Unit-2

Structural Modeling

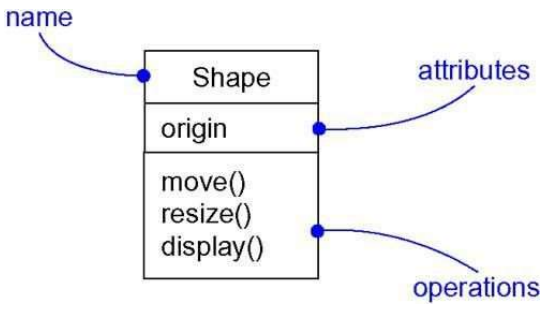
Structural Modeling:

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling.

Structural modeling captures the static features of a system.

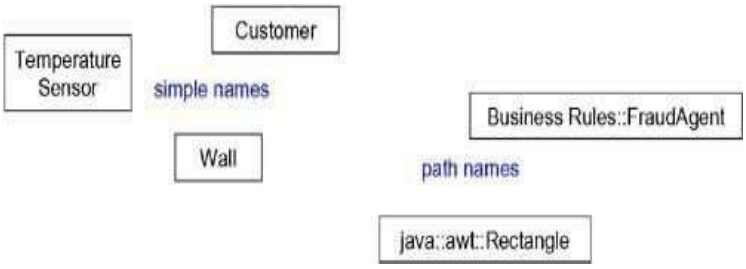
Classes:

- Classes are the most important building block of any object-oriented system.
- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces. Classes are used to capture the vocabulary of thesystem.
- We can use classes to represent software things, hardware things, and even things thatare purely conceptual.
- A class is an abstraction of the things that are a part of your vocabulary.
- A class is not an individual object, but represents a whole set of objects.
- The most important parts of an abstraction: its name, attributes, and operations.
- Graphically, a class is rendered as a rectangle.



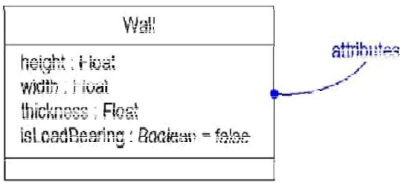
Names:

- A class name must be unique.
- Every class must have a name that distinguishes it from other classes.
- A name is a textual string. That name alone is known as a simple name.
- A path name is the class name prefixed by the name of the package in which that classlives.



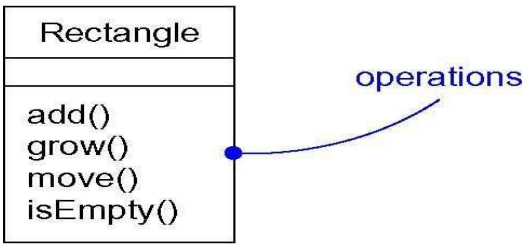
Attributes:

- Attributes are related to the semantics of aggregation.
- An attribute is a named property of a class that describes a range of values that instances ofthe property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute name may be text, just like a class name.
- An attribute by stating its class and possibly a default initial value.



Operations:

- An *operation* is the implementation of a service that can be requested from any object of theclass to affect behavior.
- A class may have any number of operations or no operations at all.
- An operation name is a short verb or verb phrase that represents some behavior of its enclosing class.

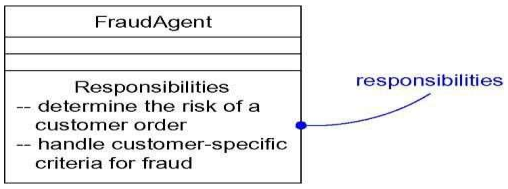


Organizing Attributes and Operations:

- When drawing a class, you don't have to show every attribute and every operation at once.
- In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view.

Responsibilities:

- Responsibilities are an example of a defined stereotype.
- A responsibility is a contract or an obligation of a class.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.

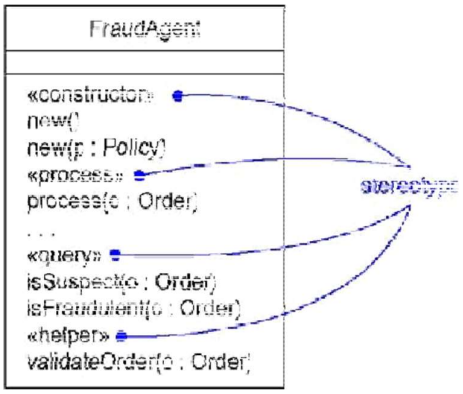


Common Modeling Techniques of Class:

1. Modeling the Vocabulary of a System
2. Modeling the Distribution of Responsibilities in a System.
3. Modeling Non-software Things.
4. Modeling Primitive Types.

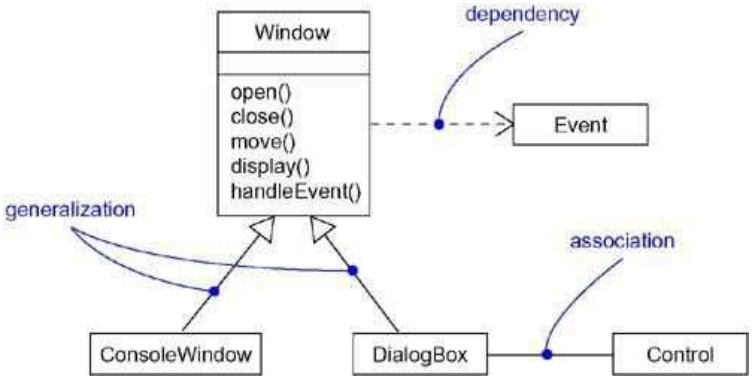
When drawing classes in the UML

- Show only those properties of the class that are important to understanding the abstractionin its context.
- Organize long lists of attributes and operations by grouping them according to their category.
- Show the related classes in same class diagrams.



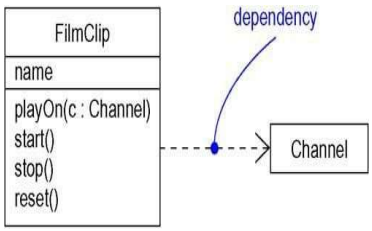
Relationships:

In object-oriented modeling, there are three kinds of relationships that are especially important: *dependencies, generalization and association.*



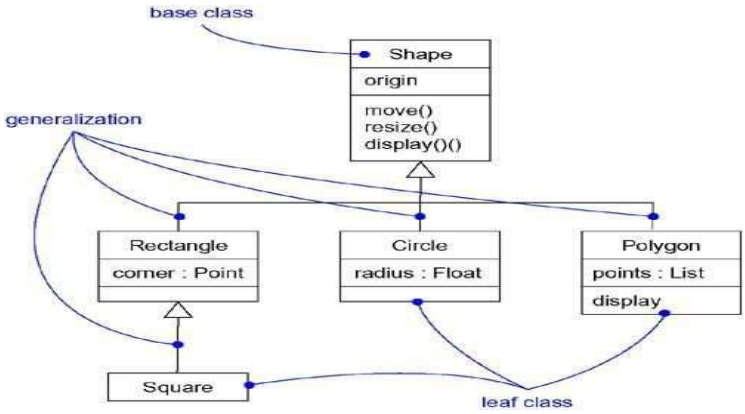
Dependency:

- A dependency is a using relationship that states that a change in specification of one thingmay affect another thing that uses, but not necessarily the reverse.
- Graphically, a dependency is rendered as a dashed directed line, directed to the thing beingdepended on.
- Use dependencies when you want to show one thing using another.



Generalization:

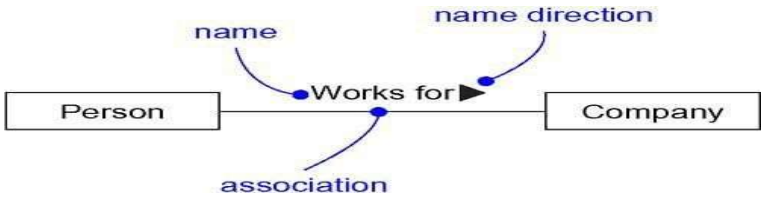
- A *generalization* is a relationship between a general thing (superclass or parent)and a morespecific kind of that thing (subclass or child).
- Generalization is sometimes called an "is-a-kind-of" relationship: one thing is-a-kind-of amore general thing.
- A class can have zero, one, or more parents.
- A class that has no parents and one or more children is called a root class or a base class.
- A class that has no children is called a leaf.
- A class that has exactly one parent is said to use single inheritance;
- A class with more than one parent is said to use multiple inheritance.



Association:

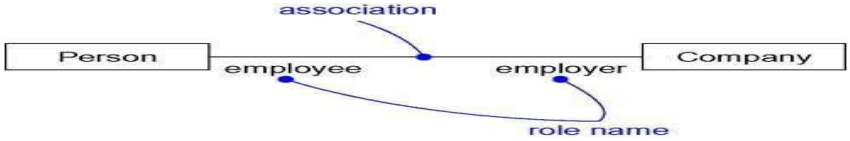
- Associations and dependencies (but not generalization relationships) may be reflective.
- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.
- An association connecting two classes, you can navigate from an object of one class to anobject of the other class, and vice versa.

Name: An association can have a name, and you use that name to describe the nature of therelationship.



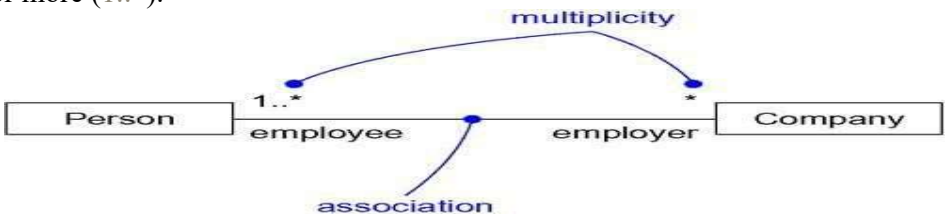
Role: Roles are related to the semantics of interfaces, when a class participates in an association, ithas a specific role that it plays in that relationship.

- A role is just the face the class at the near end of the association presents to the class at theother end of the association



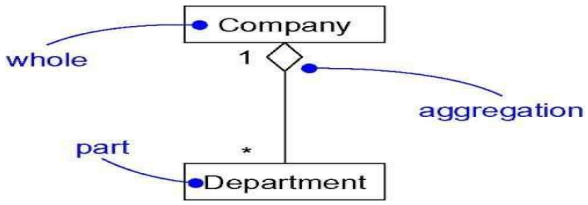
Multiplicity: An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association.

- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*).



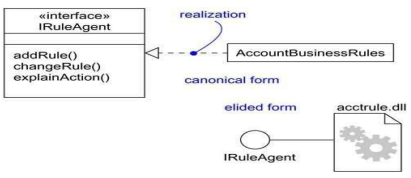
Aggregation: A plain association between two classes to represents a "has-a" relationship, meaning that an object of the whole has objects of the part

- To model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation

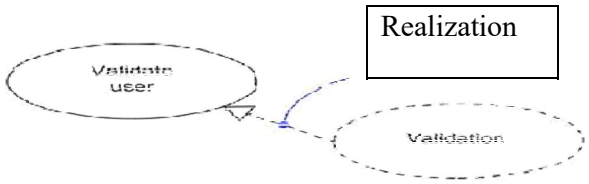


Realization:

- A *realization* is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
- A *realization* is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
- Use realization in two circumstances: in the context of interfaces and in the context of collaborations.



Realization of Interface

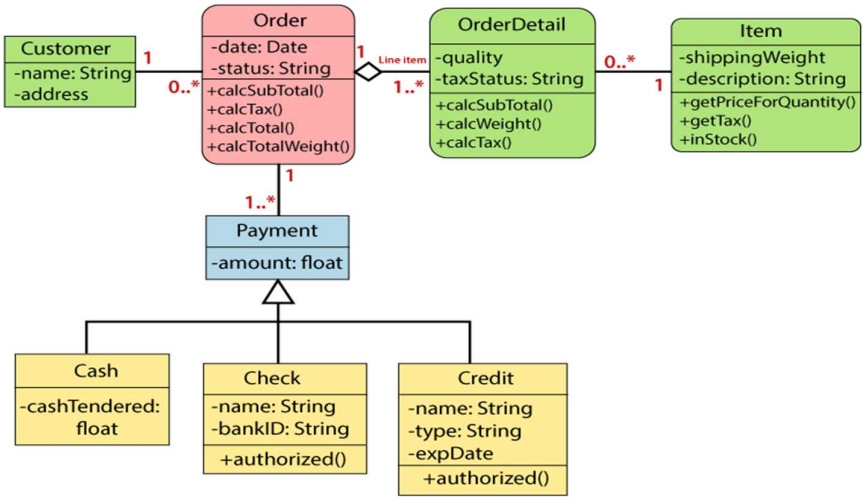


Realization of Usecase

Class Diagrams:

A *class diagram* shows a set of classes, interfaces, and collaborations and their relationships.

- Class diagrams are the most common diagram found in modeling object-oriented systems.
- We can use class diagrams to model 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 important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.



Object Diagrams:

- An object diagram shows a set of objects and their relationships at a point in time.
- Object diagrams model the instances of things contained in class diagrams.
- 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 ofobjects, their state, and their relationships.

- Object diagrams are not only important for visualizing, specifying, and documenting structural models, but also for constructing the static aspects of systems through forward and reverse engineering.

Object diagrams commonly contain

- Objects
- Links
- Like all other diagrams object diagrams may contain notes and constraints.
- Object diagrams may also contain packages or subsystems.

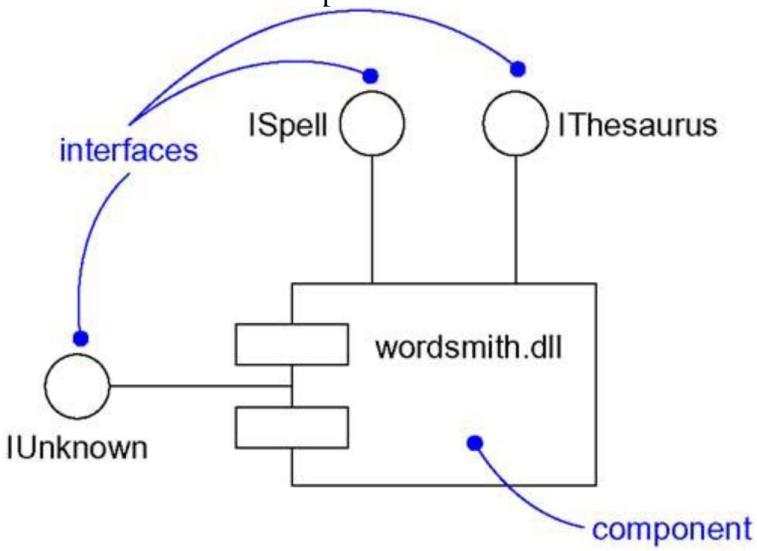
Interfaces and Packages:

Interface:

Interfaces define a line between the specification of what an abstraction does and the implementation of how that abstraction does it. An interface is a collection of operations that are used to specify a service of a class or a component.

You use interfaces to visualize, specify, construct, and document the seams within your system. Types and roles provide a mechanism for you to model the static and dynamic conformance of an abstraction to an interface in a specific context.

A well-structured interface provides a clear separation between the outside view and the inside view of an abstraction, making it possible to understand and approach an abstraction without having to dive into the details of its implementation.



Terms and Concepts:

An *interface* is a collection of operations that are used to specify a service of a class or a component. 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. A *role* is the behavior of an entity participating in a particular context. Graphically, an interface is rendered as a circle; in its expanded form, an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names:

An interface name must be unique within its enclosing package. 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 in which that interface lives. An interface may be drawn showing only its name.

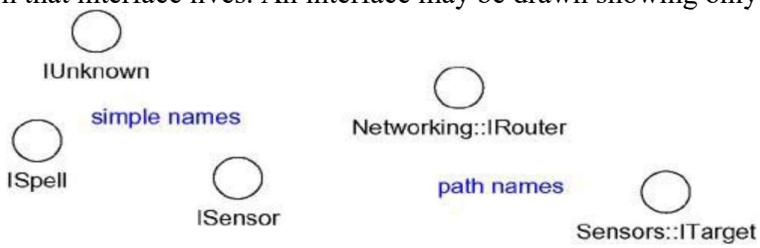


Fig: Simple and Path Names

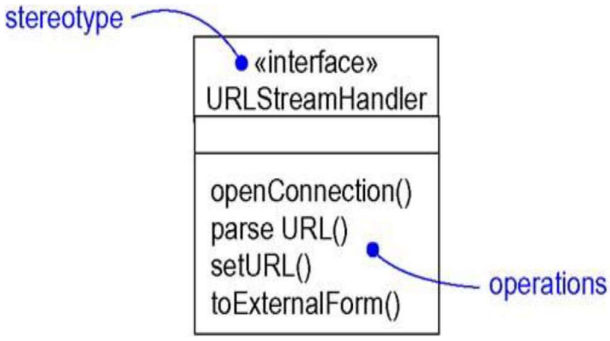
Note:

An interface name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate an interface name and the name of its enclosing package) and may continue over several lines. In practice, interface names are short nouns or noun phrases drawn from the vocabulary of the system you are modelling.

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, nor do they specify any implementation. Like a class, an interface may have any number of operations. These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.

When you visualize an interface in its normal form as a circle, by definition, you suppress the display of these operations.

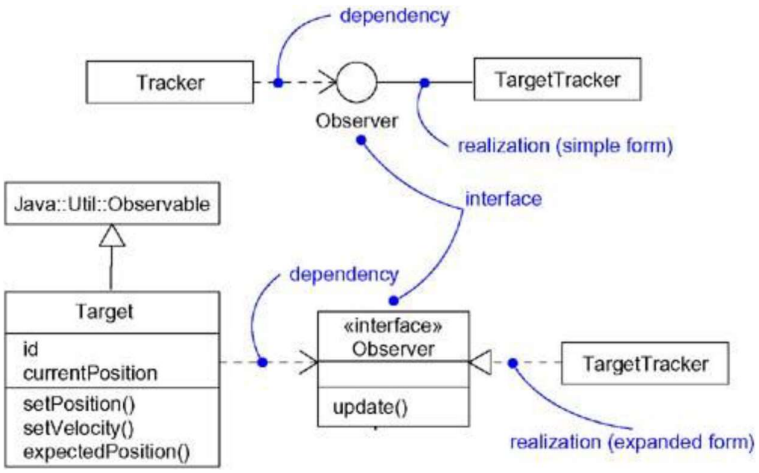


Relationships:

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.

An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces. A class or a component may depend on many interfaces.

The below diagram illustrates as 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.

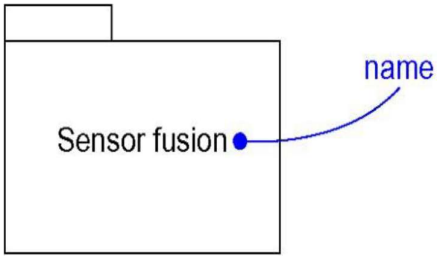


In both cases, you attach the class or component that builds on an interface with a dependency relationship from the element to the interface.

Packages:

In the UML, the chunks that organize a model are called packages. A package is a general purpose mechanism for organizing elements into groups. Packages help you organize the elements in your models so that you can more easily understand them. Packages also let you control access to their contents so that you can control the seams in your system's architecture.

The UML provides a graphical representation of package. This notation permits you to visualize groups of elements that can be manipulated as a whole and in a way that lets you control the visibility of and access to individual elements.

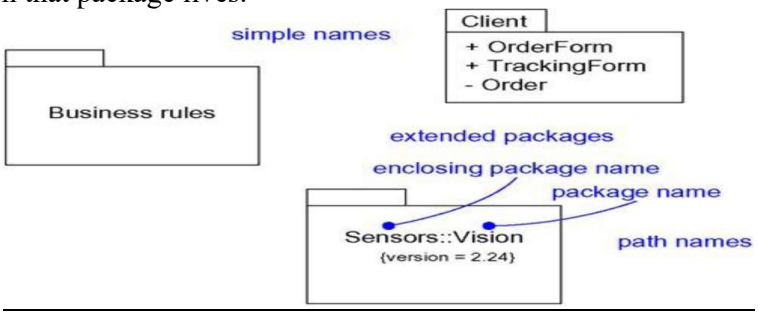


Terms and Concepts:

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

Names:

A package name must be unique within its enclosing package. 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.



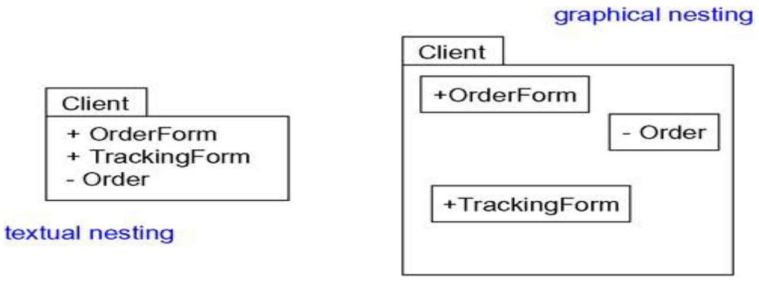
Owned Elements:

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.

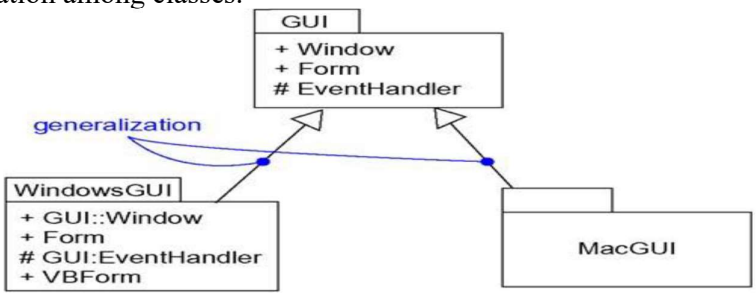
A package forms a namespace, which means that elements of the same kind must be named uniquely within the context of its enclosing package.

The below diagram illustrates that you can explicitly show the contents of a package either textually or graphically. Note that when you show these owned elements, you place the name of the package in the tab.



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.



Standard Elements:

All of the UML's extensibility mechanisms apply to packages. The UML defines five standard stereotypes that apply to packages.

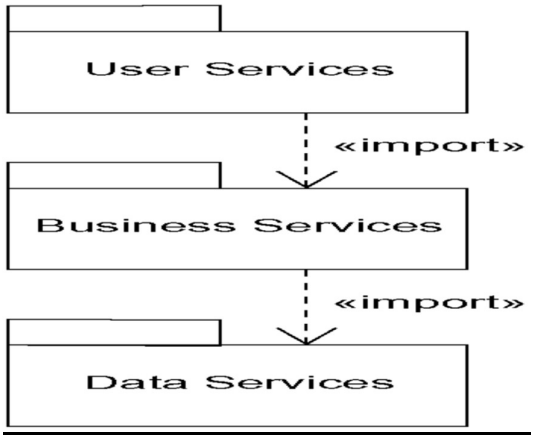
1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents of another package
4. subsystem	Specifies a package representing an independent part of the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

Common Modeling Techniques:

Modeling Groups of Elements:

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:

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.