UNIT-II

INTRODUCTION TO UML

Why We Model

· The importance of modeling
· Four principles of modeling
· The essential blueprints of a software system
· Object-oriented modeling

The Importance of Modeling

If you want to build a dog house, you can pretty much start with a pile of lumber, some nails, and a few basic tools, such as a hammer, saw, and tape measure. In a few hours, with little prior planning, you'll likely end up with a dog house that's reasonably functional, and you can probably do it with no one else's help. As long as it's big enough and doesn't leak too much, your dog will be happy. If it doesn't work out, you can always start over, or get a less demanding dog. If you want to build a house for your family, you can start with a pile of lumber, some nails, and a few basic tools, but it's going to take you a lot longer, and your family will certainly be more demanding than the dog. In this case, unless you've already done it a few dozen times before, you'll be better served by doing some detailed planning before you pound the first nail or lay the foundation. At the very least, you'll want to make some sketches of how you want the house to look. If you want to build a quality house that meets the needs of your family and of local building codes, you'll need to draw some blueprints as well, so that you can think through the intended use of the rooms and the practical details of lighting, heating, and plumbing. Given these plans, you can start to make reasonable estimates of the amount of time and materials this job will require. Although it is humanly possible to build a house yourself, you'll find it is much more efficient to work with others, possibly subcontracting out many key work products or buying prebuilt materials. As long as you stay true to your plans and stay within the limitations of time and money, your family will most likely be satisfied. If it doesn't work out, you can't exactly get a new family, so it is best to set expectations early and manage change carefully. If you want to build a high-rise office building, it would be infinitely stupid for you to start with a pile of lumber, some nails, and a few basic tools. Because you are probably using other people's money, they will demand to have input into the size, shape, and style of the building. Often, they will change their minds, even after you've started building. You will want to do extensive planning, because the cost of failure is high. You will be just a part of a much larger group responsible for developing and deploying the building, and so the team will need all sorts of blueprints and models to communicate with one another. As long as you get the right people and the right tools and actively manage the process of transforming an architectural concept into reality, you will likely end up with a building that will satisfy its tenants. If you want to keep building buildings, then you will want to be certain to balance the desires of your tenants with the realities of building technology, and you will want to treat the rest of your team professionally, never placing them at any risk or driving them so hard that they burn out.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling. However, it's definitely true that the larger and more complex the system,

the more important modeling becomes, for one very simple reason: *We build models of complex systems because we cannot comprehend such a system in its entirety.* There are limits to the human ability to understand complexity. Through modeling, we narrow the problem we are studying by focusing on only one aspect at a time. This is essentially the approach of "divide-and-conquer" that Edsger Dijkstra spoke of years ago: Attack a hard problem by dividing it into a series of smaller problems that you can solve. Furthermore, through modeling, we amplify the human intellect. A model properly chosen can enable the modeler to work at higher levels of abstraction.

Principles of Modeling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling. First, *The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.* In software, the models you choose can greatly affect your world view. If you build a system through the eyes of a database developer, you will likely focus on entity-relationship models that push behavior into triggers and stored procedures. If you build a system through the eyes of a structured analyst, you will likely end up with models that are algorithmic-centric, with data flowing from process to process. If you build a system through the eyes of an object-oriented developer, you'll end up with a system whose architecture is centered around a sea of classes and the patterns of interaction that direct how those classes work together. Any of these approaches might be right for a given application and development culture, although experience suggests that the object-oriented view is superior in crafting resilient architectures, even for systems that might have a large database or computational element. That fact notwithstanding, the point is that each world view leads to a different kind of system, with different costs and benefits.

Second,

Every model may be expressed at different levels of precision. If you are building a high rise, sometimes you need a 30,000-foot view• for instance, to help your investors visualize its look and feel. Other times, you need to get down to the level of the studs• for instance, when there's a tricky pipe run or an unusual structural element. The same is true with software models. Sometimes, a quick and simple executable model of the user interface is exactly what you need; at other times, you have to get down and dirty with the bits, such as when you are specifying cross-system interfaces or wrestling with networking bottlenecks. In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

Third,

The best models are connected to reality. A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details. In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In objectoriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

Fourth,
No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models. If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans. The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.
The same is true of object-oriented software systems. To understand the architecture of such a system, you need several complementary and interlocking views: a use case view (exposing the requirements of the system), a design view (capturing the vocabulary of the problem space and the solution space), a process view (modeling the distribution of the system's processes and threads), an implementation view (addressing the physical realization of the system), and a deployment view (focusing on system engineering issues). Each of these views may have structural, as well as behavioral, aspects. Together, these views represent the blueprints of software.

Object-Oriented Modeling

Civil engineers build many kinds of models. Most commonly, there are structural models that help people visualize and specify parts of systems and the way those parts relate to one another. Depending on the most important business or engineering concerns, engineers might also build dynamic models• for instance, to help them to study the behavior of a structure in the presence of an earthquake. Each kind of model is organized differently, and each has its own focus. In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective. The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure or function. This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones. There's nothing inherently evil about such a point of view except that it tends to yield brittle systems. As requirements change (and they will) and the system grows (and it will), systems built with an algorithmic focus turn out to be very hard to maintain.

A Conceptual Model of the UML

To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML. Once you have grasped these ideas, you will be able to read UML models and create some basic ones. As you gain more experience in applying the UML, you can build on this conceptual model, using more advanced features of the language.

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:
1. Things
2. Relationships
3. Diagrams
Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML

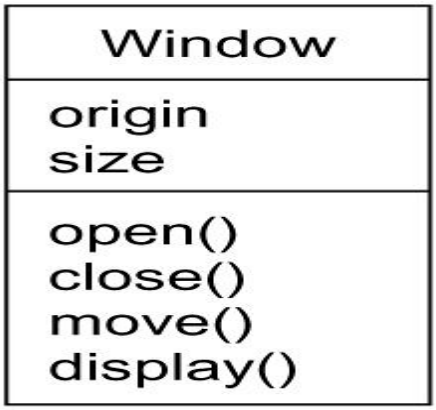
There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

These things are the basic object-oriented building blocks of the UML

Structural Things

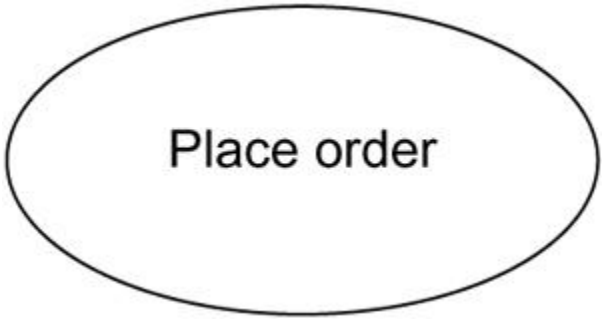
Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things. First, a *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations



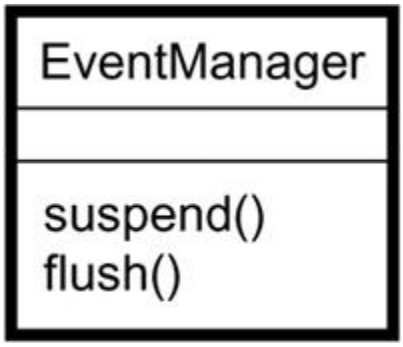
Second, an *interface* is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface,

Third, a *collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name,

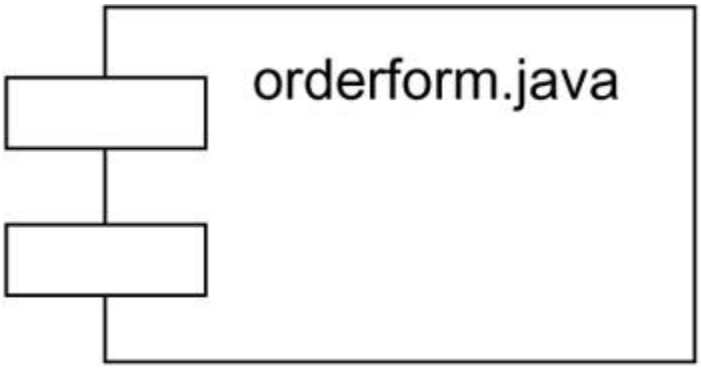
Fourth, a *use case* is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name



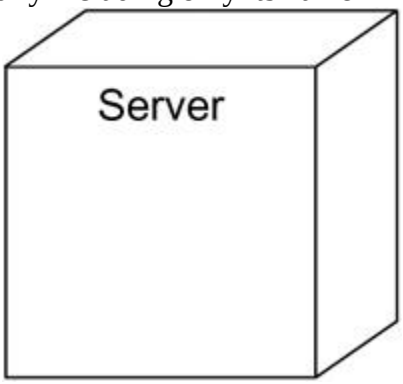
Fifth, an *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations,



Sixth, a *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, as well as components that are artifacts of the development process, such as source code files. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name



Seventh, a *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name



Behavioral Things

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioural things. objects within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in Figure 2-8.

Figure 2-8 Messages

Second, a *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in Figure 2-9.

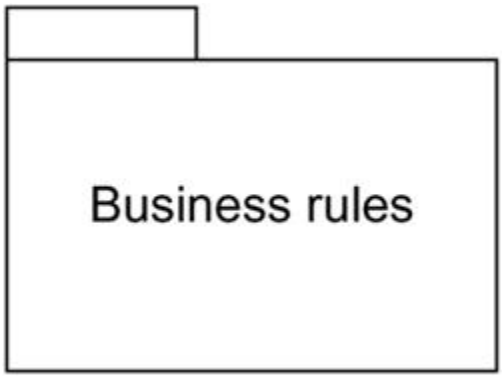
Figure 2-9 States

These two elements• interactions and state machines• are the basic behavioral things that you may include in a UML model. Semantically, these elements are usually connected to various structural elements, primarily classes, collaborations, and objects.

Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

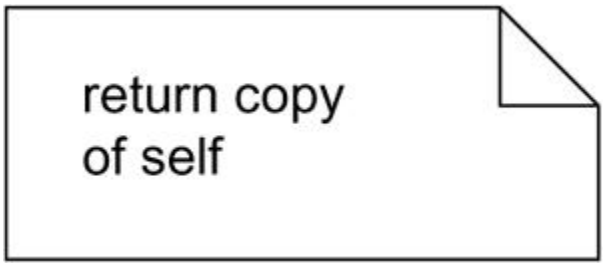
A *package* is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents,



Packages

Annotational Things

*Annotational things*are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and 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



Notes

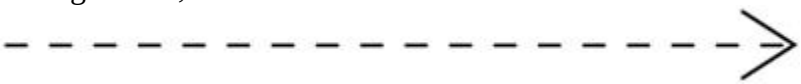
This element is the one basic annotational thing you may include in a UML model. You'll typically use notes to adorn your diagrams with constraints or comments that are best expressed in informal or formal text. There are also variations on this element, such as requirements (which specify some desired behavior from the perspective of outside the model).

Relationships in the UML

There are four kinds of relationships in the UML:

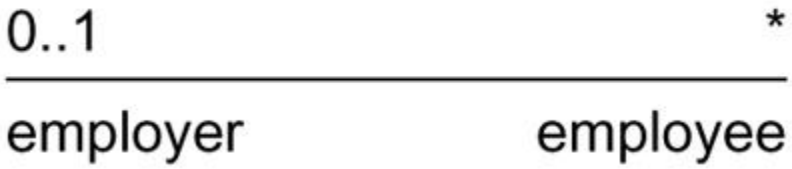
1. Dependency
2. Association
3. Generalization
4. Realization

First, a *dependency* is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label,



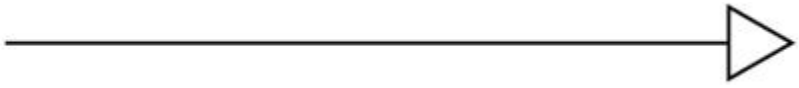
Dependencies

Second, an *association* is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names,



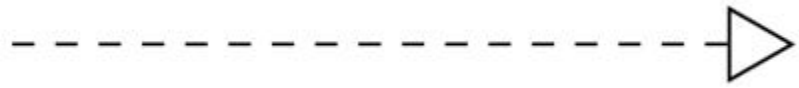
Associations

Third, a *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent,



Generalizations

Fourth, a *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship



Realization

Diagrams in the UML

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system. The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case). In theory, a diagram may contain any combination of things and relationships. In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system. For this reason, the UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

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

An *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. An shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-

ordering of messages; a *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

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

An *activity diagram* is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

A *component diagram* shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components.

Architecture

Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders• end users, analysts, developers, system integrators, testers, technical writers, and project managers• each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.
Architecture is the set of significant decisions about

· The organization of a software system
· The selection of the structural elements and their interfaces by which the system is composed
· Their behavior, as specified in the collaborations among those elements
· The composition of these structural and behavioral elements into progressively larger Subsystems
· The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

As Figure 2-20 illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

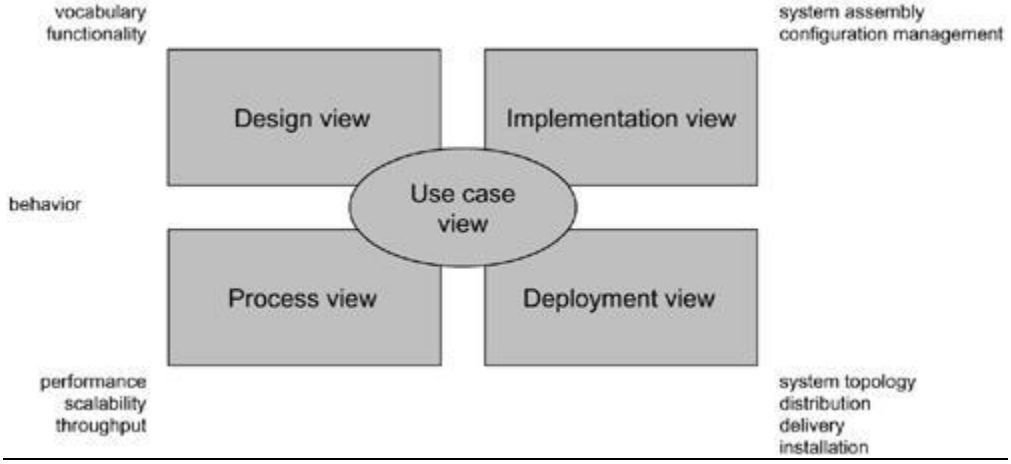


Figure 2-20 Modeling a System's Architecture

The *use case view* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *design view* of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *process view* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The *implementation view* of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another• nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express every one of these five views and their interactions.

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. Well-structured classes have crisp boundaries and form a part of a balanced distribution of responsibilities across the system.

In the UML, all of these things are modeled as classes. A class is an abstraction of the things that are a part of your vocabulary. A class is not an individual object, but rather represents a whole set of objects. Thus, you may conceptually think of "wall" as a class of objects with certain common properties, such as height, length, thickness, load-bearing or not, and so on. You may also think of individual instances of wall, such as "the wall in the southwest corner of my study." In software, many programming languages directly support the concept of a class. That's excellent, because it means that the abstractions you create can often be mapped directly to a programming language, even if these are abstractions of nonsoftware things, such as "customer," "trade," or "conversation."

The UML provides a graphical representation of class, as well, as Figure 4-1 shows. This notation permits you to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations.

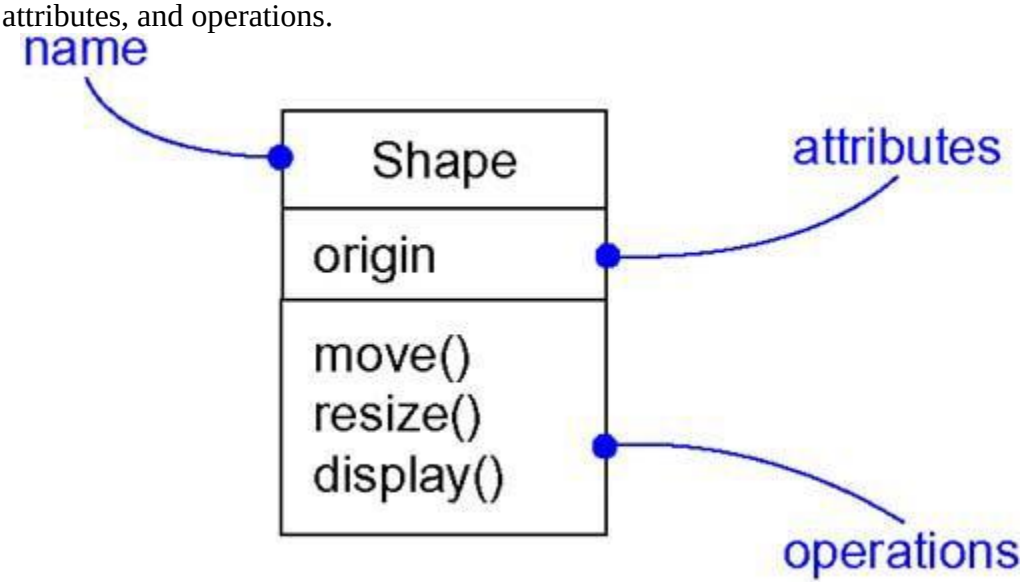


Figure 4-1 Classes

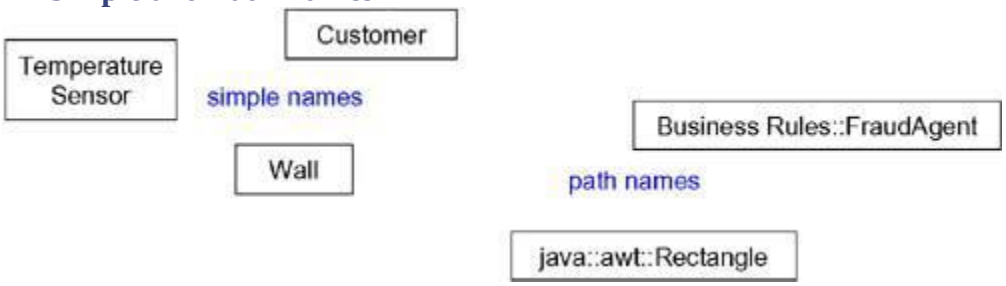
Terms and Concepts

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.

Names

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 class lives. A class may be drawn showing only its name, as

Figure 4-2 Simple and Path Names



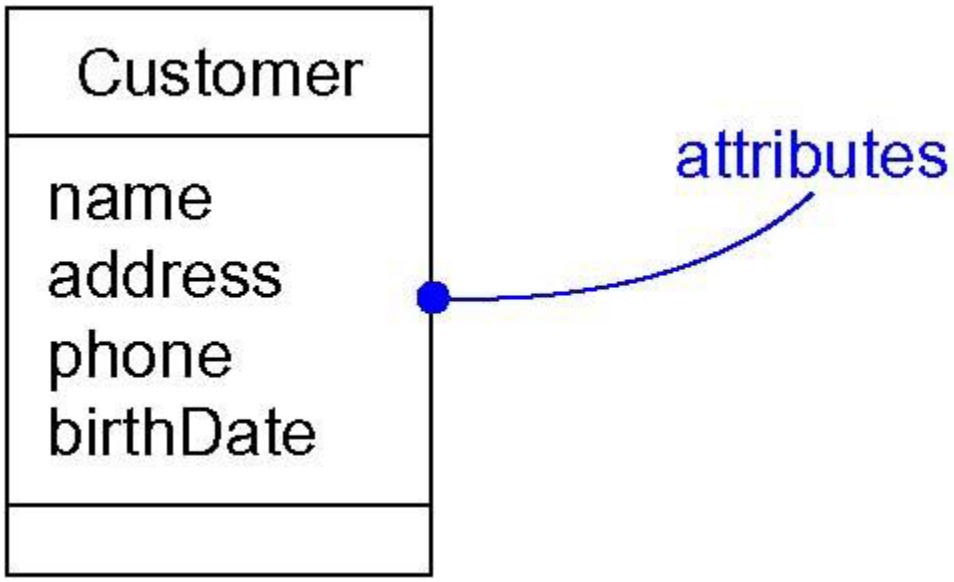
A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a class name, as in Customer or TemperatureSensor.

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 the thing you are modeling that is shared by all objects of that class.

For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth. An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass. At a given moment, an object of a class will have specific values for every one of its class's attributes. Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names, as shown in Figure 4-3.

Figure 4-3 Attributes

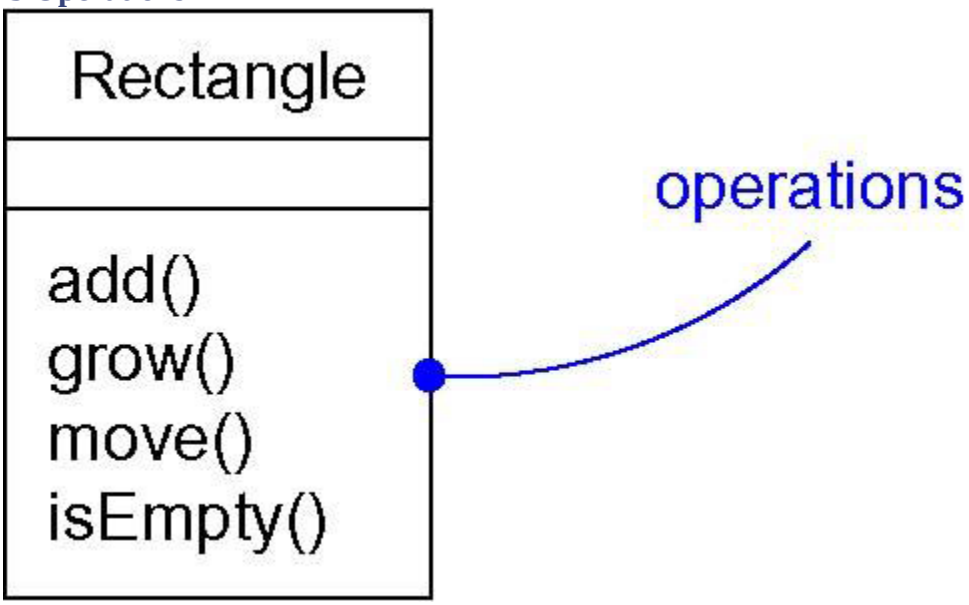


An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class. Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in name or loadBearing.

Operations

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all. For example, in a windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure 4-5.

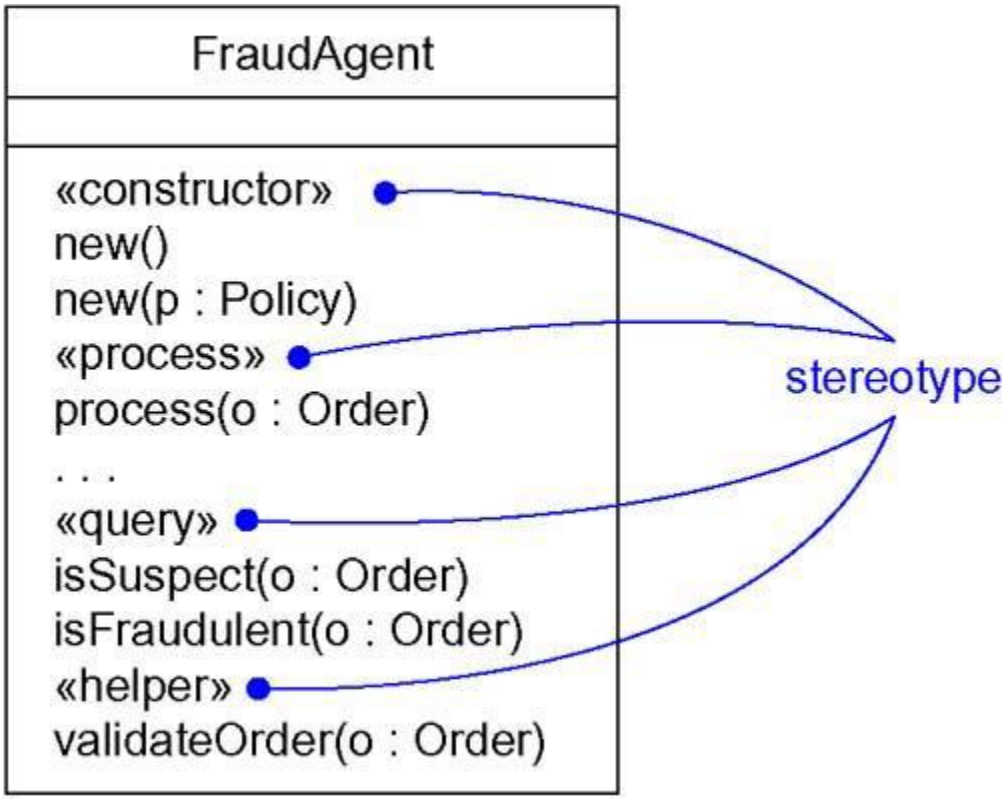
Figure 4-5 Operations



An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in move or isEmpty.

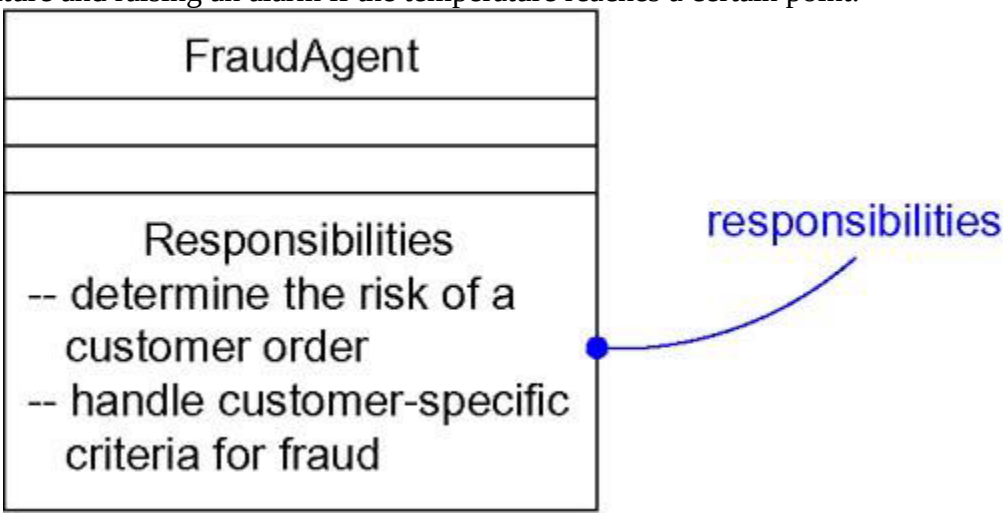
Organizing Attributes and Operations

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). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. An empty compartment doesn't necessarily mean there are no attributes or operations, just that you didn't choose to show them. You can explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("...").



Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.



Relationships

In object-oriented modeling, there are three kinds of relationships that are especially important: *dependencies,* which represent using relationships among classes (including refinement, trace, and bind relationships); *generalizations,* which link generalized classes to their specializations; and *associations,* which represent structural relationships among objects. Each of these relationships provides a different way of combining your abstractions.

Building webs of relationships is not unlike creating a balanced distribution of responsibilities among your classes. Over-engineer, and you'll end up with a tangled mess of relationships that make your model incomprehensible; under-engineer, and you'll have missed a lot of the richness of your system embodied in the way things collaborate.

If you are building a house, things like walls, doors, windows, cabinets, and lights will form part of your vocabulary. None of these things stands alone, however. Walls connect to other walls. Doors and windows are placed in walls to form openings for people and for light. Cabinets and lights are physically attached to walls and ceilings. You group walls, doors, windows, cabinets, and lights together to form higher-level things, such as rooms.

Not only will you find structural relationships among these things, you'll find other kinds of relationships, as well. For example, your house certainly has windows, but there are probably many kinds of windows. You might have large bay windows that don't open, as well as small kitchen windows that do. Some of your windows might open up and down; others, like patio windows, will slide left and right. Some windows have a single pane of glass; others have double.

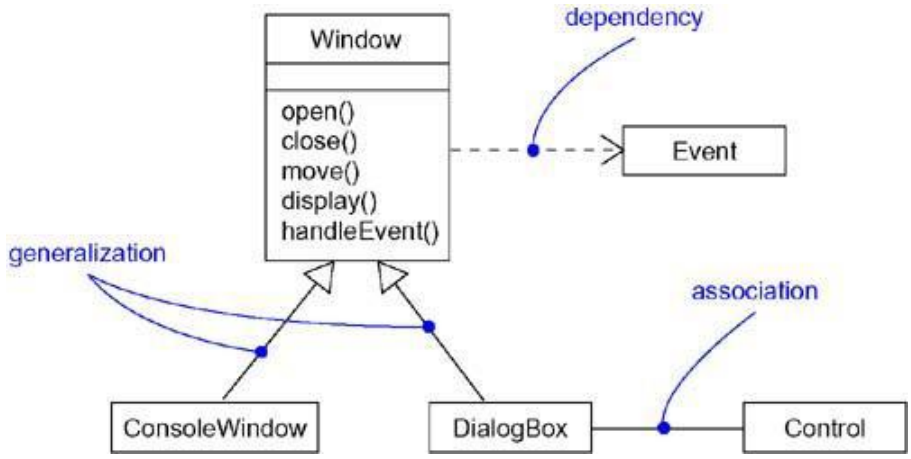
In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships. In object-oriented modeling, there are three kinds of relationships that are most important: dependencies, generalizations, and associations.

Dependencies are using relationships. For example, pipes depend on the water heater to heat the water they carry.

Generalizations connect generalized classes to more-specialized ones in what is known as subclass/superclass or child/parent relationships. For example, a bay window is a kind of window with large, fixed panes; a patio window is a kind of window with panes that open side to side.

Associations are structural relationships among instances. For example, rooms consist of walls and other things; walls themselves may have embedded doors and windows; pipes may pass through walls.

The UML provides a graphical representation for each of these kinds of relationships, as Figure 5-1 shows. This notation permits you to visualize relationships apart from any specific programming language, and in a way that lets you emphasize the most important parts of a relationship: its name, the things it connects, and its properties.

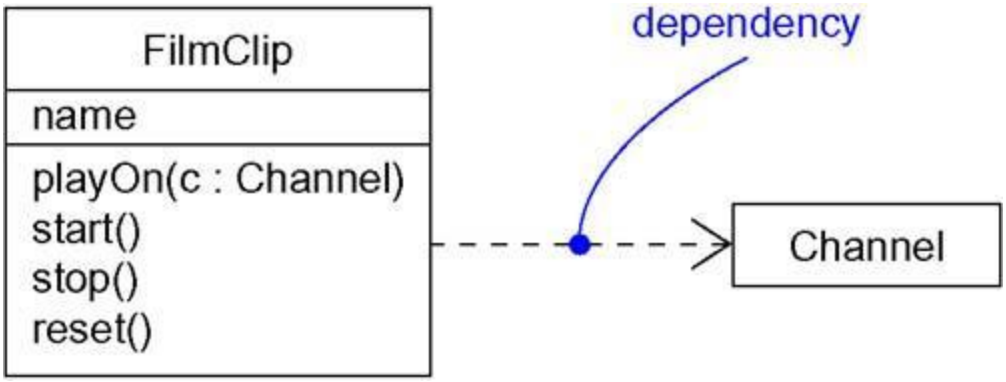


A *relationship* is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

Dependency

A *dependency* is a using relationship that states that a change in specification of one thing (for example, class Event) may affect another thing that uses it (for example, class Window), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.

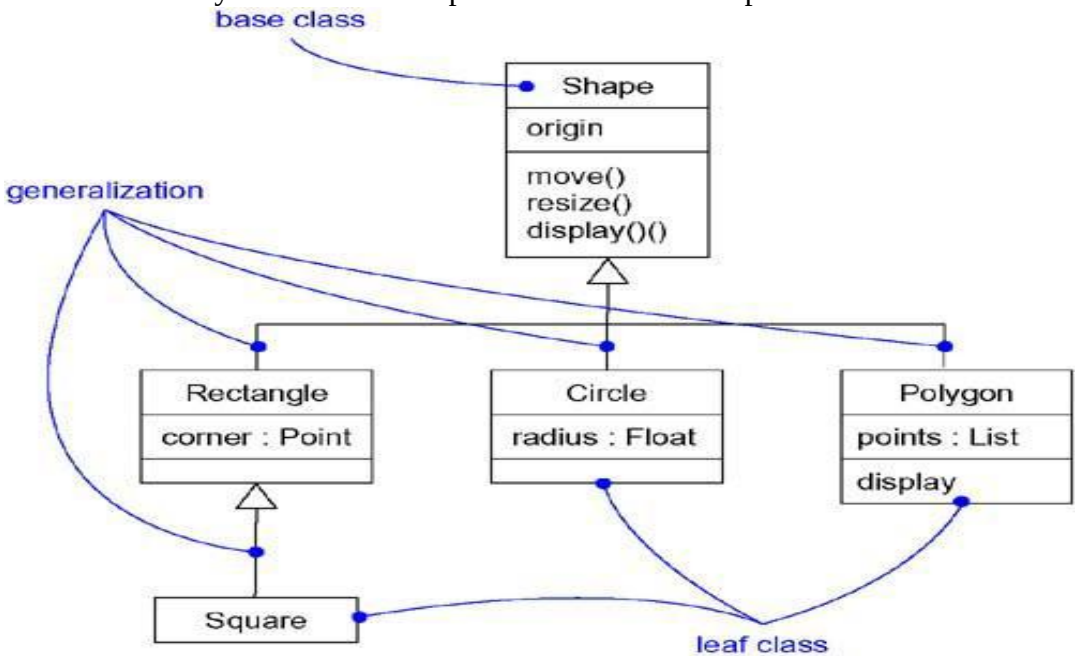
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation; see Figure 5-2. This is very much a using relationship• if the used class changes, the operation of the other class may be affected, as well, because the used class may now present a different interface or behavior. In the UML you can also create dependencies among many other things, especially notes and packages.



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). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often• but not always• the child has attributes and operations in addition to those found in its parents. An operation of a

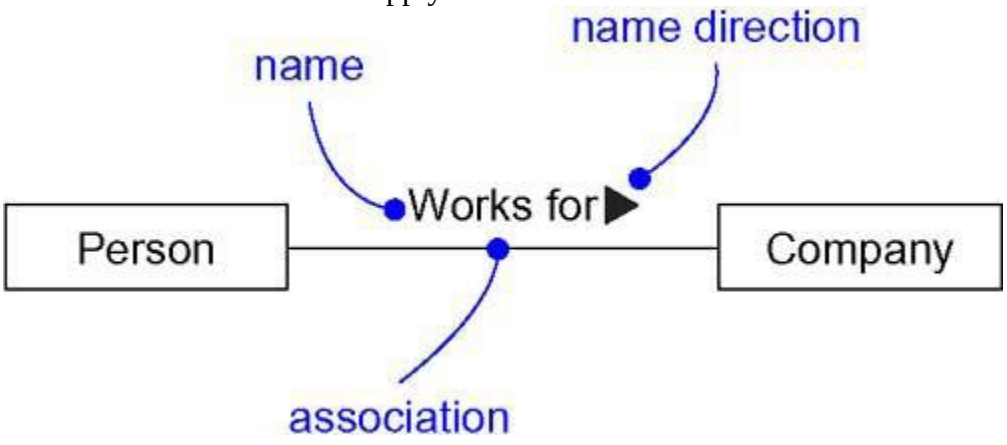
child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in Figure 5-3. Use generalizations when you want to show parent/child relationships.



A class may 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 class. 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

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have 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. Use associations when you want to show structural relationships. Beyond this basic form, there are four adornments that apply to associations.



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. This "how many" is called the multiplicity of an association's role, and is written as an expression that evaluates to a range of values or an explicit value as in Figure 5-6. When you state a multiplicity at one end of an association, you are specifying that, for each object of the class at the opposite end, there must be that many objects at the near end.

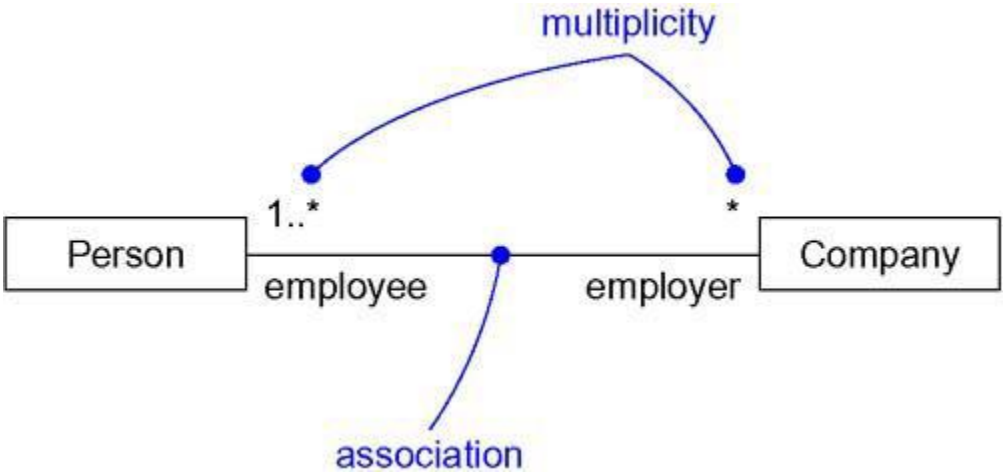


Figure 5-6 Multiplicity

Aggregation

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes, you will want 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, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part. Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end, as shown in Figure 5-7.

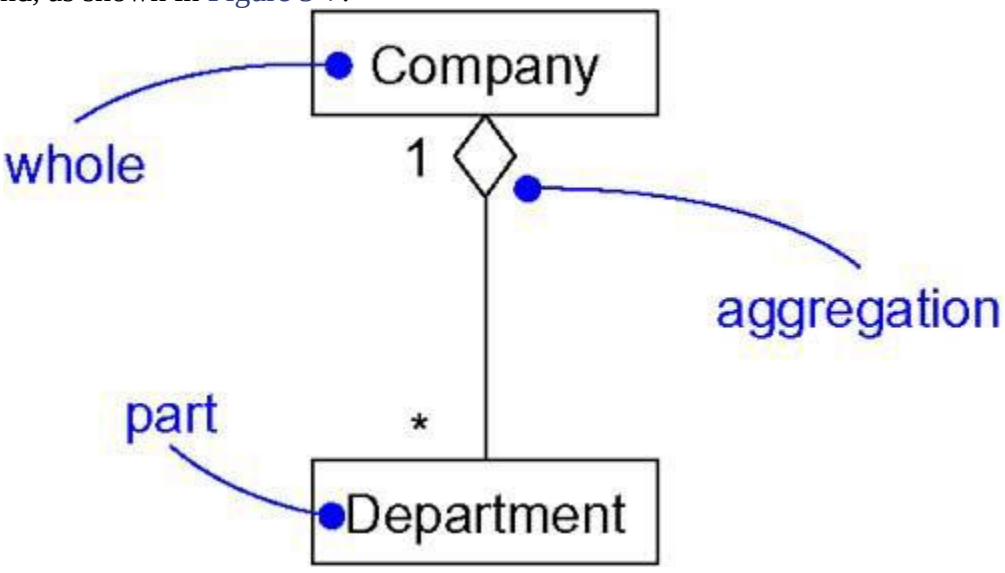


Figure 5-7 Aggregation

Common Mechanisms

The UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language: specifications, adornments, common divisions, and extensibility mechanisms. This chapter explains the use of two of these common mechanisms, adornments and extensibility mechanisms.

Notes are the most important kind of adornment that stands alone. A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. You use notes to attach information to a model, such as requirements, observations, reviews, and explanations.

The UML's extensibility mechanisms permit you to extend the language in controlled ways. These mechanisms include stereotypes, tagged values, and constraints. A stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification. A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. You use these mechanisms to tailor the UML to the specific needs of your domain and your development culture.

The UML provides a graphical representation for comments and constraints, called a note, as Figure 6-1 shows. This notation permits you to visualize a comment directly. In conjunction with the proper tools, notes also give you a placeholder to link to or embed other documents.

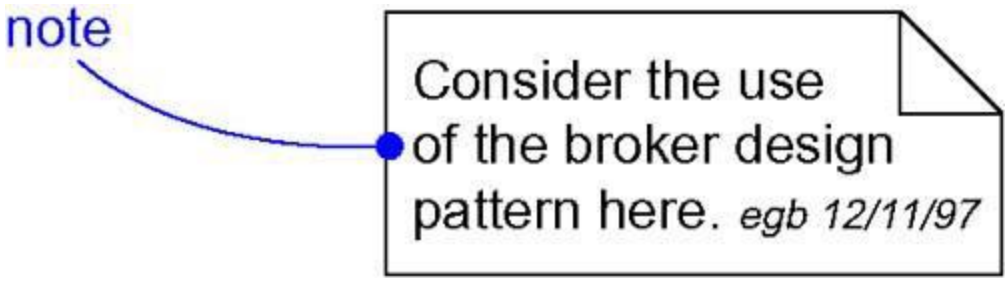


Figure 6-1 Notes

Stereotypes, tagged values, and constraints are the mechanisms provided by the UML to let you add new building blocks, create new properties, and specify new semantics. For example, if you are modeling a network, you might want to have symbols for routers and hubs; you can use stereotyped nodes to make these things appear as primitive building blocks. Similarly, if you are part of your project's release team, responsible for assembling, testing, and then deploying releases, you might want to keep track of the version number and test results for each major subsystem. You can use tagged values to add this information to your models. Finally, if you are modeling hard real time systems, you might want to adorn your models with information about time budgets and deadlines; you can use constraints to capture these timing requirements.

The UML provides a textual representation for stereotypes, tagged values, and constraints, as

Figure 6-2 shows. Stereotypes also let you introduce new graphical symbols so that you can provide visual cues to your models that speak the language of your domain and your development culture.

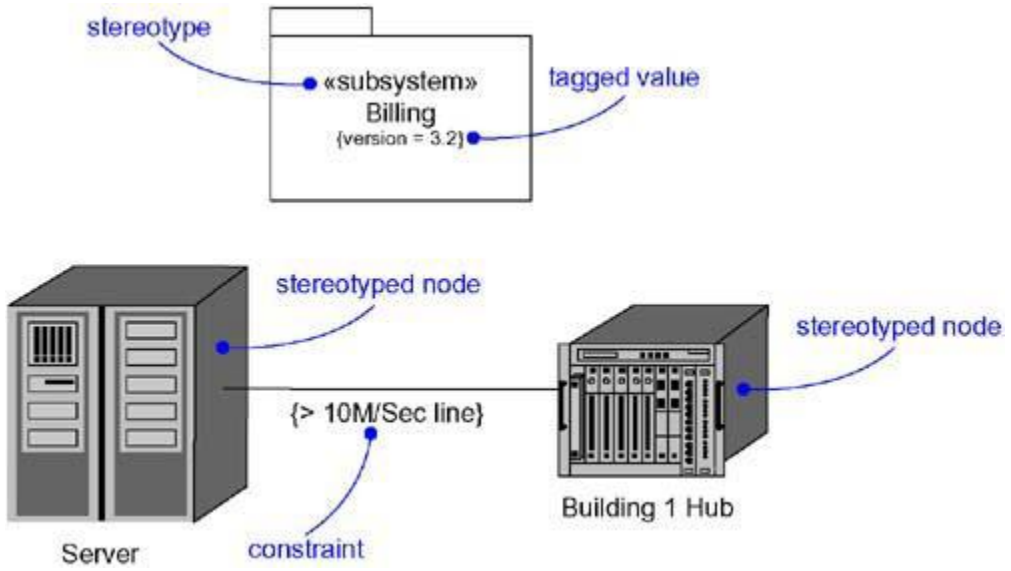


Figure 6-2 Stereotypes, Tagged Values, and Constraints

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 *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. As an option, the stereotyped element may be rendered by using a new icon associated with that stereotype.

A *tagged value* is an extension of the properties of a UML element, allowing you to create new information in that element's specification. Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.

A *constraint* is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones. 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. As an alternative, you can render a constraint in a note.

Stereotypes

The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model. However, sometimes you'll want to introduce new things that speak the vocabulary of your domain and look like primitive building blocks.

A stereotype is not the same as a parent class in a parent/child generalization relationship. Rather, you can think of a stereotype as a metatype, because each one creates the equivalent of a new class in the UML's metamodel. For example, if you are modeling a business process, you'll want to introduce things like workers, documents, and policies. Similarly, if you are

following a development process, such as the Rational Unified Process, you'll want to model using boundary, control, and entity classes. This is where the real value of stereotypes comes in. When you stereotype an element such as a node or a class, you are in effect extending the UML by creating a new building block just like an existing one but with its own special properties (each stereotype may provide its own set of tagged values), semantics (each stereotype may provide its own constraints), and notation (each stereotype may provide its own icon).

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, »named«) and placed above the name of another element. As a visual cue, you may define an icon for the stereotype and render that icon to the right of the name (if you are using the basic notation for the element) or use that icon as the basic symbol for the stereotyped item. All three of these approaches are illustrated in Figure 6-5.

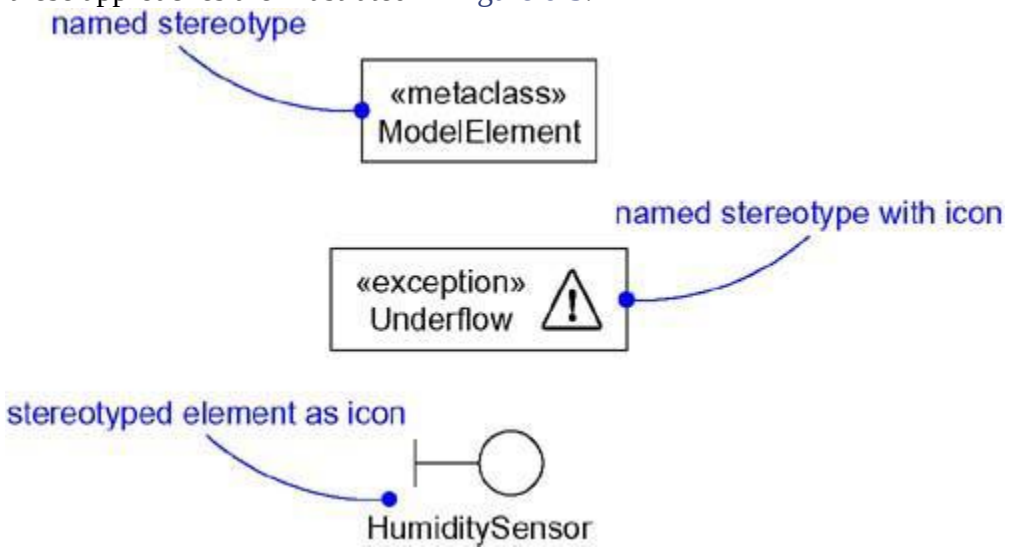


Figure 6-5 Stereotypes

Tagged Values

Every thing 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.

You can define tags for existing elements of the UML, or you can define tags that apply to individual stereotypes so that everything with that stereotype has that tagged value. A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances. For example, as Figure 6-6 shows, you might want to specify the number of processors installed on each kind of node in a deployment diagram, or you might want to require that every component be stereotyped as a library if it is intended to be deployed on a client or a server.

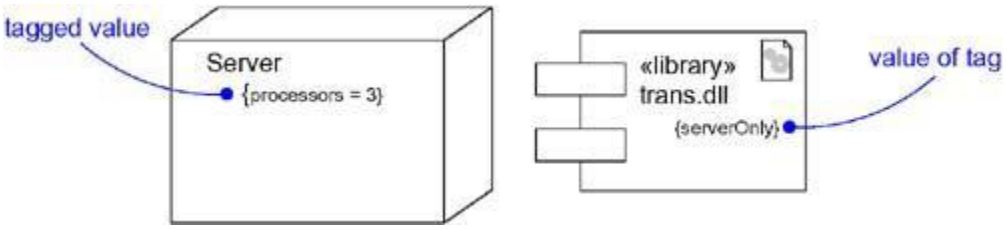


Figure 6-6 Tagged Values

Constraints

Everything in the UML has its own semantics. Generalization implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or change existing rules. A constraint specifies conditions that must be held true for the model to be well-formed. For example, as Figure 6-7 shows, you might want to specify that, across a given association, communication is encrypted. Similarly, you might want to specify that among a set of associations, only one is manifest at a time.

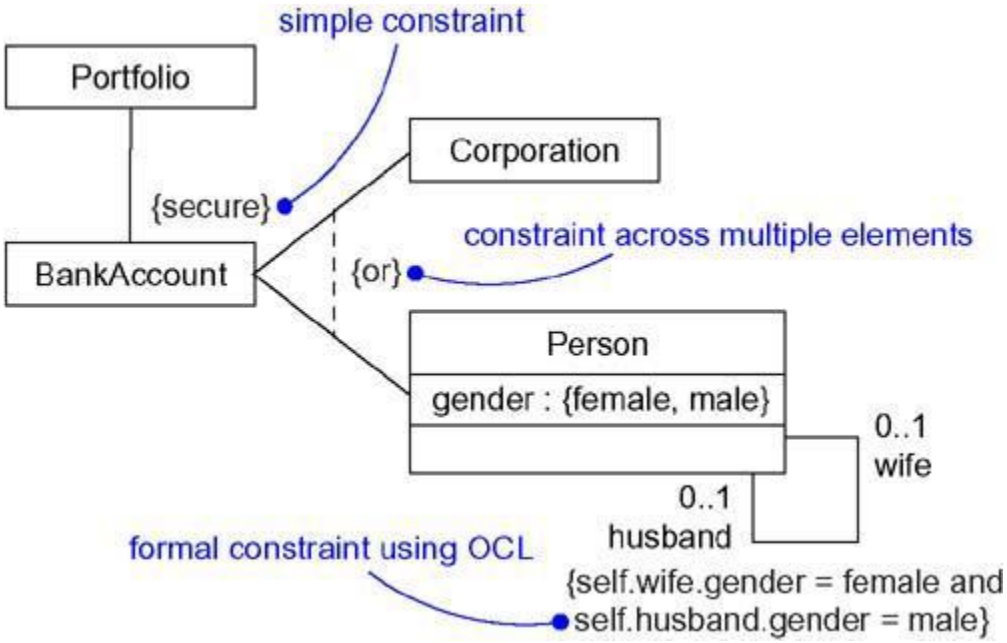


Figure 6-7 Constraint

Class Diagram

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. You use class diagrams 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.

Object Diagram

An *object diagram* shows a set of objects and their relationships. You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams. 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.