UNIT – II

Basic Structural Modeling & Advanced Structural Modeling

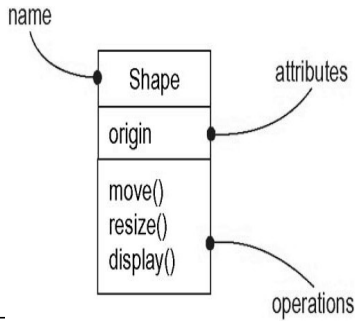
Basic Structural Modeling:

Introduction:

- Basic structural modeling deals with the selection of structural elements and their interface by which system is composed
- There structural elements include, classes, relationships, common mechanisms and various diagrams
- More advanced structural modeling provides more in-depth information about the classes, relationships, interfaces, roles, packages and instances
- Common modeling techniques of structural element gives how to model these elements under various situations.

Class

- Classes are important building block of any object oriented system.
- A Class is a description of set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces.
- Graphically, a class is rendered as a rectangle
- Classes are used to capture vocabulary of a system to represent software things and hardware things.
- A class is abstraction of things that are part of your vocabulary
- Class is not individual object but represent whole set of objects



Graphical Representation of Class in UML

Terms and Concepts

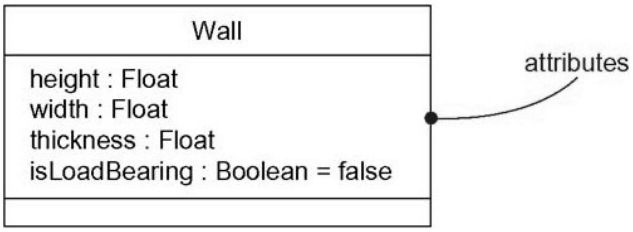
Names

Every class must have a name that distinguishes it from other classes.
A name is a textual string and can be a simple name or a path name which is prefixed by the name of the package in which that class lives.

Simple Name **Path Name**

Attributes

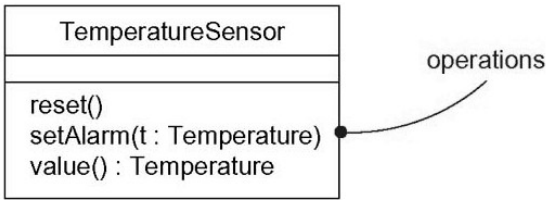
- An attribute is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of thing you are modeling that is shared by all objects of that class.
- An attribute can be given by stating its class and default initial value
- Graphically attributes are listed just below class name



Attributes and Their Class

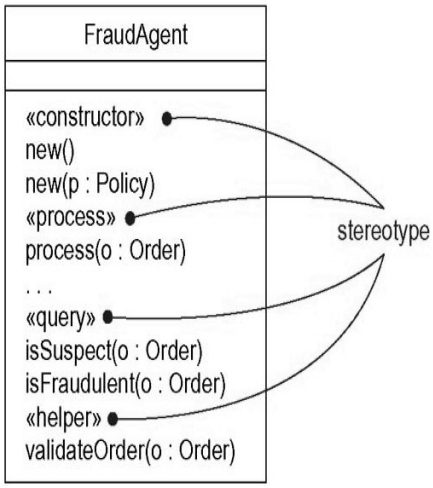
Operations

- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and a return type



Organizing Attributes and Operations:

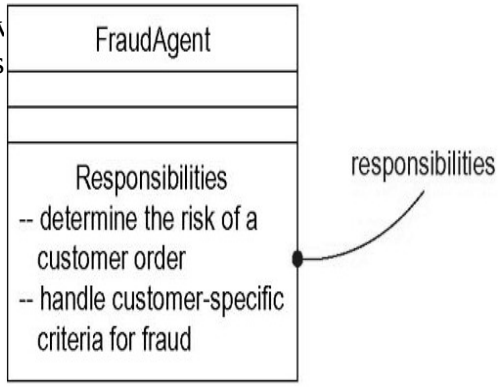
- No need to show every attribute and every operation at once.
- So we can elide(show only some (or) none of class attributes and operations) a class
- An empty compartment doesn't means that there are no operations it is just we didn't choose to show.
- We can explicitly specify that there are more attributes and are shown by ending each list with ellipsis ("...")
- To better organize long lists of attributes and operations, prefix each group using stereotypes.



Stereotypes for class features

Responsibilities

- A Responsibility is a contract (duty) or an obligation of a class. All object of class has some kind of behavior and state.
  Ex: Fraud agent - responsible for processing orders
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may hav nber of responsibilities
- Graphically, res e compartment at the bottom of the class icon



Other features:

- Attribute, operations and responsibilities are most common features are useful in building models of your classes.
- We could even specify other features such as visibility of individual attributes and operations, whether language is polymorphic (or) constant, exceptions that object of class might handle.

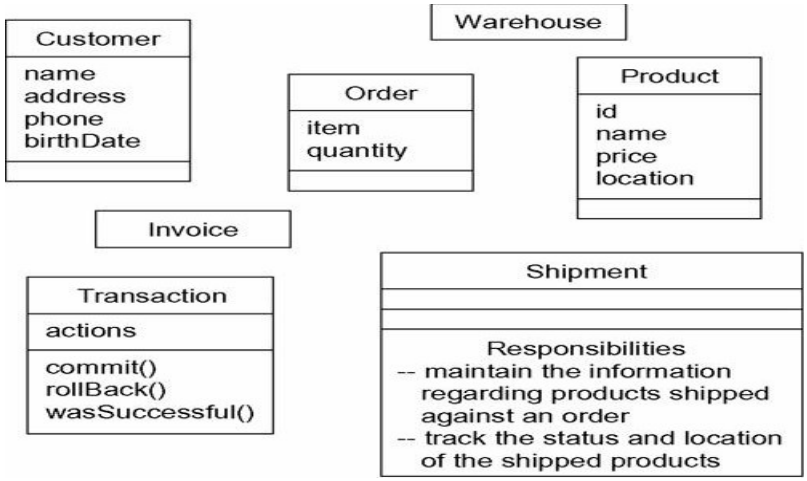
Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly used to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- Each of these abstractions is part of vocabulary of system

To model the vocabulary of a system

- Identify those **things** that users or implementers use to describe the problem or solution.
- For each abstraction, identify a **set of responsibilities** and there should be good responsibility to balance them among classes
- Provide the **attributes and operations** that are needed to carry out these responsibilities for each class.



Modeling the Distribution of Responsibilities in a System

- Once you start modeling, you should see your abstractions provide a balanced set of responsibilities.

To model the distribution of responsibilities in a system

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole,
  i) Split classes that have too many responsibilities into smaller abstractions
  ii) Collapse tiny classes that have trivial responsibilities into larger ones, and
  iii) Reallocate responsibilities so that each abstraction reasonably stands on its own.

- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities so that no class within collaboration does too much or too little.

Ex: Distribution of responsibilities among model, view and controller classes. Here all classes work together.

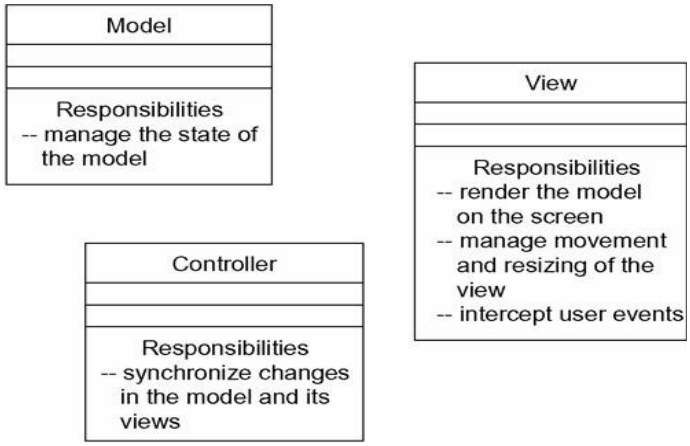


Fig. Modeling the distribution of responsibilities in a system

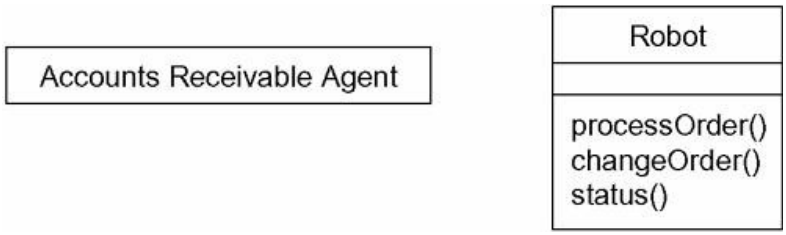
Modeling Nonsoftware Things

- Sometimes, the things you model may never have an analog in software
- For ex, who send invoices and robots that automatically package order for shipping from a warehouse which is a part of real system.
- Your application might not have any software that represents them

To model nonsoftware things

- Model the thing you are abstracting as a class.
- Create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

Ex:



Modeling Primitive Types

- Things you model may be drawn directly from the programming language you are using to implement a solution.
- Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types

To model primitive types

- Model the thing you are abstracting as a type or an enumeration, representing class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

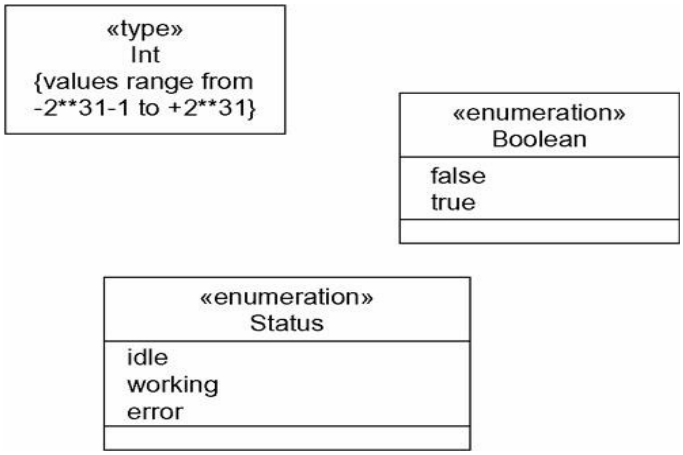


Fig. Modeling primitive types

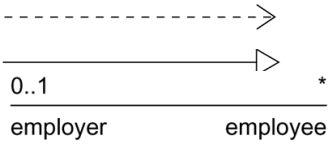
- Here things are represented just like classes but are explicitly marked via stereotypes
- Things like integer is represented by class Int, Boolean and states are modeled a enumerates with their individual value provided as attributes

Relationships

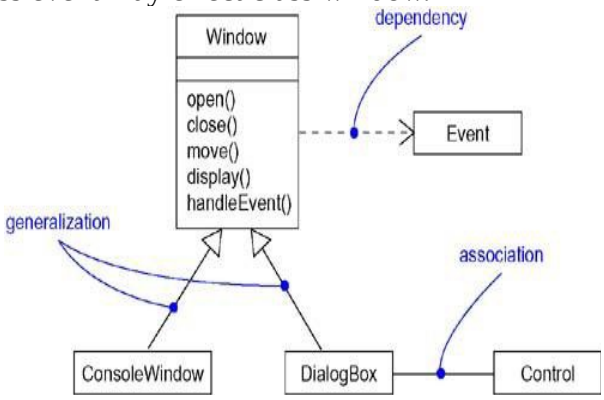
- In the UML, things can connect to one another using relationships.
- When a system is modeled we identify vocabulary of system. Then we observe how the elements are interconnected to one another.

Terms and Concepts:

- Relationship is a connection among things
- The relationship describe how things collaborate with each other
- A relationship illustrated as path with different types of lines to indicate kind of relationship
- There are three most important relationship are:
  - Dependencies
  - Generalizations
  - Associations

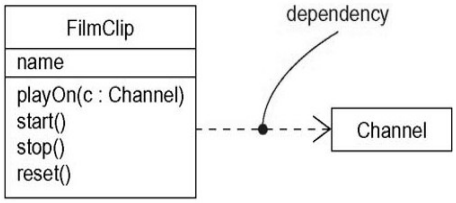


Ex: In the example any change in class event may effect class window.



Dependency

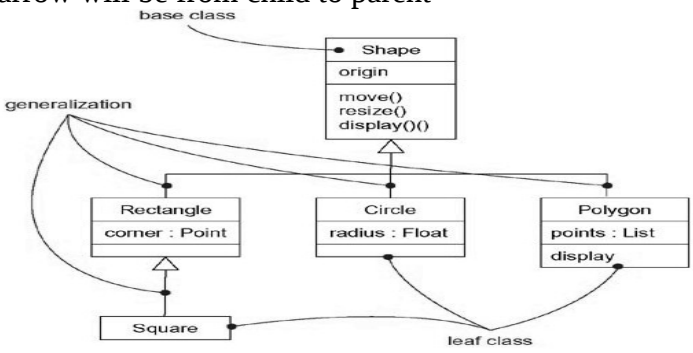
- A dependency is a **using** relationship that states that a change of one thing may affect another thing that uses it but not necessarily the reverse.
- Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.
- Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

Generalization (is - a - kind of")

- A generalization is a relationship between a general thing (called the super class or parent) and a more specific kind of that thing (called the subclass or child).
- Generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations
- A class may have zero, (or) one or more parents
- A class that has no parents and one (or) more children is called root class (or) base class
- A class with no children is called leaf class
- A class with single parent is called single inheritance
- A class with more parents is called multiple inheritance
- Generalization is used to show inheritance relation
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent
- The direction of arrow will be from child to parent

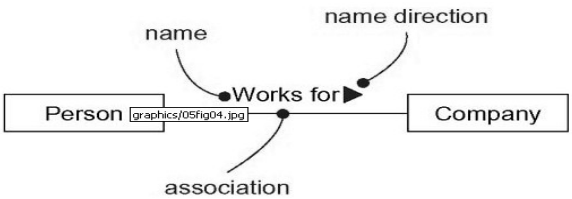


Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
- Association is connection between classes, we can navigate from object of one class and object of other class.
- An association that connects exactly two classes is called a binary association
- An associations that connect more than two classes; these are called n-ary associations
- Graphically, an association is rendered as a solid line connecting the same or different classes.
- Beyond this basic form, there are four adornments that apply to associations

a. Name

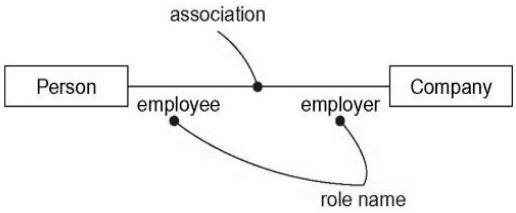
- An association can have a name, and you use that name to describe the nature of the relationship so that there is no ambiguity
- We can give a direction to name by providing a direction triangle those points in direction, we intend to read name



Association Names

b. Role

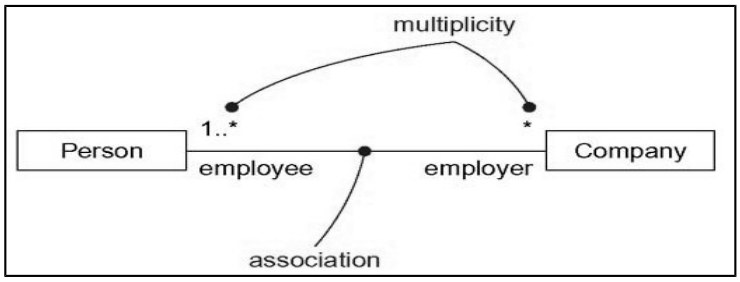
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

c. <u>Multiplicity</u>

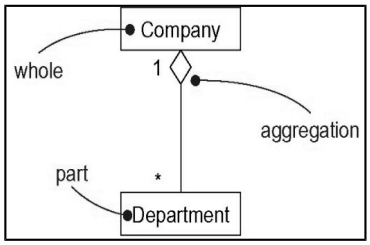
- It's important to state how many objects may be connected across an instance of an association. This count is known as multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..* or simply *), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

d. <u>Aggregation (or) "has-a"</u>

- Aggregation is a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").
- Aggregation is a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

<u>Common Modeling Techniques</u>

<u>Modeling Simple Dependencies</u>

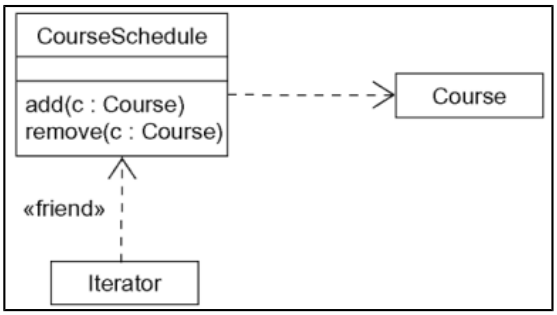
The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.

To model this dependencies

Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

The following figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university.

This figure shows a dependency from CourseSchedule to Course, because Course is used in both add and remove operations of CourseSchedule.
The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.



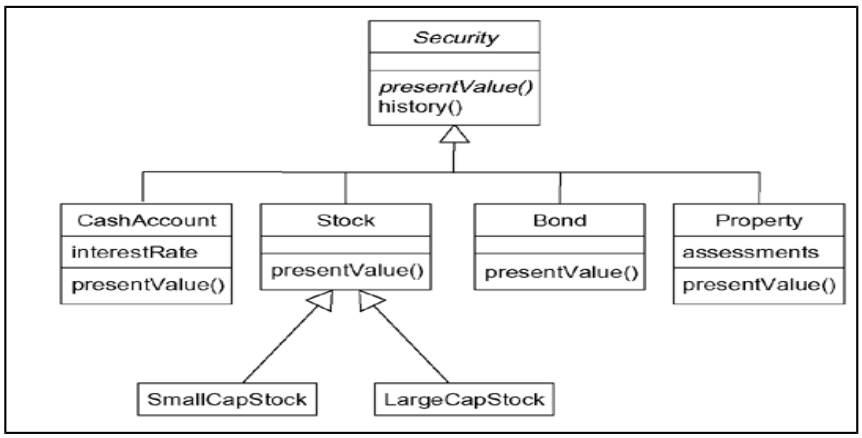
Dependency Relationships

Modeling Single Inheritance

When we model vocabulary of system you will often come across several classes. Classes may have similar behavior to other classes. So a better way is to model class from which specialized one inherits.

To model single inheritance relationship the following should be seen:

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Refine these common responsibilities, attributes, and operations to a more general class. If necessary or create a new class to which you can assign these
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



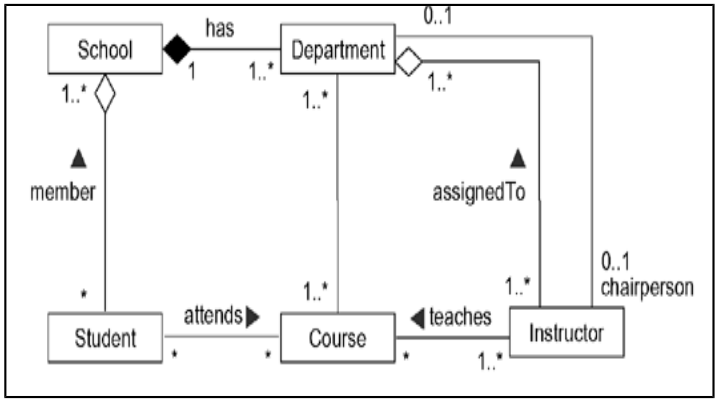
Inheritance Relationships

Modeling Structural Relationships

- Dependency and generalization relationships are one-sided.
- Associations are, bidirectional
- Given an association between two classes, you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

To model structural relationships

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association.
- For each of these associations, specify a multiplicity, as well as role names.
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole



Structural Relationships

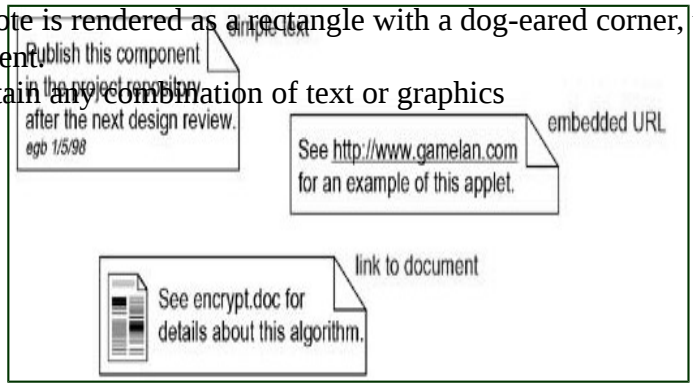
There's an association between Student and Course, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students. The relationships between School and the classes Student and Department are a bit different. Here you'll see aggregation relationships. A school has zero or more students, each student may be a registered member of one or more schools, a school has one or more departments, and each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that School is a whole and that Student and Department are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from their school. You'll also see that there are two associations between Department and Instructor. One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally, departments are at a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

Common Mechanisms

- Modeling is all about communication
- Usually in human languages, vocabulary extends itself as time passes, therefore dictionaries are printed everywhere.
- Similarly in UML when system is designed all concepts and terms can't be expressed then languages so some mechanisms are provided to cover details in the form of comments and constraints. So that we can better understand the system.

Note

- A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements.
- Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.
- A note may contain any combination of text or graphics



Notes

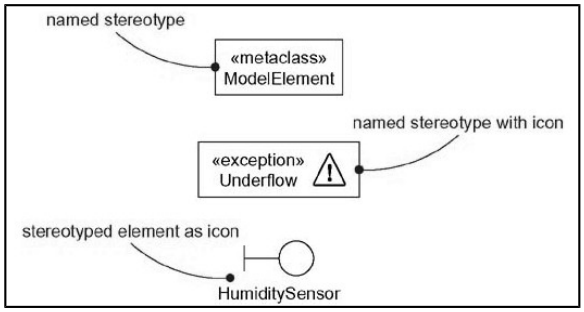
- Note has no semantic impact i.e., it do not alter meaning of model
- Notes are used to specify requirements review and explanations
- Note may contain any combination of text (or) graphics
- We can extend one note to another document

Other adornments:

- Adornments are textual or graphical item that are added to element basic notation and we need to visualize details from elements specification.(adding extra details to the basic notation)
- Example, basic notation for association is line but this may be adorned with details such as roles and multiplicity of each end.
- Most adornment are rendered by placing text near element or by adding graphic symbol to notation some time we can add extra compartment to provide the information.

Stereotypes (new building blocks)

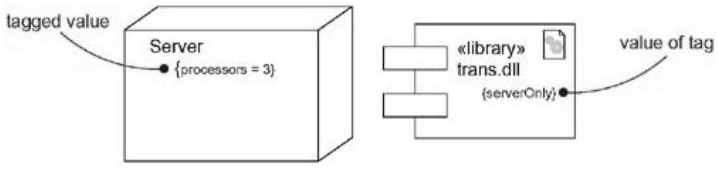
- A stereotype is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem.
- Graphically, a stereotype is rendered as a name enclosed by guillemets(<< >>) and placed above the name of another element



Stereotypes

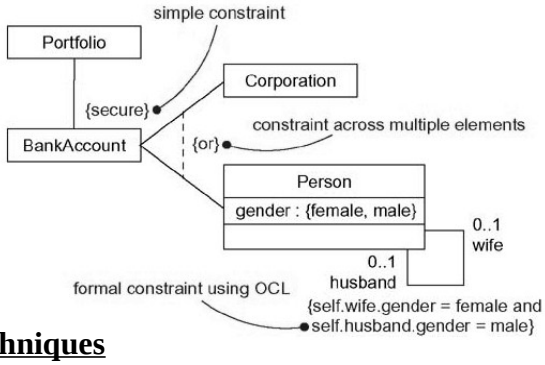
Tagged Values(new properties)

- Everything in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.
- A tagged value as metadata because its value applies to the element itself, not its instances.
- A tagged value is an extension of the properties of a UML element, allowing you to create new information in that element's specification.
- In its simplest form, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- That string includes a name (the tag), a separator (the symbol =), and a value (of the tag).



Constraints

- A constraint specifies conditions that must be held true for the model to be well-formed.
- A constraint is rendered as a string enclosed by brackets and placed near the associated element
- Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.



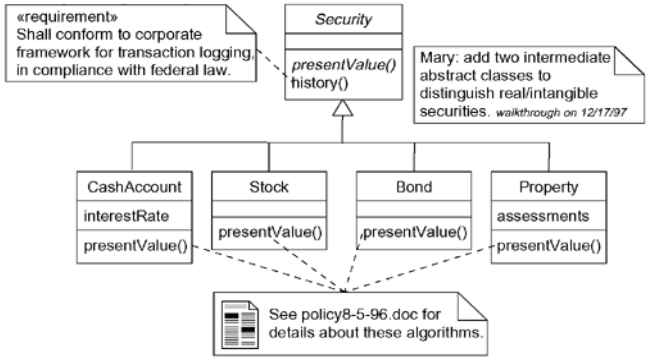
Common Modeling Techniques

Modeling Comments

The most common purpose for which you'll use notes is to write down free-form observations, reviews, or explanations.

To model a comment,

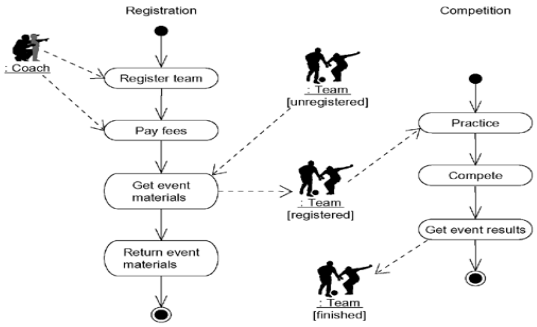
- Put your comment as text in a note and place it adjacent to the element to which it refers
- Remember that you can hide or make visible the elements of your model as you see fit.
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model



Modeling New Building Blocks

- The UML's building blocks—classes, interfaces, collaborations, components, nodes, associations, and so on—are generic enough to address most of the things you'll want to model.
- However, if you want to extend your modeling vocabulary or give distinctive visual cues to certain kinds of abstractions that often appear in your domain, you need to use stereotypes
- **To model new building blocks,**
  - Make sure there's no way to express what you want by using basic UML
  - If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model and define a new stereotype for that thing
  - Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.

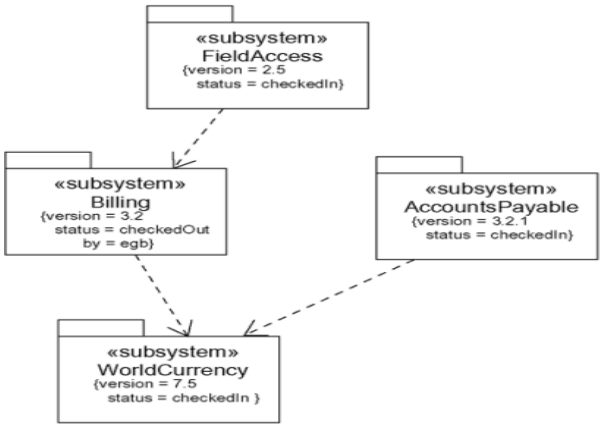
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype



Modeling New Building Blocks

Modeling New Properties

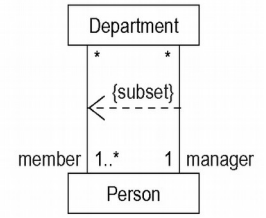
- The basic properties of the UML's building blocks—attributes and operations for classes, the contents of packages, and so on—are generic enough to address most of the things you'll want to model.
- However, if you want to extend the properties of these basic building blocks, you need to use tagged values.
- To model new properties,
    1. First, make sure there's no way to express what you want by using basic UML
    2. If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype.



Modeling New Properties

Modeling New Semantics

- To express new semantics or that you need to modify the UML's rules, then you need to write a constraint.
- **To model new semantics,**
    - First, make sure there's not already a way to express what you want by using basic UML
    - If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers
    - If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



Modeling New Semantics

Diagrams

- When you view a software system from any perspective using the UML, you use diagrams to organize the elements of interest.
- The UML defines nine kinds of diagrams, which you can mix and match to assemble each view.
- Not only the nine kinds of diagrams you could create your own diagrams.
- You'll use the UML's diagrams in two basic ways:
  - to specify models from which you'll construct an executable system (forward engineering)
  - and to reconstruct models from parts of an executable system (reverse engineering).

System

- A system is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints

Subsystem

- A subsystem is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.

Model

- A model is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture

View

- View is a projection into the organization and structure of a system's model, focused on one aspect of that system

Diagram

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- A diagram is just a graphical projection into the elements that make up a system
- Each diagram provides a view into the elements that make up the system
- Typically, you'll view the static parts of a system using one of the four following diagrams.
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram
- You'll often use five additional diagrams to view the dynamic parts of a system.
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

The UML defines these nine kinds of diagrams.

- Every diagram you create will most likely be one of these nine or occasionally of another kind.
- Every diagram must have a name that's unique in its context so that you can refer to a specific diagram and distinguish one from another.
- You can show any combination of elements in the UML in the same diagram. For example, you might show both classes and objects in the same diagram.

Structural Diagrams

- The UML's four structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.

- The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.
  - Class diagram : Classes, interfaces, and collaborations
  - Object diagram : Objects
  - Component diagram : Components
  - Deployment diagram : Nodes

Class Diagram

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams are used to illustrate the static design view of a system.
- Class diagrams that include active classes are used to address the static process view of a system.
- Class diagrams are the most common diagram found in modeling object-oriented systems.

Object Diagram

- An object diagram shows a set of objects and their relationships.
- Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.
- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

- A component diagram shows a set of components and their relationships.
- We use component diagrams to illustrate the static implementation view of a system.
- Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- A deployment diagram shows a set of nodes and their relationships.
- We use deployment diagrams to illustrate the static deployment view of architecture.
- Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

- The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.
  - Use case diagram : Organizes the behaviors of the system
  - Sequence diagram : Focused on the time ordering of messages
  - Collaboration diagram : Focused on the structural organization of objects that send and receive messages
  - Statechart diagram : Focused on the changing state of a system driven by events
  - Activity diagram : Focused on the flow of control from activity to activity

Use Case Diagram

- A use case diagram shows a set of use cases and actors and their relationships.
- We apply use case diagrams to illustrate the static use case view of a system.
- Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Sequence Diagram

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.
- We use sequence diagrams to illustrate the dynamic view of a system.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Collaboration Diagram

- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.
- We use collaboration diagrams to illustrate the dynamic view of a system.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Sequence and collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.

Statechart Diagram

- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
- Statechart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.
- We use statechart diagrams to illustrate the dynamic view of a system.
- They are especially important in modeling the behavior of an interface, class, or collaboration.

Activity Diagram

- Activity diagrams emphasize the flow of control among objects.
- An activity diagram shows the flow from activity to activity within a system.
- An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.
- We use activity diagrams to illustrate the dynamic view of a system.
- Activity diagrams are especially important in modeling the function of a system.

Common Modeling Techniques

Modeling Different Views of a System

- When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions
- To model a system from different views,
  - Decide which views you need to best express the architecture of your system
  - For each of these views, decide which part you need to create to capture the essential details of that view.
  - As part of your process planning, decide which of these diagrams you want to put under some sort of formal or semi-formal control.
- For example, if you are modeling a simple monolithic(Single) application that runs on a single machine, you might need only the following handful of diagrams

• Use case view	:	Use case diagrams
• Design view	:	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
• Process view	:	None required
• Implementation view	:	None required

- If yours is a reactive system or if it focuses on process flow, you'll probably want to include statechart diagrams and activity diagrams, respectively, to model your system's behavior.
- Similarly, if yours is a client/server system, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.
- Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

• Use case view	:	Use case diagrams Activity diagrams (for behavioral modeling)
• Design view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling) * Statechart diagrams (for behavioral modeling)
• Process view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling)
• Implementation view	:	Component diagram
• Deployment view	:	Deployment diagrams

Modeling Different Levels of Abstraction

- Not only do you need to view a system from several angles, you'll also find people involved in development who need the same view of the system but at different levels of abstraction
- Basically, there are two ways to model a system at different levels of abstraction:
  - By presenting diagrams with different levels of detail against the same model
  - By creating models at different levels of abstraction with diagrams that trace from one model to another.
- To model a system at different levels of abstraction by presenting diagrams with different levels of detail,
  - Consider the needs of your readers, and start with a given model
  - If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction which means that they'll need to reveal a lot of detail
  - If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction which means that they'll hide a lot of detail
  - Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

Building blocks and relationships:

- Hide those that are not relevant to the intent of your diagram or the needs of your reader.

Adornments:

- Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

Flow:

- In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.

Stereotypes:

- In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

- The main advantage of this approach is that you are always modeling from a common semantic repository.

- The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

To model a system at different levels of abstraction by creating models at different levels of abstraction.

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

Use cases and their realization:

Use cases in a use case model will trace to collaborations in a design model.

Collaborations and their realization:

Collaborations will trace to a society of classes that work together to carry out the collaboration.

Components and their design:

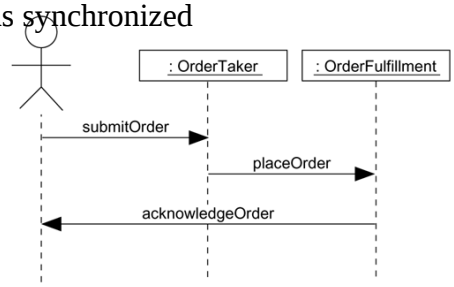
Components in an implementation model will trace to the elements in a design model.

Nodes and their components:

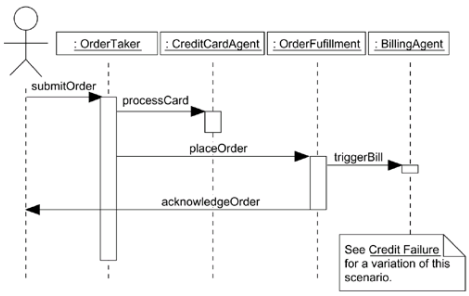
Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models.

The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized



Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

* Both of these diagrams work against the same model, but at different levels of detail.

Modeling Complex Views

- To model complex views,
  - First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.
  - If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher level collaborations, then render only those packages or collaborations in your diagram.
  - If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
  - If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Advanced Structural Modeling

- A relationship is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations.
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different relationships.

Dependency

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line.
- A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships.
- There are 17 such stereotypes, all of which can be organized into six groups.

- First, there are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1	**bind**	Specifies that the source instantiates the target template using the given actual parameters
2	**derive**	Specifies that the source may be computed from the target
3	**friend**	Specifies that the source is given special visibility into the target
4	**instanceOf**	Specifies that the source object is an instance of the target classifier
5	**instantiate**	Specifies that the source creates instances of the target
6	**powertype**	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
7	**refine**	Specifies that the source is at a finer degree of abstraction than the target
8	**use**	Specifies that the semantics of the source element depends on the semantics of the public part of the target

bind:

bind includes a list of actual arguments that map to the formal arguments of the template.

derive

When you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual.

friend

When you want to model relationships such as found with C++ friend classes.

instanceOf

When you want to model the relationship between a class and an object in the same diagram, or between a class and its metaclass.

instantiate

When you want to specify which element creates objects of another.

powertype

When you want to model classes that cover other classes, such as you'll find when modeling databases

refine

When you want to model classes that are essentially the same but at different levels of abstraction.

use

When you want to explicitly mark a dependency as a using relationship

* There are two stereotypes that apply to dependency relationships among packages.

9	**access**	Specifies that the source package is granted the right to reference the elements of the target package
10	**import**	A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

* Two stereotypes apply to dependency relationships among use cases:

11	**extend**	Specifies that the target use case extends the behavior of the source
12	**include**	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

* There are three stereotypes when modeling interactions among objects.

13	**become**	Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
14	**call**	Specifies that the source operation invokes the target operation
15	**copy**	Specifies that the target object is an exact, but independent, copy of the source

* We'll use become and copy when you want to show the role, state, or attribute value of one object at different points in time or space.

* You'll use call when you want to model the calling dependencies among operations.

* One stereotype you'll encounter in the context of state machines is

16	**?send**	Specifies that the source operation sends the target event

* We'll use send when you want to model an operation dispatching a given event to a target object.
* The send dependency in effect lets you tie independent state machines together.

Finally, one stereotype that you'll encounter in the context of organizing the elements of your system into subsystems and models is

17	**?trace**	Specifies that the target is an historical ancestor of the source

* We'll use trace when you want to model the relationships among elements in different models

Generalization

A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child).

In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,

The UML defines one stereotype and four constraints that may be applied to generalization relationships.

1	**? implementation**	Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability

?implementation

- We'll use implementation when you want to model private inheritance, such as found in C++.

Next, there are four standard constraints that apply to generalization relationships

1	**complete**	Specifies that all children in the generalization have been specified in the model and that no additional children are permitted
2	**incomplete**	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted
3	**disjoint**	Specifies that objects of the parent may have no more than one of the children as a type
4	**overlapping**	Specifies that objects of the parent may have more than one of the children as a type

complete

- We'll use the complete constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy);

incomplete

- We'll use incomplete to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show everything in the model).

Disjoint & overlapping

- These two constraints apply only in the context of multiple inheritance.
- We'll use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another.

We use associations when you want to show structural relationships.

There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

For advanced uses, there are a number of other properties you can use to model subtle details, such as

Navigation

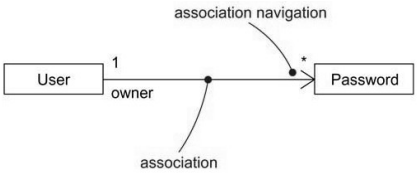
Vision

Qualification

Various flavors of aggregation.

Navigation

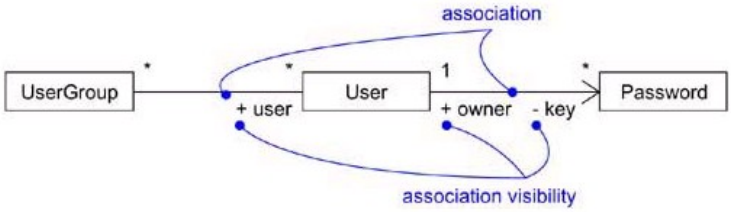
- Unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.
- However, there are some circumstances in which you'll want to limit navigation to just one direction.



Navigation

Visibility

- Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation.
- However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.
- In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name the visibility of a role is public.
- Private visibility indicates that objects at that end are not accessible to any objects outside the association.
- Protected visibility indicates that objects at that end are not accessible to any objects outside the association, e



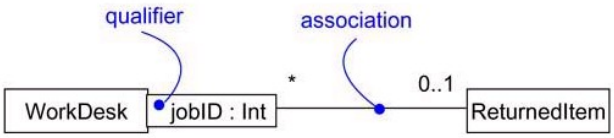
Visibility

Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end?

In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association.

You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the re



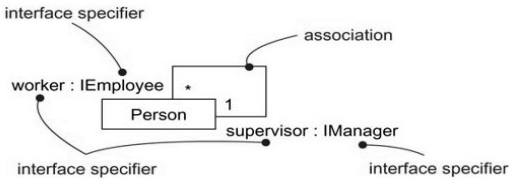
Qualification

Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or a component

Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class.

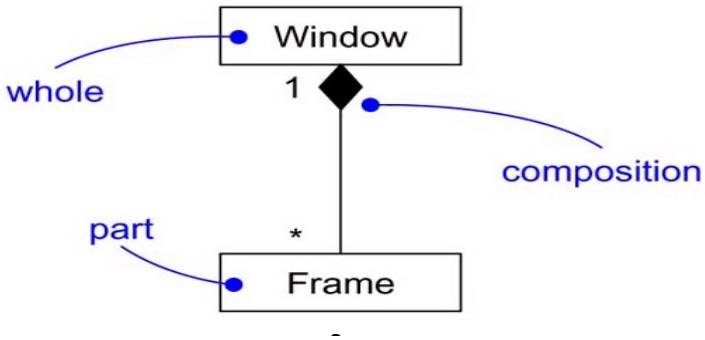
However, in the context of an association with another target class, a source class may choose to present only part of its face to the world



- a Person class may realize many interfaces: IManager, IEmployee, IOfficer, and so on
- you can model the relationship between a supervisor and her workers with a one-to-many
- association, explicitly labeling the roles of this association as supervisor and worker
- In the context of this association, a Person in the role of supervisor presents only the IManager face to the worker; a Person in the role of worker presents only the IEmployee face to the supervisor. As the figure shows, you can explicitly show the type of role using the syntax rolename : iname, where iname is some interface of the other classifier.

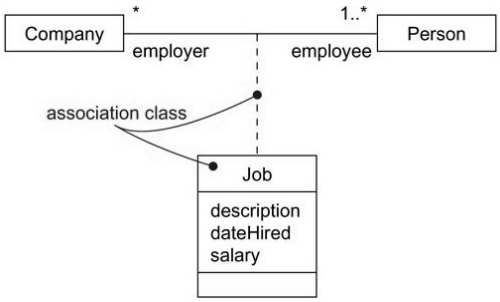
Composition

* Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."
* Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.
* Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.
* This means that, in a composite aggregation, an object may be a part of only one composite at a time
* In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts



Association Classes

* In an association between two classes, the association itself might have properties.
* An association class can be seen as an association that also has class properties, or as a class that also has association properties.
* We render an association class as a class symbol attached by a dashed line to an association



Association Classes

Constraints

* UML defines five constraints that may be applied to association relationships.

1	**implicit**	Specifies that the relationship is not manifest but, rather, is only conceptual
2	**ordered**	Specifies that the set of objects at one end of an association are in an explicit order
3	**changeable**	Links between objects may be added, removed, and changed freely
4	**addOnly**	New links may be added from an object on the opposite end of the association
5	**frozen**	A link, once added from an object on the opposite end of the association, may not be modified or deleted

Implicit

* If you have an association between two base classes, you can specify that same association between two children of those base classes

* You can specify that the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

Ordered

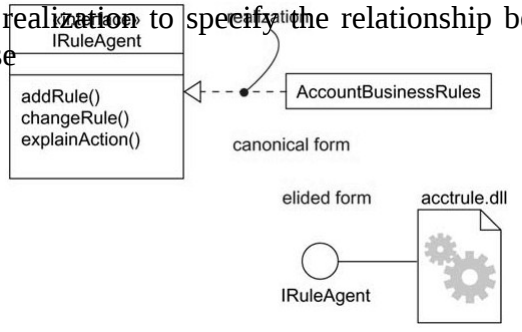
* For example, in a User/Password association, the Passwords associated with the User might be kept in a least-recently used order, and would be marked as ordered.

Finally, there is one constraint for managing related sets of associations:

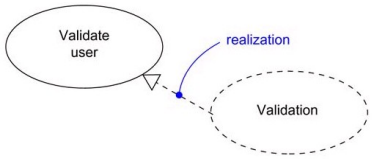
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

1. Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship.
2. A realization is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
3. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
4. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations
5. Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it
6. You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case



Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard
3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled
4. When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.

- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.

- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.

- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.

- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.

- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, type and roles

Interface

- An interface is a collection of operations that are used to specify a service of a class or a component

type

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

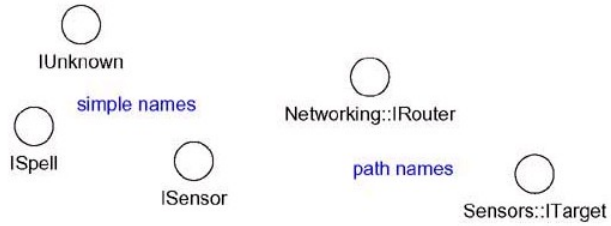
role

- A role is the behavior of an entity participating in a particular context.

an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

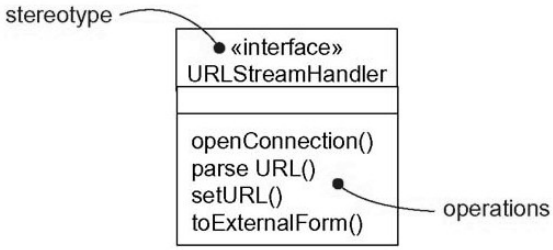
- Every interface must have a name that distinguishes it from other interfaces.
- A name is a textual string. That name alone is known as a simple name;
- A path name is the interface name prefixed by the name of the package



Simple and Path Names

Operations

- An interface is a named collection of operations used to specify a service of a class or of a component.
- Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation
- These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- You can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties

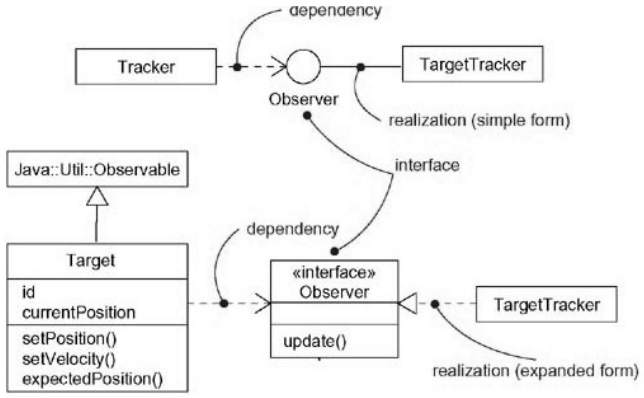


Operations

Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces
- We can show that an element realizes an interface in two ways.
  - First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.

- Second, you can use the expanded form in which you render an interface as a stereotyped class,
  which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.



Relationships

Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.

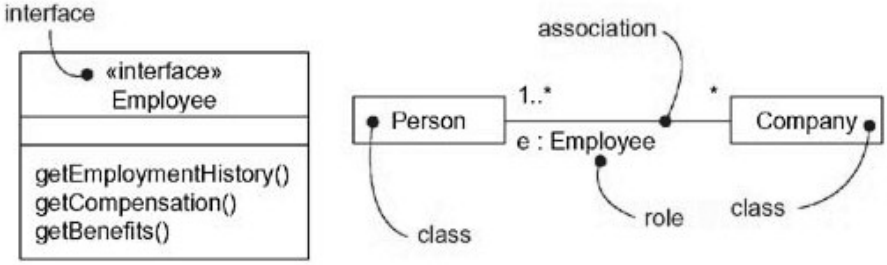
- First, you may attach pre- and post conditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.

- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.

- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

Types and Roles

A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.
For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.
an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.
In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.



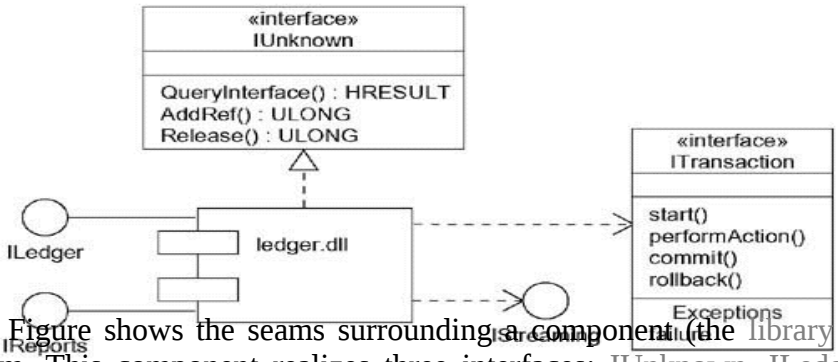
Roles

- A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.
- If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype type
- Type is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not.

Common Modeling Techniques

➢ Modeling the Seams in a System

- The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.
- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,



- The above Figure shows the seams surrounding a component (the library ledger.dll) drawn from a financial system. This component realizes three interfaces: IUnknown, ILedger, and IReports. In this diagram, IUnknown is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by ledger.dll and are exported to other components for them to build on.
- As this diagram also shows, ledger.dll imports two interfaces, IStreaming and ITransaction, the latter of which is shown in its expanded form. These two interfaces are required by the ledger.dll component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces.
- By identifying interfaces such as ITransaction, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

➢ Modeling Static and Dynamic Typeseling Static and Dynamic Types

- Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created.
- Even so, that object will likely play different roles over time.
- Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow,

➢ To model a dynamic type

o Specify the different possible types of that object by rendering each type as a class stereotyped as
type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).

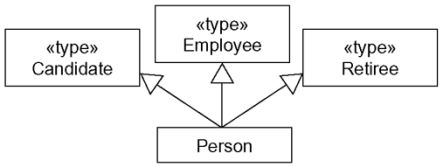
o Model all the roles of the object may take on at any point in time. You can do so in two ways:

1.) First, in a class diagram, explicitly type each role that the class plays in its association with Other classes. Doing this specifies the face instances of that class put on in the context of the associated object.

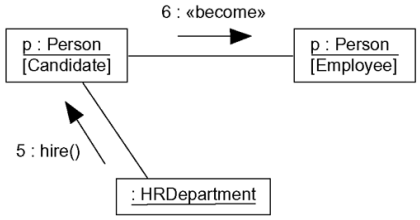
2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

o In an interaction diagram, properly render each instance of the dynamically typed class. Display
the role of the instance in brackets below the object's name.

o To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.



Modeling Static Types



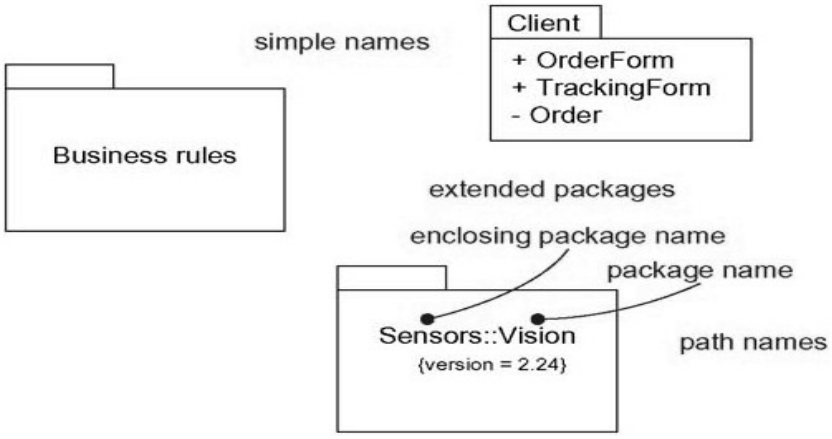
Modeling Dynamic Types

Package

"A package is a general-purpose mechanism for organizing elements into groups." Graphically, a package is rendered as a tabbed folder.

Names

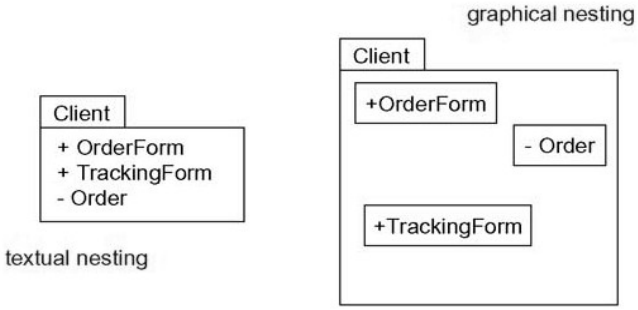
- Every package must have a name that distinguishes it from other packages. A name is a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



Simple and Extended Package

Owned Elements

A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.
Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.
Elements of different kinds may have the same name within a package. Thus, you can have a class named Timer, as well as a component named Timer, within the same package.
Packages may own other packages. This means that it's possible to decompose your models hierarchically.
We can explicitly show the contents of a package either textually or graphically.



Owned Elements

Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.
Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package.
Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.
We specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol.

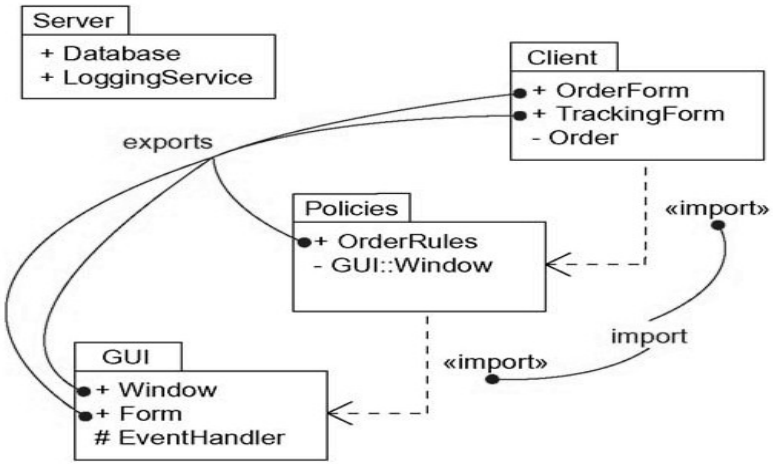
Importing and Exporting

In the UML, you model an import relationship as a dependency adorned with the stereotype import
Actually, two stereotypes apply here—import and access— and both specify that the source package has access to the contents of the target.

Import adds the contents of the target to the source's namespace

Access does not add the contents of the target

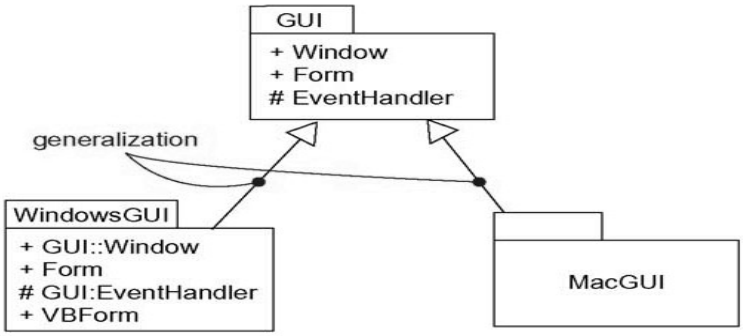
The public parts of a package are called its exports.
The parts that one package exports are visible only to the contents of those packages that explicitly import the package.
Import and access dependencies are not transitive



Importing and Exporting

Generalization

There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages
Generalization among packages is very much like generalization among classes
Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as Windows GUI) can be used anywhere a more general package (such as GUI) is used



Generalization among Packages

Standard Elements

- All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).
- The UML defines five standard stereotypes that apply to packages

1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents of another package
4. subsystem	Specifies a package representing an independent part of the entire system being modeled
5. system	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes

Common Modeling Techniques

Modeling Groups of Elements

- The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.
- There is one important distinction between classes and packages:

- Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);
- Classes do have identity (classes have instances, which are elements of a running system).

To model groups of elements,

- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies
- In the case of families of packages, connect specialized packages to their more general part via generalizations

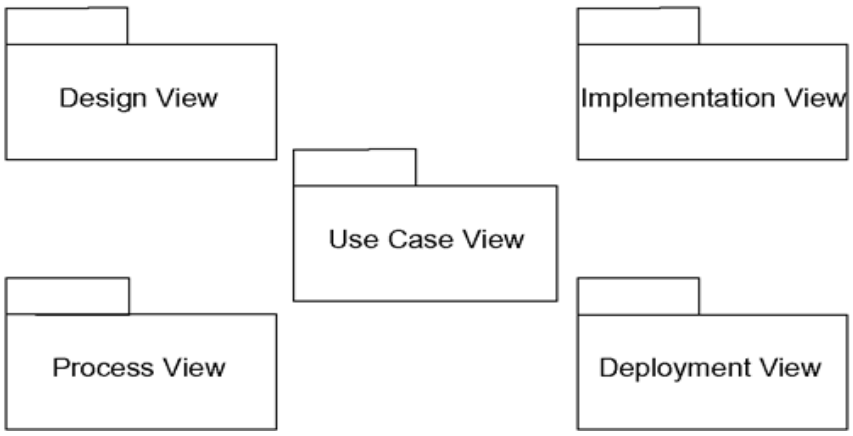
Modeling Architectural Views

- We can use packages to model the views of an architecture.
- Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
- This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
- Second, these packages own all the abstractions germane to that view.(Implementation view)
- To model architectural views,
- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.

- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.

- As necessary, further group these elements into their own packages.

- There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.



Modeling Architectural Views