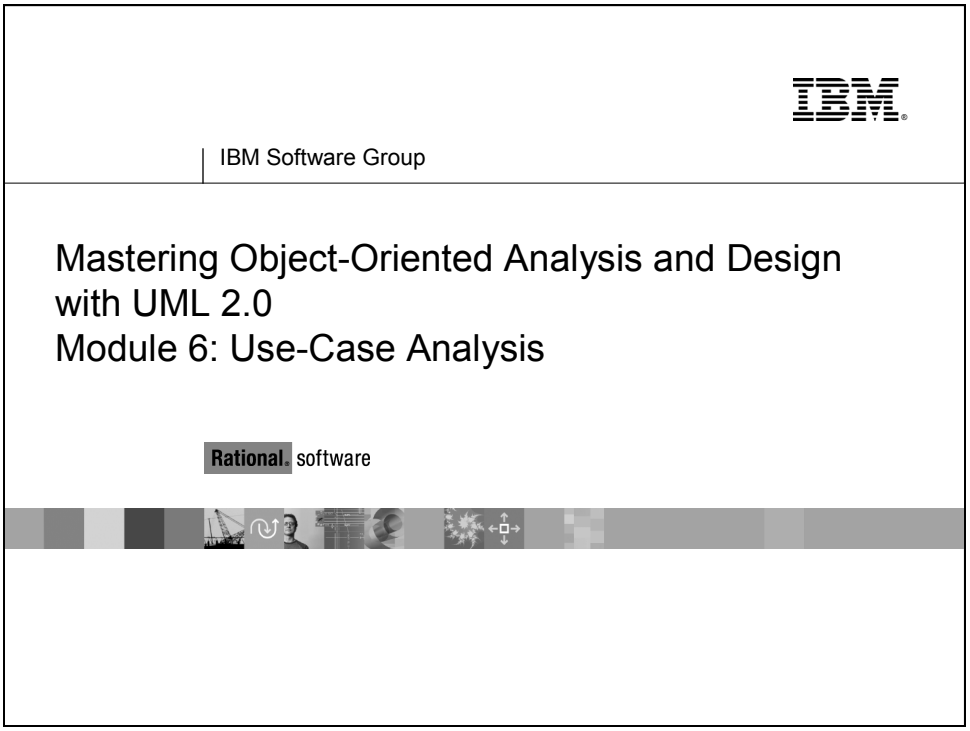
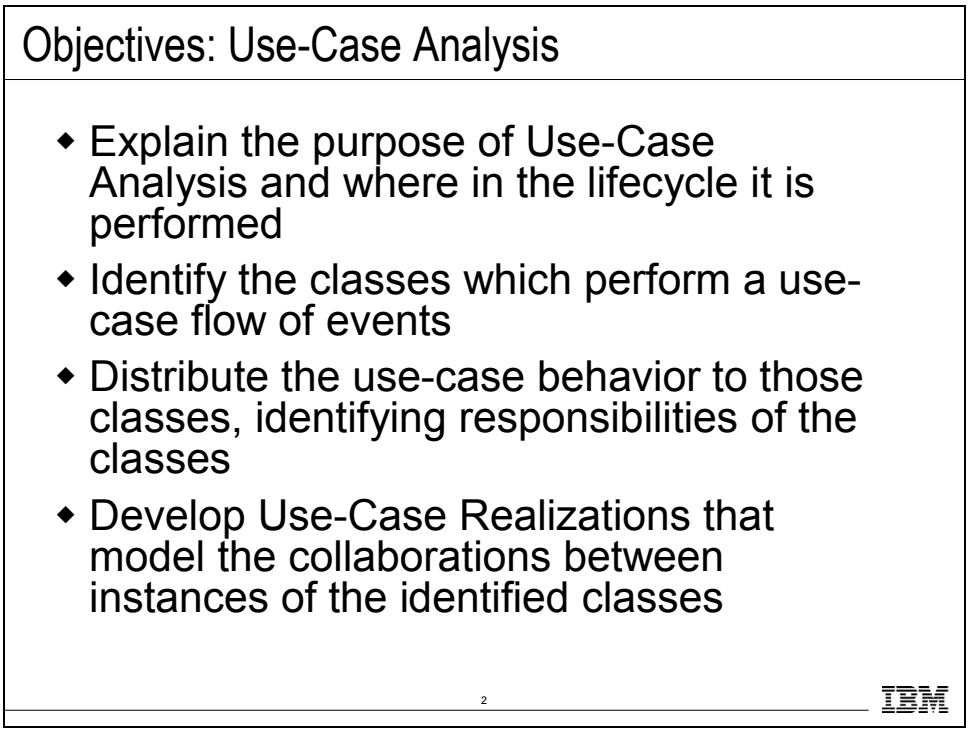
▸ ▸ ▸ Module 6
Use-Case Analysis



Topics

Objectives: Use-Case Analysis

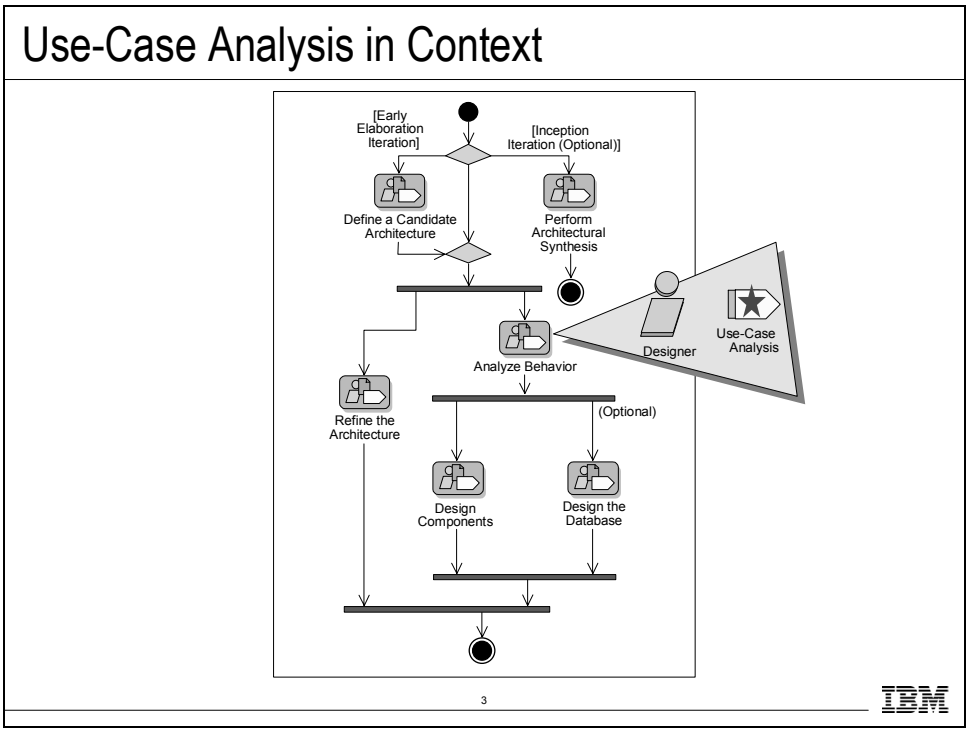


Use-Case Analysis is where we identify the initial classes of our system.

As the analysis classes are defined and the responsibilities are allocated to them, we will also note the usage of architectural mechanisms, more specifically, the usage of any analysis mechanisms defined in Architectural Analysis.

The analysis classes and the initial Use-Case Realizations are the key model elements being developed in this activity. These will be refined in the remaining Analysis and Design activities.

Use-Case Analysis in Context



As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. **Use-Case Analysis** is an activity in the Analyze Behavior workflow detail.

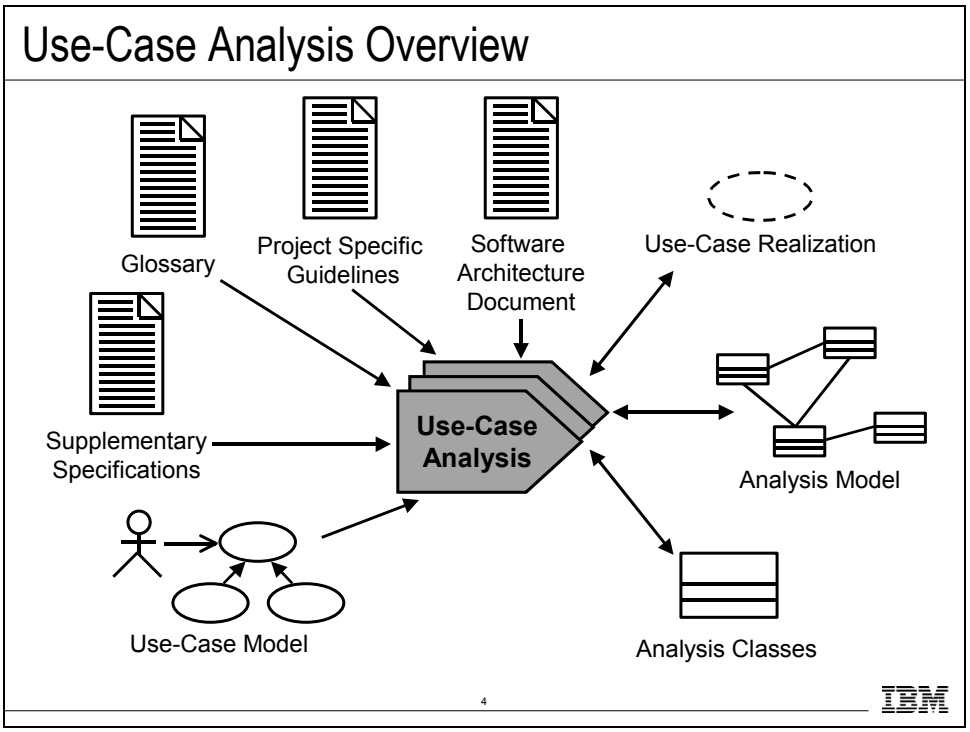
At this point, we have made an initial attempt at defining our architecture — we have defined the upper layers of our architecture, the key abstractions, and some key analysis mechanisms. This initial architecture, along with the software requirements defined in the Requirements discipline, guides and serves as input to the **Use-Case Analysis** activity.

An instance of **Use-Case Analysis** is performed for each use case to be developed during an iteration. The focus during **Use-Case Analysis** is on a particular use case.

In **Use-Case Analysis**, we identify the analysis classes and define their responsibilities. As the analysis classes and their responsibilities are defined, we will also note the usage of any architectural (more specifically, analysis) patterns defined in Architectural Analysis. The architectural layers and their dependencies may affect the allocation of responsibility to the defined analysis classes.

The allocation of responsibility is modeled in Use-Case Realizations that describe how analysis classes collaborate to perform use cases. The Use-Case Realizations will be refined in the Use-Case Design Model.

Use-Case Analysis Overview



Use-Case Analysis is performed by the designer, once per iteration per Use-Case Realization. What event flows, and therefore what Use-Case Realizations you are going to work on during the current iteration are defined prior to the start of **Use-Case Analysis** in Architectural Analysis.

Purpose

- To identify the classes that perform a use case's flow of events
- To distribute the use case behavior to those classes, using Use-Case Realizations
- To identify the responsibilities, attributes and associations of the classes
- To note the usage of architectural mechanisms

Input Artifacts

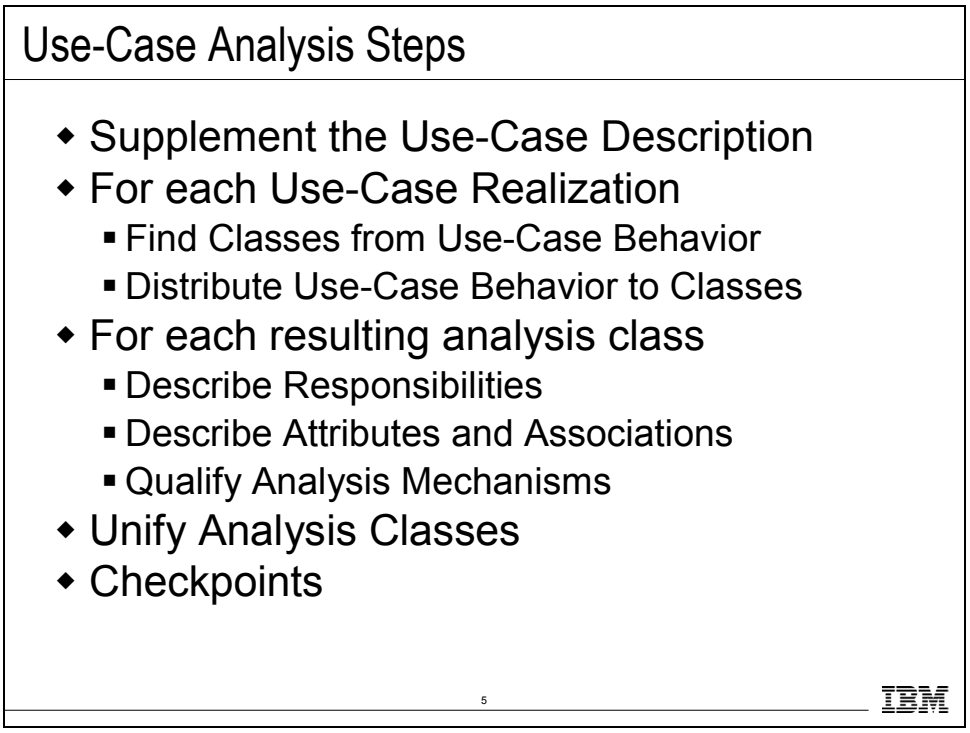
- Glossary
- Supplementary Specifications
- Use-Case
- Use-Case Model
- Use-Case Realization
- Software Architecture Document
- Analysis Class
- Analysis Model
- Project Specific Guidelines

Resulting Artifacts

- Analysis Classes
- Analysis Model
- Use-Case Realizations

Note: We will not be developing a separate Analysis Model in this course.

Use-Case Analysis Steps



The above are the major steps of the **Use-Case Analysis** activity.

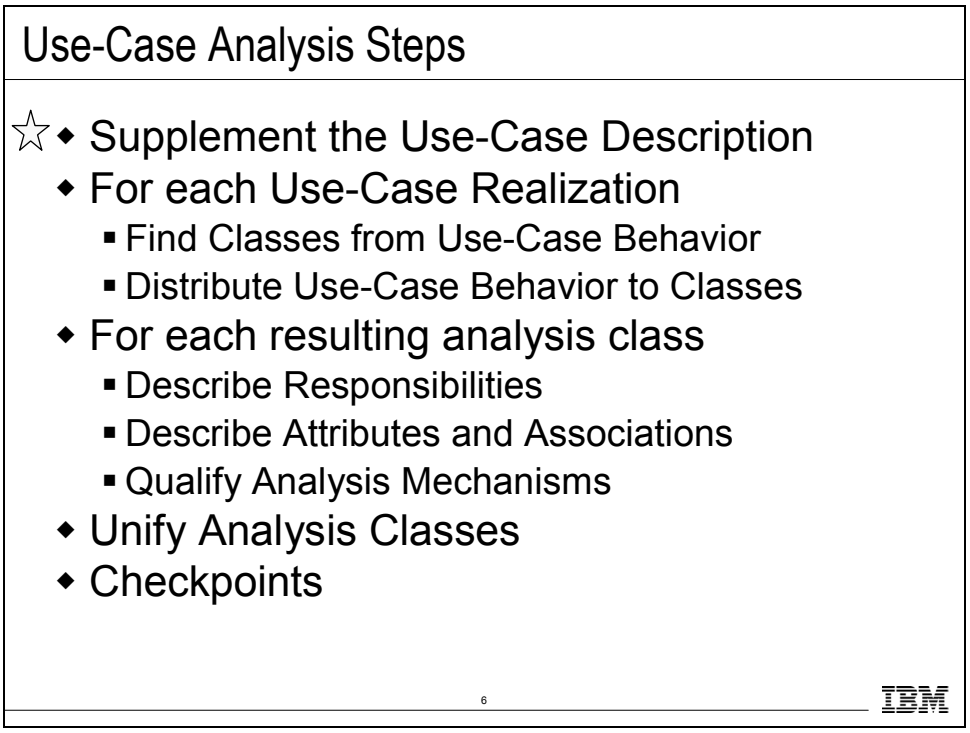
First we must review the use-case descriptions developed in the Requirements discipline. Chances are, they will need some enhancements to include enough detail to begin developing a model.

Next, we study the use-case flow of events, identify analysis classes, and allocate use-case responsibilities to the analysis classes. Based on these allocations, and the analysis class collaborations, we can begin to model the relationships between the identified analysis classes.

Once the use case has been analyzed, we need to take a good look at the identified classes, making sure they are thoroughly documented and identify which analysis and mechanisms they implement.

Last, but not least, we need to make sure that our developed Analysis Model is consistent.

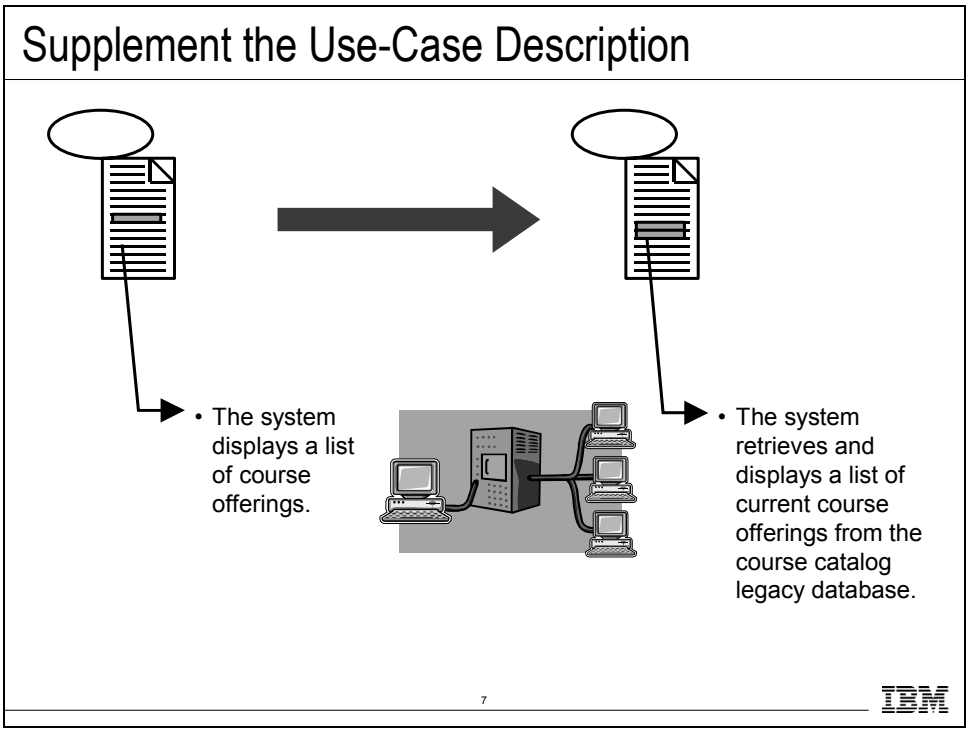
Use-Case Analysis Steps



The purpose of the Supplement the Descriptions of the Use Case is to capture additional information needed in order to understand the required internal behavior of the system that may be missing from the Use-Case Description written for the customer of the system. This information will be used as input to the rest of the steps in **Use-Case Analysis** and is used to assist in the allocation of responsibility.

Note: In some cases, we may find that some requirements were incorrect or not well-understood. In those cases, the original use-case flow of events should be updated (for example, iterate back to the Requirements discipline).

Supplement the Use-Case Description

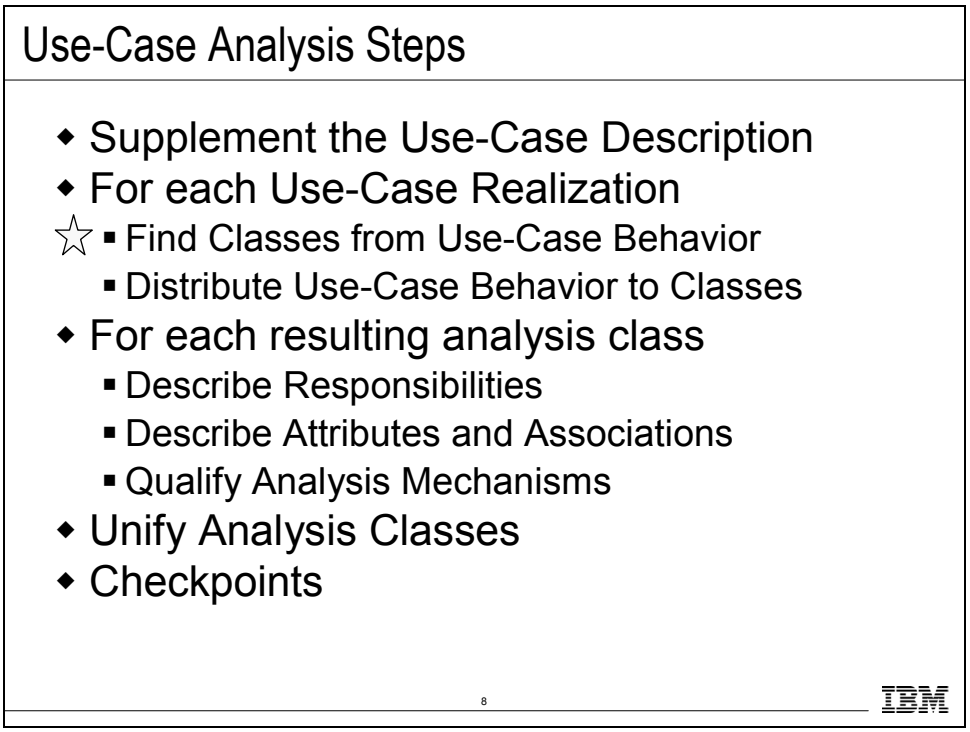


The description of each use case is not always sufficient for finding analysis classes and their objects. The customer generally finds information about what happens inside the system uninteresting, so the use-case descriptions may leave such information out. In these cases, the use-case description reads like a “black-box” description, in which internal details on what the system does in response to an actor’s actions is either missing or very summarily described. To find the objects that perform the use case, you need to have the “white box” description of what the system does from an internal perspective.

For example, in the case of the Course Registration System, the student might prefer to say ”the system displays a list of course offerings.” While this might be sufficient for the student, it gives us no real idea of what really happens inside the system. In order to form an internal picture of how the system really works, at a sufficient level of detail to identify objects, we might need additional information.

Taking the Register for Courses use case as an example, the expanded description would read as: “The system retrieves a list of current course offerings from the course catalog legacy database.” This level of detail gives us a clear idea of what information is required and who is responsible for providing that information.

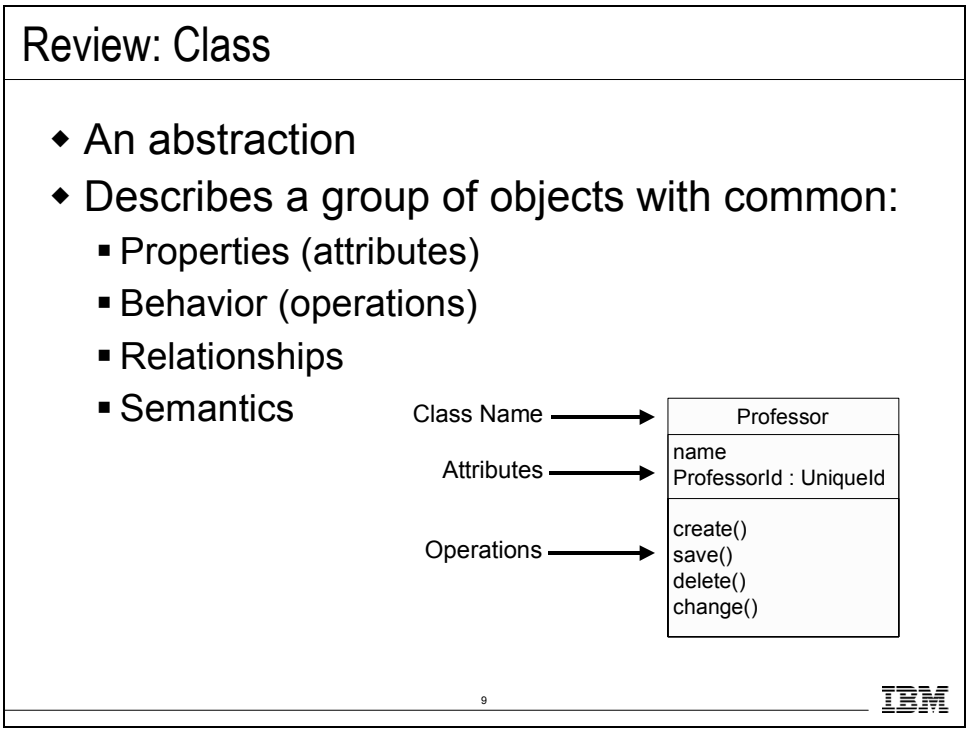
Use-Case Analysis Steps



Now that we have a more detailed understanding of the Requirements, as documented in the use case, we can identify the candidate analysis classes for our system.

The purpose of the Find Classes from Use-Case Behavior step is to identify a candidate set of model elements (analysis classes) that will be capable of performing the behavior described in the use case.

Review: Class



As discussed in the Concepts of Object Orientation module, a class is a description of a group of objects with common properties (attributes), common behavior (operations), common relationships, and common semantics.

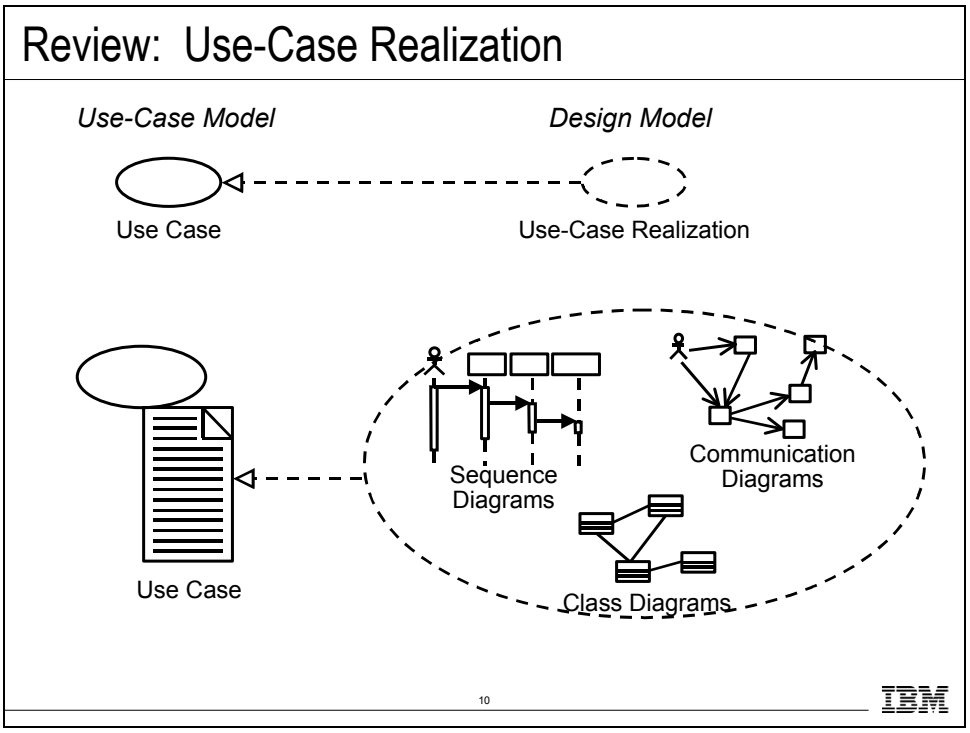
A class is an abstraction in that it:

- Emphasizes relevant characteristics.
- Suppresses other characteristics.

A class is comprised of three sections:

- The first section contains the class name.
- The second section shows the structure (attributes).
- The third section shows the behavior (operations).

Review: Use-Case Realization



As discussed in the Analysis and Design Overview module, a Use-Case Realization describes how a particular use case is realized within the Design Model in terms of collaborating objects. A Use-Case Realization in the Design Model can be traced to a use case in the Use-Case Model. A realization relationship is drawn from the Use-Case Realization to the use case it realizes.

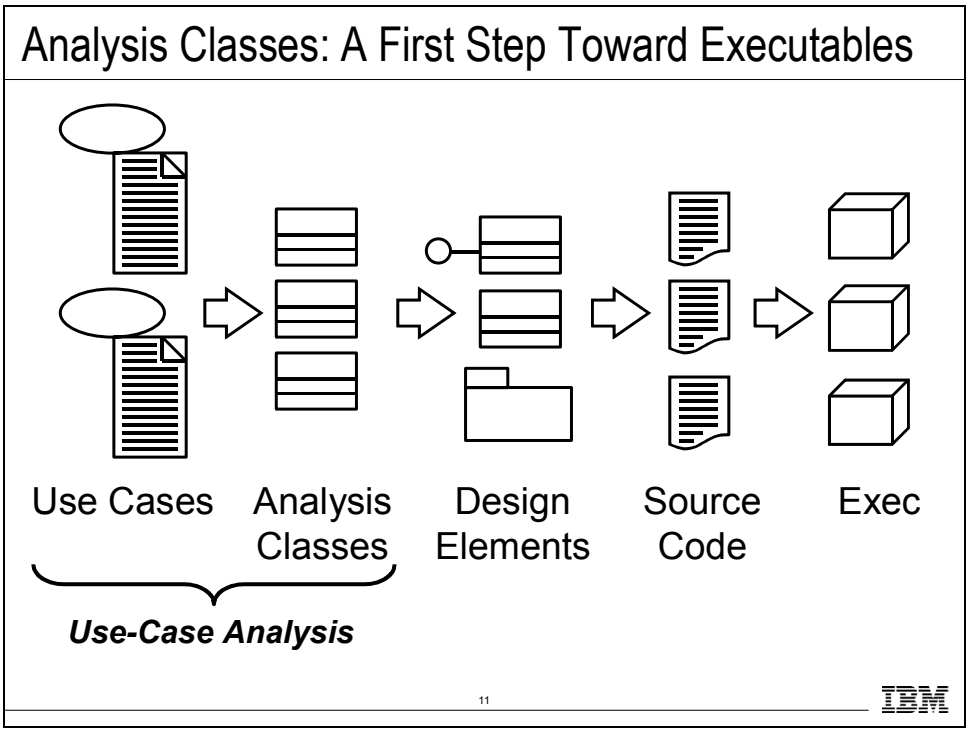
A Use-Case Realization is one possible realization of a use case. A Use-Case Realization can be represented using a set of diagrams (the number and type may vary by project).

- Interaction diagrams (Sequence and/or Communication diagrams) can be used to describe how the use case is realized in terms of collaborating objects. These diagrams model the detailed collaborations of the Use-Case Realization.
- Class diagrams can be used to describe the classes that participate in the realization of the use case, as well as their supporting relationships. These diagrams model the context of the Use-Case Realization.

During analysis activities (**Use-Case Analysis**), the Use-Case Realization diagrams are outlined. In subsequent design activities (Use-Case Design), these diagrams are refined and updated according to more formal class interface definitions.

A designer is responsible for the integrity of the Use-Case Realization. He or she must coordinate with the designers responsible for the classes and relationships employed in the Use-Case Realization. The Use-Case Realization can be used by class designers to understand the class's role in the use case and how the class interacts with other classes. This information can be used to determine or refine the class responsibilities and interfaces.

Analysis Classes: A First Step Toward Executables

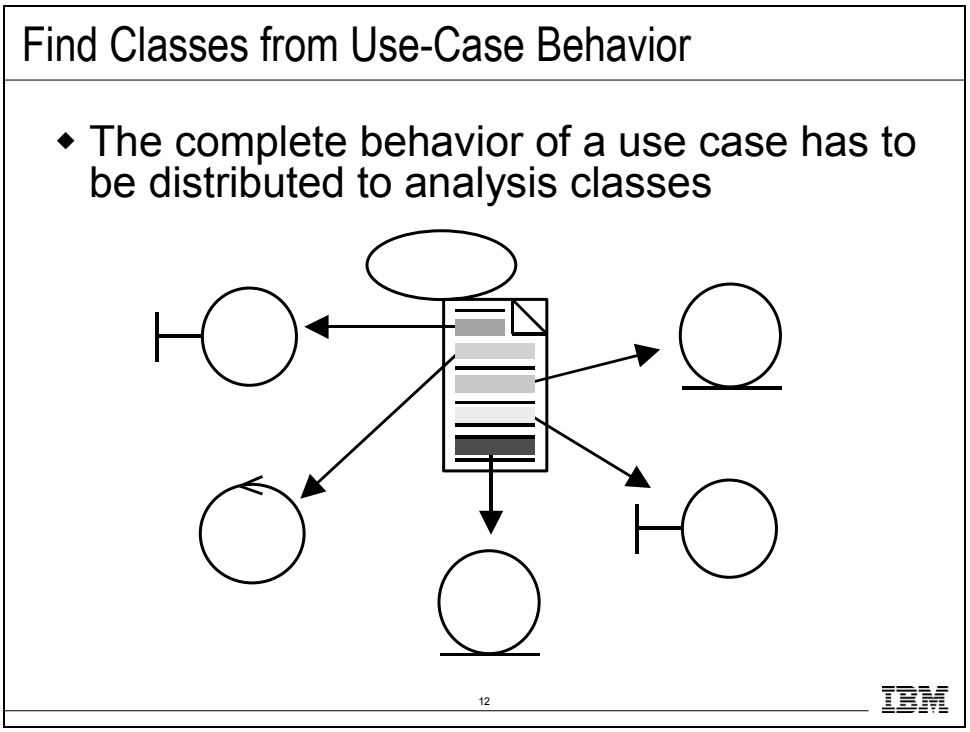


Finding a candidate set of roles is the first step in the transformation of the system from a mere statement of required behavior to a description of how the system will work.

The analysis classes, taken together, represent an early conceptual model of the system. This conceptual model evolves quickly and remains fluid for some time as different representations and their implications are explored. Formal documentation can impede this process, so be careful how much energy you expend on maintaining this “mode”’ in a formal sense; you can waste a lot of time polishing a model that is largely expendable. Analysis classes rarely survive into the design unchanged. Many of them represent whole collaborations of objects, often encapsulated by subsystems.

Analysis classes are “proto-classes,” which are essentially "clumps of behavior." These analysis classes are early conjectures of the composition of the system; they rarely survive intact into Implementation. Many of the analysis classes morph into something else later (subsystems, components, split classes, or combined classes). They provide you with a way of capturing the required behaviors in a form that we can use to explore the behavior and composition of the system. Analysis classes allow us to "play" with the distribution of responsibilities, re-allocating as necessary.

Find Classes from Use-Case Behavior



The technique for finding analysis classes described in this module uses three different perspectives of the system to drive the identification of candidate classes. These three perspectives are:

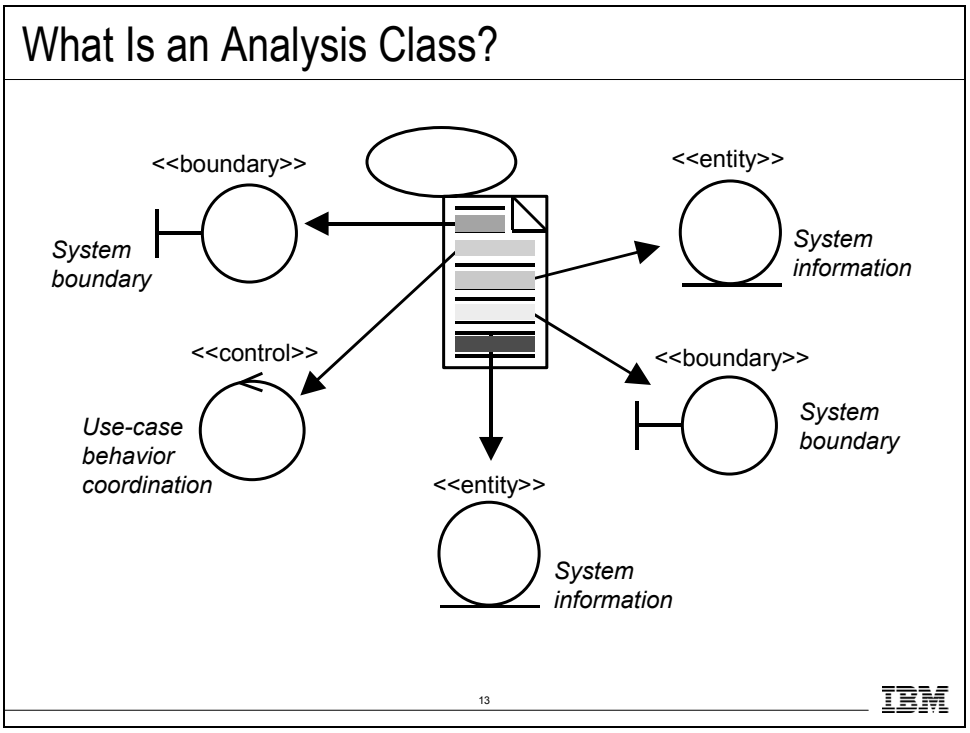
- The boundary between the system and its actors
- The information the system uses
- The control logic of the system

The use of stereotypes to represent these perspectives (for example, boundary, control, and entity) results in a more robust model because they isolate those things most likely to change in a system: the interface/environment, the control flow, and the key system entities. These stereotypes are conveniences used during Analysis that disappear in Design.

Identification of classes means just that: They should be identified, named, and described briefly in a few sentences.

The different stereotypes are discussed in more detail throughout this module.

What Is an Analysis Class?



Analysis classes represent an early conceptual model for “things in the system that have responsibilities and behavior.” Analysis classes are used to capture a “first-draft” rough-cut of the Object Model of the system.

Analysis classes handle primarily functional requirements. They model objects from the problem domain. Analysis classes can be used to represent "the objects we want the system to support" without making a decision about how much of them to support with hardware and how much with software.

Three aspects of the system are likely to change:

- The boundary between the system and its actors
- The information the system uses
- The control logic of the system

In an effort to isolate the parts of the system that will change, the following types of analysis classes are identified with a “canned” set of responsibilities:

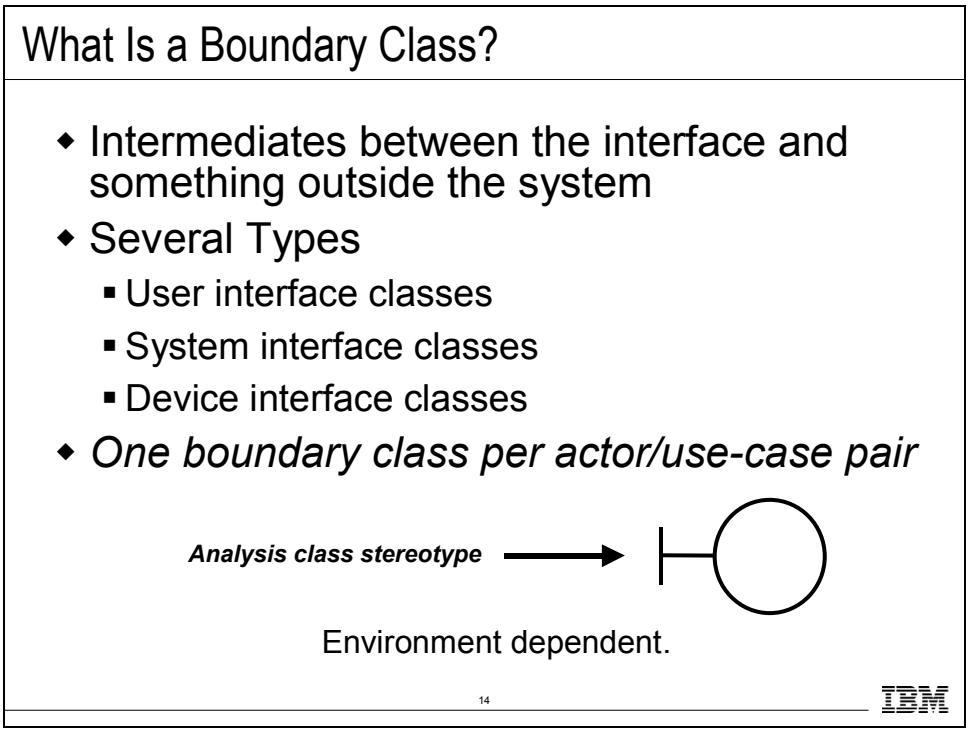
- Boundary
- Entity
- Control

Stereotypes may be defined for each type. These distinctions are used during Analysis, but disappear in Design.

The different types of analysis classes can be represented using different icons or with the name of the stereotype in guillemets (<< >>): <<boundary>>, <<control>>, <<entity>>.

Each of these types of analysis classes are discussed on the following slides.

What Is a Boundary Class?



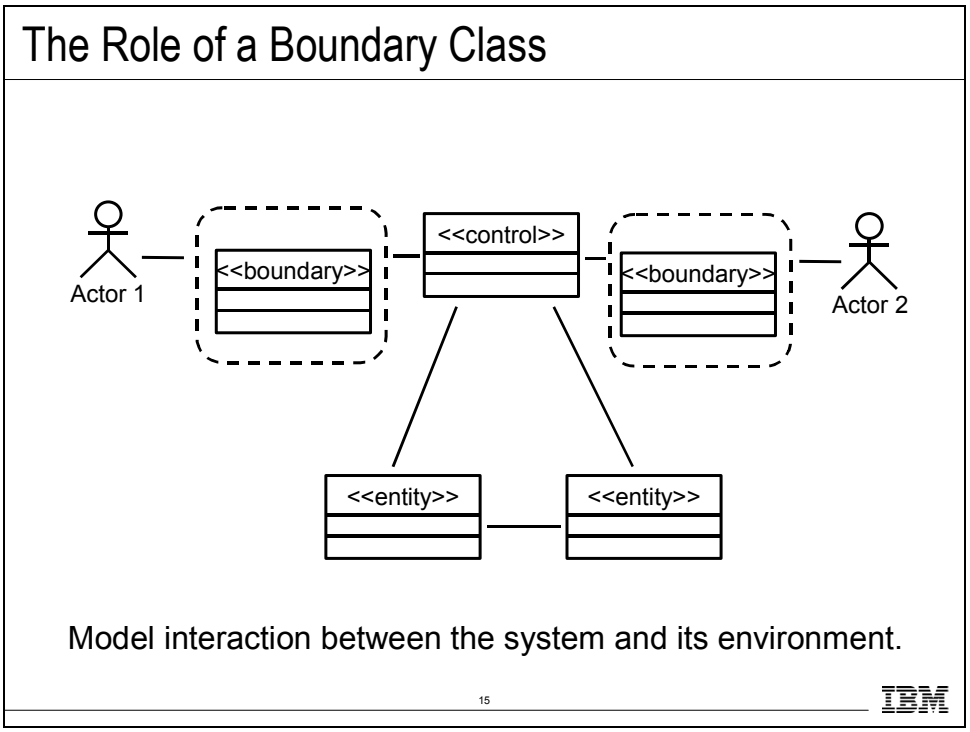
A boundary class intermediates between the interface and something outside the system. Boundary classes insulate the system from changes in the surroundings (for example, changes in interfaces to other systems and changes in user requirements), keeping these changes from affecting the rest of the system.

A system can have several types of boundary classes:

- **User interface classes**—Classes that intermediate communication with human users of the system.
- **System interface classes**—Classes that intermediate communication with other systems. A boundary class that communicates with an external system is responsible for managing the dialog with the external system; it provides the interface to that system for the system being built.
- **Device interface classes**—Classes that provide the interface to devices which detect external events. These boundary classes capture the responsibilities of the device or sensor.

One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair.

The Role of a Boundary Class



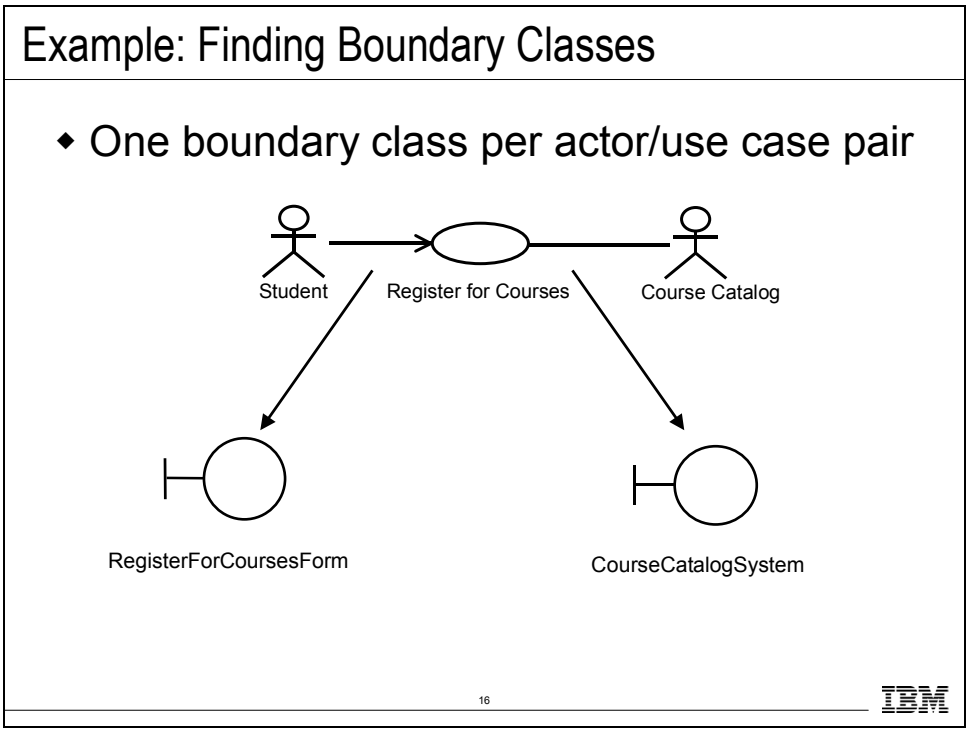
A boundary class is used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface).

Boundary classes model the parts of the system that depend on its surroundings. They make it easier to understand the system because they clarify the system's boundaries and aid design by providing a good point of departure for identifying related services. For example, if you identify a printer interface early in the design, you will realize that you must also model the formatting of printouts.

Because boundary classes are used between actors and the working of the internal system (actors can only communicate with boundary classes), they insulate external forces from internal mechanisms and vice versa. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.

A boundary object (an instance of a boundary class) can outlive a use-case instance if, for example, it must appear on a screen between the performance of two use cases. Normally, however, boundary objects live only as long as the use-case instance.

Example: Finding Boundary Classes



The goal of Analysis is to form a good picture of how the system is composed, not to design every last detail. In other words, identify boundary classes only for phenomena in the system or for things mentioned in the flow of events of the Use-Case Realization.

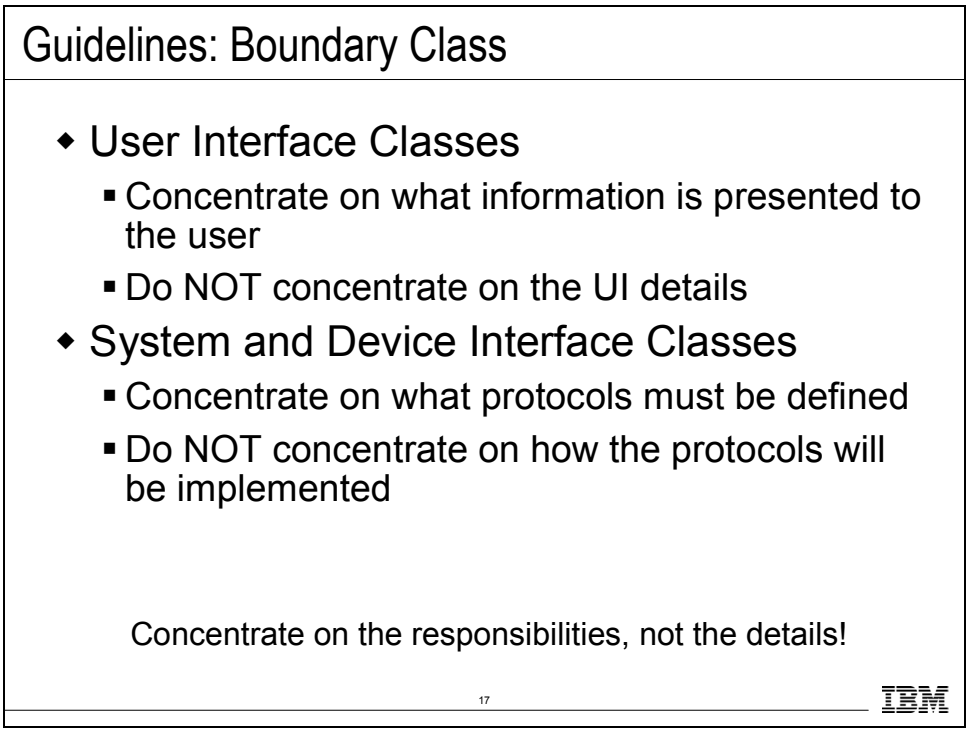
Consider the source for all external events and make sure there is a way for the system to detect these events.

One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair. This class can be viewed as having responsibility for coordinating the interaction with the actor. This may be refined as a more detailed analysis is performed. This is particularly true for window-based GUI applications where there is typically one boundary class for each window, or one for each dialog box.

In the above example:

- The RegisterForCoursesForm contains a Student's "schedule-in-progress." It displays a list of Course Offerings for the current semester from which the Student may select courses to be added to his or her Schedule.
- The CourseCatalogSystem interfaces with the legacy system that provides the unabridged catalog of all courses offered by the university. This class replaces the CourseCatalog abstraction originally identified in Architectural Analysis.

Guidelines: Boundary Class

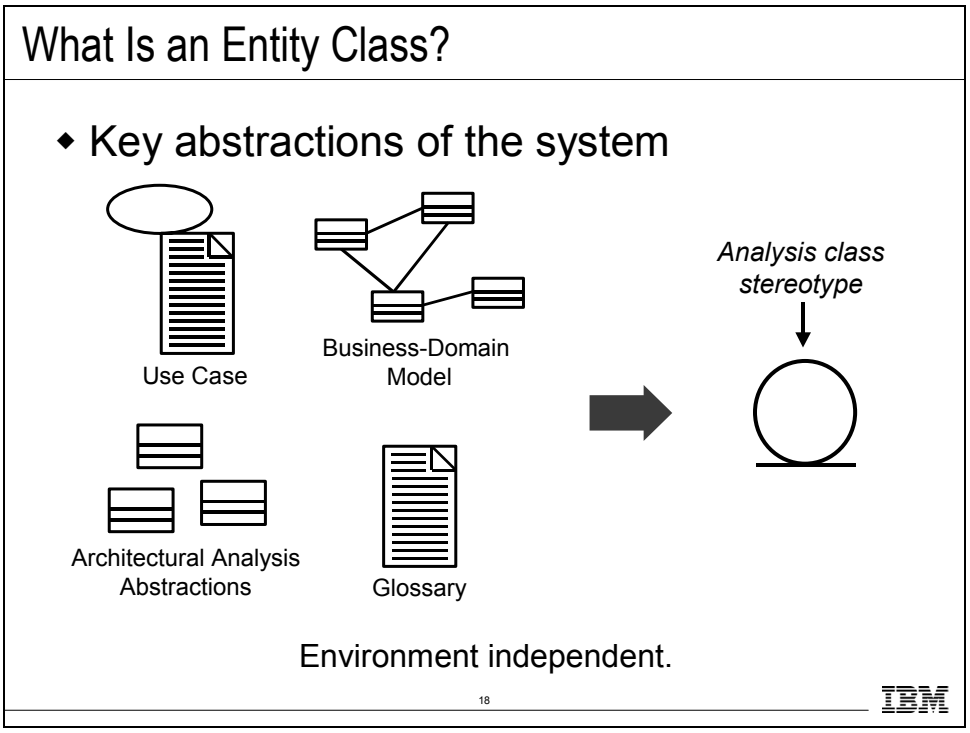


When identifying and describing analysis classes, be careful not to spend too much time on the details. Analysis classes are meant to be a first cut at the abstractions of the system. They help to clarify the understanding of the problem to be solved and represent an attempt at an idealized solution (Analysis has been called "idealized Design").

User Interface Classes: Boundary classes may be used as "holding places" for GUI classes. The objective is not to do GUI design in this analysis, but to isolate all environment-dependent behavior. The expansion, refinement and replacement of these boundary classes with actual user-interface classes (probably derived from purchased UI libraries) is a very important activity of Class Design and will be discussed in the Class Design module. Sketches or screen captures from a user-interface prototype may have been used during the Requirements discipline to illustrate the behavior and appearance of the boundary classes. These may be associated with a boundary class. However, only model the key abstractions of the system; do not model every button, list, and widget in the GUI.

System and Device Interface Classes: If the interface to an existing system or device is already well-defined, the boundary class responsibilities should be derived directly from the interface definition. If there is a working communication with the external system or device, make note of it for later reference during design.

What Is an Entity Class?



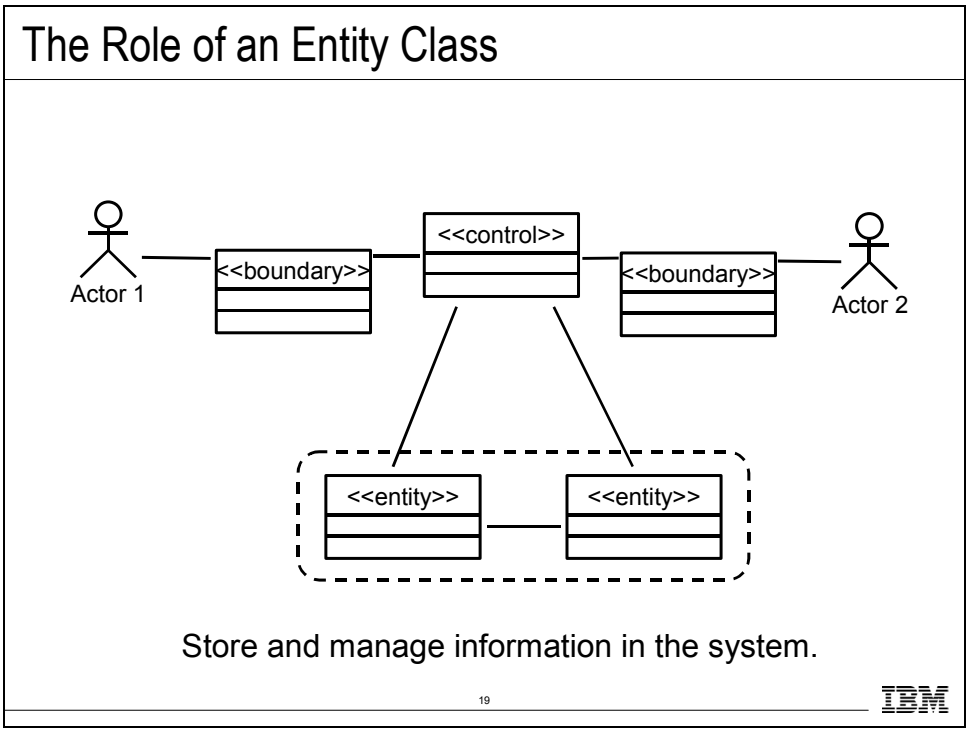
Entity objects represent the key concepts of the system being developed. Entity classes provide another point of view from which to understand the system, because they show the logical data structure. Knowing the data structure can help you understand what the system is supposed to offer its users.

Frequent sources of inspiration for entity classes are the:

- Glossary (developed during requirements)
- Business-Domain Model (developed during business modeling, if business modeling has been performed)
- Use-case flow of events (developed during requirements)
- Key abstractions (identified in Architectural Analysis)

As mentioned earlier, sometimes there is a need to model information about an actor within the system. This is not the same as modeling the actor (actors are external. by definition). In this case, the information about the actor is modeled as an entity class. These classes are sometimes called “surrogates or proxies.”

The Role of an Entity Class

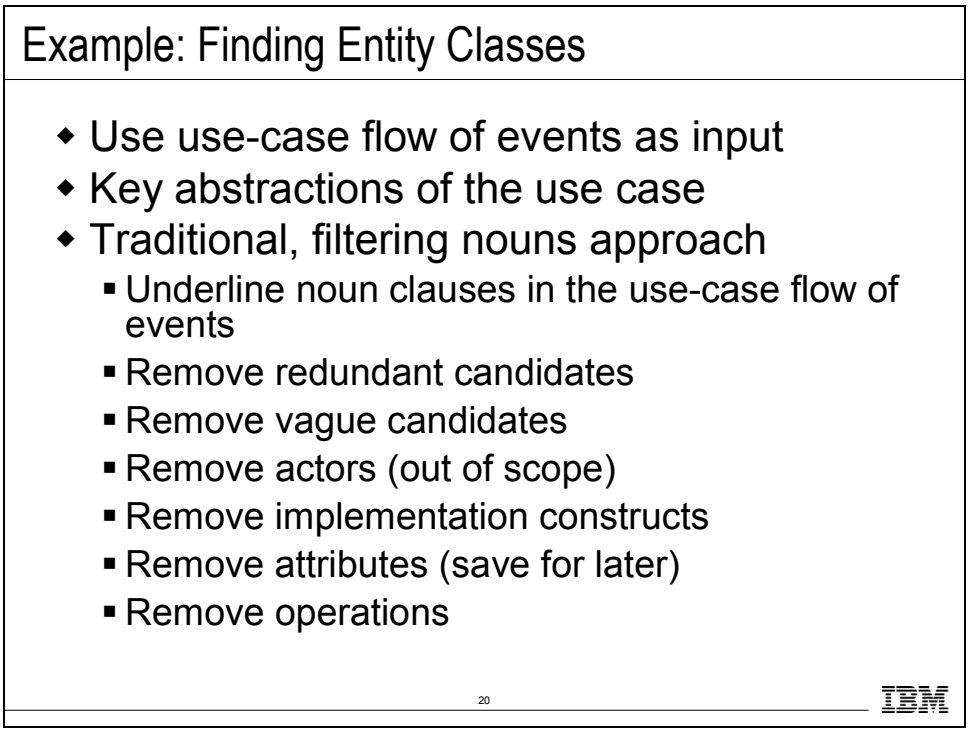


Entity classes represent stores of information in the system. They are typically used to represent the key concepts that the system manages. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or a real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the lifetime of the system.

The main responsibilities of entity classes are to store and manage information in the system.

An entity object is usually not specific to one Use-Case Realization and sometimes it is not even specific to the system itself. The values of its attributes and relationships are often given by an actor. An entity object may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon the entity object represents. Entity objects are independent of the environment (the actors).

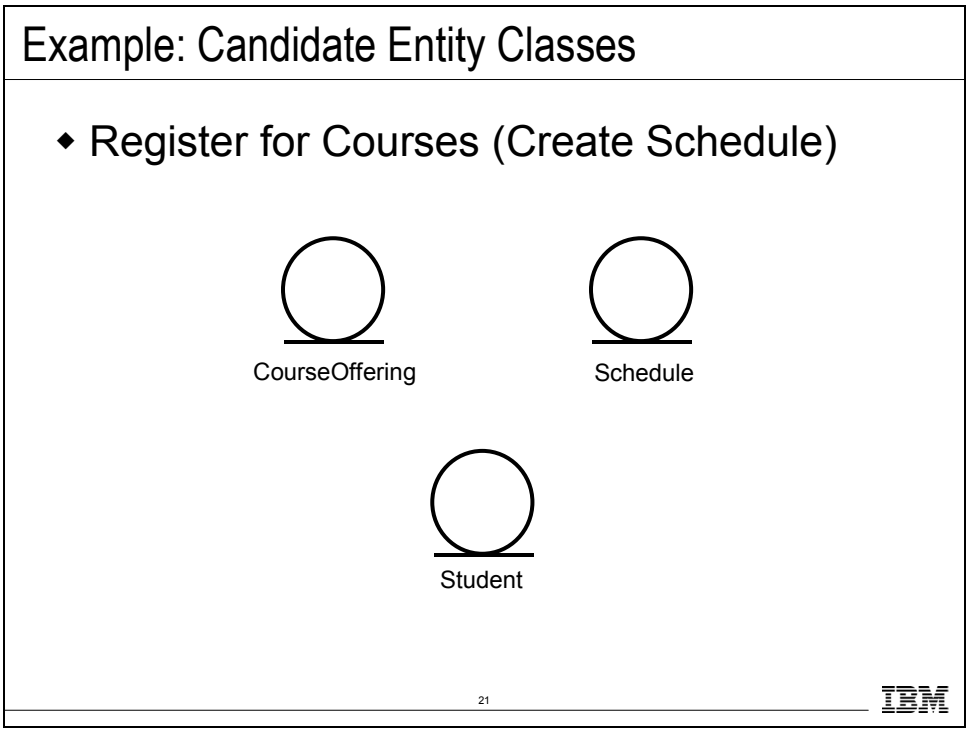
Example: Finding Entity Classes



Taking the use-case flow of events as input, underline the noun phrases in the flow of events. These form the initial candidate list of analysis classes.

Next, go through a series of filtering steps where some candidate classes are eliminated. This is necessary due to the ambiguity of the English language. The result of the filtering exercise is a refined list of candidate entity classes. While the filtering approach does add some structure to what could be an ad-hoc means of identifying classes, people generally filter as they go rather than blindly accepting all nouns and then filtering.

Example: Candidate Entity Classes



The following are the definitions for each of the classes shown in the above diagram:

CourseOffering: A specific offering for a course, including days of the week and times.

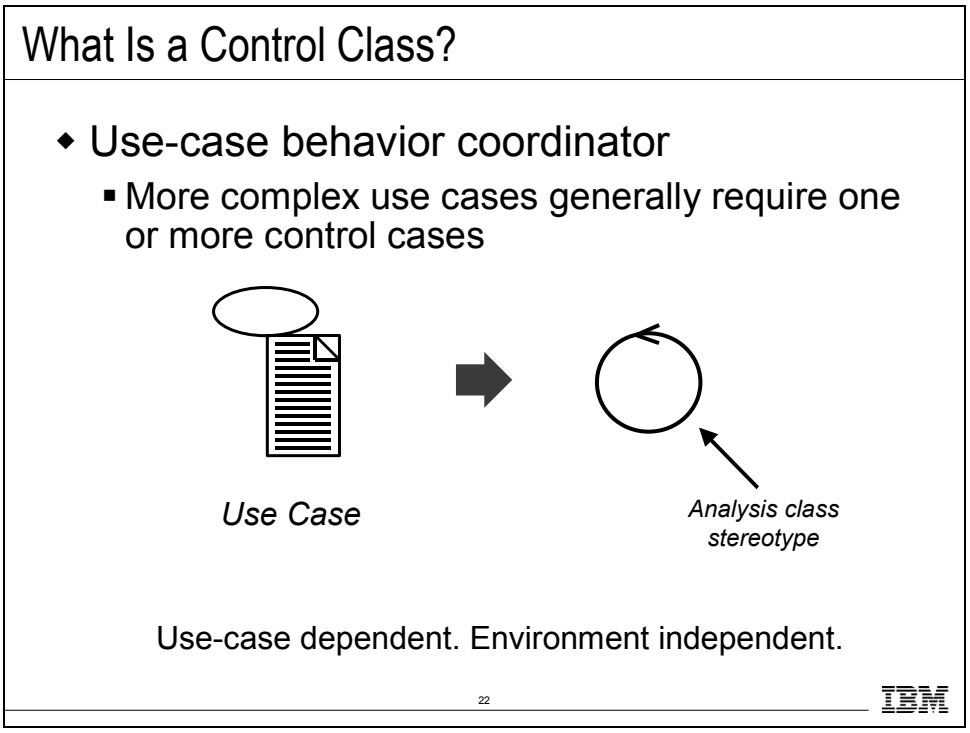
Schedule: The courses a student has selected for the current semester.

Student: A person enrolled in classes at the university.

As mentioned earlier, sometimes there is a need to model information about an actor within the system. This is not the same as modeling the actor. (Actors are external by definition.) These classes are sometimes called "surrogates".

For example, a course registration system maintains information about the student that is independent of the fact that the student also plays a role as an actor in the system. This information about the student is stored in a "Student" class that is completely independent of the "actor" role the student plays. The Student class will exist whether or not the student is an actor to the system.

What Is a Control Class?



Control classes provide coordinating behavior in the system. The system can perform some use cases without control classes by using just entity and boundary classes. This is particularly true for use cases that involve only the simple manipulation of stored information. More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control classes include transaction managers, resource coordinators, and error handlers.

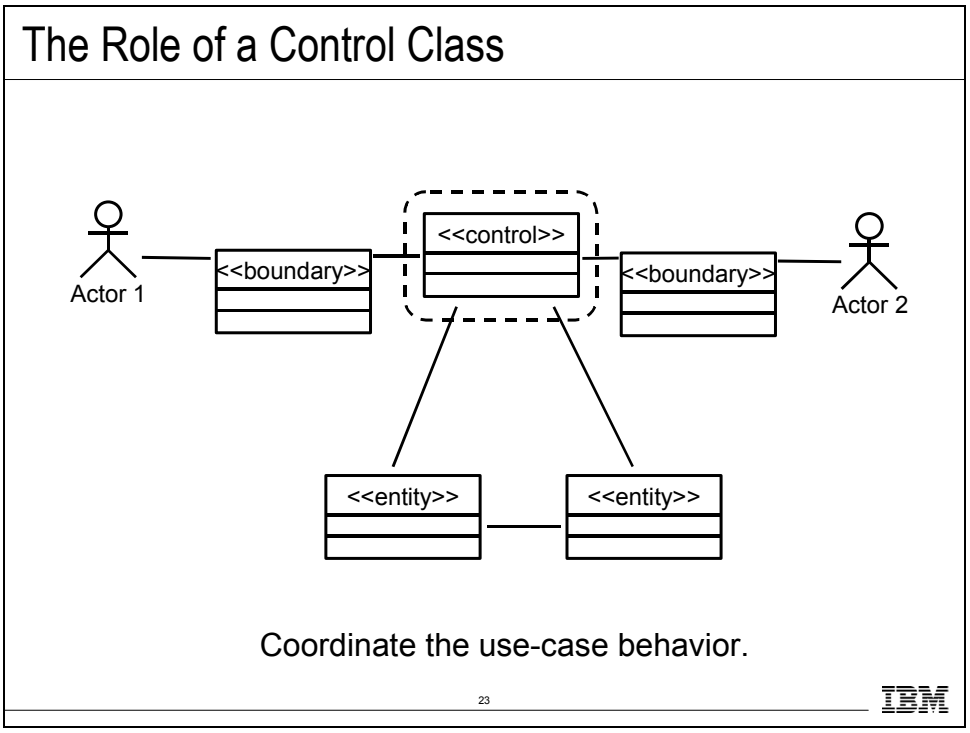
Control classes effectively decouple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also decouple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

- Is surroundings-independent (does not change when the surroundings change).
- Defines control logic (order between events) and transactions within a use case.
- Changes little if the internal structure or behavior of the entity classes changes.
- Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
- Is not performed in the same way every time it is activated (flow of events features several states).

Although complex use cases may need more than one control class it is recommended, for the initial identification of control classes, that only one control class be created per use case.

The Role of a Control Class



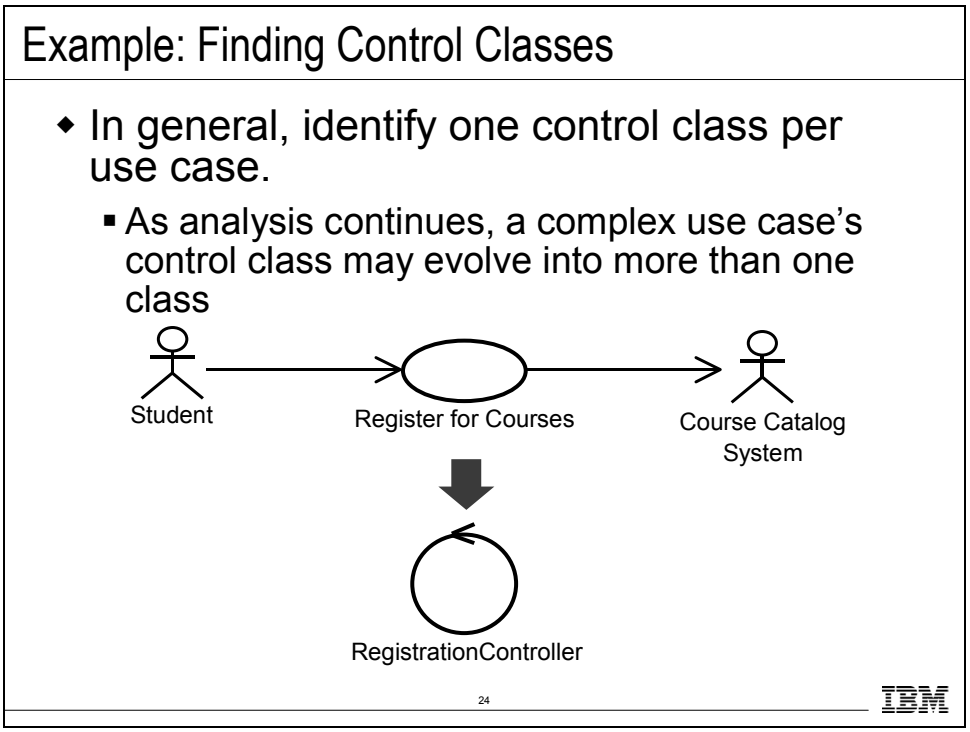
A control class is a class used to model control behavior specific to one or more use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case-specific behavior.

The behavior of a control object is closely related to the realization of a specific use case. In many scenarios, you might even say that the control objects "run" the Use-Case Realizations. However, some control objects can participate in more than one Use-Case Realization if the use-case tasks are strongly related. Furthermore, several control objects of different control classes can participate in one use case. Not all use cases require a control object. For example, if the flow of events in a use case is related to one entity object, a boundary object may realize the use case in cooperation with the entity object. You can start by identifying one control class per Use-Case Realization, and then refine this as more Use-Case Realizations are identified, and commonality is discovered.

Control classes can contribute to understanding the system, because they represent the dynamics of the system, handling the main tasks and control flows.

When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

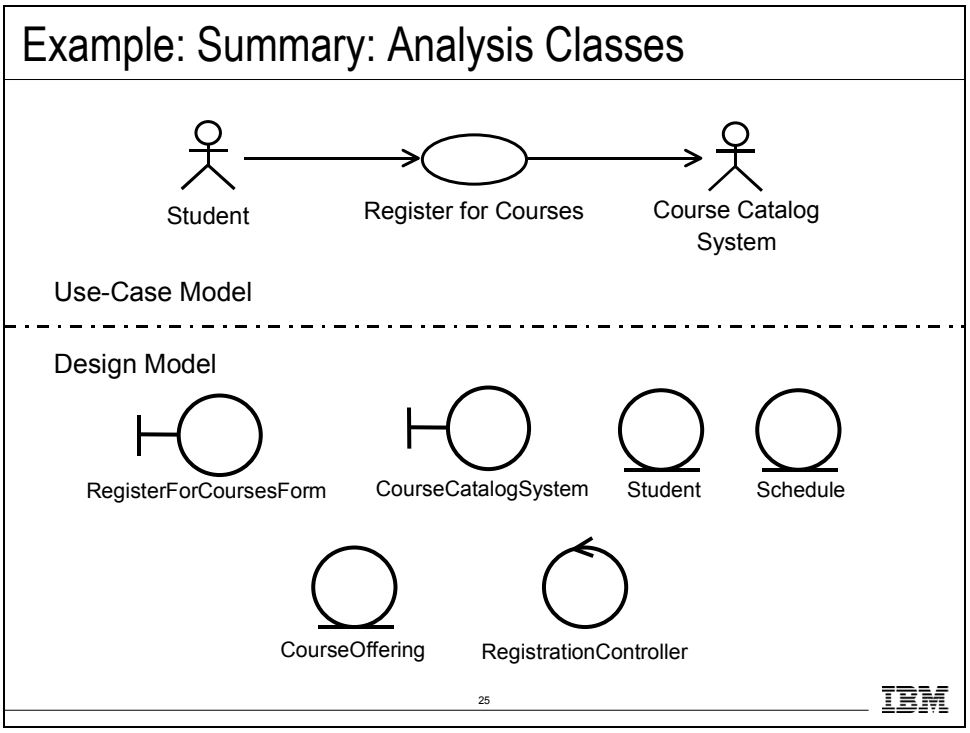
Example: Finding Control Classes



One recommendation is to identify one control class per use case. However, this can become more than one use case as analysis continues. Remember that more complex use cases generally require one or more control cases. Each control class is responsible for orchestrating/controlling the processing that implements the functionality described in the associated use case.

In the above example, the RegistrationController <<control>> class has been defined to orchestrate the Register for Courses processing within the system.

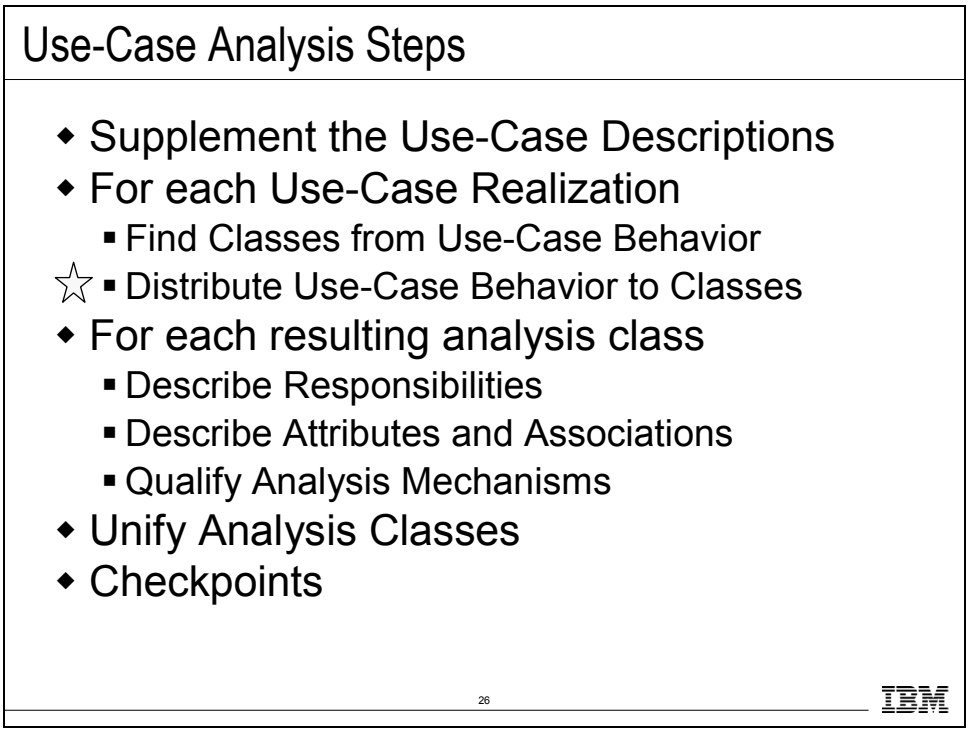
Example: Summary: Analysis Classes



For each Use-Case Realization, there is one or more class diagrams depicting its participating classes, along with their relationships. These diagrams help to ensure that there is consistency in the use-case implementation across subsystem boundaries. Such class diagrams have been called "View of Participating Classes" diagrams (VOPC, for short).

The diagram on this slide shows the classes participating in the "Register for Courses" use case. Class relationships will be discussed later in this module.

Use-Case Analysis Steps

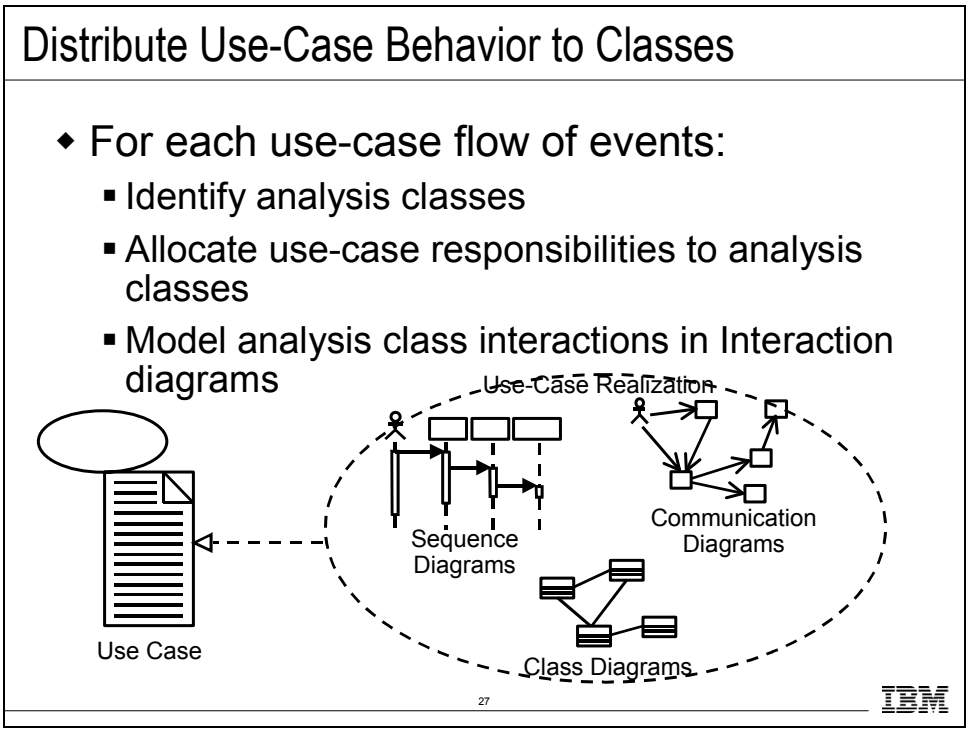


Now that we have identified the candidate analysis classes, we need to allocate the responsibilities of the use case to the analysis classes and model this allocation by describing the way the class instances collaborate to perform the use case in Use-Case Realization.

The purpose of "Distribute Use-Case Behavior to Classes" is to:

- Express the use-case behavior in terms of collaborating analysis classes
- Determine the responsibilities of analysis classes

Distribute Use-Case Behavior to Classes



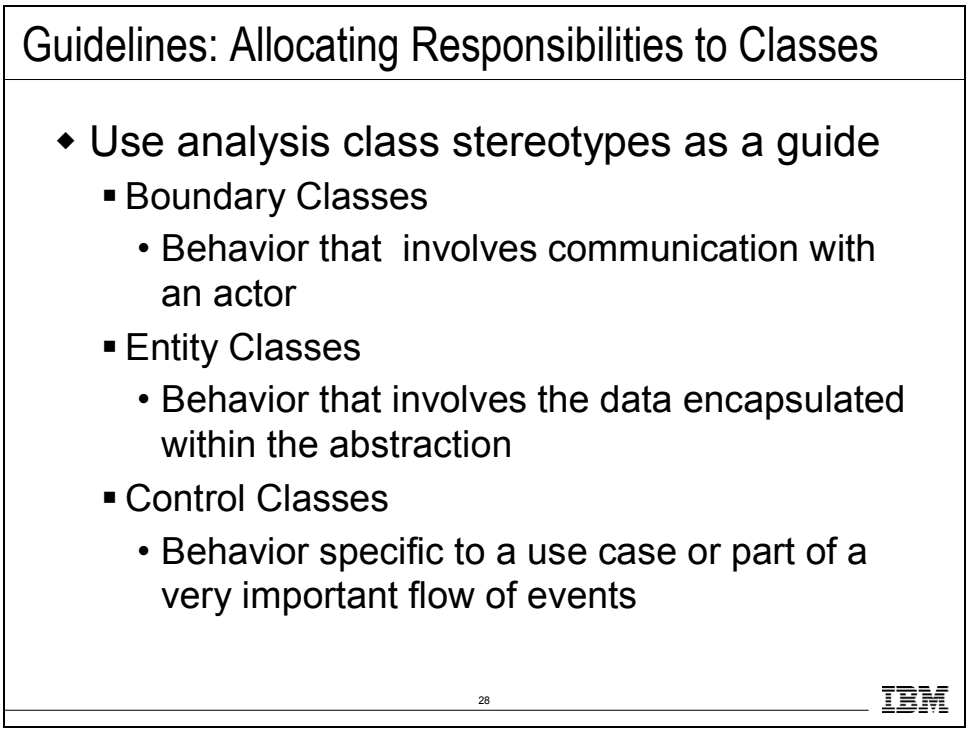
You can identify analysis classes responsible for the required behavior by stepping through the flow of events of the use case. In the previous step, we outlined some classes. Now it is time to see exactly where they are applied in the use-case flow of events.

In addition to the identified analysis classes, the Interaction diagram should show interactions of the system with its actors. The interactions should begin with an actor, since an actor always invokes the use case. If you have several actor instances in the same diagram, try keeping them in the periphery of that diagram.

Interactions *between* actors should *not* be modeled. By definition, actors are external, and are out of scope of the system being developed. Thus, you do not include interactions between actors in your system model. If you need to model interactions between entities that are external to the system that you are developing (for example, the interactions between a customer and an order agent for an order-processing system), those interactions are best included in a Business Model that drives the System Model.

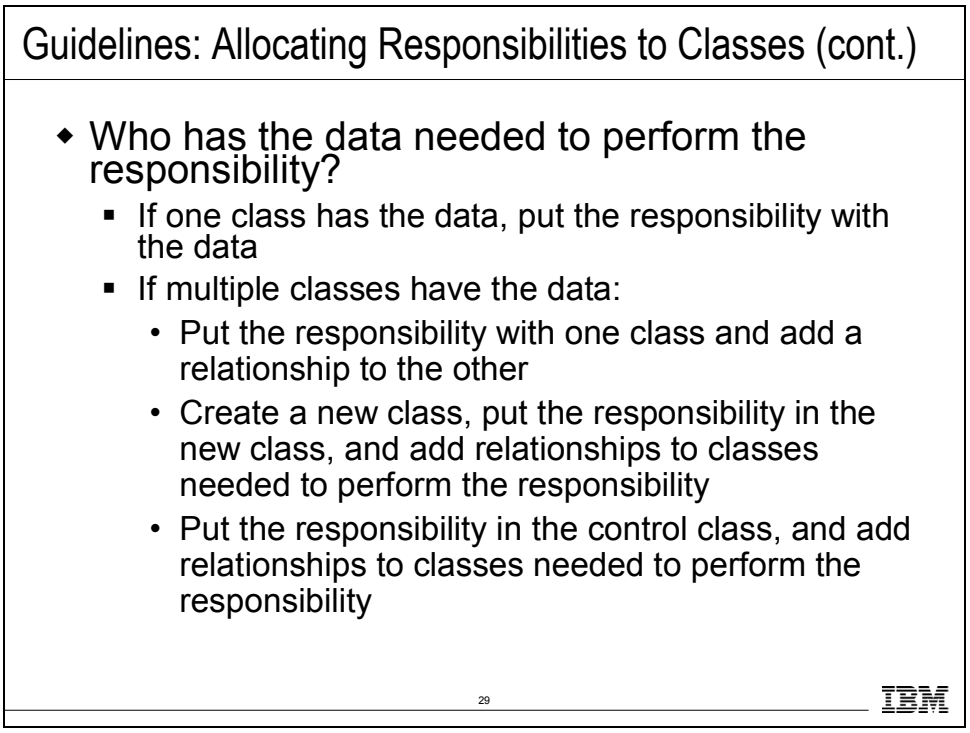
Guidelines for how to distribute behavior to classes are described on the next slide.

Guidelines: Allocating Responsibilities to Classes



The allocation of responsibilities in analysis is a crucial and sometimes difficult activity. These three stereotypes make the process easier by providing a set of canned responsibilities that can be used to build a robust system. These predefined responsibilities isolate the parts of the system that are most likely to change: the interface (boundary classes), the use-case flow of events (control classes), and the persistent data (entity classes).

Guidelines: Allocating Responsibilities to Classes (cont.)



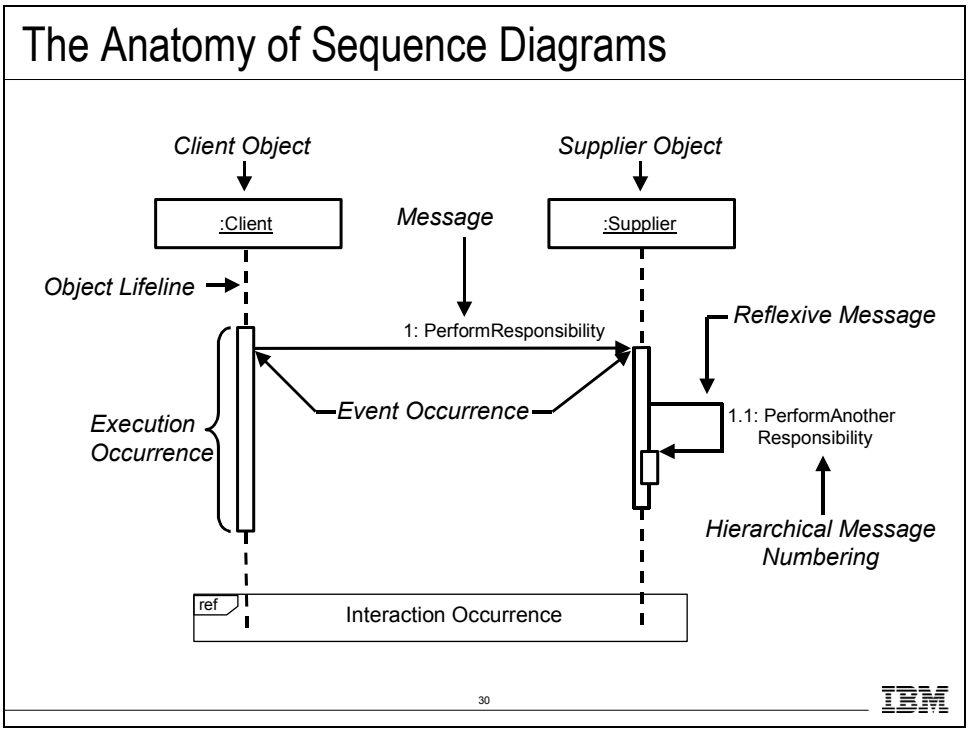
A driving influence on where a responsibility should go is the location of the data needed to perform the operation.

The best case is that there is one class that has all the information needed to perform the responsibility. In that case, the responsibility goes with the data (after all, that is one of the tenets of OO — data and operations together).

If this is not the case, the responsibility may need to be allocated to a "third party" class that has access to the information needed to perform the responsibility. Classes and/or relationships might need to be created to make this happen. Be careful when adding relationships — all relationships should be consistent with the abstractions they connect. Do not just add relationships to support the implementation without considering the overall effect on the model. Class relationships will be discussed later in this module.

When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. You should create new classes only when you are sure that there is no existing object that can perform the behavior.

The Anatomy of Sequence Diagrams



A Sequence diagram describes a pattern of interaction among objects, arranged in a chronological order. It shows the objects participating in the interaction and the messages they send.

An **object** is shown as a vertical dashed line called the "lifeline." The lifeline represents the existence of the object at a particular time. An object symbol is drawn at the head of the lifeline, and shows the name of the object and its class separated by a colon and underlined.

A **message** is a communication between objects that conveys information with the expectation that activity will result. A message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. For a reflexive message, the arrow starts and finishes on the same lifeline. The arrow is labeled with the name of the message and its parameters. The arrow may also be labeled with a sequence number.

Execution Occurrence represents the relative time that the flow of control is focused in an object, thereby representing the time an object is directing messages. Execution occurrence is shown as narrow rectangles on object lifelines.

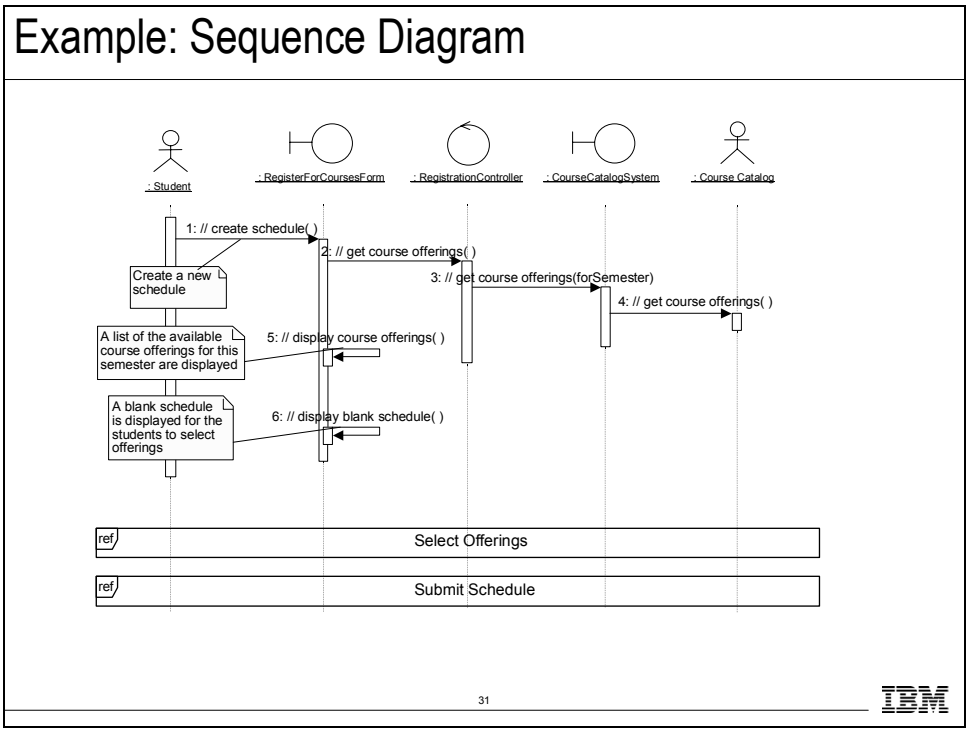
Event Occurrence represents the sending or receipt of messages.

Interaction Occurrence is a reference to an interaction within the definition of another interaction.

Hierarchical numbering bases all messages on a dependent message. The dependent message is the message whose execution occurrence the other messages originate in. For example, message 1.1 depends on message 1.

Notes describe the flow of events textually.

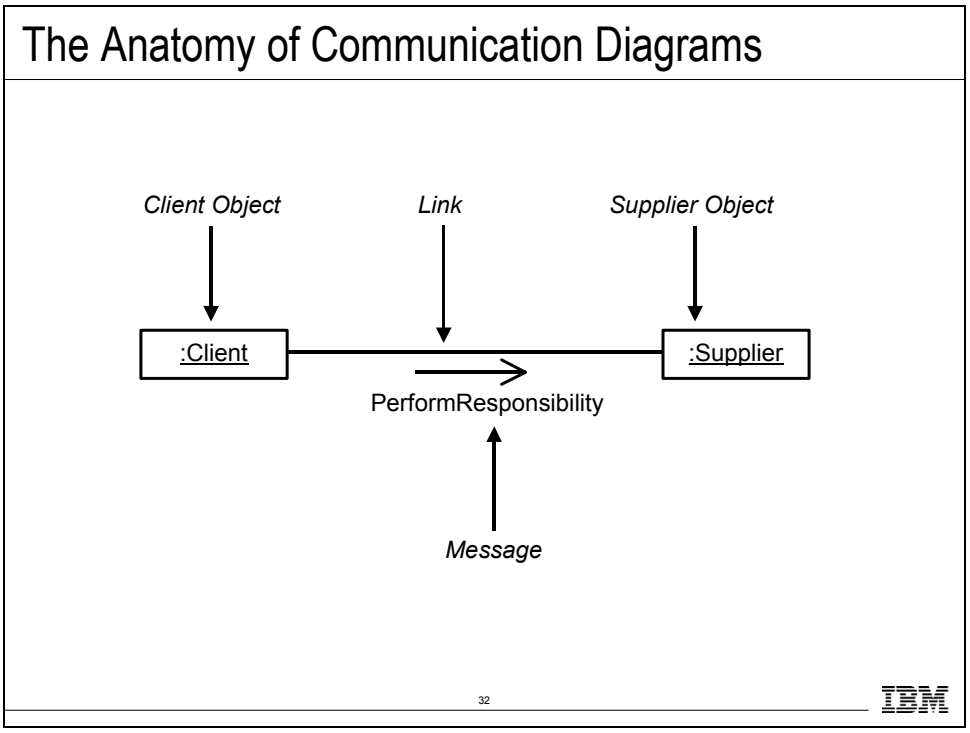
Example: Sequence Diagram



The above example shows the object interactions to support the Create a Schedule sub-flow of the Register for Courses use case. Some of the rationale for responsibility allocation is as follows:

- The RegisterForCoursesForm knows what data it needs to display and how to display it. It does not know where to go to get it. That is one of the RegistrationController's responsibilities.
- Only the RegisterForCoursesForm interacts with the Student actor.
- Only the CourseCatalogSystem class interacts with the external legacy Course Catalog System.
- Note the inclusion of the actors. This is important as the diagram explicitly models what elements communicate with the "outside world."

The Anatomy of Communication Diagrams



A Communication diagram describes a pattern of interaction among objects. It shows the objects participating in the interaction by their links to each other and the messages that they send to each other.

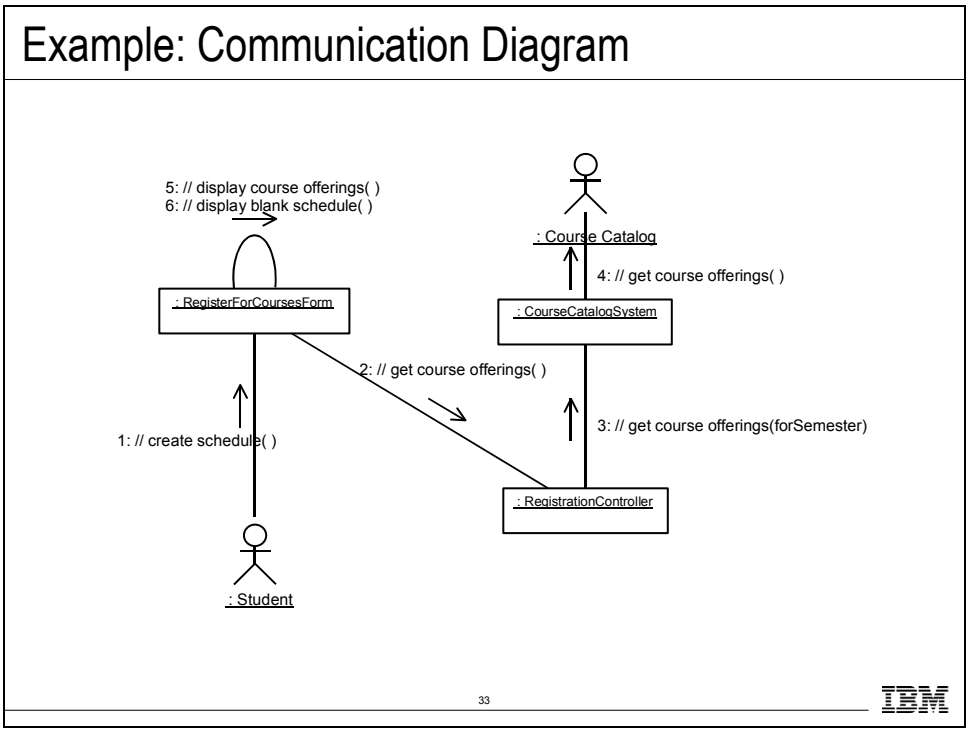
An **object** is represented in one of three ways:

- Objectname:Classname
- ObjectName
- :ClassName

A **link** is a relationship between objects that can be used to send messages. In Communication diagrams, a link is shown as a solid line between two objects. An object interacts with, or navigates to, other objects through its links to these objects. A link is defined as an instance of an association.

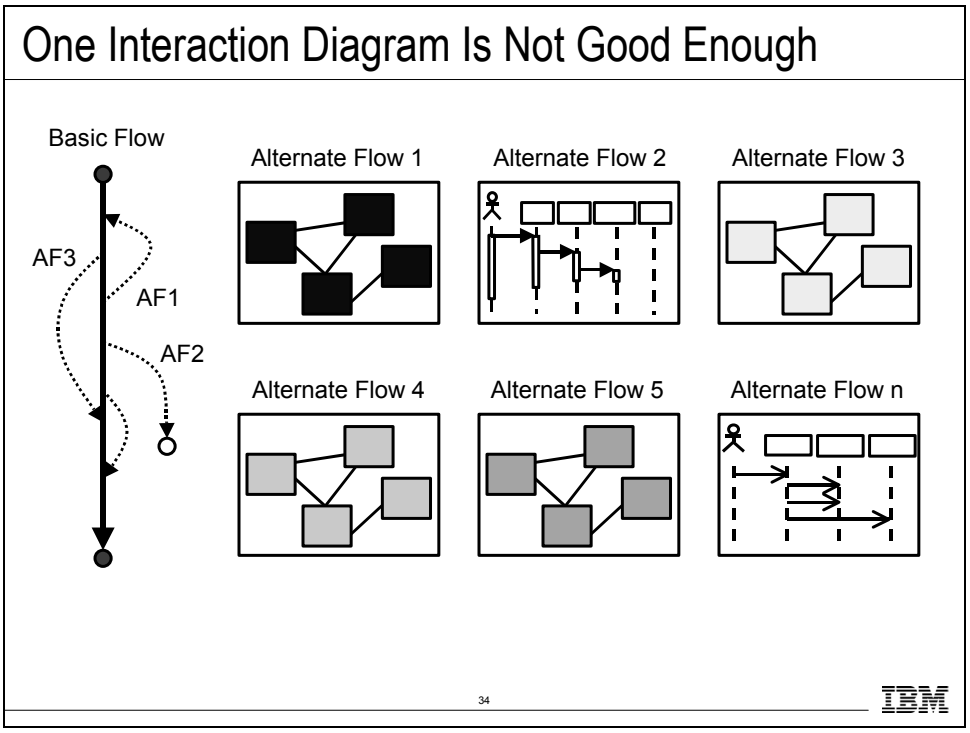
A **message** is a communication between objects that conveys information with the expectation that activity will result. In Communication diagrams, a message is shown as a labeled arrow placed near a link. This means that the link is used to transport or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message). The arrow is labeled with the name of the message and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often used in Communication diagrams because they are the only way of describing the relative sequencing of messages. A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. You can later assign the message by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

Example: Communication Diagram



The above example shows the collaboration of objects to support the Register for Courses use case, Create a Schedule subflow. It is the "Communication diagram equivalent" of the Sequence diagram shown earlier.

One Interaction Diagram Is Not Good Enough



Model most of the flow of events to make sure that all requirements on the operations of the participating classes are identified. Start with describing the basic flow, which is the most common or most important flow of events. Then describe variants such as exceptional flows. You do not have to describe all the flow of events, as long as you employ and exemplify all operations of the participating objects. Very trivial flows can be omitted, such as those that concern only one object.

Examples of exceptional flows include the following:

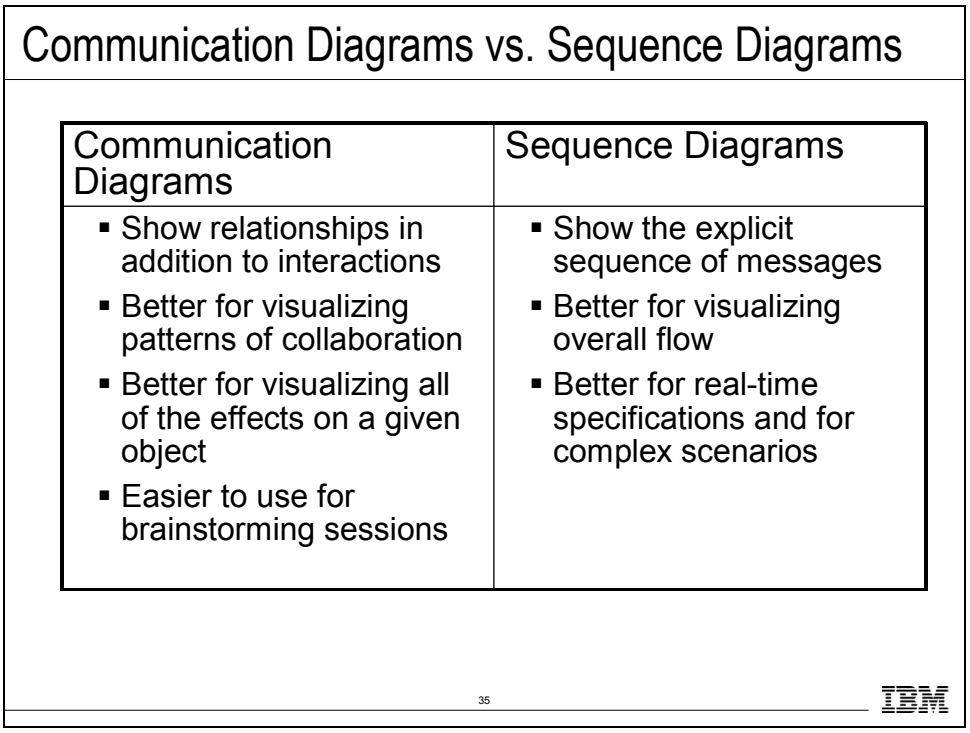
- Error handling. What should the system do if an error is encountered?
- Time-out handling. If the user does not reply within a certain period, the use case should take some special measures.
- Handling of erroneous input to the objects that participate in the use case (for example, incorrect user input).

Examples of optional flows include the following:

- The actor decides-from a number of options — what the system is to do next.
- The subsequent flow of events depends on the value of stored attributes or relationships.
- The subsequent flow of events depends on the type of data to be processed.

You can use either Communication or Sequence diagrams.

Communication Diagrams vs. Sequence Diagrams



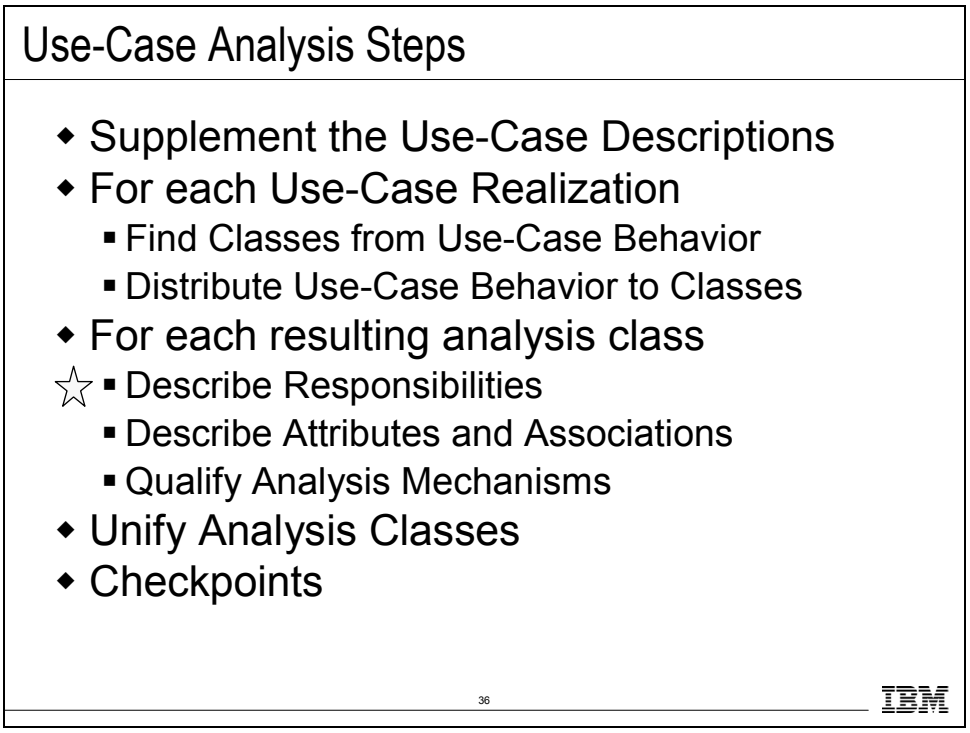
Sequence diagrams and Communication diagrams express similar information, but show it in different ways.

Communication diagrams emphasize the structural collaboration of a society of objects and provide a clearer picture of the patterns of relationships and control that exist amongst the objects participating in a use case. Communication diagrams show more structural information (that is, the relationships among objects). Communication diagrams are better for understanding all the effects on a given object and for procedural design.

Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for complex scenarios. A Sequence diagram includes chronological sequences, but does not include object relationships. Sequence numbers are often omitted in Sequence diagrams, in which the physical location of the arrow shows the relative sequence. On Sequence diagrams, the time dimension is easier to read, operations and parameters are easier to present, and a larger number of objects are easier to manage than in Communication diagrams.

Both Sequence and Communication diagrams allow you to capture semantics of the use-case flow of events; they help identify objects, classes, interactions, and responsibilities; and they help validate the architecture.

Use-Case Analysis Steps

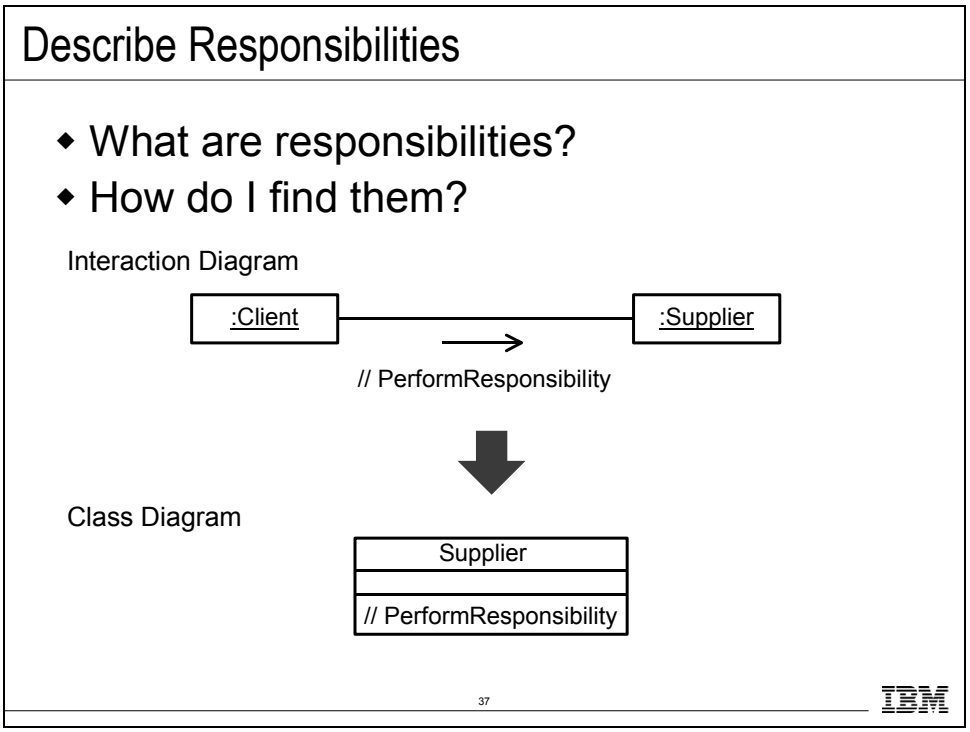


At this point, analysis classes have been identified and use-case responsibilities have been allocated to those classes. This was done on a use-case-by-use-case basis, with a focus primarily on the use-case flow of events. Now it is time to turn our attention to each of the analysis classes and see what each of the use cases will require of them. A class and its objects often participate in several Use-Case Realizations. It is important to coordinate all the requirements on a class and its objects that different Use-Case Realizations may have.

The ultimate objective of these class-focused activities is to to document what the class knows and what the class does. The resulting Analysis Model gives you a big picture and a visual idea of the way responsibilities are allocated and what such an allocation does to the class collaborations. Such a view allows the analyst to spot inconsistencies in the way certain classes are treated in the system, for example, how boundary and control classes are used.

The purpose of the Describe Responsibilities step is namely to describe the responsibilities of the analysis classes.

Describe Responsibilities



A responsibility is a statement of something an object can be asked to provide. Responsibilities evolve into one (or more) operations on classes in design; they can be characterized as:

- The actions that the object can perform.
- The knowledge that the object maintains and provides to other objects.

Responsibilities are derived from messages on Interaction diagrams. For each message, examine the class of the object to which the message is sent. If the responsibility does not yet exist, create a new responsibility that provides the requested behavior.

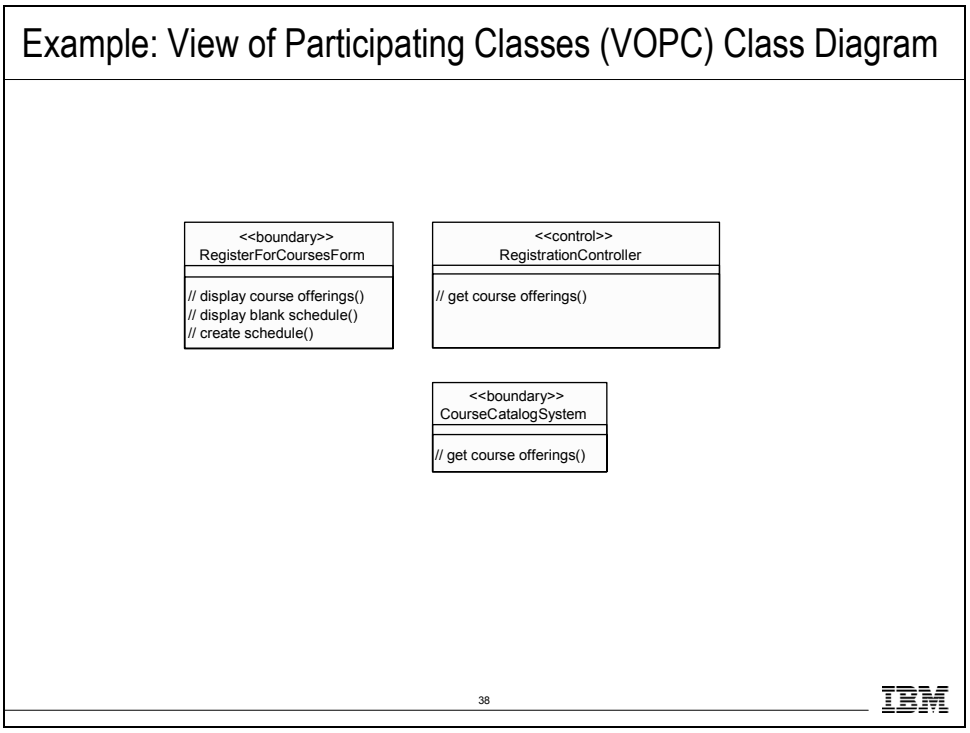
Other responsibilities will derive from nonfunctional requirements. When you create a new responsibility, check the nonfunctional requirements to see if there are related requirements that apply. Either augment the description of the responsibility, or create a new responsibility to reflect this.

Analysis class responsibilities can be documented in one of two ways:

- As "analysis" **operations**: When this approach is chosen, it is important that some sort of naming convention be used. This naming convention indicates that the operation is being used to describe the responsibilities of the analysis class and that these "analysis" operations WILL PROBABLY change/evolve in design.
- **Textually**: In this approach, the analysis class responsibilities are documented in the description of the analysis classes.

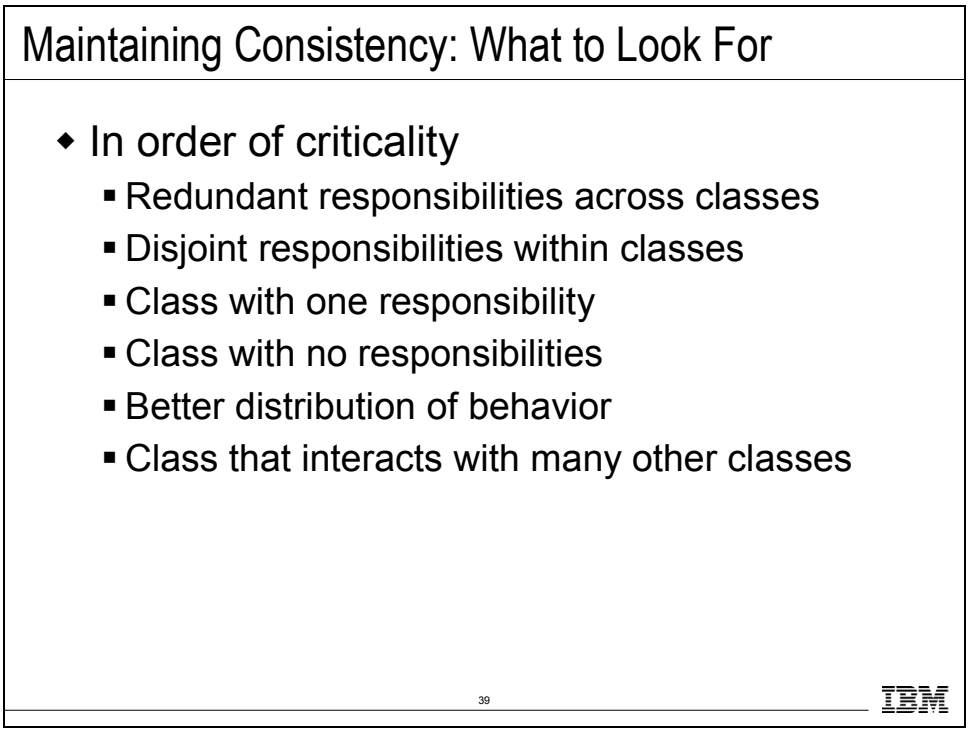
For the OOAD course example, we will use the "analysis" operation approach. The naming convention that will be used is that the "analysis" operation name will be preceded by '//'.

Example: View of Participating Classes (VOPC) Class Diagram



The View of Participating Classes (VOPC) class diagram contains the classes whose instances participate in the Use-Case Realization Interaction diagrams, as well as the relationships required to support the interactions. We will discuss the relationships later in this module. Right now, we are most interested in what classes have been identified, and what responsibilities have been allocated to those classes.

Maintaining Consistency: What to Look For



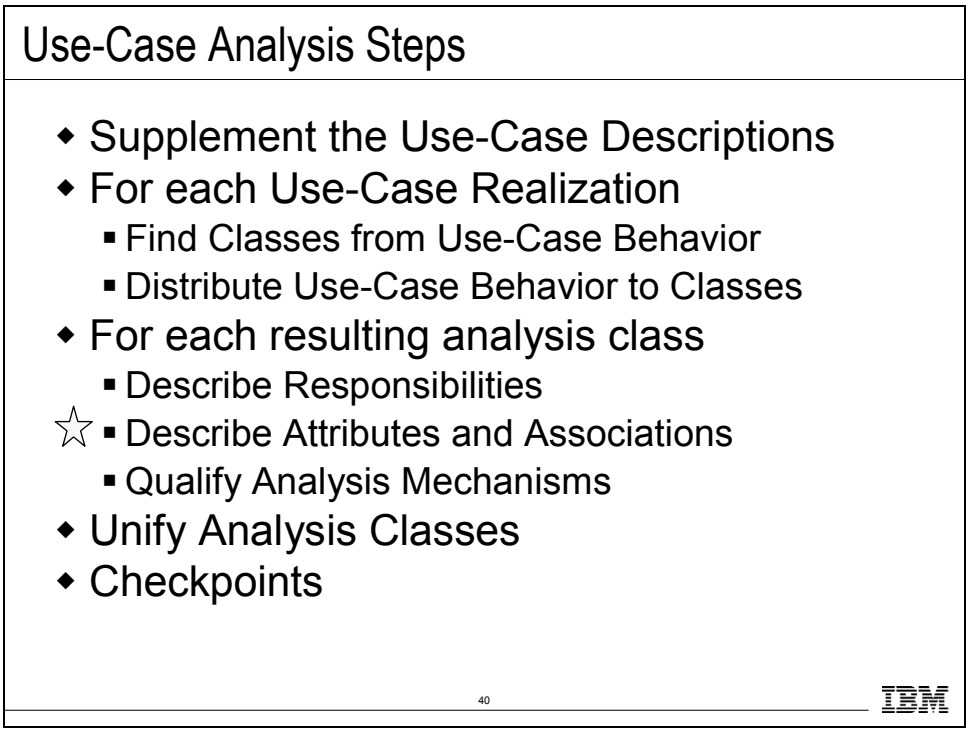
Examine classes to ensure they have consistent responsibilities. When a class's responsibilities are disjoint, split the object into two or more classes. Update the Interaction diagrams accordingly.

Examine classes to ensure that there are not two classes with similar responsibilities. When classes have similar responsibilities, combine them and update the Interaction diagrams accordingly.

Sometimes a better distribution of behavior becomes evident while you are working on another Interaction diagram. In this case, go back to the previous Interaction diagram and redo it. It is better (and easier) to change things now than later in design. Take the time to set the diagrams right, but do not get hungup trying to optimize the class interactions.

A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

Use-Case Analysis Steps

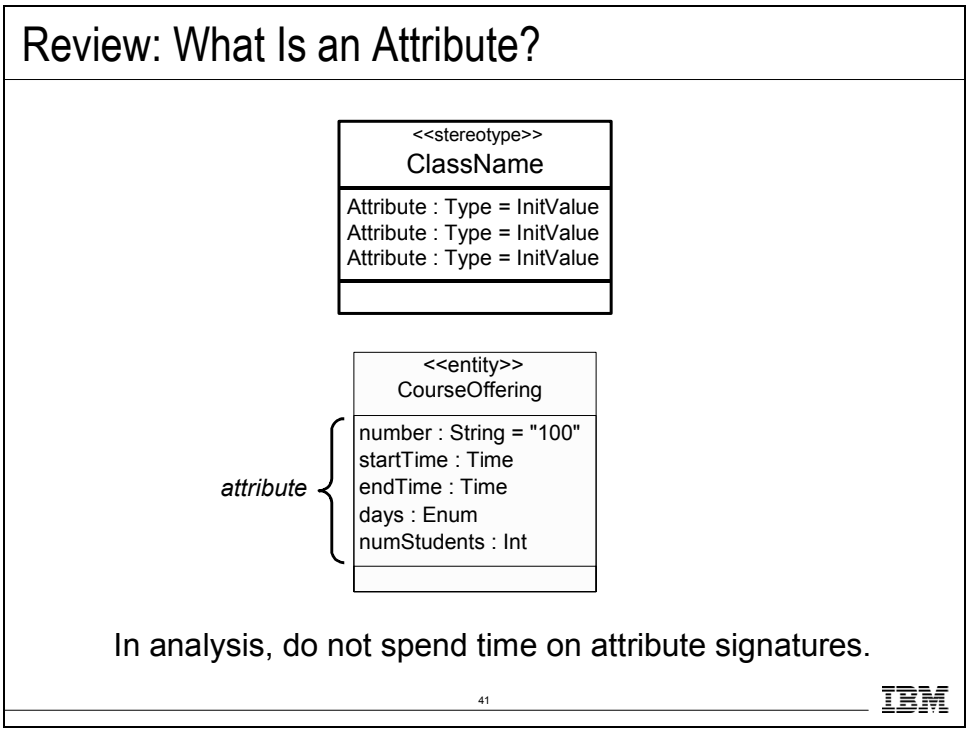


Now that we have defined the analysis classes and their responsibilities, and have an understanding of how they need to collaborate, we can continue our documentation of the analysis classes by describing their attributes and associations.

The purpose of Describe Attributes and Operations is to:

- Identify the other classes on which the analysis class depends.
- Define the events in other analysis classes that the class must know about.
- Define the information that the analysis class is responsible for maintaining.

Review: What Is an Attribute?

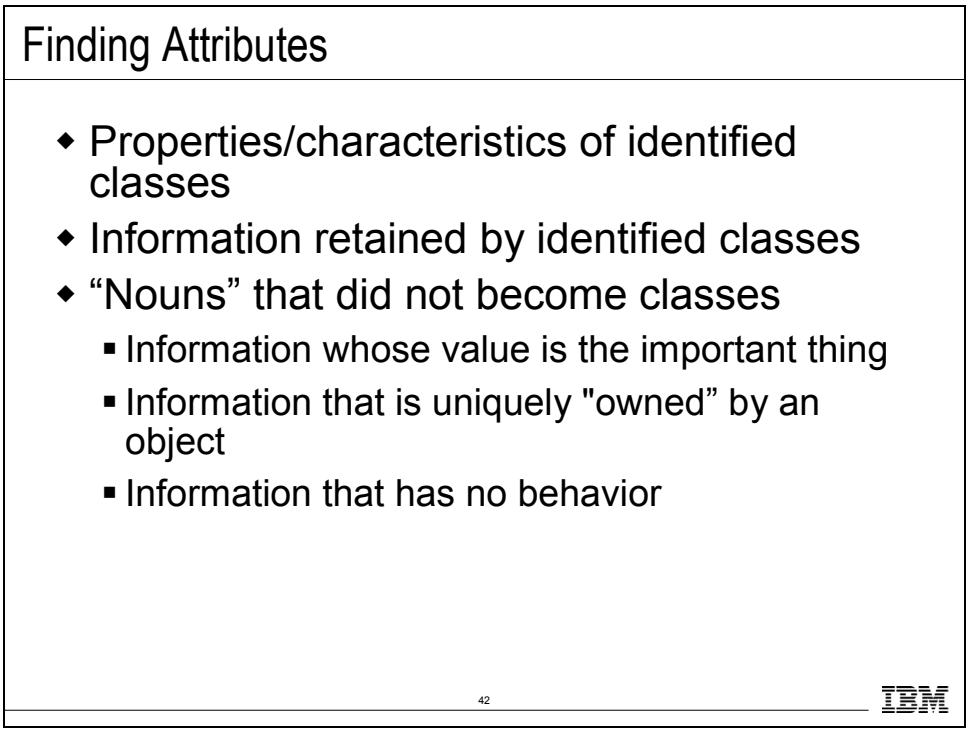


Attributes are used to store information. They are atomic things with no responsibilities.

The attribute name should be a noun that clearly states what information the attribute holds. The description of the attribute should describe what information is to be stored in the attribute; this can be optional when the information stored is obvious from the attribute name.

During Analysis, the attribute types should be from the domain, and not adapted to the programming language in use. For example, in the above diagram, enum will need to be replaced with a true enumeration that describes the days the CourseOffering is offered (for example, MWF or TR).

Finding Attributes



Sources of possible attributes:

- Domain knowledge
- Requirements
- Glossary
- Domain Model
- Business Model

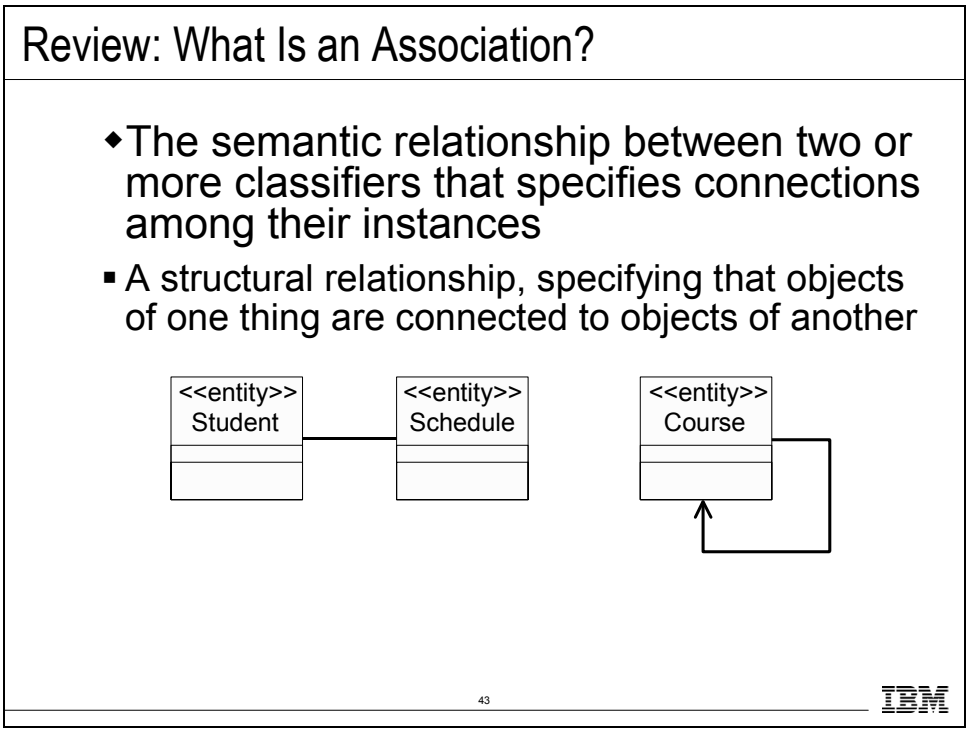
Attributes are used instead of classes where:

- Only the value of the information, not it's location, is important
- The information is uniquely "owned" by the object to which it belongs; no other objects refer to the information.
- The information is accessed by operations that only get, set, or perform simple transformations on the information; the information has no "real" behavior other than providing its value.

If, on the other hand, the information has complex behavior, or is shared by two or more objects, the information should be modeled as a separate class.
Attributes are domain-dependent. (An object model for a system includes those characteristics that are relevant for the problem domain being modeled.)

Remember, the process is use-case-driven. Thus, all discovered attributes should support at least one use case. For this reason, the attributes that are discovered are affected by what functionality/domain is being modeled.

Review: What Is an Association?



Associations represent structural relationships between objects of different classes; they connect instances of two or more classes together for some duration.

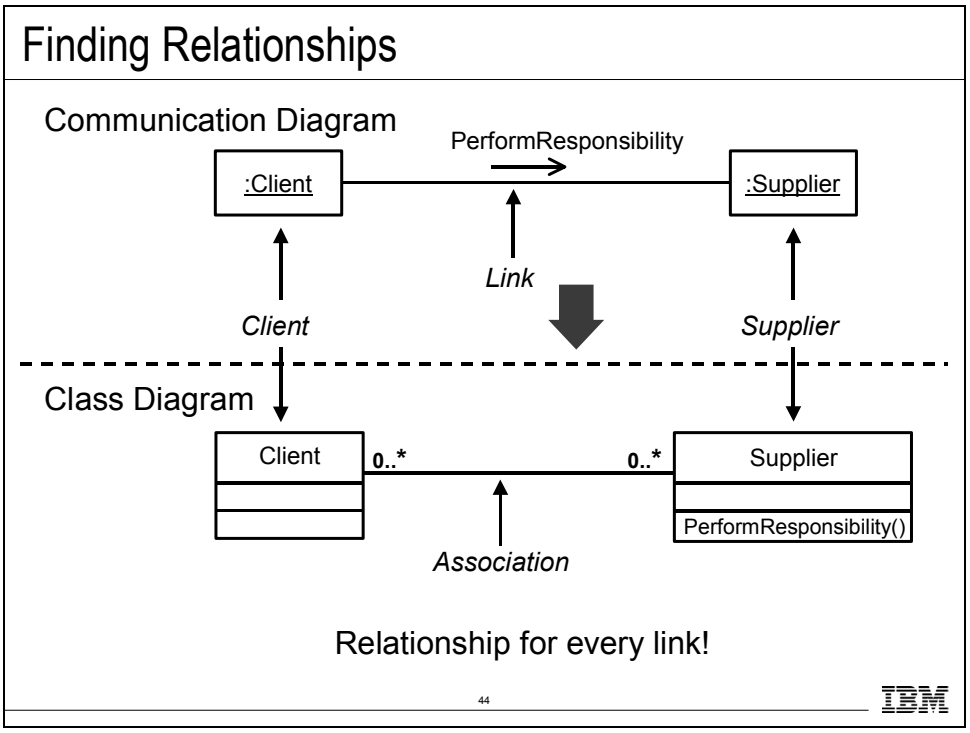
You can use associations to show that objects know about other objects. Sometimes, objects must hold references to each other to be able to interact; for example, to send messages to each other. Thus, in some cases, associations may follow from interaction patterns in Sequence diagrams or Communication diagrams.

Most associations are simple (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible. Sometimes a class has an association to itself. This does not necessarily mean that an instance of that class has an association to itself; more often, it means that one instance of the class has associations to other instances of the same class.

An association may have a name that is placed on, or adjacent to the association path. The name of the association should reflect the purpose of the relationship and be a verb phrase. The name of an association can be omitted, particularly if role names are used.

Avoid names like "has" and "contains," as they add no information about what the relationships are between the classes.

Finding Relationships



To find relationships, start studying the links in the Communication diagrams. Links between classes indicate that objects of the two classes need to communicate with one another to perform the use case. Thus, an association or an aggregation is needed between the associated classes.

Reflexive links do not need to be instances of reflexive relationships; an object can send messages to itself. A reflexive relationship is needed when two different objects of the same class need to communicate.

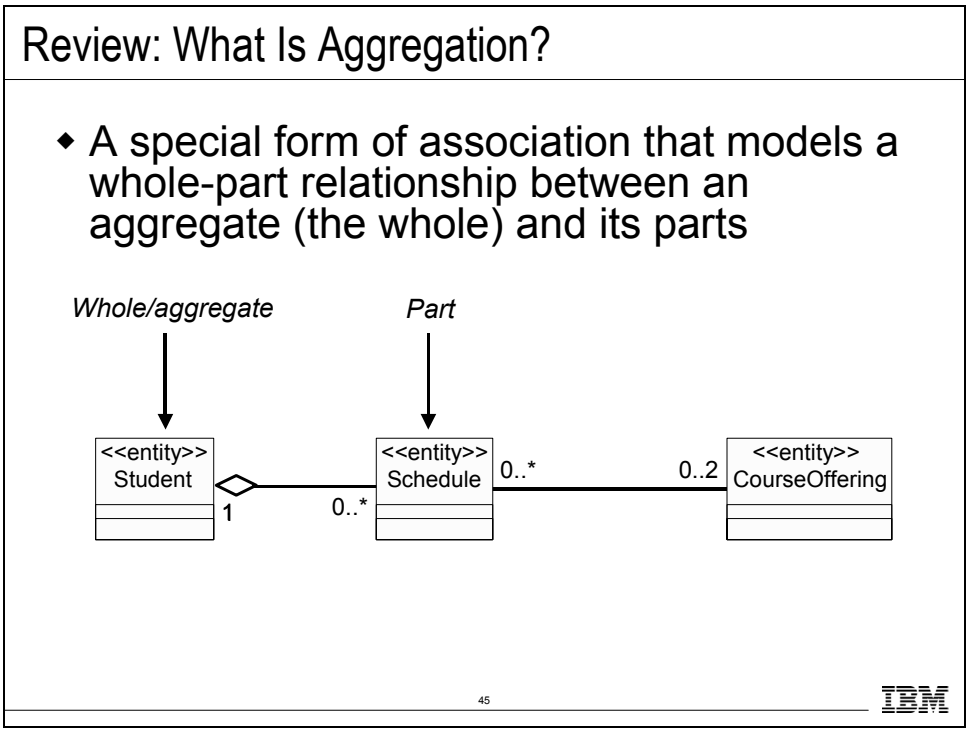
The navigability of the relationship should support the required message direction. In the above example, if navigability was not defined from the Client to the Supplier, then the PerformResponsibility message could not be sent from the Client to the Supplier.

Focus only on associations needed to realize the use cases; do not add associations you think "might" exist unless they are required based on the Interaction diagrams.

Remember to give the associations role names and multiplicities. You can also specify navigability, although this will be refined in Class Design.

Write a brief description of the association to indicate how the association is used, or what relationships the association represents.

Review: What Is Aggregation?



Aggregation is a stronger form of association which is used to model a whole-part relationship between model elements. The whole/aggregate has an aggregation association to the its constituent parts. A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.

Since aggregation is a special form of association, the use of multiplicity, roles, navigation, and so forth is the same as for association.

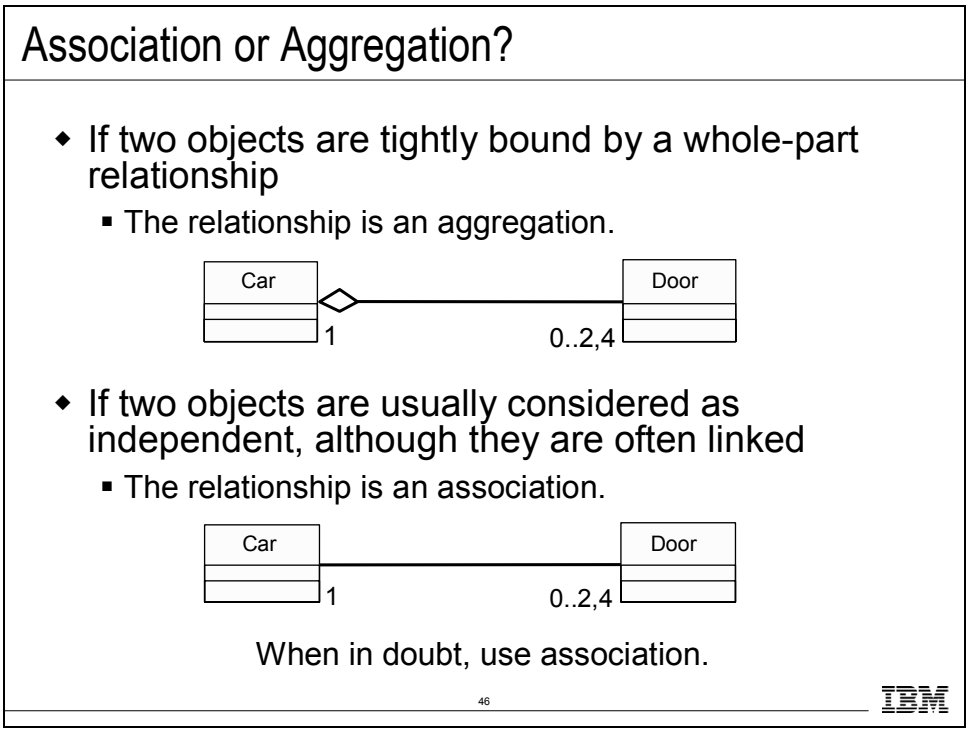
Sometimes a class may be aggregated with itself. This does not mean that an instance of that class is composed of itself (that would be silly). Instead, it means that one instance if the class is an aggregate composed of other instances of the same class.

Some situations where aggregation may be appropriate include:

- An object is physically composed of other objects (for example, car being physically composed of an engine and four wheels).
- An object is a logical collection of other objects (for example, a family is a collection of parents and children).
- An object physically contains other objects (for example, an airplane physically contains a pilot).

In the example on the slide, the relationship from Student to Schedule is modeled as an aggregation, because a Schedule is inherently tied to a particular Student. A Schedule outside of the context of a Student makes no sense in this Course Registration System. The relationship from Schedule to CourseOffering is an association because CourseOfferings may appear on multiple Schedules.

Association or Aggregation?

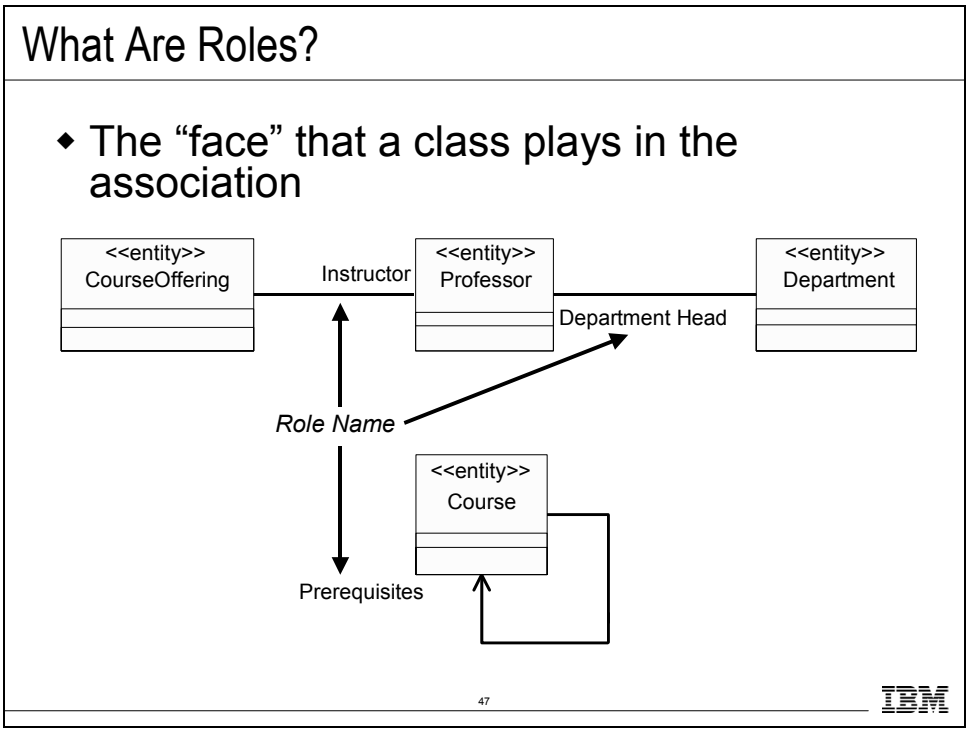


Aggregation should be used only where the "parts" are incomplete outside the context of the whole. If the classes can have independent identity outside the context provided by other classes, or if they are not parts of some greater whole, then the association relationship should be used.

When in doubt, an association is more appropriate. Aggregations are generally obvious. A good aggregate should be a natural, coherent part of the model. The meaning of aggregates should be simple to understand from the context. Choosing aggregation is only done to help clarify; it is not something that is crucial to the success of the modeling effort.

The use of aggregation versus association is dependent on the application you are developing. For example, if you are modeling a car dealership, the relationship between a car and the doors might be modeled as aggregation, because the car always comes with doors, and doors are never sold separately. However, if you are modeling a car parts store, you might model the relationship as an association, as the car (the body) might be independent of the doors.

What Are Roles?



Each end of an association has a role in relationship to the class on the other end of the association. The role specifies the face that a class presents on each side of the association. A role must have a name, and the role names on opposite sides of the association must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object.

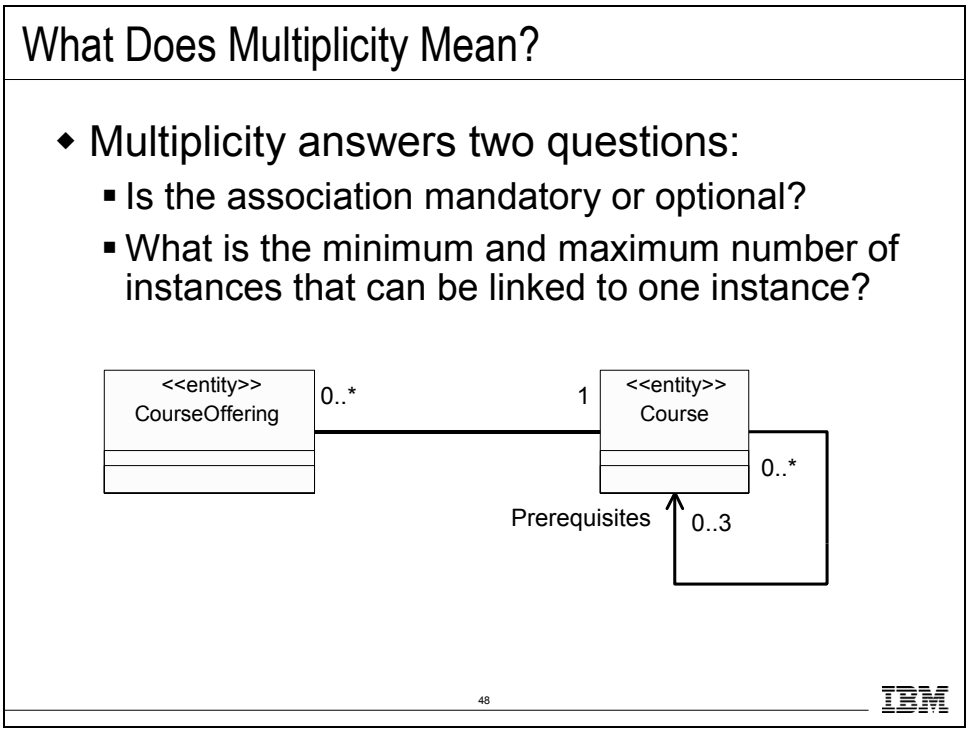
The use of association names and role names is mutually exclusive: one would not use both an association name and a role name. For each association, decide which conveys more information.

The role name is placed next to the end of the association line of the class it describes.

In the case of self-associations, role names are essential to distinguish the purpose for the association.

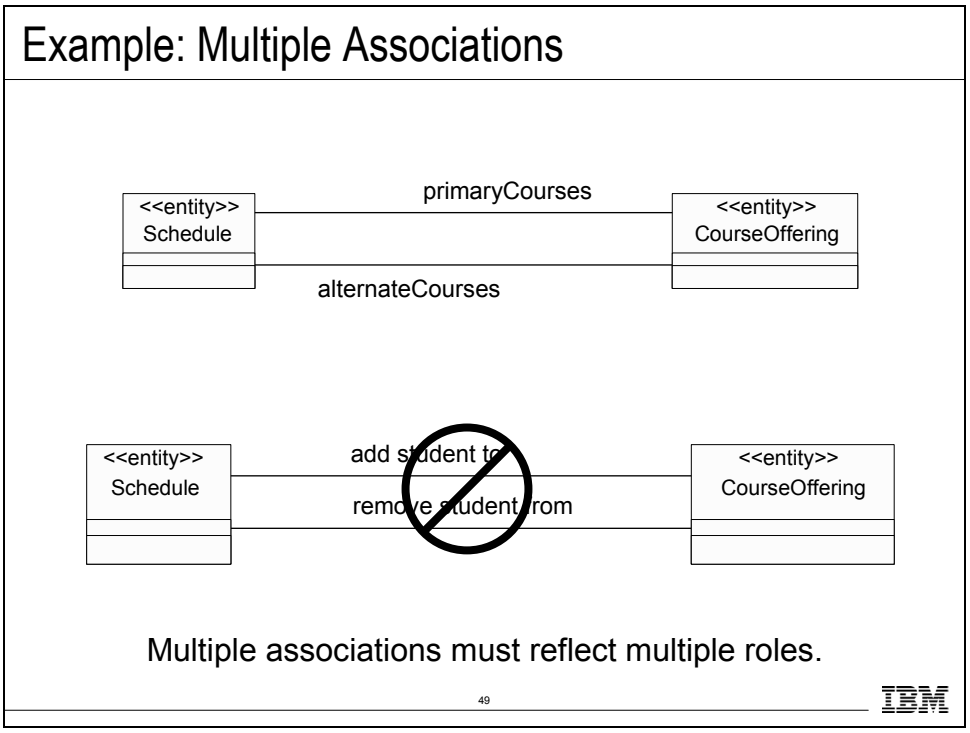
In the above example, the Professor participates in two separate association relationships, playing a different role in each.

What Does Multiplicity Mean?



- Multiplicity lets you know the lower and the upper bound number of relationships that a given object can have with another object. Many times you do not know what the maximum number of instances may be, and you will use the "*" to specify that this number is unknown.
- The most important question that multiplicity answers: Is the association mandatory? A lower bound number that is greater than zero indicates that the relationship is mandatory.
- This example indicates that a course object can be related to zero or more course offerings. You can tell that the relationship is optional because the lower bound number is zero. The upper bound number of the relationship is unknown, as indicated by the "*". If you read the association the other way, you will see that a given course offering object can be related to only one course. This relationship is mandatory and indicates that it is not possible for a course offering object to exist without an associated course object.

Example: Multiple Associations



There can be multiple associations between the same two classes, but they should represent distinct relationships, and DIFFERENT ROLES; they should not be just for invoking different operations.

If there is more than one association between two classes then they MUST be named.

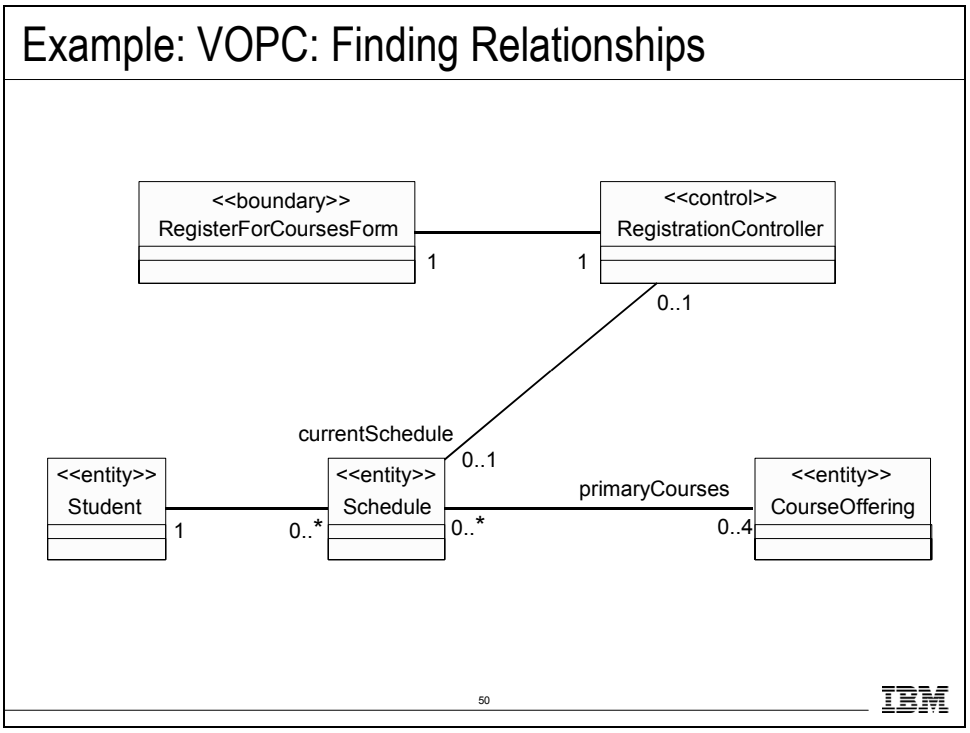
It is unusual to find more than one association between the same two classes. Occurrences of multiple associations should be carefully examined.

To determine if multiple associations are appropriate, look at instances of the classes. ClassA and ClassB have two associations between them, Assoc1 and Assoc2. If an instance of ClassA has a link with two SEPARATE instances of ClassB, then multiple associations are valid.

In the above example, the top diagram is an appropriate use of multiple associations, but the bottom diagram is not. In the valid case, two associations are required between Schedule and CourseOffering, as a Schedule can contain two kind of CourseOfferings, primary and alternate. These must be distinguishable, so two separate associations are used. In the invalid case, the two relationships represent two operations of CourseOffering, not two roles of CourseOffering.

Remember, Students and CourseOfferings are related via the Schedule class. Students are enrolled in a CourseOffering if there is a relationship between the Student's Schedule and the CourseOffering.

Example: VOPC: Finding Relationships



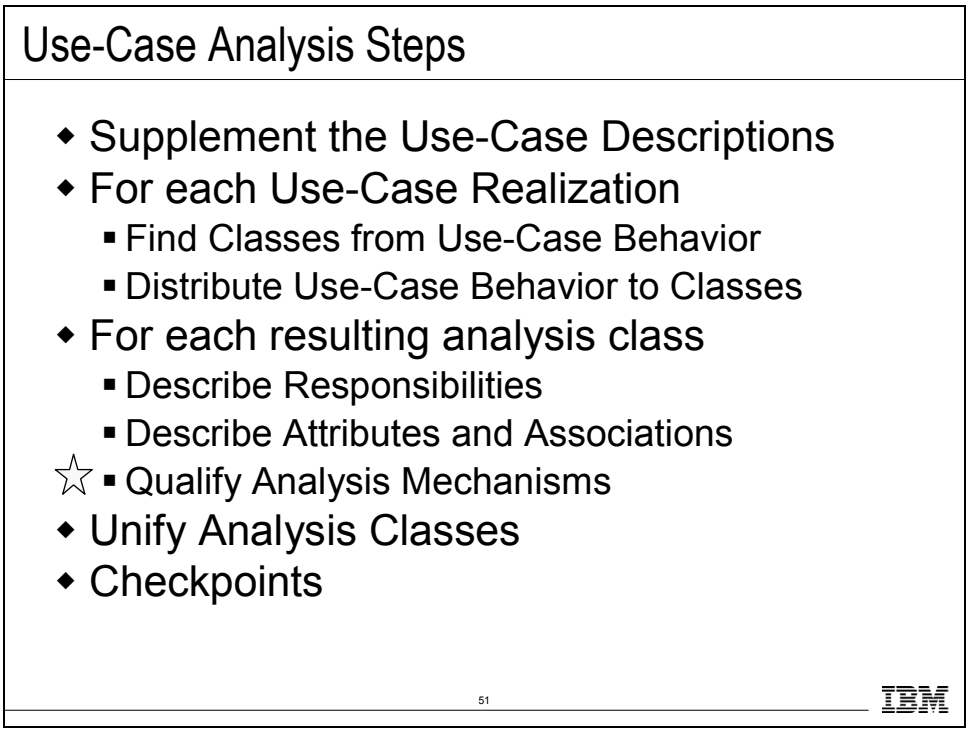
The diagram on this slide shows classes that collaborate to perform the "Register for Courses" use case, along with their relationships. As noted earlier, this type of diagram is called a View of Participating Classes (VOPC) diagram.

The relationships in this diagram are defined using the Interaction diagrams for the Register for Courses use case provided earlier in this module.

Rationale for relationships:

- From RegisterForCoursesForm to RegistrationController: There is one controller for each Schedule being created (for example, each Student registration session).
- From RegistrationController to Schedule. A RegistrationController deals with one Schedule at a time (the current Schedule for the Student registering for courses). Note the use of the "currentSchedule" role name. **Note**: Many RegisterForCoursesForms can be active at one time (for different sessions/students), each with their own RegistrationController.
- From Schedule to CourseOffering: Each Schedule may have up to four primary Course Offerings and up to two alternate Course Offerings. A particular Course Offering may appear on many Schedules as either a primary or an alternate.
- From Student to Schedule: A Student may have many schedules or none. A Schedule is only associated with a single Student and does not exist without a Student to be associated to.

Use-Case Analysis Steps



At this point, we have a pretty good understanding of the analysis classes, their responsibilities, and the collaborations required to support the functionality described in the use cases.

Now we must look into how each of the defined analysis classes implements the analysis mechanisms identified in Architectural Analysis.

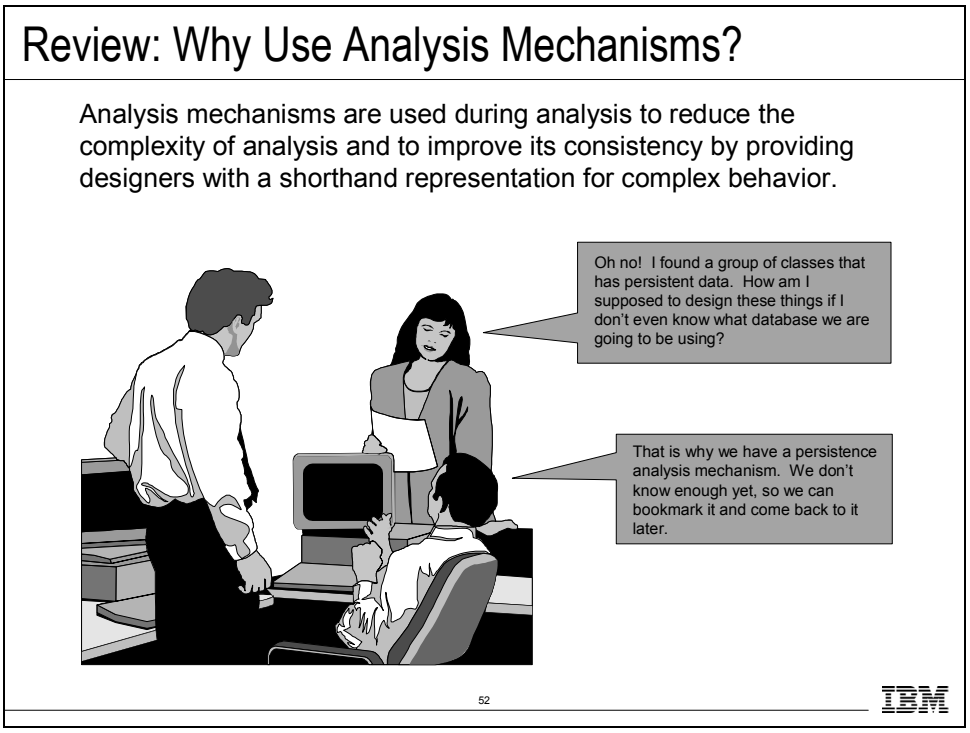
The purpose of the Qualify Analysis Mechanisms step is to:

- Identify analysis mechanisms (if any) used by the class.
- Provide additional information about how the class applies the analysis mechanism.

For each such mechanism, qualify as many characteristics as possible, giving ranges where appropriate, or when there is still much uncertainty.

Different architectural mechanisms will have different characteristics, so this information is purely descriptive and need only be as structured as necessary to capture and convey the information. During Analysis, this information is generally quite speculative, but the capturing activity has value, since conjectural estimates can be revised as more information is uncovered. The analysis mechanism characteristics should be documented with the class.

Review: Why Use Analysis Mechanisms?



An analysis mechanism represents a pattern that constitutes a common solution to a common problem. These mechanisms may show patterns of structure, patterns of behavior, or both. They are used during Analysis to reduce the complexity of Analysis, and to improve its consistency by providing designers with a shorthand representation for complex behavior. Analysis mechanisms are primarily used as "placeholders" for complex technology in the middle and lower layers of the architecture. By using mechanisms as "placeholders" in the architecture, the architecting effort is less likely to become distracted by the details of mechanism behavior.

Mechanisms allow the Analysis effort to focus on translating the functional requirements into software concepts without bogging down in the specification of relatively complex behavior needed to support the functionality but which is not central to it. Analysis mechanisms often result from the instantiation of one or more architectural or analysis patterns**.** Persistence provides an example of analysis mechanisms. A persistent object is one that logically exists beyond the scope of the program that created it. During analysis, we do not want to be distracted by the details of how we are going to achieve persistence.This gives rise to a "persistence" analysis mechanism that allows us to speak of persistent objects and capture the requirements we will have on the persistence mechanism without worrying about what exactly the persistence mechanism will do or how it will work.

Analysis mechanisms are typically, but not necessarily, unrelated to the problem domain, but instead are "computer science" concepts. As a result, they typically occupy the middle and lower layers of the architecture. They provide specific behaviors to a domain-related class or component, or correspond to the implementation of cooperation between classes and/or components.

Describing Analysis Mechanisms

Describing Analysis Mechanisms

- Collect all analysis mechanisms in a list
- Draw a map of the client classes to the analysis mechanisms
- Identify characteristics of the analysis mechanisms

53

IBM

In Architectural Analysis, the possible analysis mechanisms were identified and defined.

From that point on, as classes are defined, the required analysis mechanisms and analysis mechanism characteristics should be identified and documented. Not all classes will have mechanisms associated with them. Also, it is not uncommon for a client class to require the services of several mechanisms.

A mechanism has characteristics, and a client class uses a mechanism by qualifying these characteristics. This is to discriminate across a range of potential designs. These characteristics are part functionality, and part size and performance.

Example: Describing Analysis Mechanisms

Example: Describing Analysis Mechanisms

- Analysis class to analysis mechanism map

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

54

IBM

As analysis classes are identified, it is important to identify the analysis mechanisms that apply to the identified classes.

The classes that must be persistent are mapped to the persistency mechanism.

The classes that are maintained within the legacy Course Catalog system are mapped to the legacy interface mechanism.

The classes for which access must be controlled (that is, control who is allowed to read and modify instances of the class) are mapped to the security mechanism. Note: The legacy interface classes do not require additional security as they are read-only and are considered readable by all.

The classes that are seen to be distributed are mapped to the distribution mechanism. The distribution identified during analysis is that which is specified/implied by the user in the initial requirements. Distribution will be discussed in detail in the Describe Distribution module. For now, just take it as an architectural given that all control classes are distributed for the OOAD course example and exercise.

Example: Describing Analysis Mechanisms (continued)

Example: Describing Analysis Mechanisms (continued)

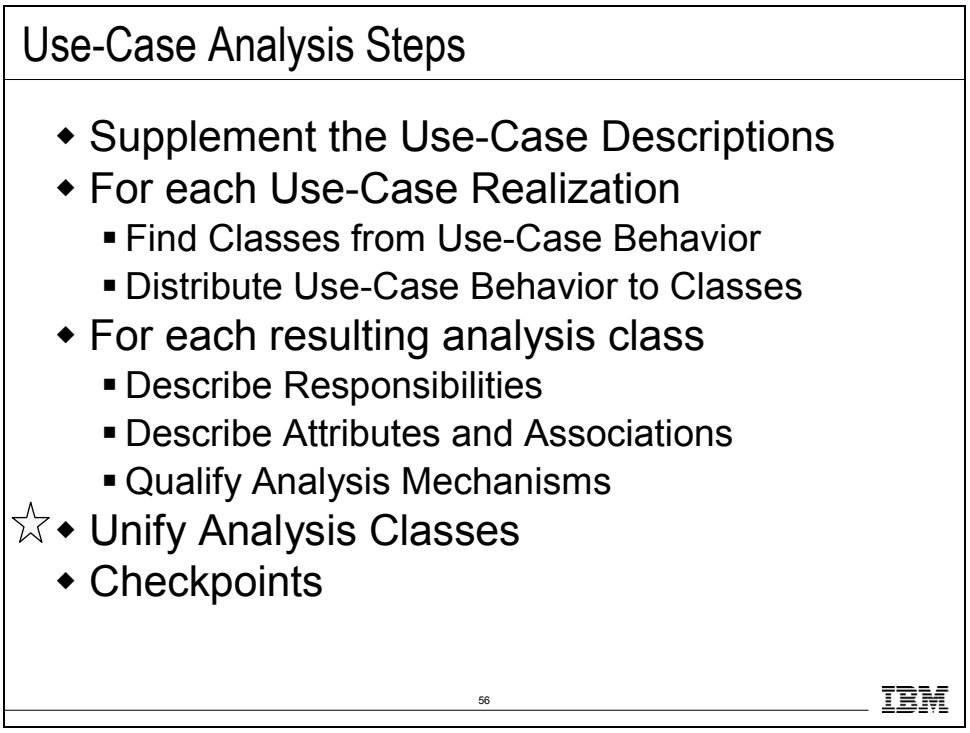
- Analysis mechanism characteristics
- Persistency for Schedule class:
  - Granularity: 1 to 10 Kbytes per product
  - Volume: up to 2,000 schedules
  - Access frequency
    - Create: 500 per day
    - Read: 2,000 access per hour
    - Update: 1,000 per day
    - Delete: 50 per day
  - Other characteristics

55

IBM

The above is just an example of how the characteristics for an analysis mechanism would be documented for a class. For scoping reasons, the analysis mechanisms and their characteristics are not provided for all of the analysis classes.

Use-Case Analysis Steps

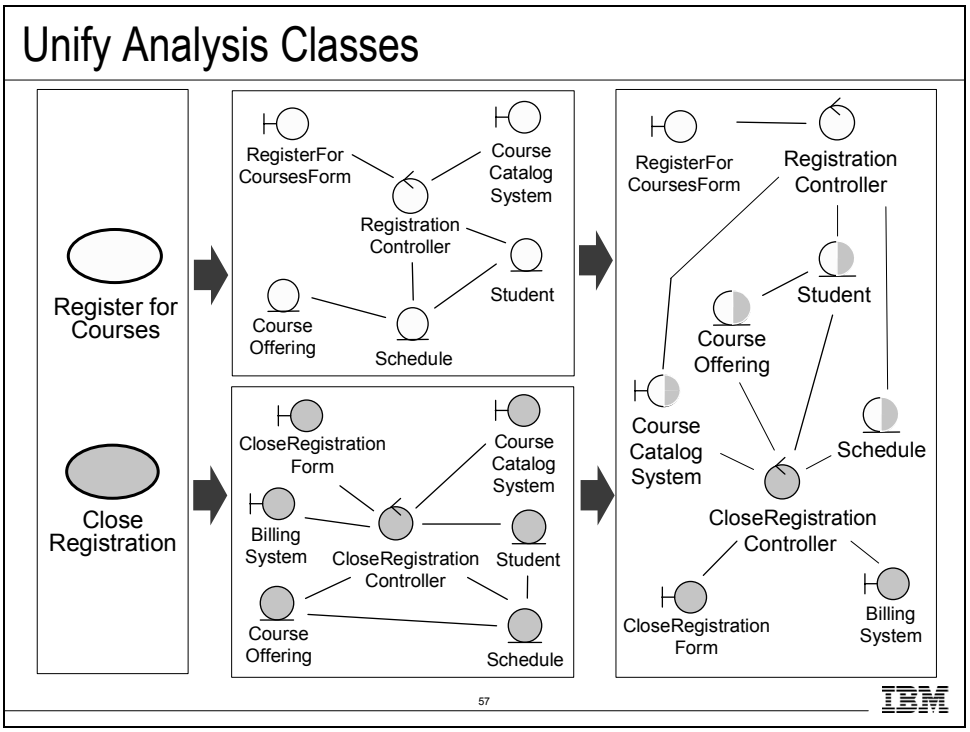


At this point, we have a pretty good understanding of the analysis classes, their responsibilities, the analysis mechanisms they need to implement, and the collaborations required to support the functionality described in the use cases.

Now we must review our work and make sure that it is as complete and as consistent as possible before moving on to the architecture activities.

The purpose of Unify Analysis Classes is to ensure that each analysis class represents a single well-defined concept, with non-overlapping responsibilities.

Unify Analysis Classes



Before the Design work can be done, the analysis classes need to be filtered to ensure that a minimum number of new concepts have been created.

Different use cases will contribute to the same classes. In the example above, the classes CourseCatalogSystem, CourseOffering, Schedule and Student participate in both the Register for Courses and Close Registration use cases.

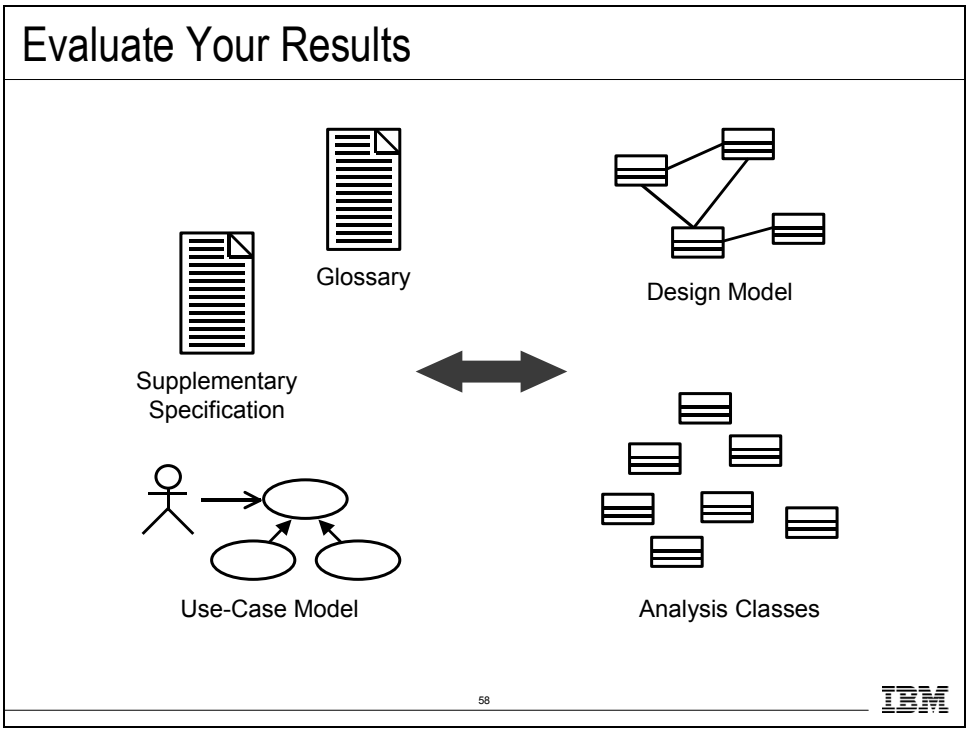
A class can participate in any number of use cases. It is therefore important to examine each class for consistency across the whole system.

Merge classes that define similar behaviors or that represent the same phenomenon.

Merge entity classes that define the same attributes, even if their defined behavior is different; aggregate the behaviors of the merged classes.

When you update a class, you should update any "supplemental" use-case descriptions (described earlier in this module), where necessary. Sometimes an update to the original Requirements (that is, use cases) may be necessary, but this should be controlled, as the Requirements are the contract with the user/customer, and any changes must be verified and controlled.

Evaluate Your Results



We now have a pretty good feeling about our Analysis Model. Now it is time to review our work for completeness and consistency.

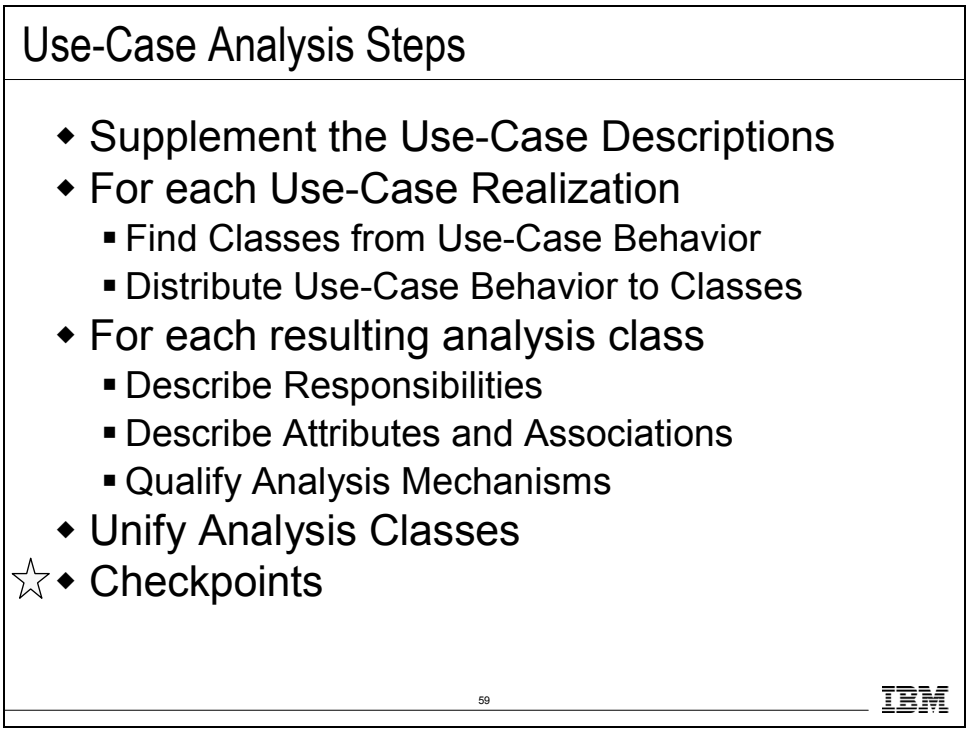
Be sure to:

- Verify that the analysis classes meet the functional requirements made on the system.
- Verify that the analysis classes and their relationships are consistent with the collaborations they support.

It is very important that you evaluate your results at the conclusion of the **Use-Case Analysis**.

The number of reviews, the formality of the reviews, and when they are performed will vary, depending on the process defined for the project.

Use-Case Analysis Steps



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

In this module, we will concentrate on those checkpoints that the designer is most concerned with (that is, looks for).

Checkpoints: Analysis Classes

Checkpoints: Analysis Classes

- Are the classes reasonable?
- Does the name of each class clearly reflect the role it plays?
- Does the class represent a single well-defined abstraction?
- Are all attributes and responsibilities functionally coupled?
- Does the class offer the required behavior?
- Are all specific requirements on the class addressed?

60

IBM

The above checkpoints for the analysis classes might be useful.

Note: All checkpoints should be evaluated with regards to the use cases being developed for the current iteration.

The class should represent a single well-defined abstraction. If not, consider splitting it.

The class should not define any attributes or responsibilities that are not functionally coupled to the other attributes or responsibilities defined by that class.

The classes should offer the behavior the Use-Case Realizations and other classes require.

The class should address all specific requirements on the class from the requirement specification.

Remove any attributes and relationships if they are redundant or are not needed by the Use-Case Realizations.

Checkpoints: Use-Case Realizations

Checkpoints: Use-Case Realizations

- Have all the main and/or sub-flows been handled, including exceptional cases?
- Have all the required objects been found?
- Has all behavior been unambiguously distributed to the participating objects?
- Has behavior been distributed to the right objects?
- Where there are several Interaction diagrams, are their relationships clear and consistent?

61

IBM

The above checkpoints for the Use-Case Realizations might be useful.

Note: All checkpoints should be evaluated with regards to the use cases being developed for the current iteration.

The objects participating in a Use-Case Realization should be able to perform all of the behavior of the use case.

If there are several Interaction diagrams for the Use-Case Realization, it is important that it is easy to understand which Interaction diagrams relate to which flow of events. Make sure that it is clear from the flow of events description how the diagrams are related to each other.

Review

Review: Use-Case Analysis

- What is the purpose of Use-Case Analysis?
- What is a Use-Case Realization?
- What is an analysis class? Name and describe the three analysis stereotypes.
- Describe some considerations when allocating responsibilities to analysis classes.
- What two questions does multiplicit answer?

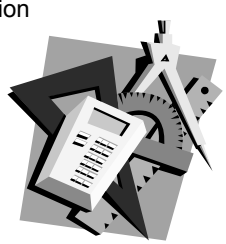
62

IBM

Exercise: Use-Case Analysis

Exercise: Use-Case Analysis

- Given the following:
  - Use-Case Model, especially the use-case flows of events
    - Exercise Workbook: Payroll Requirements, Use-Case Model section
  - Key abstractions/classes
    - Payroll Exercise Solution, Architectural Analysis section
  - The Supplementary Specification
    - Exercise Workbook: Payroll Requirements, Supplementary Specification section
  - The possible analysis mechanisms
    - Exercise Workbook: Payroll Architecture Handbook, Architectural Mechanisms, Analysis Mechanisms section



63

IBM

The goal of this exercise is to identify classes that must collaborate to perform a use case, allocate the use-case responsibilities to those classes, and diagram the collaborations.

Good sources for the analysis classes are the Glossary and any analysis classes defined during Architectural Analysis.

References to the givens:

- Use-Case Model: Exercise Workbook - Payroll Requirements, Use-Case Model section.
- Key abstractions: Payroll Exercise Solution, Architectural Analysis section.
- Supplementary Specification: Exercise Workbook - Payroll Requirements, Supplementary Specification section.
- The analysis mechanisms we are concentrating on in this course include: persistency, distribution, security, and legacy interface). See the Exercise Workbook: *Payroll Architecture Handbook*, Architectural Mechanisms, Analysis Mechanisms section for more information on these analysis mechanisms.

Exercise: Use-Case Analysis (continued)

Exercise: Use-Case Analysis (continued)

- Identify the following for a particular use case:
  - The analysis classes, along with their:
    - Brief descriptions
    - Stereotypes
    - Responsibilities
  - The collaborations needed to implement the use case
  - Analysis class attributes and relationships

64

IBM

When identifying analysis classes from the use-case flows of events, use the analysis stereotypes for guidance (boundary, control, and entity).

Be sure to define the identified classes. These definitions become very important as you start to allocate responsibilities to those classes.

The relationships to be identified are those needed to support the collaborations modeled in the use-case Interaction diagrams. The attributes to be identified are the "obvious" properties of the identified classes. More attributes may be defined during later Class Design.

For each identified analysis class, determine if any of the analysis mechanisms apply. To make this decision, the Supplementary Specification may be needed.

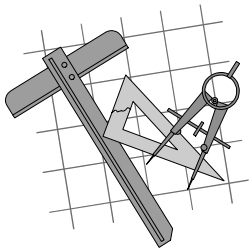
Refer to the following slides if needed:

- Example: Finding Boundary Classes – p. 6-16
- Example: Finding Entity Classes – p. 6-20
- Example: Finding Control Classes – p. 6-24
- Describe Responsibilities – p. 6-37
- Finding Attributes – p. 6-42
- Finding Relationships – p. 6-44
- Example: Describing Analysis Mechanisms – p. 6-55

Exercise: Use-Case Analysis (continued)

Exercise: Use-Case Analysis (continued)

- Produce the following for a particular use case:
  - Use-Case Realization Interaction diagram for at least one of the use-case flows of events
  - VOPC class diagram, containing the analysis classes, their stereotypes, responsibilities, attributes, and relationships
  - Analysis class to analysis mechanism map



65

IBM

Start with diagramming the basic flow and then do the other sub-flows if you have time.

The Interaction diagrams may be Communication or Sequence diagrams. On an Interaction diagram, sending a message to an object means that you are allocating responsibility for performing that task to the object.

Be sure to use the "//" naming convention for responsibilities.

Refer to the following slides if needed:

- Review: What is a Use-Case Realization? – p. 6-10
- Example: Sequence Diagram – p. 6-31
- Example: Communication diagram – p. 6-33
- Example: VOPC Finding Relationship – p. 6-50

Exercise: Review

Exercise: Review

- Compare your Use-Case Realization with the rest of the class
  - Do the Interaction diagrams carry out the use-case flow of events?
  - Are the stereotypes behaving properly?
  - Is each association supported by a link?
  - Does each association have multiplicity assigned?
  - Have role names been assigned? Do they accurately represent the face the class plays in the relationship?

66

IBM

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

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