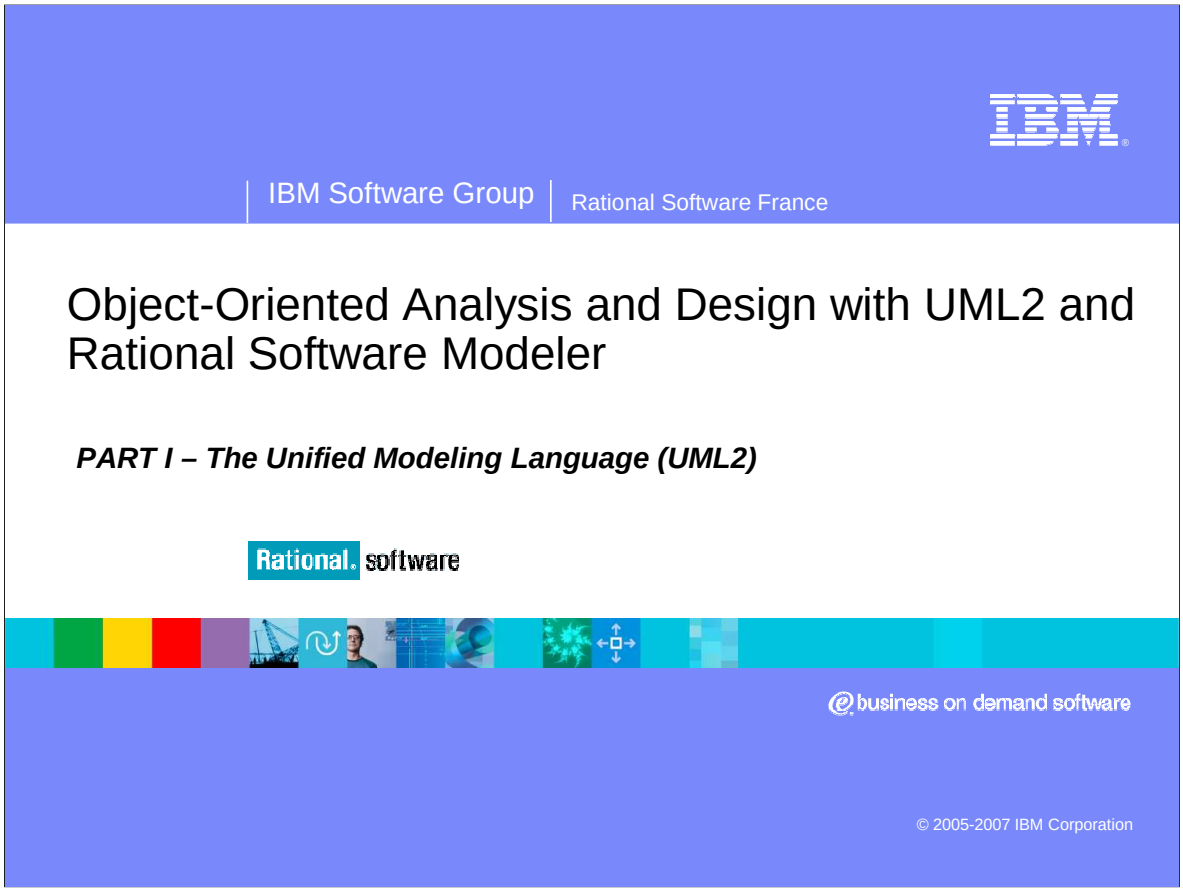
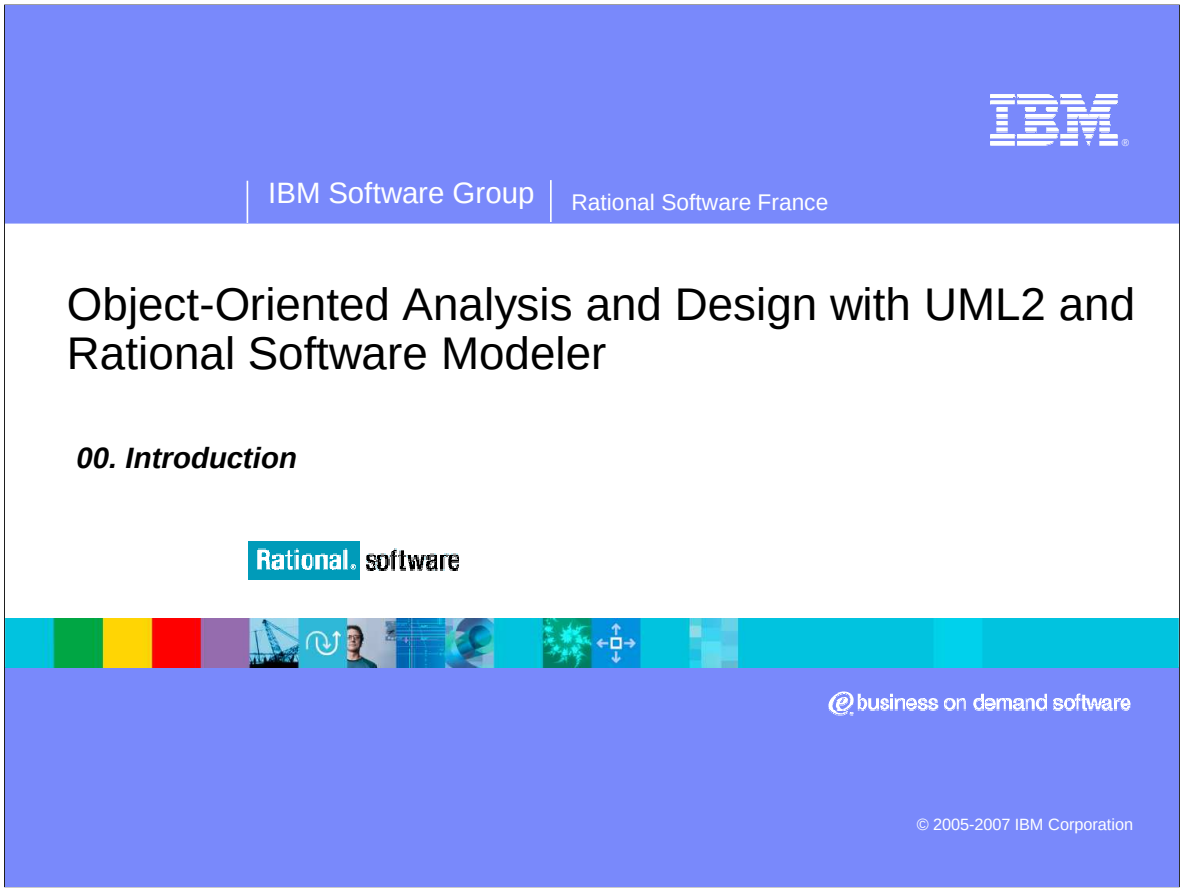


Table of Contents



Introductions

- Your organization
- Your role
- Your background, experience
  - Object technology experience
  - Software development experience
- Your expectations for this course
- One interesting fact about you!

Course Objectives

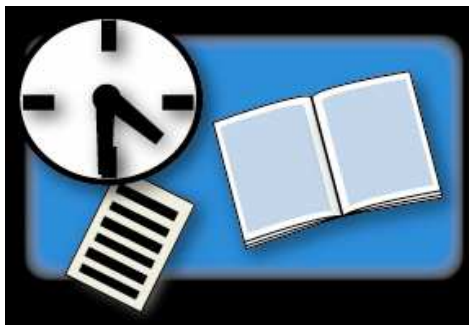
- Understand the concepts of Object-Orientation
- Capture system requirements with use cases
- Identify classes, objects and relations, and creating interaction diagrams and class diagrams
- Build an analysis model with analysis classes
- Understand Model-Driven Development (MDD) and Model-Driven Architecture (MDA)
- Apply a use-case driven, architecture-centered and iterative process to build a robust design model

Agenda

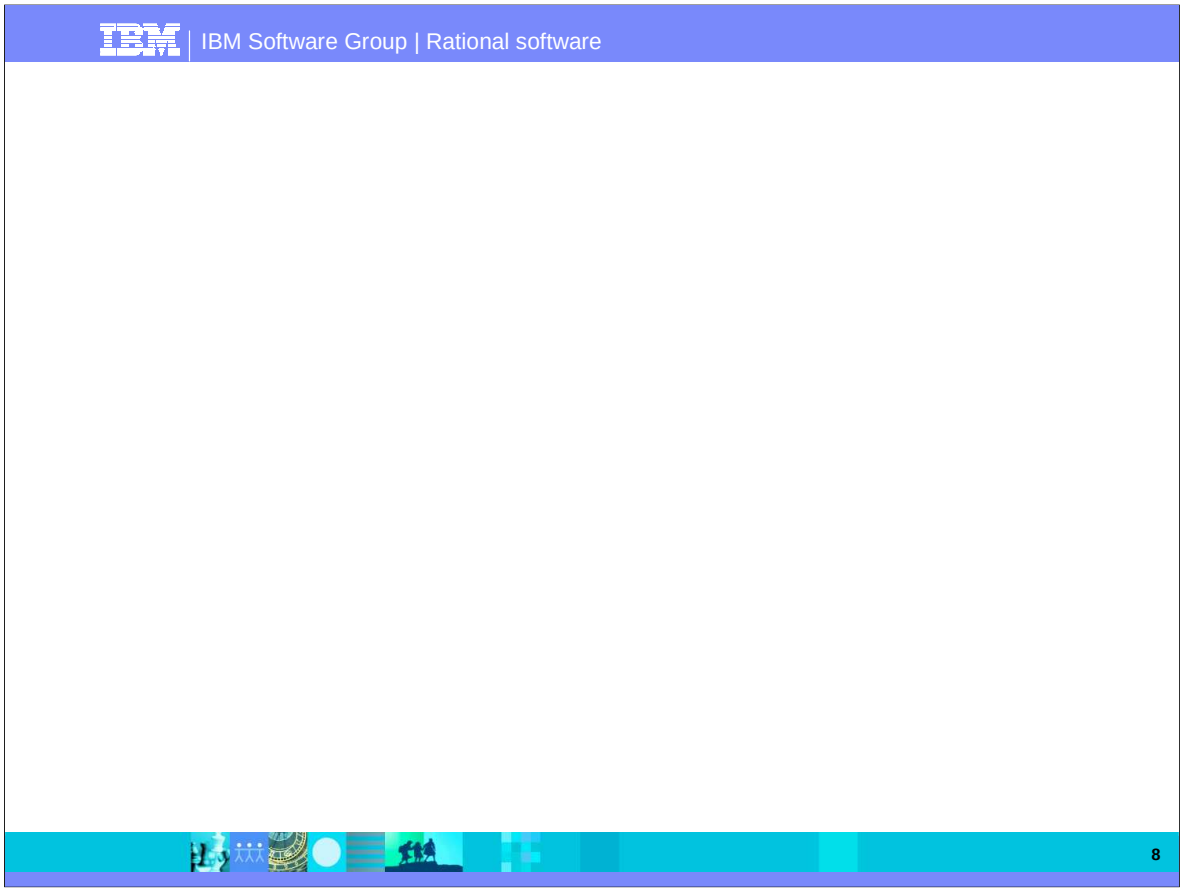
- Day 1:
  - UML 2
- Day 2:
  - Object-Oriented Analysis (OOA)
- Day 3:
  - Object-Oriented Analysis (OOA) (cont.)
  - Object-Oriented Design (OOD)
- Day 4:
  - Object-Oriented Design (OOD) (cont.)

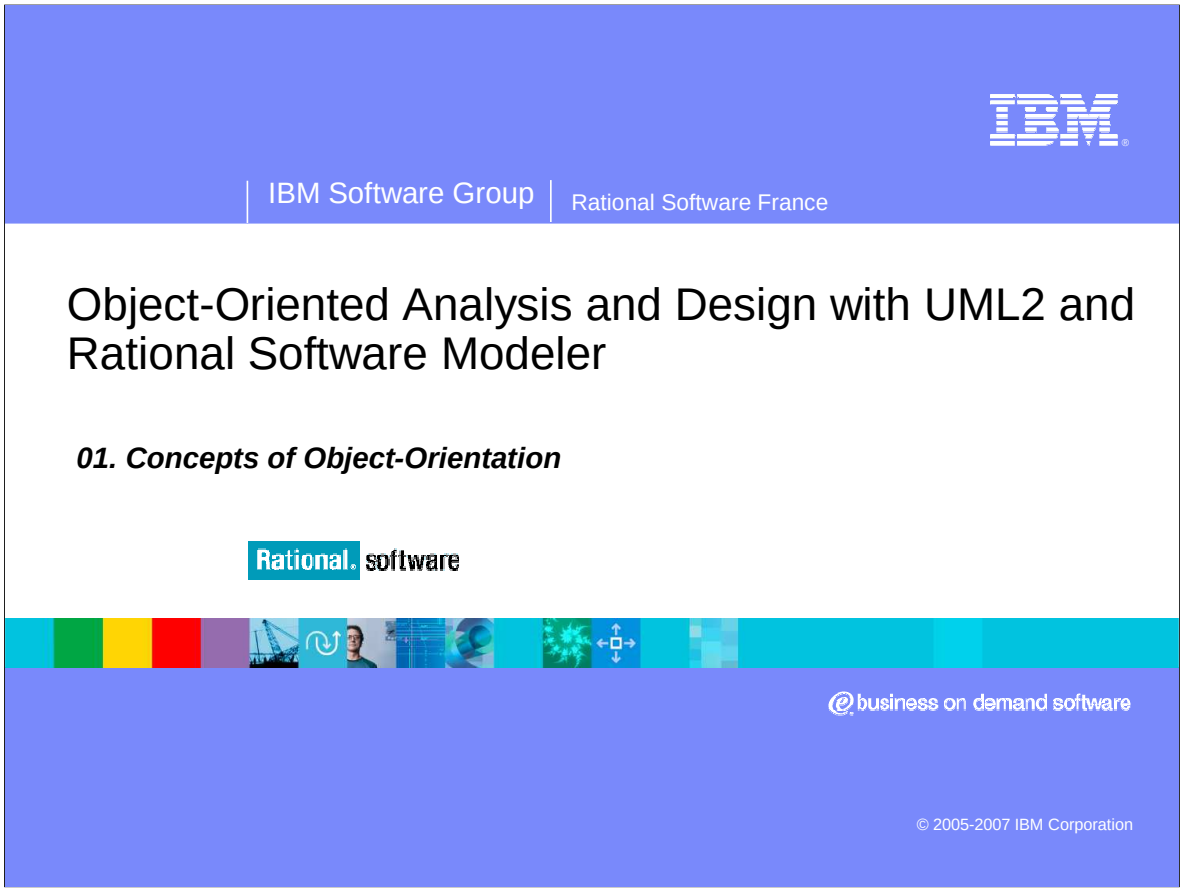
- The labs for days 2 to 4 will be based on IBM Rational Software Modeler 7 or above

Logistics



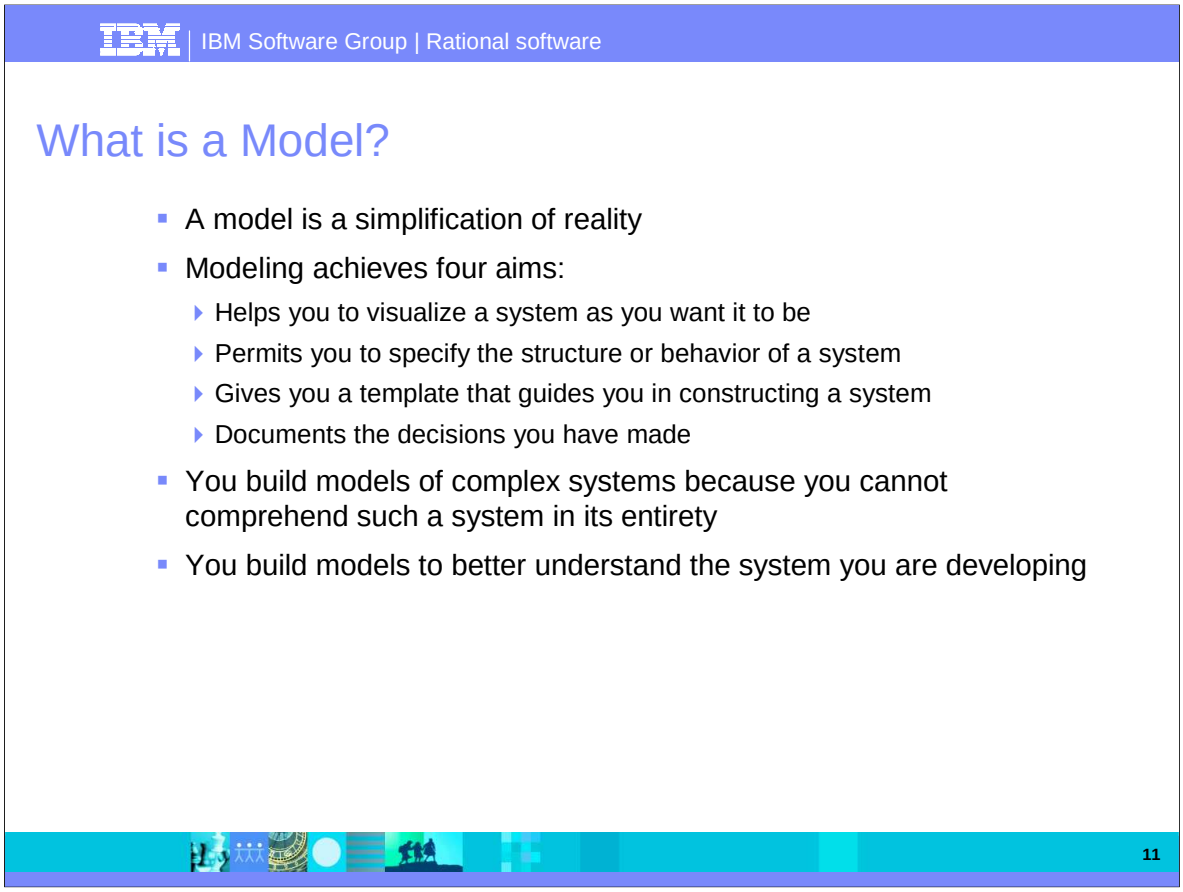
Morning
1 Fifteen minute break
Lunch
1 Hour
Afternoon
1 Fifteen minute break





Where Are We?

- What is Modeling?
  - Principles of Object-Orientation
  - The Unified Modeling Language



Software teams often do not model.

Many software teams build applications approaching the problem like they were building paper airplanes :

- Start coding from project requirements
- Work longer hours and create more code
- Lacks any planned architecture
- Doomed to failure

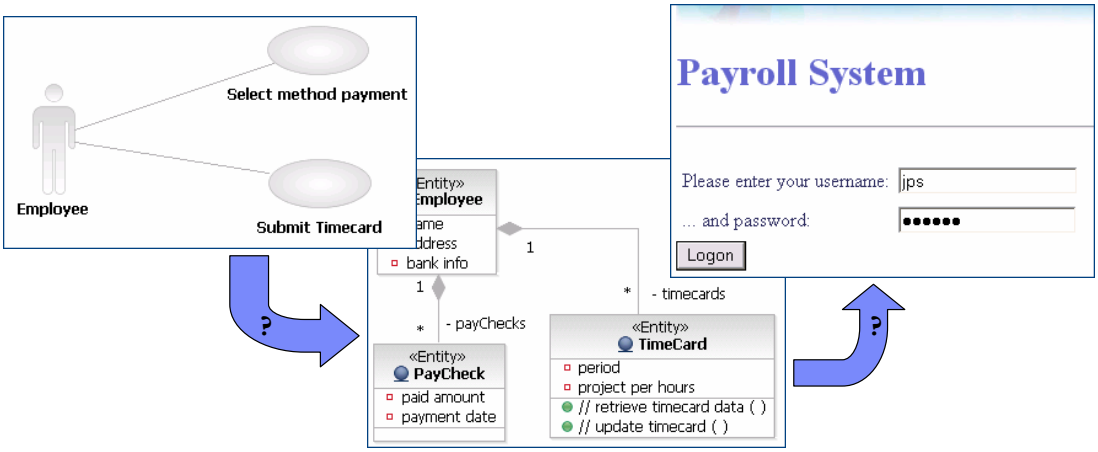
Modeling is a common thread to successful projects.

Some general facts about models:

- The model you create influences how the problem is attacked.
- Every model may be expressed at different levels of precision.
- The best models are connected to reality.
- No single model is sufficient.

Model-Driven Development

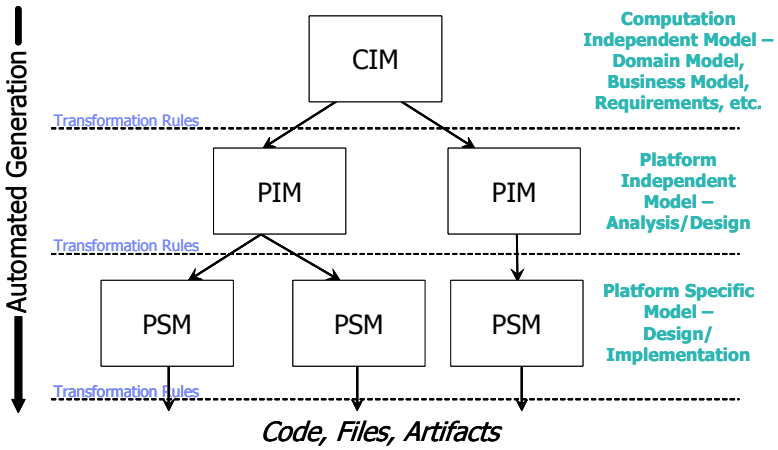
- A natural evolution of object-oriented technologies
- The encapsulation of business logic in (UML) models
- The use of these models to *automate* the development of applications, code generation, testing and maintenance



Model-Driven Architecture (MDA)

- An architectural style
- An OMG initiative

http://www.omg.com/



Where Are We?

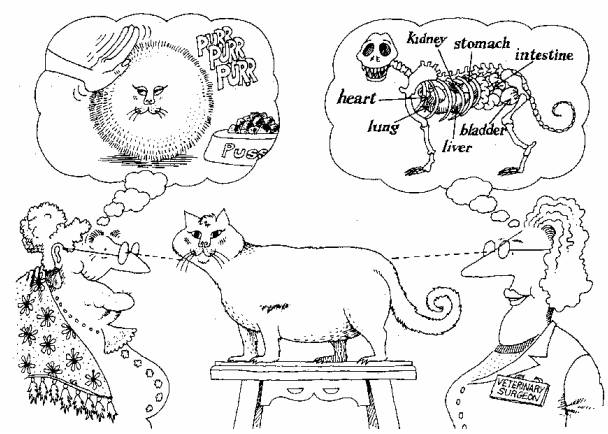
- What is Modeling?
- Principles of Object-Orientation
- The Unified Modeling Language

Basic Principles of Object Orientation

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

What Is Abstraction?

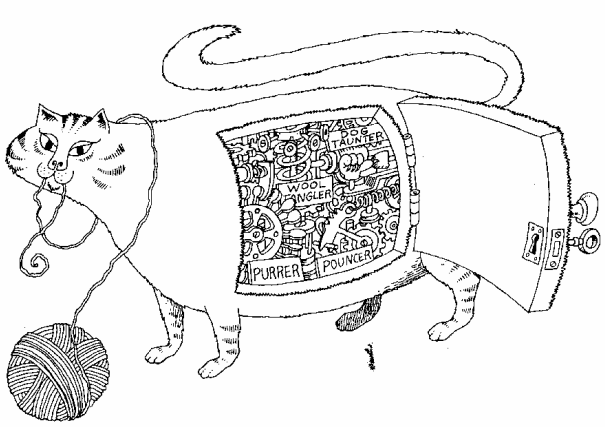
- The essential characteristics of an entity that distinguishes it from all other kinds of entities
- Depends on the perspective of the viewer
- Is not a concrete manifestation, denotes the ideal essence of something



From *Object-Oriented Analysis and Design with Applications* by Grady Booch, 1994

What Is Encapsulation?

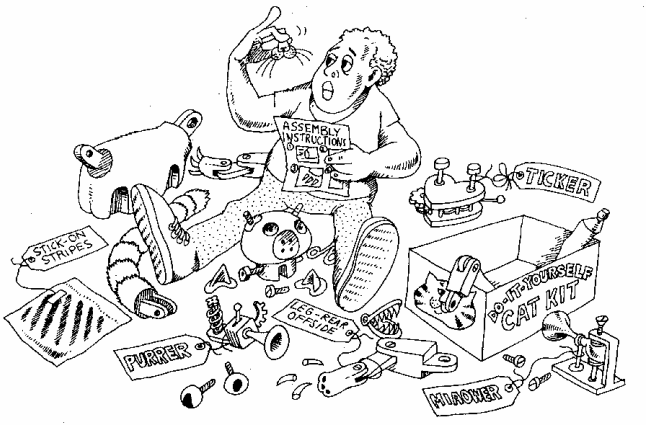
- Hides implementation from clients
  - Clients depend on interface
  - Improves the resiliency of the system, i.e. its ability to adapt to change



From *Object-Oriented Analysis and Design with Applications* by Grady Booch , 1994

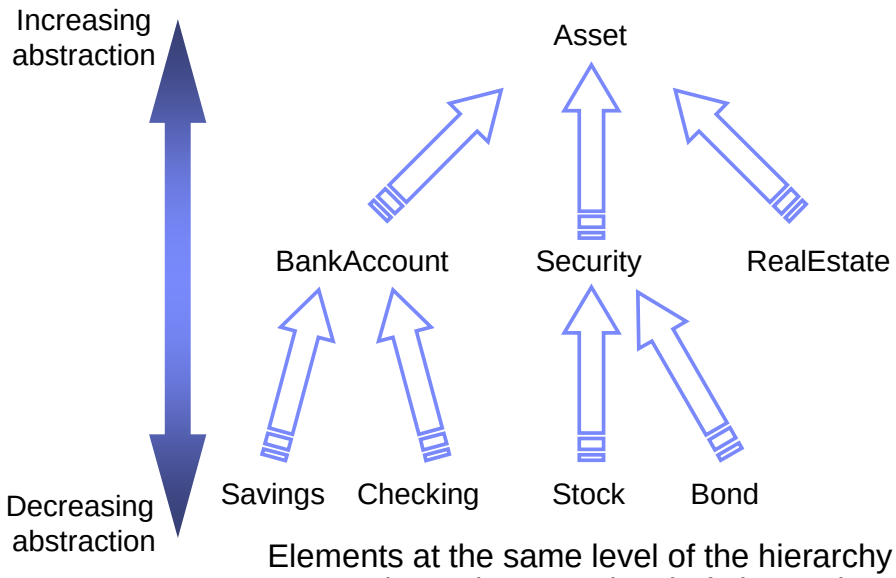
What Is Modularity?

- Breaks up something complex into manageable pieces.
- Helps people understand complex systems



From *Object-Oriented Analysis and Design with Applications* by Grady Booch , 1994

What Is Hierarchy?



Elements at the same level of the hierarchy should be at the same level of abstraction

Where Are We?

- What is Modeling?
- Principles of Object-Orientation
- ⇨ The Unified Modeling Language

What Is UML?

- UML is a language for:
  - Visualizing
  - Specifying
  - Constructing
  - Documenting

  the artifacts of a software-intensive system
- UML covers all aspects of software development:
  - System architecture
  - Requirements
  - Tests
  - Project planning
  - Etc.

History of UML

Current version = 2.1

UML 2.0 (2004)

UML 1.5 (March, '03)

UML 1.1 (Sept. '97)

UML 1.0 (Jan. '97)

UML 0.9 and UML 0.91 (June '96) (Oct. '96)

Unified Method 0.8 (OOPSLA '95)

OOSE

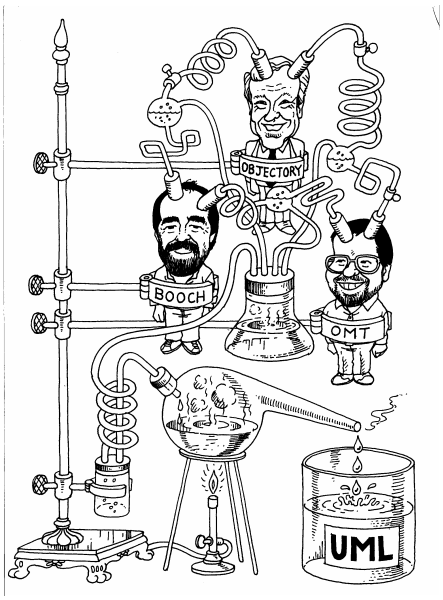
Other Methods

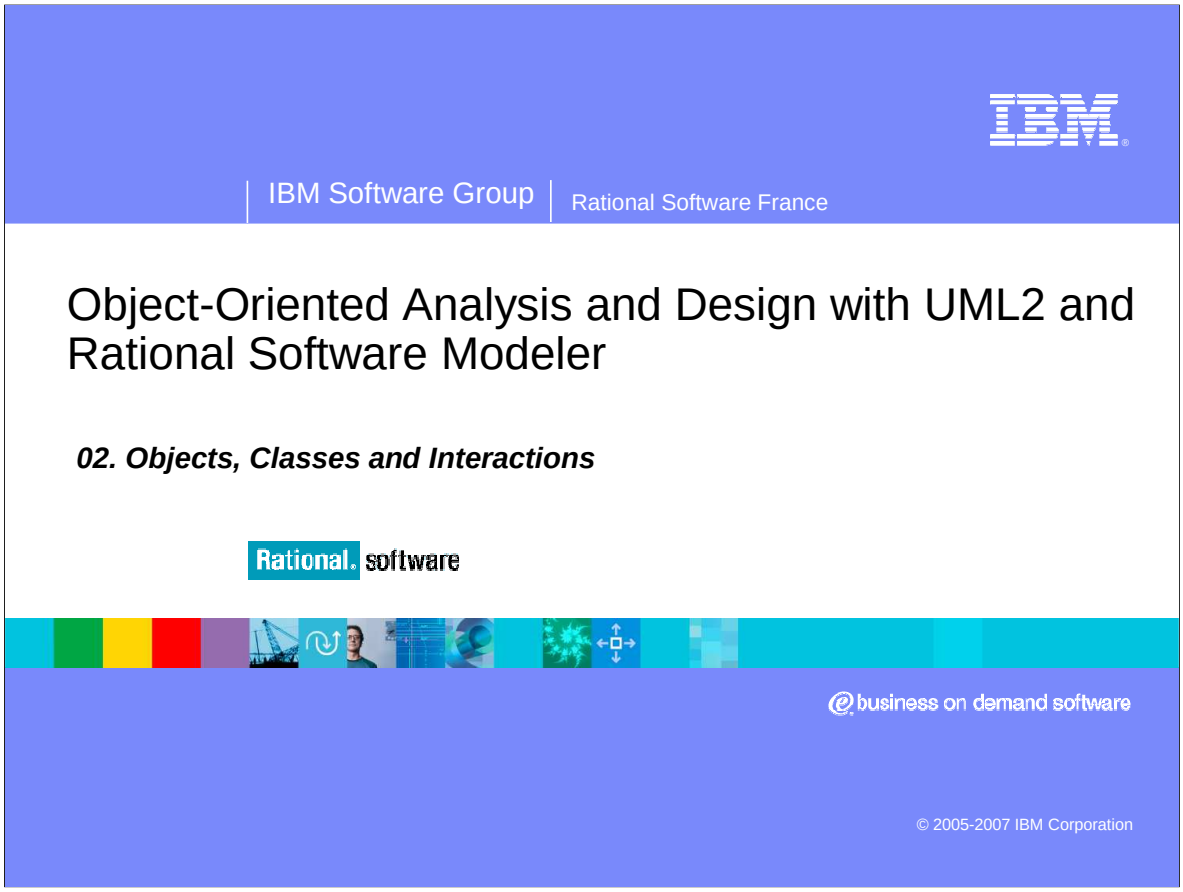
Booch '91

Booch '93

OMT - 1

OMT - 2



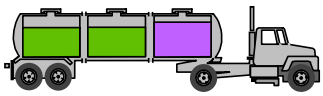


Where Are We?

- What is an Object?
  - What is a Class?
  - Interaction Diagrams
  - Sequence Diagrams
  - Communication Diagrams

What Is an Object?

- Informally, an object represents an entity, either physical, conceptual, or software
  - Physical entity



Truck

  - Conceptual entity

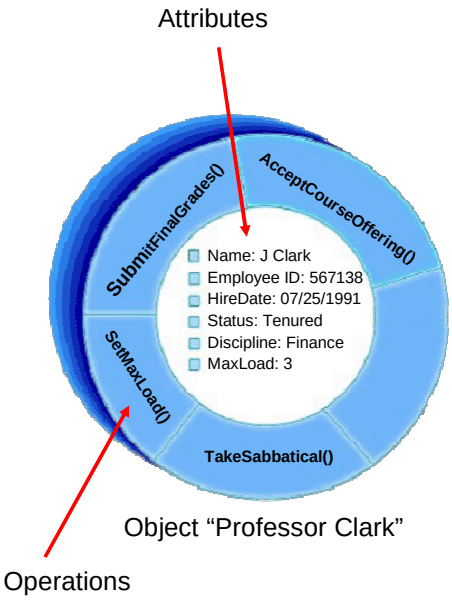
Chemical Process

  - Software entity

Linked List

A More Formal Definition

- An object is an entity with a well-defined boundary and *identity* that encapsulates *state* and *behavior*
  - State is represented by attributes and relationships
  - Behavior is represented by operations, methods, and state machines



The **state** of an object is one of the possible conditions in which an object may exist. State normally changes over time.

The state of an object is usually implemented by a set of properties called attributes, along with the values of the properties and the links the object may have with other objects.

State is not defined by a “state” attribute or set of attributes. Instead, state is defined by the total of an object’s attributes and links. For example, if Professor Clark’s status changed from Tenured to Retired, the state of the Professor Clark object would change.

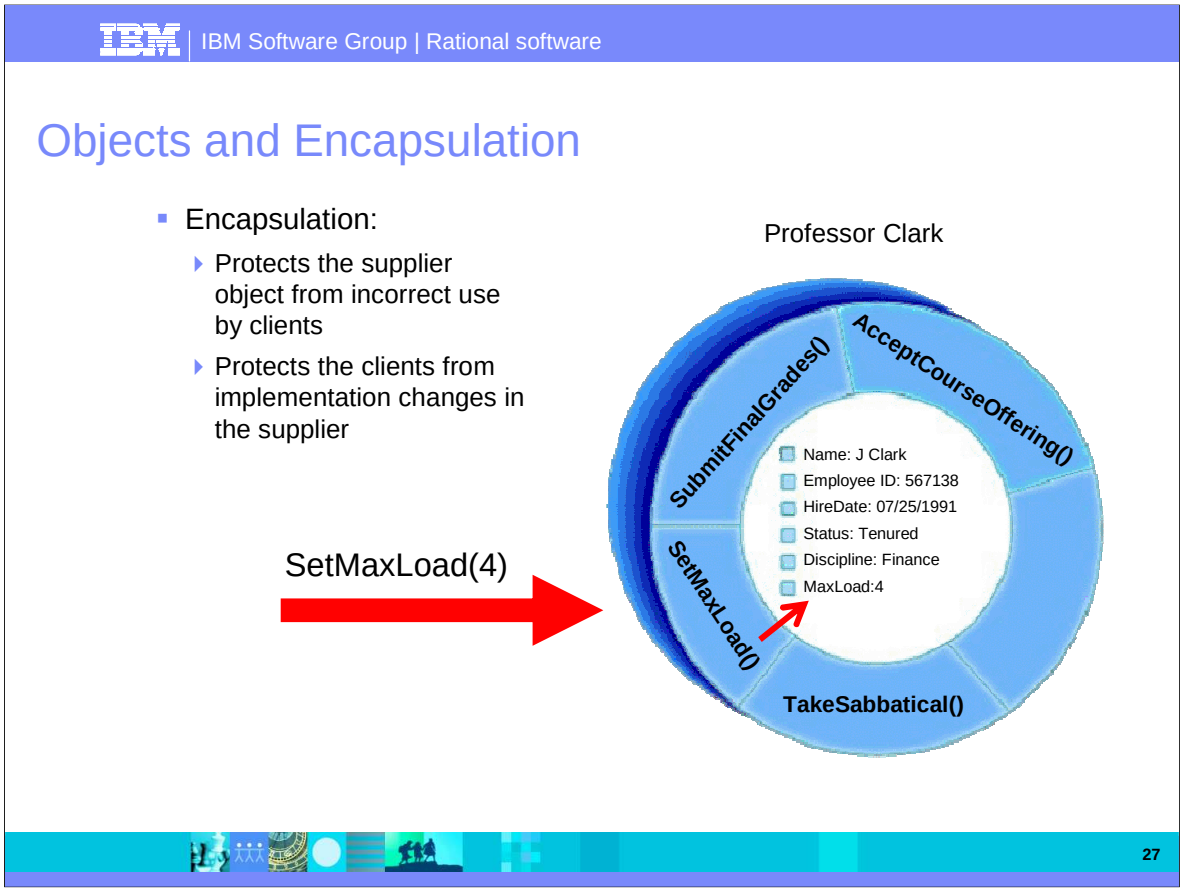
The second characteristic of an object is that it has **behavior**. Objects are intended to mirror the concepts that they are modeled after, including behavior.

Behavior determines how an object acts and reacts to requests from other objects.

Object behavior is represented by the operations that the object can perform. For example, Professor Clark can choose to take a sabbatical once every five years. The Professor Clark object represents this behavior through the TakeSabbatical() operation.

In the real world, two people can share the same characteristics: name, birth date, job description. Yet, there is no doubt that they are two individuals with a unique **identity**.

The same concept holds true for objects. Although two objects may share the same state (attributes and relationships), they are separate, independent objects with their own unique identity.



The key to encapsulation is an object's **interface**. The object interface ensures that all communication with the object takes place through a set of predefined operations. Data inside the object is only accessible by the object's operations. No other object can reach inside the object and change its attribute values.

For example, Professor Clark needs to have her maximum course load increased from three classes to four classes per semester. Another object makes a request to Professor Clark to set the maximum course load to four. The attribute, MaxLoad, is then changed by the SetMaxLoad() operation.

Encapsulation is beneficial in this example because the requesting object does not need to know how to change the maximum course load. In the future, the number or variables that are used to define the maximum course load may be increased, but it doesn't affect the requesting object. It depends on the operation interface for the Professor Clark object.

Where Are We?

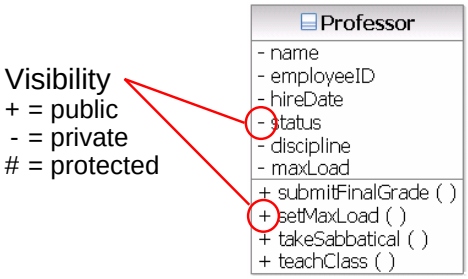
- What is an Object?
- What is a Class?
- Interaction Diagrams
- Sequence Diagrams
- Communication Diagrams

What Is a Class?

- A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and *semantics*
  - An object is an instance of a class
- A class is an abstraction in that it
  - Emphasizes relevant characteristics
  - Suppresses other characteristics

Representing Classes in UML

- A class is represented using a rectangle often with three compartments:
  - The class name
    - It should be a simple name and it should reflect exactly what the class is and does
  - The structure (attributes)
  - The behavior (operations)



public (+)

Visible to all elements that can access the contents of the namespace that owns it

private (-)

Only visible inside the namespace that owns it

protected (#)

Visible to elements that have a generalization relationship to the namespace that owns it

package (~)

Only named elements that are not owned by packages can be marked as having package visibility

Any element marked as having package visibility is visible to all elements within the nearest enclosing package

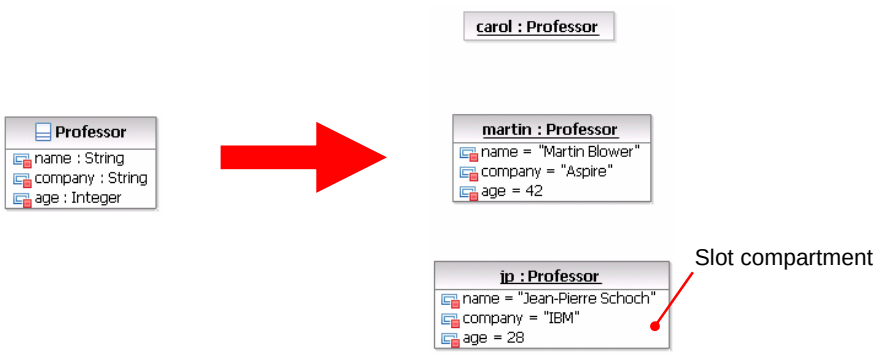
Representing Classes in UML (cont.)

- Style Guidelines
  - Capitalize the first letter of class names
  - Begin attribute and operation names with a lowercase letter
- Compartments
  - Only the name compartment is mandatory
  - Additional compartments may be supplied to show other properties

Representation with the attribute compartment suppressed, operation visibility shown as “decoration”, and the operation compartment filtered to show only two operations

The Relationship between Classes and Objects

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects.
- Classes are not collections of objects



Where Are We?

- What is an Object?
- What is a Class?
- Interaction Diagrams
- Sequence Diagrams
- Communication Diagrams

UML Diagrams

- UML diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model
  - Each diagram has a *contents area*
  - As an option, it may have a *frame* and a *heading* as shown on the right
- Two main types
  - *Structure Diagrams* show the static structure of the objects in a system
  - *Behavior Diagrams* show the dynamic behavior of the objects
    - Include interaction diagrams

Note: A complete taxonomy of UML diagrams is provided in module 4

Interaction Diagrams

- Objects interact through messages
  - A client object sends a “message” to a supplier object to perform some activity
- Interaction diagrams show the interactions between collaborating objects
  - Sequence Diagram (the most common variant)
    - Time oriented view of object interaction
  - Communication Diagram
    - Structural view of messaging objects
  - Specialized Variants (not covered in this module)
    - Timing Diagram: time constraint view of messages involved in an interaction
    - Interaction Overview Diagram: define interactions in a way that promotes overview of the control flow

Objects need to collaborate:

- Each object is responsible for its own behavior and status
- No one object can carry out every responsibility on its own

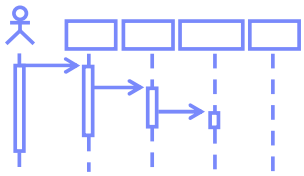
Where Are We?

- What is an Object?
- What is a Class?
- Interaction Diagrams
- Sequence Diagrams
- Communication Diagrams

IBM Software Group | Rational software

What Is a Sequence Diagram?

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages
- The diagram shows:
  - The objects participating in the interaction
  - The sequence of messages exchanged



Sequence Diagrams

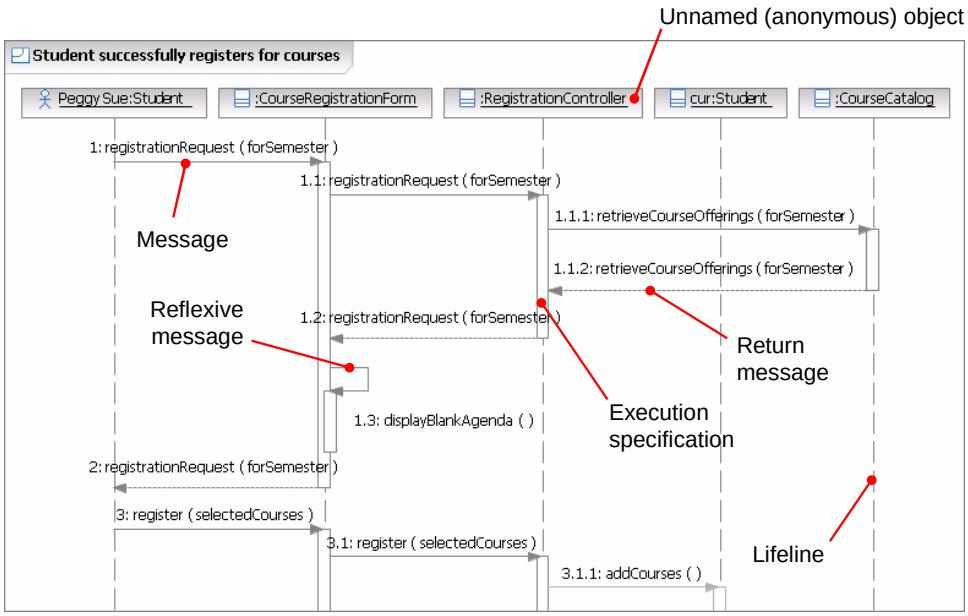
37

The **sequence diagram** is “the mainstay of dynamic modeling” (*The Object Primer, Third Edition*, Scott W. Ambler, 2004). It describes a pattern of interaction among objects, arranged in a chronological order.

Sequence diagrams are used to:

- Illustrate use-case realizations.
- Illustrate detailed structural designs.
- Model the detailed design of an operation or service.

Example



An object is represented by its **lifeline**, a rectangle forming its “head” followed by a vertical line, which may be dashed.

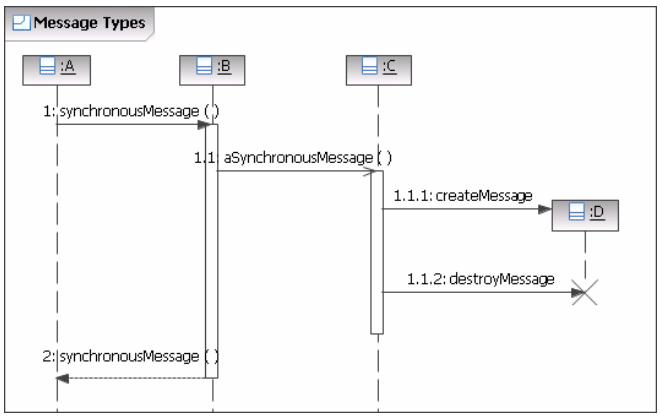
Note the inclusion of the **actor instance** *(Peggy Sue:Student).* This is important as it explicitly models what elements communicate with the “outside world.”

A **message** reflects either an operation call and start of execution or a sending and reception of a signal. The object’s class and the message’s operation may be initially unspecified.

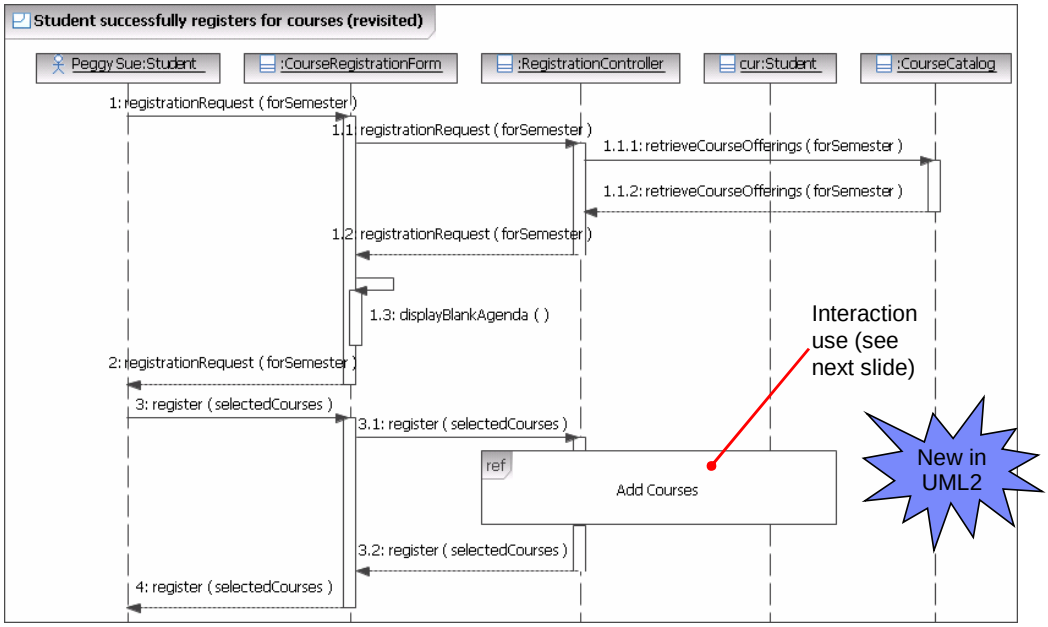
An **Execution specification** is a specification of the execution of a unit of behavior or action within the Lifeline.

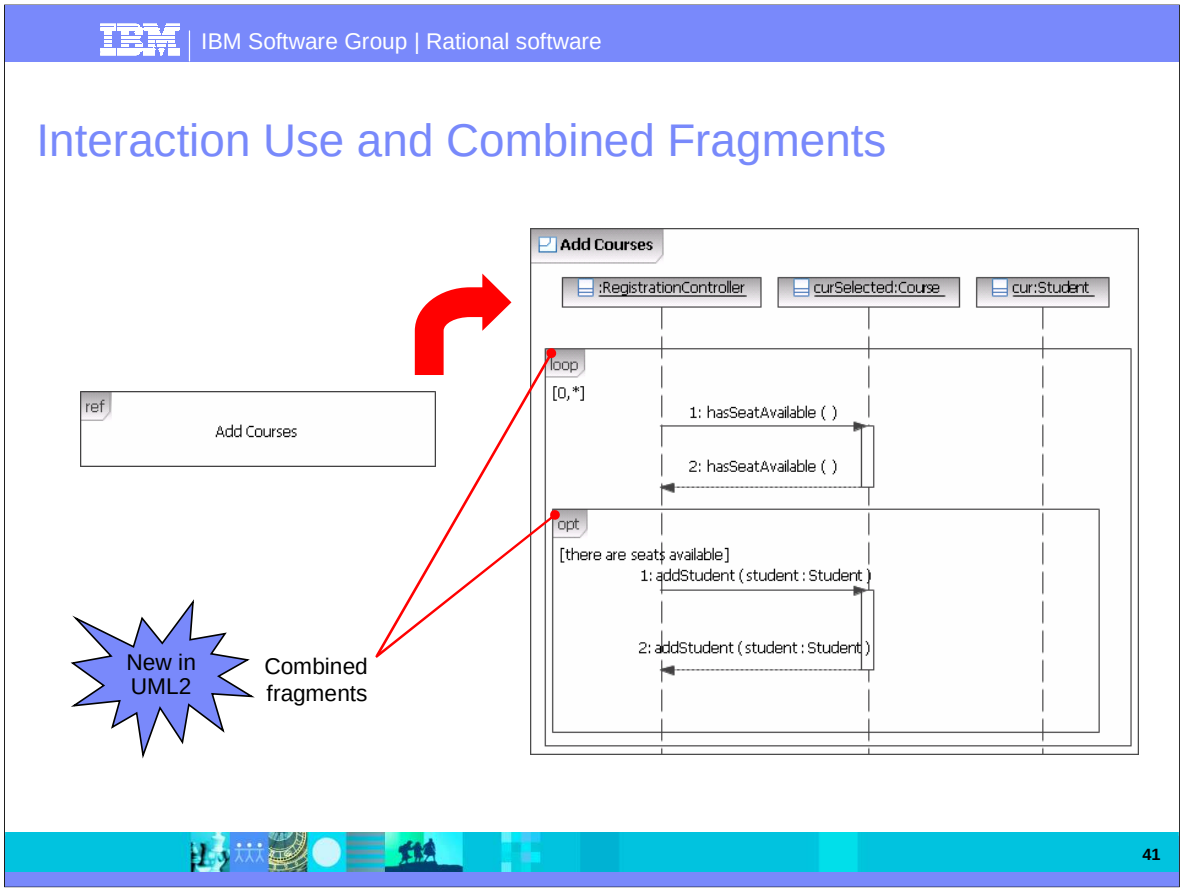
Message Types

- Synchronous message
  - Call to an operation
- Asynchronous message
  - Asynchronous call to an operation
  - Asynchronous send action of a signal
- Create Message
- Delete Message
- Reply message to an operation call



Interaction Use





An **Interaction Use** allows multiple interactions to reference an interaction that represents a common portion of their specification.

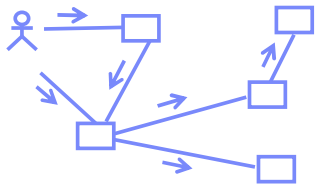
Other useful **Combined Fragments** include alt (represents a choice of behavior, opt (option), break (if a guard is included, and the guard is true, the rest of the enclosing Interaction Fragment is ignored), par (parallel). The operator critical can be used to indicate that a (critical) region is treated atomically by the enclosing fragment. More advanced operators: neg (negative), assert and ignore/consider, strict (strict sequencing) / seq (weak sequencing).

Where Are We?

- What is an Object?
- What is a Class?
- Interaction Diagrams
- Sequence Diagrams
- ⇨ Communication Diagrams

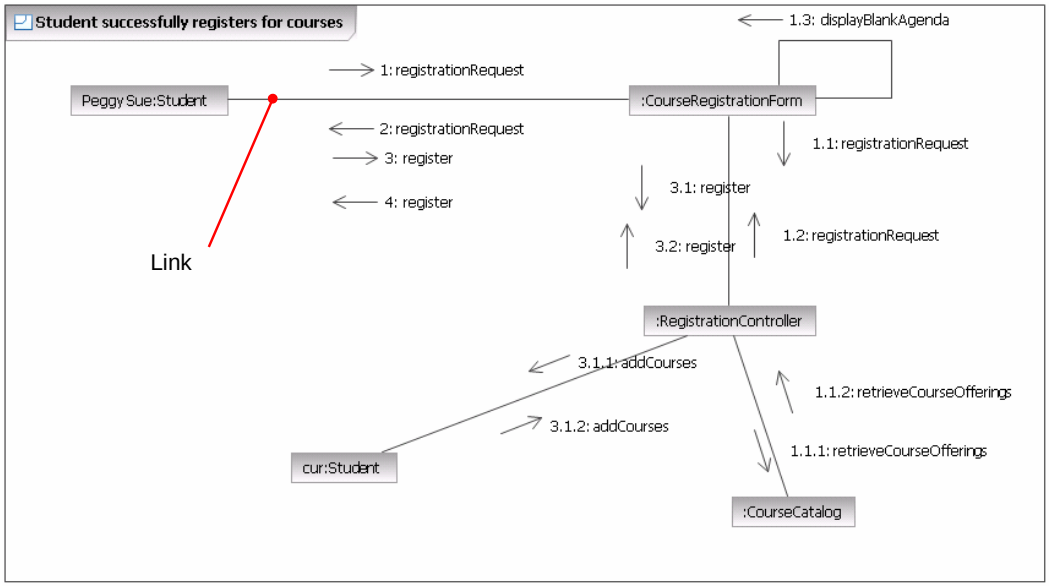
What Is a Communication Diagram?

- A communication diagram emphasizes the organization of the objects that participate in an interaction
- The communication diagram shows:
  - The objects participating in the interaction
  - Links between the objects
  - Messages passed between the objects



Communication Diagrams

Example

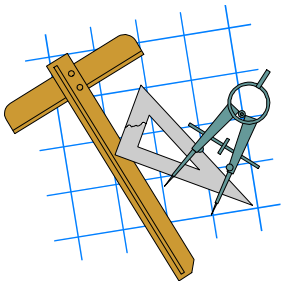


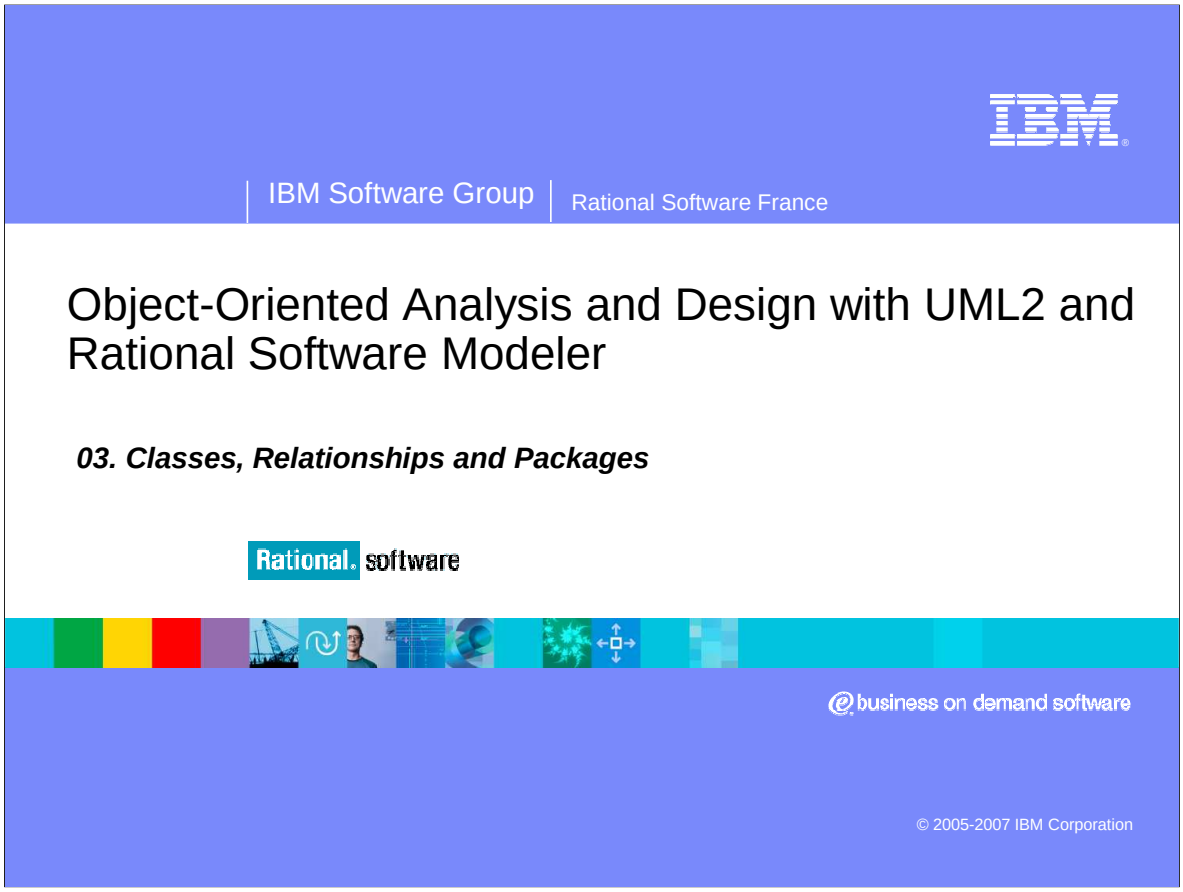
Communication vs. Sequence Diagrams

- In UML 1.x, collaboration diagrams (as they were called) and sequence diagrams were completely equivalent
  - The major difference, which is retained in UML 2.x, is the ability to explicitly show links in collaboration/communication diagrams
- In UML 2.x, communication diagrams are much less expressive and precise than sequence diagrams
  - UML 2.1 define communication diagrams as “simple Sequence Diagrams that use none of the structuring mechanisms such as InteractionUses and CombinedFragments”

Exercise

- Perform the exercise provided by the instructor (lab 1)





Where Are We?

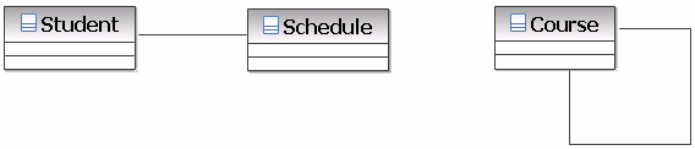
- Associations
- Dependencies
- Generalizations
- Packages
- Miscellaneous Topics

Classes Collaborate

- Objects collaborate with each other by sending messages
  - To send a message to an object, the client object must have a link with the supplier
- If there is a link between two objects, there must be some kind of relationship between the corresponding classes:
  - The relationship may be a structural relationship: association and its variants (aggregation, composition)
  - Or a non-structural relationship, called a dependency

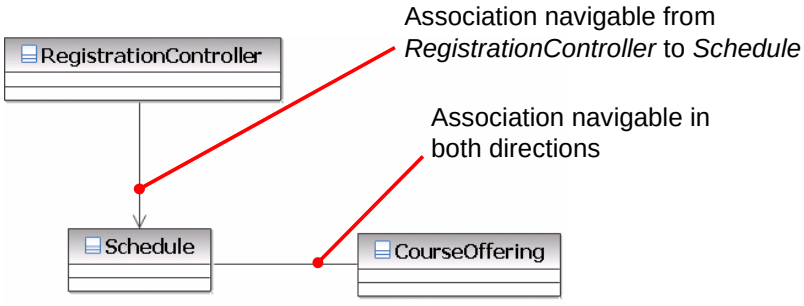
What Is an Association?

- The semantic relationship between two or more classifiers that specifies connections among their instances
- A structural relationship specifying that objects of one thing are connected to objects of another thing



What is Navigability?

- Navigability indicates that it is possible to navigate from one class to another (or more precisely from one instance of a class to another instance of the same class or of another class)
  - Associations are bi-directional by default
  - An association with an arrow is a one-way association
    - The client class can navigate to the supplier class, but not the other way around

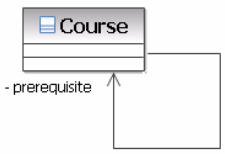


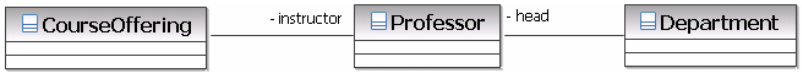
Naming Associations

- To clarify its meaning, an association can be named
- The name is represented by a label as shown below
- Usually a verb or an expression starting with a verb

Association Roles

- A role name specifies the role that a class plays in its relationship with another class
- Role names are typically names or noun phrases
- A role name is placed near the association next to the class to which it applies (as a role)
  - Each association end may be named





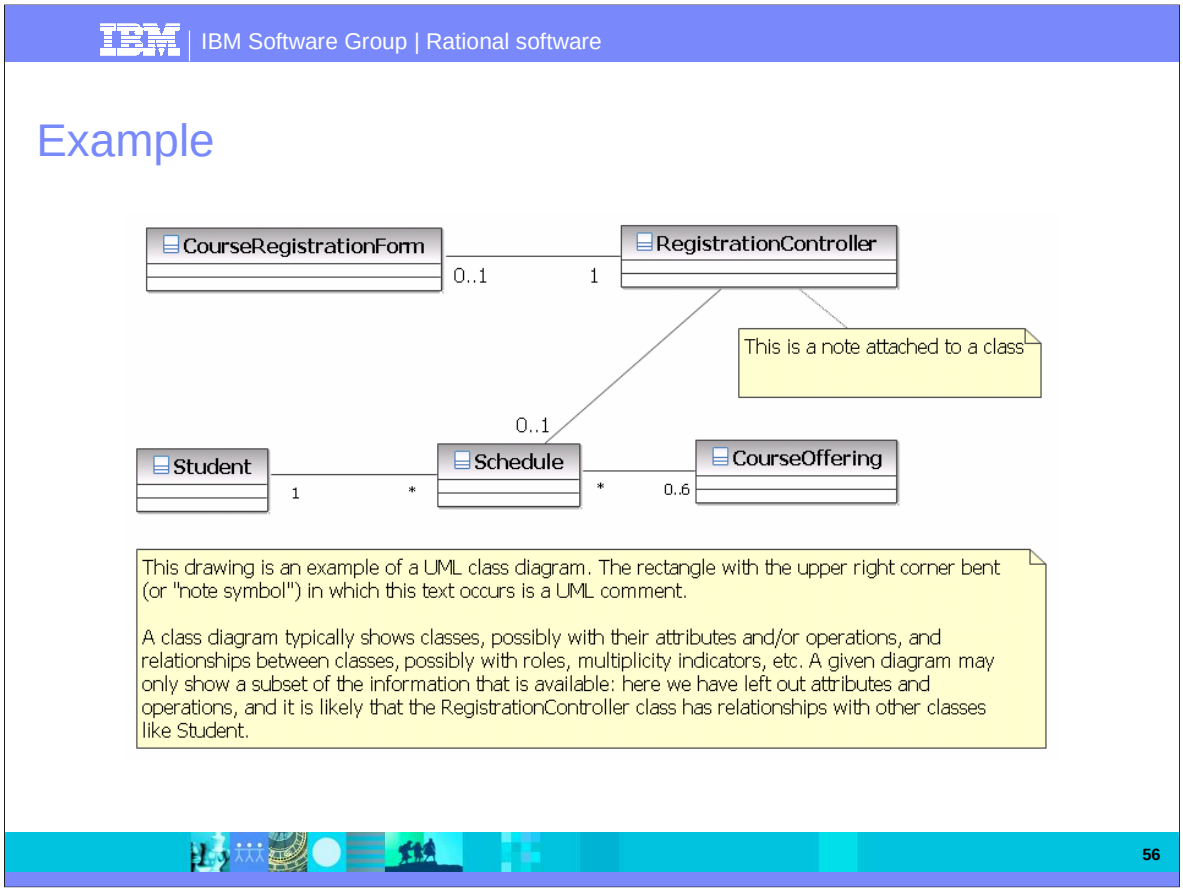
What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class
- For each association, there are two multiplicity decisions to make, one for each end of the association
  - For each instance of Professor, many Course Offerings may be taught
  - For each instance of Course Offering, there may be either one or zero Professor as the instructor

Multiplicity Indicators

- Unspecified
- Exactly one
- Zero or more (no upper limit)
- One or more
- Zero or one (optional)
- Specific range
- Multiple, disjoint ranges

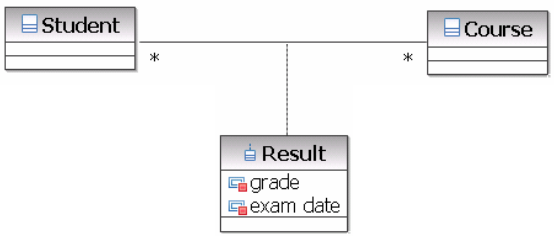
1
0..*
*
1..*
0..1
2..4
2, 4..6



Although UML only has the notion of a comment, RSM has separate menu entries for notes and comments. Comments in RSM map directly to comments in UML and, as such, will appear in the model hierarchy. Notes are constructs that are bound to the actual diagrams. If a diagram is deleted, its notes are also deleted.

Association Classes

- Take the n-n relationship between *Student* and *Course*
- If you have to capture the grade received by a student for a given course, where would you place the grade? On *Student*? On *Course*?



- The answer is on the association itself by adding an association class, called *Result*
- The association class represents the association of exactly one student and one course

Qualified Associations

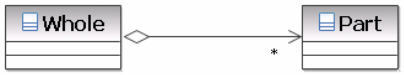
- A qualifier is an attribute of an association whose values partition the set of objects related to an object across an association
  - A qualified association represents a lookup table (which can be implemented as a hash table for instance)
- The multiplicity of the target class is often 0..1 but it may be 0..*
- Example:

  - In the context of the *WorkDesk*, you'd have a *jobId* that would identify a particular *ReturnedItem*. In that sense, *jobId* is an attribute of the association. Then, given an object of type *WorkDesk* and a particular value for *jobId*, you can navigate to 0 or 1 object of type *ReturnedItem*.

In RSM, to add a qualified association, you must right-click the association role (here *returneditem*) in the Project Explorer, then select *Add UML > Qualifier*.

What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts
  - An aggregation is an “is a part-of” relationship
- Multiplicity is represented like other associations



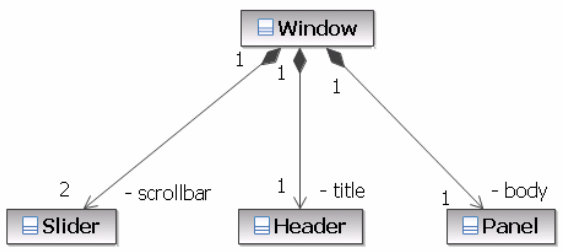
Aggregation is used to model a whole-part relationship between model elements. There are many examples of whole-part relationships: a Library contains Books, Departments are made up of Employees, a Computer is composed of a number of Devices.

A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.

An aggregation relationship that has a multiplicity greater than one for the aggregate is called **shared**. Destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph or a tree with many roots. Shared aggregations are used when one instance is a part of two other instances. So, the same instance can participate in two different aggregations.

What Is a Composition?

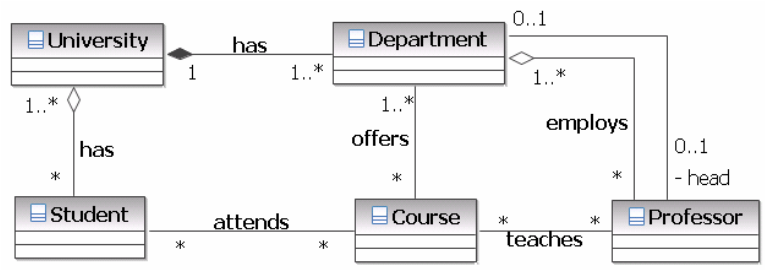
- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate



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.

Example



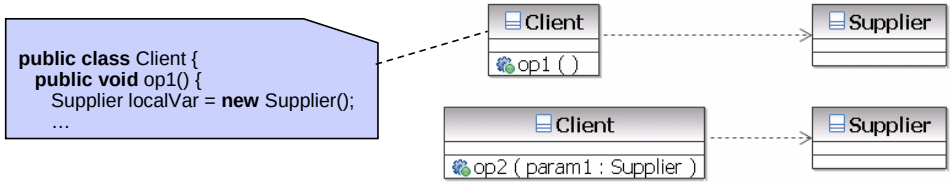
Where Are We?

- Associations
- Dependencies
- Generalizations
- Packages
- Miscellaneous Topics

What Is A Dependency?

- A non-structural relationship between two classes
  - The client needs access to the services provided by the supplier
  - But doesn't need to maintain a permanent relationship with the supplier objects (transient relationship)

- A dependency may result from:
  - A local declaration within the body of an operation (op1 below)
  - The supplier appears as a parameter type (op2 in the example)



Where Are We?

- Associations
- Dependencies
- Generalizations
- Packages
- Miscellaneous Topics

What Is Generalization?

- A relationship among classes where one class shares the structure and/or behavior of one or more classes
  - A subclass inherits its parent's attributes, operations, and relationships
  - A subclass may:
    - Add additional attributes, operations, relationships
    - Redefine inherited operations (use caution!)
- Defines a hierarchy of abstractions in which a subclass inherits from one or more super-classes
- Is an "is a kind of" relationship
- Single or multiple inheritance

Generalization can be defined as:

- A specialization/generalization relationship, in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). (*The Unified Modeling Language User Guide*, Booch, 1999.)
- The subclass may be used where the super-class is used, but not vice versa.
- The child inherits from the parent.
- Generalization is transitive. You can always test your generalization by applying the "is a kind of" rule. You should always be able to say that your generalized class "is a kind of" the parent class.
- The terms "generalization" and "inheritance" are generally interchangeable, but if you need to distinguish, generalization is the name of the relationship. Inheritance is the mechanism that the generalization relationship represents/models.

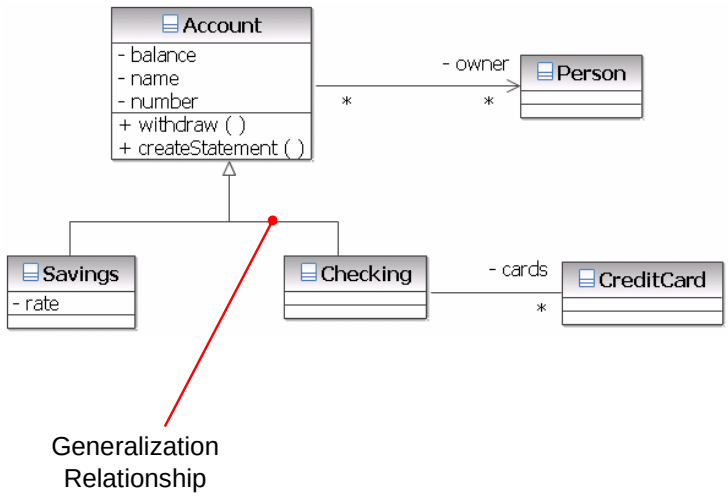
Inheritance can be defined as:

- The mechanism by which more specific elements incorporate the structure and behavior of more general elements. (*The Unified Modeling Language User Guide*, Booch, 1999.)
- Single inheritance: The subclass inherits from only one super-class (has only one parent).
- Multiple inheritance: The subclass inherits from more than one super-class (has multiple parents).

Example

Superclass, parent or ancestor

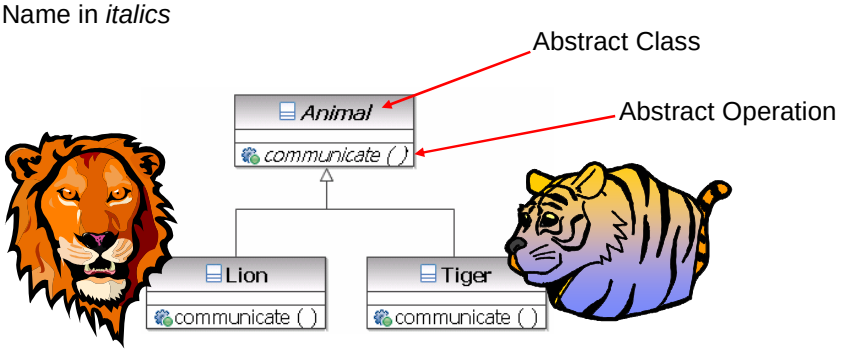
Subclasses, children or descendants



Generalization Relationship

What is an Abstract Class?

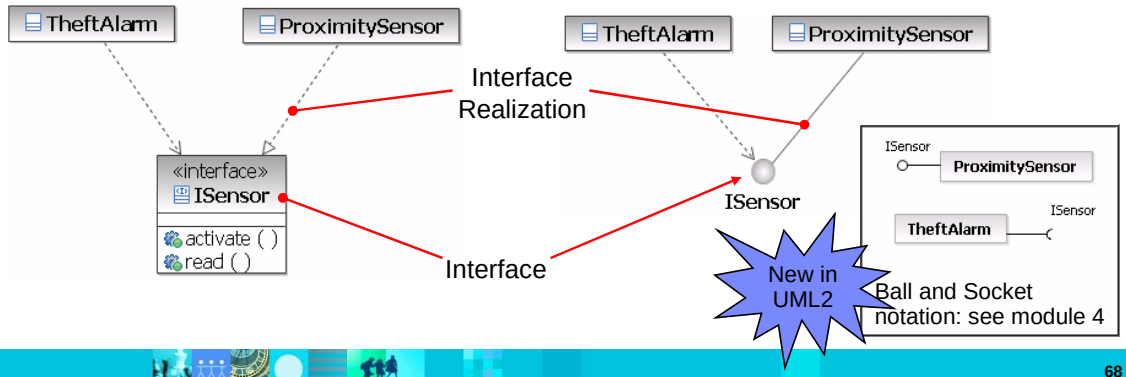
- Abstract classes cannot be instantiated
  - Name in *italics*
- Abstract operations: the subclasses must provide the implementation
  - Name in *italics*



All objects are either lions or tigers

What Is An Interface?

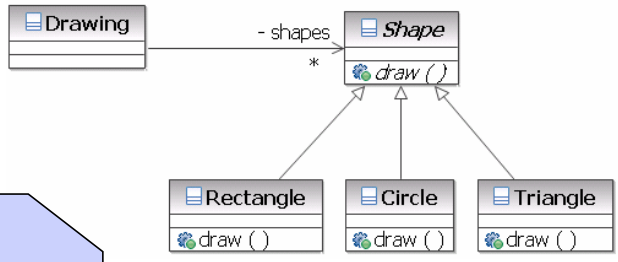
- A “contract” between providers and consumers of services
  - Equivalent to an abstract class in which all operations are abstract
  - Provided interfaces: interfaces the element exposes to its environment
  - Required interfaces: interfaces the element requires from other elements
  - Note: UML interfaces can have attributes (not possible for Java interfaces)
- The provider of the interface is said to realize the interface



The label <<interface>> is called a stereotype. Stereotypes are formally introduced later on in this module.

What Is Polymorphism?

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



Without polymorphism:

```
Shape s = …;
If (s instanceof Rectangle) {
   ((Rectangle) s).draw();
else if (s instanceof Circle) {
   ((Circle) s).draw();
else if (s instanceof Triangle) {
   ((Triangle) s).draw();
}
```

With polymorphism:

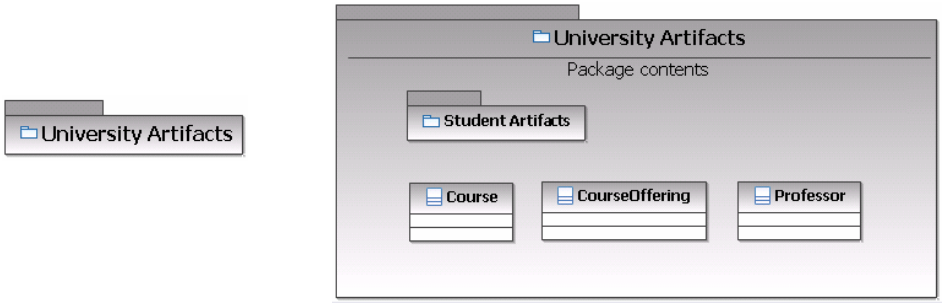
```
Shape s = …;
s.draw();
```

Where Are We?

- Associations
- Dependencies
- Generalizations
- Packages
- Miscellaneous Topics

What Is a Package?

- A general purpose mechanism for organizing elements into groups
- A model element that can contain other model elements
  - Grouping of logically related elements
- A package can be used:
  - To organize the model under development
  - As a unit of configuration management

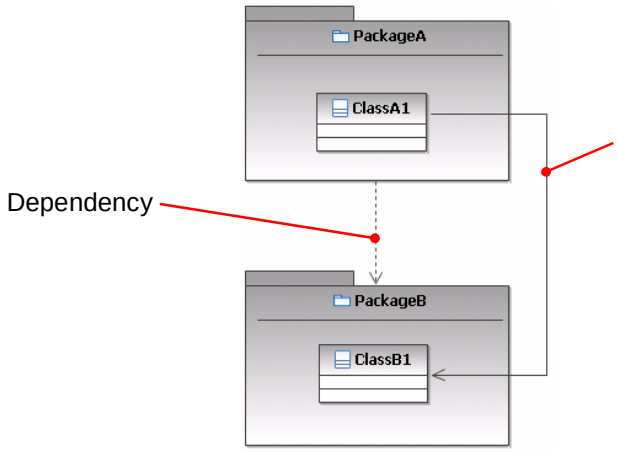


A **Package** can be defined as:

- A general purpose mechanism for organizing elements into groups. (*The Unified Modeling Language User Guide*, Booch, 1999.)
- Models can contain hundreds and even thousands of model elements. The sheer number of these elements can quickly become overwhelming. Therefore, it's critical to group model elements into logical collections to maintain and easily read the model (application of modularity and hierarchy).
- Packages are a general grouping mechanism for grouping elements into semantically related groups. A package contains classes that are needed by a number of different packages, but are treated as a "behavioral unit."
- A package is simply a grouping mechanism. No semantics are defined for its instances. Thus, packages do not necessarily have a representation in implementation, except maybe to represent a directory.
- In the UML, a package is represented as a tabbed folder.
- Package diagrams depict dependencies between packages and are now formalized in UML 2.

Package Relationships

- A package A depends on a package B if there is at least one element from A that depends on at least one element from B



Where Are We?

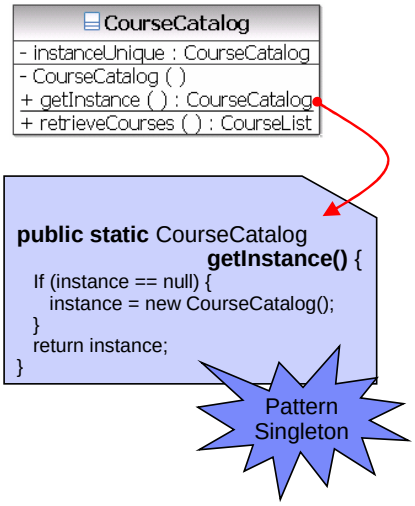
- Associations
- Dependencies
- Generalizations
- Packages
- ⇨ Miscellaneous Topics

What Is A Stereotype?

- A stereotype is a mechanism used to extend the vocabulary of UML
  - Represented textually (<<mystereo>>) and/or graphically
  - Any UML element may be stereotyped
- Stereotypes are grouped into collections of stereotypes, called *profiles*
- Can be defined for specific domains and/or applications
  - Pre-defined or custom
- A stereotype may have its own properties

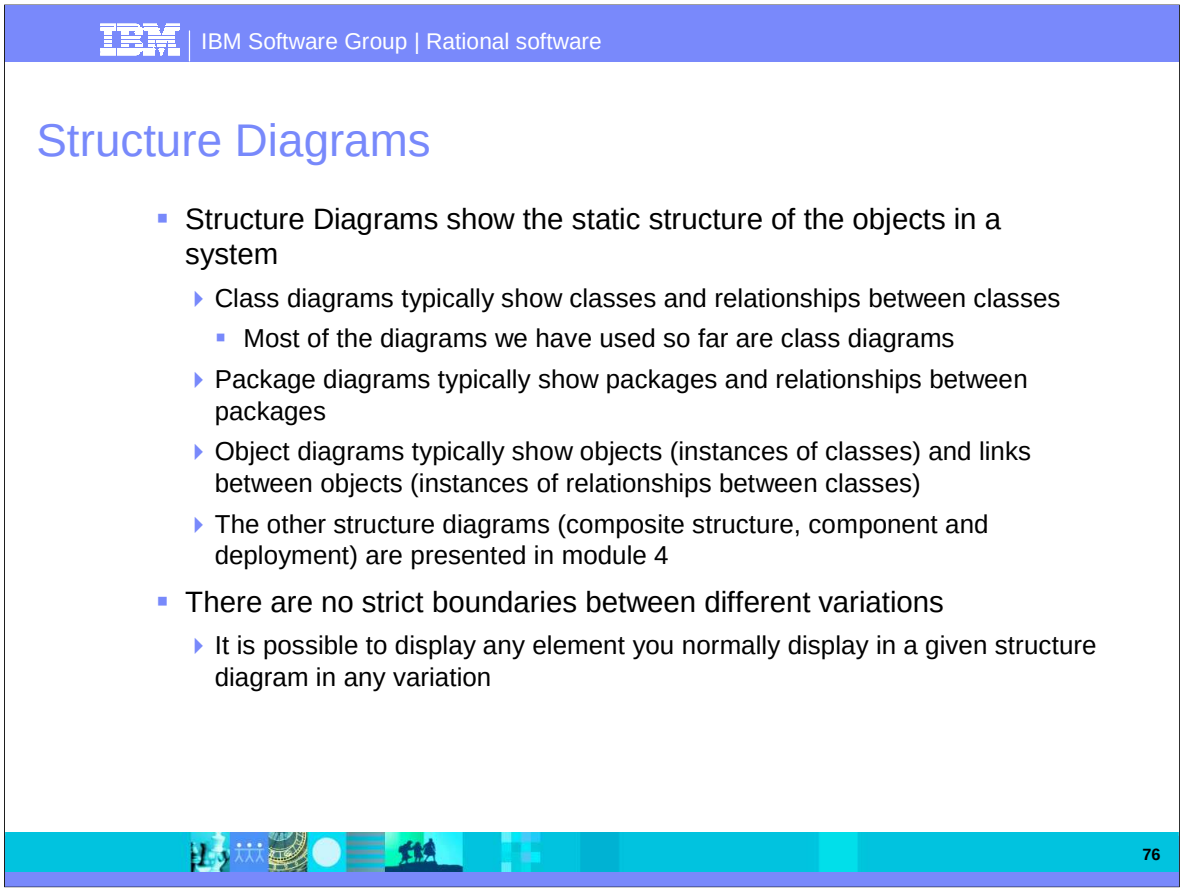
Classifier Scope

- 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
  - "Static" attributes and operations in languages like Java and C++



And some client code:

```
CourseCatalog cat = CourseCatalog.getInstance();
CourseList courses = cat.retrieveCourses();
```

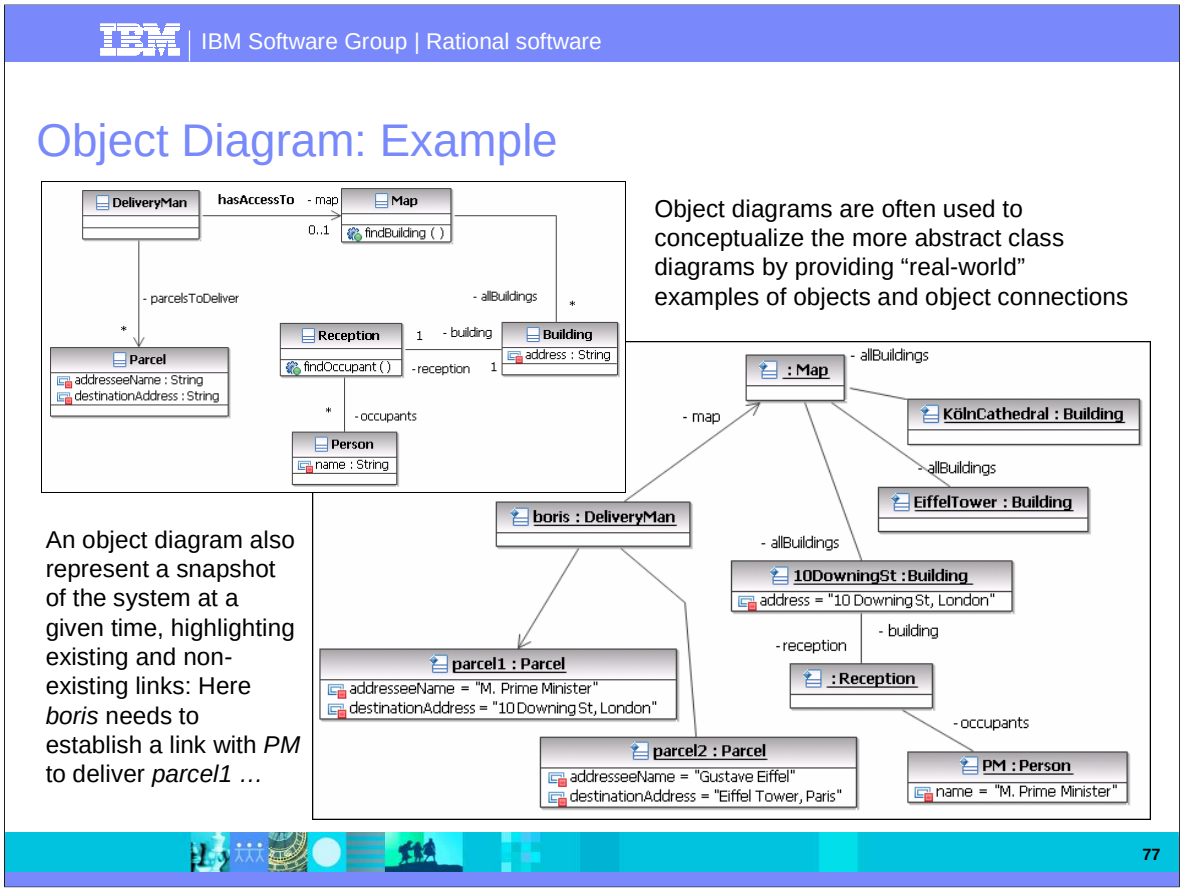


Class diagrams are used for a variety of purposes. According to Scott Ambler (The Elements of UML Style, 2005), "they are used to:

- explore domain concepts in the form of a domain model,
- analyze requirements in the form of a conceptual/analysis model,
- depict the detailed design of object-oriented or object-based software."

Any diagram that depicts only packages (and their interdependencies) is considered a **package diagram**. The term "UML package diagrams" is in fact new to UML 2.

One important use of packages is to logically organize the design of your system. Another fundamental – but possibly underestimated – use of packages is to provide a high-level overview of the system, showing the main parts/components and their interdependencies. This is a topic we will discuss in detail at several points in this course.



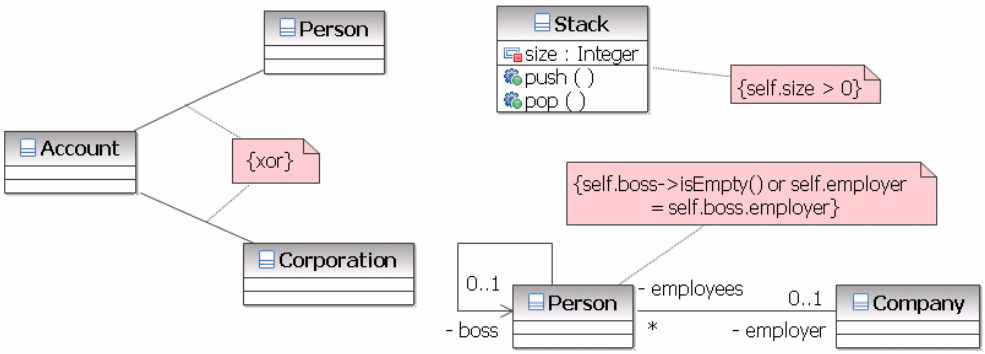
The example above illustrates one use of object diagrams that is often overlooked, and yet points to the real challenge of the object-oriented analysis and design approach: traditional systems were focused on how to implement the system algorithms. In OO algorithms are encapsulated in specialized objects (and as much as possible existing objects) and the challenge has moved to identifying the objects that provide the correct functionality.

If we go back to our example, the class diagram makes it clear that the purpose of this system is for a *DeliveryMan* to deliver *Parcel*s to a person identified by the parcel attributes. The object diagram shows the state of the system BEFORE the *DeliveryMan* object *boris* has started its processing: *boris* has access to the parcels to deliver and to a map, but not (yet) to the addressee (the object *PM* in the case of *parcel1*). *boris* will use its map to identify the correct *Building* using the *Parcel*'s address. Having access to the building, it can ask the *Reception* object (a singleton in the scope of a given building) to identify the occupant given the addressee's name. At that point, *boris* will have a (presumably transient) link to *PM* and will be able to complete the processing of *parcel1*. Alternatively, *boris* could deliver the parcel to the *Reception* object, which, in turn, will identify the addressee and hand over the parcel to *PM*. No matter what the solution is, *boris* will need to obtain some kind of *Receipt* object (not shown here) in exchange…

•In UML2, an object is also called an *instance specification*.

What Is A Constraint?

- A condition or restriction (a Boolean expression) expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element
- Some are predefined in UML (“xor”), others may be user-defined
- OCL (Object Constraint Language) is a predefined language for writing constraints

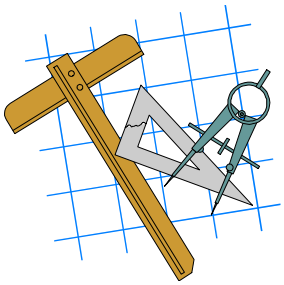


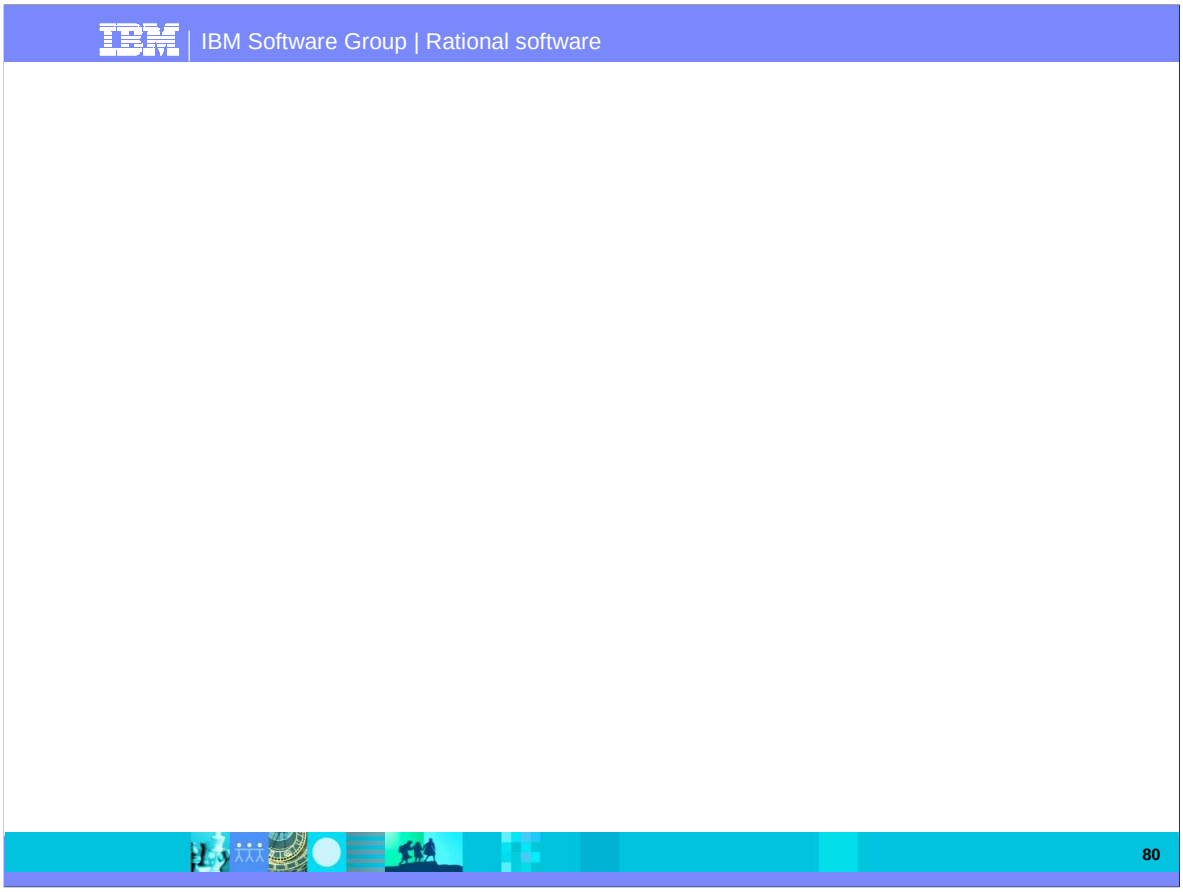
RSM Implementation Note: the use of a note to express the constraint is optional in UML. For instance, it should be possible to draw a dashed line with the label xor between the Account-Person and Account-Corporation associations. In RSM, the use of the note symbol is imposed.

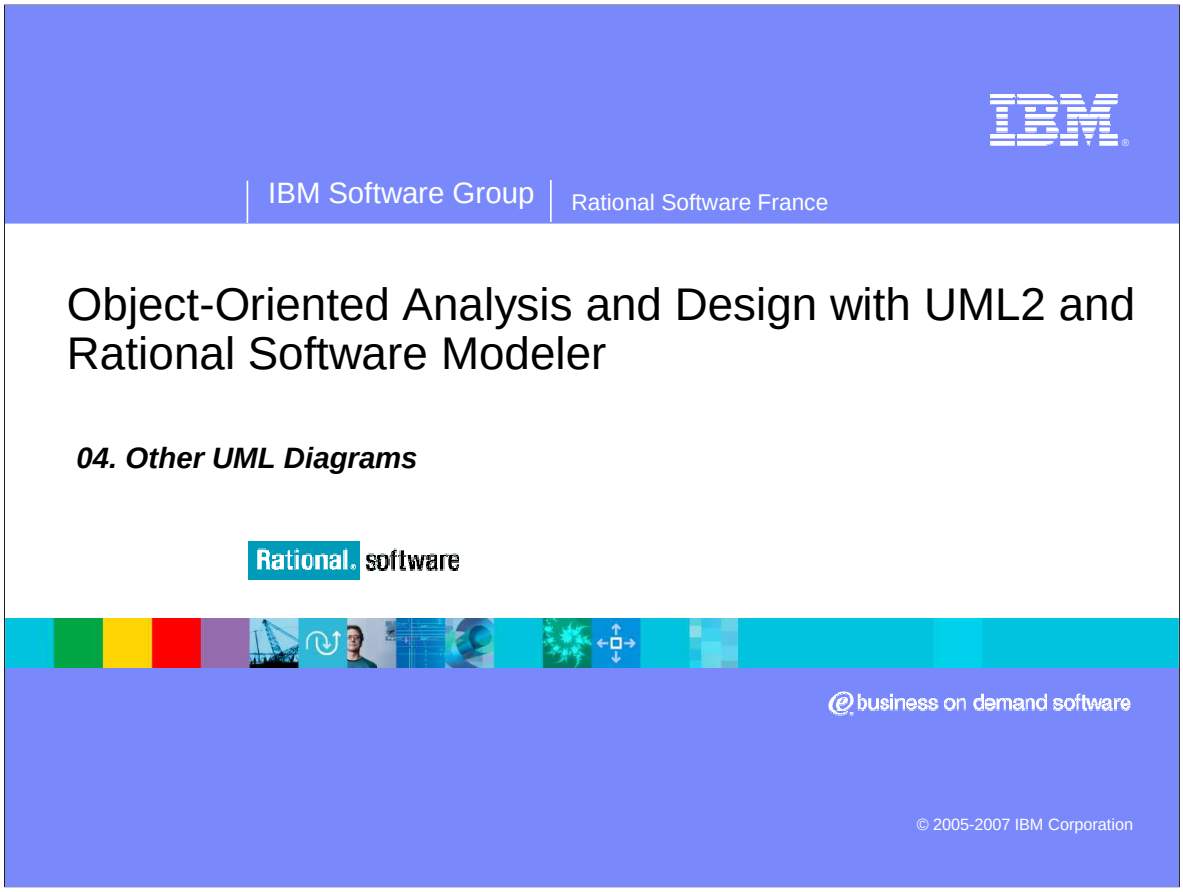
RSM has a built-in OCL editor with completion lists. The 2 examples on the right were written using this editor.

Exercise

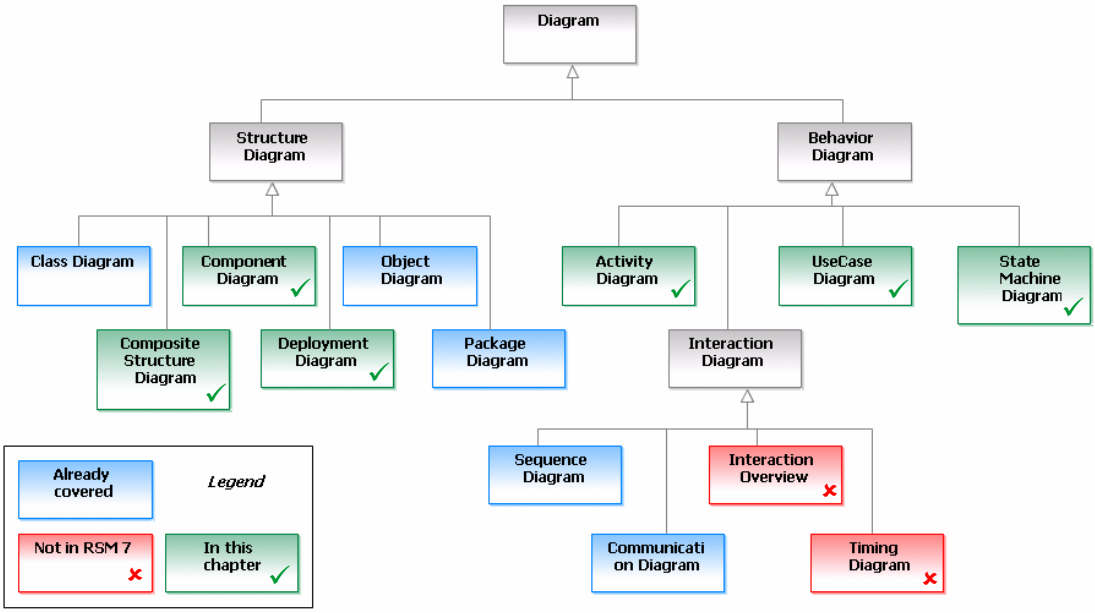
- Perform the exercise provided by the instructor (lab 2)







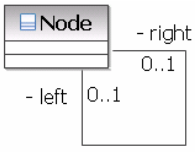
The Taxonomy of Diagrams in UML 2.1



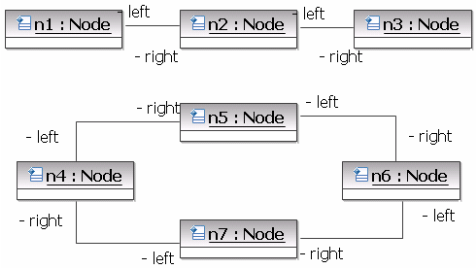
Composite Structure Diagrams

New in UML2

- A composite structure diagram can be used to represent the internal structure of a classifier, for instance a class
  - Allows to show details otherwise not visible on other diagrams, as illustrated in the following example:

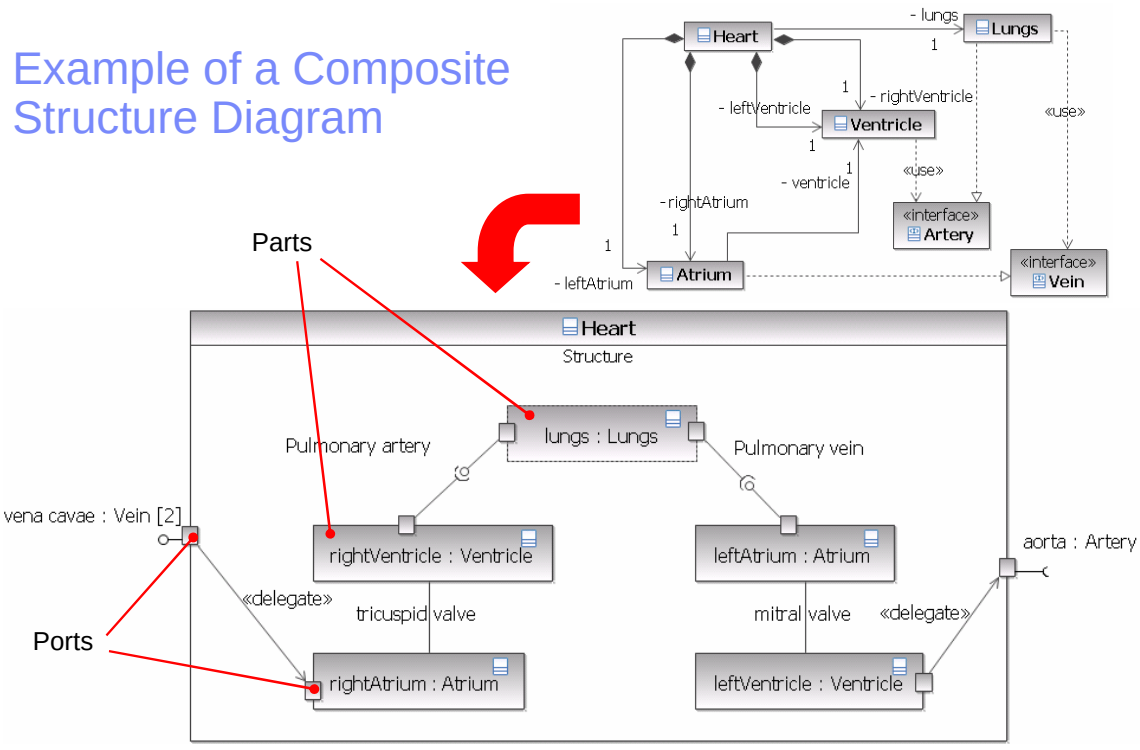


Same class diagram describes multiple implementations!



- External structure of the classifier described using *ports*, “collection points” for the classifier’s provided and required interfaces
- Internal structure shown using *parts* and *connectors* in the *structure compartment* (see example next slide) or in a *structure diagram*

Example of a Composite Structure Diagram

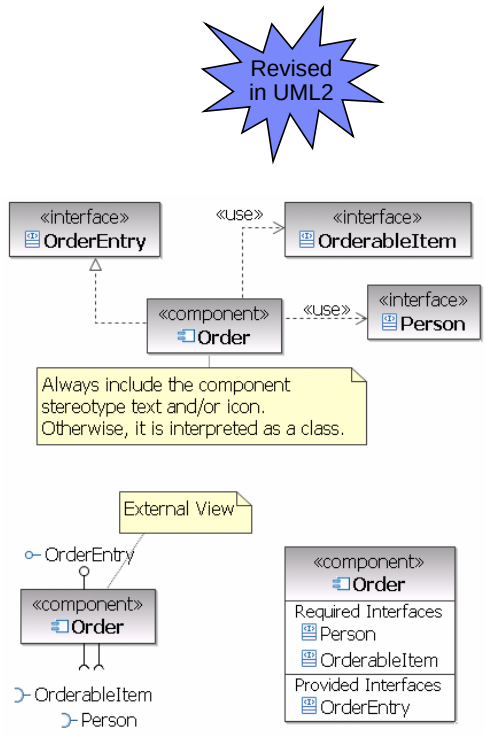


IBM Software Group | Rational software

Components

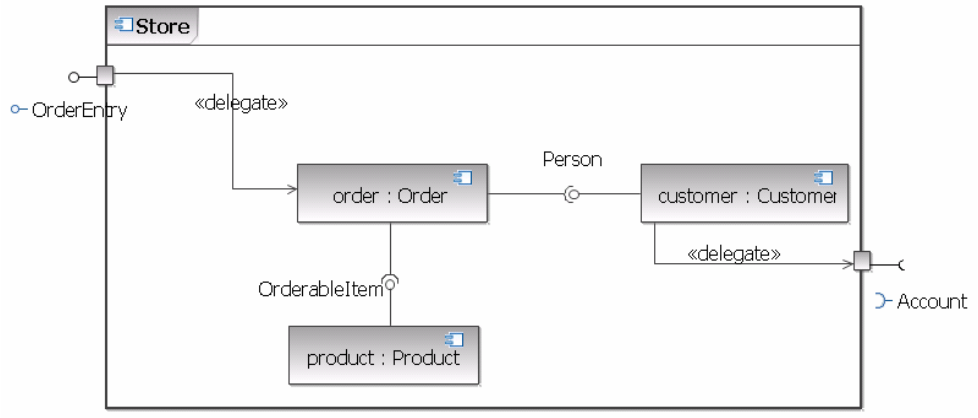
- Definition of a component:
  - Specifies a formal contract of the services that it provides to its clients (*provided interfaces*) and those that it requires from other components or services in the system (*required interfaces*)
  - Can be *replaced* at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces
- Components can be used to:
  - Provide a high-level, architectural view of the system
  - Represent the logical components that will be running on the physical systems

Three views of the same component



The relationship between a component and an interface is an interface realization. UML2 introduces also a **component realization** for a component to realize (or implement) other classifiers, including other components.

Example of a Component Diagram



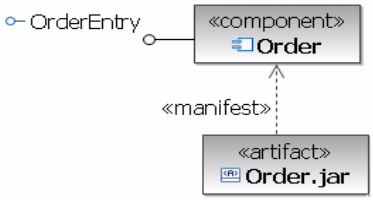
The structure diagram of the Store component ▲

Subsystems

- A subsystem is a specialized version of a component, but it does not add anything to it
  - The decision to use a subsystem vs. a component is up to the methodology of the modeler
  - Subsystems are often equated to larger components
  - Component stereotyped <<subsystem>>
  - Note: There are other pre-defined UML2 component stereotypes (e.g. <<implement>>, <<specification>>, <<process>>, <<service>>)

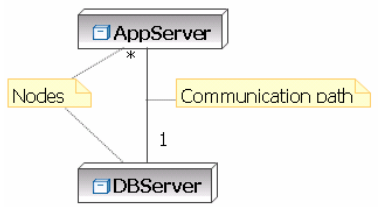
Artifacts

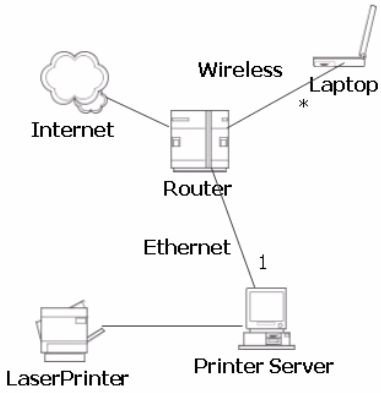
- An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system
  - Examples: source files, scripts, and binary executable files
  - Standard UML stereotypes: <<document>>, <<executable>>, <<file>>, <<library>>, <<script>>, <<source>>
- The physical rendering of one or more model elements by an artifact is shown with a <<manifest>> dependency
  - May be further stereotyped (e.g. <<tool generated>>)
- Artifacts may have composition associations to other artifacts that are nested within it



Deployment: Nodes and Communication Paths

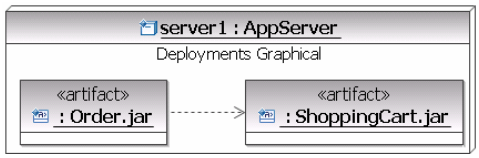
- A node is computational resource upon which artifacts may be deployed for execution
  - Examples: application server, client workstation, mobile device, embedded device
- Nodes can be connected to represent a network topology by using communication paths
  - Specific network topologies can then be defined through links between node instances
- Hierarchical nodes (i.e., nodes within nodes) can be modeled using composition associations, or by defining an internal structure

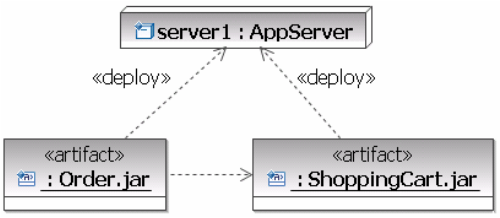


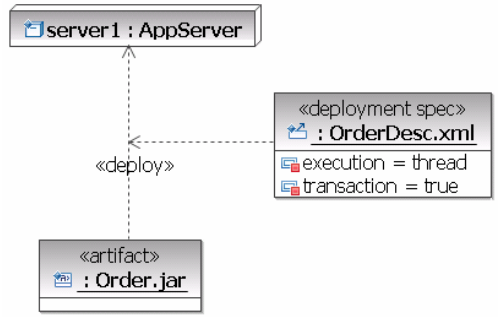


Deploying Artifacts

- A deployment is the allocation of an artifact or artifact instance to a deployment target
- A deployment specification can be used to specify the execution parameters of a component artifact that is deployed on a node



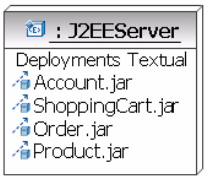




Two equivalent visual representations of the deployment of artifacts to a deployment target (including th e dependency between the artifacts)

Deployment: Execution Environments

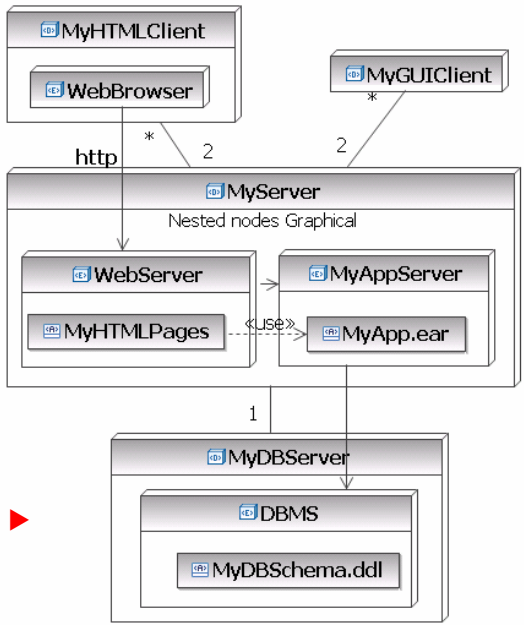
- An Execution Environment is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts
  - Examples: OS, workflow engine, database system, J2EE container
- Execution Environment instances are assigned to node instances by using composite associations (the Execution Environment plays the role of the part)
- Execution Environments can be nested (e.g., a database Execution Environment may be nested in an operating system Execution Environment)



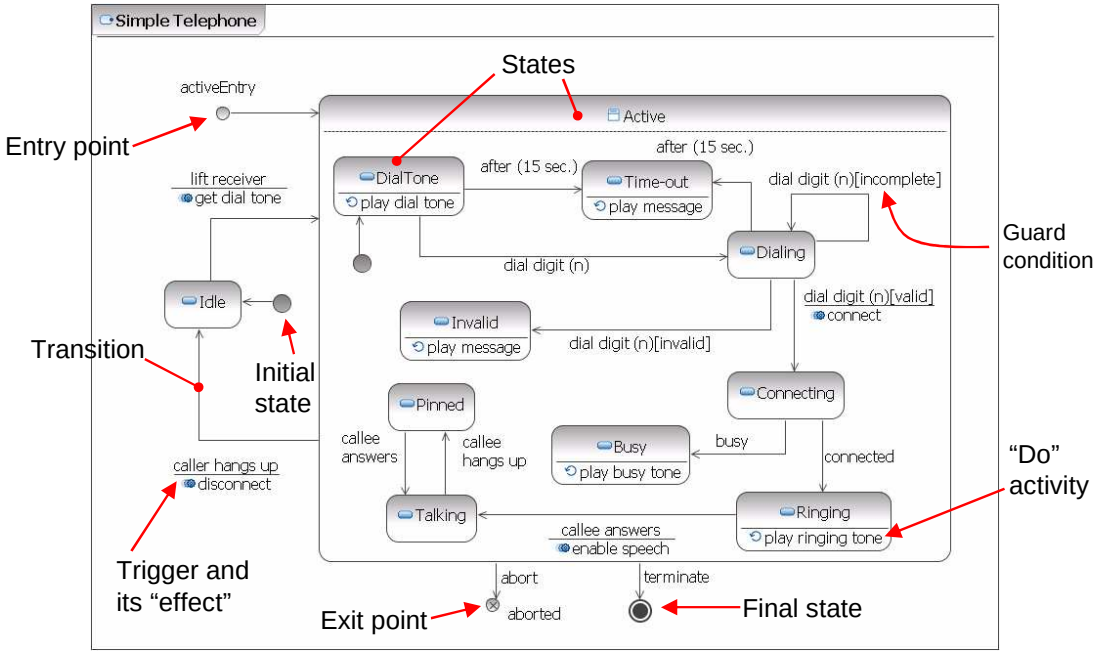
Deployment: Devices

- A Device is a physical computational resource with processing capability upon which artifacts may be deployed for execution
- Devices may be complex (i.e., they may consist of other devices)

A complete deployment diagrams with devices, execution environments and artifacts

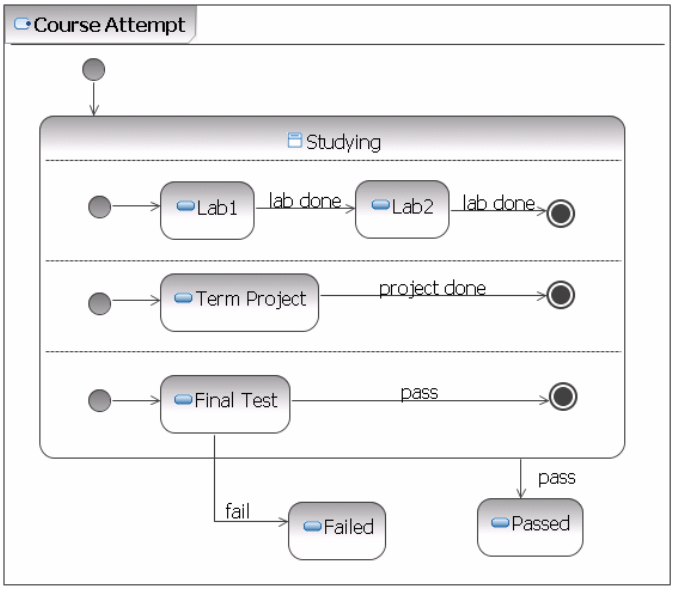


State Machine Diagrams

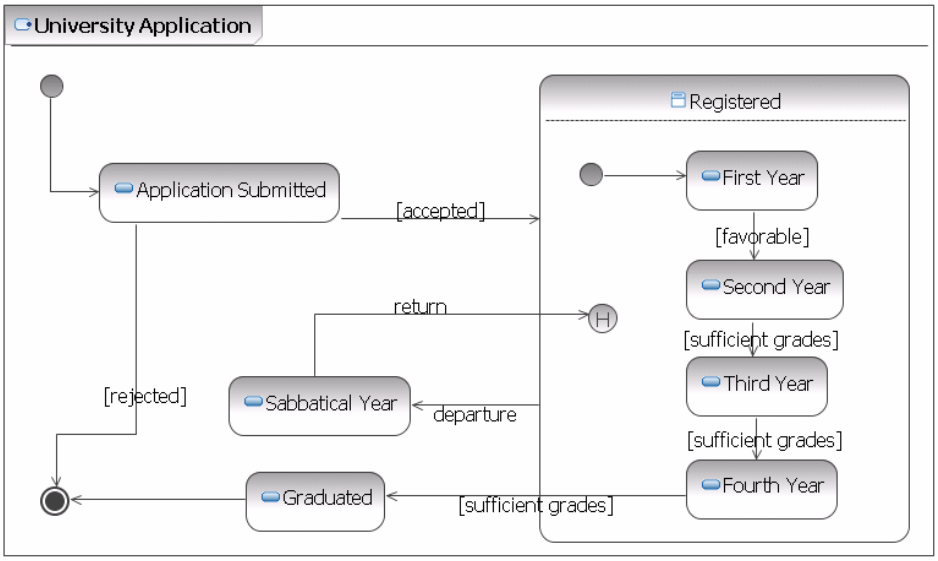


A **state machine diagram** describes the states an object or interaction may be in, as well as the transitions between states. Typically used to explore the design of a complex class or component.

Orthogonal State with Regions



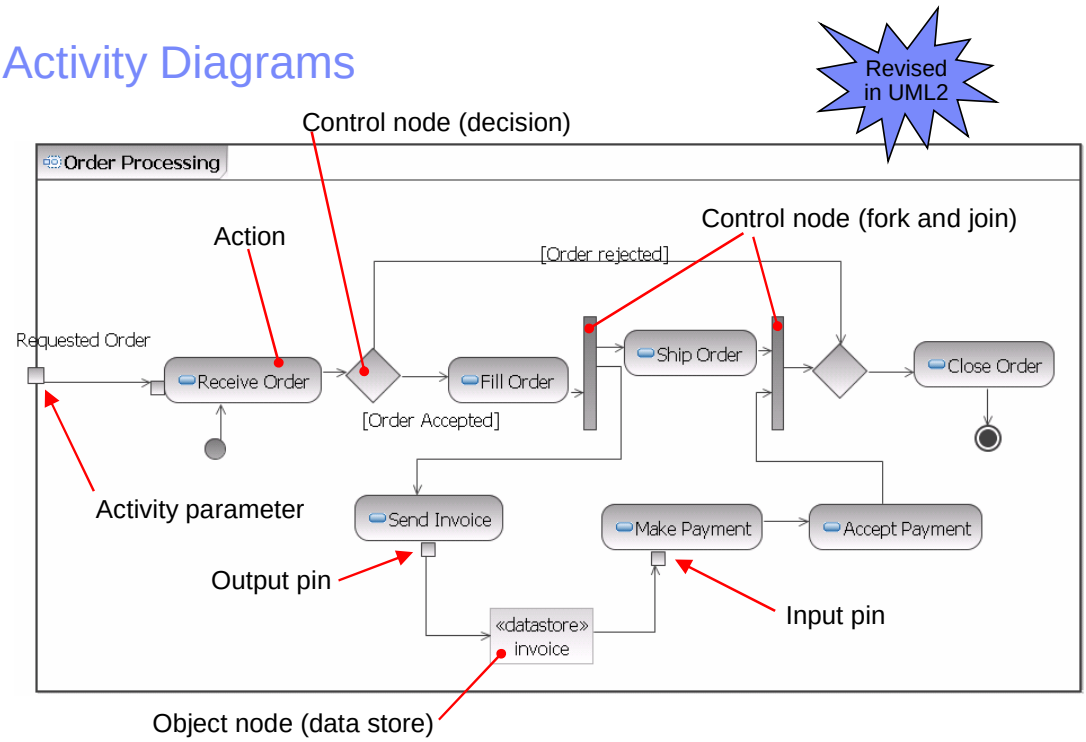
State Machine Diagram with History



If the transition terminates on a **shallow history** pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is target of the transition originating from the history pseudostate.

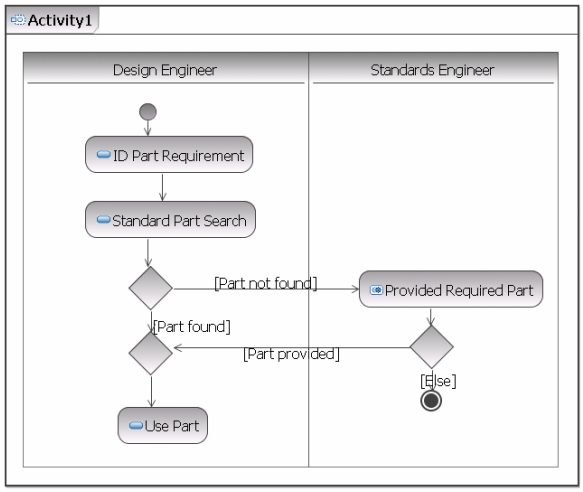
Deep history entry: The rule here is the same as for shallow history except that the rule is applied recursively to all levels in the active state configuration below this one.

Activity Diagrams

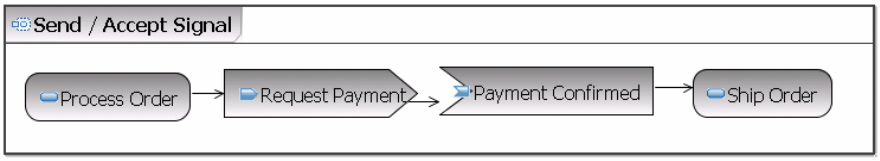


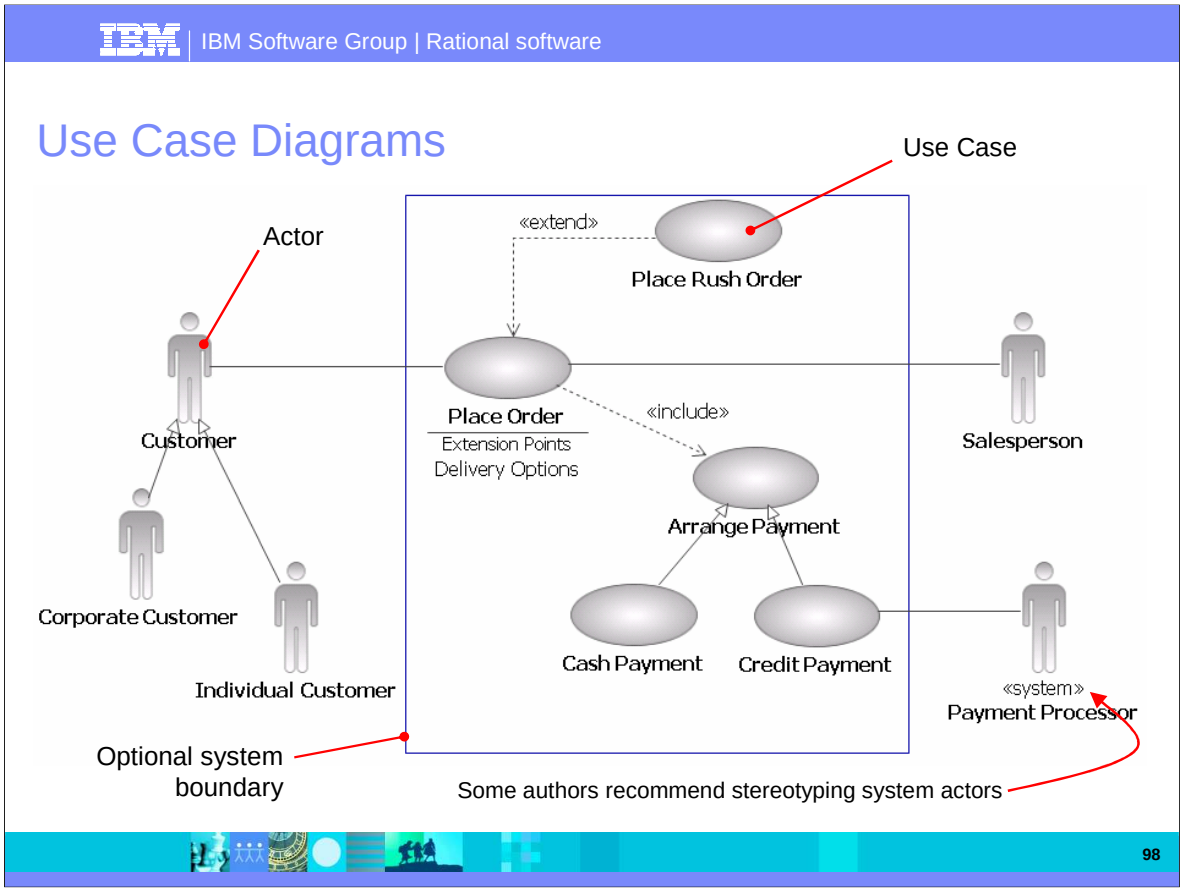
Activity Diagrams (cont.)

Activity diagram with partitions ▶



Activity diagram with Send / Receive Signal Actions ▼





An **actor** models a type of role played by an entity that interacts with the system, but which is *external* to the system. Actors may represent roles played by human users, external hardware, or other systems.

An actor is active (initiates a use case) or passive.

Some authors, like Scott Ambler (*The Elements of UML 2.0 Style, 2005*), recommend stereotyping system actors

A **use case** is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors.

An **include** relationship between two use cases means that the behavior defined in the including use case is included in the behavior of the base use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases.

An **extend** relationship means that the extending use case continues the behavior of a base use case by inserting additional action sequences. The extending use case can only extend the base use case at specific **extension point** and only when the extension conditions (if any) are fulfilled.