UNIT-2
Interfaces, type and roles

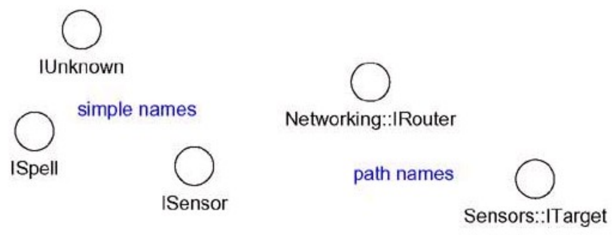
Interface
- An interface is a collection of operations that are used to specify a service of a class or a component

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.

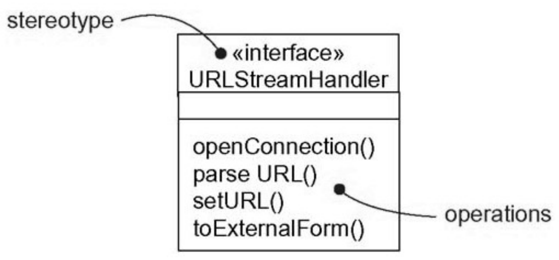
an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

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



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

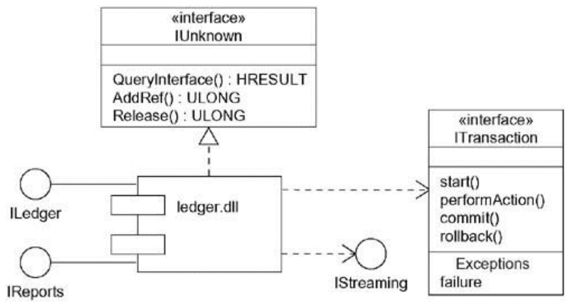
- We can show that an element realizes an interface in two ways.
  - First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.

  - 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.

<u>Common Modeling Techniques</u>

➢ **<u>Modeling the Seams in a System</u>**

- 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.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
- Package logically related sets of these operations and signals as interfaces.
- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.

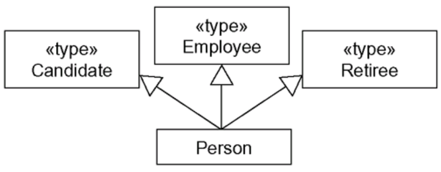


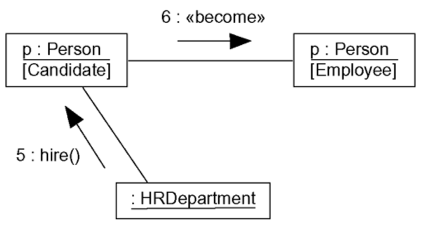
➢ **<u>Modeling Static and Dynamic Typeseling Static and Dynamic Types</u>**

- <u>To model a dynamic type</u>
- 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).
- Model all the roles the 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.

- 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.
- 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.





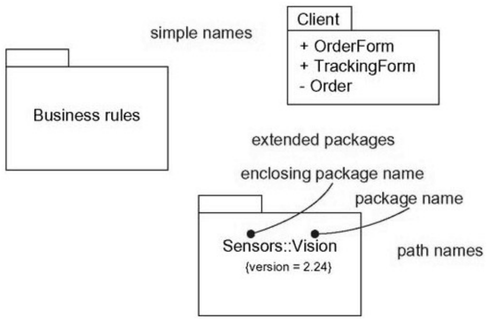
Modeling Static Types **Modeling Dynamic Types**

Package

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

Names

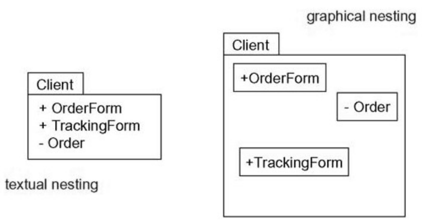
- Every package must have a name that distinguishes it from other packages. A name is a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with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



Simple and Extended Package

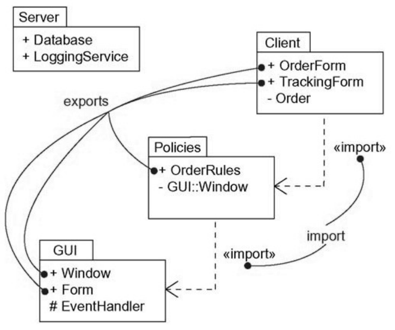
Owned Elements

A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.
We can explicitly show the contents of a package either textually or graphically.

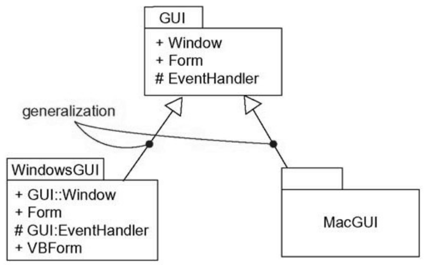


Visibility
You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.
Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's 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



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 WindowsGUI) can be used anywhere a more general package (such as



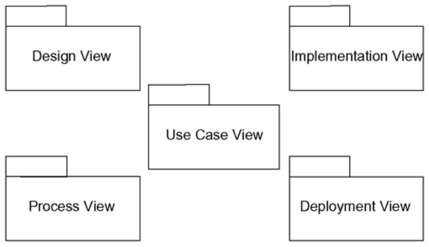
Generalization Among Packages

Common Modeling Techniques
1. Modeling 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

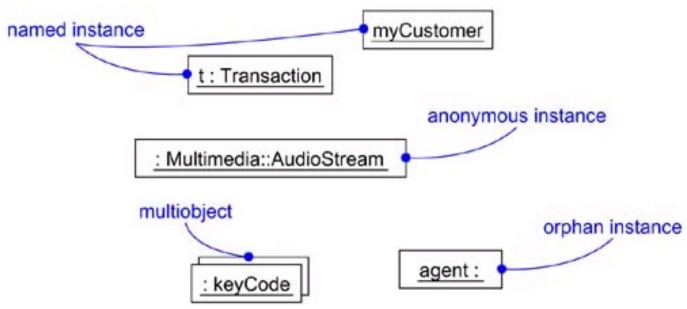
<u>2. Modeling Architectural Views</u>

- 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.



Instances

- An instance is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations.
- Graphically, an instance is rendered by underlining its name.



<u>Named, Anonymous, Multiple, and Orphan Instances</u>

<u>Names</u>

- Every instance must have a name that distinguishes it from other instances within its context.
- Typically, an object lives within the context of an operation, a component, or a node.

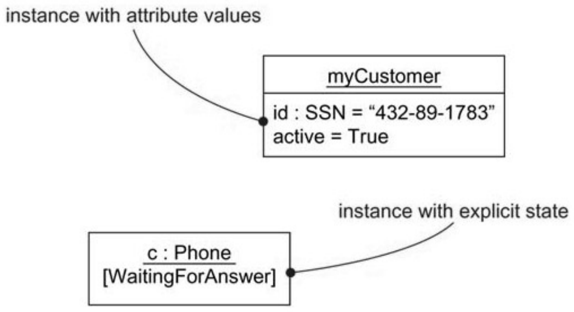
- A name is a textual string. That name alone is known as a simple name. or it may be a path name.

Operations

- The operations you can perform on an object are declared in the object's abstraction
- For example, if the class Transaction defines the operation commit, then given the instance t : Transaction, you can write expressions such as t.commit()

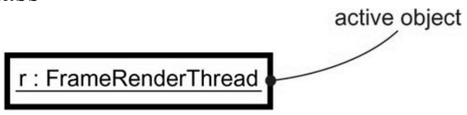
State

An object also has state. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space. When you operate on an object, you typically change its state;
when you query an object, you don't change its state



Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active from those that are passive. You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class



Active Objects

Stereotyped Objects

The UML defines two standard stereotypes that apply to the dependency relationships among objects and among classes:

1. instanceOf	Specifies that the client object is an instance of the supplier classifier
2. instantiate	Specifies that the client class creates instances of the supplier class

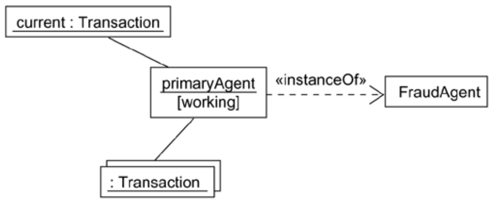
There are also two stereotypes related to objects that apply to messages and transitions:

1. become	Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. copy	Specifies that the client object is an exact but independent copy of the supplier

Common Modeling Techniques

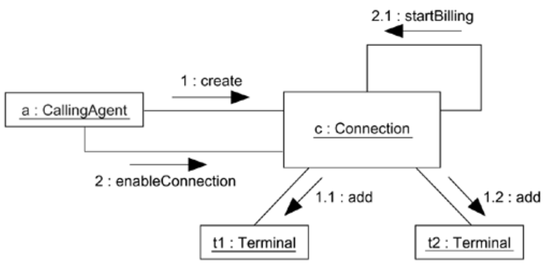
Modeling Concrete Instances

- When you model concrete instances, you are in effect visualizing things that live in the real world
- Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.



Modeling Prototypical Instances

- Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects
- Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the properties of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an interaction diagram or an activity diagram.


2.1 : startBilling
1 : create
a : CallingAgent
c : Connection
2 : enableConnection
1.1 : add
1.2 : add
t1 : Terminal
t2 : Terminal