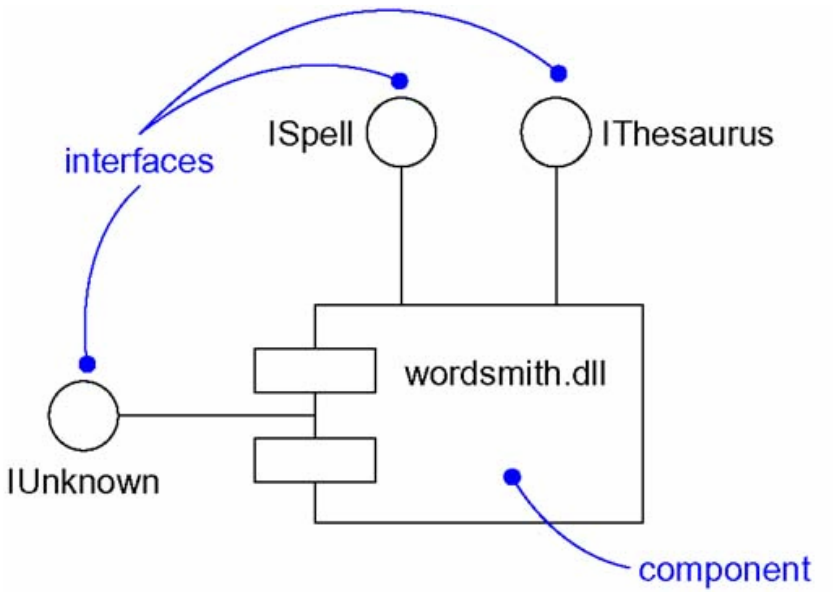
UML (Unified Modeling Language)

6 Interfaces ,Packages, Instances, Object Diagrams

Software Institute of
Nanjing University

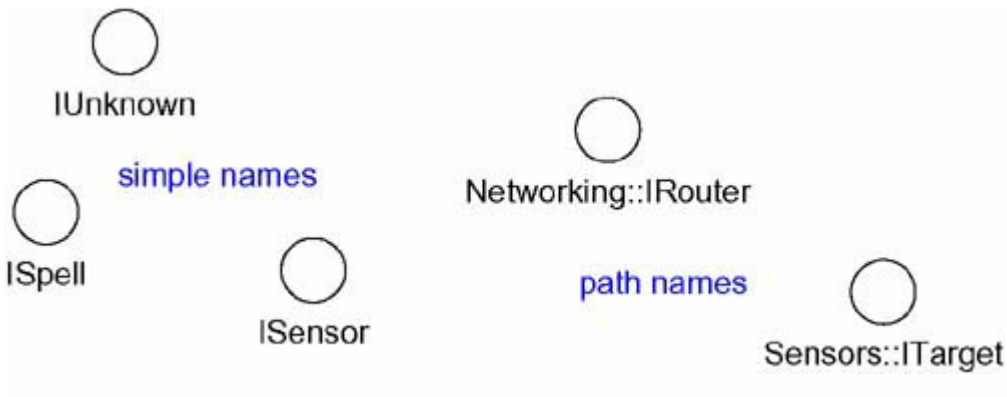
1. Interface

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



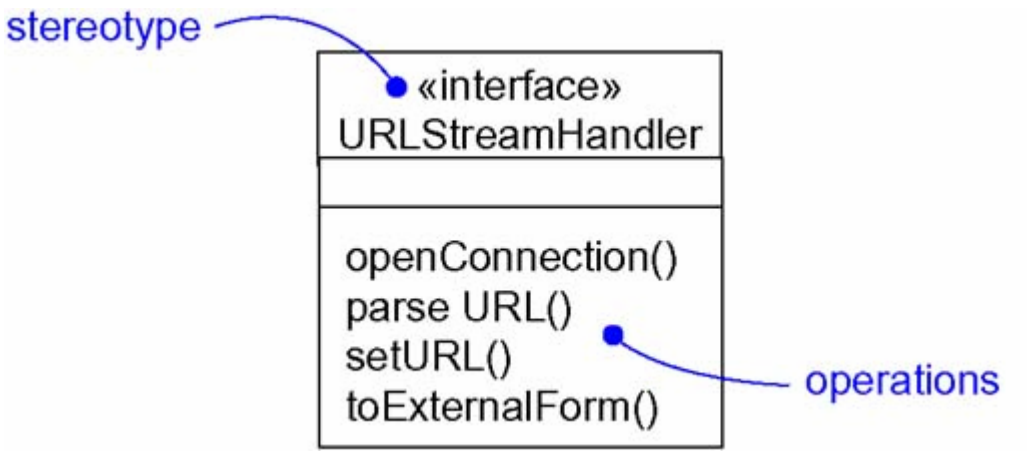
1.1 Names

- 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 in which that interface lives.



1.2 Operations

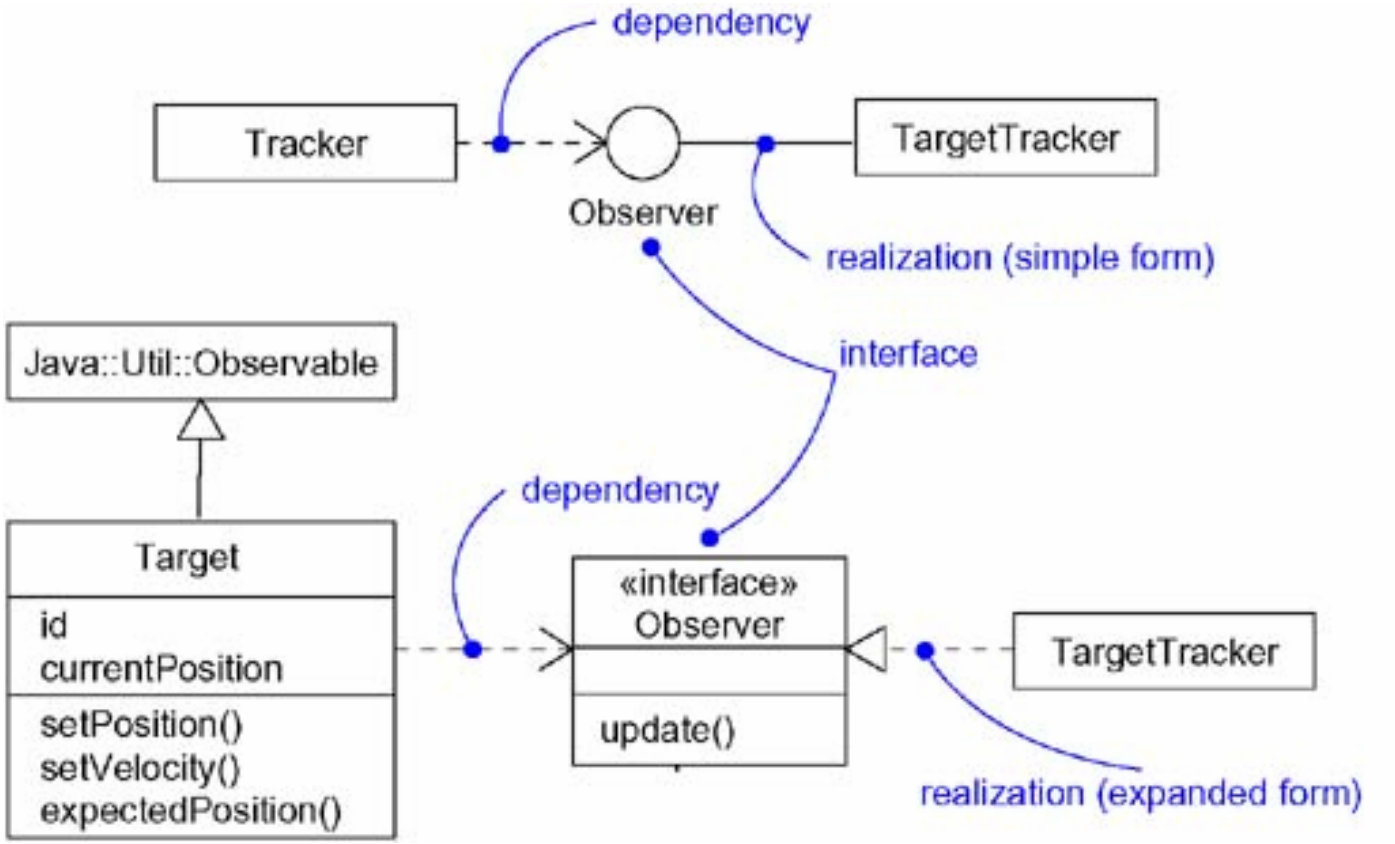
- An interface is a named collection of operations used to specify a service of a class or of a component.



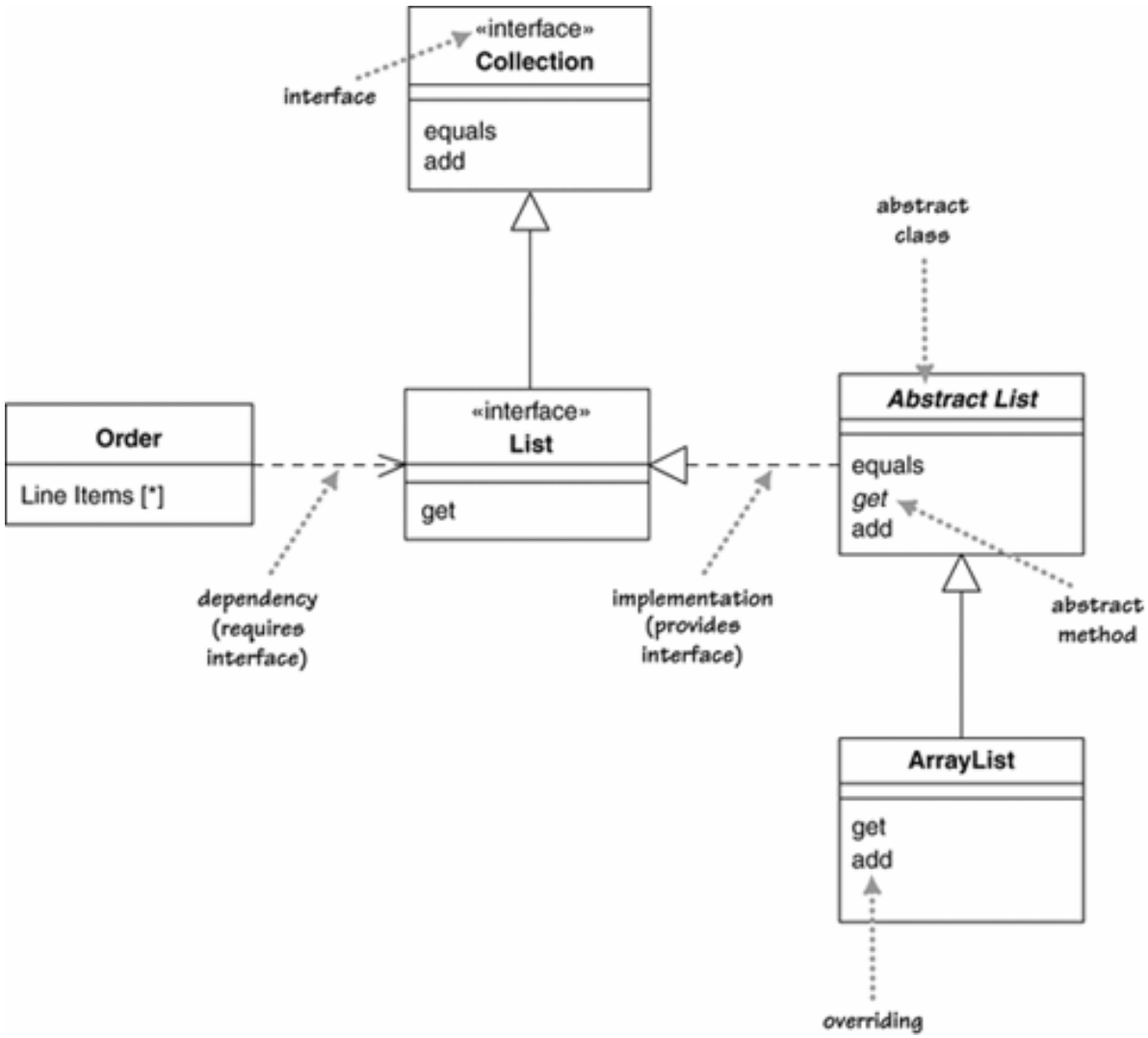
1.3 Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships.
- In addition, an interface may participate in realization relationships.
  - Realization is a semantic relationship between two classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.

1.3 Relationships (cont')

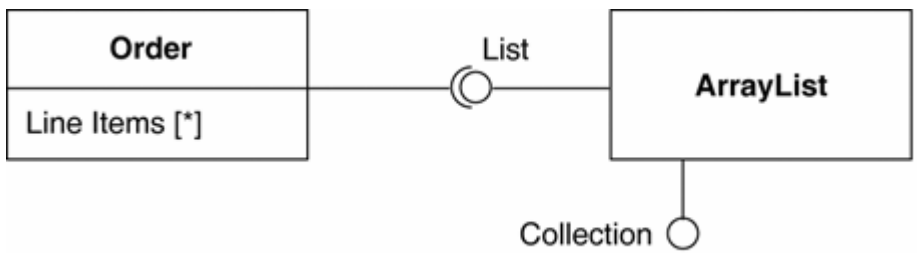


1.3 Relationships (cont')



1.3 Relationships (cont')

- UML 2



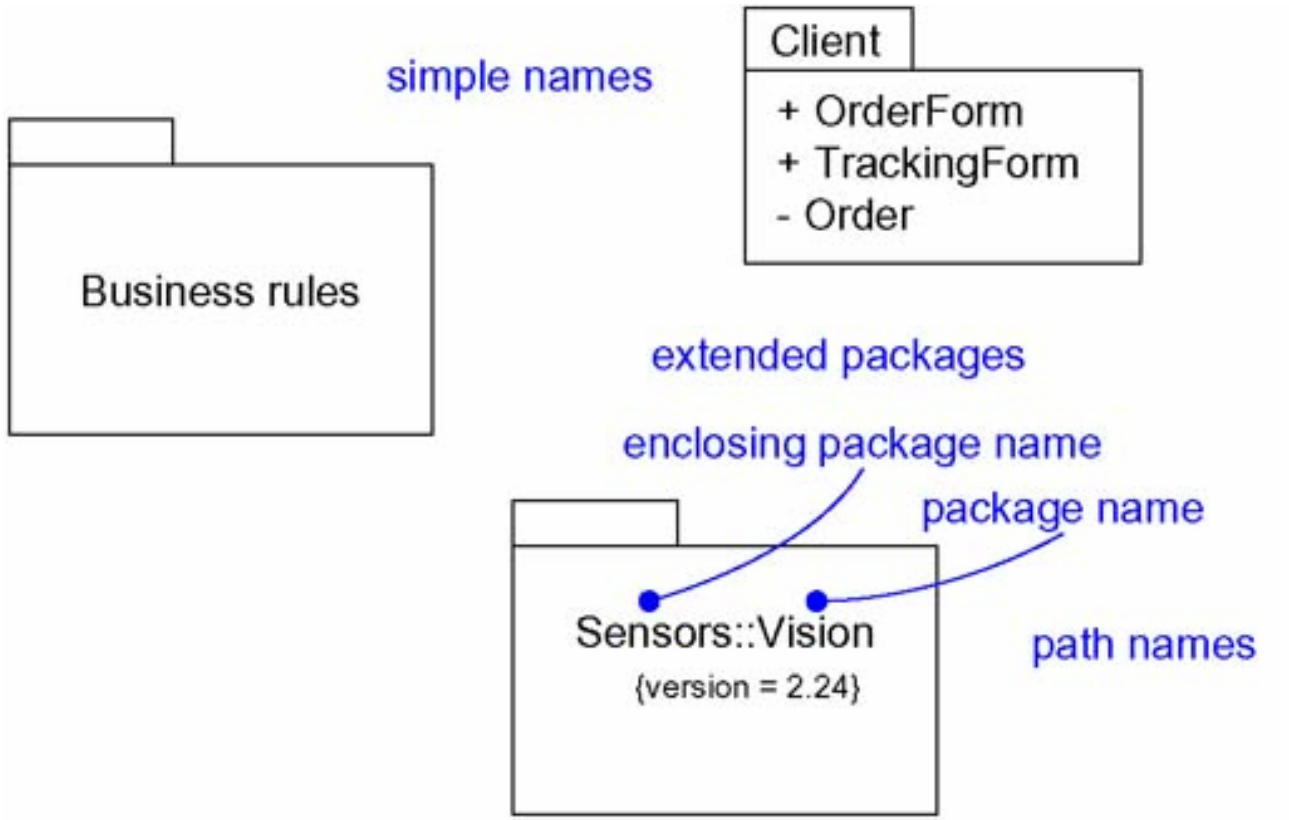
2. Packages

- A *package* is a general-purpose mechanism for organizing elements into groups. Graphically, a package is rendered as a tabbed folder.
- Well-designed packages group elements that are semantically close and that tend to change together. Well-structured packages are therefore loosely coupled and very cohesive, with tightly controlled access to the package's contents.

2.1 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.

2.1 Names (cont')



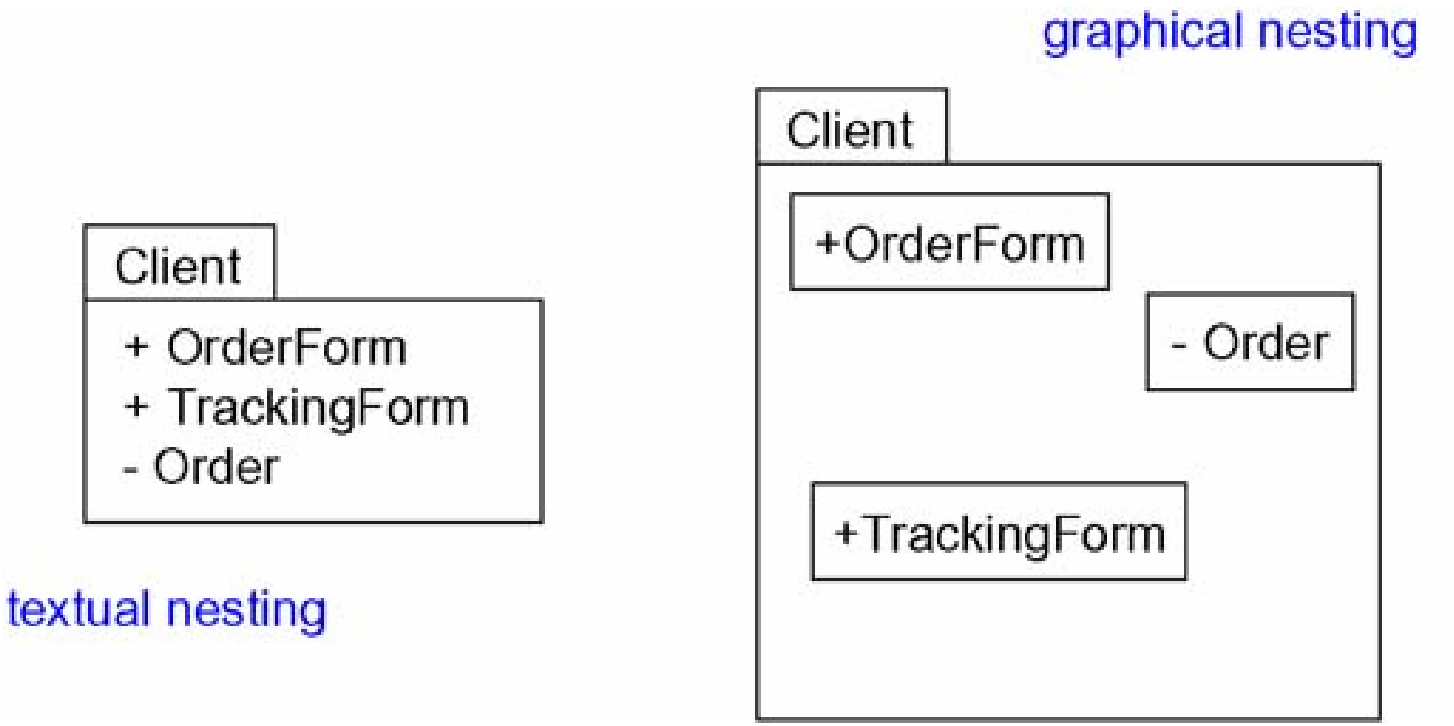
2.2 Owned Elements

- 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.

2.2 Owned Elements (cont')

- A package forms a namespace, which means that elements of the same kind must be named uniquely within the context of its enclosing package.
- Elements of different kinds may have the same name within a package.
- Packages may own other packages. This means that it's possible to decompose your models hierarchically.

2.2 Owned Elements (cont')



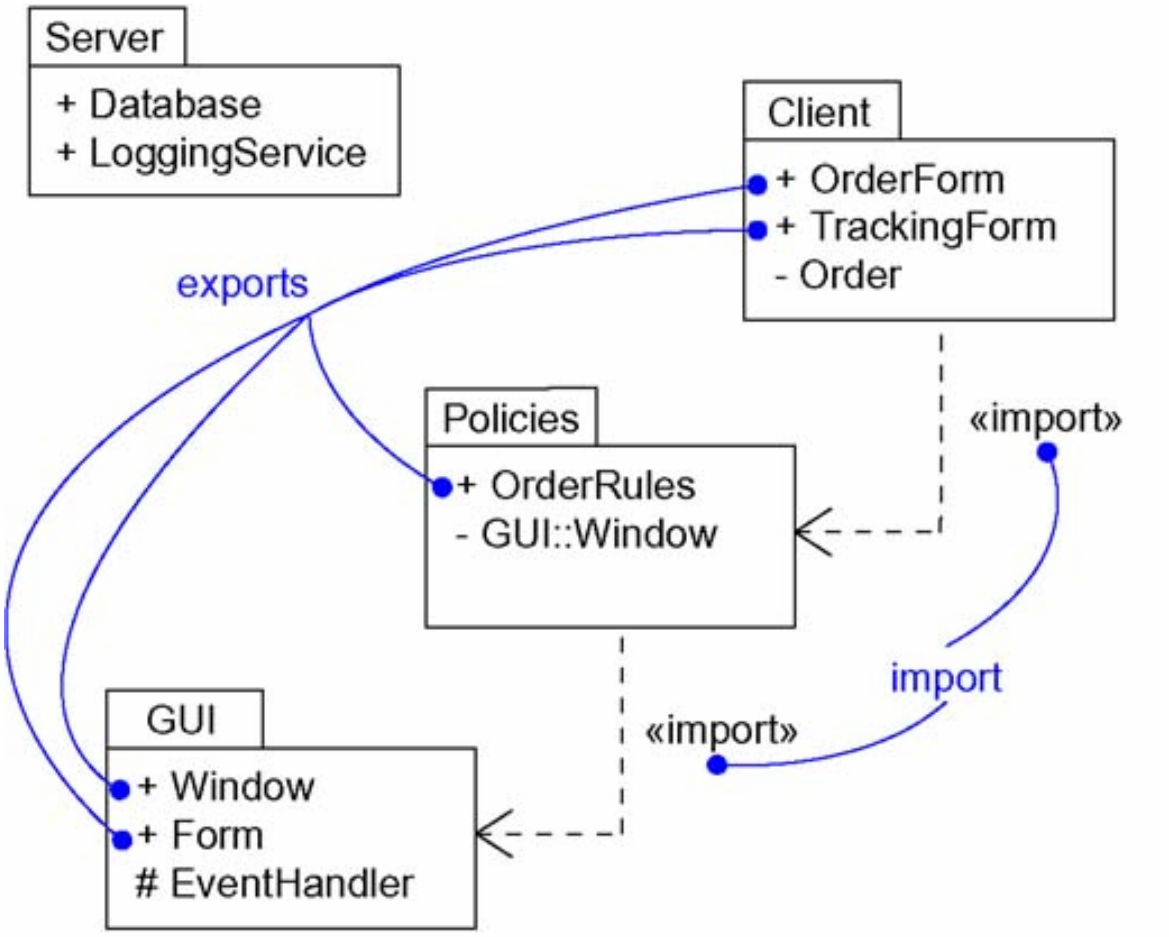
2.3 Visibility

- 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.

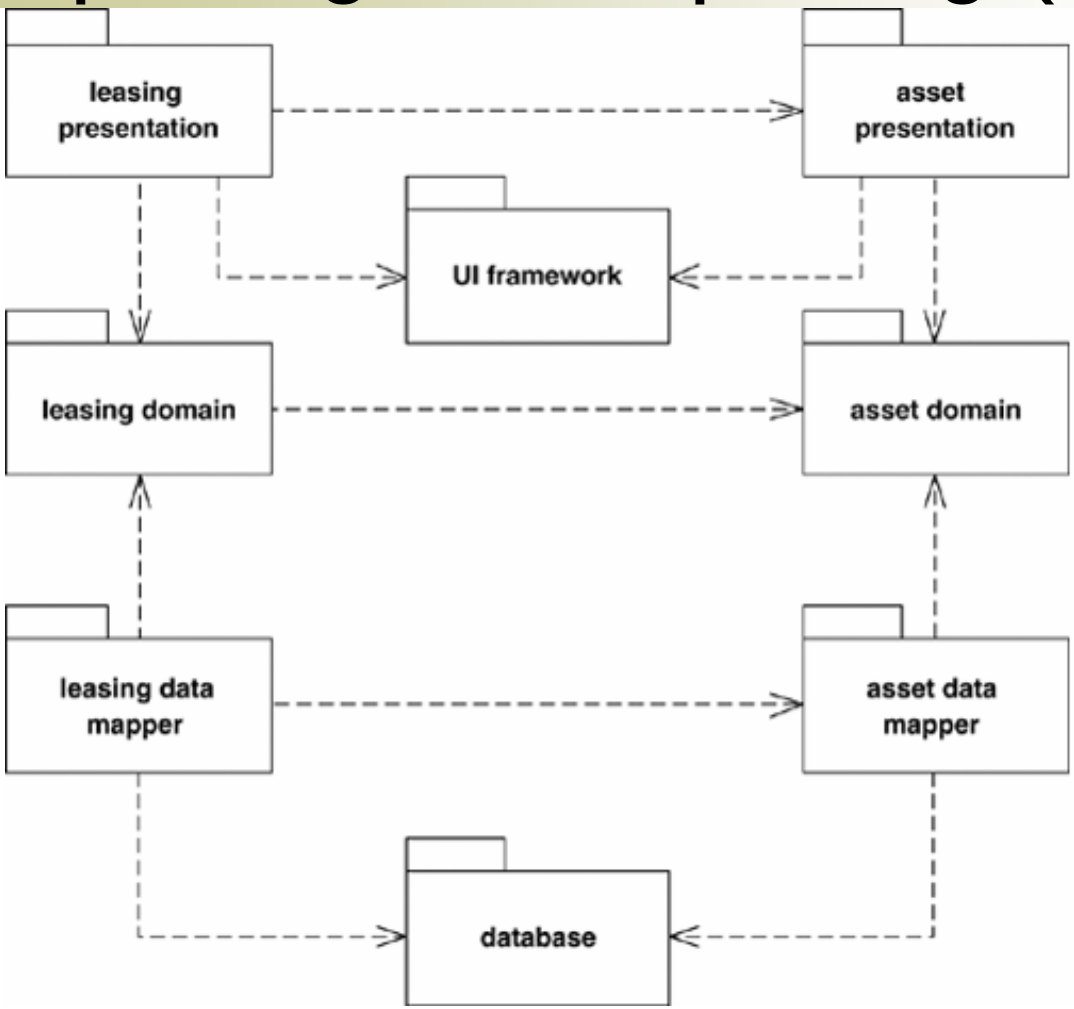
2.4 Importing and Exporting

- The public parts of a package are called its exports.
- two stereotypes : import and access
  - 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

2.4 Importing and Exporting (cont')



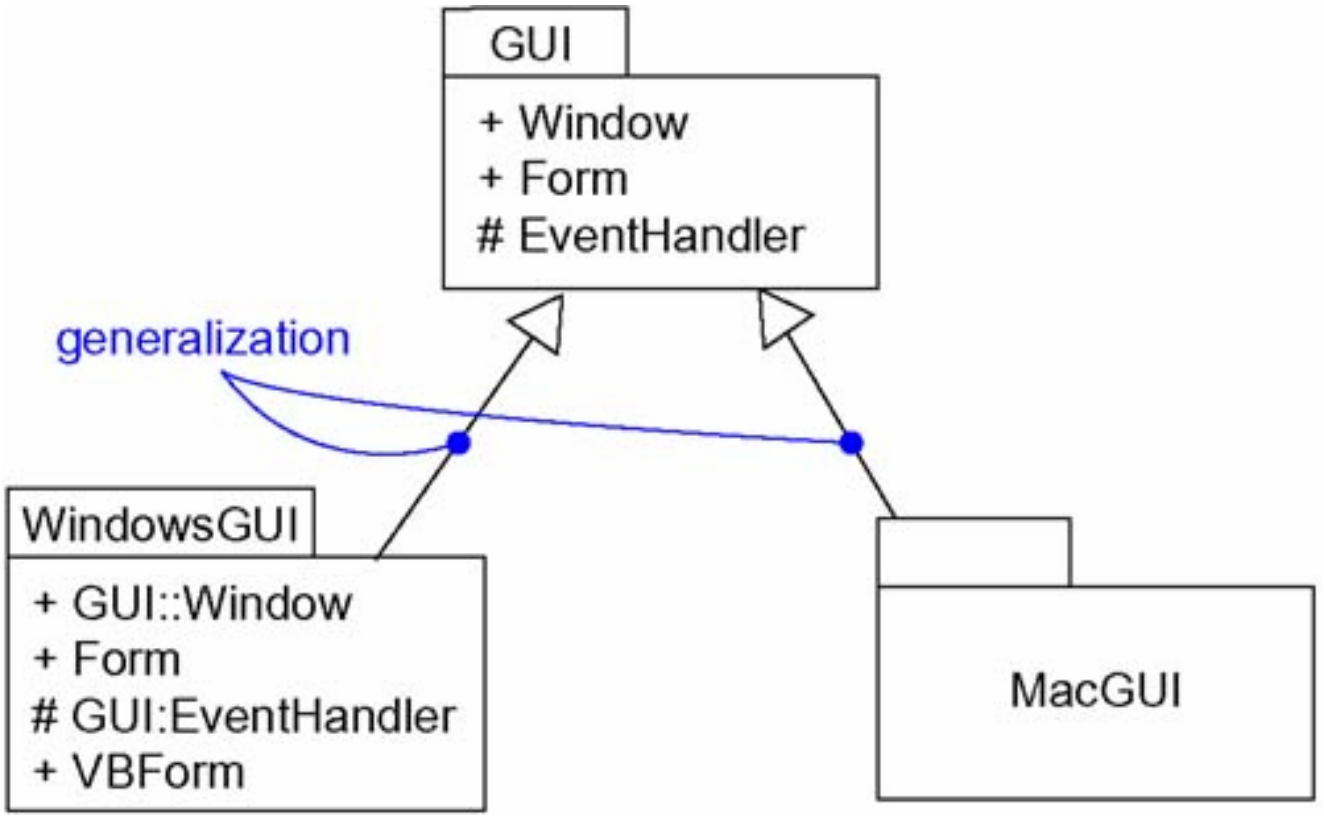
2.4 Importing and Exporting (cont')



2.5 Generalization

- Generalization among packages is very much like generalization among classes.
  - These specialized packages inherit the public and protected elements of the more general package.
  - Packages can replace more general elements and add new ones.
- Packages involved in generalization relationships follow the same principle of substitutability as do classes.

2.5 Generalization (cont')



2.6 Standard Elements

- <<facade>>
- <<framework>>
- <<stub>>
- <<subsystem>>
- <<system>>

3. Instances

- An *instance* is a concrete manifestation of an abstraction to which a set of operations may be applied and which may have a state that stores the effects of the operation.

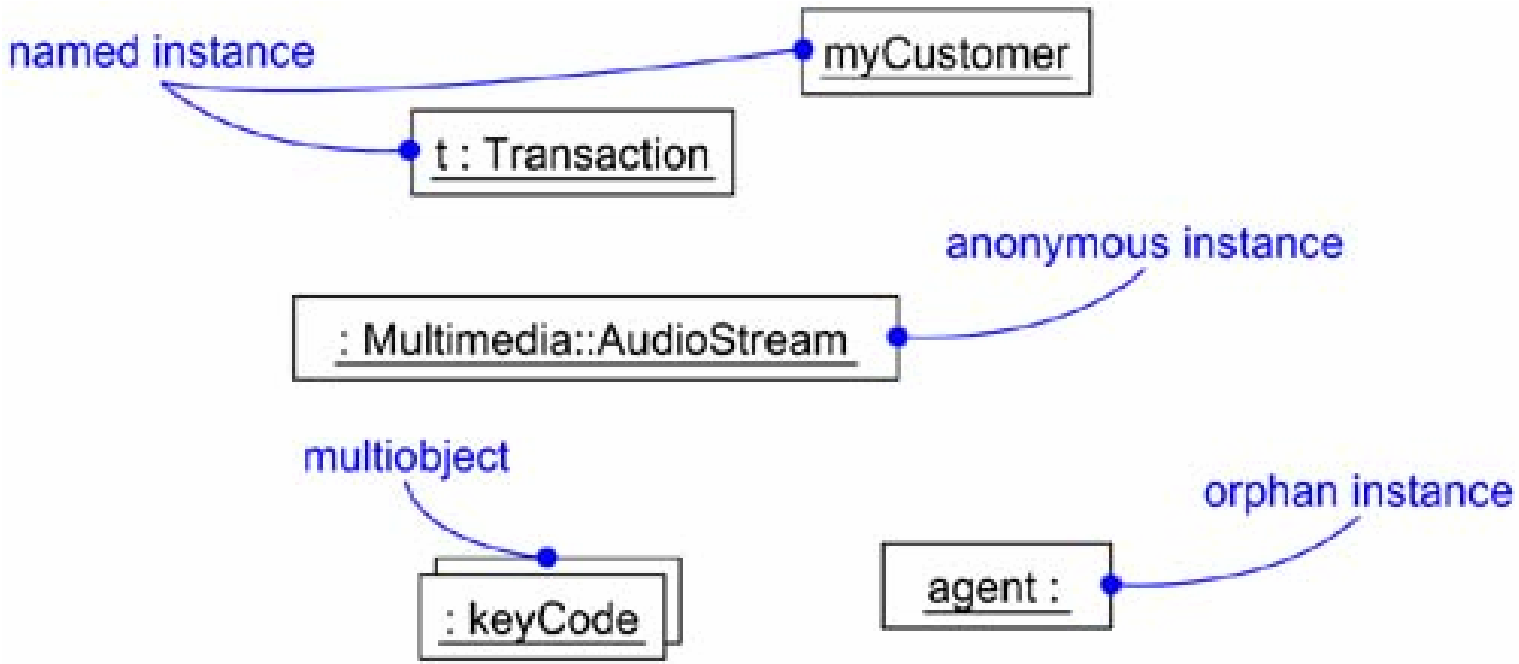
3.1 Abstractions and Instances

- Instances don't stand alone; they are almost always tied to an abstraction.
- In the UML, to indicate an instance, you underline its name.
- When you model instances, you'll place them in object diagrams or in interaction and activity diagrams.

3.2 Names

- A *name* is a textual string
- *simple name & path name*
  - simply name an object and elide its abstraction
  - anonymous instances
  - orphan instances
  - multiobjects

3.2 Names (cont')



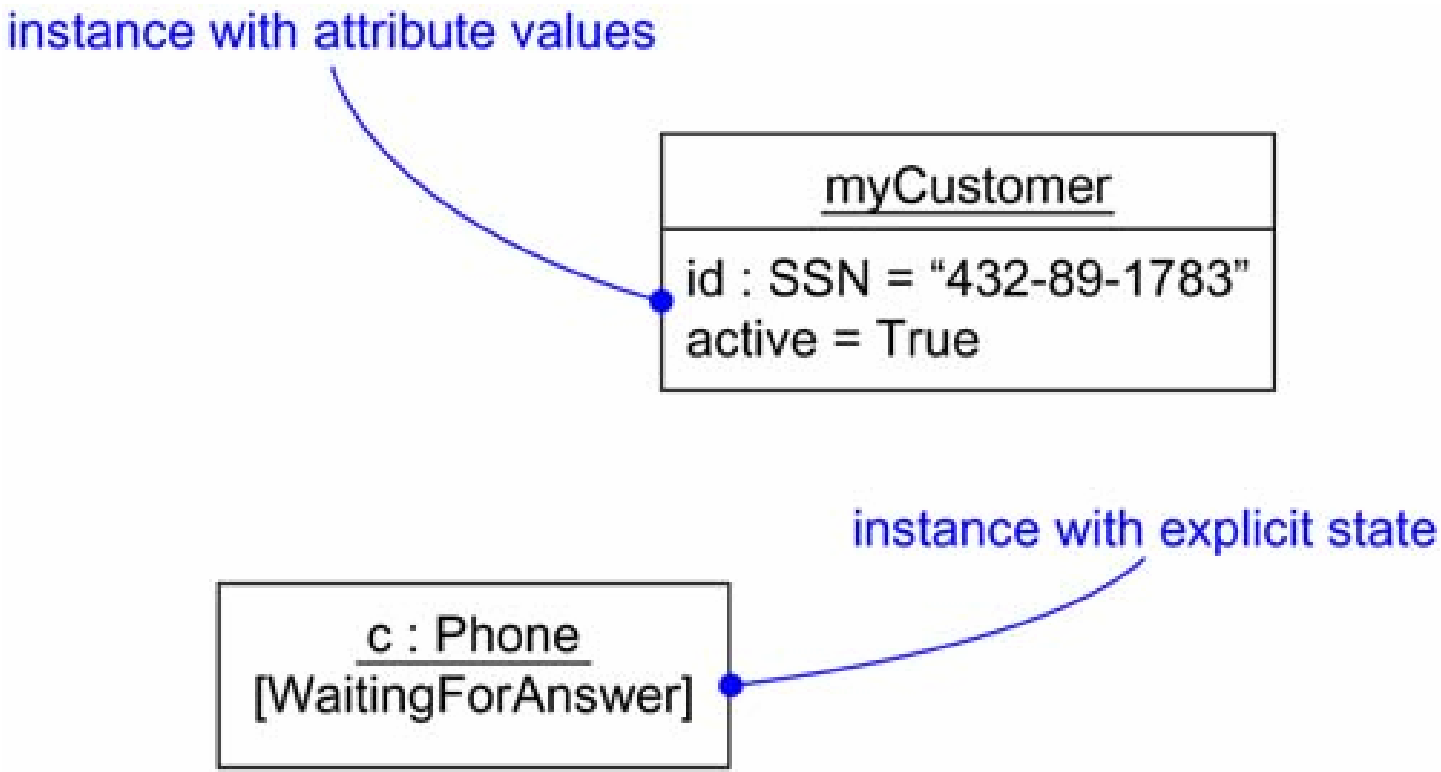
3.3 Operations

- The operations you can perform on an object are declared in the object's abstraction.

3.4 State

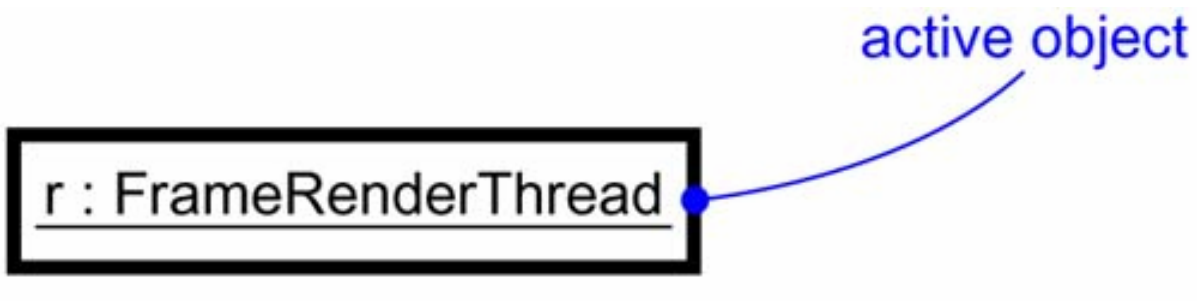
- An object also has state, which in this sense encompasses all the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties. Used in...
  - interaction diagram
  - state machine

3.4 State (cont')



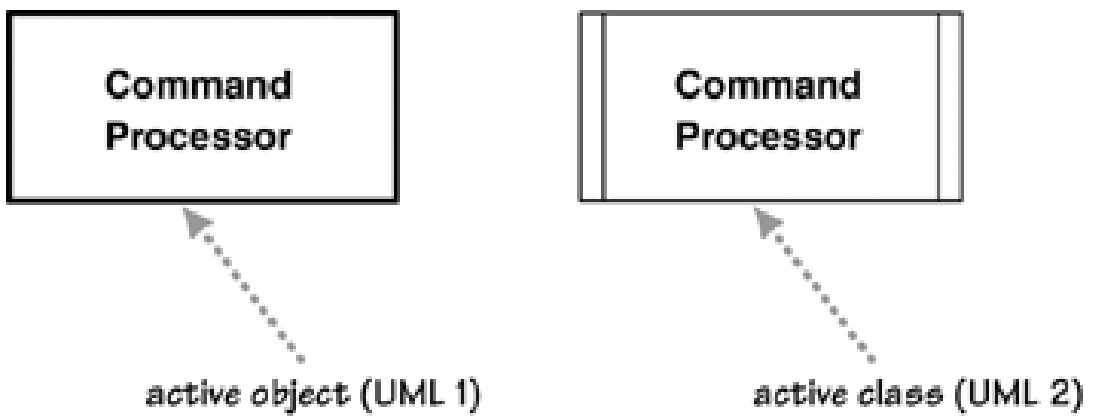
3.5 Other Features

- UML provides a visual cue to distinguish elements that are active (those that are part of a process or thread and represent a root of a flow of control) from those that are passive.



3.5 Other Features (cont')

- In UML 2, an active class has extra vertical lines on the side; in UML 1, it had a thick border and was called an active object.



3.5 Other Features (cont')

- There are two other elements in the UML that may have instances.
  - The first is a link. A link is a semantic connection among objects. An instance of an association is therefore a link.
  - The second is a class-scoped attribute and operation. A class-scoped feature is in effect an object in the class that is shared by all instances of the class.

3.6 Standard Elements

- All of the UML's extensibility mechanisms apply to objects.
- two standard stereotypes that apply to the dependency relationships among objects and among classes
  - <<instanceOf>>
  - <<instantiate>>

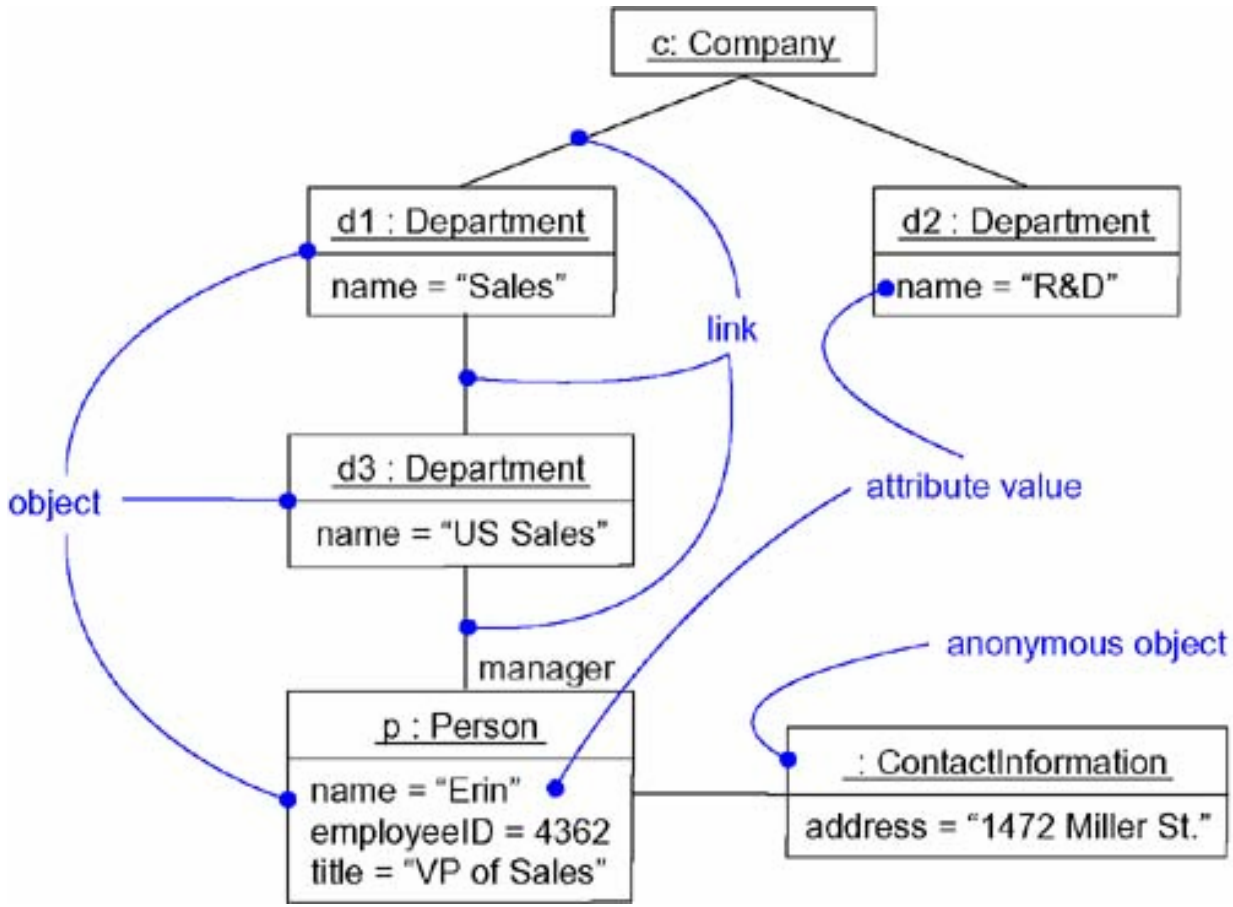
3.6 Standard Elements

- There are also two stereotypes related to objects that apply to messages and transitions:
  - <<become>>
  - <<copy>>
- The UML defines a standard constraint that applies to objects:
  - {transient}

4. Object Diagrams

- Object diagrams model the instances of things contained in class diagrams. An object diagram shows a set of objects and their relationships at a point in time.
- You use object diagrams to model the static design view or static process view of a system.
  - This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.

4. Object Diagrams (cont')



4.1 Contents

- Object diagrams commonly contain
  - Objects
  - Links
- Like all other diagrams, object diagrams may contain notes and constraints.

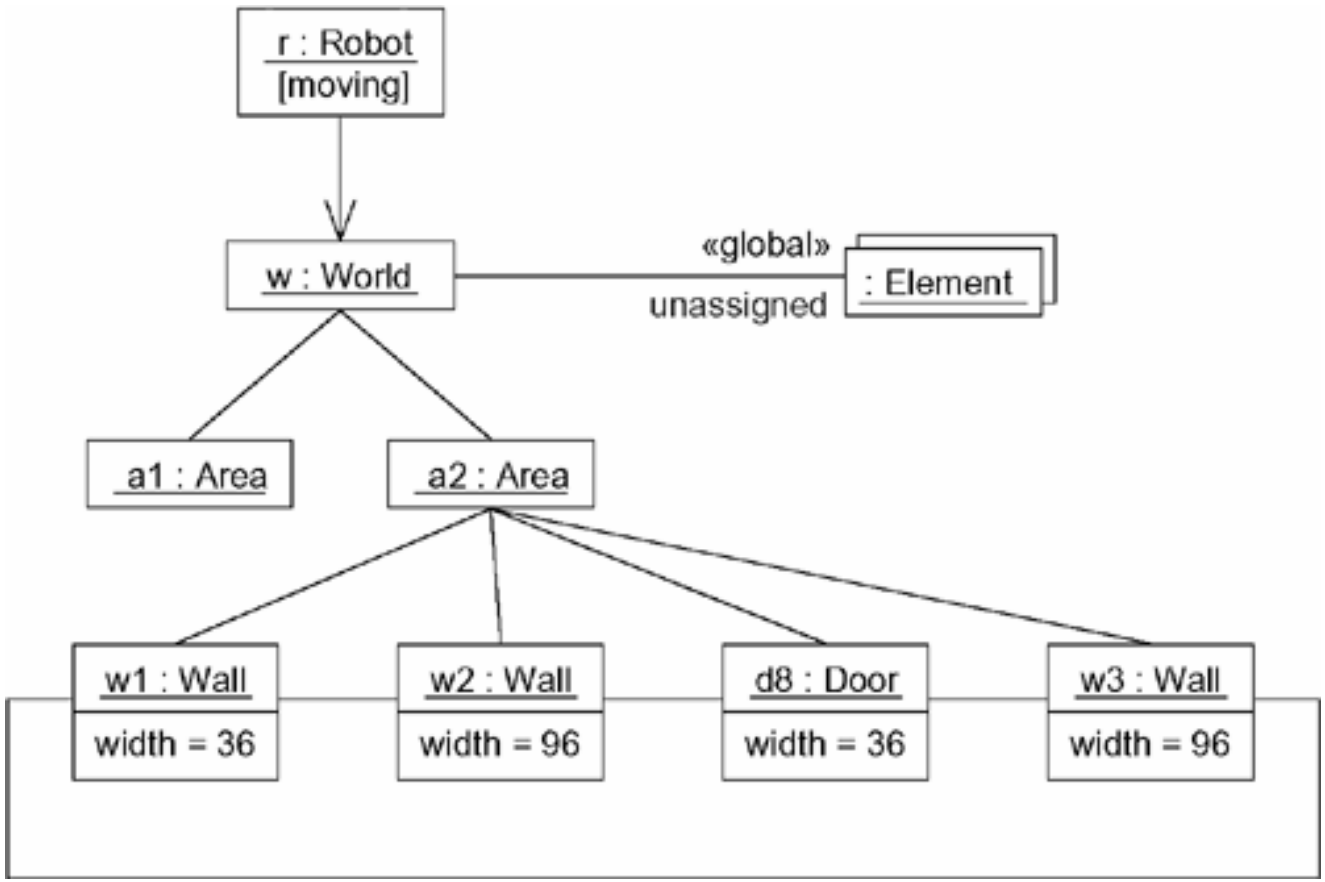
4.2 Common Uses

- You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams, but from the perspective of real or prototypical instances.
- This view primarily supports the functional requirements of a system—that is, the services the system should provide to its end users.
- Object diagrams let you model static data structures.

4.3 To model object structures

- Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time.
- An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram.

4.3 To model object structures



4.4 Forward and Reverse Engineering

- Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value.
- Reverse engineering (the creation of a model from code) an object diagram is a very different thing.