well.

Association Class



Association Class

- A class is “attached” to an association
- Contains properties of a relationship
- Has one instance per link

An association class is a class that is connected to an association. It is a full-fledged class and can contain attributes, operations, and other associations.

Association classes allow you to store information about the relationship itself. The association class includes information that is not appropriate for, or does not belong in, the classes at either end of the relationship.

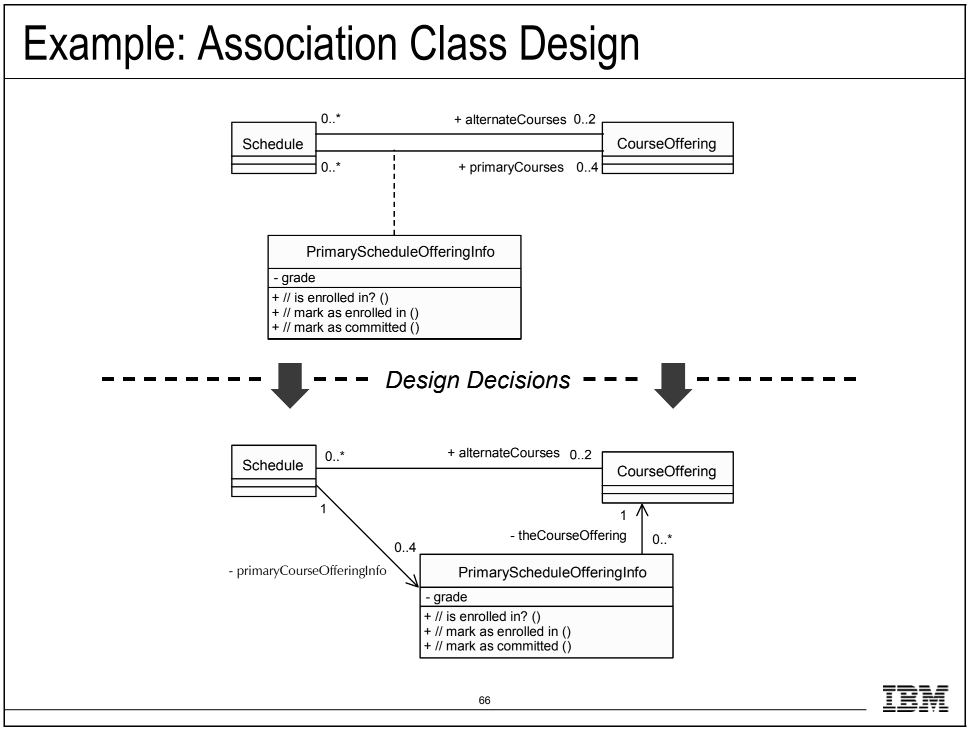
There is an instance of the association class for every instance of the relationship (that is, for every link).

In many cases, association classes are used to resolve many-to-many relationships, as shown in the example above. In this case, a Schedule includes multiple primary CourseOfferings and a CourseOffering can appear on multiple schedules as a primary. Where would a Student’s grade for a primary CourseOffering “live”? It cannot be stored in Schedule because a Schedule contains multiple primary CourseOfferings. It cannot be stored in CourseOffering because the same CourseOffering can appear on multiple Schedules as primary. Grade is really an attribute of the relationship between a Schedule and a primary CourseOffering.

The same is true of the status of a CourseOffering, primary or alternate, on a particular Schedule.

Thus, association classes were created to contain such information. Two classes related by generalization were created to leverage the similarities between what must be maintained for primary and alternate CourseOfferings. Remember, Students can only enroll in and receive a grade in a primary CourseOffering, not an alternate.

Example: Association Class Design



Example: Association Class Design

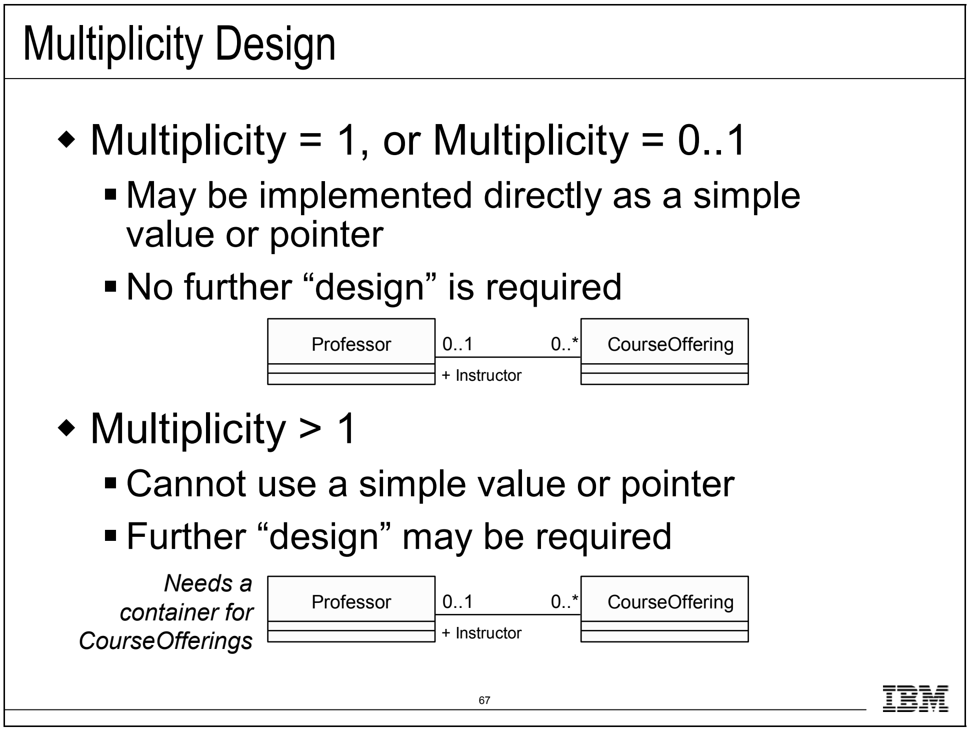
If there are attributes on the association itself (represented by "association classes"), create a design class to represent the “association class,” with the appropriate attributes. Interpose this class between the other two classes, and by establishing associations with appropriate multiplicity between the association class and the other two classes.

The above example demonstrates how an association class can be further designed.

When defining the navigability between the resulting classes, it was decided not to provide navigation directly to CourseOffering from Schedule (must go through PrimaryScheduleOfferingInfo).

Note: Association class design needs to be done only if the implementation does not directly support association classes and an exact visual model of the implementation is required. The above example is hypothetical. It is not included in the Course Registration Model provided with the course since the additional design details because only complicated the Design Model.

Multiplicity Design



Multiplicity Design

- Multiplicity = 1, or Multiplicity = 0..1
  - May be implemented directly as a simple value or pointer
  - No further “design” is required

Professor
0..1
0..*
CourseOffering
+ Instructor

- Multiplicity > 1
  - Cannot use a simple value or pointer
  - Further “design” may be required

Needs a container for CourseOfferings

Professor
0..1
0..*
CourseOffering
+ Instructor
67

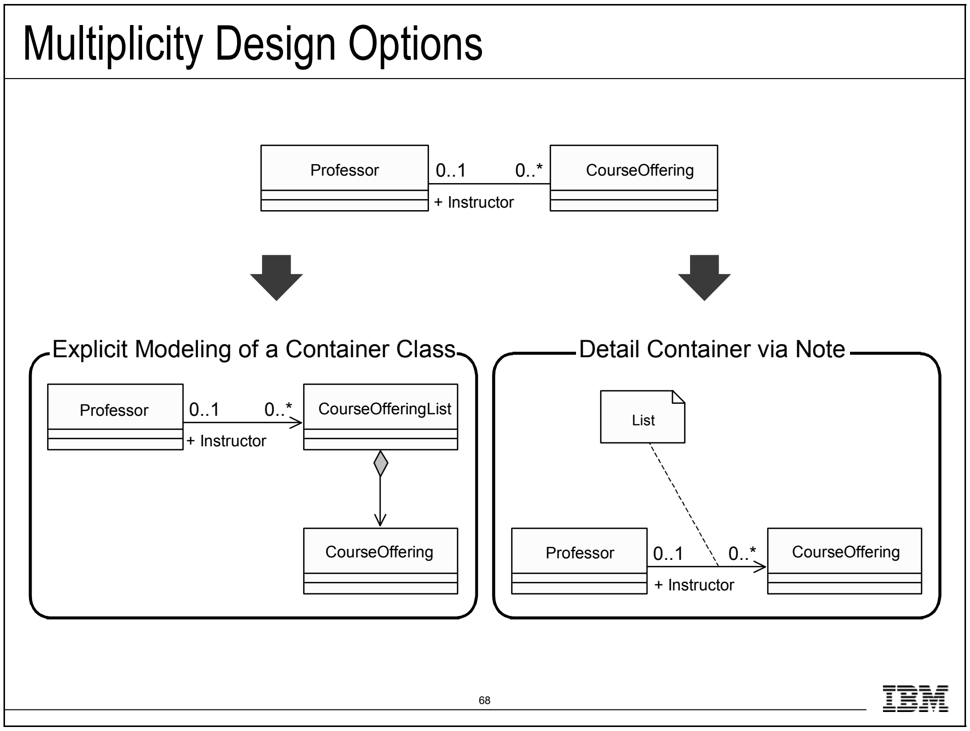
Multiplicity is the number of instances that participate in an association. Initial estimates of multiplicity made during analysis must be updated and refined during design.

All association and aggregation relationships must have multiplicity specified.

For associations with a multiplicity of 1 or 0..1, further design is not usually required, as the relationship can be implemented as a simple value or a pointer.

For associations with a multiplicity upper bound that is greater than 1, additional decisions need to be made. This is usually designed using “container” classes. A container class is a class whose instances are collections of other objects. Common container classes include sets, lists, dictionaries, stacks, and queues.

Multiplicity Design Options

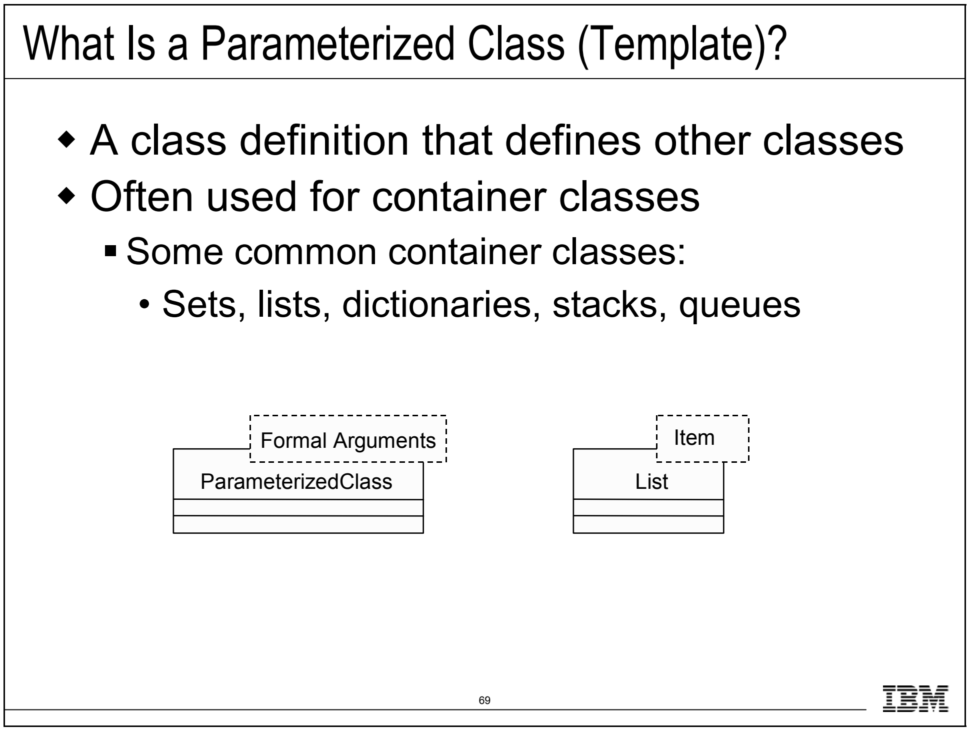


Multiplicity Design Options

The design of a relationship with a multiplicity greater than one can be modeled in multiple ways. You can explicitly model a container class, or you can just indicate what kind of container class will be used. The latter approach keeps the diagrams from getting too cluttered with very detailed implementation decisions. However, the former approach may be useful if you want to do code generation from your model.

Another option that is a more refined version of the first approach is to use a parameterized class (template) as the container class. This is discussed in more detail on the next few slides.

What Is a Parameterized Class (Template)?



What Is a Parameterized Class (Template)?

- A class definition that defines other classes
- Often used for container classes
  - Some common container classes:
    - Sets, lists, dictionaries, stacks, queues

In the UML, parameterized classes are also known as templates.

The formal arguments are expressed as a parameter list (for example: [T: class, size: Int]).

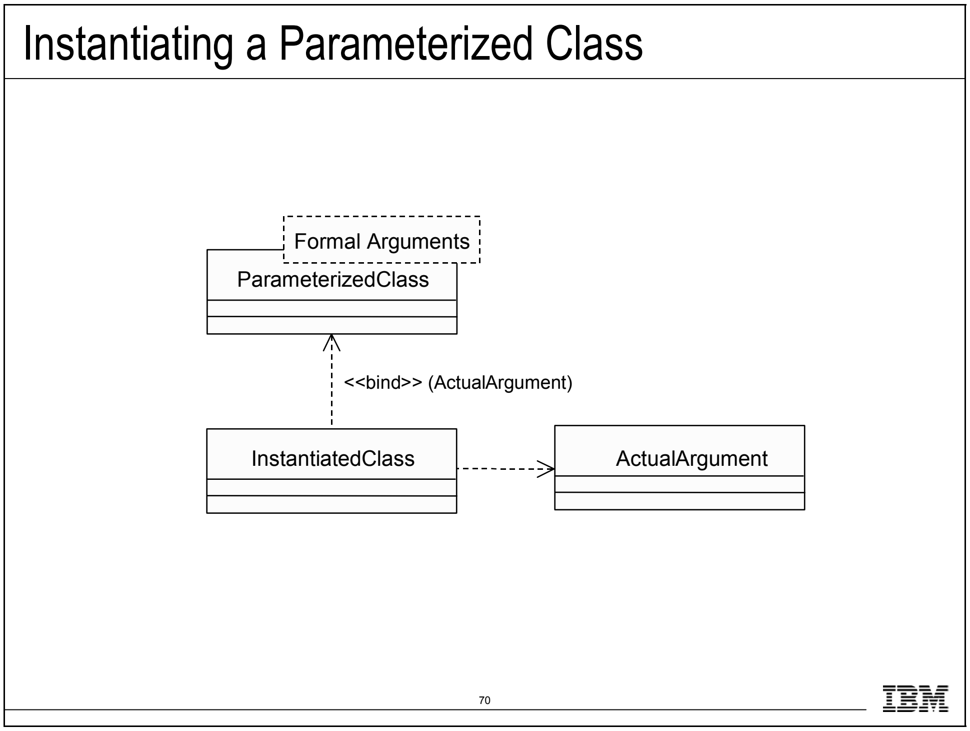
A parameterized class cannot be used directly; it must be instantiated to be used. It is instantiated by supplying actual arguments for the formal arguments. Instantiation is discussed on the next slide.

Container classes can be modeled as parameterized classes. Common container classes, like List, from existing implementation libraries are often just noted as a standard approach, and are not modeled at all. The standard mechanism is noted in the Design Document, or as a note on the Class diagram.

Consider your implementation language when deciding to use these. Parameterized classes are not supported in every language. For example, C++ supports templates, while Java does not.

See the language-specific appendices for more information.

Instantiating a Parameterized Class

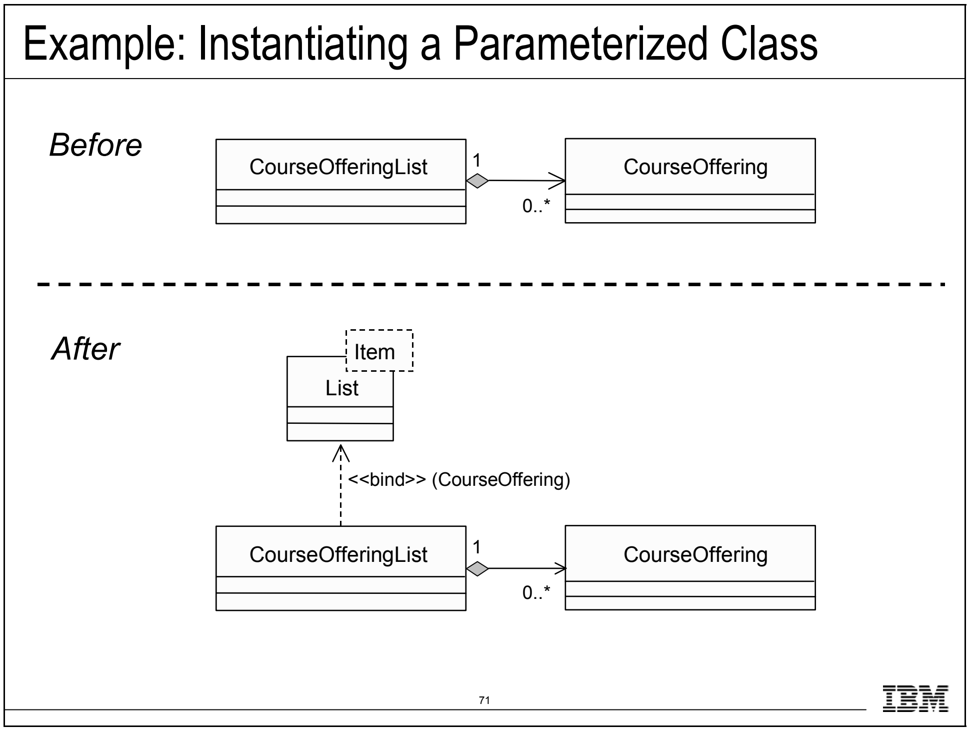


Instantiating a Parameterized Class

Using a parameterized class requires that you specify a bound element (that is, actual arguments for the parameterized classes' formal arguments).

A dependency stereotyped with <<bind>> is used to specify that the source instantiates the parameterized class using the actual parameters.

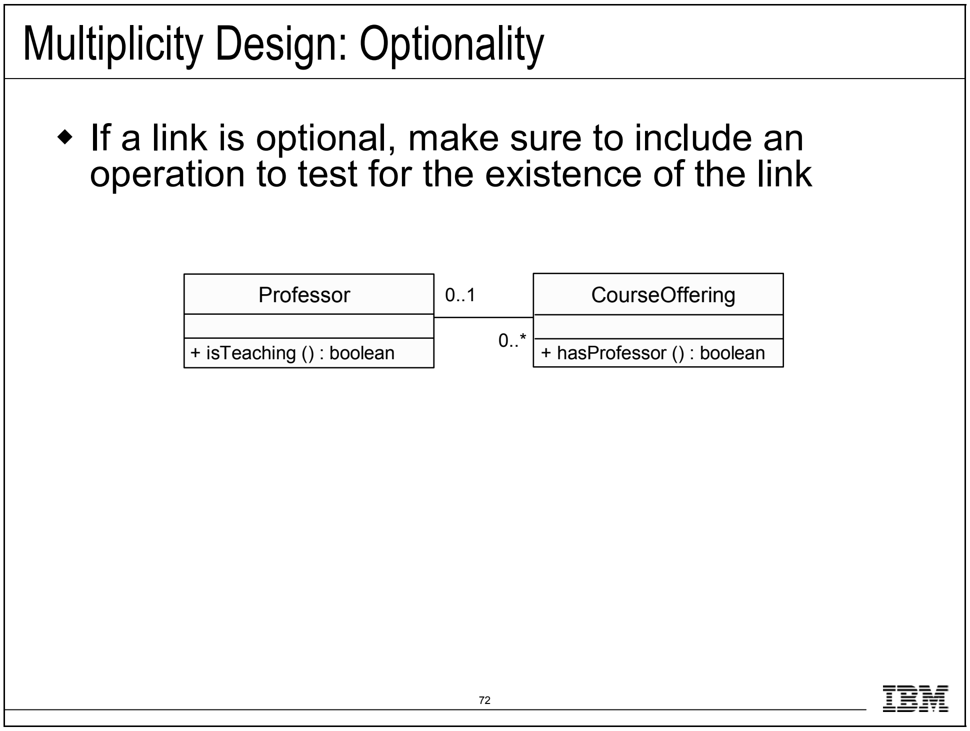
Example: Instantiating a Parameterized Class



Example: Instantiating a Parameterized Class

Initially, we just defined a CourseOfferingList. During Detailed Design, it is decided that the List class will be used as a base class for the CourseOfferingList.

Multiplicity Design: Optionality



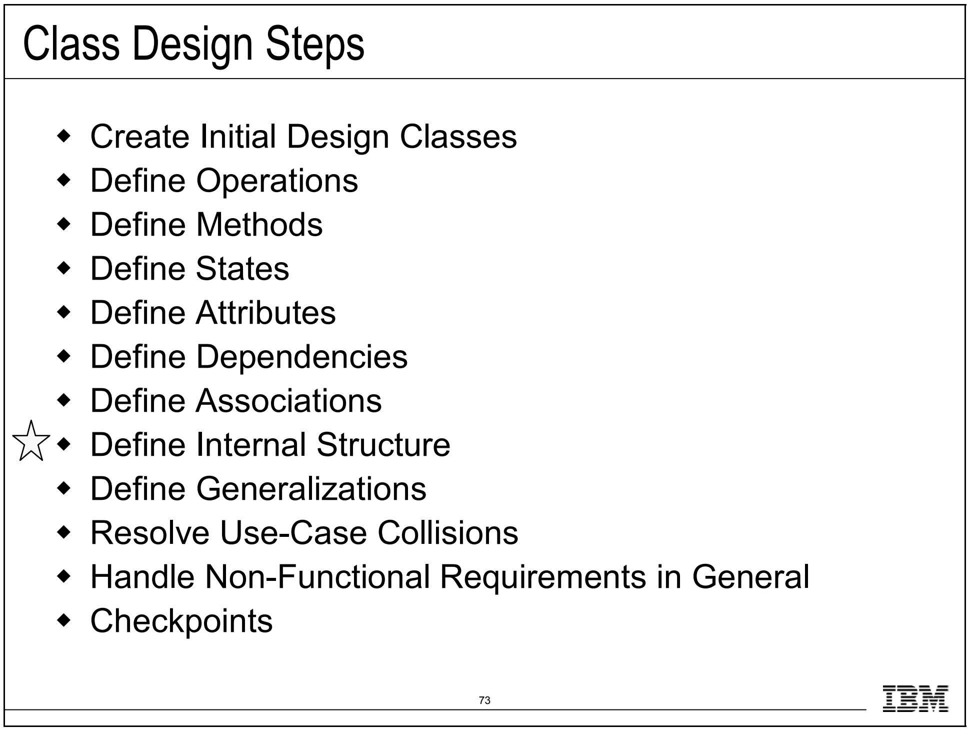
Multiplicity Design: Optionality

- If a link is optional, make sure to include an operation to test for the existence of the link

If a link is optional, an operation to test for the existence of the link should be added to the class.

For example, if a Professor can be on sabbatical, a suitable operation should be included in the Professor class to test for the existence of the CourseOffering link.

Define Internal Structure



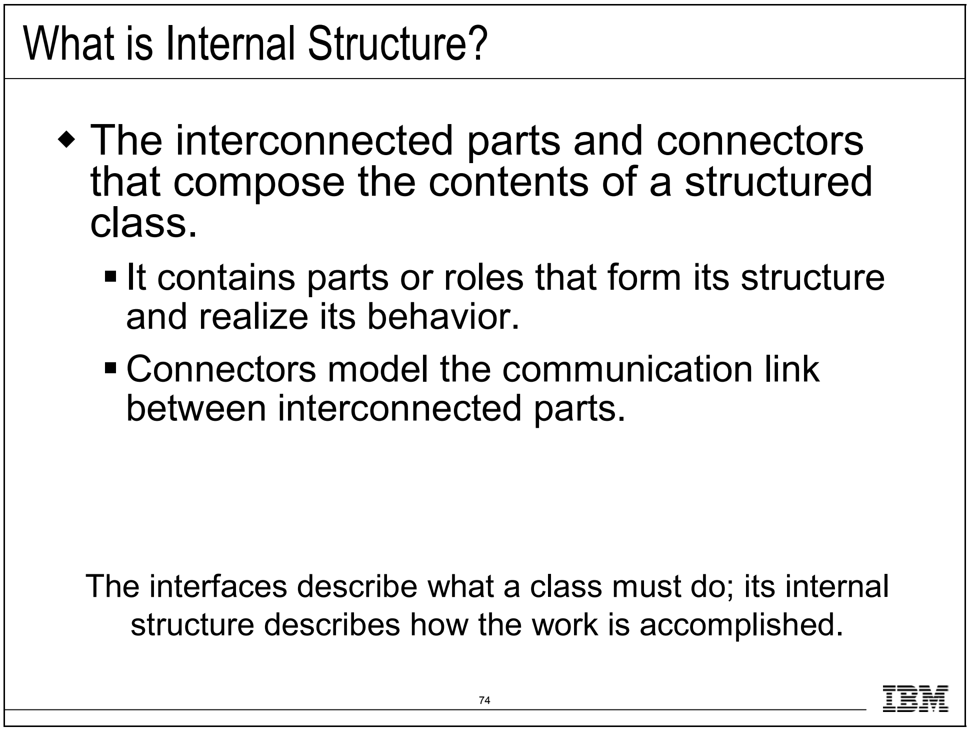
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- ☆ Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

73

In Analysis, inheritance that was intrinsic to the problem domain may have been defined. **Class Design** is where generalizations are defined to improve/ease the implementation. Design is the real activity of inventing inheritance.

What is Internal Structure?

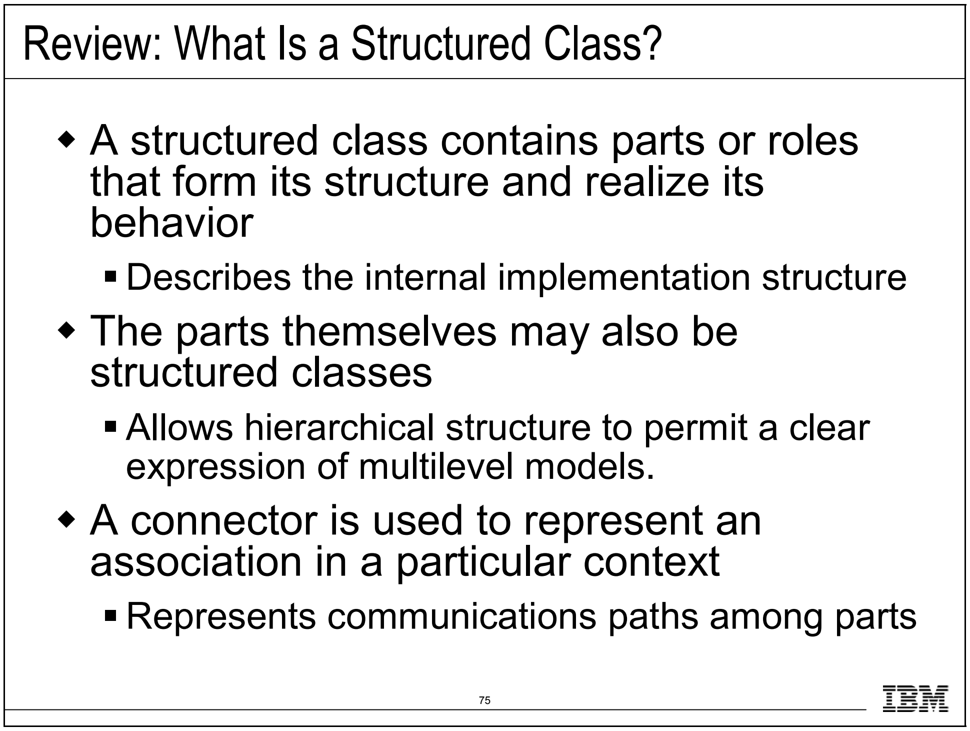


What is Internal Structure?

- The interconnected parts and connectors that compose the contents of a structured class.
  - It contains parts or roles that form its structure and realize its behavior.
  - Connectors model the communication link between interconnected parts.

The interfaces describe what a class must do; its internal structure describes how the work is accomplished.

Review: What Is a Structured Class?



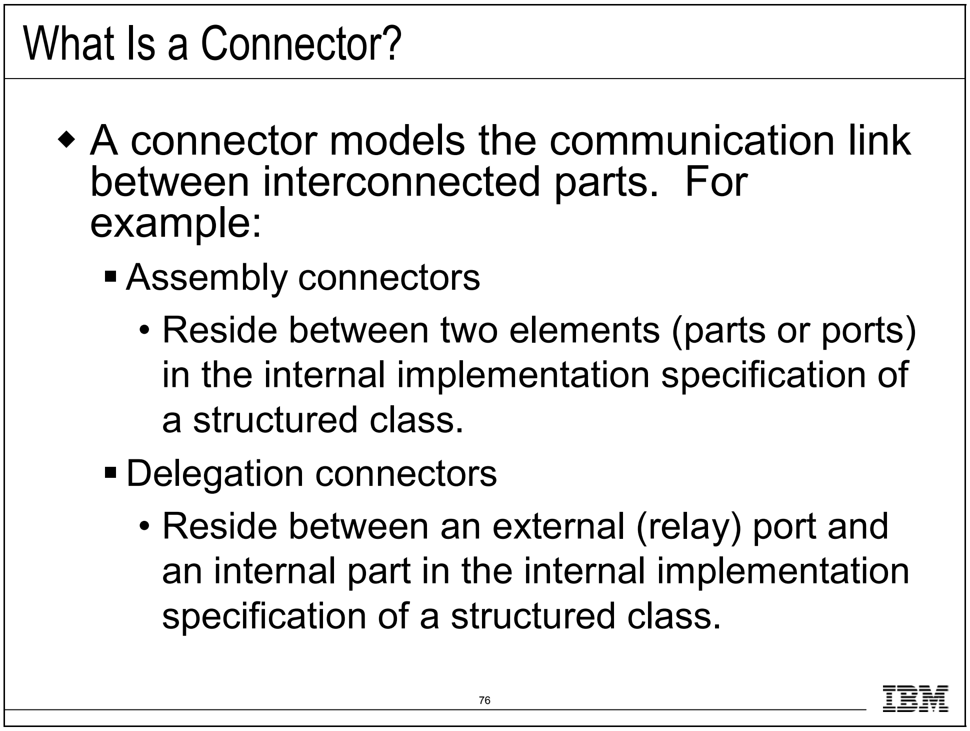
Review: What Is a Structured Class?

- A structured class contains parts or roles that form its structure and realize its behavior
  - Describes the internal implementation structure
- The parts themselves may also be structured classes
  - Allows hierarchical structure to permit a clear expression of multilevel models.
- A connector is used to represent an association in a particular context
  - Represents communications paths among parts

75

A **role** is a constituent element of a structured class that represents the appearance of an instance (or possibly set of instances) within the context defined by the structured class.

What Is a Connector?



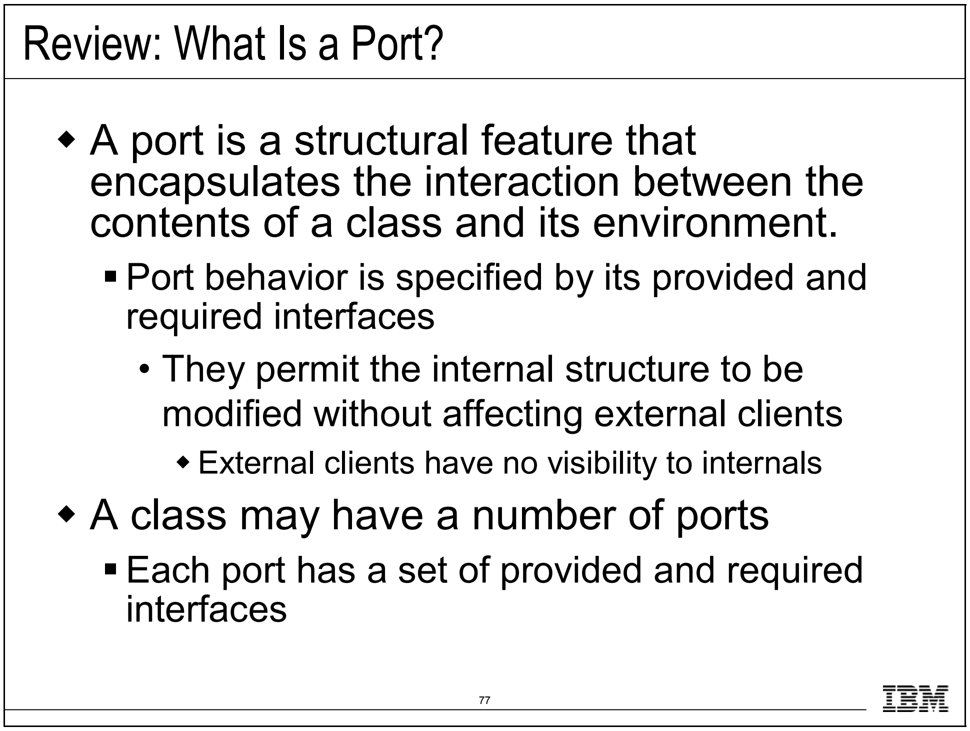
What Is a Connector?

- A connector models the communication link between interconnected parts. For example:
  - Assembly connectors
    - Reside between two elements (parts or ports) in the internal implementation specification of a structured class.
  - Delegation connectors
    - Reside between an external (relay) port and an internal part in the internal implementation specification of a structured class.

76

For delegation connectors, environment connections to the external port are treated as going to the internal element at the other end of the delegation connector.

Review: What Is a Port?



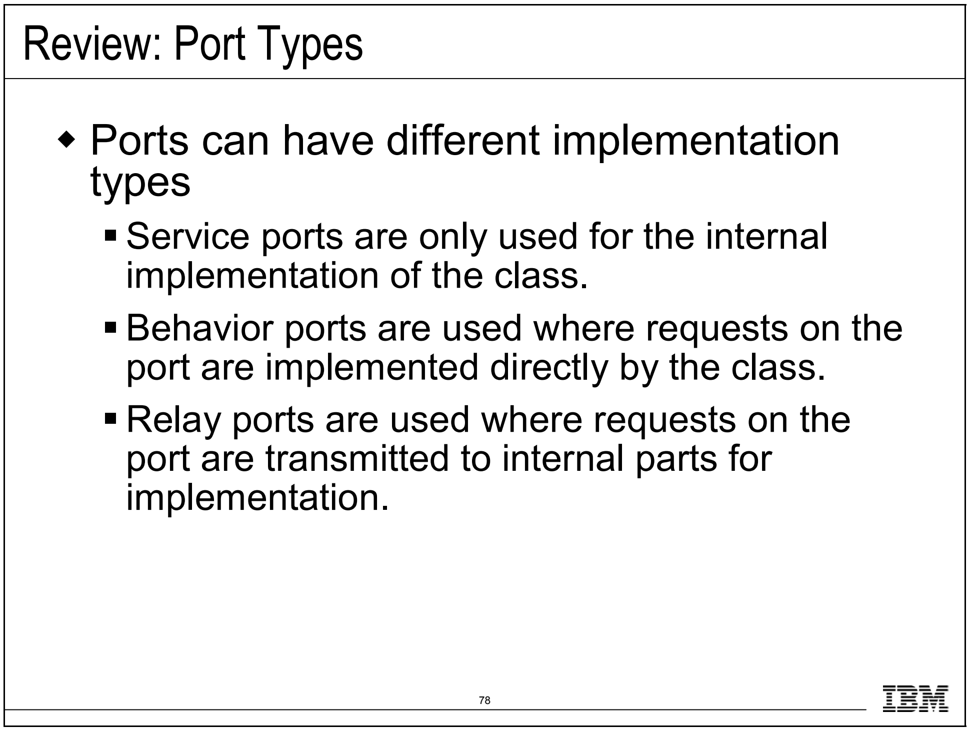
Review: What Is a Port?

- A port is a structural feature that encapsulates the interaction between the contents of a class and its environment.
  - Port behavior is specified by its provided and required interfaces
    - They permit the internal structure to be modified without affecting external clients
      - External clients have no visibility to internals
- A class may have a number of ports
  - Each port has a set of provided and required interfaces

Since the port is a structural element, it's created and destroyed along with its structured class.

Another class connected to a port may request the provided services from the owner of the port but must also be prepared to supply the required services to the owner.

Review: Port Types



Review: Port Types

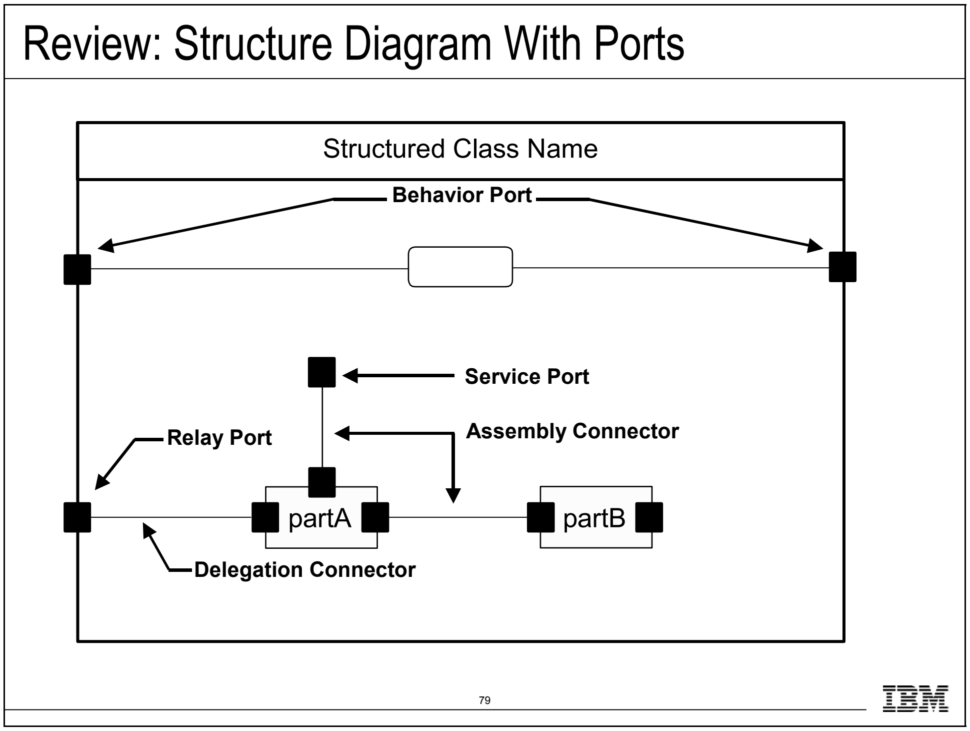
- Ports can have different implementation types
  - Service ports are only used for the internal implementation of the class.
  - Behavior ports are used where requests on the port are implemented directly by the class.
  - Relay ports are used where requests on the port are transmitted to internal parts for implementation.

78

The use of **service ports** are rare because the main purpose of ports is to encapsulate communication with the environment. These ports are located inside the class boundary.

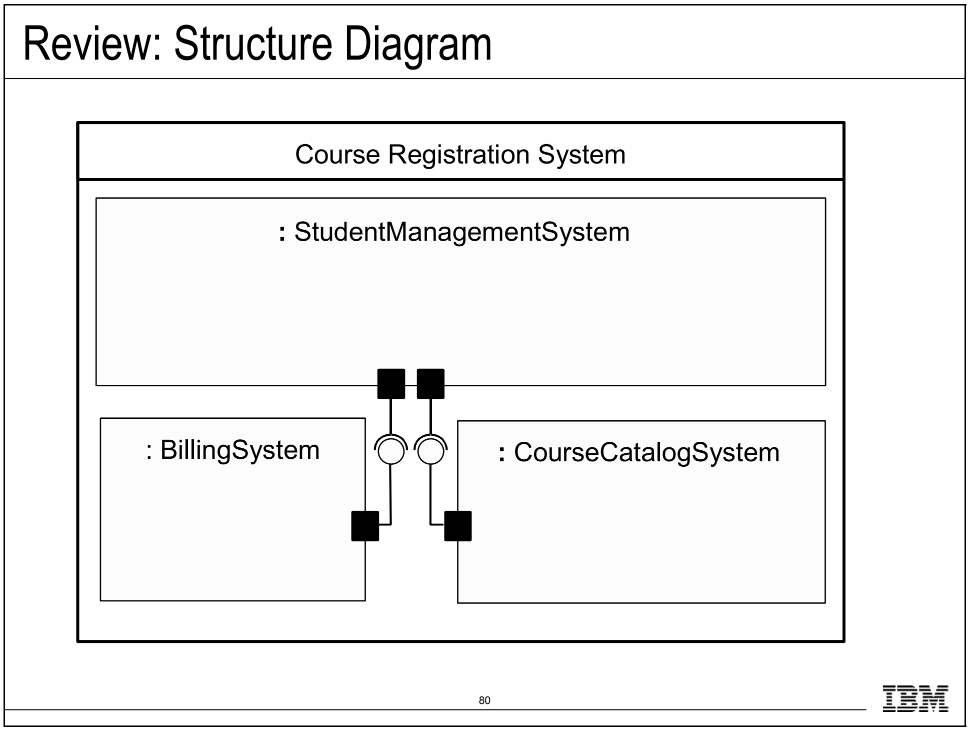
Behavior ports are shown by a line from the port to a small state symbol (a rectangle with rounded corners). This is meant to suggest a state machine, although other forms of behavior implementation are also permitted.

Review: Structure Diagram With Ports



Review: Structure Diagram With Ports

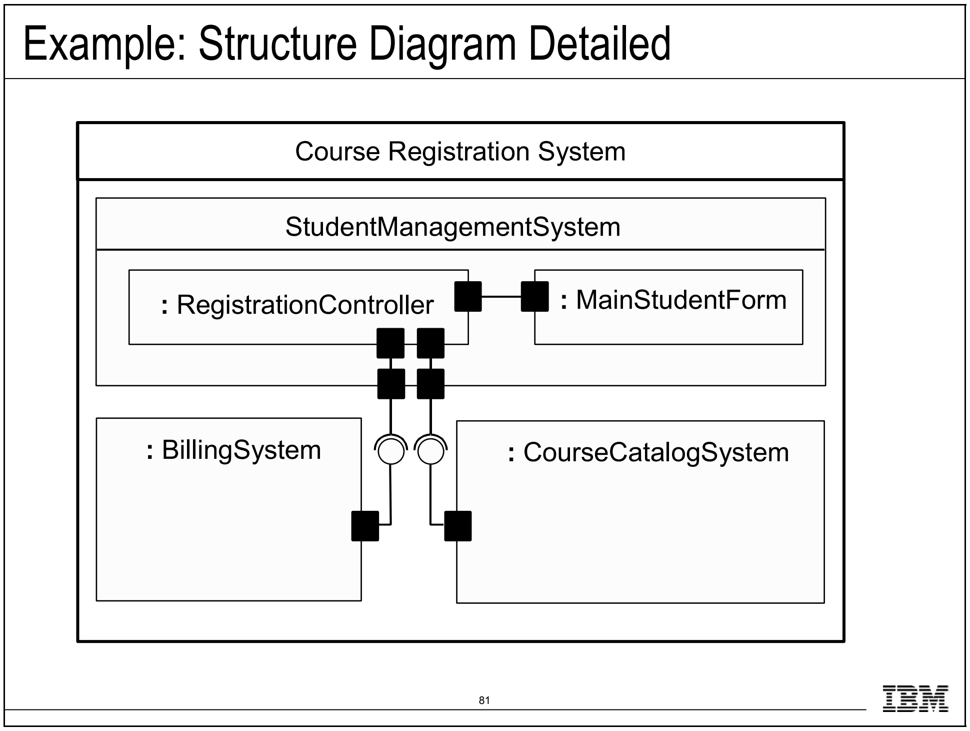
Review: Structure Diagram



Review: Structure Diagram

As the system is further decomposed, each of the parts may be a structured class which contains parts themselves. This is a very effective method to visualize the system architecture.

Example: Structure Diagram Detailed



Example: Structure Diagram Detailed

Course Registration System

StudentManagementSystem

: RegistrationController

: MainStudentForm

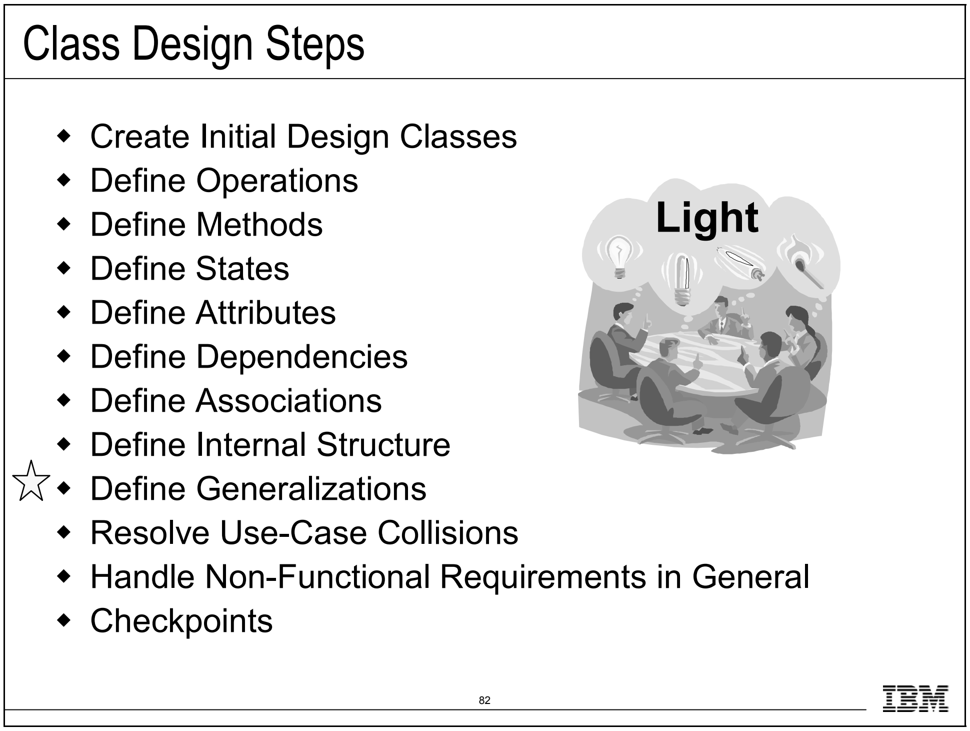
: BillingSystem

: CourseCatalogSystem

81

As the system is further decomposed, each of the parts may be a structured class which contains parts themselves. This is a very effective method to visualize the system architecture.

Define Generalizations



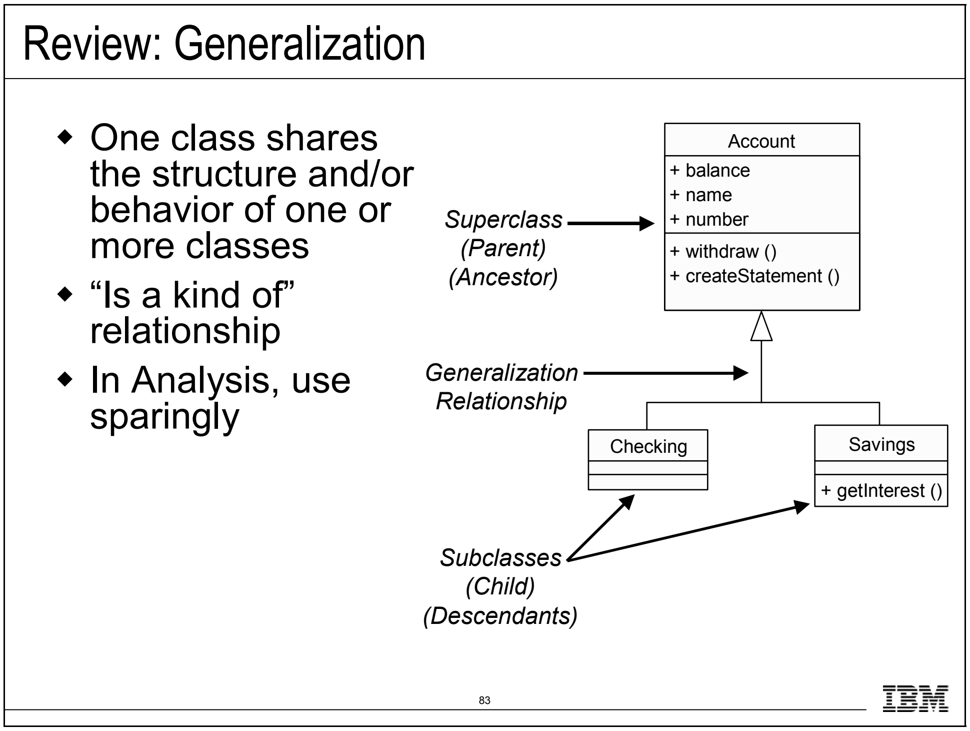
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- ☆ Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Light
82
IBM

In Analysis, inheritance that was intrinsic to the problem domain may have been defined. **Class Design** is where generalizations are defined to improve/ease the implementation. Design is the real activity of inventing inheritance.

Review: Generalization



Review: Generalization

- One class shares the structure and/or behavior of one or more classes
- “Is a kind of” relationship
- In Analysis, use sparingly

Superclass (Parent) (Ancestor)

Account
+ balance
+ name
+ number
+ withdraw ()
+ createStatement ()

Generalization Relationship

Checking

Savings
+ getInterest ()

Subclasses (Child) (Descendants)

83

As discussed in Concepts of Object Orientation, generalization is a relationship among classes where one class shares the structure and/or behavior of one or more classes. This slide is repeated here for review purposes.

Generalization refines a hierarchy of abstractions in which a sub-class inherits from one or more super-classes.

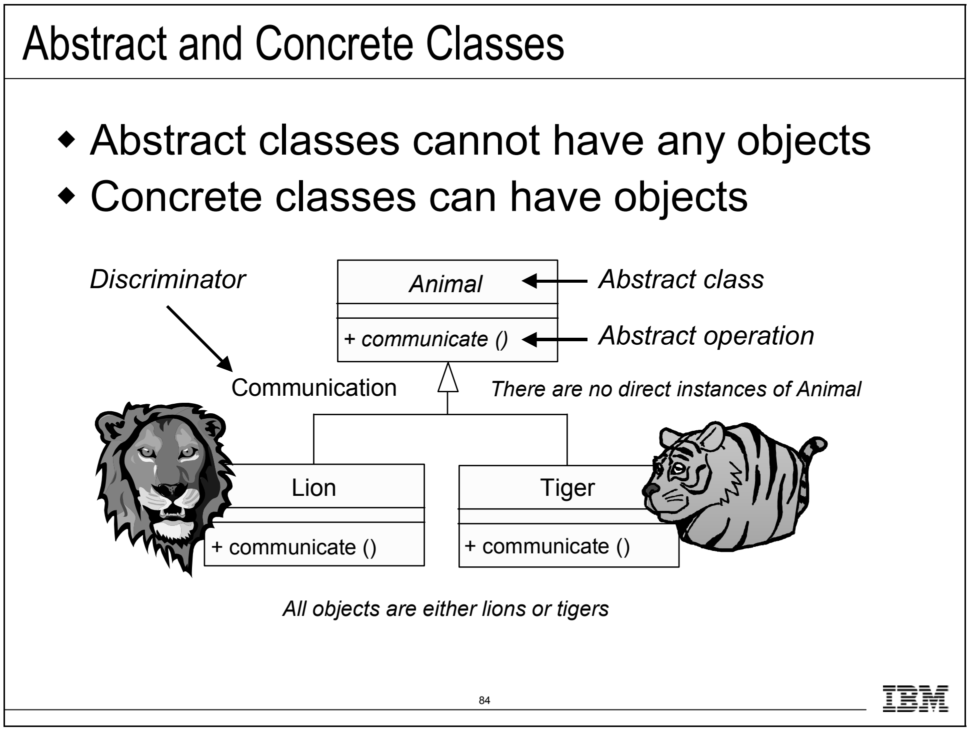
Generalization is an “is a kind of” relationship. You should always be able to say that your generalized class “is a kind of” the parent class.

The terms “ancestor” and “descendent” may be used instead of “super-class” and “sub-class”.

In Analysis, generalization should be used only to model shared behavioral semantics (that is, generalization that passes the “"is a"” test). Generalization to promote and support reuse is determined in Design. In Analysis, the generalization should be used to reflect shared definitions/semantics. This promotes “brevity of expression.” The use of generalization makes the definitions of the abstractions easier to document and understand.

When generalization is found, a common super-class is created to contain the common attributes, associations, aggregations, and operations. The common behavior is removed from the classes that are to become sub-classes of the common super-class. A generalization relationship is drawn from the sub-class to the super-class.

Abstract and Concrete Classes



Abstract and Concrete Classes

- Abstract classes cannot have any objects
- Concrete classes can have objects

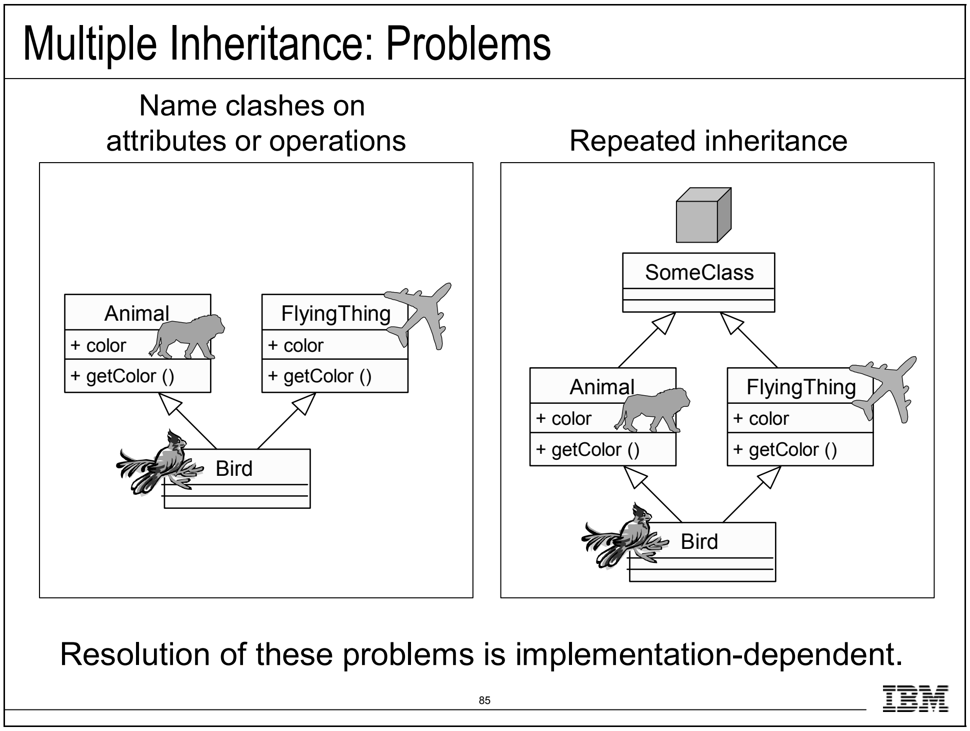
A class that exists only for other classes to inherit from it is an **abstract** class. Classes that are to be used to instantiate objects are **concrete** classes.

An operation can also be tagged as abstract. This means that no implementation exists for the operation in the class where it is specified. A class that contains at least one abstract operation must be an abstract class. Classes that inherit from an abstract class must provide implementations for the abstract operations. Otherwise, the operations are considered abstract within the subclass, and the subclass is considered abstract, as well. Concrete classes have implementations for all operations.

In the UML, you designate a class as abstract by putting the class name in italics. For abstract operations, you put the operation signature in italics. The name of the abstract item can also be shown in italics.

A **discriminator** can be used to indicate on what basis the generalization/specialization occurred. A discriminator describes a characteristic that differs in each of the subclasses. In the above example, we generalized/specialized in the area of communication.

Multiple Inheritance: Problems



Multiple Inheritance: Problems

Name clashes on attributes or operations

Repeated inheritance

Resolution of these problems is implementation-dependent.

In practice, multiple inheritance is a complex design problem and may lead to implementation difficulties.

In general, multiple inheritance causes problems if any of the multiple parents have structure or behavior that overlaps. If the class inherits from several classes, you must check how the relationships, operations, and attributes are named in the ancestors.

Specifically, there are two issues associated with multiple inheritance:

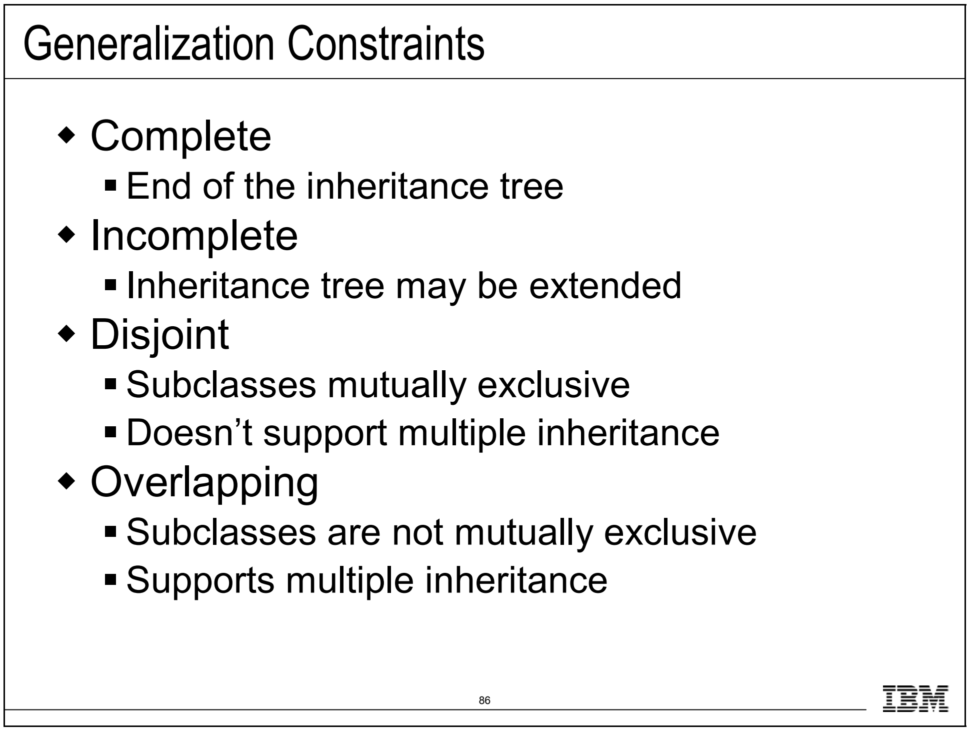
Name collisions: Both ancestors have attributes and/or operations with the same name. If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.

Repeated inheritance: The same ancestor is being inherited by a descendant more than once. When this occurs, the inheritance hierarchy will have a "diamond shape" as shown above. The descendents end up with multiple copies of an ancestor.
If you are using repeated inheritance, you must have a clear definition of its semantics; in most cases this is defined by the programming language supporting the multiple inheritance.

In general, the programming language rules governing multiple inheritance are complex, and often difficult to use correctly. Therefore, it is recommended that you use multiple inheritance only when needed and always with caution.

Note: You can use delegation as a workaround to multiple inheritance problems. Delegation is described later in this module.

Generalization Constraints



Generalization Constraints

- Complete
  - End of the inheritance tree
- Incomplete
  - Inheritance tree may be extended
- Disjoint
  - Subclasses mutually exclusive
  - Doesn’t support multiple inheritance
- Overlapping
  - Subclasses are not mutually exclusive
  - Supports multiple inheritance

The UML defines four standard constraints for generalization relationships:

Complete: This constraint indicates the end of the inheritance hierarchy. All children in the generalization relationship have been defined in the model. No more children can be defined. The “leaves” of the inheritance hierarchy cannot be specialized any further. Use the Complete constraint to explicitly note that the generalization hierarchy has not been fully specified in the model.

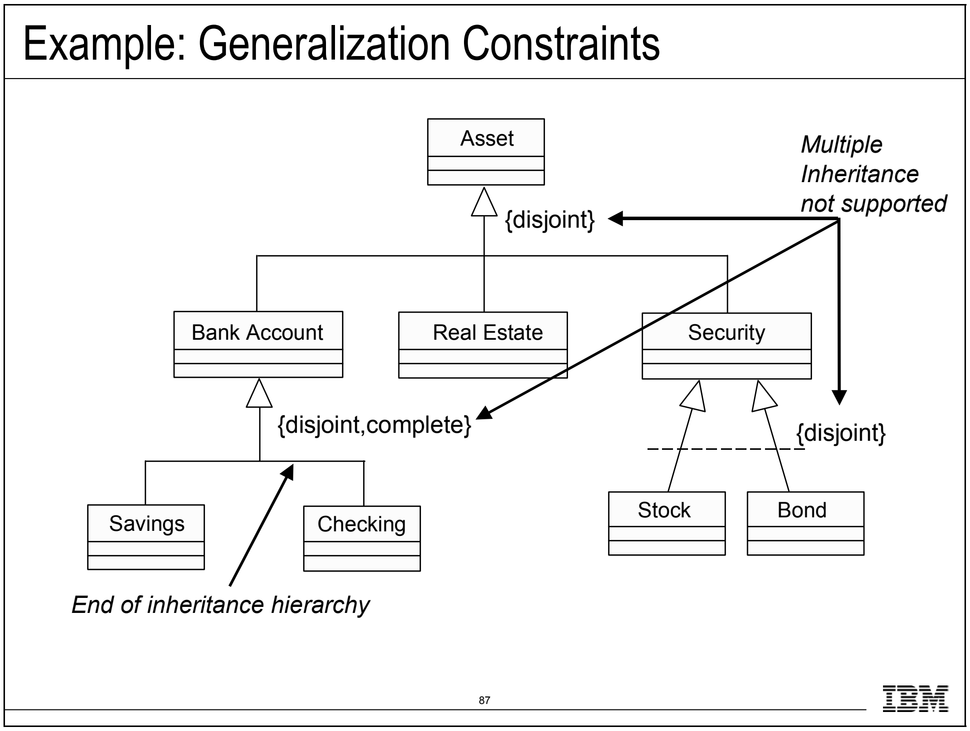
Incomplete: All children in the generalization relationship have not been defined in the model. More children may be defined. The “leaves” of the inheritance hierarchy may be specialized. Use the Incomplete constraint to explicitly note that the generalization hierarchy has not been fully specified in the model.

The following two constraints only apply in the context of multiple inheritance:

Disjoint: An object of the parent cannot have more than one of the children as its type (that is, subclasses are mutually exclusive). Disjoint is used to support the modeling of static classification, where an object cannot change its type at run-time.

Overlapping: An object of the parent may have more than one of the children as its type. Overlapping is used to support the modeling of dynamic classification, where an object can change its type at run-time. It shows the potential types of an object.

Example: Generalization Constraints



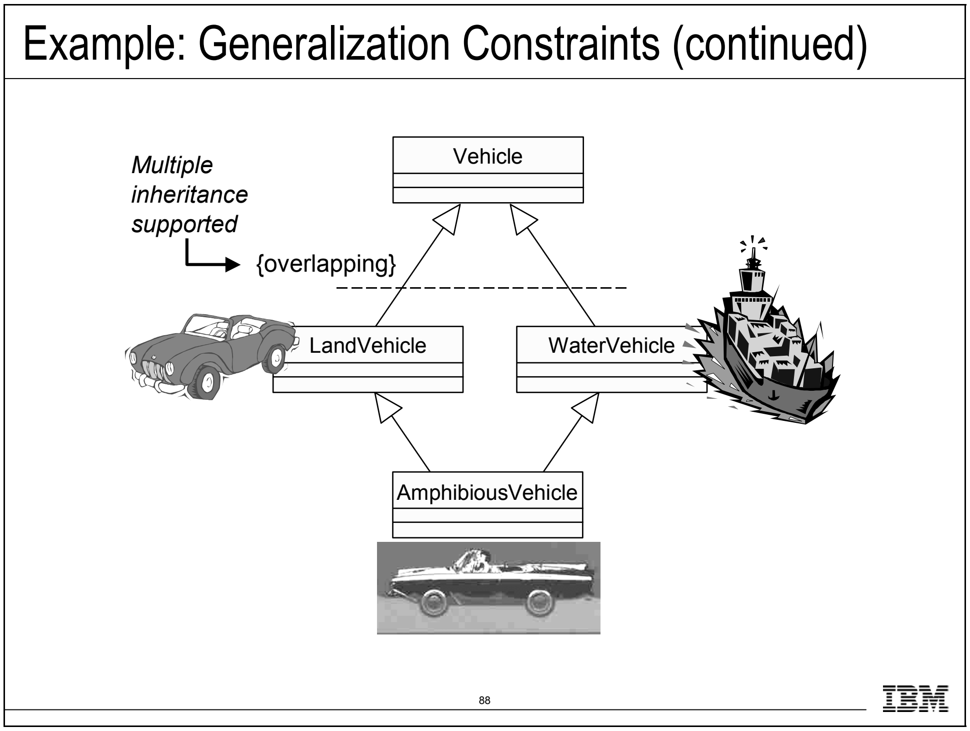
Example: Generalization Constraints

This example demonstrates the use of the Complete and Disjoint constraints:

Complete: The Savings and Checking classes cannot be specialized (a generalization relationship cannot be defined in which they are the parent). These classes (and any siblings) mark the end of the inheritance hierarchy.

Disjoint: A BankAccount object cannot be both a Savings and a Checking account (that is, no multiple inheritance where parents in the multiple inheritance are the children of BankAccount).

Example: Generalization Constraints (continued)



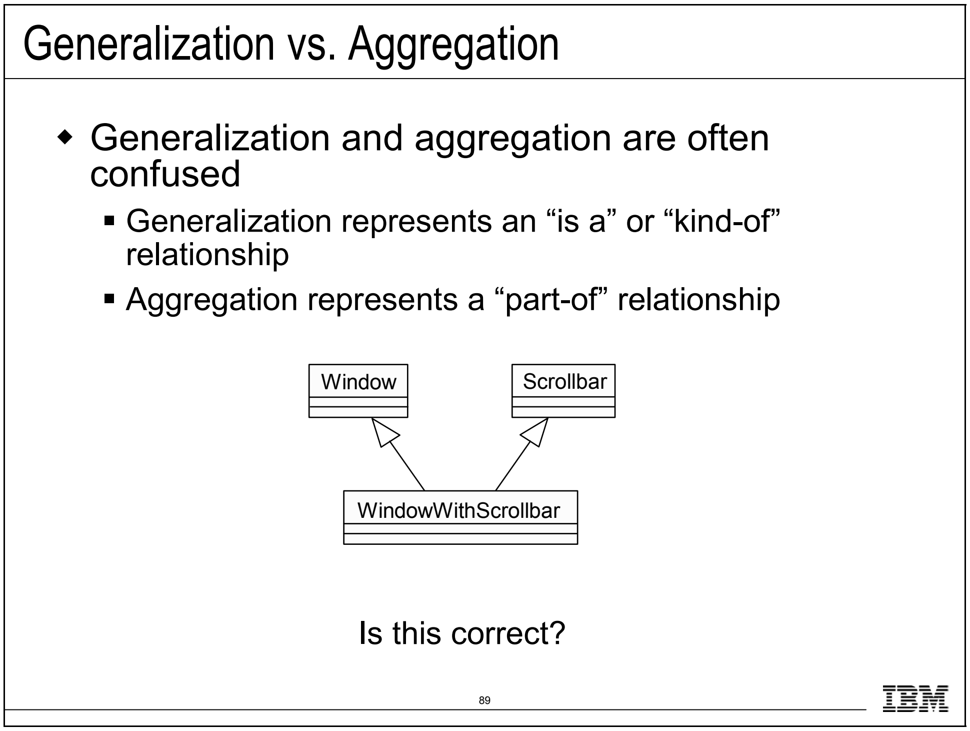
Example: Generalization Constraints (continued)

Multiple inheritance supported

{overlapping}

This example demonstrates the use of the overlapping constraint. The AmphibiousVehicle class can inherit from both LandVehicle and WaterVehicle, which both inherit from Vehicle.

Generalization vs. Aggregation



Generalization vs. Aggregation

- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship

Is this correct?

The key phrases “is a” and “part of” help to determine correct relationship.

With inheritance:

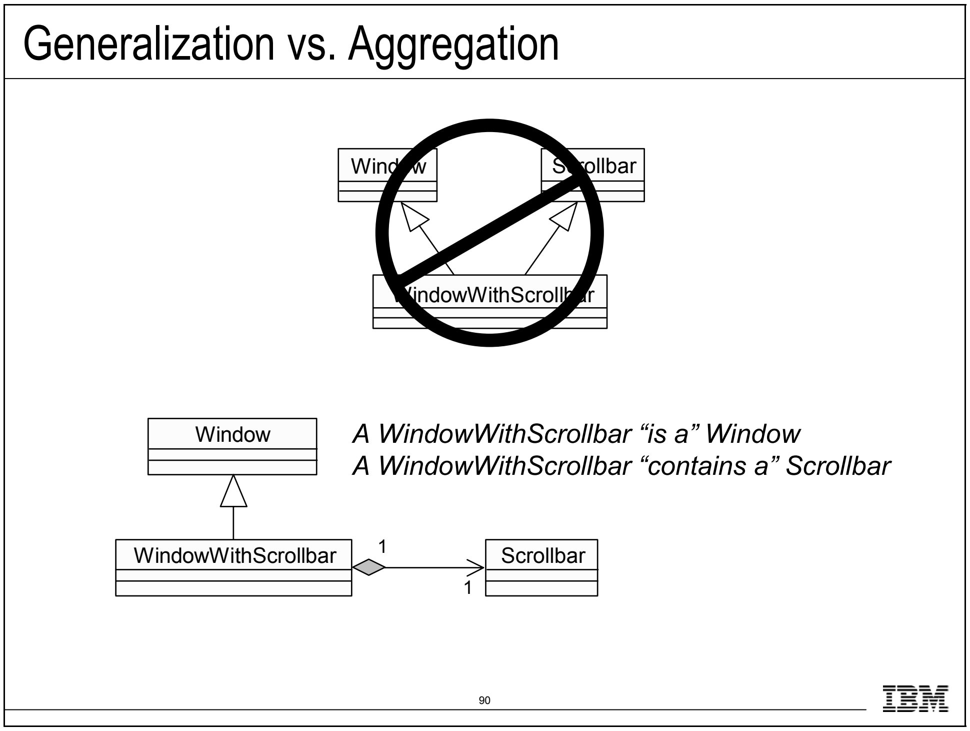
- Keywords “is a”
- One object

With Aggregation:

- Keywords “part of”
- Relates multiple objects

Is the above example a proper use of generalization? If not, what would be a better way to model the information that maintains generalization“ "is a"” semantics?

Generalization vs. Aggregation



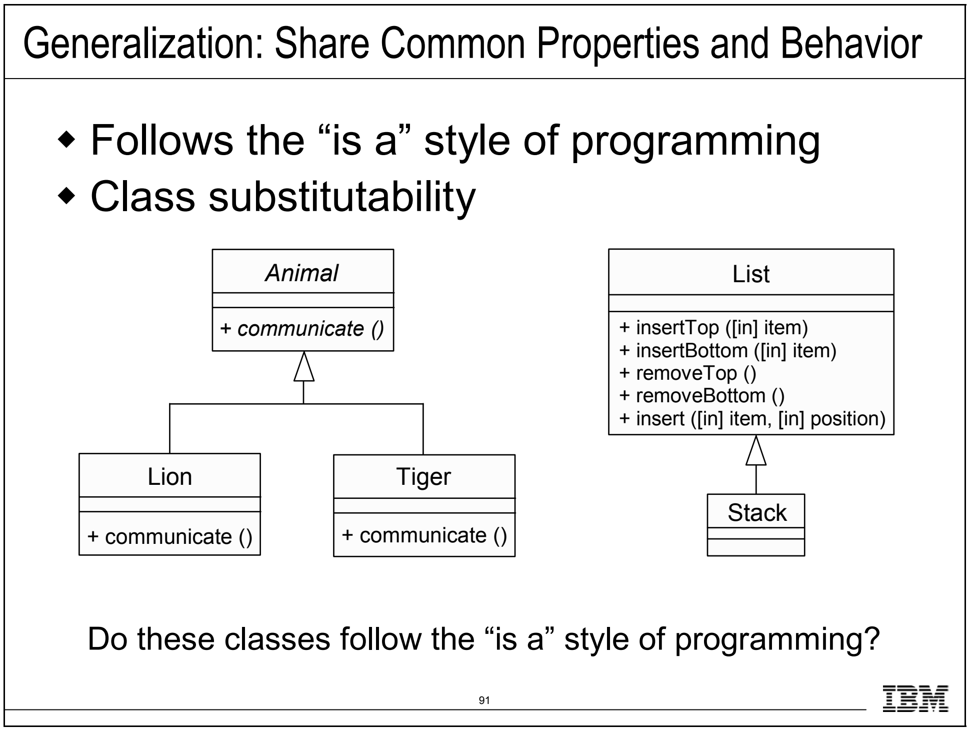
Generalization vs. Aggregation

A WindowWithScrollbar “is a” Window
A WindowWithScrollbar “contains a” Scrollbar

In this example, Scrollbar is “part of” a WindowWithScrollbar and, as such, the relationship is a composition. Scrollbar object is created/destroyed along with the WindowWithScrollbar object.

WindowWithScrollbar, on the other hand, “is a” Window, so its relationship to Window is a generalization.

Generalization: Share Common Properties and Behavior



Generalization: Share Common Properties and Behavior

- Follows the “is a” style of programming
- Class substitutability

Animal

+ communicate ()

Lion

+ communicate ()

Tiger

+ communicate ()

List

+ insertTop ([in] item)
+ insertBottom ([in] item)
+ removeTop ()
+ removeBottom ()
+ insert ([in] item, [in] position)

Stack

Do these classes follow the “is a” style of programming?

91

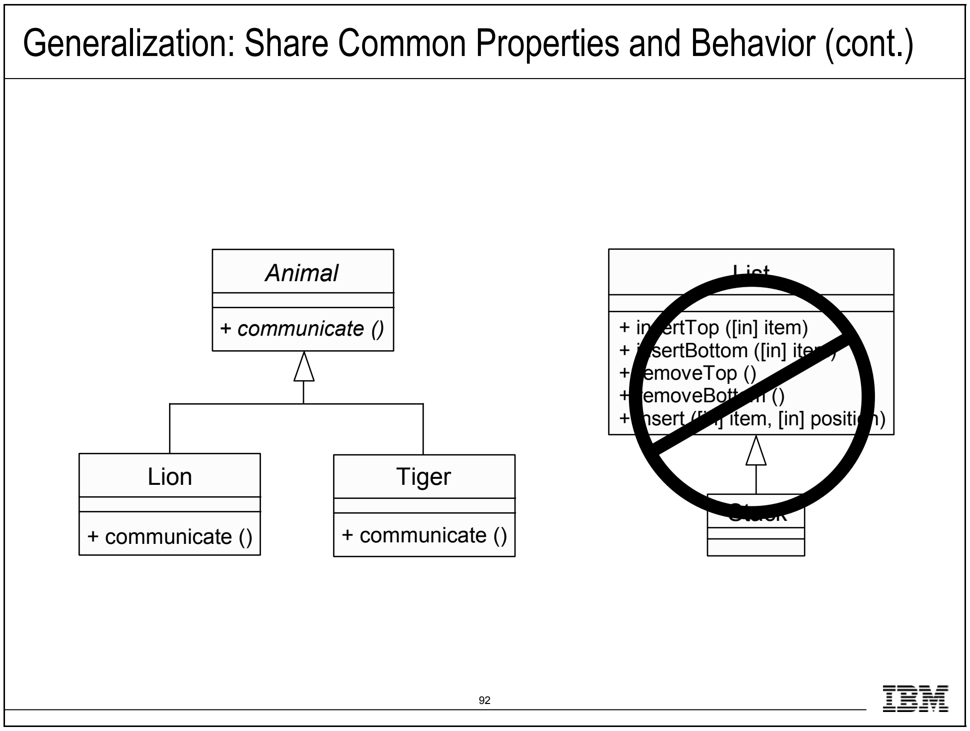
IBM

A subtype is a type of relationship expressed with inheritance. A subtype specifies that the descendent is a type of the ancestor and must follow the rules of the “is a” style of programming.

The “is a” style of programming states that the descendent "is a" type of the ancestor and can fill in for all its ancestors in any situation.

The “is a” style of programming passes the Liskov Substitution Principle, which states: “If for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.”

Generalization: Share Common Properties and Behavior (cont.)

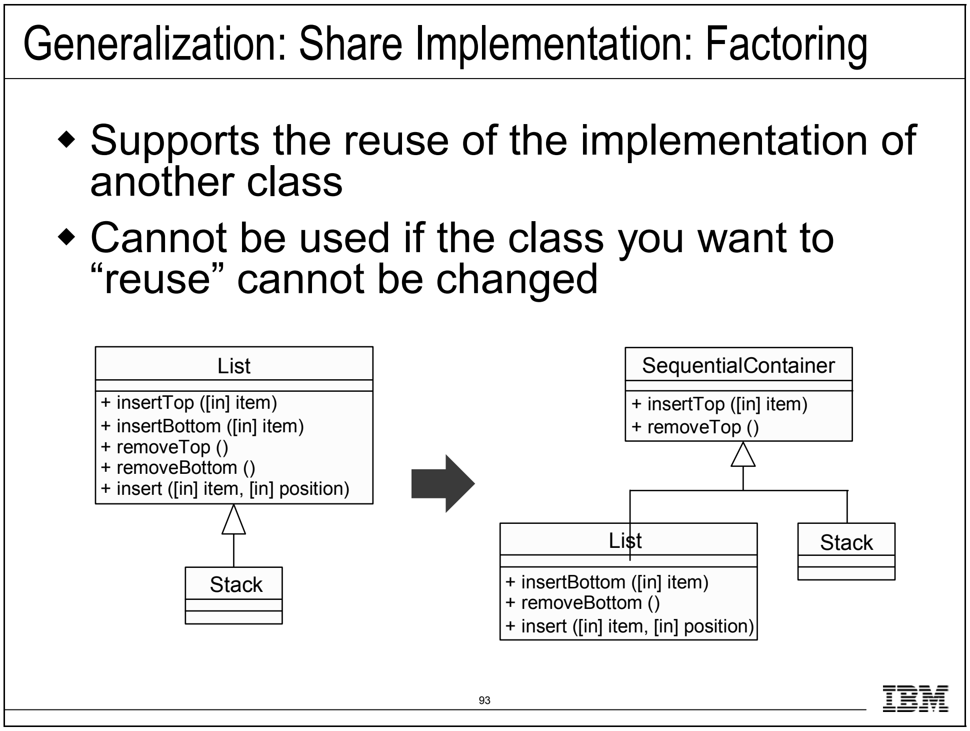


Generalization: Share Common Properties and Behavior (cont.)

The classes on the left-hand side of the diagram do follow the "is a" style of programming: a Lion is an Animal and a Tiger is an animal.

The classes on the right side of the diagram do *not* follow the “is a” style of programming: a Stack is not a List. Stack needs some of the behavior of a List but not all of the behavior. If a method expects a List, then the operation insert(position) should be successful. If the method is passed a Stack, then the insert (position) will fail.

Generalization: Share Implementation: Factoring



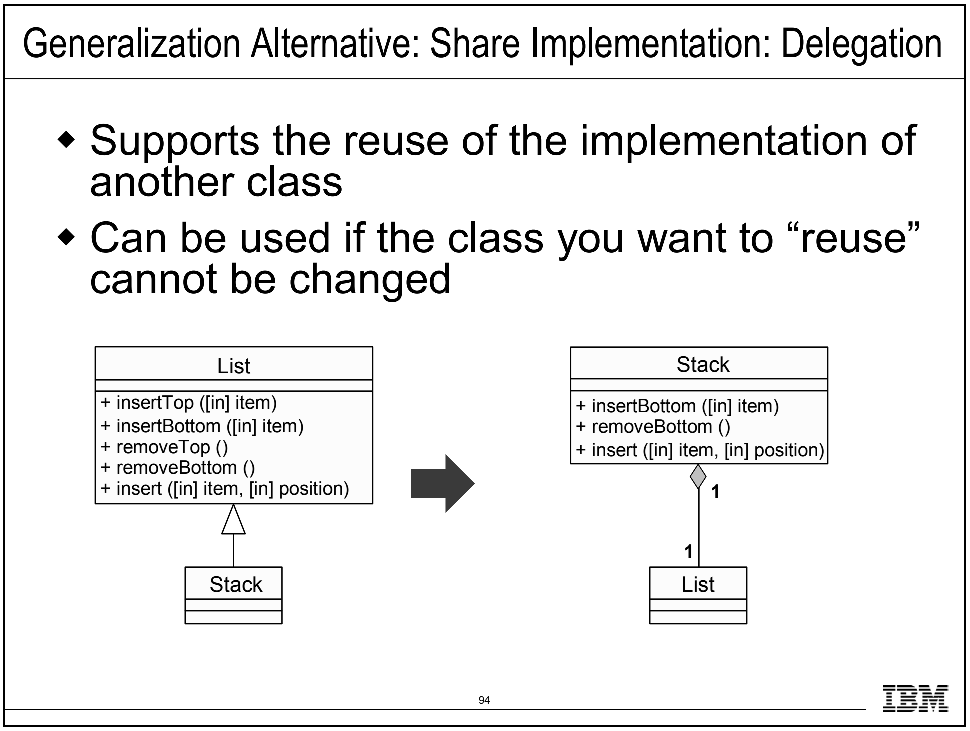
Generalization: Share Implementation: Factoring

- Supports the reuse of the implementation of another class
- Cannot be used if the class you want to "reuse" cannot be changed

Factoring is useful if there are some services provided by a class that you want to leverage in the implementation of another class.

When you factor, you extract the functionality you want to reuse and inherit from the new base class.

Generalization Alternative: Share Implementation: Delegation



Generalization Alternative: Share Implementation: Delegation

- Supports the reuse of the implementation of another class
- Can be used if the class you want to "reuse" cannot be changed

List

+ insertTop ([in] item)
+ insertBottom ([in] item)
+ removeTop ()
+ removeBottom ()
+ insert ([in] item, [in] position)

Stack

Stack

+ insertBottom ([in] item)
+ removeBottom ()
+ insert ([in] item, [in] position)

1

1

List

94

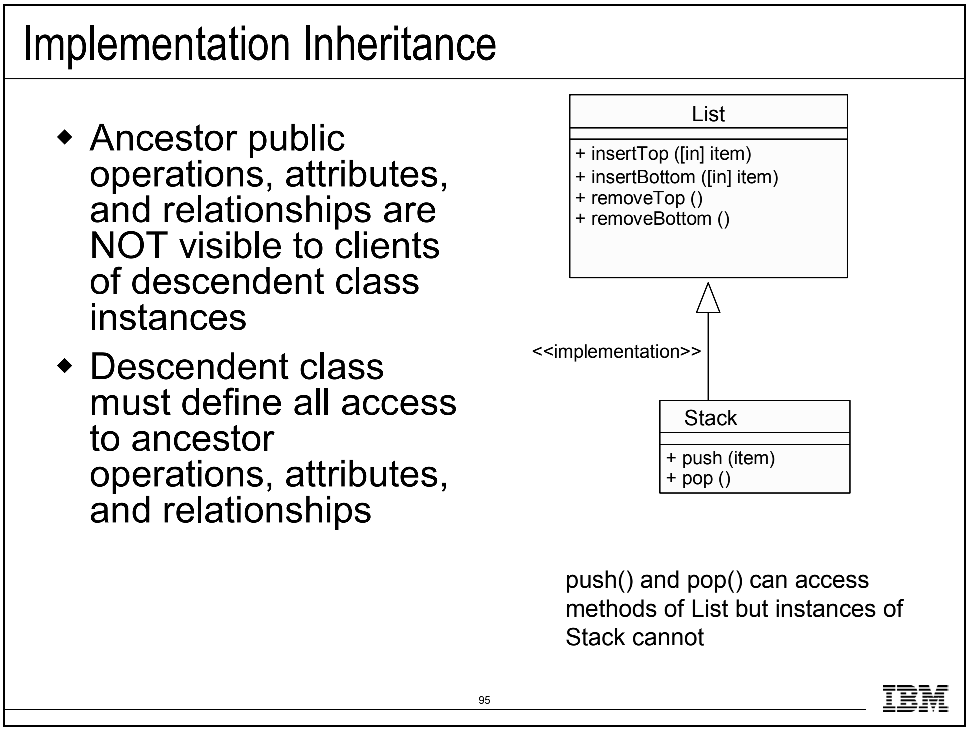
With delegation, you use a composition relationship to "reuse" the desired functionality. All operations that require the "reused" service are "passed through" to the contained class instance.

With delegation, you lose the benefit of polymorphic behavior, but you do have a model that more clearly expresses the nature of the domain (being that it is NOT "is a").

This is commonly used by mix-ins. Implementing mix-ins with composition permits run-time binding to objects rather than compile-time binding enforced by inheritance.

Note: You can also use delegation as a workaround to multiple inheritance problems discussed earlier. Have the sub-class inherit from one of the super-classes, and then use aggregation from the subclass to access the structure and behaviors of the other classes.

Implementation Inheritance



Implementation Inheritance

- Ancestor public operations, attributes, and relationships are NOT visible to clients of descendent class instances
- Descendent class must define all access to ancestor operations, attributes, and relationships

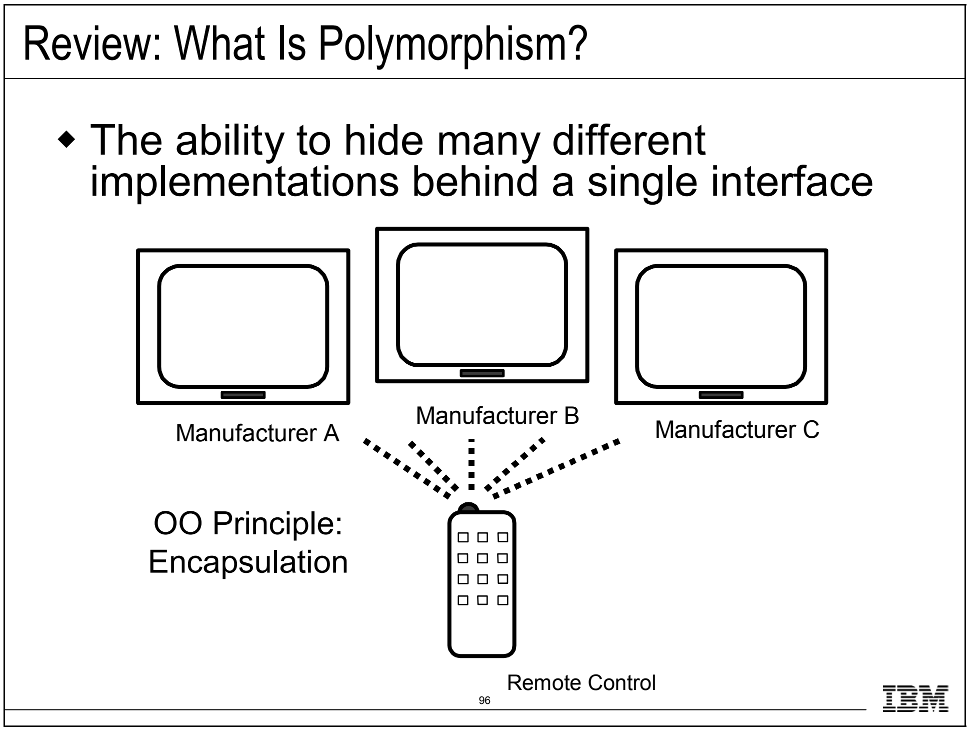
push() and pop() can access methods of List but instances of Stack cannot

An <<implementation>> stereotype can be used to model implementation inheritance. Implementation inheritance is where the implementation of a specific element is inherited or reused.

Implementation inheritance often leads to illogical inheritance hierarchies that are difficult to understand and to maintain. Therefore, it is recommended that you use inheritance *only* for implementation reuse, unless something else is recommended in using your programming language. Maintenance of this kind of reuse is usually quite tricky.

In the above example, any change in the class List can imply large changes of all classes inheriting from the class List. Be aware of this and inherit only stable classes. Inheritance will actually freeze the implementation of the class List because changes to it are too expensive.

Review: What Is Polymorphism?



Review: What Is Polymorphism?

- The ability to hide many different implementations behind a single interface

Manufacturer A

Manufacturer B

Manufacturer C

OO Principle: Encapsulation

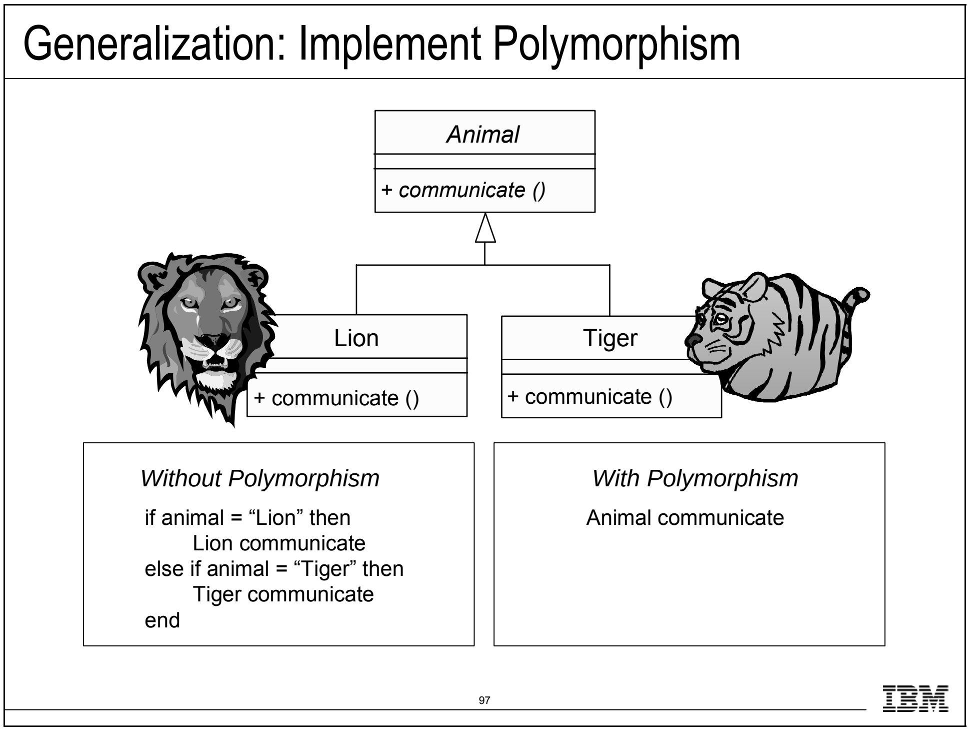
Remote Control

96

Polymorphism was first introduced in the Concepts of Object Orientation module. This slide is repeated here for review purposes.

The Greek term polymorphos means "having many forms". Every implementation of the interface must implement at least the interface. The implementation can in some cases implement more than the interface.

Generalization: Implement Polymorphism



Generalization: Implement Polymorphism

Animal

+ communicate ()

Lion

+ communicate ()

Tiger

+ communicate ()

Without Polymorphism

```
if animal = “Lion” then
        Lion communicate
else if animal = “Tiger” then
        Tiger communicate
end
```

With Polymorphism

```
Animal communicate
```

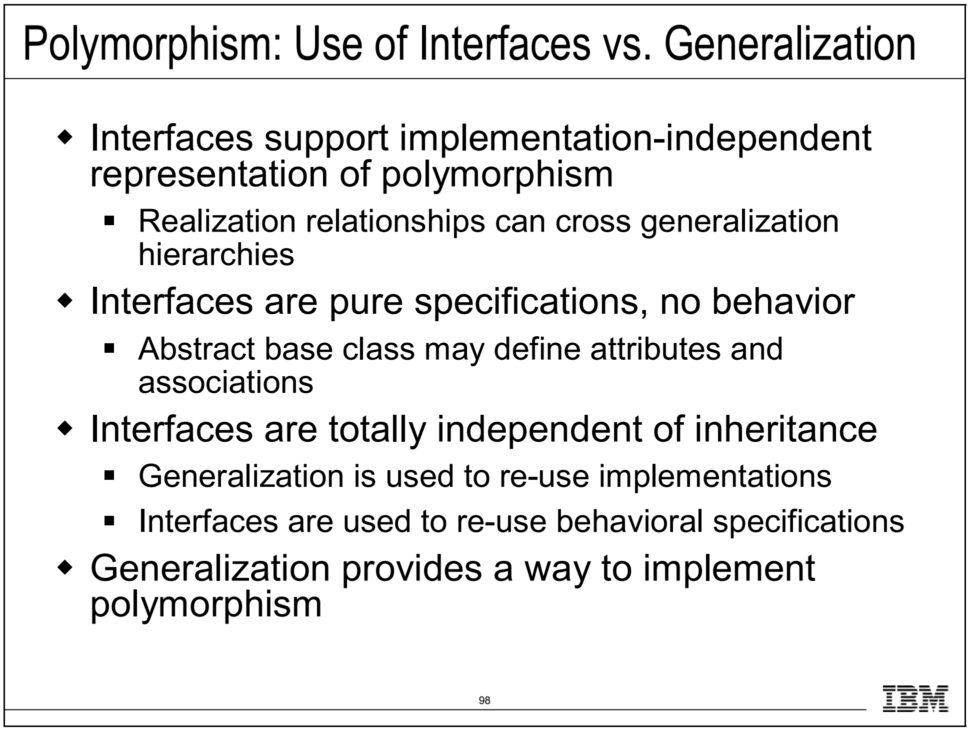
Inheritance provides a way to implement polymorphism in cases where polymorphism is implemented the same way for a set of classes. This means that abstract base classes that simply declare inherited operations, but which have no implementations of the operations, can be replaced with interfaces. Inheritance can now be restricted to inheritance of implementations only, if desired.

Polymorphism is not generalization; generalization is one way to implement polymorphism. Polymorphism through generalization is the ability to define alternate methods for ancestor class operations in the descendent classes. This can reduce the amount of code to be written, as well as help abstract the interface to descendent classes.

Polymorphism is an advantage of inheritance realized during implementation and at run time.

Programming environments that support polymorphism use dynamic binding, meaning that the actual code to execute is determined at run time rather than compile time.

Polymorphism: Use of Interfaces vs. Generalization



Polymorphism: Use of Interfaces vs. Generalization

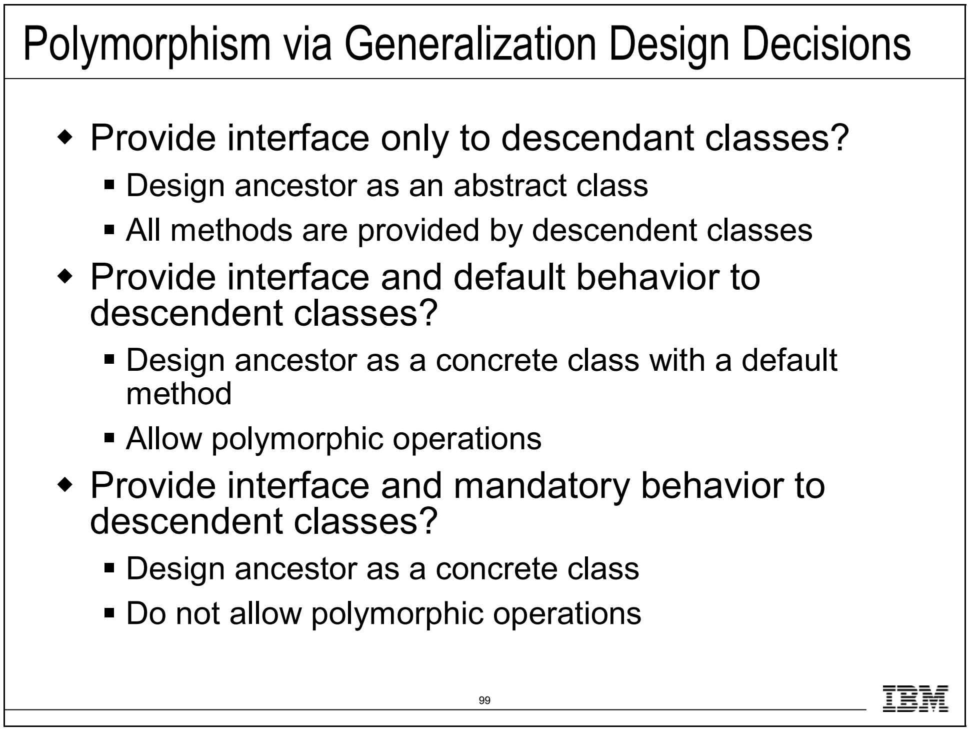
- Interfaces support implementation-independent representation of polymorphism
  - Realization relationships can cross generalization hierarchies
- Interfaces are pure specifications, no behavior
  - Abstract base class may define attributes and associations
- Interfaces are totally independent of inheritance
  - Generalization is used to re-use implementations
  - Interfaces are used to re-use behavioral specifications
- Generalization provides a way to implement polymorphism

98

There are several differences between the use of generalization (abstract base classes) and interfaces:

- Interfaces allow us to define polymorphism in a declarative way, unrelated to implementation. Two elements are polymorphic with respect to a set of behaviors if they realize the same interfaces. Interfaces are orthogonal to class inheritance lineage; two different classifiers may realize the same interface but be unrelated in their class hierarchies.
- Interfaces are purely specifications of behavior (a set of operation signatures). An abstract base class may define attributes and associations as well.
- Interfaces are totally independent of inheritance. Generalization is employed to re-use implementations; interfaces are employed to re-use and formalize behavioral specifications
- Generalization provides a way to implement polymorphism in cases where polymorphism is implemented the same way for a set of classes. The use of generalization to support polymorphism was discussed earlier in this module.

Polymorphism via Generalization Design Decisions



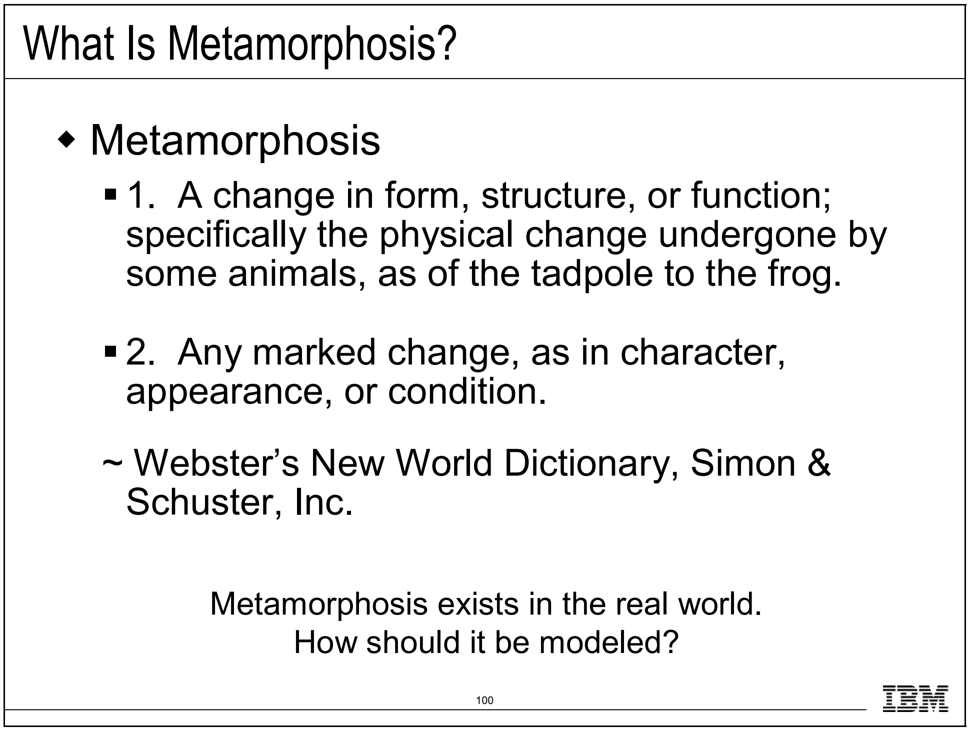
Polymorphism via Generalization Design Decisions

- Provide interface only to descendant classes?
  - Design ancestor as an abstract class
  - All methods are provided by descendent classes
- Provide interface and default behavior to descendent classes?
  - Design ancestor as a concrete class with a default method
  - Allow polymorphic operations
- Provide interface and mandatory behavior to descendent classes?
  - Design ancestor as a concrete class
  - Do not allow polymorphic operations

When designing the use of generalization to support polymorphism, there are three basic decisions that must be made:

- Do you want to provide a function's interface only to descendant classes? If so, design the ancestor as an abstract class, and only design methods for the descendent classes.
- Do you want to provide a function's interface and default behavior to descendent classes? If so, design the ancestor as a concrete class with a default method, and allow descendents to redefine the method.
- Do you want to provide a function's interface and mandatory behavior to descendent classes? If so, design the ancestor as a concrete class and disallow descendents from defining their own method for the operations.

What Is Metamorphosis?



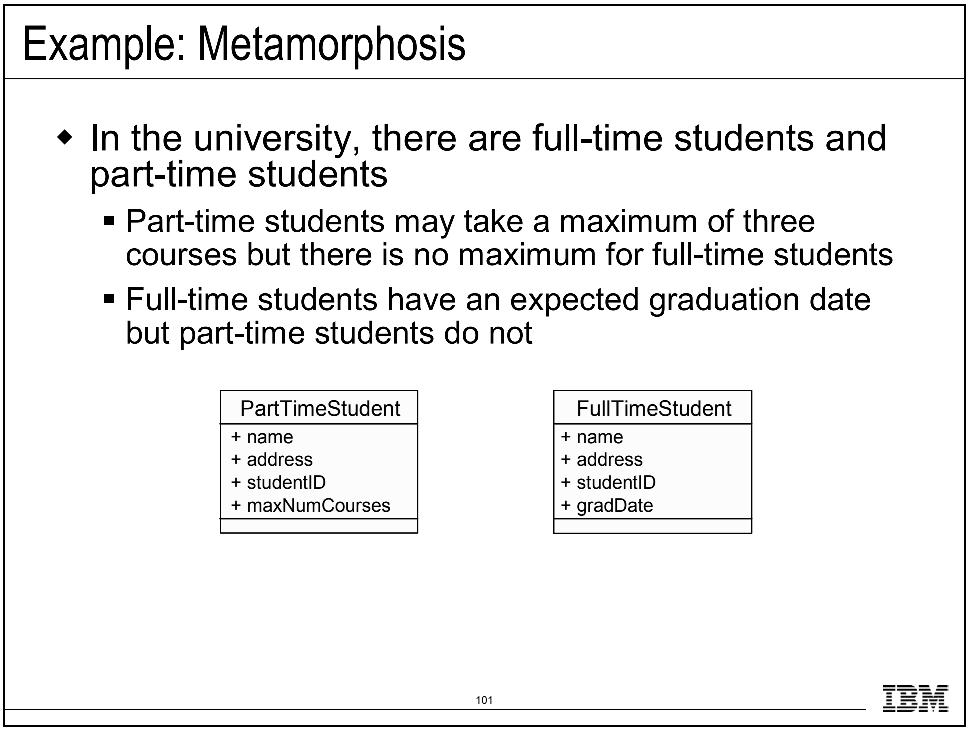
What Is Metamorphosis?

- Metamorphosis
  - 1. A change in form, structure, or function; specifically the physical change undergone by some animals, as of the tadpole to the frog.
  - 2. Any marked change, as in character, appearance, or condition.

  ~ Webster’s New World Dictionary, Simon & Schuster, Inc.

Metamorphosis exists in the real world.
How should it be modeled?

Example: Metamorphosis

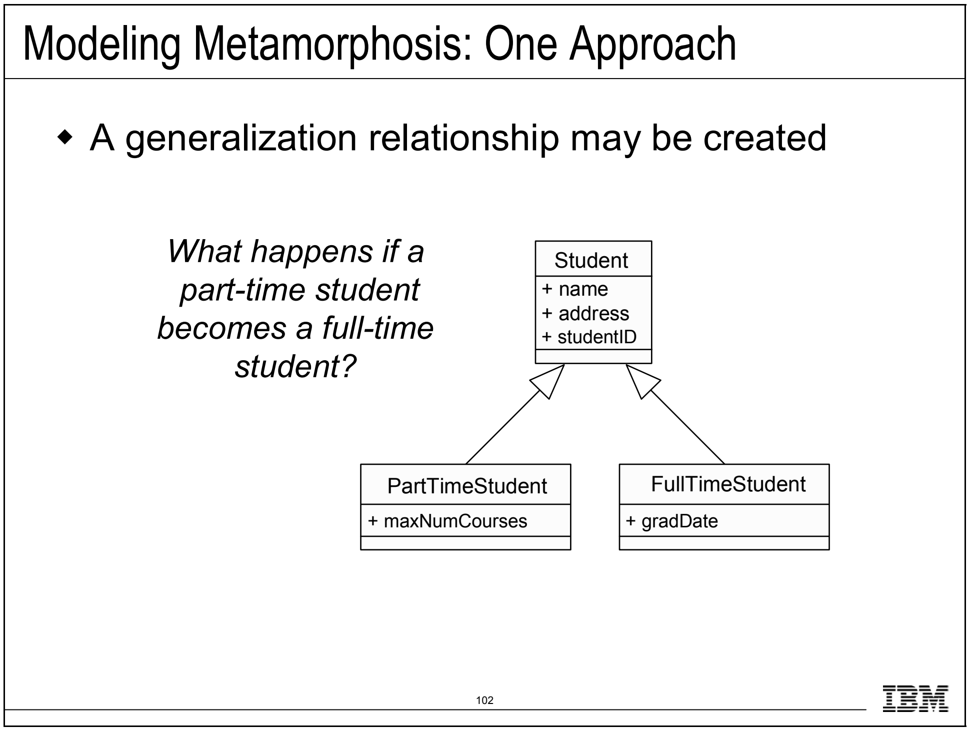


Example: Metamorphosis

- In the university, there are full-time students and part-time students
  - Part-time students may take a maximum of three courses but there is no maximum for full-time students
  - Full-time students have an expected graduation date but part-time students do not

PartTimeStudent
+ name
+ address
+ studentID
+ maxNumCourses
FullTimeStudent
+ name
+ address
+ studentID
+ gradDate

Modeling Metamorphosis: One Approach

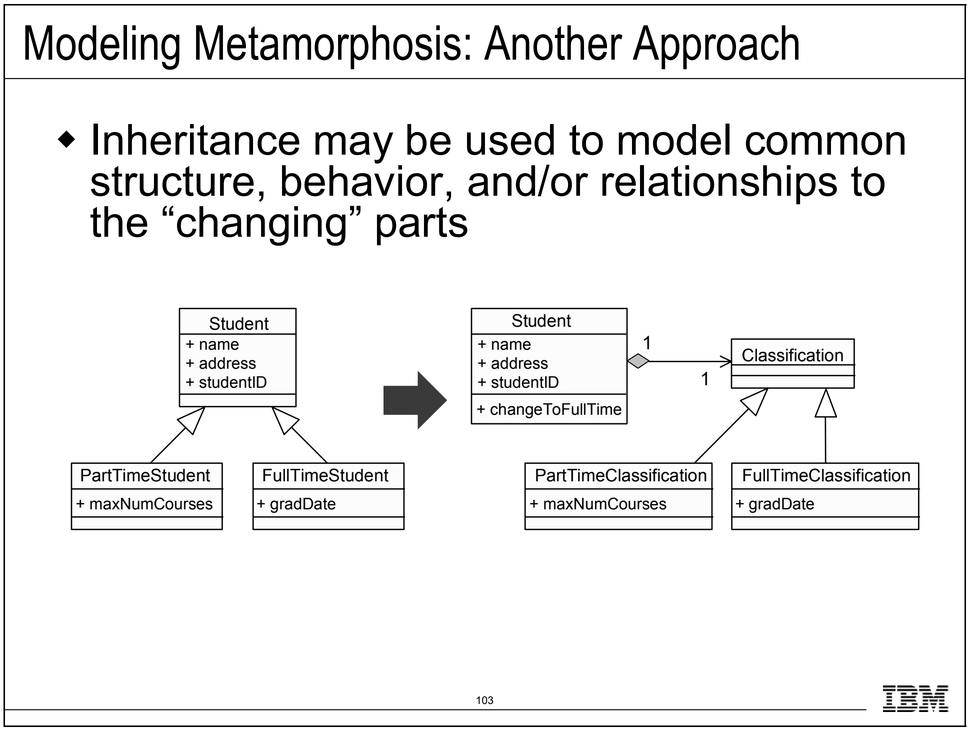


Modeling Metamorphosis: One Approach

- A generalization relationship may be created

What happens if a part-time student becomes a full-time student?

Modeling Metamorphosis: Another Approach



Modeling Metamorphosis: Another Approach

- Inheritance may be used to model common structure, behavior, and/or relationships to the “changing” parts

Student
+ name
+ address
+ studentID

PartTimeStudent
+ maxNumCourses

FullTimeStudent
+ gradDate

Student
+ name
+ address
+ studentID
+ changeToFullTime

1

1

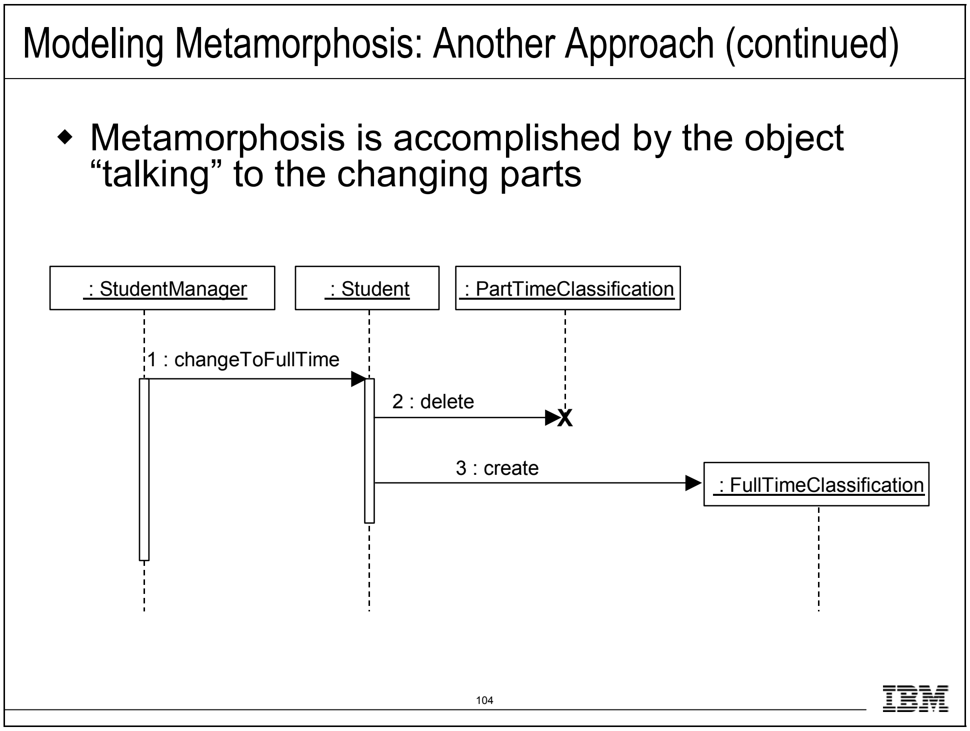
Classification

PartTimeClassification
+ maxNumCourses

FullTimeClassification
+ gradDate

This example shows the “before” and “after” with regard to implementing metamorphosis.

Modeling Metamorphosis: Another Approach (continued)

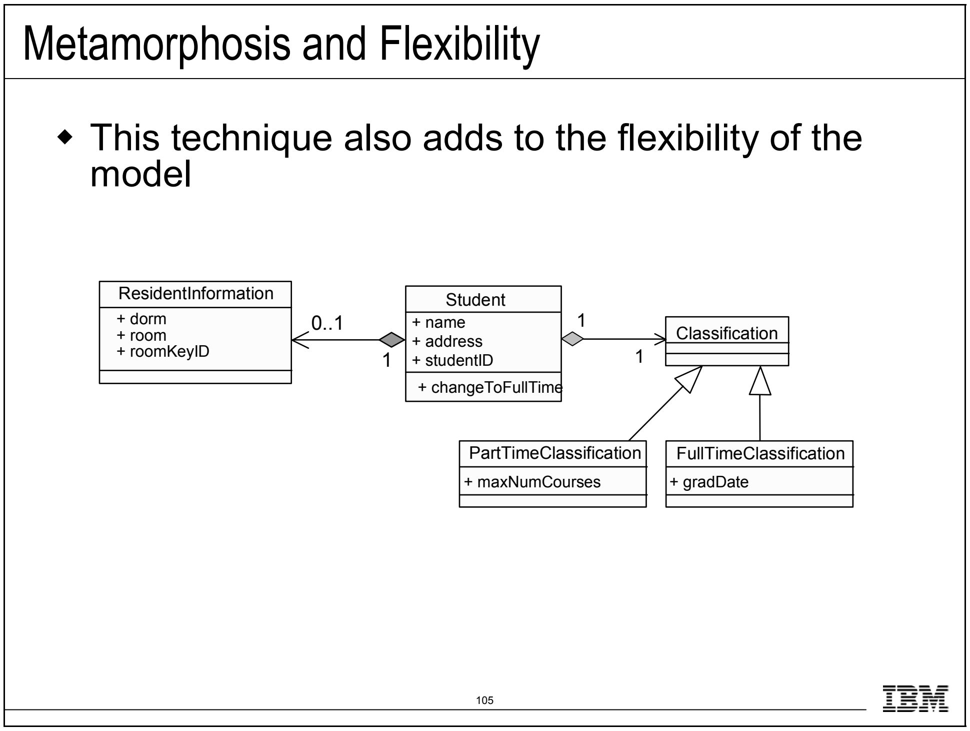


Modeling Metamorphosis: Another Approach (continued)

- Metamorphosis is accomplished by the object "talking" to the changing parts

The "x" referred here in the sequence diagram is a stop. It shows that the life of the instance has ended. This is a preferred way of showing the termination of the instance. The "create" introduces a new object in the sequence diagram.

Metamorphosis and Flexibility



Metamorphosis and Flexibility

- This technique also adds to the flexibility of the model

ResidentInformation
+ dorm
+ room
+ roomKeyID

0..1

1

Student
+ name
+ address
+ studentID
+ changeToFullTime

1

1

Classification

PartTimeClassification
+ maxNumCourses

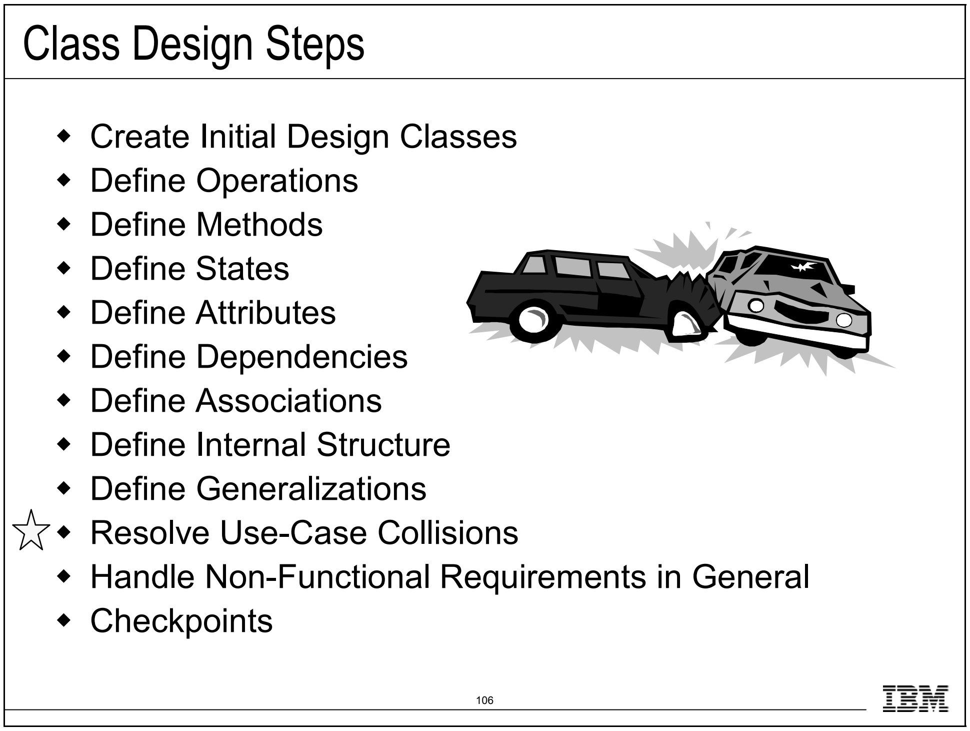
FullTimeClassification
+ gradDate

105

In this example, a student might also live on campus. In this case, there is a dorm identifier, a room number, and a room key number.

ResidentInformation is just a hypothetical class. It does not exist in the Course Registration model supplied with the course materials.

Resolve Use-Case Collisions



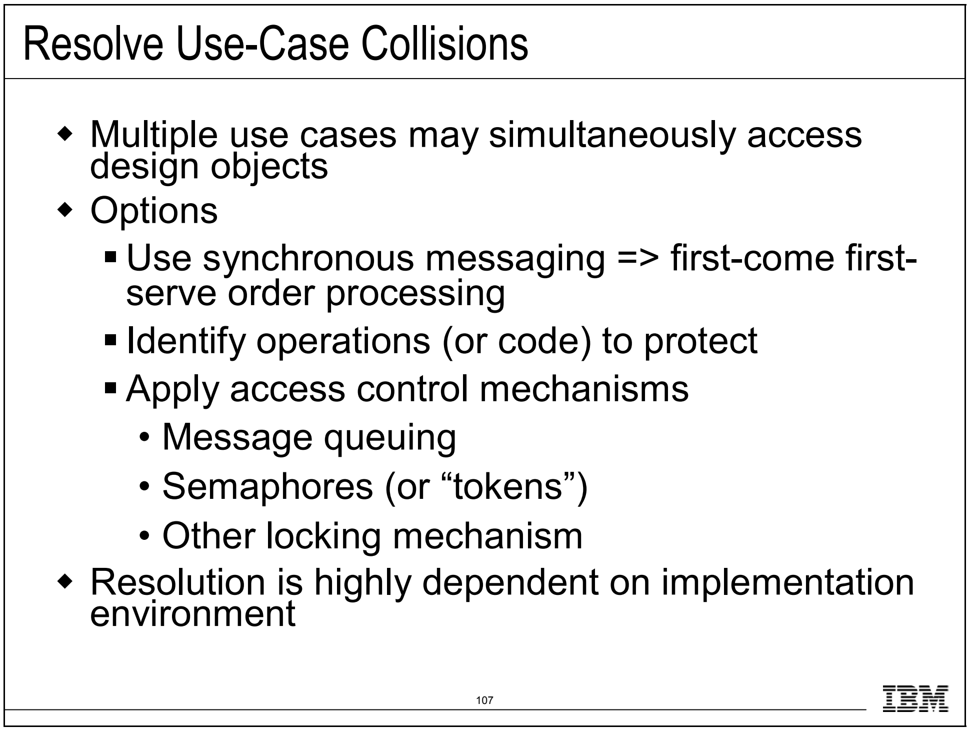
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- ☆ Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

106

The purpose of this step is to prevent concurrency conflicts caused when two or more use cases access instances of the design class simultaneously, in potentially inconsistent ways.

Resolve Use-Case Collisions



Resolve Use-Case Collisions

- Multiple use cases may simultaneously access design objects
- Options
  - Use synchronous messaging => first-come first-serve order processing
  - Identify operations (or code) to protect
  - Apply access control mechanisms
    - Message queuing
    - Semaphores (or “tokens”)
    - Other locking mechanism
- Resolution is highly dependent on implementation environment

One of the difficulties with proceeding use case-by-use case through the design process is that two or more use cases may simultaneously attempt to invoke operations on design objects in potentially conflicting ways. In these cases, concurrency conflicts must be identified and resolved explicitly.

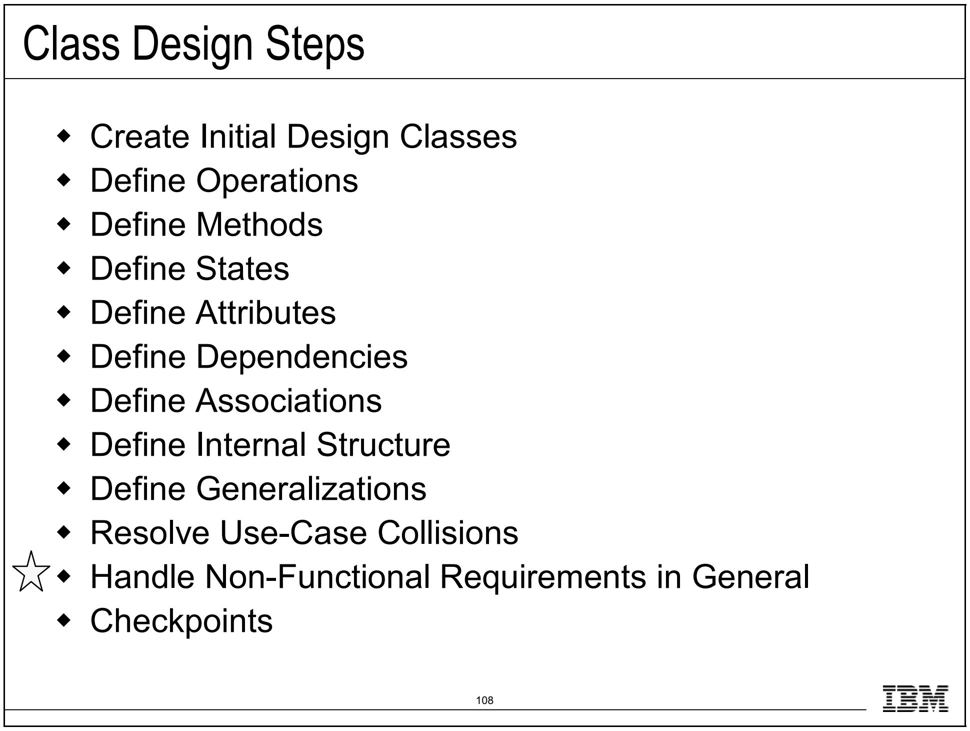
If synchronous messaging is used, execution of an operation will block subsequent calls to the objects until the operation completes. Synchronous messaging implies a first-come first-served ordering to message processing. This may resolve the concurrency conflict, as in cases where all messages have the same priority, or where every message runs within the same execution thread. In cases where an object may be accessed by different threads of execution, explicit mechanisms must be used to prevent or resolve the concurrency conflict.

For each object that may be accessed concurrently by different threads of execution, identify the code sections that must be protected from simultaneous access. If code is not available, identify which operations must be protected.

Next, select or design appropriate access control mechanisms to prevent conflicting simultaneous access. Examples of these mechanisms include message queuing to serialize access, use of semaphores (or “tokens”) to allow access to only one thread at a time, or other variants of locking mechanisms.

The choice of mechanism tends to be highly implementation —dependent, and typically varies with the programming language and operating environment. See the project-specific Design Guidelines for guidance on selecting concurrency mechanisms.

Handle Non-Functional Requirements in General



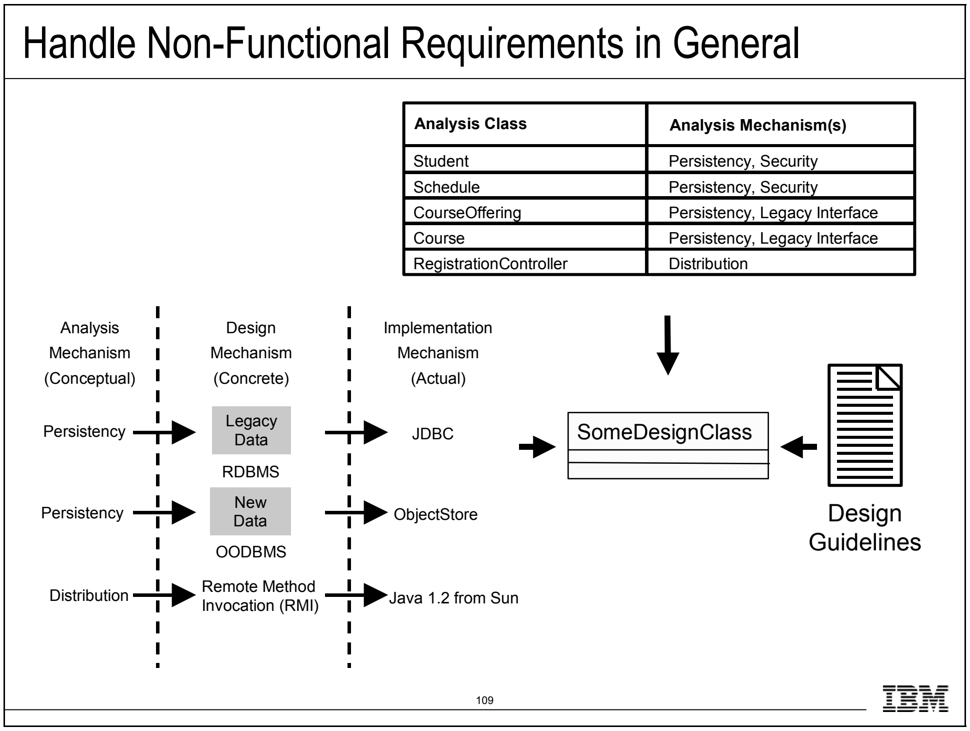
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- ☆ Handle Non-Functional Requirements in General
- Checkpoints

108

The purpose of this step is to make sure the design classes are refined to handle general non-functional requirements as stated in the Design Guidelines specific to the project (that is, to make sure that all mechanisms mapped to the class have been taken into account).

Handle Non-Functional Requirements in General



Handle Non-Functional Requirements in General

Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

Analysis Mechanism (Conceptual)

Design Mechanism (Concrete)

Implementation Mechanism (Actual)

Persistency → Legacy Data RDBMS → JDBC

Persistency → New Data OODBMS → ObjectStore

Distribution → Remote Method Invocation (RMI) → Java 1.2 from Sun

SomeDesignClass

Design Guidelines

109

The design classes should be refined to handle general non-functional requirements as stated in the Design Guidelines specific to the project. Important inputs to this step are the non-functional requirements on a class that may already be stated in its special requirements and responsibilities. Such requirements are often specified in terms of what architectural (analysis) mechanisms are needed to realize the class. In this step the class is refined to incorporate the design mechanisms corresponding to these analysis mechanisms.

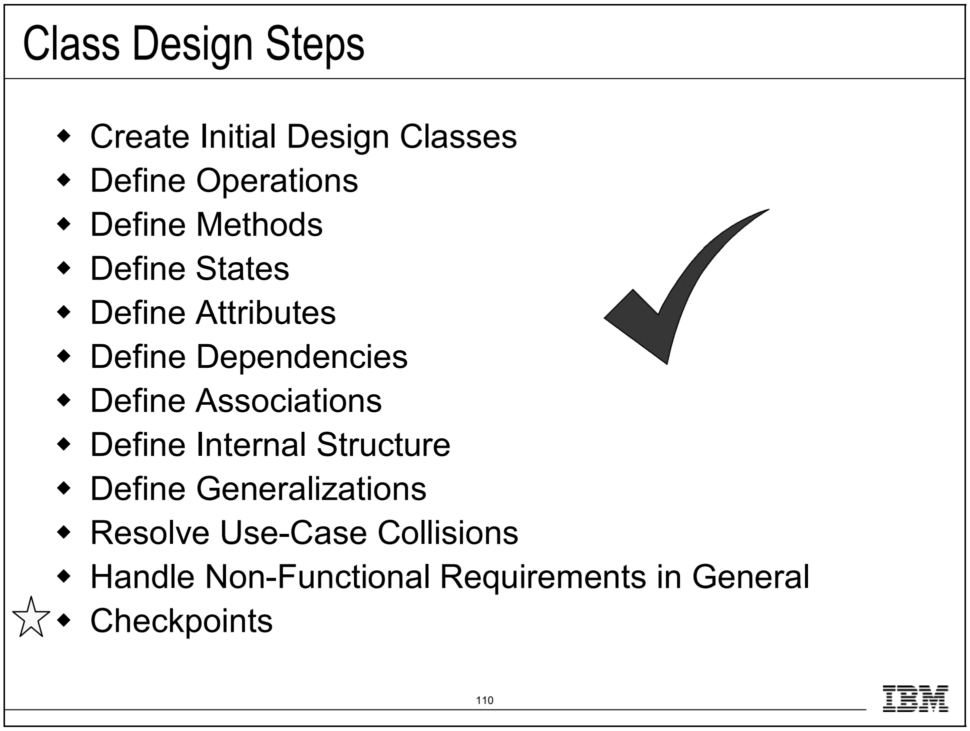
If you remember, the available design mechanisms were identified and characterized by the architect during Identify Design Mechanisms and were documented in the Design Guidelines.

In this step, the designer should, for each design mechanism needed, qualify as many characteristics as possible, giving ranges where appropriate.

Several general design guidelines and mechanisms that need to be taken into consideration when classes are designed, including:

- How to use existing products and components
- How to adapt to the programming language
- How to distribute objects
- How to achieve acceptable performance
- How to achieve certain security levels
- How to handle errors

Class Design Steps



Class Design Steps

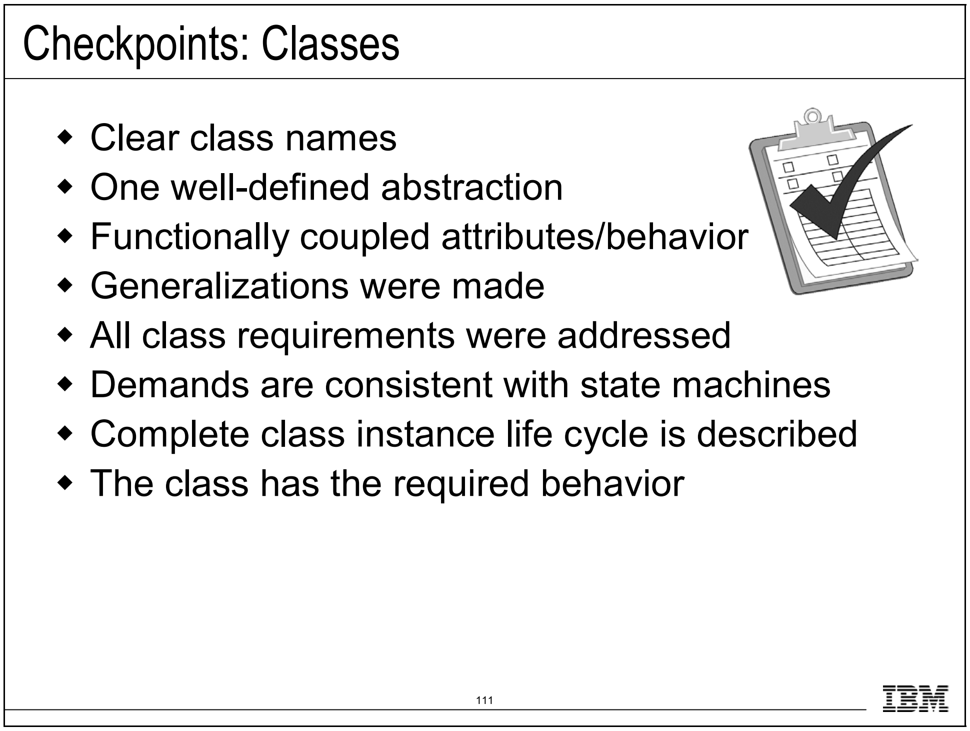
- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- ☆ Checkpoints

110

In this step, we verify that the Design Model fulfills the requirements of the system, is consistent with respect to the general design guidelines, and that it serves as a good basis for its implementation.

You should check the Design Model at this stage to verify that your work is headed in the right direction. There is no need to review the model in detail, but you should consider the checkpoints described on the next few slides.

Checkpoints: Classes



Checkpoints: Classes

- Clear class names
- One well-defined abstraction
- Functionally coupled attributes/behavior
- Generalizations were made
- All class requirements were addressed
- Demands are consistent with state machines
- Complete class instance life cycle is described
- The class has the required behavior

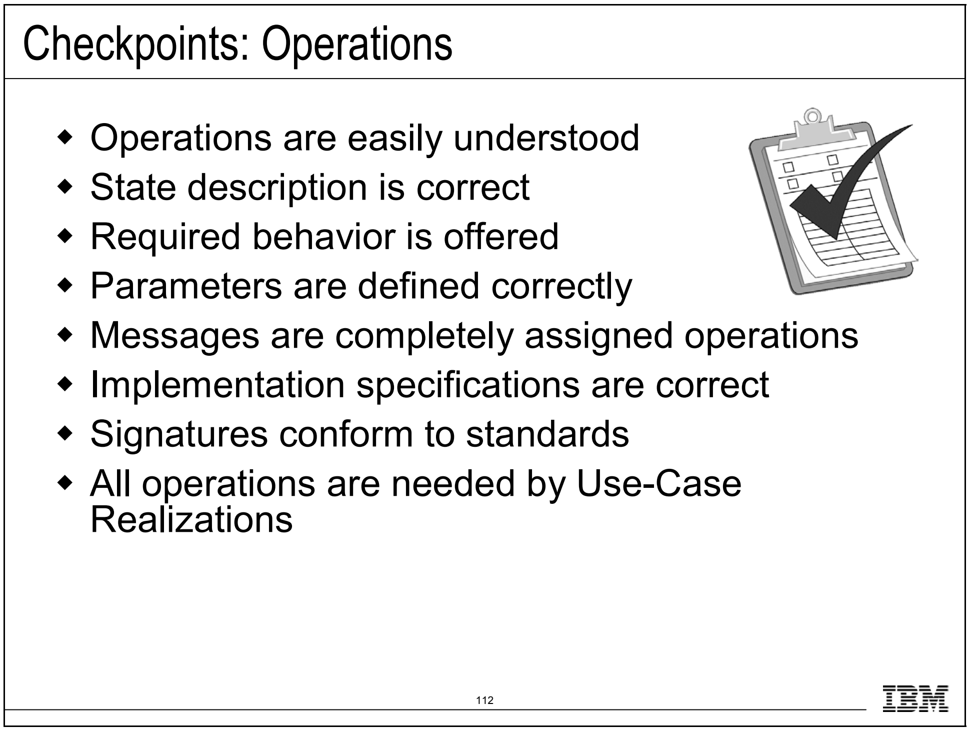
111

These are the questions you should be asking at this stage:
Does the name of each class clearly reflect the role it plays?

- Does the class represent a single well-defined abstraction?
  If the class does not represent a single well-defined abstraction, you should consider splitting it.
- Are all attributes and responsibilities functionally coupled?
  The class should only define attributes, responsibilities, or operations that are functionally coupled to the other attributes, responsibilities, or operations defined by that class.
- Are there any class attributes, operations or relationships that should be generalized, that is, moved to an ancestor?
- Are all specific requirements on the class addressed?
- Are the demands on the class consistent with any state machines that model the behavior of the class and its instances?

  The demands on the class (as reflected in the class description and by the objects in sequence diagrams) should be consistent with any state diagram that models the behavior of the class and its objects.
- Is the complete life cycle of an instance of the class described?
  The complete lifecycle of an instance of the class should be described. Each object must be created, used, and removed by one or more Use-Case Realizations.
- Does the class offer the required behavior?
  The classes should offer the behavior that the Use-Case Realizations and other classes require.

Checkpoints: Operations



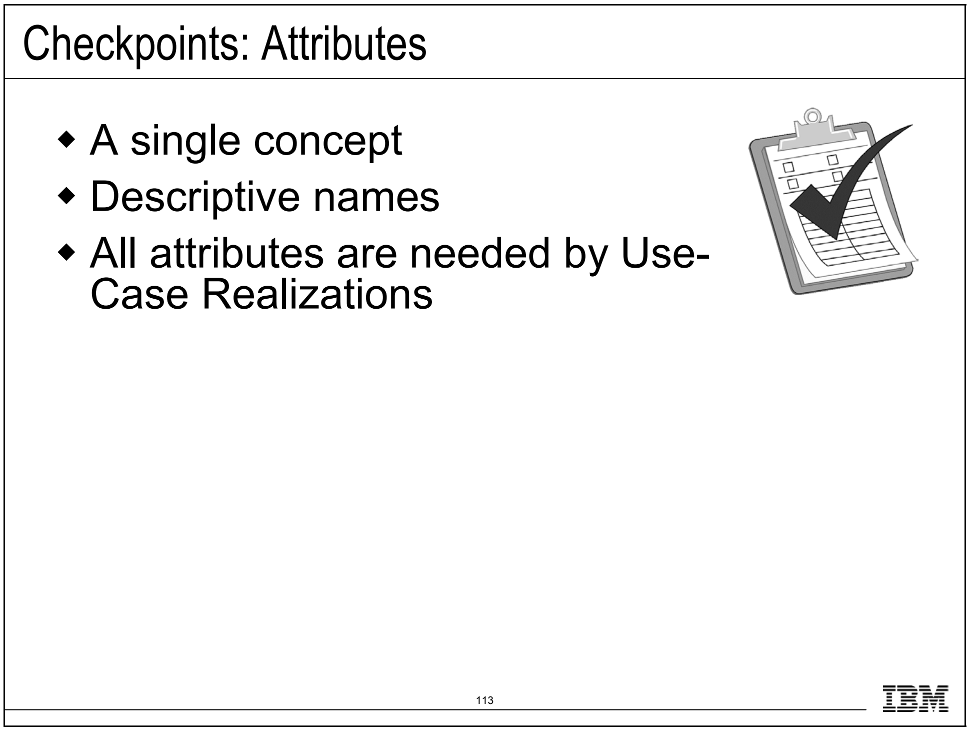
Checkpoints: Operations

- Operations are easily understood
- State description is correct
- Required behavior is offered
- Parameters are defined correctly
- Messages are completely assigned operations
- Implementation specifications are correct
- Signatures conform to standards
- All operations are needed by Use-Case Realizations

These are the questions you should ask about the operations of each class:

- Are the operations understandable?
  The names of the operations should be descriptive and the operations should be understandable to those who want to use them.
- Is the state description of the class and its objects' behavior correct?
- Does the class offer the behavior required of it?
- Have you defined the parameters correctly?
  Make sure that the parameters have been defined for each operation, and that there are not too many parameters for an operation.
- Have you assigned operations for the messages of each object completely?
- Are the implementation specifications (if any) for an operation correct?
- Do the operation signatures conform to the standards of the target programming language?
- Are all the operations needed by the Use-Case Realizations?
  Remove any operations that are redundant or not needed by the Use-Case Realizations.

Checkpoints: Attributes



Checkpoints: Attributes

- A single concept
- Descriptive names
- All attributes are needed by Use-Case Realizations

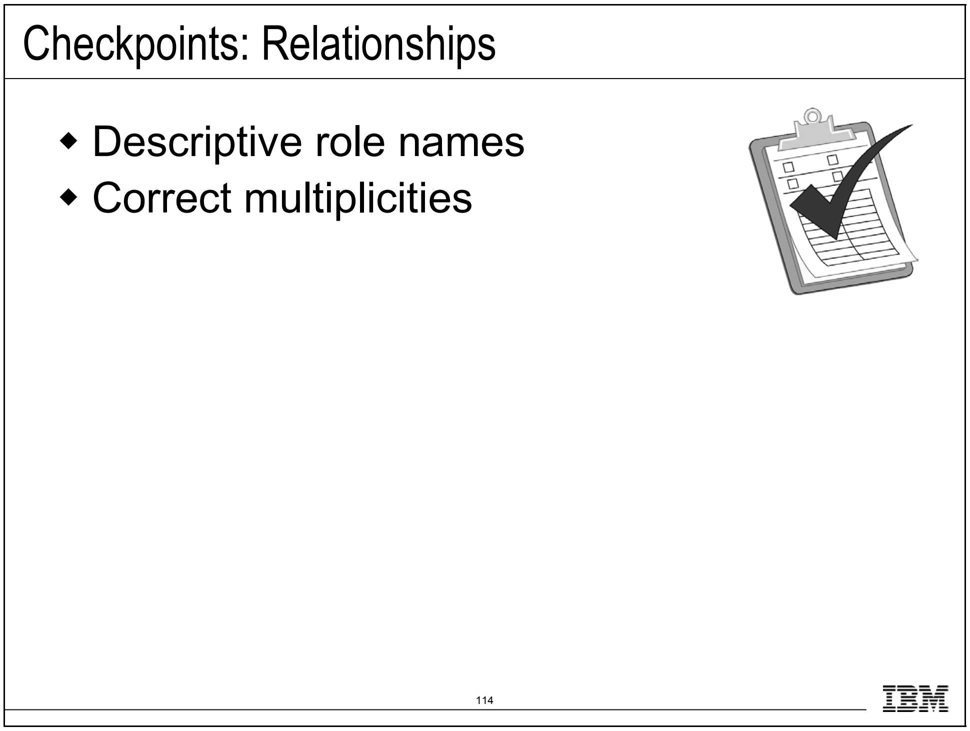
When thinking about attributes consider the following questions:

- Does each attribute represent a single conceptual thing?
- Are the names of the attributes descriptive?
- Are all the attributes needed by the Use-Case Realizations?

Remove any attributes that are redundant or not needed by the Use-Case Realizations.

Be sure to identify and define any applicable default attribute values.

Checkpoints: Relationships



Checkpoints: Relationships

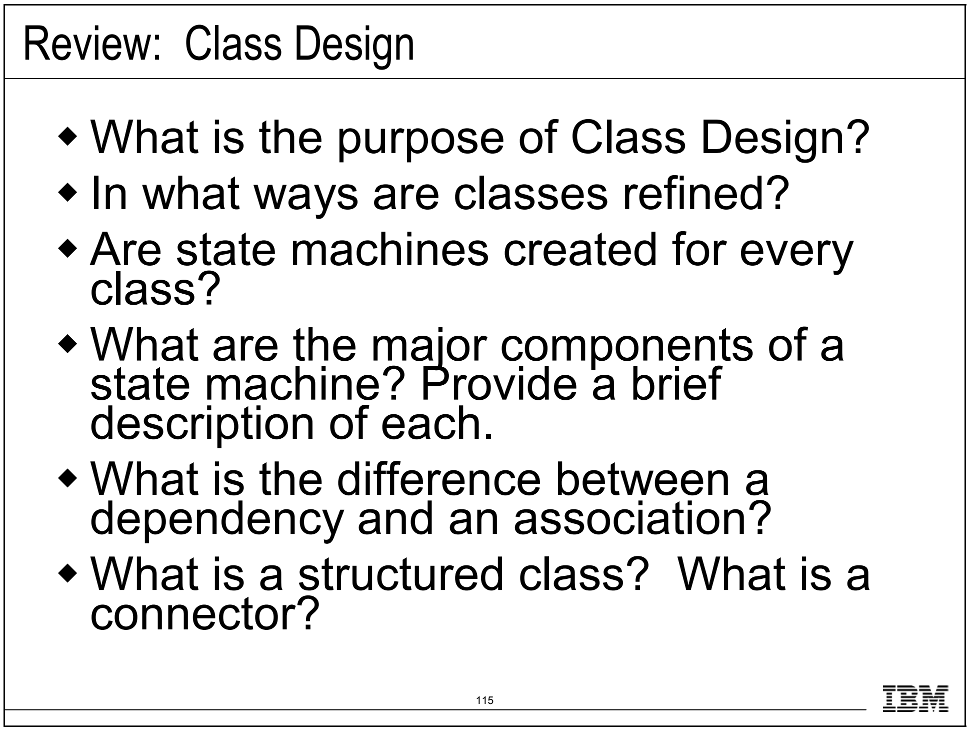
- Descriptive role names
- Correct multiplicities

114

- Are the role names descriptive?
- Are the multiplicities of the relationships correct?

The role names of the aggregations and associations should describe the relationship between the associated class and the relating class.

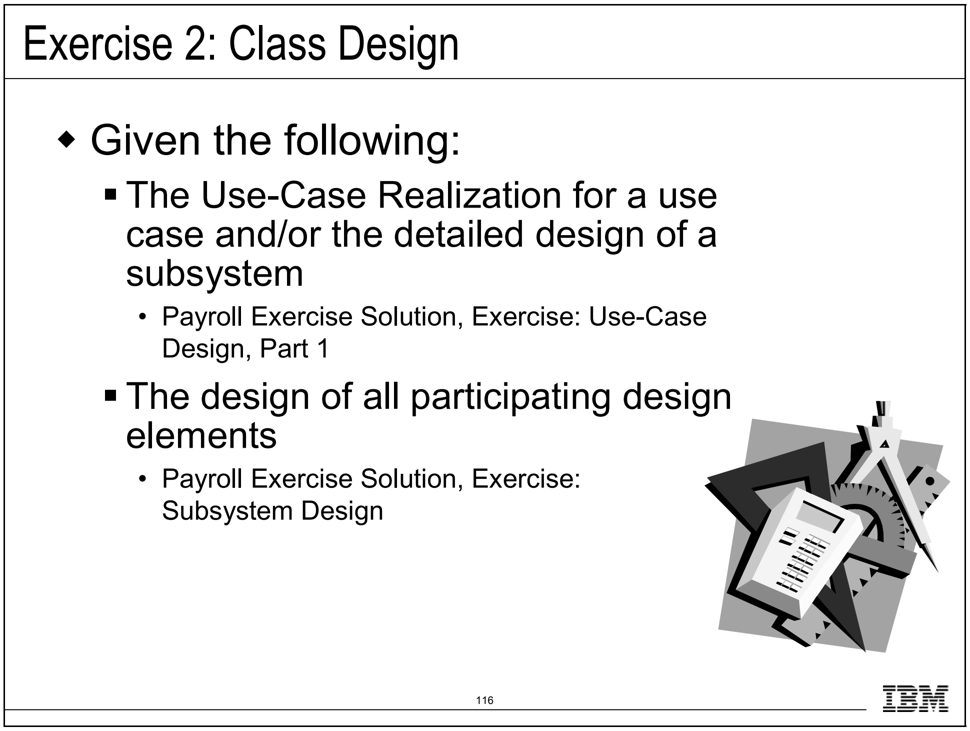
Review



Review: Class Design

- What is the purpose of Class Design?
- In what ways are classes refined?
- Are state machines created for every class?
- What are the major components of a state machine? Provide a brief description of each.
- What is the difference between a dependency and an association?
- What is a structured class? What is a connector?

Exercise 2: Class Design



Exercise 2: Class Design

- Given the following:
  - The Use-Case Realization for a use case and/or the detailed design of a subsystem
    - Payroll Exercise Solution, Exercise: Use-Case Design, Part 1
  - The design of all participating design elements
    - Payroll Exercise Solution, Exercise: Subsystem Design

The goal of this exercise is to refine the relationships that a design class has with other design elements (that is, to design the class relationships).

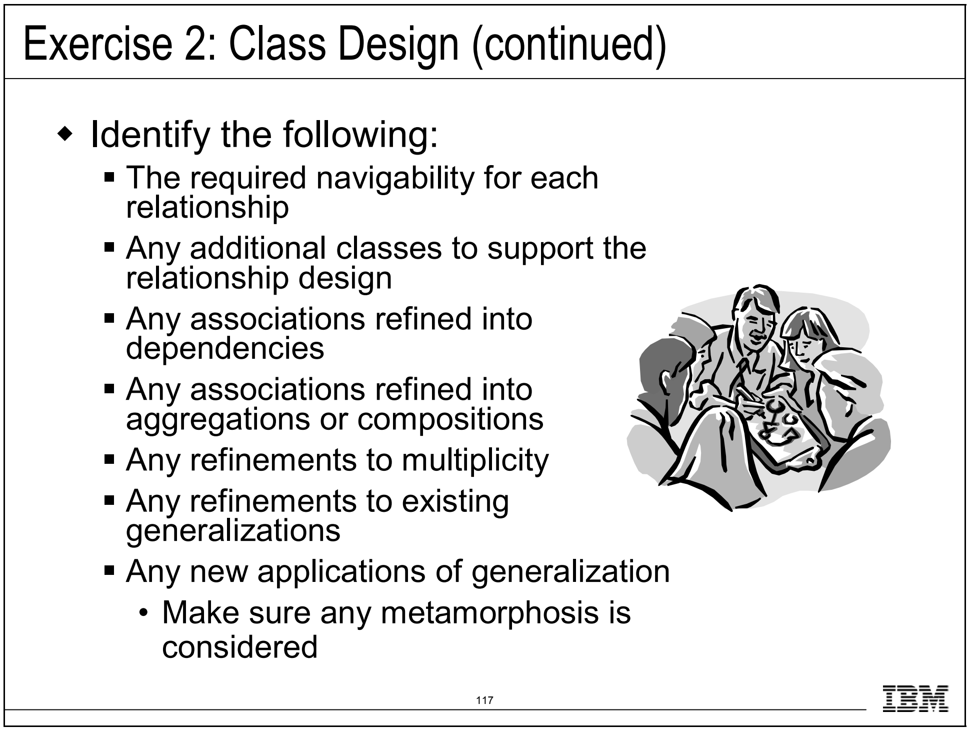
For this exercise, we will focus our efforts on the Use-Case Realizations that were developed in the Use-Case Design module and/or the subsystem interface realizations developed in the Subsystem Design module.

At this point, the design of the model elements (classes and subsystems) includes a first attempt at attributes and operations, with complete signatures, as well as any defined relationships.

References to the givens:

- **Use-Case Realizations:** Payroll Exercise Solution, Exercise: Use-Case Design, Part 1.
- **Subsystem interface realizations:** Payroll Exercise Solution, Exercise: Subsystem Design section.

Exercise 2: Class Design (continued)



Exercise 2: Class Design (continued)

- Identify the following:
  - The required navigability for each relationship
  - Any additional classes to support the relationship design
  - Any associations refined into dependencies
  - Any associations refined into aggregations or compositions
  - Any refinements to multiplicity
  - Any refinements to existing generalizations
  - Any new applications of generalization
    - Make sure any metamorphosis is considered

During relationship design, you must look at each relationship for its:

- **Type:** Need to decide if the relationship is appropriate. In most cases, you will look at each association to decide if it should remain an association or be refined into a dependency relationship. When making this decision, it is important to define the relationship visibility (that is, field, local, global, or parameter).
- **Navigability:** Is the defined navigability still valid? For two-way/bi-directional relationships, are both directions really needed? (Remember, implementing bi-directional relationships is more expensive.)
- **Multiplicity:** Is the defined multiplicity still valid?

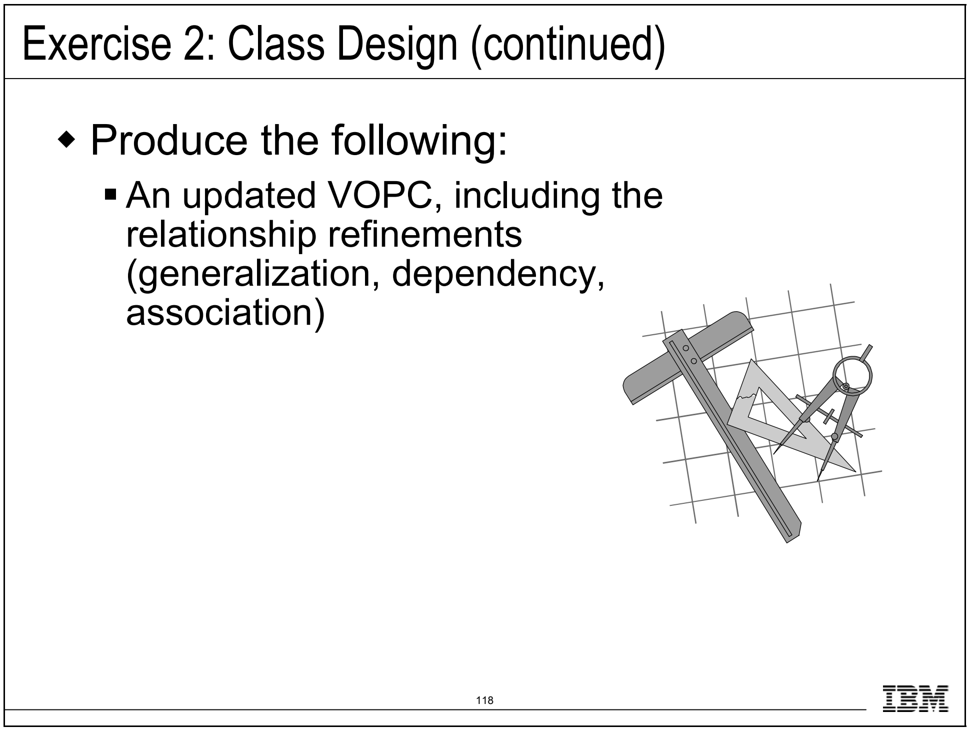
When designing the relationships, you will need to consult both the static and dynamic uses of the relationship (that is, the VOPC and the Use-Case Realization interaction diagrams).

Be sure to note the rationale for your choices. For example, for every association that is refined into a dependency, note the visibility behind it (for example, local, global, parameter).

Examine each of the class diagrams produced up to this point and determine if and where generalization could be used.

Hint: Keep in mind metamorphosis and see if the technique applies.

Exercise 2: Class Design (continued)



Exercise 2: Class Design (continued)

- Produce the following:
  - An updated VOPC, including the relationship refinements (generalization, dependency, association)

118

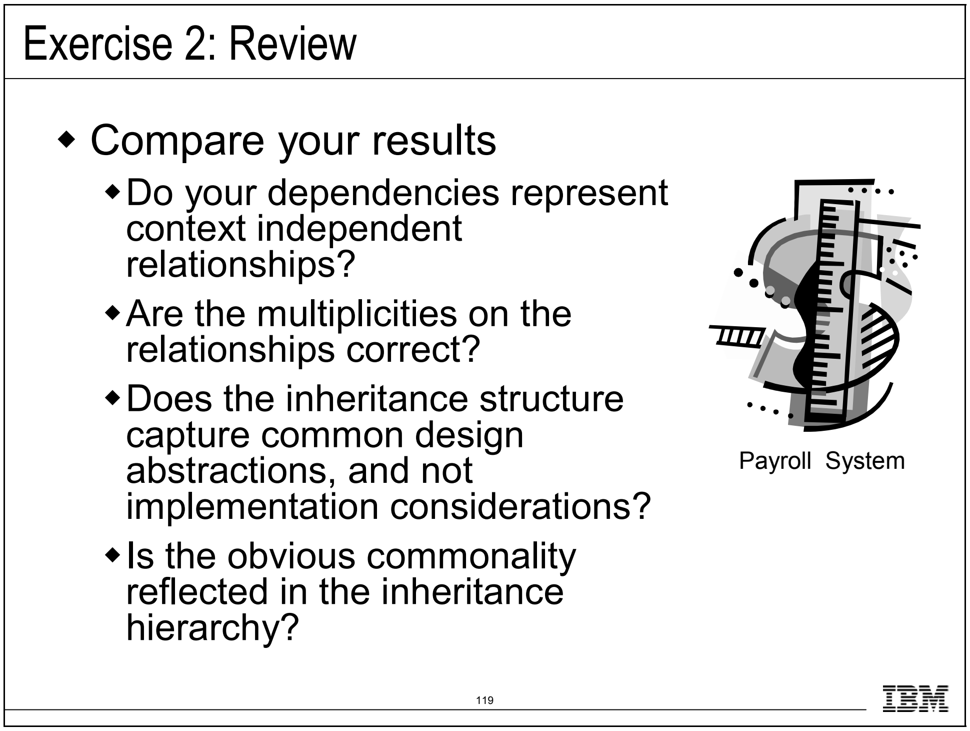
The Class diagram you produce can include any number of classes, as long as it effectively demonstrates the use of generalization.

If the generalization changes affect Use-Case Realization diagrams, these diagrams should be updated, as well. Not all generalization changes will require Use-Case Realization refinements (however, generalization to support metamorphosis usually do).

References to sample diagrams within the course that are similar to what should be produced are:

- **Define Dependencies**: 13-52 and 13-53.
- **Generalization**: slides 13-93; 13-94, 13-102.

Exercise 2: Review



Exercise 2: Review

- Compare your results
  - Do your dependencies represent context independent relationships?
  - Are the multiplicities on the relationships correct?
  - Does the inheritance structure capture common design abstractions, and not implementation considerations?
  - Is the obvious commonality reflected in the inheritance hierarchy?

Payroll System

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

- Do you dependencies represent context independent relationships?
- Are the multiplicities on the relationships correct?
- Does the inheritance structure capture common design abstractions, and not implementation considerations?
- Is the obvious commonality reflected in the inheritance hierarchy?