UNIT-I

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing Complex Systems.

COMPLEXITY

Systems: Systems are constructed by interconnecting components (Boundaries, Environments, Characters, Emergent Properties), which may well be systems in their own right. The larger the number of these components and relationships between them, higher will be the complexity of the overall system.

Software Systems: Software systems are not any different from other systems with respect to these characteristics. Thus, they are also embedded within some operational environment, and perform operations which are clearly defined and distinguished from the operations of other systems in this environment. They also have properties which emerge from the interactions of their components and/or the interactions of themselves with other systems in their environment. A system that embodies one or more software subsystems which contribute to or control a significant part of its overall behavior is what we call a software intensive system. As examples of complex software-intensive systems, we may consider stock and production control systems, aviation systems, rail systems, banking systems, health care systems and so on.

Complexity: Complexity depends on the number of the components embedded in them as well as the relationships and the interactions between these components which carry;

- Impossible for humans to comprehend fully
- Difficult to document and test
- Potentially inconsistent or incomplete
- Subject to change
- No fundamental laws to explain phenomena and approaches

THE STRUCTURE OF COMPLEX SYSTEMS

Examples of Complex Systems: The structure of personal computer, plants and animals, matter, social institutions are some examples of complex system.

The structure of a Personal Computer: A personal computer is a device of moderate complexity. Major elements are CPU, monitor, keyboard and some secondary storage devices. CPU encompasses primary memory, an ALU, and a bus to which peripheral devices are attached. An ALU may be divided into registers which are constructed from NAND gates, inverters and so on. All are the hierarchical nature of a complex system.

The structure of Plants: Plants are complex multicellular organism which are composed of cells which is turn encompasses elements such as chloroplasts, nucleus, and so on. For example, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil. Roots interact with stems, which transport these raw materials up to the leaves. The leaves in turn use water and minerals provided by stems to produce food through photosynthesis.

The structure of Animals: Animals exhibit a multicultural hierarchical structure in which collection of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system) and so on.

The structure of Matter: Nuclear physicists are concerned with a structural hierarchy of matter. Atoms are made up of electrons, protons and neutrons. Elements and elementary particles but protons, neutrons and other particles are formed from more basic components called quarks, which eventually formed from pro-quarks.

The structure of Social institutions: In social institutions, group of people join together to accomplish tasks that can not be done by made of divisions which in turn contain branches which in turn encompass local offices and so on.

THE INHERENT COMPLEXITY OF SOFTWARE

The Properties of Complex and Simple Software Systems: Software may involve elements of great complexity which is of different kind.

Some software systems are simple.

- These are the largely forgettable applications that are specified, constructed, maintained,

and used by the same person, usually the amateur programmer or the professional developer working in isolation.

- Such systems tend to have a very limited purpose and a very short life span.
- We can afford to throw them away and replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality, Such applications are generally more tedious than difficult to develop; consequently, learning how to design them does not interest us.

Some software systems are complex.

- The applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries; and systems for the command and control of real-world entities, such as the routing of air or railway traffic.
- Software systems such as world of industrial strength software tend to have a long life span, and over time, many users come to depend upon their proper functioning.
- The frameworks that simplify the creation of domain-specific applications, and programs that mimic some aspect of human intelligence.
- Although such applications are generally products of research and development they are no less complex, for they are the means and artifacts of incremental and exploratory development.

Why Software is inherently Complex

The complexity of software is an essential property not an accidental one. The inherent complexity derives from four elements; the complexity of the problem domain, the difficultly of managing the developmental process, the flexibility possible through software and the problems of characterizing the behavior of discrete systems.

1. The complexity of the problem domain

- Complex requirements
- Decay of system

The first reason has to do with the relationship between the application domains for which software systems are being constructed and the people who develop them. Often, although

software developers have the knowledge and skills required to develop software they usually lack detailed knowledge of the application domain of such systems. This affects their ability to understand and express accurately the requirements for the system to be built which come from the particular domain. Note, that these requirements are usually themselves subject to change. They evolve during the construction of the system as well as after its delivery and thereby they impose a need for a continuous evolution of the system. Complexity is often increased as a result of trying to preserve the investments made in legacy applications. In such cases, the components which address new requirements have to be integrated with existing legacy applications. This results into interoperability problems caused by the heterogeneity of the different components which introduce new complexities.

Consider the requirement for the electronic systems of a multi-engine aircraft, a cellular phone switching system or a cautious (traditional) robot. The row functionality of such systems is difficult enough to comprehend. External complexity usually springs from the impedance mismatch that exists between the users of a system and its developers. Users may have only vague ideas of what they want in a software system. Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the system. A further complication is that the requirement of a software system is often change during its development. Once system is installed, the process helps developers master the problem domain, enabling them to ask better questions that illuminate the done existing system every time its requirements change because a large software system is a capital investment. It is software maintenance when we correct errors, evolution when we respond to changing environments and preservations, when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation.

2. The Difficulty of Managing the Development Process

- Management problems
- Need of simplicity

The second reason is the complexity of the software development process. Complex software intensive systems cannot be developed by single individuals. They require teams of developers. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other.

This complexity often gets even more difficult to handle if the teams do not work in one location

but are geographically dispersed. In such situations, the management of these processes becomes an important subtask on its own and they need to be kept as simple as possible.

None person can understand the system whose size is measured in hundreds of thousands, or even millions of lines of code. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes even thousand modules. The amount of work demands that we use a team of developers and there are always significant challenges associated with team development more developer's means more complex communication and hence more difficult coordination.

3. The flexibility possible through software

- Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess

The third reason is the danger of flexibility. Flexibility leads to an attitude where developers develop system components themselves rather than purchasing them from somewhere else. Unlike other industrial sectors, the production depth of the software industry is very large. The construction or automobile industries largely rely on highly specialized suppliers providing parts. The developers in these industries just produce the design, the part specifications and assemble the parts delivered. The software development is different: most of the software companies develop every single component from scratch. Flexibility also triggers more demanding requirements which make products even more complicated.

Software offers the ultimate flexibility. It is highly unusual for a construction firm to build an onsite steel mill to forge (create with hammer) custom girders (beams) for a new building. Construction industry has standards for quality of row materials, few such standards exist in the software industry.

4. The problem of characterizing the behavior of discrete systems

- Numerous possible states
- Difficult to express all states

The final reason for complexity according to Booch is related to the difficulty in describing the behavior of software systems. Humans are capable of describing the static structure and properties of complex systems if they are properly decomposed, but have problems in describing their behavior. This is because to describe behavior, it is not sufficient to list the

properties of the system. It is also necessary to describe the sequence of the values that these properties take over time.

Within a large application, there may be hundreds or even thousands of variables well as more than one thread of control. The entire collection of these variables as well as their current values and the current address within the system constitute the present state of the system with discrete states. Discrete systems by their very nature have a finite number of possible states.

The Consequences of Unrestrained Complexity

The more complex the system, the more open it is to total breakdown. Rarely would a builder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition the software crisis, but frankly, a malady that has carried on this long must be called normal. Sadly, this crisis translates into the squandering of human resources - a most precious commodity - as well as a considerable loss of opportunities. There are simply not enough good developers around to create all the new software that users need. Furthermore, a significant number of the developmental personnel in any given organization must often be dedicated to the maintenance or preservation of geriatric software. Given the indirect as well as the direct contribution of software to the economic base of most industrialized countries, and considering the ways in which software can amplify the powers of the individual, it is unacceptable to allow this situation to continue.

THE FIVE ATTRIBUTES OF A COMPLEX SYSTEM

There are five attribute common to all complex systems. They are as follows:

1. Hierarchical and interacting subsystems

Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems and so on, until some lowest level of elementary components is reached.

2. Arbitrary determination of primitive components

The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system class structure and the object structure are not completely independent each object in object structure represents a specific instance of some class.

3. Stronger intra-component than inter-component link

Intra-component linkages are generally stronger than inter-component linkages. This fact has the involving the high frequency dynamics of the components-involving the internal structure of the components – from the low frequency dynamic involving interaction among components.

4. Combine and arrange common rearranging subsystems

Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements. In other words, complex systems have common patterns. These patterns may involve the reuse of small components such as the cells found in both plants and animals, or of larger structures, such as vascular systems, also found in both plants and animals.

5. Evolution from simple to complex systems

A complex system that works is invariably bound to have evolved from a simple system that worked.....A complex system designed from scratch never works and can't be patched up to make it work. You have to start over, beginning with a working simple system.

Booch has identified five properties that architectures of complex software systems have in common.

Firstly, every complex system is decomposed into a hierarchy of subsystems. This decomposition is essential in order to keep the complexity of the overall system manageable. These subsystems, however, are not isolated from each other, but interact with each other.

Very often subsystems are decomposed again into subsystems, which are decomposed and so on. The way how this decomposition is done and when it is stopped, i.e. which components are considered primitive, is rather arbitrary and subject to the architects decision.

The decomposition should be chosen, such that most of the coupling is between components that lie in the same subsystem and only a loose coupling exists between components of different subsystem. This is partly motivated by the fact that often different individuals are in charge with the creation and maintenance of subsystems and every additional link to other subsystems does imply an higher communication and coordination overhead.

Certain design patterns re-appear in every single subsystem. Examples are patterns for iterating

over collections of elements, or patterns for the creation of object instances and the like.

The development of the complete system should be done in slices so that there is an increasing number of subsystems that work together. This facilitates the provision of feedback about the overall architecture.

ORGANIZED AND DISORGANIZED COMPLEXITY

Simplifying Complex Systems

- Usefulness of abstractions common to similar activities

e.g. driving different kinds of motor vehicle

- Multiple orthogonal hierarchies

e.g. structure and control system

- Prominent hierarchies in object-orientation " class structure " " object structure "

e. g. engine types, engine in a specific car

One mechanism to simplify concerns in order to make them more manageable is to identify and understand abstractions common to similar objects or activities. We can use a car as an example (which are considerable complex systems). Understanding common abstractions in this particular example would, for instance, involve the insight that clutch, accelerator and brakes facilitate the use of a wide range of devices, namely transport vehicles depending on transmission of power from engine to wheels).

Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other. In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, and electrical). In object-orientation, the class structure and the object structure relationship is the simplest form of related hierarchy. It forms a canonical representation for object oriented analysis. The canonical form of a complex system – the discovery of common abstractions and mechanisms greatly facilitates are standing of complex system. For example, if a pilot already knows how to fly a given aircraft, it is easier to know how to fly a similar one. May different hierarchies are present within the complex system. For example an aircraft may be studied by

decomposing it into its propulsion system. Flight control system and so on the decomposition represent a structural or "part of" hierarchy. The complex system also includes an "Is A" hierarchy. These hierodules for class structure and object structure combining the concept of the class and object structure together with the five attributes of complex system, we find that virtually all complex system take on the same (canonical) form as shown in figure. There are two orthogonal hierarchies of system, its class structure and the object structure.

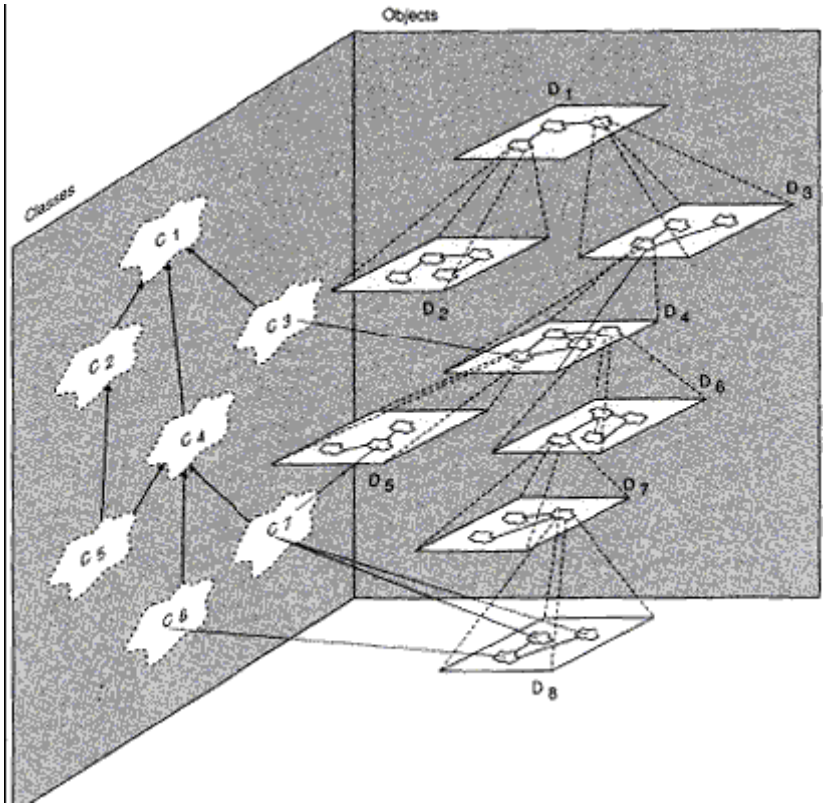


Figure: Canonical form of a complex system

The figure represents the relationship between two different hierarchies: a hierarchy of objects and a hierarchy of classes. The class structure defines the 'is-a' hierarchy, identifying the commonalities between different classes at different levels of abstractions. Hence class C4 is also a class C1 and therefore has every single property that C1 has. C4, however, may have more specific properties that C1 does not have; hence the distinction between C1 and C4. The object structure defines the 'part-of' representation. This identifies the composition of an object from component objects, like a car is composed from wheels, a steering wheel, a chassis and an engine. The two hierarchies are not entirely orthogonal as objects are instances of certain

classes. The relationship between these two hierarchies is shown by identifying the instance-of relationship as well. The objects in component D8 are instances of C6 and C7 As suggested by the diagram, there are many more objects then there are classes. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

APPROACHING A SOLUTION

Hampered by human limitations

- dealing with complexities
- memory
- communications

When we devise a methodology for the analysis and design of complex systems, we need to bear in mind the limitations of human beings, who will be the main acting agents, especially during early phases. Unlike computers, human beings are rather limited in dealing with complex problems and any method need to bear that in mind and give as much support as possible.

Human beings are able to understand and remember fairly complex diagrams, though linear notations expressing the same concepts are not dealt with so easily. This is why many methods rely on diagramming techniques as a basis. The human mind is also rather limited. Miller revealed in 1956 that humans can only remember 7 plus or minus one item at once. Methods should therefore encourage its users to bear these limitations in mind and not deploy overly complex diagrams.

The analysis process is a communication intensive process where the analyst has to have intensive communications with the stakeholders who hold the domain knowledge. Also the design process is a communication intensive process, since the different agents involved in the design need to agree on decompositions of the system into different hierarchies that are consistent with each other.

The Limitations of the human capacity for dealing with complexity: Object model is the organized complexity of software. As we begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways with little commonality among either the parts or their interactions. This is an example of disorganized complexity. In complex system, we find many parts that must interact in a multitude of intricate ways with little

commonality among either the parts or their intricate. This is an example in an air traffic control system, we must deal with states of different aircraft at once, and involving such it is absolutely impossible for a single person to keep track of all these details at once.

BRINGING ORDER TO CHAOS

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

The Role of Abstraction: Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object. For example, when studying about how photosynthesis works in a plant, we can focus upon the chemical reactions in certain cells in a leaf and ignore all other parts such as roots and stems. Objects are abstractions of entities in the real world.

In general abstraction assists people's understanding by grouping, generalizing and chunking information.

Object-orientation attempts to deploy abstraction. The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an ‘is-a’ relationship defines the inheritance between these classes.

The role of Hierarchy: Identifying the hierarchies within a complex software system makes understanding of the system very simple. The object structure is important because it illustrates how different objects collaborate with one another through pattern of interaction (called mechanisms). By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells, we come to explicitly distinguish the common and distinct properties of different objects, which helps to master their inherent complexity.

Different hierarchies support the recognition of higher and lower orders. A class high in the ‘is-a’ hierarchy is a rather abstract concept and a class that is a leaf represents a fairly concrete concept. The ‘is-a’ hierarchy also identifies concepts, such as attributes or operations, that are

common to a number of classes and instances thereof. Similarly, an object that is up in the part-of hierarchy represents a rather coarse-grained and complex objects, assembled from a number of objects, while objects that are leafs are rather fine grained. But note that there are many other forms of patterns which are nonhierarchical: interactions, 'relationships'.

The role of Decomposition: Decomposition is important techniques for copying with complexity based on the idea of divide and conquer. In dividing a problem into a sub problem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to sub problems that are small enough so that they can be conquered. After all the sub problems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem. The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

Algorithmic (Process Oriented) Decomposition: In Algorithmic decomposition, each module in the system denotes a major step in some ovrall process.

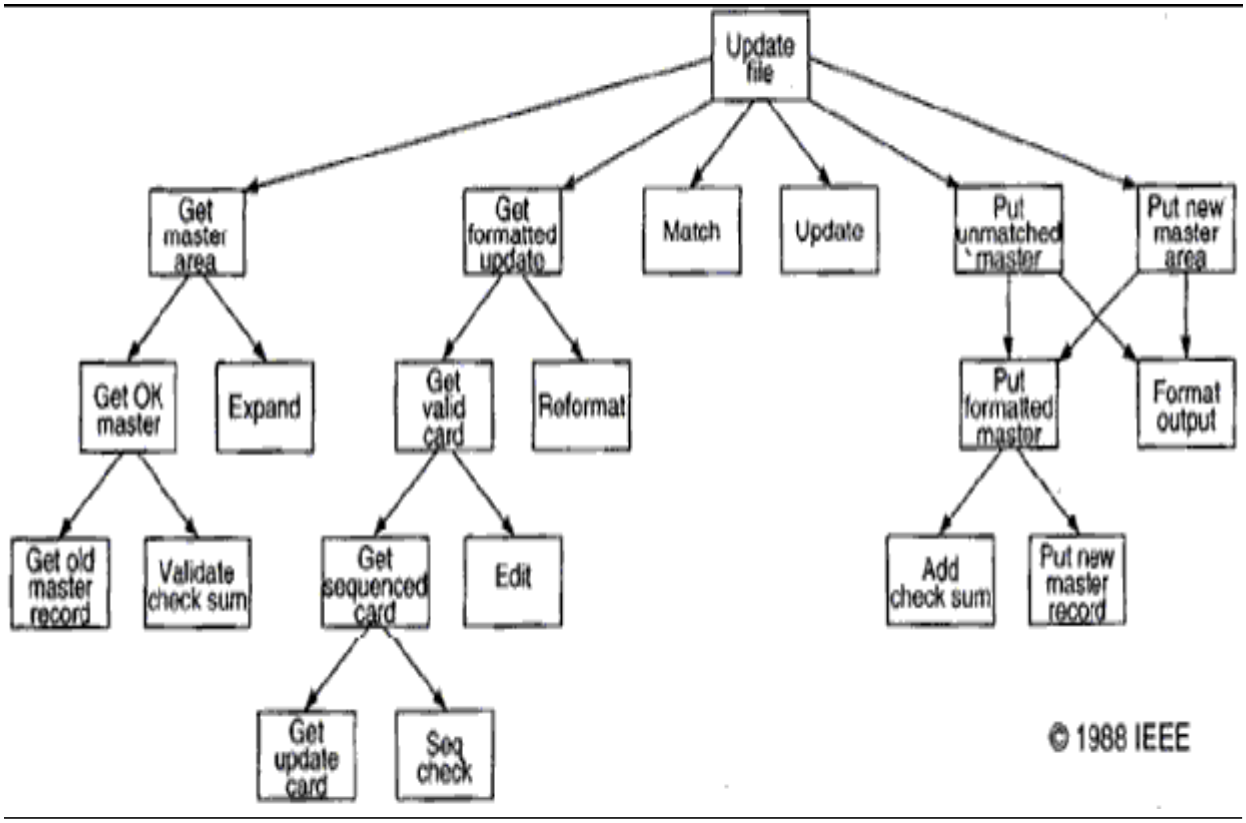


Figure: Algorithmic decomposition

Object oriented decomposition: Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem as shown in figure. We know the world as a set of autonomous agents that collaborate to perform some higher level behavior. Get formatted update thus does not exist as an independent algorithm; rather it is an operation associated with the object file of updates. Calling this operation creates another object, update to card. In this manner, each object in our solution embodies its own unique behavior .Each hierarchy in layered with the more abstract classes and objects built upon more primitive ones especially among the parts of the object structure, object in the real world. Here decomposition is based on objects and not algorithms.

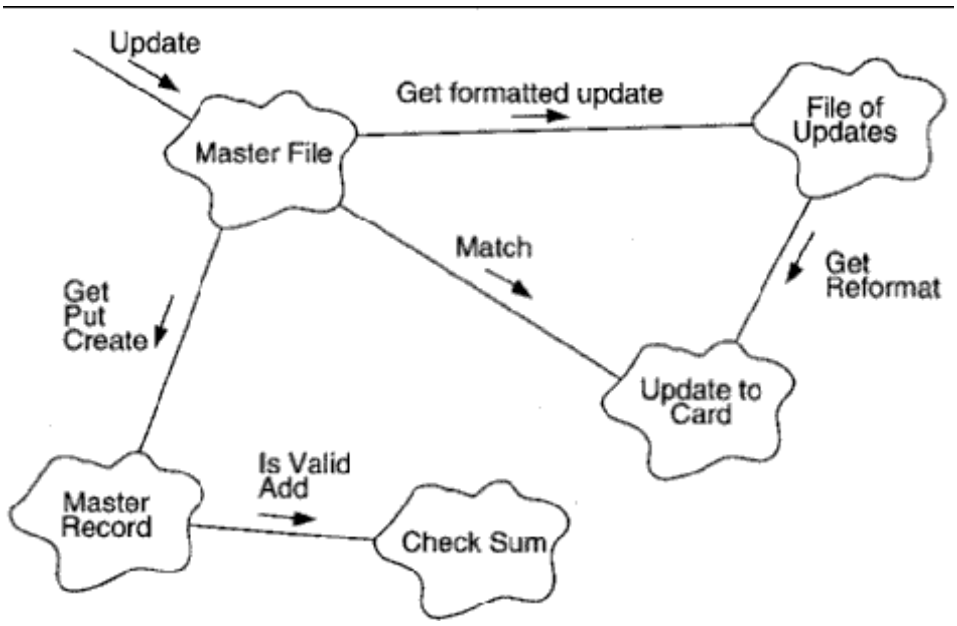


Figure: Object oriented decomposition

Algorithmic versus object-oriented decomposition: The algorithmic view highlights the ordering of events and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act. We must start decomposing a system either by algorithms or by objects then use the resulting structure as the framework for expressing the other perspective generally object-oriented view is applied because this approach is better at helping us organize the inherent complexity of software systems. Object oriented algorithm has a number of advantages over algorithmic decomposition. Object oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important

economy of expression and are also more resident to change and thus better able to involve over time and it also reduces risks of building complex software systems. Object oriented decomposition also directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

Process-oriented decompositions divide a complex process, function or task into simpler sub processes until they are simple enough to be dealt with. The solutions of these sub functions then need to be executed in certain sequential or parallel orders in order to obtain a solution to the complex process. Object-oriented decomposition aims at identifying individual autonomous objects that encapsulate both a state and a certain behavior. Then communication among these objects leads to the desired solutions.

Although both solutions help dealing with complexity, we have reasons to believe that an object-oriented decomposition is favorable because, the object-oriented approach provides for a semantically richer framework that leads to decompositions that are more closely related to entities from the real world. Moreover, the identification of abstractions supports (more abstract) solutions to be reused and the object-oriented approach supports the evolution of systems better as those concepts that are more likely to change can be hidden within the objects.

ON DESIGNING COMPLEX SYSTEMS

Engineering as a Science and an Art: Every engineering discipline involves elements of both science and art. The programming challenge is a large-scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attribute of the competent engineer. The role of the engineer as artist is particularly challenging when the task is to design an entirely new system.

The meaning of Design: In every engineering discipline, design encompasses the discipline approach we use to invent a solution for some problem, thus providing a path from requirements to implementation. The purpose of design is to construct a system that.

1. Satisfies a given (perhaps) informal functional specification
2. Conforms to limitations of the target medium
3. Meets implicit or explicit requirements on performance and resource usage
4. Satisfies implicit or explicit design criteria on the form of the artifact
5. Satisfies restrictions on the design process itself, such as its length or cost, or the available

for doing the design.

According to Stroustrup, the purpose of design is to create a clean and relatively simple internal structure, sometimes also called as architecture. A design is the end product of the design process.

The Importance of Model Building: The buildings of models have a broad acceptance among all engineering disciplines largely because model building appeals to the principles of decomposition, abstraction and hierarchy. Each model within a design describes a specific aspect of the system under consideration. Models give us the opportunity to fail under controlled conditions. We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

The Elements of Software design Methods: Design of complex software system involves an incremental and iterative process. Each method includes the following:

1. **Notation:** The language for expressing each model.
2. **Process:** The activities leading to the orderly construction of the system's mode.
3. **Tools:** The artifacts that eliminate the medium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed.

The models of Object-Oriented Development: The models of object-oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design. These models also over the spectrum of the important design decisions that we must consider in developing a complex system and so encourage us to craft implementations that embody the five attributes of well-formed complex systems.

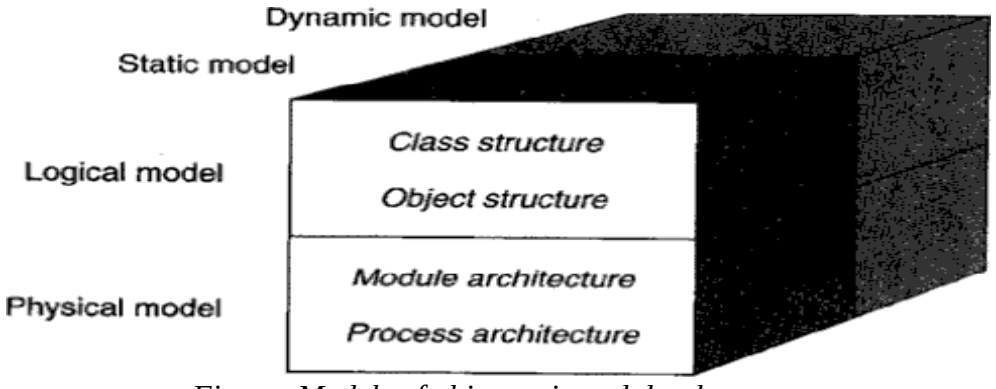


Figure: Models of object-oriented development

Booch presents a model of object-oriented development that identifies several relevant perspectives. The classes and objects that form the system are identified in a logical model. For this logical model, again two different perspectives have to be considered. A static perspective identifies the structure of classes and objects, their properties and the relationships classes and objects participate in. A dynamic model identifies the dynamic behavior of classes and objects, the different valid states they can be in and the transitions between these states.

Besides the logical model, also a physical model needs to be identified. This is usually done later in the system's lifecycle. The module architecture identifies how classes are kept in separately compliable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those. Again, for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication. Object-orientation has not, however, emerged fully formed. In fact, it has developed over a long period, and continues to change.

The elements of the object-oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

UNIT-II

Introduction to UML: Why we model, Conceptual model of UML, Architecture, Classes, Relationships, Common Mechanisms, Class diagrams, Object diagrams.

WHY WE MODEL

A model is a simplification at some level of abstraction

1. Importance of Modeling:

We build models to better understand the systems we are developing.

To help us visualize

To specify structure or behaviour

To provide template for building system

To document decisions we have made

2. Principles of Modeling:

The models we choose have a profound influence on the solution we provide Every model may be expressed at different levels of abstraction

The best models are connected to reality

No single model is sufficient, a set of models is needed to solve any nontrivial system

UML is a visual modeling language

"A picture is worth a thousand words." - old saying

Unified Modeling Language: "A language provides a vocabulary and the rules for combining words [...] for the purpose of communication.

A modeling language is a language whose vocabulary and rules focus on the conceptual and

physical representation of a system. A modeling language such as the UML is thus a standard

language for software blueprints."

Usages of UML: UML is used to

i. document designs

design patterns / frameworks

ii. Represent different views/aspects of design – visualize and construct designs static / dynamic / deployment / modular aspects

iii. Provide a next-to-precise, common, language –specify visually for the benefit of analysis, discussion, comprehension...

Object Oriented Modeling:

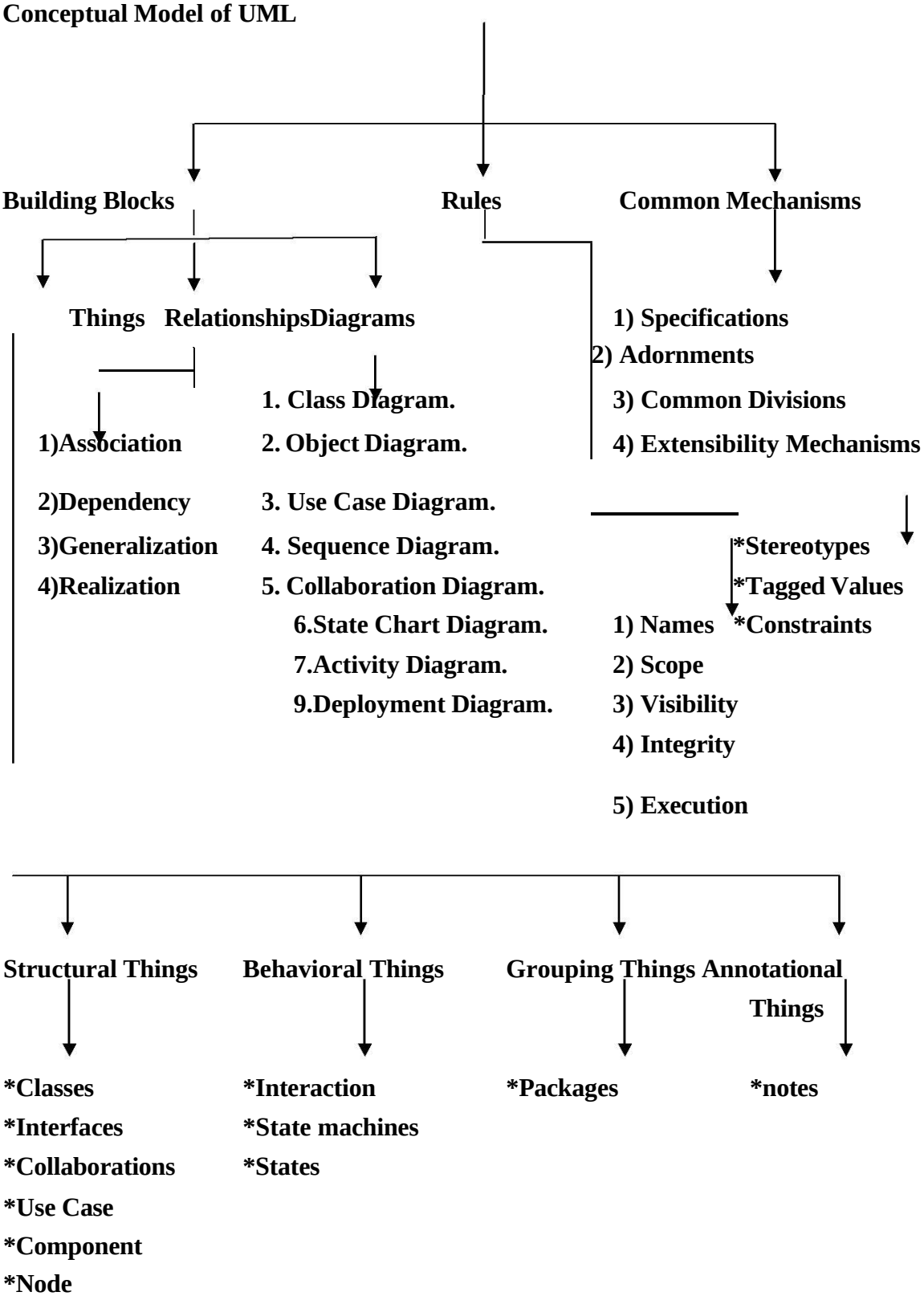
Traditionally two approaches to modeling a software system

Algorithmically – becomes hard to focus on as the requirements change

Object-oriented – models more closely real world entities

CONCEPTUAL MODEL OF THE UML

Conceptual Model of 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. You use them to write well-formed models.

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. Collectively, the structural things are called *classifiers*.

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, as shown in the following figure.

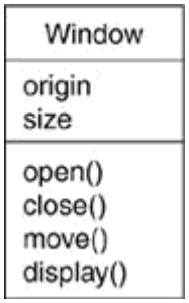
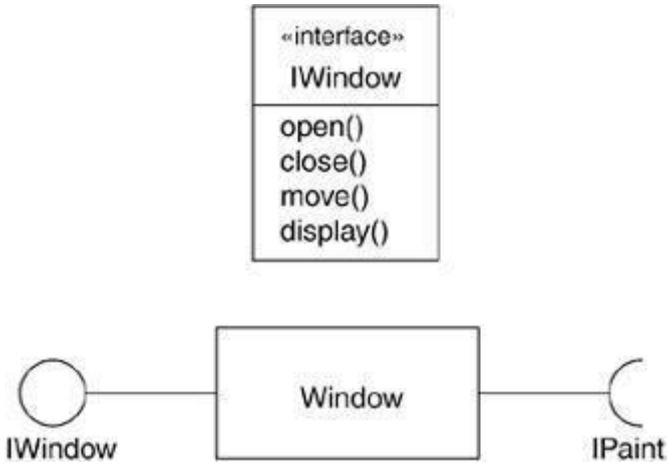


Figure: Classes

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. The declaration of an interface looks like a class with the keyword «interface» above the name; attributes are not relevant, except sometimes to show constants. An interface rarely stands alone, however. An interface provided by a class to the outside world is shown as a small circle attached to the class box by a line. An interface required by a class from some other class is shown as a small semicircle attached to the class box by a line, as shown in the following figure.

Figure: Interfaces



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. Collaborations have structural, as well as behavioral, dimensions. A given

class or object 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, sometimes including only its name, as shown in the following figure.

Figure: Collaborations

A *use case* is a description of sequences of actions that a system performs that yield observable results 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, as shown in the following figure.

Figure: Use Cases

The remaining three thingsactive classes, components, and nodesare all class-like, meaning they also describe sets of entities that share the same attributes, operations, relationships, and semantics. However, these three are different enough and are necessary for modeling certain aspects of an object-oriented system, so they warrant special treatment.

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 as a class with double lines on the left and right; it

usually includes its name, attributes, and operations, as shown in the following figure.

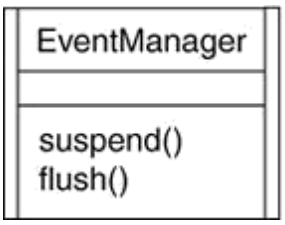
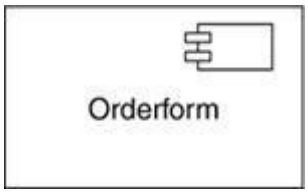


Figure: Active Classes

A component is a modular part of the system design that hides its implementation behind a set of external interfaces. Within a system, components sharing the same interfaces can be substituted while preserving the same logical behavior. The implementation of a component can be expressed by wiring together parts and connectors; the parts can include smaller components. Graphically, a component is rendered like a class with a special icon in the upper right corner, as shown in the following figure.

Figure: Components



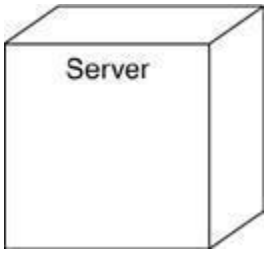
The remaining two elements' artifacts and nodes are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

An *artifact* is a physical and replaceable part of a system that contains physical information ("bits"). In a system, you'll encounter different kinds of deployment artifacts, such as source code files, executables, and scripts. An artifact typically represents the physical packaging of source or run-time information. Graphically, an artifact is rendered as a rectangle with the keyword «artifact» above the name, as shown in the following figure.

Figure: Artifacts

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, as shown in the following figure.

Figure: Nodes



These elements classes, interfaces, collaborations, use cases, active classes, components, artifacts, and nodes are the basic structural things that you may include in a UML model. There are also variations on these, such as actors, signals, and utilities (kinds of classes); processes and threads (kinds of active classes); and applications, documents, files, libraries, pages, and tables (kinds of artifacts).

Behavioral Things

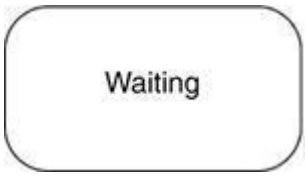
Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are three primary kinds of behavioral things.

First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects or roles 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, actions, and connectors (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as shown in the following figure.

Figure: Messages

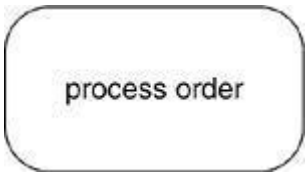
Second, a *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state 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 shown in the following figure.

Figure: States



Third, an activity is a behavior that specifies the sequence of steps a computational process performs. In an interaction, the focus is on the set of objects that interact. In a state machine, the focus is on the life cycle of one object at a time. In an activity, the focus is on the flows among steps without regard to which object performs each step. A step of an activity is called an *action*. Graphically, an action is rendered as a rounded rectangle with a name indicating its purpose. States and actions are distinguished by their different contexts.

Figure: Actions



These three elements interactions, state machines, and activities 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.

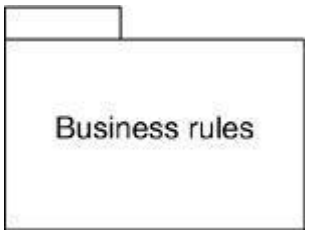
Grouping Things

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

A *package* is a general-purpose mechanism for organizing the design itself, as opposed to classes, which organize implementation constructs. 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, as shown in the following figure.

Figure: Packages

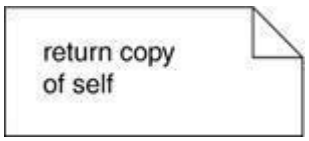


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

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, as shown in the following figure.

Figure: 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

These relationships are the basic relational building blocks of the UML. You use them to write

well-formed models.

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

Figure: Dependencies

Second, an *association* is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes. 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 end names, as shown in the following figure.

Figure: Associations

Third, a *generalization* is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). 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, as shown in the following figure.

Figure: 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, as shown in the following figure.

Figure: Realizations

These four elements are the basic relational things you may include in a UML model. There are

also variations on these four, such as refinement, trace, include, and extend.

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 paths (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 thirteen kinds of diagrams:

1. Class diagram
2. Object diagram
3. Component diagram
4. Composite structure diagram
5. Use case diagram
6. Sequence diagram
7. Communication diagram
8. State diagram
9. Activity diagram
10. Deployment diagram
11. Package diagram
12. Timing diagram
13. Interaction overview 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. Component diagrams are variants of class diagrams.

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 *component diagram* is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts. (UML distinguishes a composite structure diagram, applicable to any class, from a component diagram, but we combine the discussion because the distinction between a component and a structured class is unnecessarily subtle.)

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 communication diagrams are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects or roles, 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 *communication diagram* is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages. Sequence diagrams and communication diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts. Sequence diagrams emphasize temporal ordering, and communication diagrams emphasize the data structure through which messages flow. A timing diagram (not covered in this book) shows the actual times at which messages are exchanged.

A *state diagram* shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object. 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* shows the structure of a process or other computation as the flow of control and data from step to step within the computation. 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 *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. A node typically hosts one or more artifacts.

An artifact diagram shows the physical constituents of a system on the computer. Artifacts include files, databases, and similar physical collections of bits. Artifacts are often used

in conjunction with deployment diagrams. Artifacts also show the classes and components that they implement. (UML treats artifact diagrams as a variety of deployment diagram, but we discuss them separately.)

A package diagram shows the decomposition of the model itself into organization units and their dependencies.

A timing diagram is an interaction diagram that shows actual times across different objects or roles, as opposed to just relative sequences of messages. An interaction overview diagram is a hybrid of an activity diagram and a sequence diagram. These diagrams have specialized uses and so are not discussed in this book. See the UML Reference Manual for more details.

This is not a closed list of diagrams. Tools may use the UML to provide other kinds of diagrams, although these are the most common ones that you will encounter in practice.

Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models.

The UML has syntactic and semantic rules for

- Names What you can call things, relationships, and diagrams
- Scope The context that gives specific meaning to a name
- Visibility How those names can be seen and used by others
- Integrity How things properly and consistently relate to one another
- Execution What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

- Elided Certain elements are hidden to simplify the view
- Incomplete Certain elements may be missing
- Inconsistent The integrity of the model is not guaranteed

These less-than-well-formed models are unavoidable as the details of a system unfold and

churn during the software development life cycle. The rules of the UML encourage you but do not force you to address the most important analysis, design, and implementation questions that push such models to become well-formed over time.

Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specifications

The UML's graphical notation to visualize a system; & the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications.

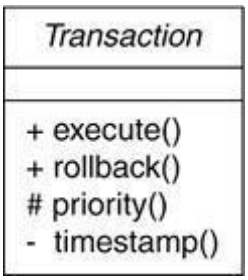
The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

Adornments

Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, the following figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

Figure: Adornments



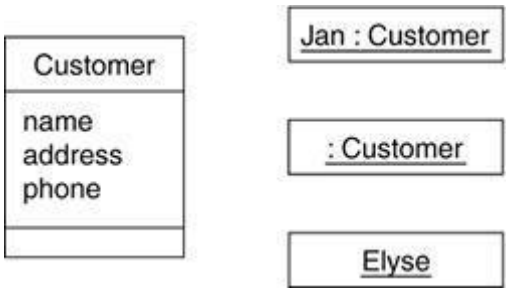
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

Common Divisions

In modeling object-oriented systems, the world often gets divided in several ways.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in the following figure. Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

Figure: Classes and Objects



In this figure, there is one class, named Customer, together with three objects: Jan (which is marked explicitly as being a customer object), Customer (an anonymous Customer object), and Elyse (which in its specification is marked as being a kind of Customer object, although it's not shown explicitly here).

Almost every building block in the UML has this same kind of class/object dichotomy. For example, you can have use cases and use case executions, components and component instances, nodes and node instances, and so on.

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in the following figure.

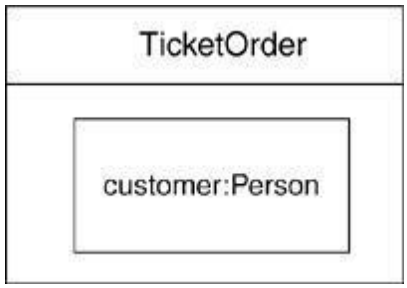
Figure: Interfaces and Implementations

In this figure, there is one component named SpellingWizard.dll that provides (implements) two interfaces, IUnknown and ISpelling. It also requires an interface, IDictionary, that must be provided by another component.

Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, you can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.

Third, there is the separation of type and role. The type declares the class of an entity, such as an object, an attribute, or a parameter. A role describes the meaning of an entity within its context, such as a class, component, or collaboration. Any entity that forms part of the structure of another entity, such as an attribute, has both characteristics: It derives some of its meaning from its inherent type and some of its meaning from its role within its context (below Figure).

Figure: Part with role and type



Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not

possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways. The UML's extensibility mechanisms include

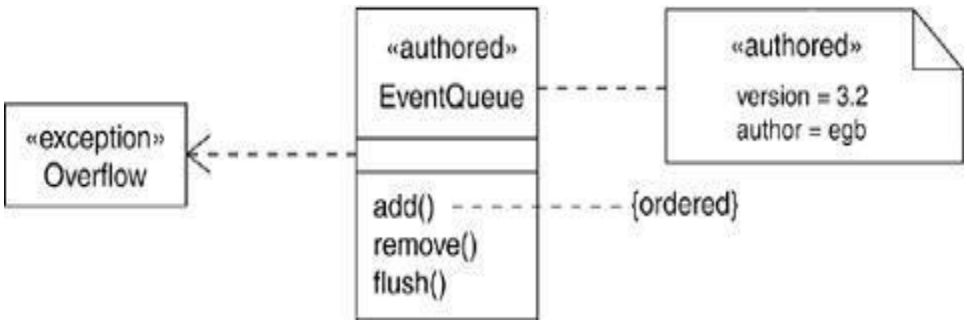
- Stereotypes
- Tagged values
- Constraints

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. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your modelsmeaning that they are treated like basic building blocksby marking them with an appropriate stereotype, as for the class Overflow in Figure 2-19.

A *tagged value* extends the properties of a UML stereotype, allowing you to create new information in the stereotype's specification. For example, if you are working on a shrink- wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. For example, the class EventQueue is extended by marking its version and author explicitly.

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As shown in the following figure shows, you can add a constraint that explicitly marks these for the operation add.

Figure: Extensibility Mechanisms



Collectively, these three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that you do so in controlled ways so that through these extensions, you remain true to the UML's purpose the communication of information.

ARCHITECTURE

Any real world system is used by different users. The users can be developers, testers, business people, analysts and many more. So before designing a system the architecture is made with different perspectives in mind. The most important part is to visualize the system from different viewer.s perspective. The better we understand the better we make the system.

UML plays an important role in defining different perspectives of a system. These perspectives are:

- Design View
- Implementation View
- Process View
- Deployment View
- Usecase View

And the centre is the **Use Case** view which connects all these four. A **Use case** represents the functionality of the system. So the other perspectives are connected with use case.

Design of a system consists of classes, interfaces and collaboration. UML provides class diagram, object diagram to support this.

Implementation defines the components assembled together to make a complete physical system. UML component diagram is used to support implementation perspective.

Process defines the flow of the system. So the same elements as used in *Design* are also used to support this perspective.

Deployment represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

Software Development Life Cycle:

The Unified Software Development Process

A software development process is the set of activities needed to transform a user,,s requirements into a software system.

Basic properties:

• use case driven

• architecture centric

• iterative and incremental

Use case Driven

Use cases

• capture requirements of the user,

• divide the development project into smaller subprojects,

• are constantly refined during the whole development process

• are used to verify the correctness of the implemented software

Architecture Centric:

• Find structures which are suitable to achive the function specified in the use cases,

• understandable,

• maintainable,

• reusable for later extensions or newly discovered use cases and describe them, so that they can be communicated between developers and users.

Inception establishes the business rationale for the project and decides on the scope of the project.

Elaboration is the phase where you collect more detailed requirements, do high-level analysis and design to establish a baseline architecture and create the plan for construction.

Construction is an iterative and incremental process. Each iteration in this phase builds production- quality software prototypes, tested and integrated as subset of the requirements of the project.

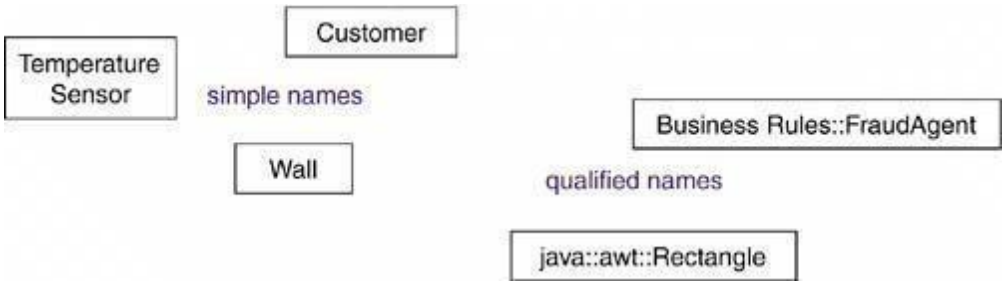
Transition contains beta testing, performance tuning and user training.

CLASSES

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.

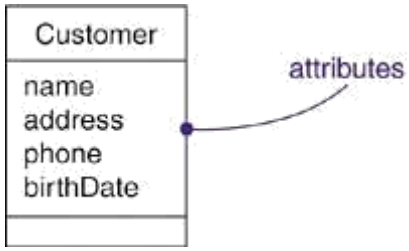
Names

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 qualified 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



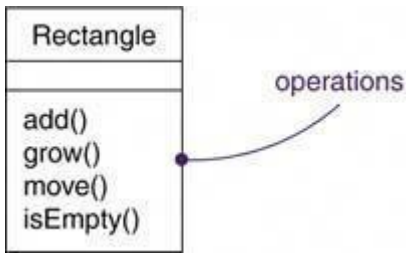
Attributes

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.

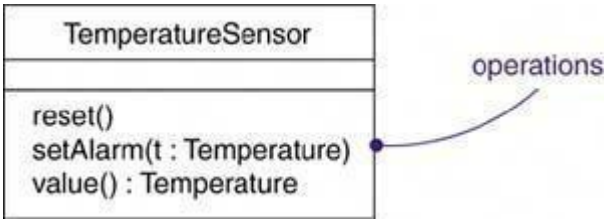


Operations

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 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



You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type.

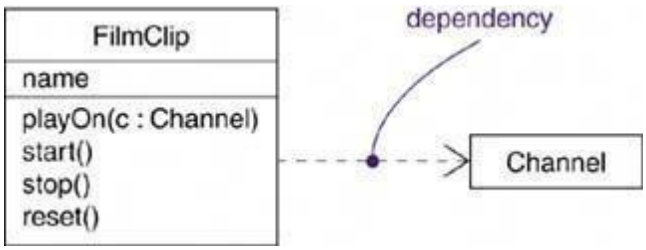


RELATIONSHIPS

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.

Dependencies

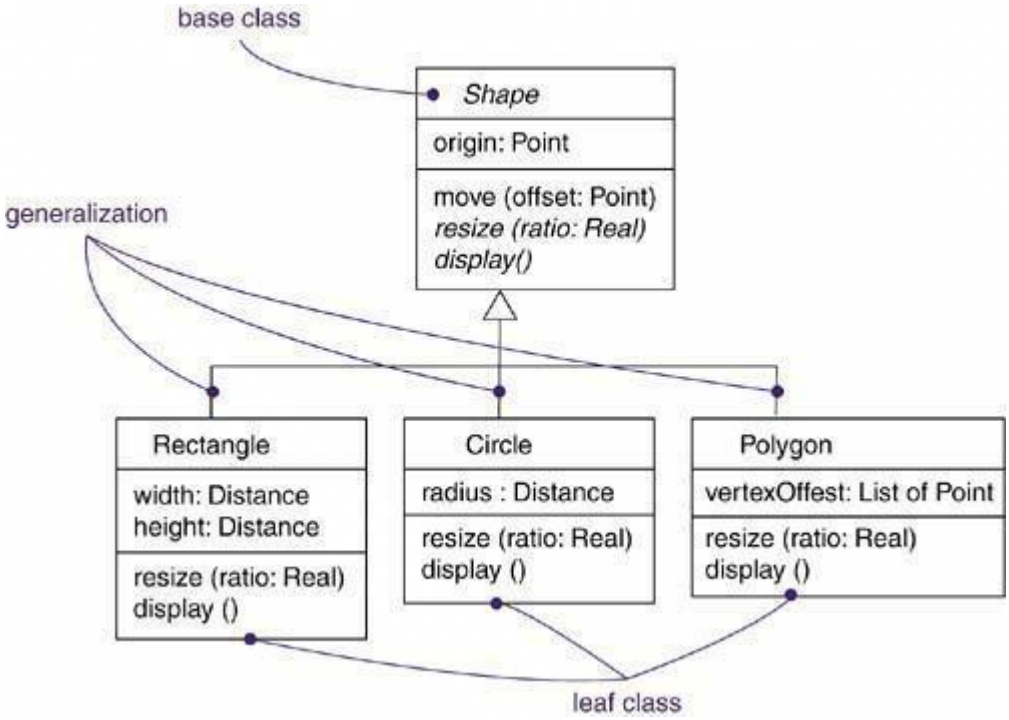
A *dependency* is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Choose dependencies when you want to show one thing using another.



Generalizations

A *generalization* is a relationship between a general kind of thing (called the superclass

or parent) and a more specific kind of 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). An objects of the child class may be used for a variable or parameter typed by the parent, but not the reverse



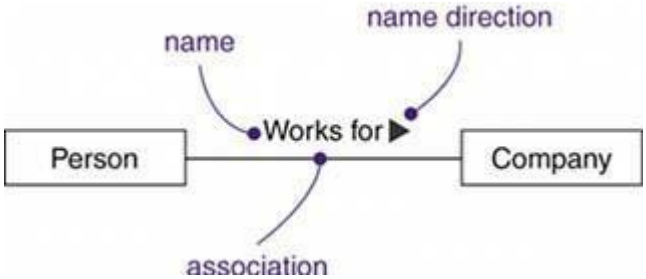
Associations

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 relate objects of one class to objects of the other class. 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*.

Beyond this basic form, there are four adornments that apply to associations.

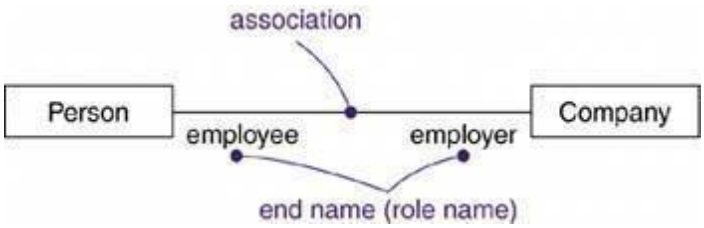
Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name.



Role

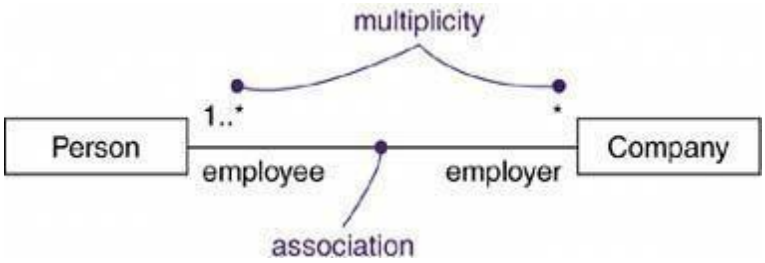
When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association. You can explicitly name the role a class plays in an association. The role played by an end of an association is called an end name (in UML1, it was called a role name). the class Person playing the role of employee is associated with the class Company playing the role of employer.



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. It represents a range of integers specifying the possible size of the set of related objects.

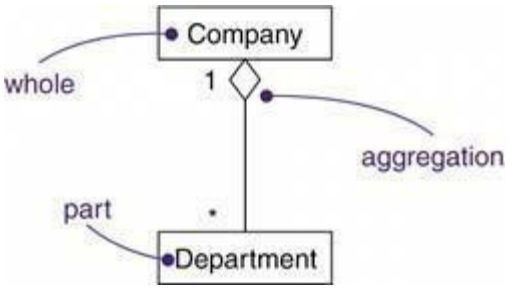
The number of objects must be in the given range. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can give an integer range (such as 2..5). You can even state an exact number (for example, 3, which is equivalent to 3..3).



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



COMMON MECHANISMS

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 (French quotation marks of the form « »), placed above the name of another element.

Optionally the stereotyped element may be rendered by using a new icon associated with that stereotype.

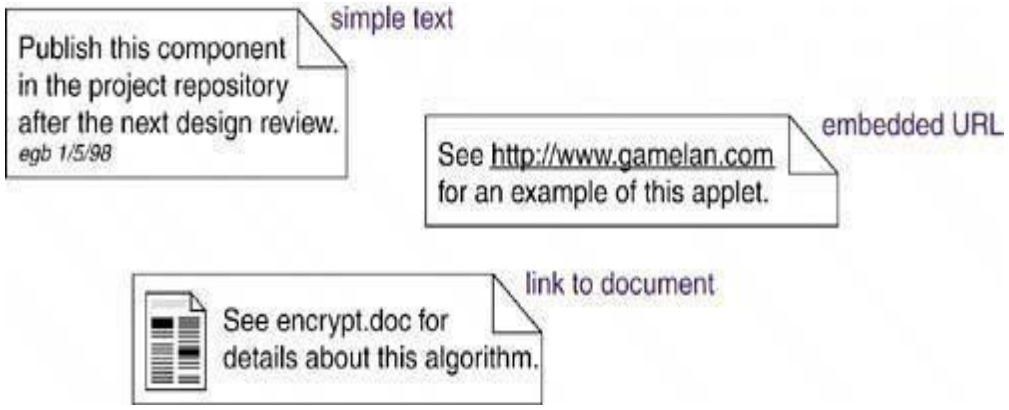
A *tagged value* is a property of a stereotype, allowing you to create new information in an element bearing that stereotype. Graphically, a tagged value is rendered as a string of the form name = value within a note attached to the object.

A *constraint* is a textual specification 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.

Notes

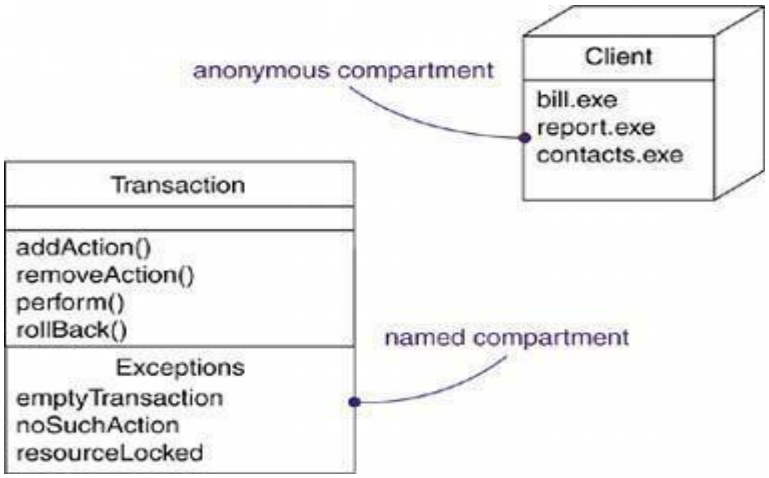
A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

A note may contain any combination of text or graphics



Other Adornments

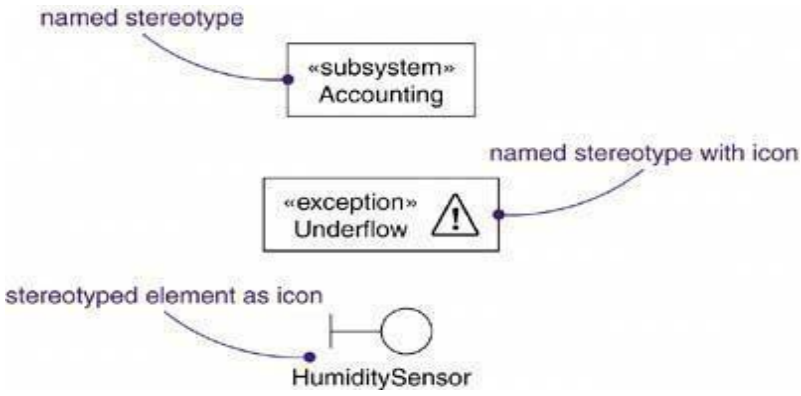
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification.

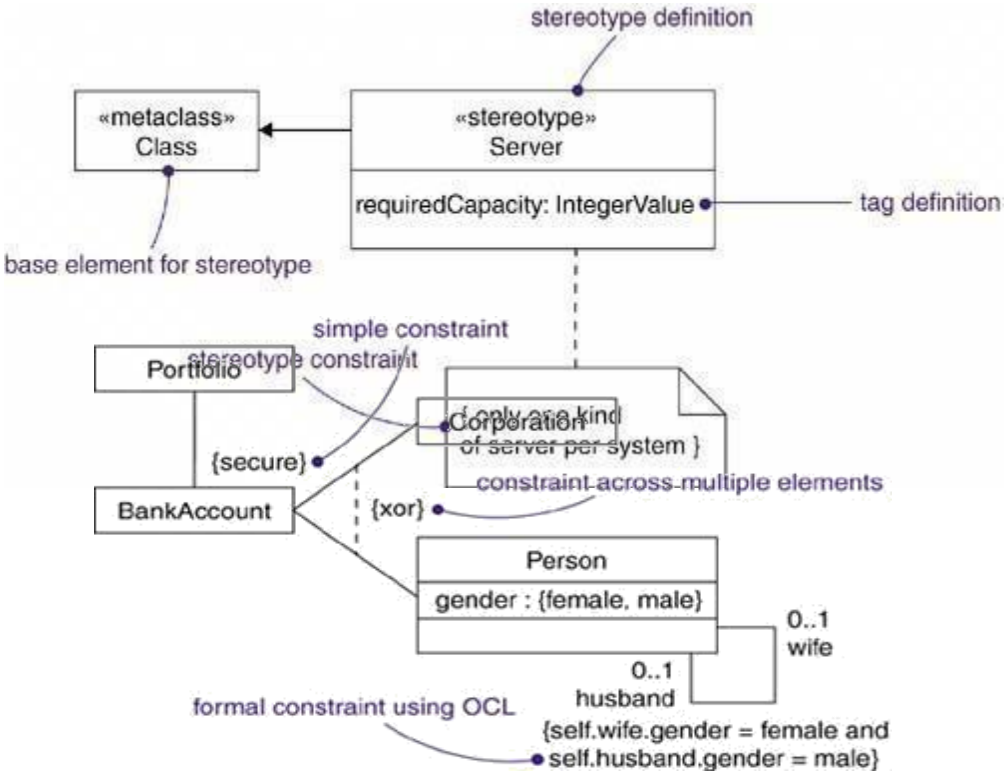


Stereotypes

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.

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, «name») and placed above the name of another element.





Tagged Values

Everything in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends, each with its own properties; and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties to a stereotype.

Constraints

Everything in the UML has its own semantics. Generalization (usually, if you know what's good for you) implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or extend existing rules. A constraint specifies conditions that a run-time configuration must satisfy to conform to the model.

• Stereotype specifies that the classifier is a stereotype that may be applied to other elements

Contents:

1. **Classes**

2. **Relationships** **UNIT-II**

I. BASIC STRUCTURAL MODELING

3. **Common Mechanisms**
4. **Diagrams**

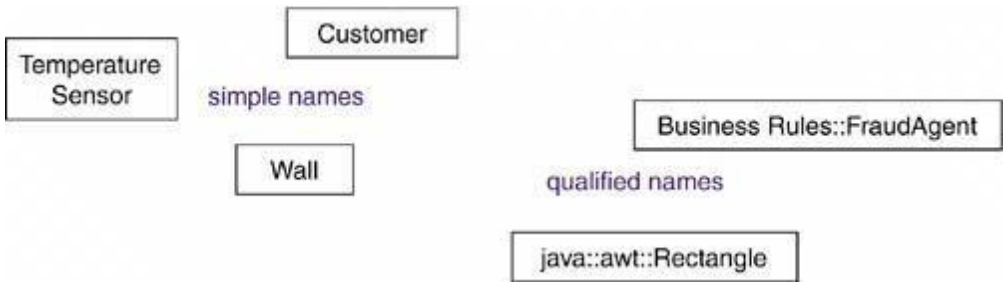
1. Classes:

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.

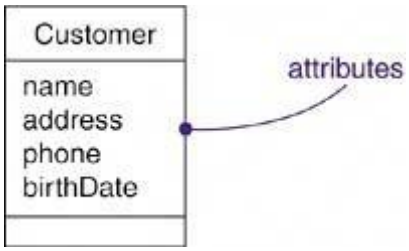
Names

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 qualified 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



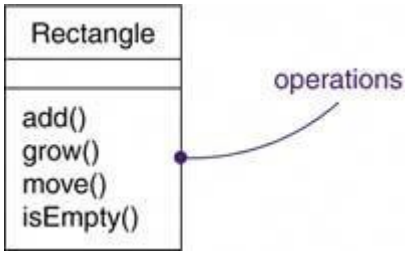
Attributes

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

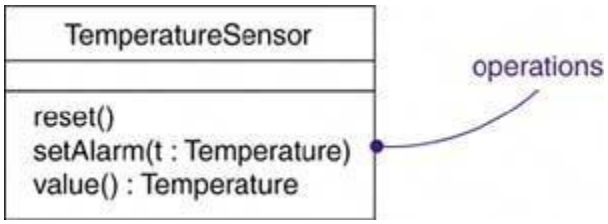


Operations

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 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



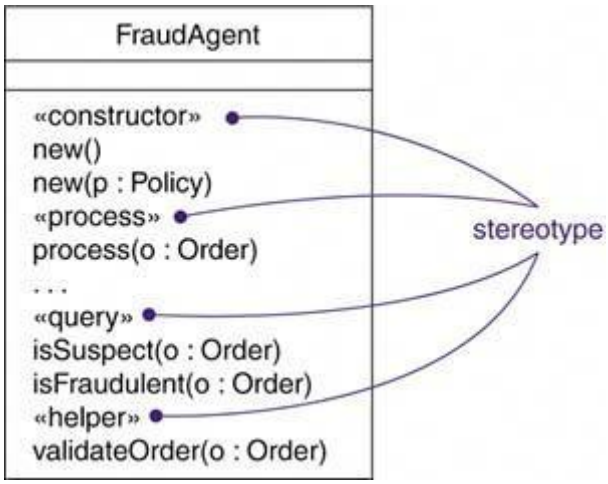
You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type



Organizing Attributes and Operations

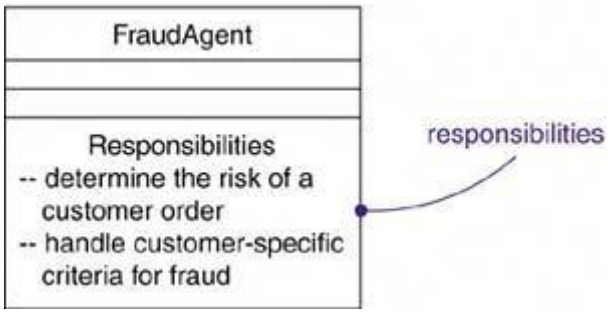
When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For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



Common Modeling Techniques

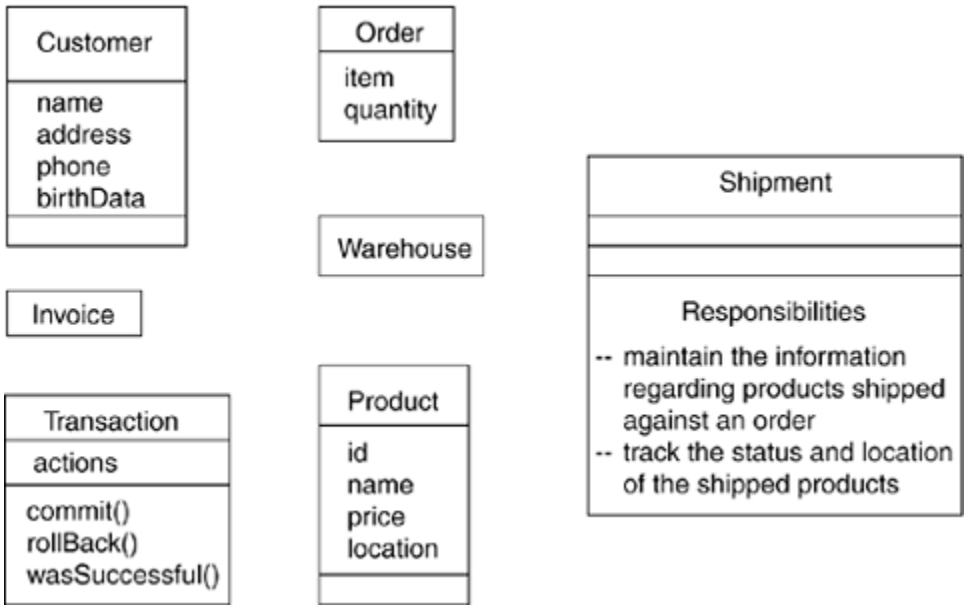
Modeling the Vocabulary of a System

You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Each of these abstractions is a part of the vocabulary of your system, meaning that, together, they represent the things that are important to users and to implementers.

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

A set of classes drawn from a retail system, including Customer, Order, and Product. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, TRansaction, which applies to orders and shipments.



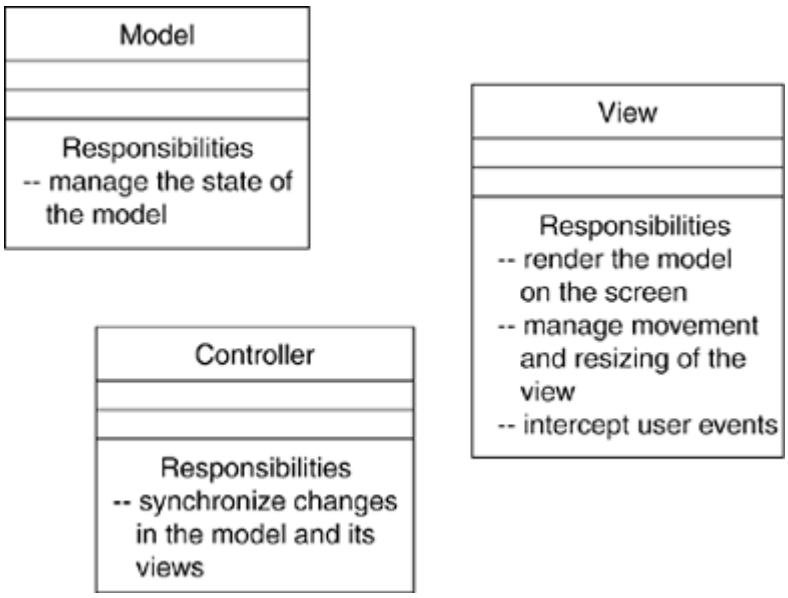
Modeling the Distribution of Responsibilities in a System

Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.

To model the distribution of responsibilities in a system,

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

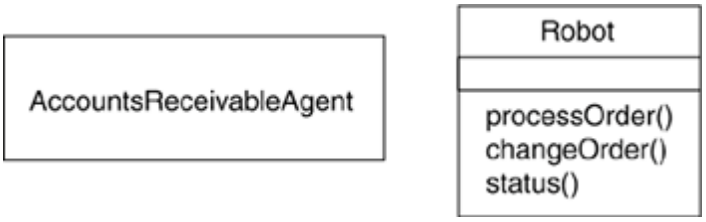
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.



Modeling Nonsoftware Things

To model nonsoftware things,

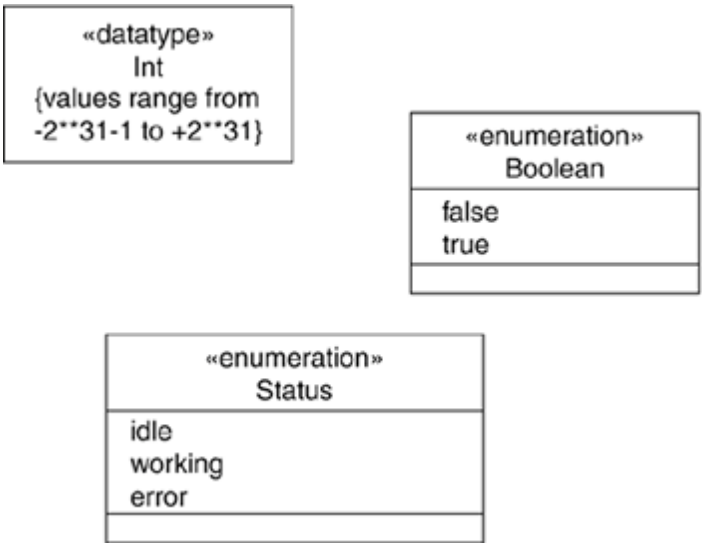
- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware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



Modeling Primitive Types

To model primitive types,

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



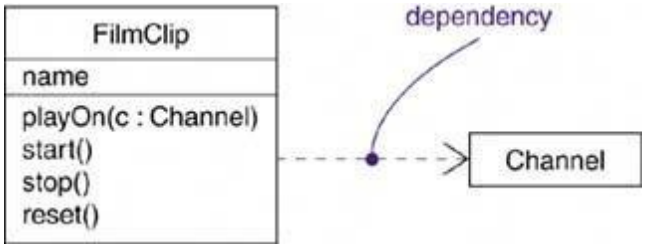
2. Relationships:

Terms and Concepts

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.

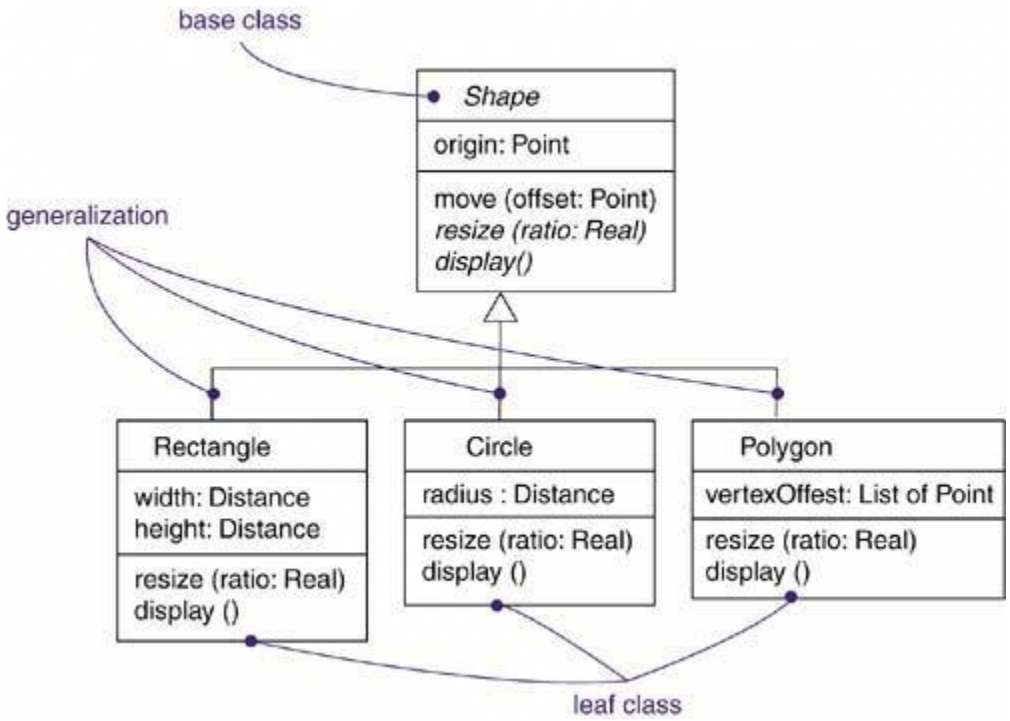
Dependencies

A *dependency* is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Choose dependencies when you want to show one thing using another.



Generalizations

A *generalization* is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of 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). An objects of the child class may be used for a variable or parameter typed by the parent, but not the reverse



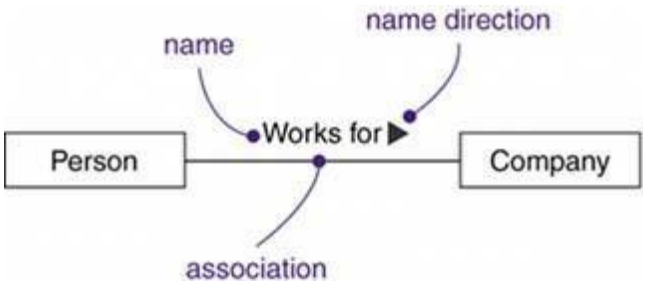
Associations

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 relate objects of one class to objects of the other class. 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*.

Beyond this basic form, there are four adornments that apply to associations.

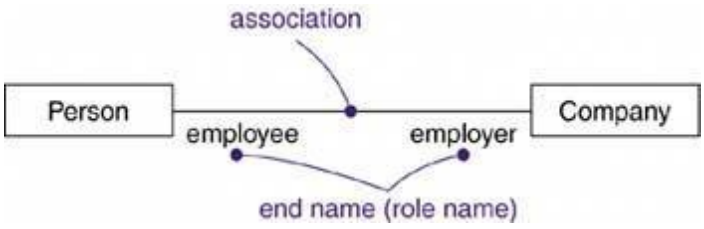
Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name.



Role

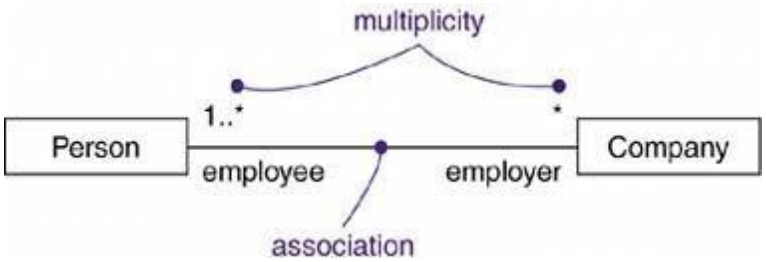
When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association. You can explicitly name the role a class plays in an association. The role played by an end of an association is called an end name (in UML1, it was called a role name). the class Person playing the role of employee is associated with the class Company playing the role of employer.



Multiplicity

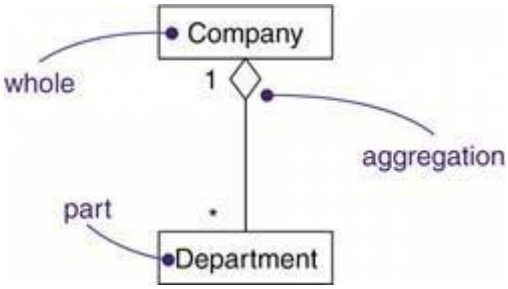
An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role. It represents a range of integers specifying the possible size of the set of related objects.

The number of objects must be in the given range. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can give an integer range (such as 2..5). You can even state an exact number (for example, 3, which is equivalent to 3..3).



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



Common Modeling Techniques

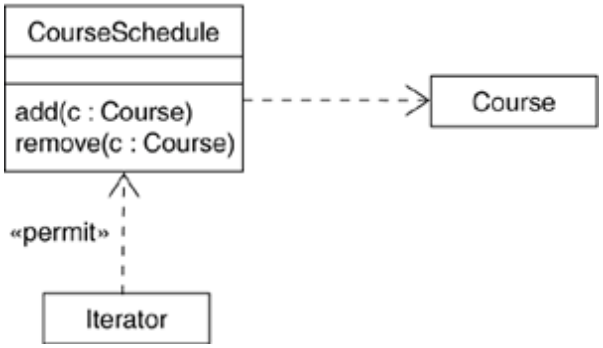
Modeling Simple Dependencies

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

To model this using relationship,

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

a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.

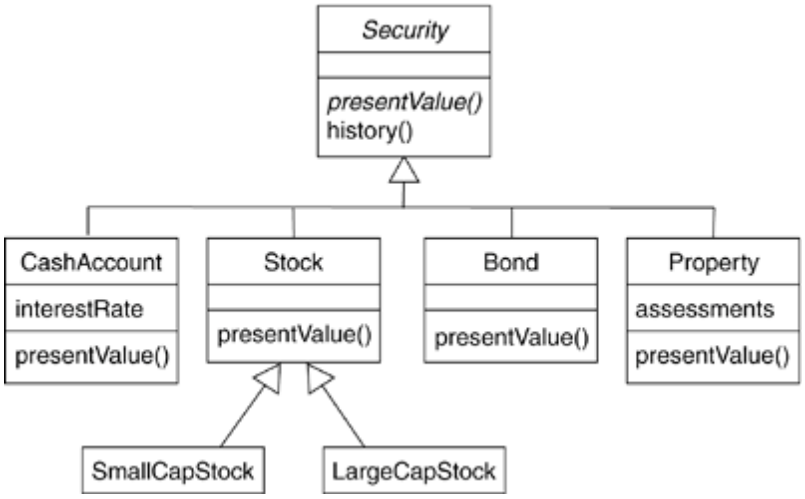


Modeling Single Inheritance

In modeling the vocabulary of your system, you will often run across classes that are structurally or behaviorally similar to others. You could model each of these as distinct and unrelated abstractions. A better way would be to extract any common structural and behavioral features and place them in more-general classes from which the specialized ones inherit.

To model inheritance relationships,

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



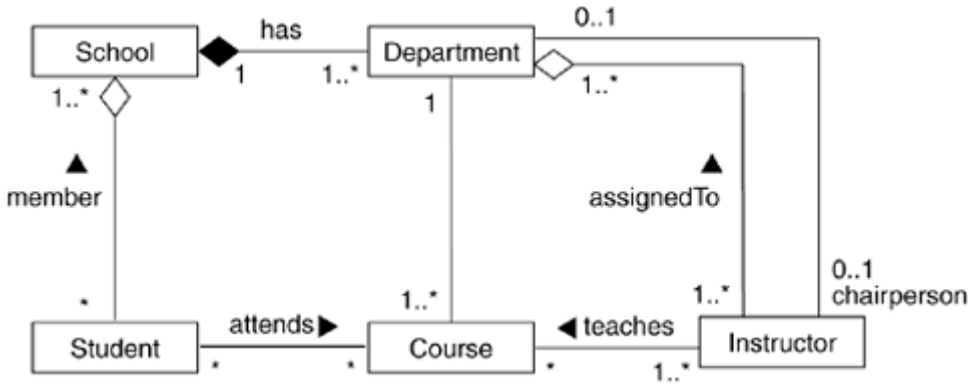
Modeling Structural Relationships

When you model with dependencies or generalization relationships, you may be modeling classes that represent different levels of importance or different levels of abstraction. Given a dependency between two classes, one class depends on another but the other class has no knowledge of the one.

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as local variables in a procedure or parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.

- For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if they help to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole with a diamond.



3. Common Mechanisms:

Terms and Concepts

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 (French quotation marks of the form « »), placed above the name of another element.

Optionally the stereotyped element may be rendered by using a new icon associated with that stereotype.

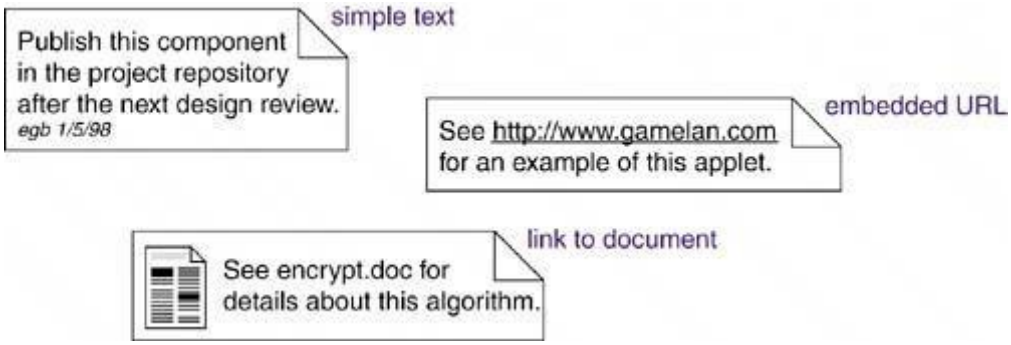
A *tagged value* is a property of a stereotype, allowing you to create new information in an element bearing that stereotype. Graphically, a tagged value is rendered as a string of the form name = value within a note attached to the object.

A *constraint* is a textual specification 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.

Notes

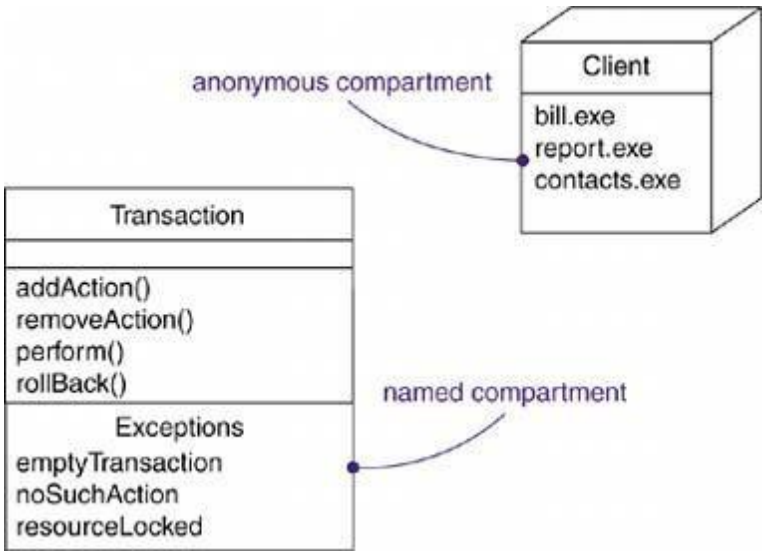
A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

A note may contain any combination of text or graphics



Other Adornments

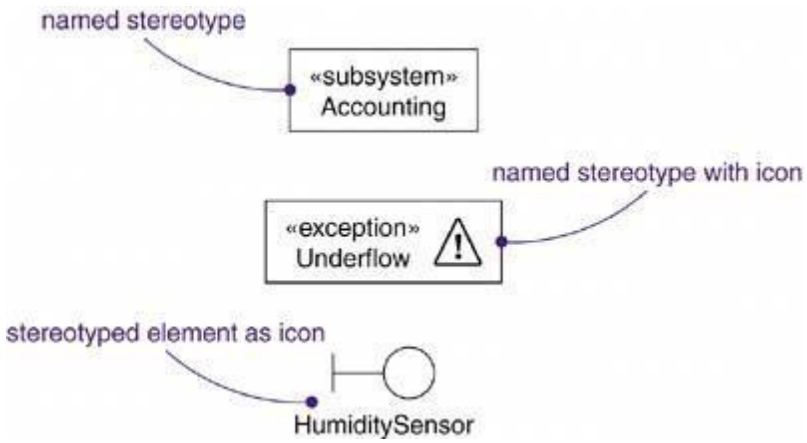
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification



Stereotypes

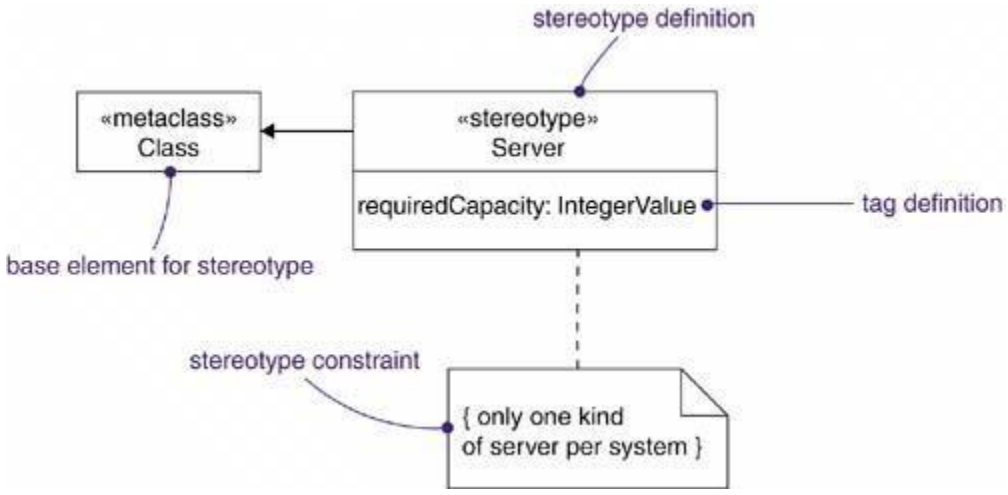
The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model.

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, «name») and placed above the name of another element.



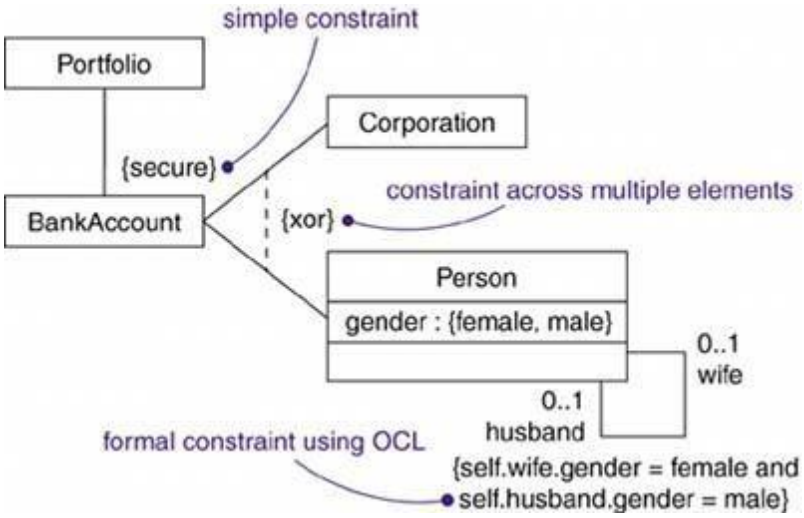
Tagged Values

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 to a stereotype.



Constraints

Everything in the UML has its own semantics. Generalization (usually, if you know what's good for you) implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or extend existing rules. A constraint specifies conditions that a run-time configuration must satisfy to conform to the model.



• stereotype Specifies that the classifier is a stereotype that may be applied to other elements

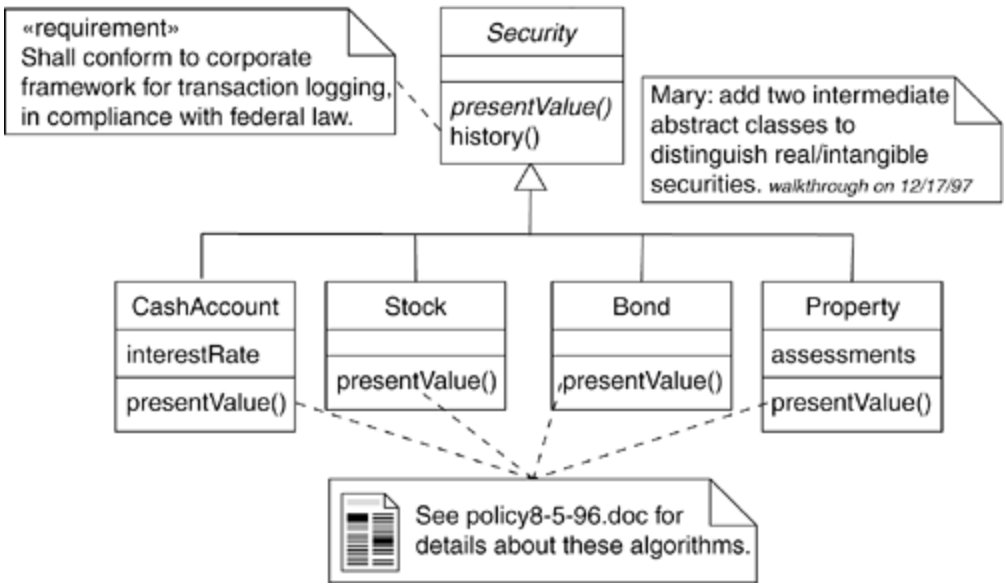
Common Modeling Techniques

Modeling Comments

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

To model a comment,

- Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, andunless they are of historic interestdiscard the others.

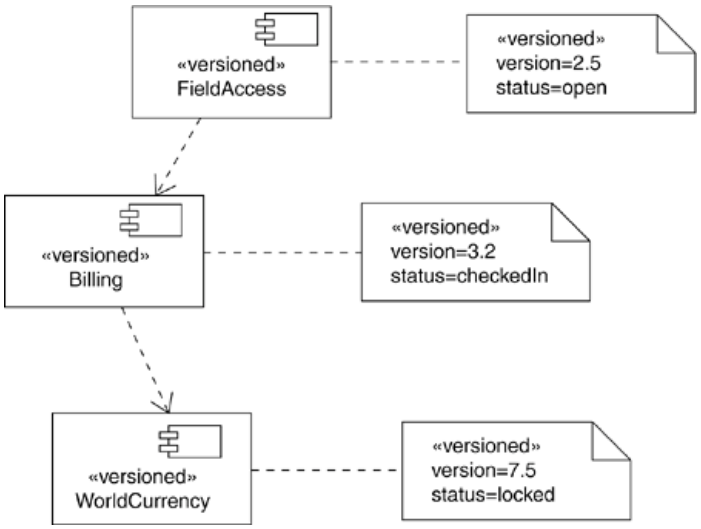


Modeling New Properties

The basic properties of the UML's building blocksattributes and operations for classes, the contents of packages

To model new properties,

- First, make sure there's not already a way to express what you want by using basic UML.
- If you re convinced there's no other way to express these semantics, define a stereotype and add the new properties to the stereotype. The rules of generalization applytagged values defined for one kind of stereotype apply to its children.

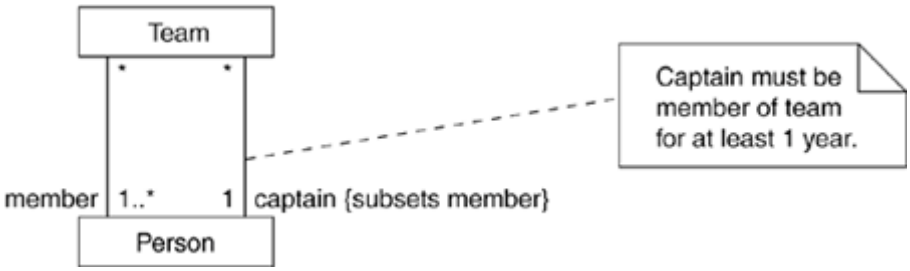


Modeling New Semantics

When you create a model using the UML, you work within the rules the UML lays down. However, if you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML.
- If you re convinced there's no other way to express these semantics, write your new semantics in a constraint placed near the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



4. Diagrams:

Terms and Concepts

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

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

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

A *view* is a projection into the organization and structure of a system's model, focused on one aspect of that system.

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

In modeling real systems, no matter what the problem domain, you'll find yourself creating the same kinds of diagrams, because they represent common views into common models. Typically, you'll view the static parts of a system using one of the following diagrams.

1. Class diagram
2. Component diagram
3. Composite structure diagram
4. Object diagram
5. Deployment diagram
6. Artifact diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

1. Use case diagram
2. Sequence diagram
3. Communication diagram
4. State diagram
5. Activity diagram

Structural Diagrams

The UML's structural diagrams exist to visualize, specify, construct, and document the static aspects of a system. You can think of the static aspects of a system as representing its relatively stable skeleton and scaffolding. Just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents, so too do the static aspects of a software system encompass the existence and placement of such things as classes, interfaces, collaborations, components, and nodes.

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

1.Class diagram	Classes, interfaces, and collaborations
2.Component diagram	Components
3.Object diagram	Objects
4.Deployment diagram	Nodes

Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system. You can think of the dynamic aspects of a system as representing its changing parts. Just as the dynamic aspects of a house encompass airflow and traffic through the rooms of a house, so too do the dynamic aspects of a software system encompass such things as the flow of messages over time and the physical movement of components across a network.

The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1.Use case diagram	Organizes the behaviors of the system
2.Sequence diagram	Focuses on the time ordering of messages
3.Collaboration diagram	Focuses on the structural organization of objects that send and receive messages
4.State diagram	Focuses on the changing state of a system driven by events
5.Activity diagram	Focuses on the flow of control from activity to activity

Common Modeling Techniques

Modeling Different Views of a System

When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions.

To model a system from different views,

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project. The five views of an architecture described earlier are a good starting point.
- For each of these views, decide which artifacts you need to create to capture the essential details of that view. For the most part, these artifacts will consist of various UML diagrams.
- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.
- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams.

Use case view	Use case diagrams
Design view	Class diagrams (for structural modeling)
Interaction view	Interaction diagrams (for behavioral modeling)
Implementation view	Composite structure diagrams
Deployment view	None required

Similarly, if yours is a client/server system, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.

Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

▪Use case view	Use case diagrams
	Sequence diagrams
▪Design view	Class diagrams (for structural modeling)
	Interaction diagrams (for behavioral modeling)
	State diagrams (for behavioral modeling)
	Activity diagrams
▪Interaction view	Interaction diagrams (for behavioral modeling)
▪Implementation view	Class diagrams
	Composite structure diagrams
▪Deployment view	Deployment diagrams

Modeling Different Levels of Abstraction

Not only do you need to view a system from several angles, you'll also find people involved in development who need the same view of the system but at different levels of abstraction

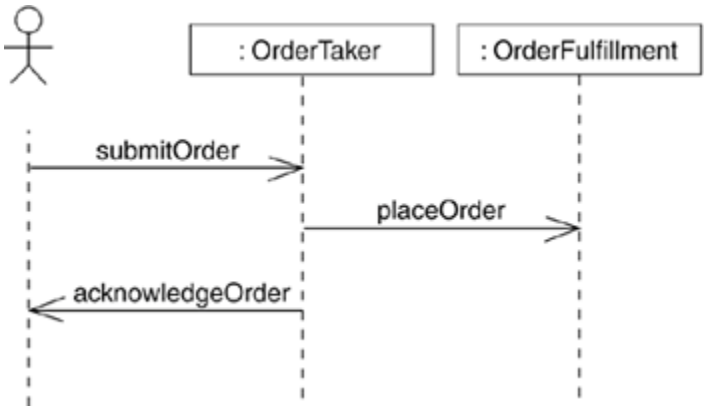
To model a system at different levels of abstraction by presenting diagrams with different levels of detail,

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to reveal a lot of detail. If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.
- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:
    1. Building blocks and relationships: Hide those that are not relevant to the intent of your diagram or the needs of your reader.
    2. Adornments: Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

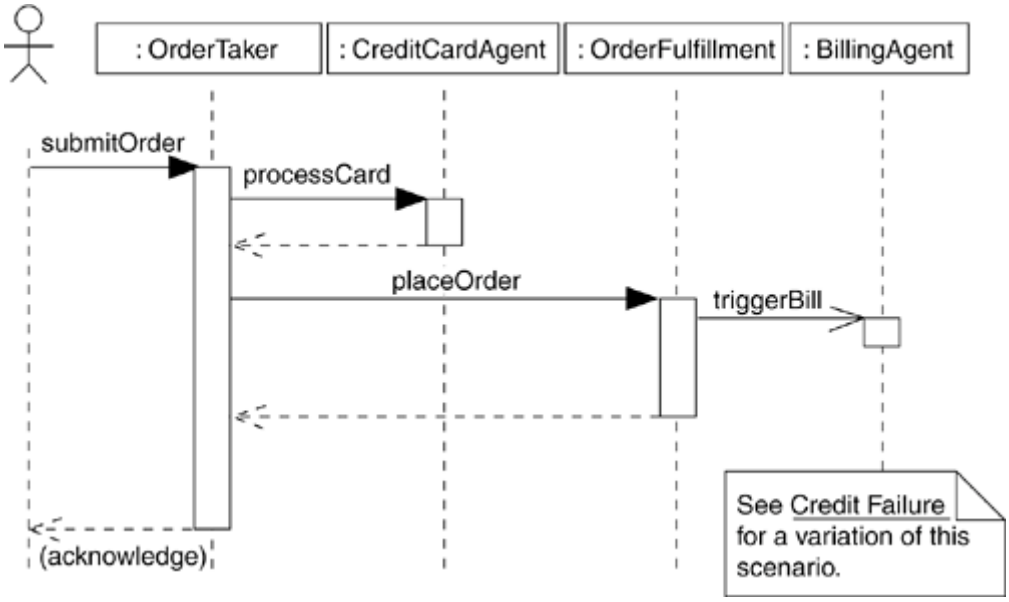
3. Flow: In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.
4. Stereotypes: In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

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

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:
    1. Use cases and their realization: Use cases in a use case model will trace to collaborations in a design model.
    2. Collaborations and their realization: Collaborations will trace to a society of classes that work together to carry out the collaboration.
    3. Components and their design: Components in an implementation model will trace to the elements in a design model.
    4. Nodes and their components: Nodes in a deployment model will trace to components in an implementation model.



Higher Level of Abstraction



Lower level of Abstraction

Modeling Complex Views

To model complex views,

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

1. ADVANCED STRUCTURAL MODELING

Terms and Concepts

A *class diagram* is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagramsa name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

Contents

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Dependency, generalization, and association relationships

Common Uses

You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a systemthe services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you re modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

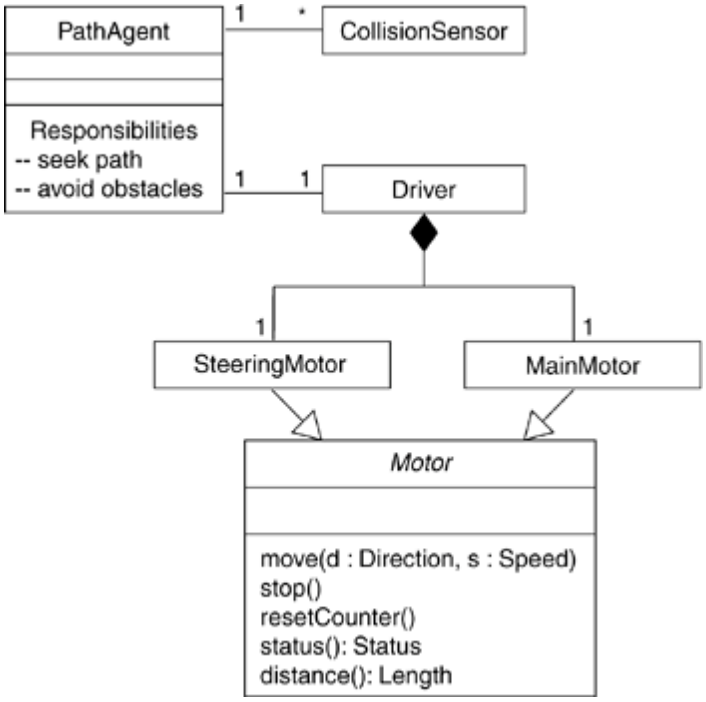
Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

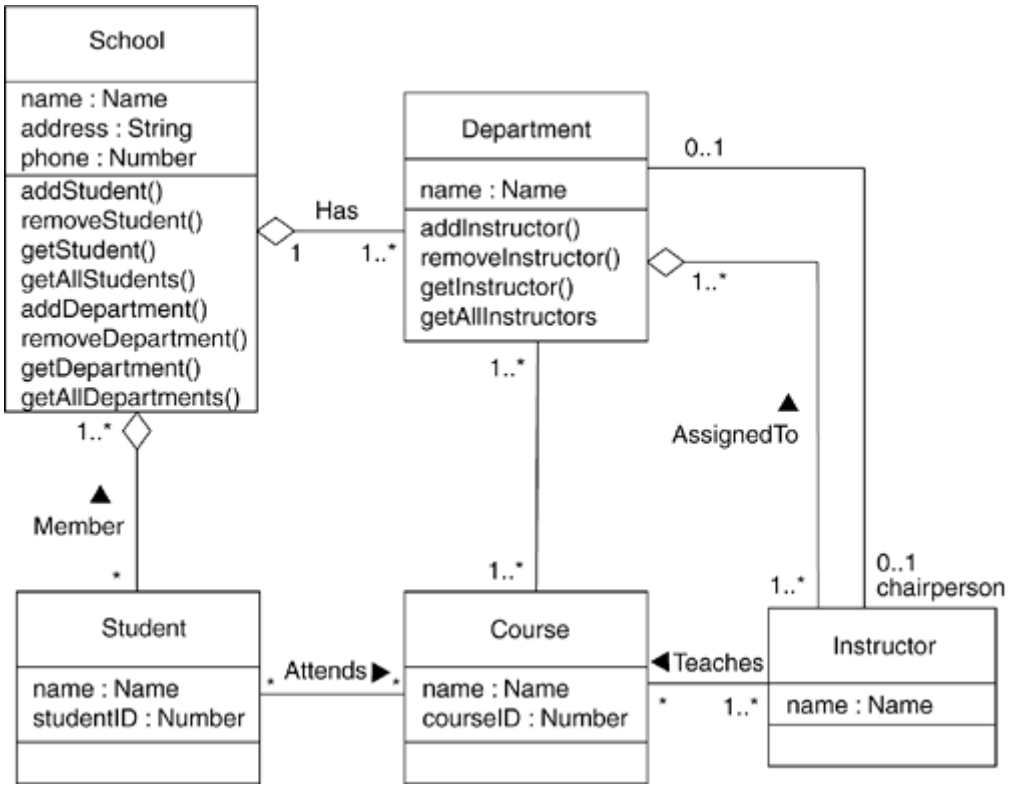


Modeling a Logical Database Schema

To model a schema,

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes. You can define your own set of stereotypes and tagged values to address database-specific details.

- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their multiplicities that relate these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

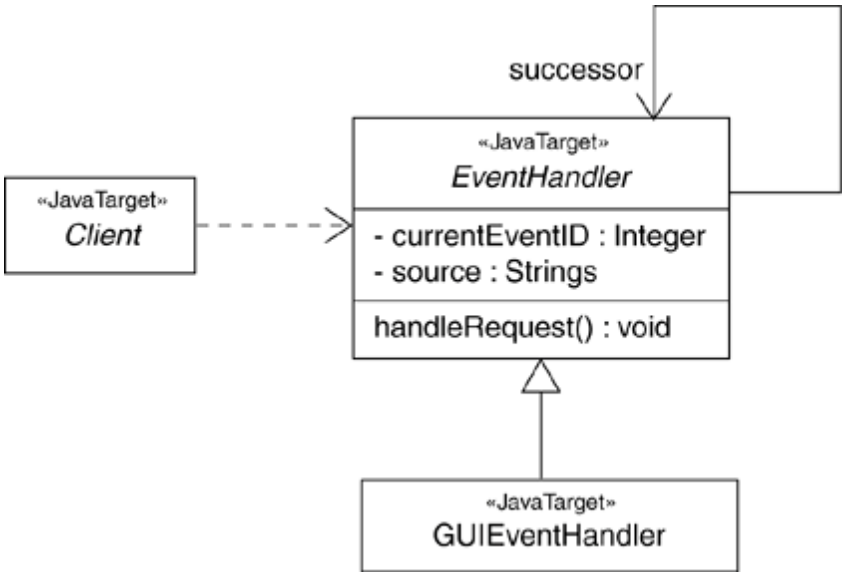


Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. In fact, this is a major reason why you need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may want to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent), or you can develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to guide implementation choices in your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to generate code.



All of these classes specify a mapping to Java, as noted in their stereotype. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class EventHandler yields the following code.

```
public abstract class EventHandler {

  EventHandler successor;
  private Integer currentEventID;
  private String source;

  EventHandler() {}
  public void handleRequest() {}

}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. You need to select portion of the code and build the model from the bottom.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.
- Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

II. ADVANCED STRUCTURAL MODELING

1. **Advanced Classes**
2. **Advanced Relationships**
3. **Interface, Type and Role**
4. **Packages**

1. Advanced Classes:

Terms and Concepts

A *classifier* is a mechanism that describes structural and behavioral features. Classifiers include classes, associations, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.

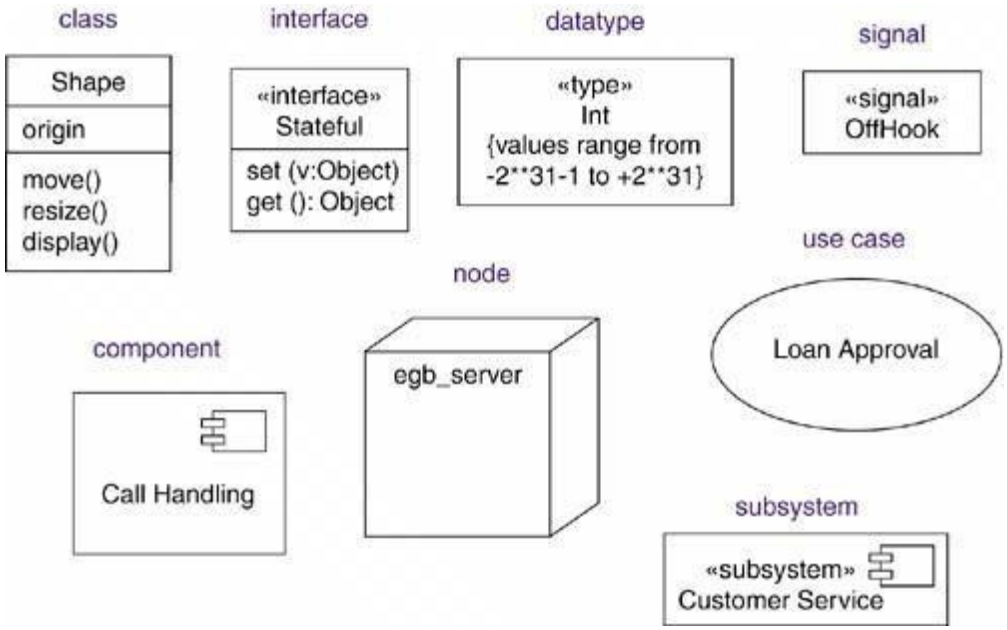
Classifiers

When you model, you'll discover abstractions that represent things in the real world and things in your solution. For example, if you are building a Web-based ordering system, the vocabulary of your project will likely include a Customer class (representing people who order products) and a

TRansaction class (an implementation artifact, representing an atomic action). In the deployed system, you might have a Pricing component, with instances living on every client node. Each of these abstractions will have instances; separating the essence and the instance of the things in your world is an important part of modeling.

The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however. The UML provides a number of other kinds of classifiers to help you model.

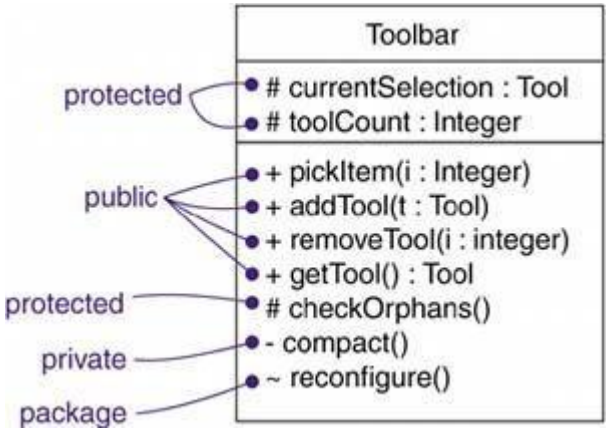
- **Interface** — A collection of operations that are used to specify a service of a class or a component
- **Datatype** — A type whose values are immutable, including primitive built-in types (such as numbers and strings) as well as enumeration types (such as Boolean)
- **Association** — A description of a set of links, each of which relates two or more objects.
- **Signal** — The specification of an asynchronous message communicated between instances
- **Component** — A modular part of a system that hides its implementation behind a set of external interfaces
- **Node** — A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
- **Use case** — A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
- **Subsystem** — A component that represents a major part of a system



Visibility

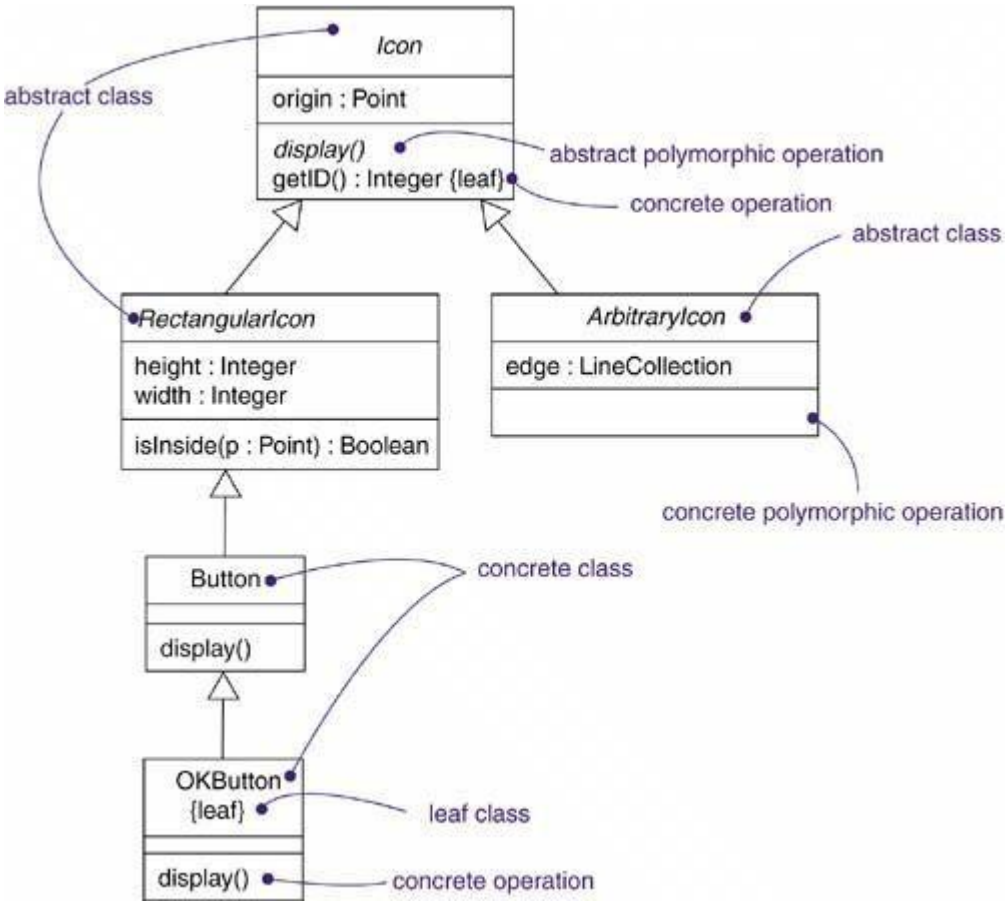
One of the design details you can specify for an attribute or operation is visibility. The visibility of a feature specifies whether it can be used by other classifiers. In the UML, you can specify any of four levels of visibility.

1. public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +.
2. protected	Any descendant of the classifier can use the feature; specified by prepending the symbol #.
3. private	Only the classifier itself can use the feature; specified by prepending the symbol -.
3. package	Only classifiers declared in the same package can use the feature; specified by prepending the symbol ~.



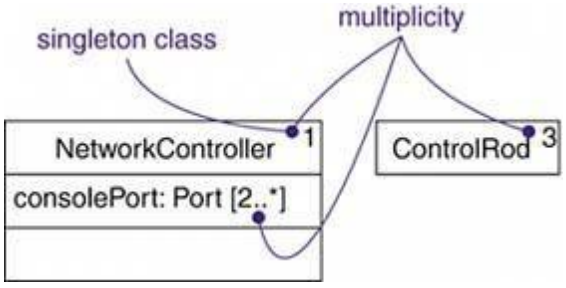
Abstract, Leaf, and Polymorphic Elements

You use generalization relationships to model a lattice of classes, with more-generalized abstractions at the top of the hierarchy and more-specific ones at the bottom. Within these hierarchies, it's common to specify that certain classes are abstractmeaning that they may not have any direct instances. In the UML, you specify that a class is abstract by writing its name in italics. Icon, RectangularIcon, and ArbitraryIcon are all abstract classes. By contrast, a concrete class (such as Button and OKButton) may have direct instances.



Multiplicity

Whenever you use a class, it's reasonable to assume that there may be any number of instances of that class (unless, of course, it is an abstract class and may not have any direct instances, although there may be any number of instances of its concrete children).



Attributes

At the most abstract level, when you model a class's structural features (that is, its attributes), you simply write each attribute's name.

visibility] name

[':' type] ['[' multiplicity] ']']
['=' initial-value]
[property-string {',' property-string}]

For example, the following are all legal attribute declarations:

origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
name : String[0..1]	Name, type, and multiplicity
origin : Point = (0,0)	Name, type, and initial value
id: Integer {readonly}	Name and property

Operations

At the most abstract level, when you model a class's behavioral features. ou can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

[visibility] name ['(' parameter-list ')']
[':' return-type]
[property-string {',' property-string}]

For example, the following are all legal operation declarations:

display	Name only
+ display	Visibility and name
set(n : Name, s : String)	Name and parameters
getID() : Integer	Name and return type
restart() {guarded}	Name and property

In an operation's signature, you may provide zero or more parameters, each of which follows the syntax

[direction] name : type [= default-value]

Direction may be any of the following values:

- in An input parameter; may not be modified
- out An output parameter; may be modified to communicate information to the caller
- inout An input parameter; may be modified to communicate information to the caller

In addition to the leaf and abstract properties described earlier, there are defined properties that you can use with operations.

1. query	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
3. guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
4. concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics; concurrent operations must be designed so that they perform correctly in case of a concurrent sequential or guarded operation on the same object.
5. static	The operation does not have an implicit parameter for the target object; it behaves like a traditional global procedure.

Template Classes

A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes.

A template may include slots for classes, objects, and values, and these slots serve as the template's parameters. You can't use a template directly; you have to instantiate it first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.

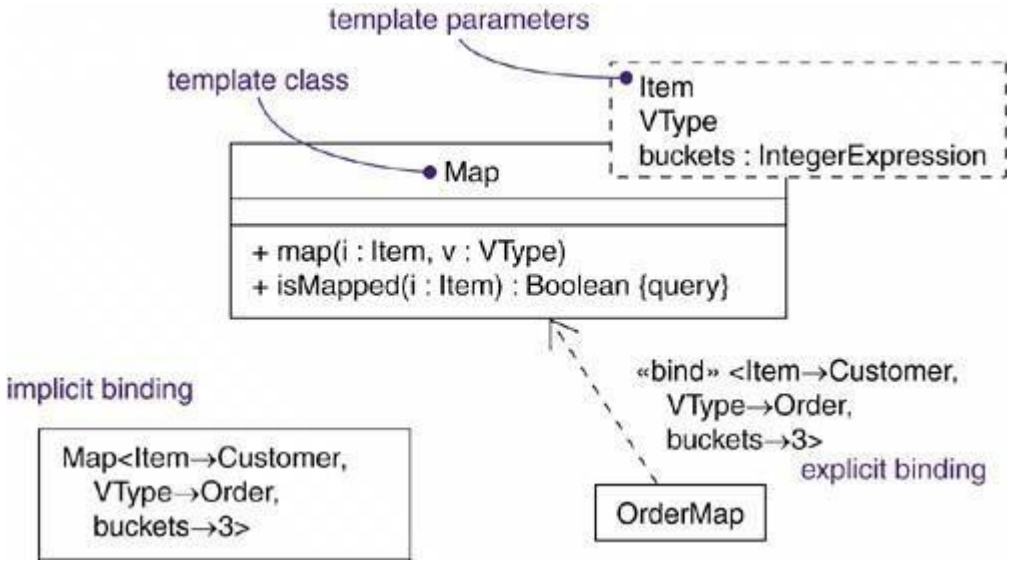
The most common use of template classes is to specify containers that can be instantiated for specific elements, making them type-safe. For example, the following C++ code fragment declares a parameterized Map class.

```
template<class Item, class VType, int Buckets>
```

```
class Map {
public:
 virtual map(const Item&, const VType&);
 virtual Boolean isMappen(const Item&) const;
 ...
};
```

You might then instantiate this template to map Customer objects to Order objects.

m : Map<Customer, Order, 3>;



Standard Elements

All of the UML's extensibility mechanisms apply to classes.

The UML defines four standard stereotypes that apply to classes.

1. metaclass	Specifies a classifier whose objects are all classes
2. powertype	Specifies a classifier whose objects are classes that are the children of a given parent class
3.stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
4. utility	Specifies a class whose attributes and operations are all static scoped

Common Modeling Techniques

Modeling the Semantics of a Class

To model the semantics of a class, choose among the following possibilities, arranged from informal to formal.

- Specify the responsibilities of the class. A responsibility is a contract or obligation of a type or class and is rendered in a note attached to the class, or in an extra compartment in the class icon.
- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as semantics) attached to the class.
- Specify the body of each method using structured text or a programming language, rendered in a note attached to the operation by a dependency relationship.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as precondition, postcondition, and invariant) attached to the operation or class by a dependency relationship.
- Specify a state machine for the class. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- Specify internal structure of the class.
- Specify a collaboration that represents the class. A collaboration 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. A collaboration has a structural part as well as a dynamic part, so you can use collaborations to specify all dimensions of a class's semantics.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using a formal language such as OCL.

Pragmatically, you'll end up doing some combination of these approaches for the different abstractions in your system.

2. Advanced Relationships Terms and Concepts

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

Dependencies

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

A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships. There are a number of stereotypes, which can be organized into several groups.

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

1. bind	Specifies that the source instantiates the target template using the given actual parameters
2.derive	Specifies that the source may be computed from the target
3.permit	Specifies that the source is given special visibility into the target
4.instanceOf	Specifies that the source object is an instance of the target classifier. Ordinarily shown using text notation in the form source : Target
5.instantiate	Specifies that the source creates instances of the target
6.powertype	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are the children of a given parent
7. refine	Specifies that the source is at a finer degree of
8. use	Specifies that the semantics of the source element depends on the semantics of the public part of the target

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

1.import	Specifies that the public contents of the target package enter the public namespace of the source, as if they had been declared in the source.
2.access	Specifies that the public contents of the target package enter the private namespace of the source. The unqualified names may be used within the source, but they may not be re-exported.

Two stereotypes apply to dependency relationships among use cases:

1. extend	Specifies that the target use case extends the behavior of the source
2.include	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

One stereotype you'll encounter in the context of interactions among objects is

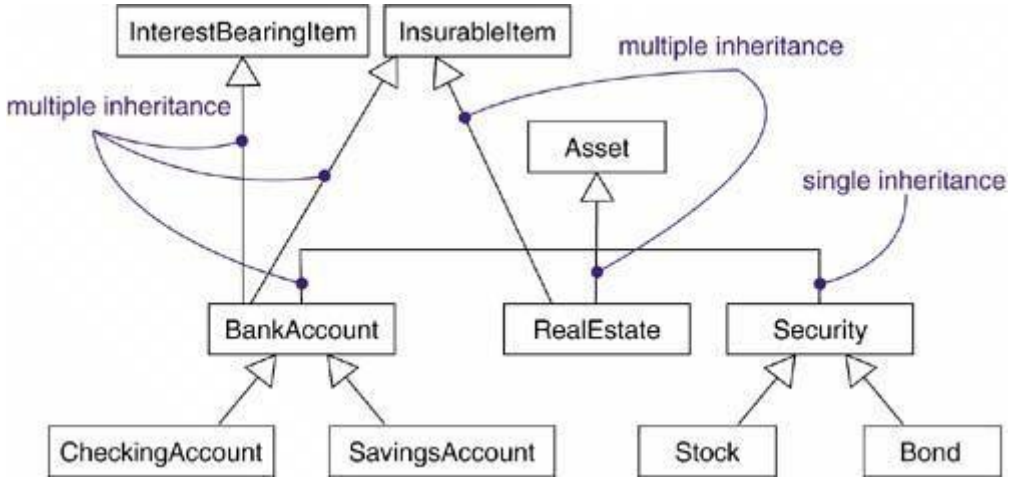
1. send	Specifies that the source class sends the target event

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

1.TRace Specifies that the target is a historical predecessor of the source from an earlier stage of development

Generalizations

A *generalization* is a relationship between a general classifier (called the superclass or parent) and a more specific classifier (called the subclass or child). For example, you might encounter the general class Window with its more specific subclass, MultiPaneWindow. With a generalization relationship from the child to the parent, the child (MultiPaneWindow) will inherit all the structure and behavior of the parent (Window).



A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines four constraints that may be applied to generalization relationships:

1. complete Specifies that all children in the generalization have been specified in the model (although some may be elided in the diagram) and that no additional children are permitted

2.incomplete Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted

3. disjoint Specifies that objects of the parent may have no more than one of the children as a type. For example, class Person can be specialized into disjoint classes Man and Woman.

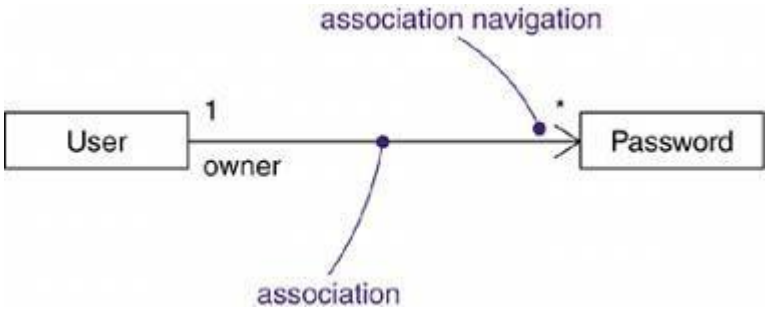
4.overlapping Specifies that objects of the parent may have more than one of the children as a type. For example, class Vehicle can be specialized into overlapping subclasses LandVehicle and WaterVehicle (an amphibious vehicle is both).

Associations

An *association* is a structural relationship, specifying that objects of one thing are connected to objects of another. For example, a Library class might have a one-to-many association to a Book class, indicating that each Book instance is owned by one Library instance.

Navigation

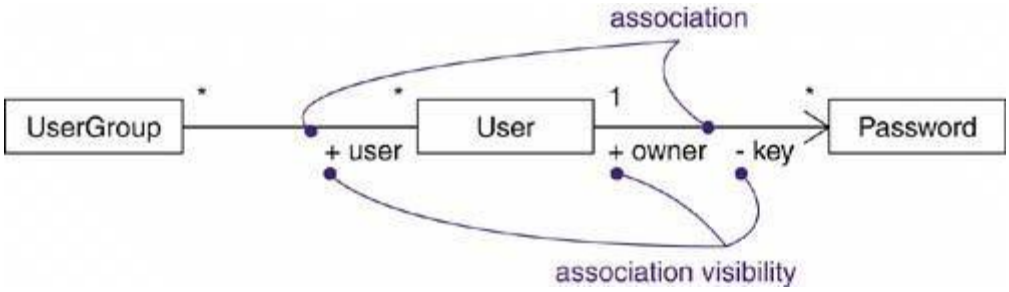
Given a plain, unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.



Visibility

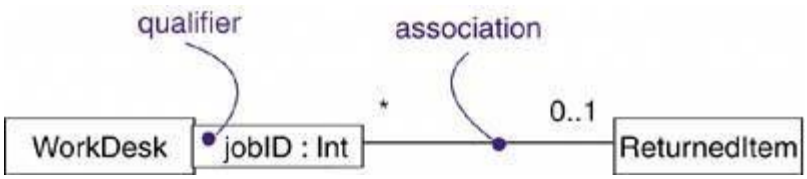
Given an association between two classes, objects of one class can see and navigate to objects of the other unless otherwise restricted by an explicit statement of navigation. However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.

There is an association between UserGroup and User and another between User and Password. Given a User object, it's possible to identify its corresponding Password objects. However, a Password is private to a User, so it shouldn't be accessible from the outside (unless, of course, the User explicitly exposes access to the Password,



Qualification

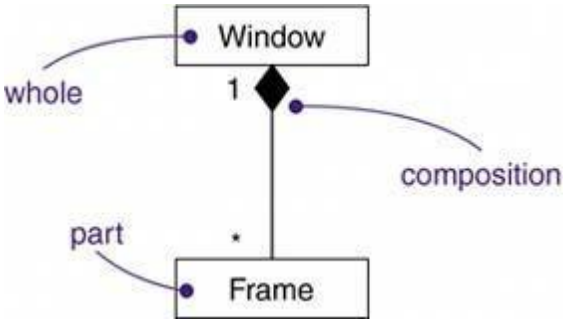
In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end? For example, consider the problem of modeling a work desk at a manufacturing site at which returned items are processed to be fixed



Composition

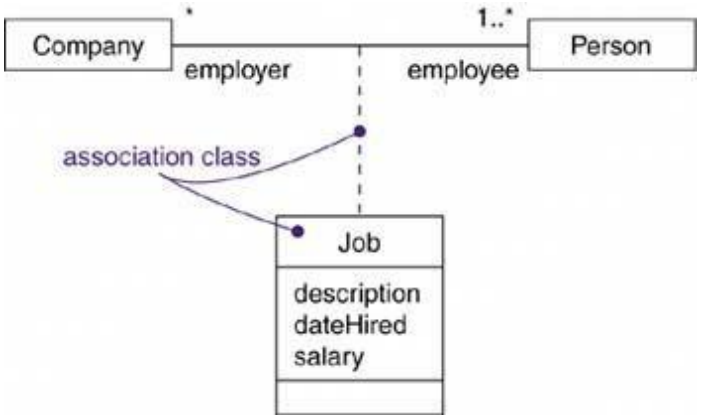
Aggregation turns out to be a simple concept with some fairly deep semantics. Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part." Simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts.

in a composite aggregation, an object may be a part of only one composite at a time. For example, in a windowing system, a Frame belongs to exactly one Window. This is in contrast to simple aggregation, in which a part may be shared by several wholes. For example, in the model of a house, a Wall may be a part of one or more Room objects.



Association Classes

In an association between two classes, the association itself might have properties. For example, in an employer/employee relationship between a Company and a Person, there is a Job that represents the properties of that relationship that apply to exactly one pairing of the Person and Company. It wouldn't be appropriate to model this situation with a Company to Job association together with a Job to Person association.



Constraints

These simple and advanced properties of associations are sufficient for most of the structural relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines five constraints that may be applied to association relationships.

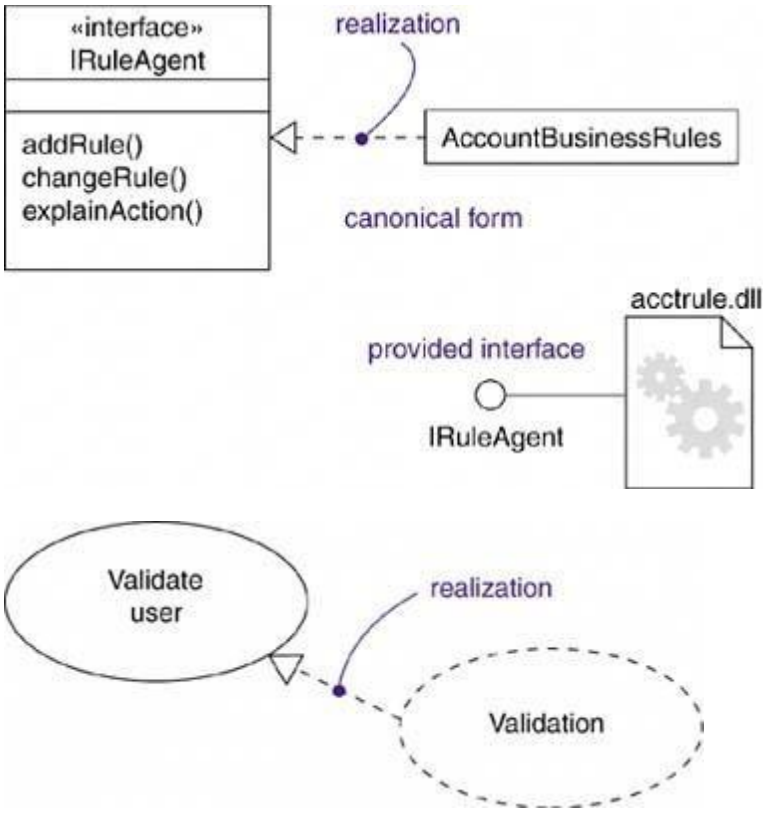
First, you can specify whether the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

1. ordered	Specifies that the set of objects at one end of an association are in an explicit order.
2. set	The objects are unique with no duplicates.
3. bag	The objects are non-unique, may be duplicates.
4. ordered set	The objects are unique but ordered.
5. list or sequence	The objects are ordered, may be duplicates.
6. readonly	A link, once added from an object on the opposite end of the association, may not be modified or deleted. The default in the absence of this constraint is unlimited changeability.

Realizations

A *realization* is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.

Realization is different enough from dependency, generalization, and association relationships that it is treated as a separate kind of relationship. Semantically, realization is somewhat of a cross between dependency and generalization, and its notation is a combination of the notation for dependency and generalization.



Common Modeling Techniques

Modeling Webs of Relationships

When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.

When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

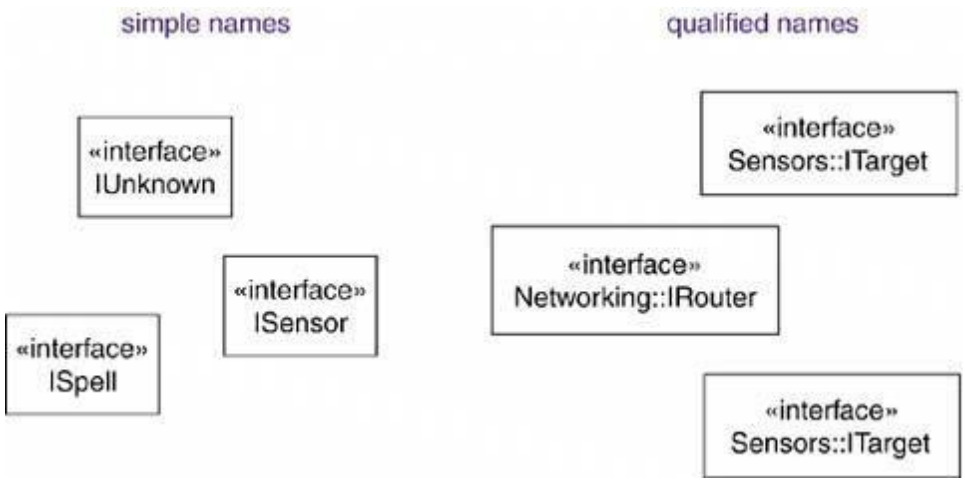
3. Interface, Types and Roles: Terms and Concepts

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

Names

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



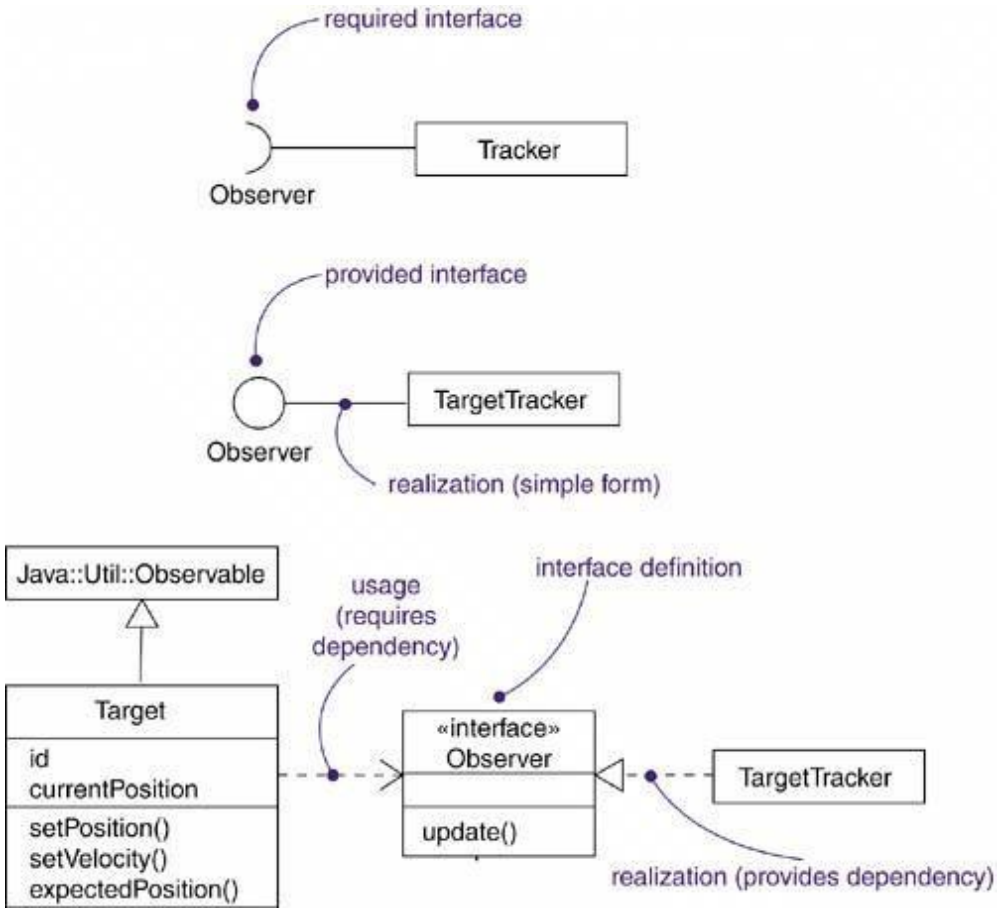
Operations

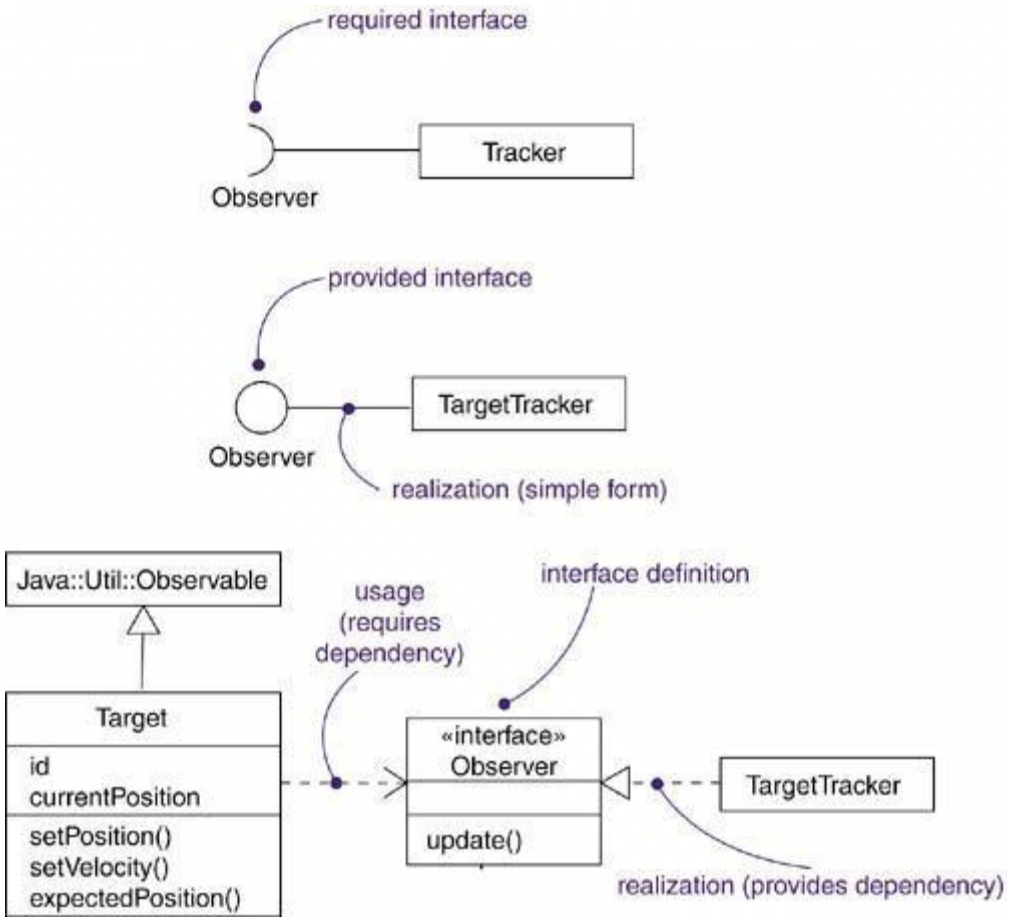
An interface is a named collection of operations used to specify a service of a class or of a component. Unlike classes or types, interfaces do not specify any implementation (so they may not include any methods, which provide the implementation of an operation). Like a class, an interface may have any number of operations. These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.

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.


required interface
Tracker
Observer
provided interface
TargetTracker
Observer
realization (simple form)
Java::Util::Observable
usage
(requires
dependency)
interface definition
Target
id
currentPosition
setPosition()
setVelocity()
expectedPosition()
«interface»
Observer
update()
TargetTracker
realization (provides dependency)



Understanding an Interface

When you are handed an interface, the first thing you'll see is a set of operations that specify a service of a class or a component. Look a little deeper and you'll see the full signature of those operations, along with any of their special properties, such as visibility, scope, and concurrency semantics.

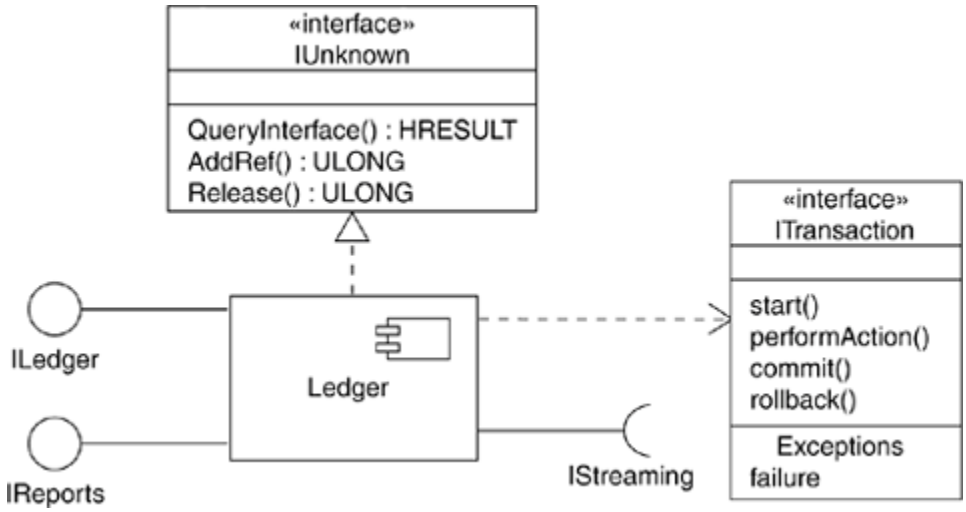
Common Modeling Techniques

Modeling the Seams in a System

o model the seams in a system,

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
- Package logically related sets of these operations and signals as interfaces.

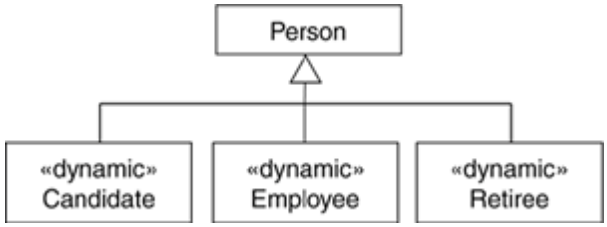
- For each such collaboration in your system, identify the interfaces it requires from (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.



Modeling Static and Dynamic Types

o model a dynamic type,

- Specify the different possible types of that object by rendering each type as a class (if the abstraction requires structure and behavior) or as an interface (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. You can mark them with the «dynamic» stereotype. (This is not a predefined UML stereotype, but one that you can add.)
- In an interaction diagram, properly render each instance of the dynamically typed class. Display the type of the instance in brackets below the object's name, just like a state. (We are using UML syntax in a novel way, but one that we feel is consistent with the intent of states.)



4. Packages

Terms and Concepts

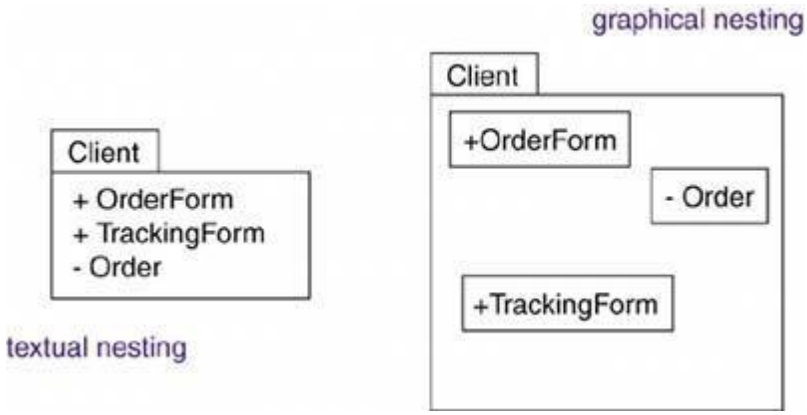
A *package* is a general-purpose mechanism for organizing the model itself into a hierarchy; it has no meaning to the execution. Graphically, a package is rendered as a tabbed folder. The name of the package goes in the folder (if its contents are not shown) or in the tab (if the contents of the folder are shown).

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 qualified name is the package name prefixed by the name of the package in which that package lives, if any. A double colon (::) separates package names.

Owned Elements

A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages. Ownership 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.

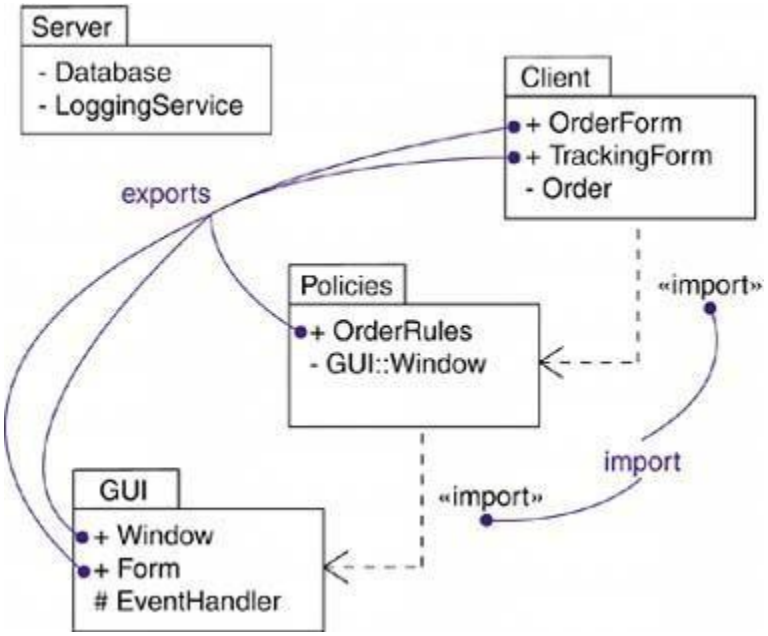


Visibility

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

Importing and Exporting

suppose that instead you put A in one package and B in another package, both packages sitting side by side. Suppose also that A and B are both declared as public parts of their respective packages. This is a very different situation. Although A and B are both public, accessing one of the classes from within the other package requires a qualified name. However, if A's package imports B's package, A can now see B directly, although still B cannot see A without a qualified name. Importing adds the public elements from the target package to the public namespace of the importing package. In the UML, you model an import relationship as a dependency adorned with the stereotype import. By packaging your abstractions into meaningful chunks and then controlling their access by importing, you can control the complexity of large numbers of abstractions.

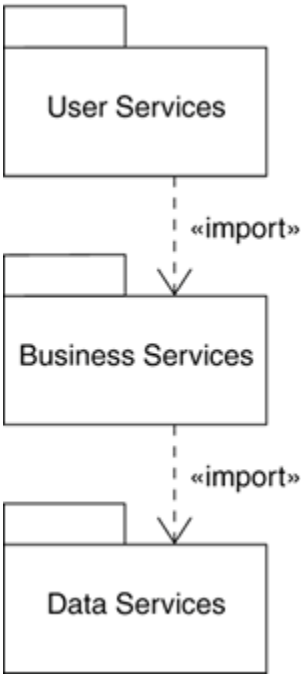


Common Modeling Techniques

Modeling Groups of Elements

To model groups of elements,

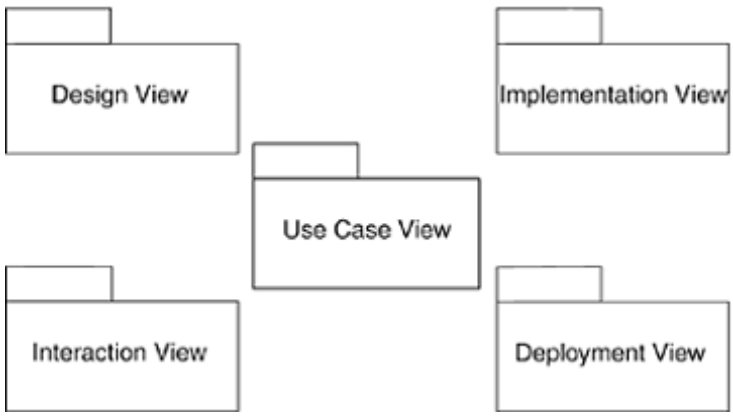
- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark 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,

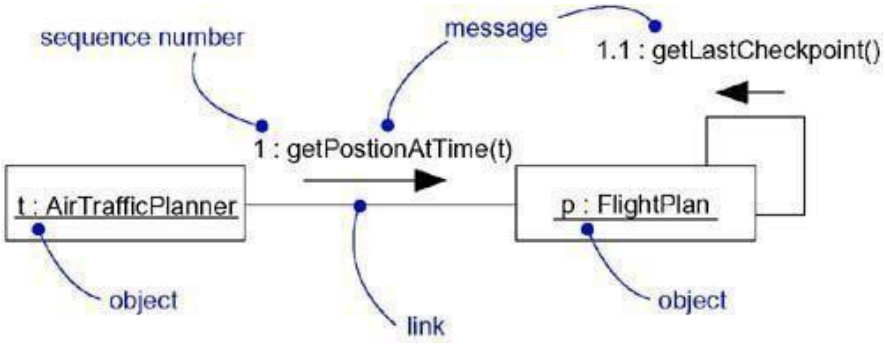
- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, an interaction view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient 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.



Basic Behavioral Modelling: Interactions, Interaction diagrams, Use cases, Use case Diagrams, Activity Diagrams.

INTERACTIONS

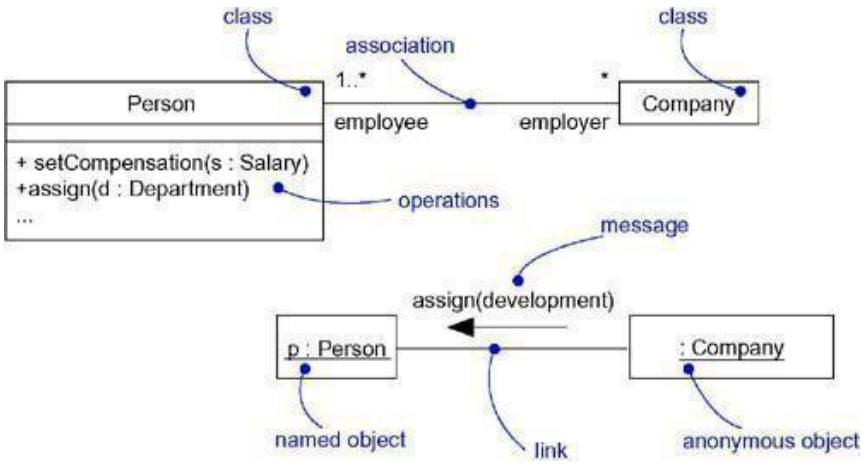
- An interaction is a behavior that is composed of a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- A message specifies the communication between objects for an activity to happen. It has following parts: its name, parameters (if any), and sequence number.
- Objects in an interaction can be concrete things or prototypical things.



A link is a semantic connection(path) among objects through which a message/s can be send. A link is an instance of an association. The semantics of link can be enhanced by using following prototypes as adornments

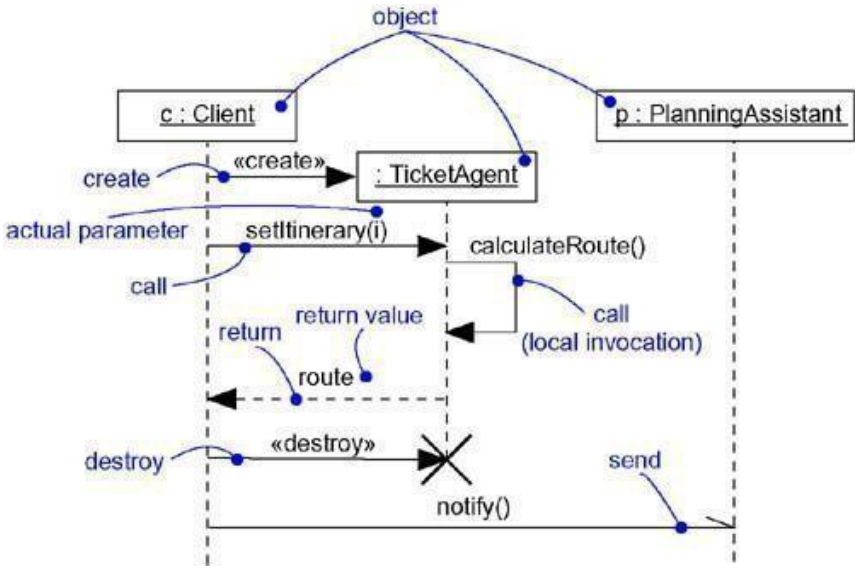
<<association>> – Specifies that the corresponding object is visible by association

<<self>> – Specifies that the corresponding object is visible because it is the dispatcher of the operation <<global>> – Specifies that the corresponding object is visible because it is in an enclosing scope <<local>> – Specifies that the corresponding object is visible because it is in a local scope <<parameter>> – Specifies that the corresponding object is visible because it is a parameter.



message – indicates an action to be done. Complex expressions can be written on arbitrary string of message.Different types of messages are:

- Call -Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
- Return – Returns a value to the caller
- Send – Sends a signal to an object
- Create – Creates an object
- Destroy – Destroys an object; an object may commit suicide by destroying itself



Sequencing

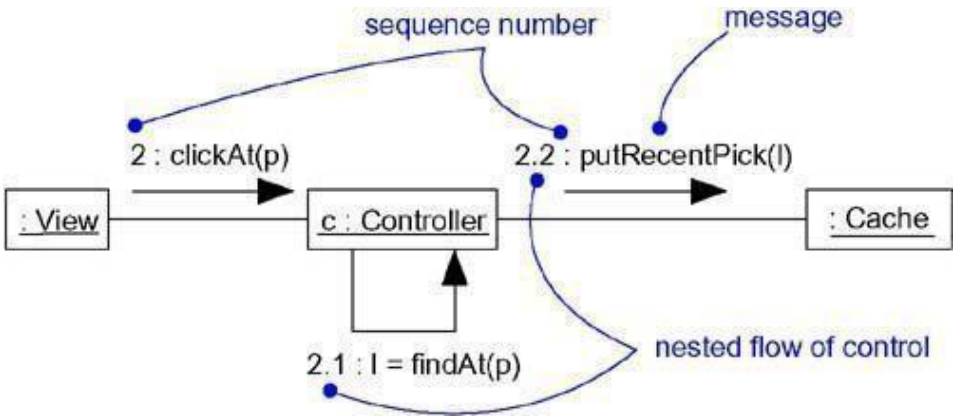
- a sequence is a stream of messages exchange between objects
- sequence must have a beginning and is rooted in some process or thread
- sequence will continue as long as the process or thread that owns it lives

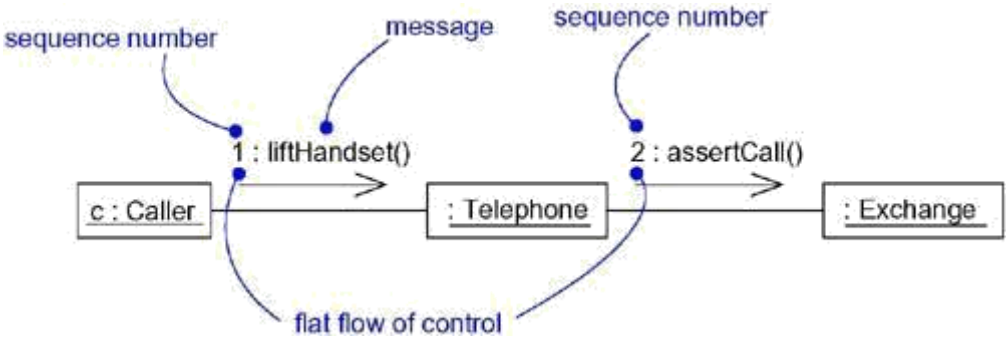
Flow of control (2 types)

In each flow of control, messages are ordered in sequence by time and are visualized by prefixing the message with a sequence number set apart by a colon separator

- A *procedural or nested flow of control* is rendered by using a filled solid arrowhead,
- A *flat flow of control* is rendered by using a stick arrowhead
- *Distinguishing one flow of control from another* by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence.
- more-complex forms of sequencing, such as iteration, branching, and guarded messages

can be modeled in UML.





Creation, Modification, and Destruction of links

Enabled by *adding the following constraints to the element*

- new – Specifies that the instance or link is created during execution of the enclosing interaction
- destroyed – Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
- transient – Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

Representation of interactions

interaction goes together with objects and messages.

represented by time ordering of its messages (sequence diagram), and by emphasizing the structural organization of these objects that send and receive messages (collaboration diagram)

Common Modeling Techniques

Modeling a Flow of Control

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an

individual operation

 Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role

 If model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction

 In time order, specify the messages that pass from object to object As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction

 Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role

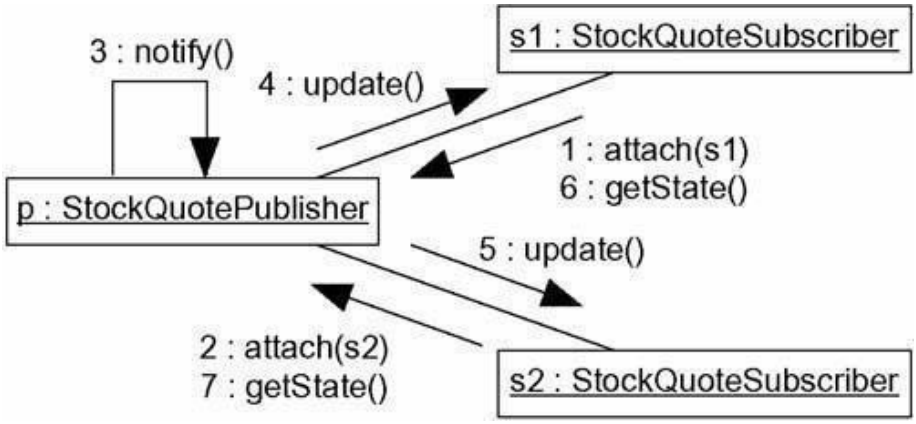


Figure: Flow of Control by time

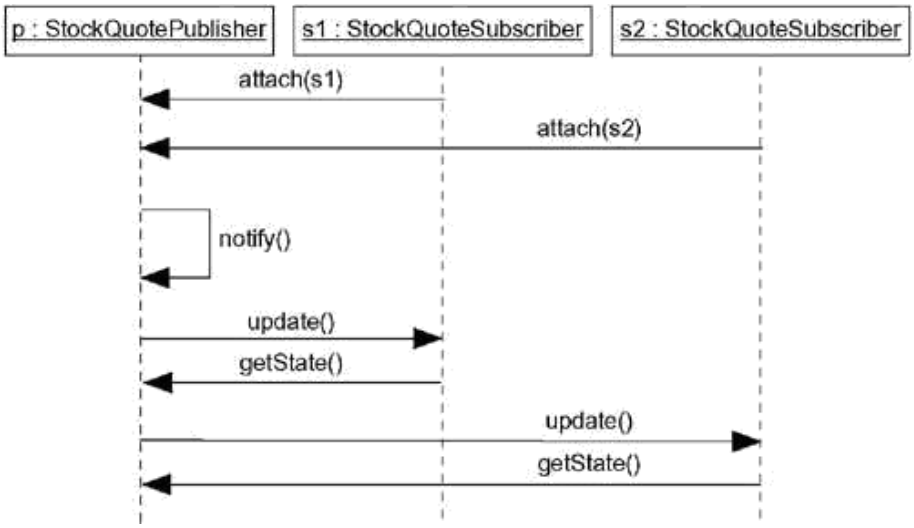


Figure: Flow of Control by Organization

INTERACTION DIAGRAMS

 An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them

 Interaction diagrams commonly contain Objects, Links, Messages

 interaction diagrams are used to model the dynamic aspects of a system

 An interaction diagram is basically a projection of the elements found in an interaction.

 It may contain notes and constraints

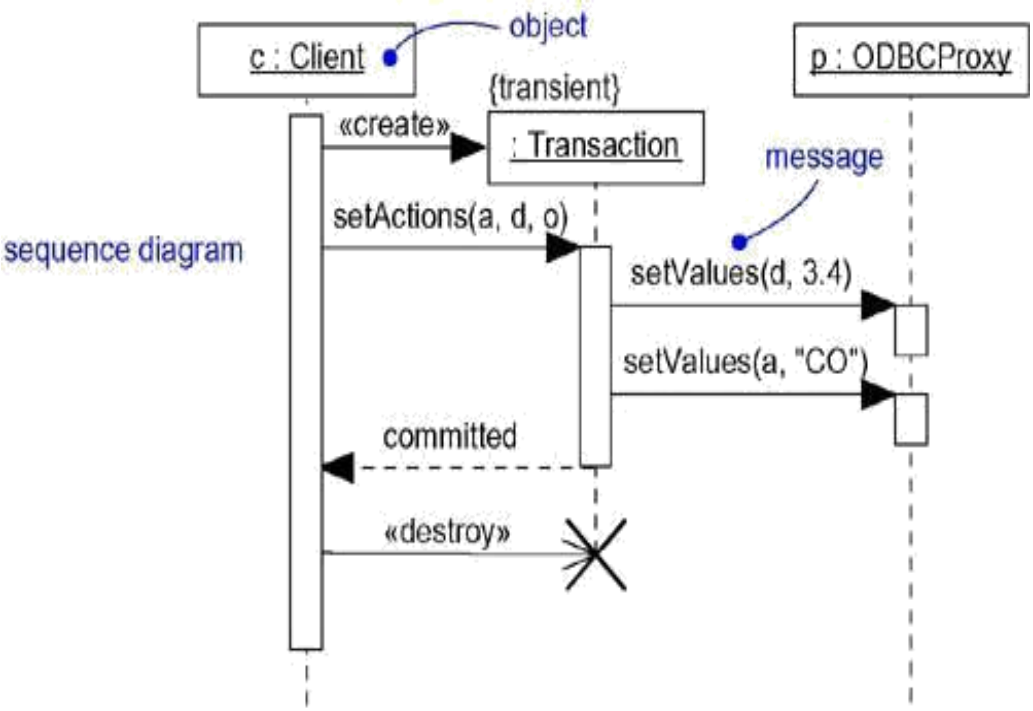
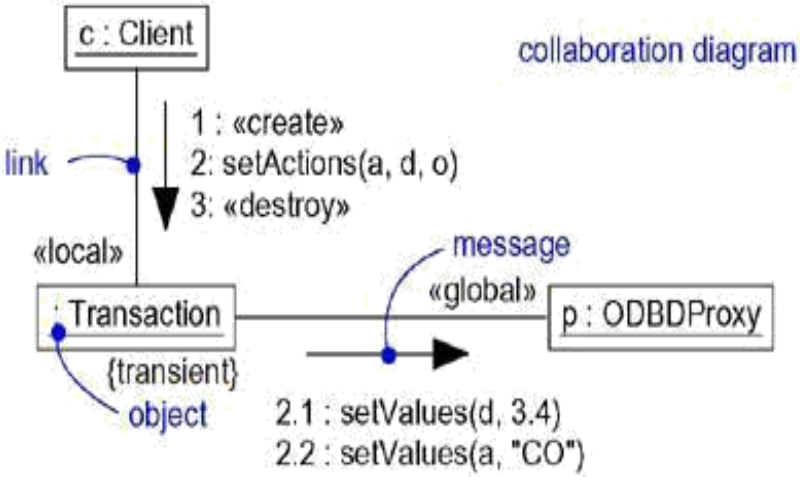


Fig: Interaction Diagrams



Sequence Diagrams

 A sequence diagram is an interaction diagram that emphasizes the time ordering of messages

 Graphically it is a table that shows objects arranged along the X axis and messages ordered in increasing time along the Y axis

 place the objects that participate in the interaction at the top of your diagram, across the X axis, object that initiates the interaction at the left, and increasingly more subordinate objects to the right

 place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom

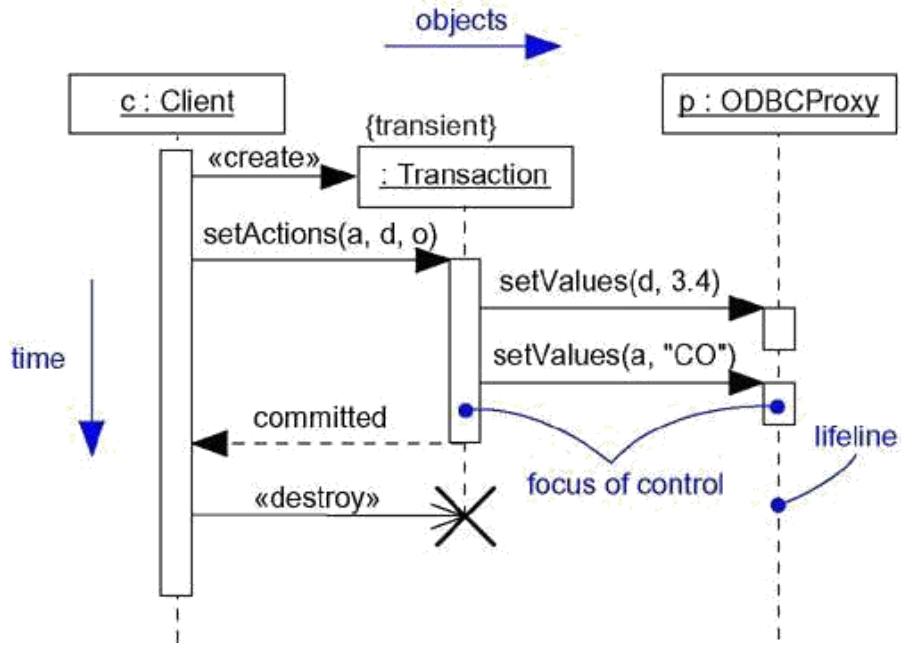


Figure: Sequence Diagram

Sequence diagrams* have *two features that distinguish* them from *collaboration diagrams

 First, there is the *object lifeline* which is a vertical dashed line that represents the existence of an object over a period of time

 Second, there is the *focus of control* which is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure

Collaboration Diagrams

 A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages

 Graphically it is a collection of vertices and arcs

 more-complex flows, involving iterations and branching are modeled as [i := 1n] (or just), [x > 0]

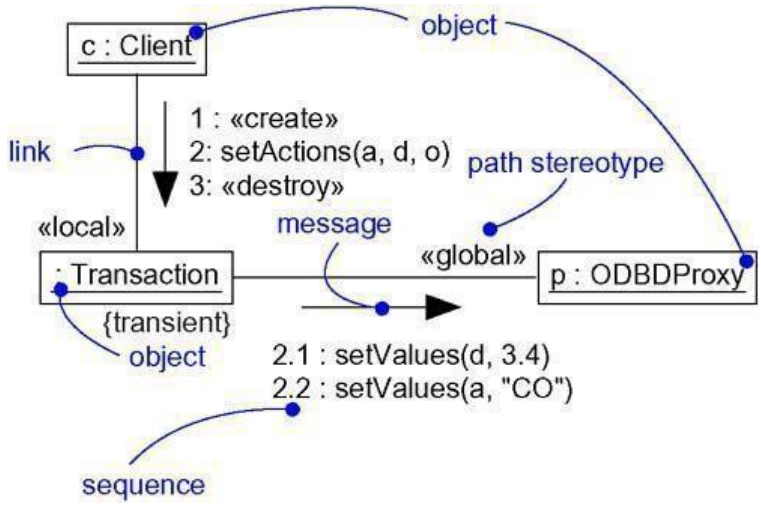


Figure: Collaboration Diagram

Collaboration diagrams* have two *features that distinguish* them from *sequence diagrams

 First, there is the *path* to indicate how one object is linked to another, attach a path stereotype to the far end of a link such as local, parameter, global, and self

 Second, there is the *sequence number* to indicate the time order of a message denoted by prefixing the message with a number, nesting is indicated by Dewey decimal numbering (eg:- 1 is the first message; 1.1 is the first message nested in message 1.)

Semantic Equivalence

sequence diagrams and collaboration diagrams are semantically equivalent that means conversion to the other is possible without any loss of information.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

To model a flow of control by time ordering,

 Set the context for the interaction, whether it is a system, subsystem, operation, or class or one scenario of a use case or collaboration

 Set the stage for the interaction by identifying which objects play a role in the interaction.

 Set the lifeline for each object. Objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages

 Starting with the message that initiates this interaction, lay out each subsequent message from

top to bottom between the lifelines, showing each message's properties .

 If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control

 If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints

 If you need to specify this flow of control more formally, attach pre- conditions and post-conditions to each message

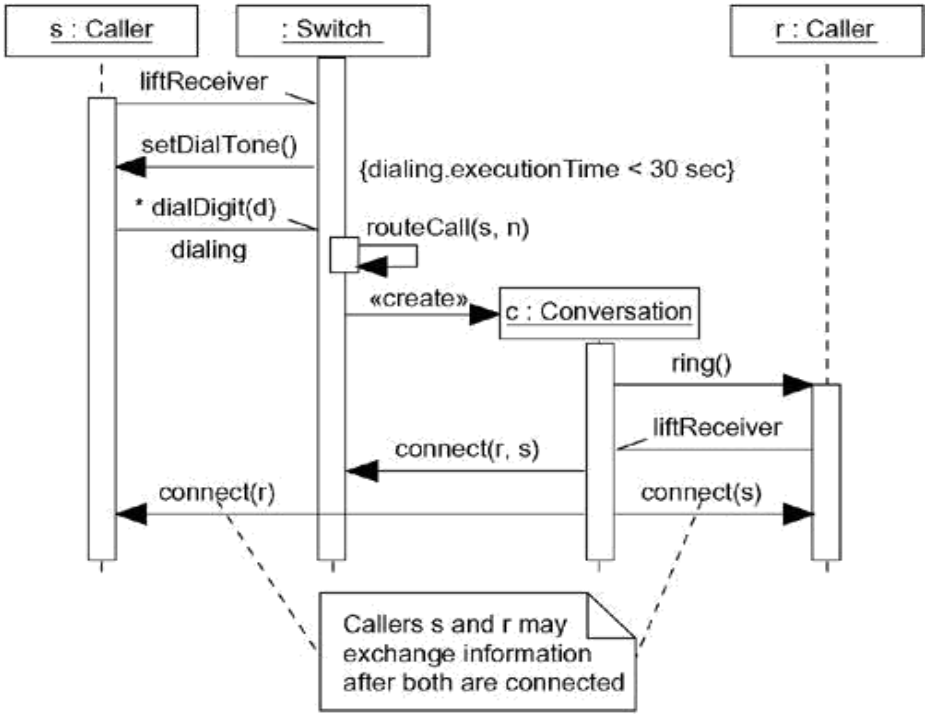


Figure: Modeling Flows of Control by Time Ordering

Modeling Flows of Control by Organization

To model a flow of control by organization

 Set the context for the interaction, whether it is a system, subsystem, operation, or class or one scenario of a use case or collaboration

 Set the stage for the interaction by identifying which objects play a role in the interaction .

 Set the initial properties of each of these objects If the attribute values, tagged values, state or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as become or copy .

 Specify the links among these objects, along which messages may pass

1. Lay out the association links first; these are the most important ones, because they represent structural connections

2. Lay out other links next, and adorn them with suitable path stereotypes (such as global and

local) to explicitly specify how these objects are related to one another

 Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate Show nesting by using Dewey decimal numbering.

 If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.

 If you need to specify this flow of control more formally, attach pre- and post-conditions to each message

The Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects

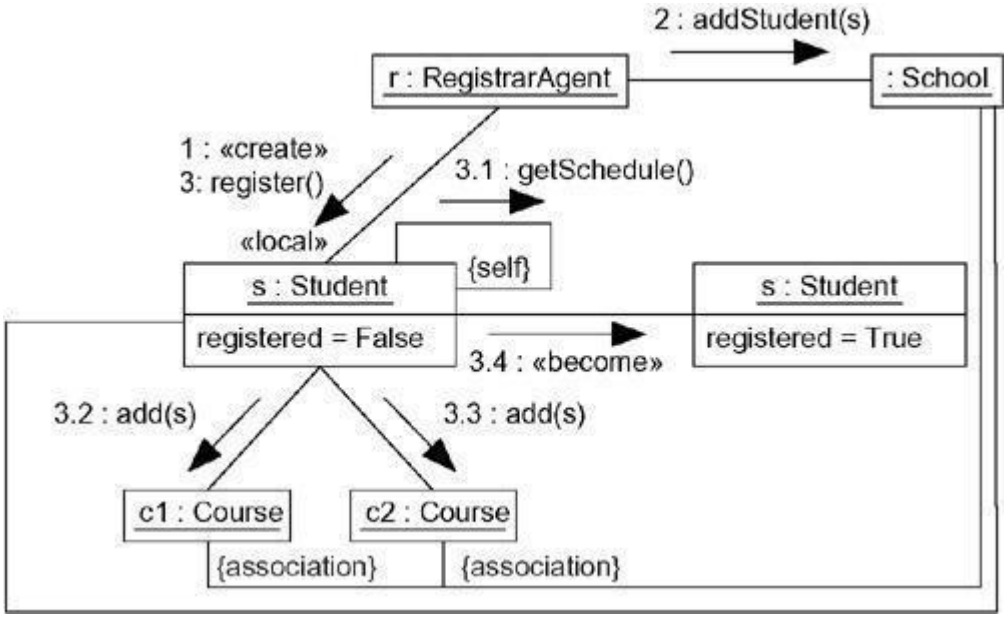


Figure: Modeling Flows of Control by Organization

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

So the purposes of interaction diagram can be describes as:

- To capture dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

USE CASES

- use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
- use case captures the intended behavior of the system (or subsystem, class, or interface) without having to specify how that behavior is implemented
- a use case is represented as an ellipse
- Every use case must have a name that distinguishes it from other use cases: simple name and path name Figure 2: Simple and Path Names

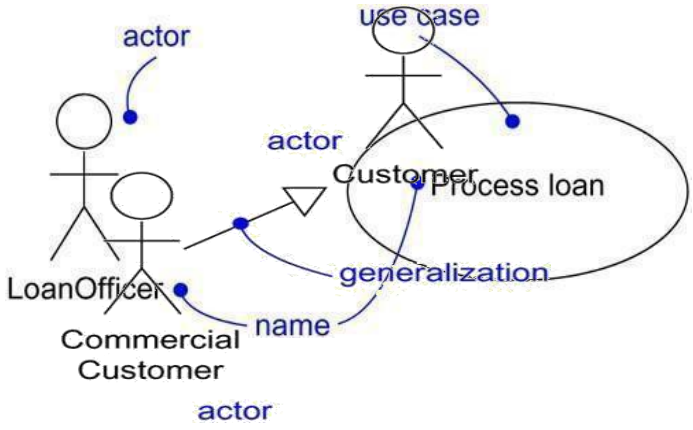


Figure 1: Actors and Use Cases

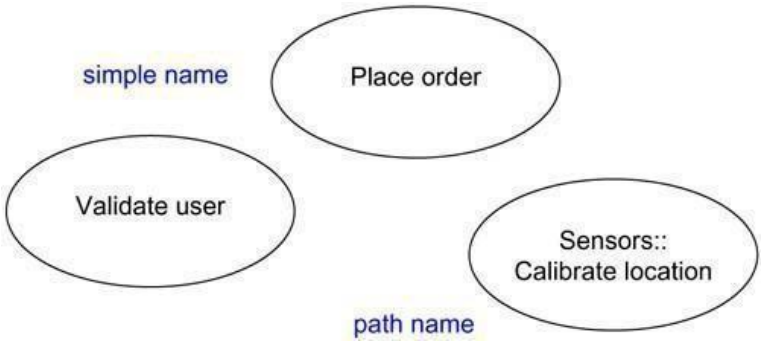


Figure 2: Simple and Path Names

Actors

- actor represents a coherent set of roles that users of use cases play when interacting with these use cases
- an actor represents a role that a human, a hardware device, or even another system plays with a system Figure 3: Actors

Actors may be connected to use cases by association

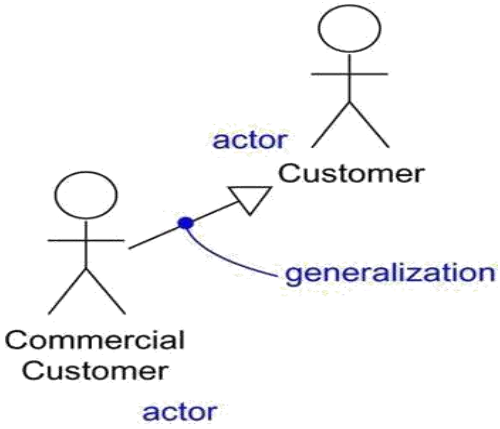


Figure 3: Actors

Use Cases & Flow of Events

flow of events include how and when the use case starts and ends when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

The behavior of a use case can be specified by describing a flow of events in text.

There can be Main flow of events and one or more Exceptional flow of events.

Use Cases and Scenarios

- A scenario is a specific sequence of actions that illustrates behavior
- Scenarios are to use cases, as instances are to classes means that scenario is basically one instance of a use case
- for each use case, there will be primary scenarios and secondary scenarios.

Use Cases and Collaborations

- Collaborations are used to implement the behavior of use cases with society of classes and other elements that work together
- It includes static and dynamic structure

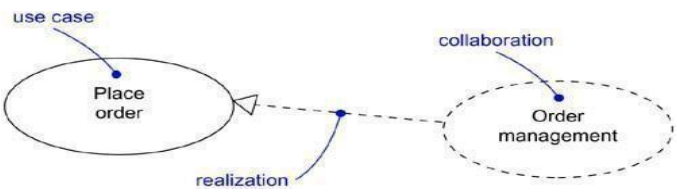


Figure 4: Use Cases and Collaborations

Organizing Use Cases

- organize use cases by grouping them in packages
- organize use cases by specifying generalization, include, and extend relationships among them

Generalization, Include and Extend

An *include relationship* between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. An include relationship can be rendered as a dependency, stereotyped as include

An *extend relationship* between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. An extend relationship can be rendered as a dependency, stereotyped as extend. *extension points* are just labels that may appear in the flow of the base use case

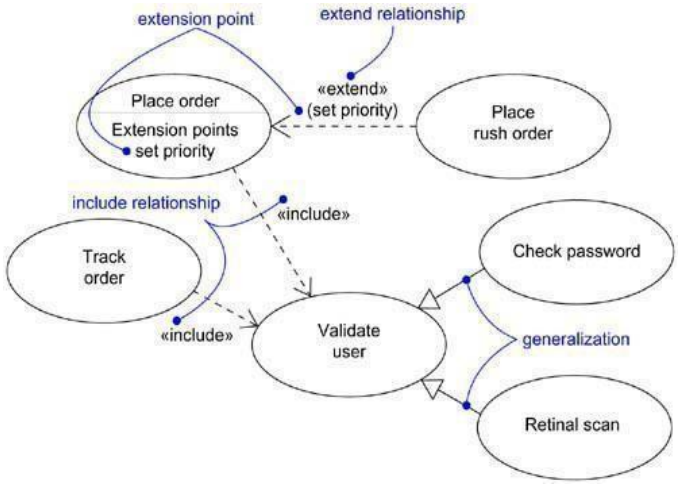


Figure 5: Generalization, Include and Extend

Modeling the Behavior of an Element

To model the behavior of an element,

- Identify the actors that interact with the element Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions
- Organize actors by identifying general and more specialized roles
- For each actor, consider the primary ways in which that actor interacts with the element Consider also interactions that change the state of the element or its environment or that involve a response to some event

- Consider also the exceptional ways in which each actor interacts with the element

- Organize these behaviors as use cases, applying include and extend

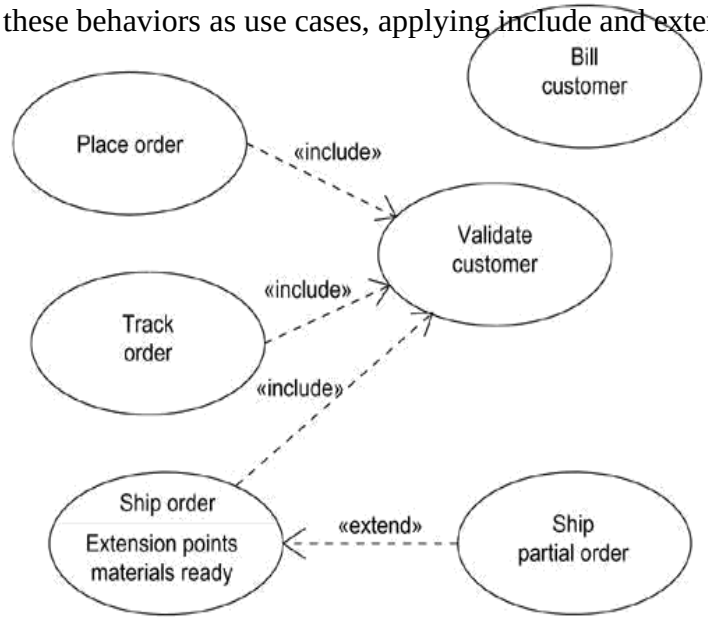


Figure 6: Modeling the Behavior of an Element

USE CASE DIAGRAMS

- A use case diagram is a diagram that shows a set of use cases and actors and their relationships
- Use case diagrams commonly contain Use cases, Actors, Dependency, generalization, and association relationships
- use case diagrams may contain packages, certain times instances of use cases, notes and constraints
- apply use case diagrams to model the static use case view of a system by modeling the context of a system and by modeling the requirements of a system

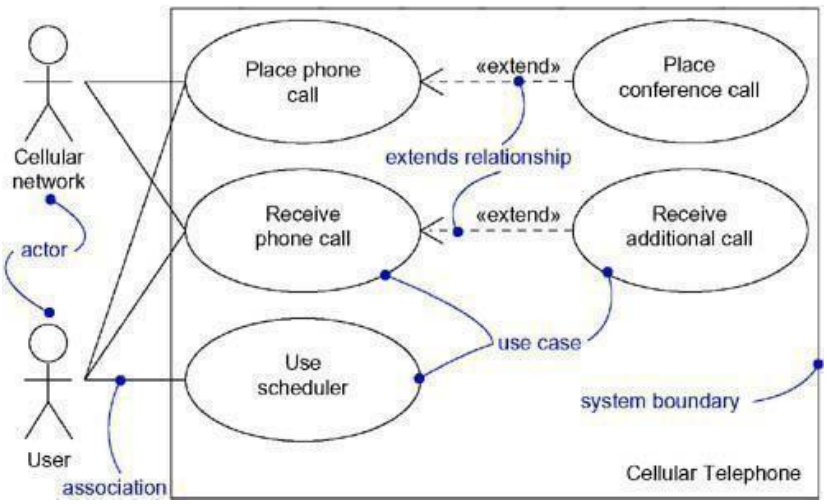


Figure 1: A Use Case Diagram

Modeling the Context of a System

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance
- Organize actors that are similar to one another in a generalization / specialization hierarchy
- provide a stereotype for each such actor
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases

Figure 2 shows the context of a credit card validation system, with an emphasis on the actors that surround the system.

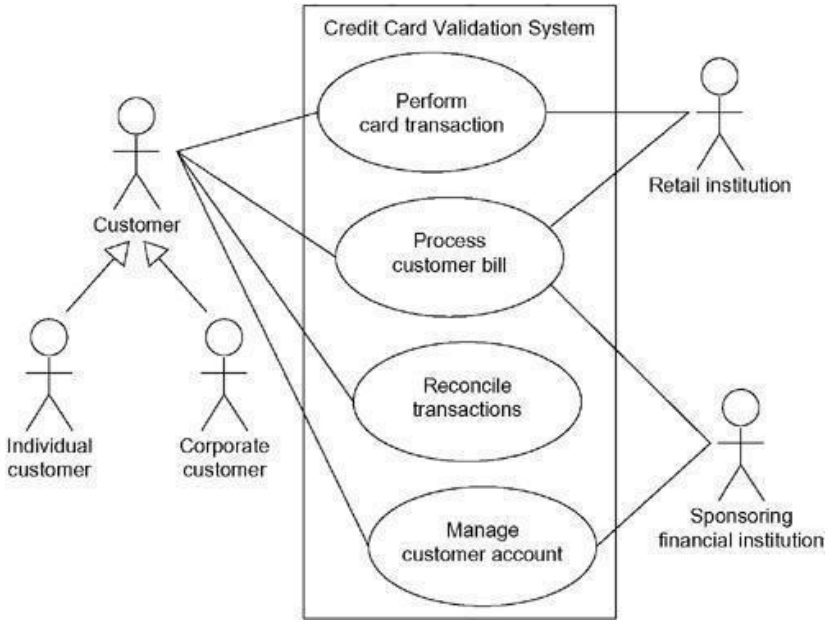


Figure 2: Modeling the Context of a System

Common Modeling Techniques

Modeling the Requirements of a System

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it
- For each actor, consider the behavior that each expects or requires the system to provide
- Name these common behaviors as use cases

- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows
- Model these use cases, actors, and their relationships in a use case diagram
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system

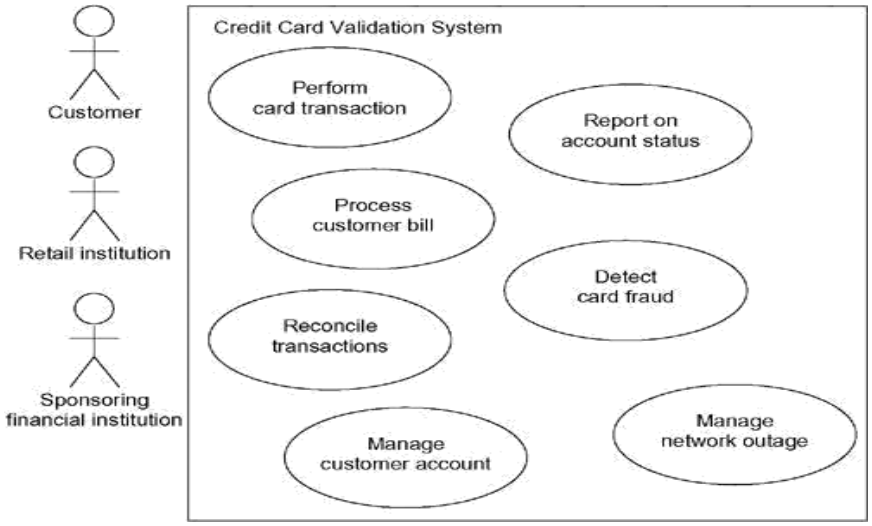


Figure 3: Modeling the Requirements of a System

ACTIVITY DIAGRAMS

- An activity diagram shows the flow from activity to activity
- an activity diagram shows the flow of an object, how its role, state and attribute values changes
- activity diagrams is used to model the dynamic aspects of a system
- Activities result in some action (Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression)
- an activity diagram is a collection of vertices and arcs
- Activity diagrams commonly contain Activity states and action states, Transitions, Objects
- activity diagrams may contain simple and composite states, branches, forks, and joins
- the initial state is represented as a solid ball and stop state as a solid ball inside a circle

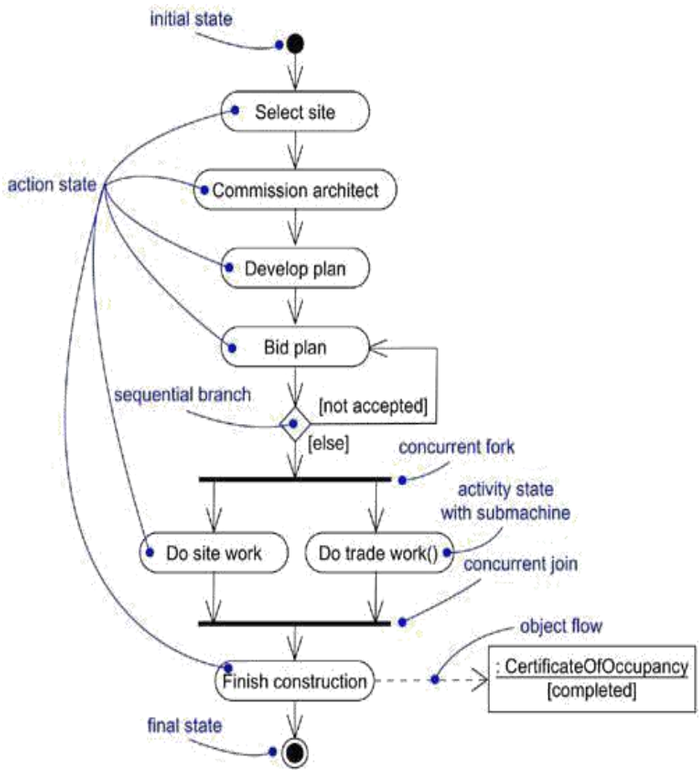


Figure 1: activity diagram

Action States

- The executable, atomic computations are called action states because they are states of the system, each representing the execution of an action
- Figure 2 Action States
- action states can't be decomposed
- action states are atomic, meaning that events may occur, but the work of the action state is not interrupted
- action state is considered to take insignificant execution time
- action states are special kinds of states in a state machine

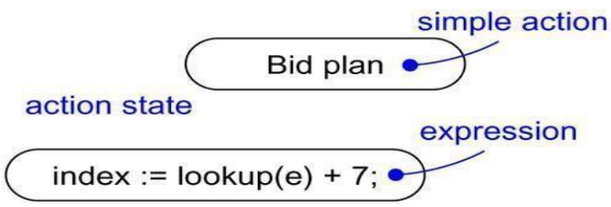


Figure 2: Action States

Activity States

- activity states can be further decomposed
- activity states are not atomic, meaning that they may be interrupted
- they take some duration to complete
- are just special kinds of states in a state machine

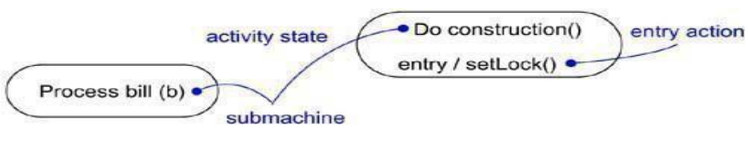


Figure 3: Activity States

Transitions

- transitions shows the path from one action or activity state to the next action or activity state
- a transition is represented as a simple directed line

Triggerless Transitions

- Triggerless Transitions are transitions where control passes immediately once the work of the source state is done

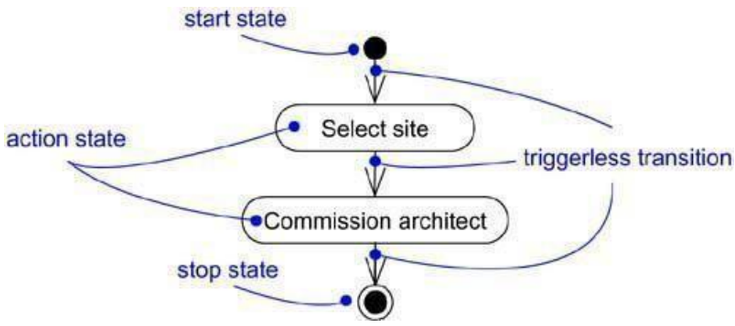


Figure 4: Triggerless Transitions

Branching

- represent a branch as a diamond
- A branch may have one incoming transition and two or more outgoing ones
- each outgoing transition contains a guard expression, which is evaluated only once on entering the branch

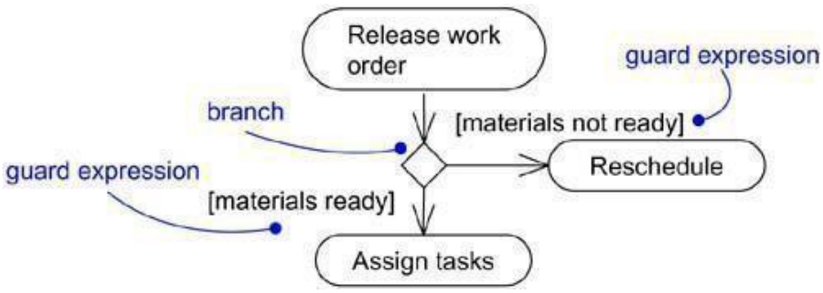


Figure 5: Branching

Forking and Joining

- A *fork* may have one incoming transition and two or more outgoing transitions each of which represents an independent flow of control
- a fork represents the splitting of a single flow of control into two or more concurrent flows of control
- Below the fork, the activities associated with each of these paths continues in parallel
- A *join* may have two or more incoming transitions and one outgoing transition
- Above the join, the activities associated with each of these paths continues in parallel
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join
- the forking and joining of the parallel flows of control are specified by a *synchronization*

bar

- A synchronization bar is rendered as a thick horizontal or vertical line

Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.

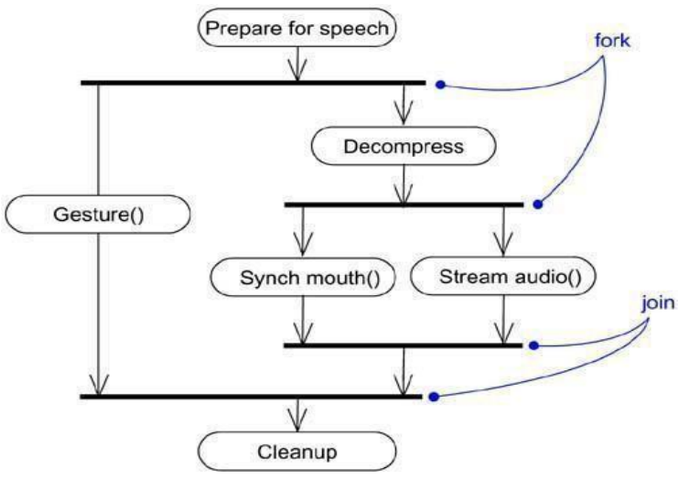


Figure 6: Forking and Joining

Swimlanes

- swimlanes partitions activity diagrams into groups having activity states where each group represents the business organization responsible for those activities
- Each swimlane has a name unique within its diagram
- swimlane represents a high-level responsibility for part of the overall activity of an activity diagram
- each swimlane is implemented by one or more classes

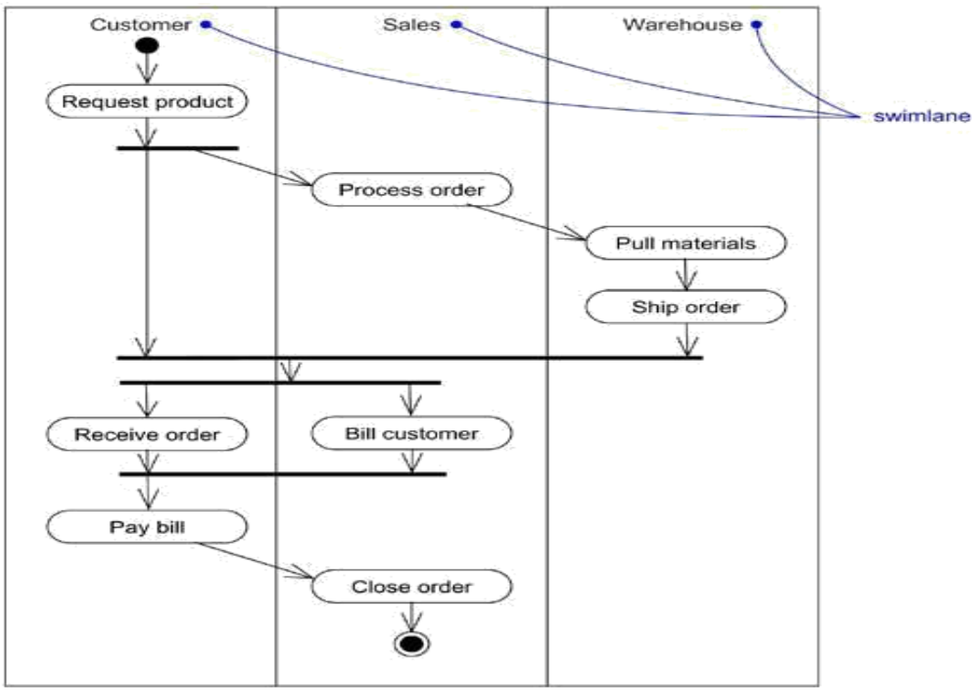


Figure 7: Swimlanes

Object Flow

- object flow indicates the participation of an object in a flow of control, it is represented with the help of dependency relationships.

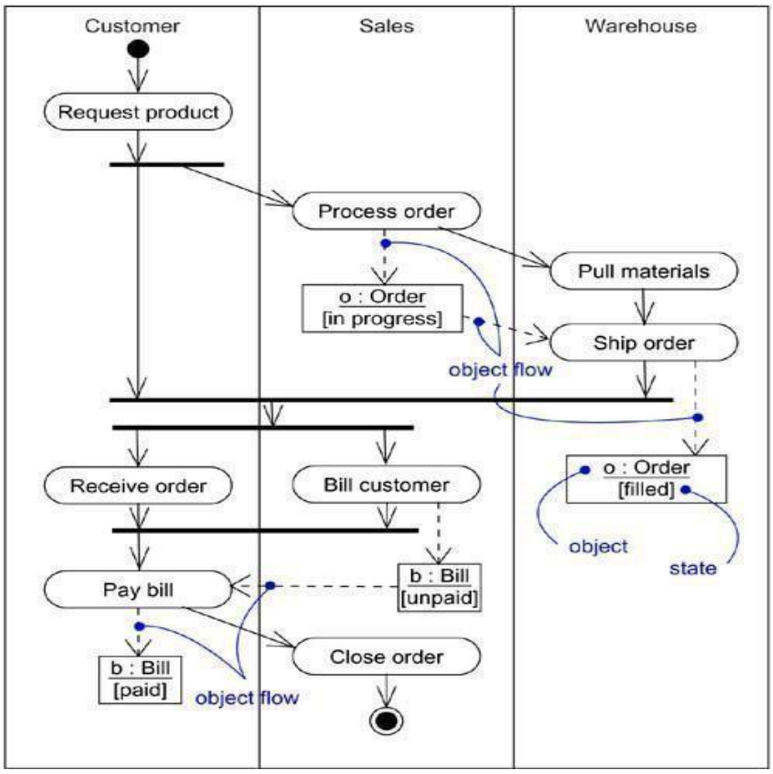


Figure 8: Object Flow

Common Modeling Techniques

Modeling a Workflow

To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.

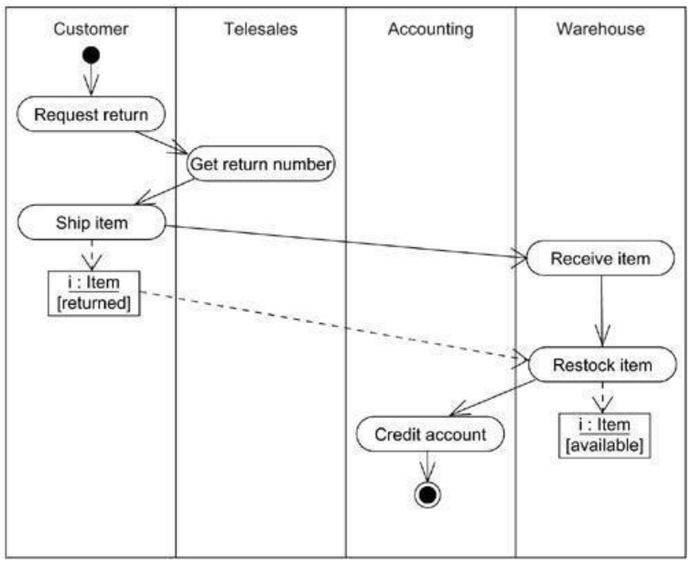


Figure: Modeling a Workflow

Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the post conditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Figure shows an activity diagram that specifies the algorithm of the operation intersection b/w lines.

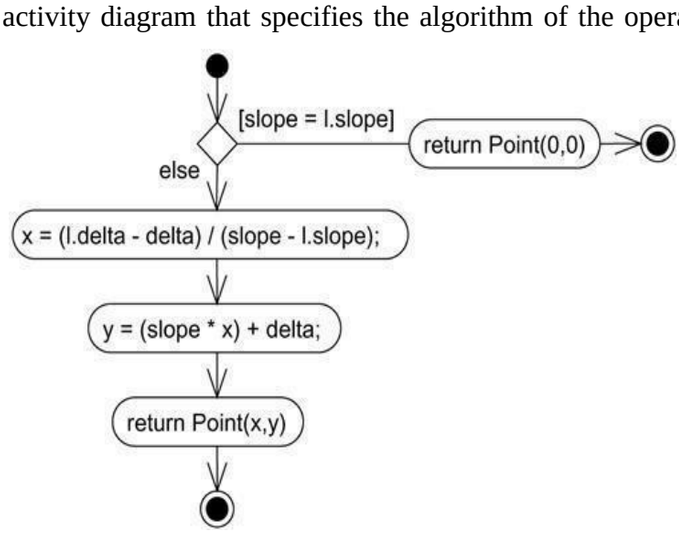


Figure 10: Modeling an Operation

UNIT – V

Advanced Behavioral Modelling: Events and signals, state machines, processes and Threads, time and space, state chart diagrams.

EVENTS AND SIGNALS

Events

- An event is the specification of a significant occurrence that has a location in time and space.
- Anything that happens is modeled as an event in UML.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition
- four kinds of events – signals, calls, the passing of time, and a change in state.
- Events may be external or internal and asynchronous or synchronous.

Asynchronous events are events that can happen at arbitrary times eg:- signal, the passing of time, and a change of state.

Synchronous events, represents the invocation of an operation eg:- Calls

External events are those that pass between the system and its actors.

Internal events are those that pass among the objects that live inside the system.

A signal is an event that represents the specification of an asynchronous stimulus communicated between instances.

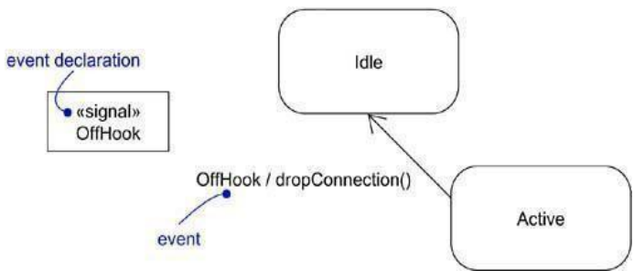


Figure 1: Events

kinds of events

1. Signal Event

A signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal

- a signal event is an asynchronous event
- signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters
- A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction
- signals are modeled as stereotyped classes and the relationship between an operation and the

events by using a dependency relationship, stereotyped as send.

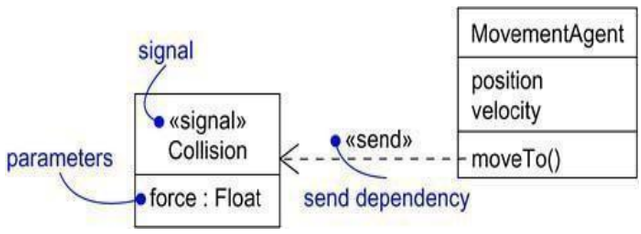


Figure: Signals

2. Call Event

- a call event represents the dispatch of an operation
- a call event is a synchronous event

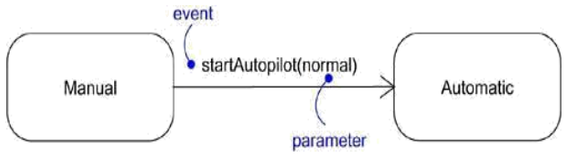


Figure: Call Events

3. Time and Change Events

- A *time event* is an event that represents the passage of time.
- modeled by using the keyword ‘after’ followed by some expression that evaluates to a period of time which can be simple or complex.
- A *change event* is an event that represents a change in state or the satisfaction of some condition
- modeled by using the keyword ‘when’ followed by some Boolean expression

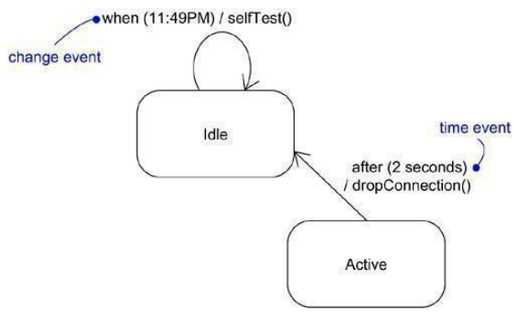


Figure: Time and Change Events

4. Sending and Receiving Events

For synchronous events (Sending or Receiving) like call event, the sender and the receiver are in a rendezvous (the sender dispatches the signal and wait for a response from the receiver) for the duration of the operation. When an object calls an operation, the sender dispatches the operation and then waits for the receiver.

For asynchronous events (Sending or Receiving) like signal event, the sender and receiver do not rendezvous ie,the sender dispatches the signal but does not wait for a response from the receiver. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.

Call events can be modeled as operations on the class of the object.

Named signals can be modeled by naming them in an extra compartment of the class

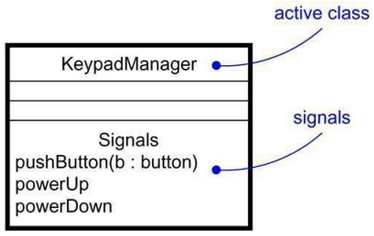


Figure: Signals and Active Classes

Common Modeling Techniques Modeling family of signals

To model a family of signals,

 Consider all the different kinds of signals to which a given set of active objects may respond.

 Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.

 Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

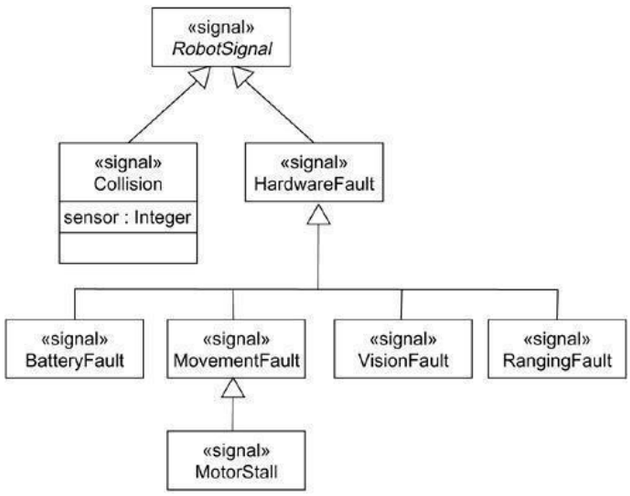


Figure: Modeling Families of Signals

Modeling Exceptions

To model exceptions,

 For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.

 Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.

 For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.

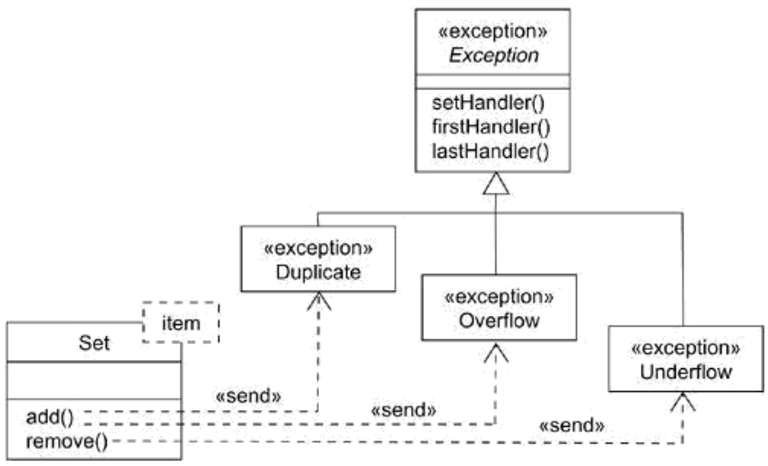


Figure: Modeling Exceptions

STATE MACHINES

 A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events.

 Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.

 State machines are used to specify the behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past.

 state machines are used to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

 A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

 An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle, Activating, Active, and Shutting Down.

 a state name must be unique within its enclosing state

 A state has five parts: *Name, Entry/exit actions, Internal transitions* – Transitions that are handled without causing a change in state,

 Substates – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates,

 Deferred events – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

 initial state indicates the default starting place for the state machine or substate and is represented as a filled black circle

 final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle

Initial and final states are pseudo-states

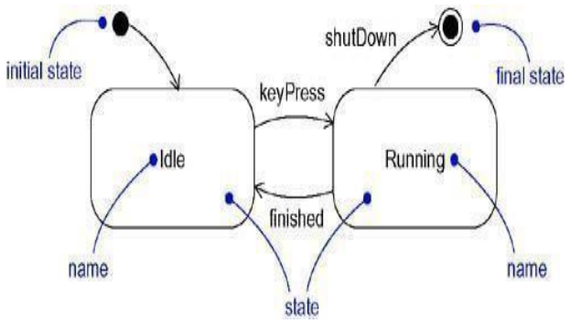


Figure: States

Transitions

 A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.

 Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.

 A transition has five parts:

Source state – The state affected by the transition,

Event trigger – a stimulus that can trigger a source state to fire on satisfying guard condition,

Guard condition – Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger,

Action – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object,

Target state – The state that is active after the completion of the transition.

 A transition may have multiple sources as well as multiple targets

 A *self-transition*is a transition whose source and target states are the same

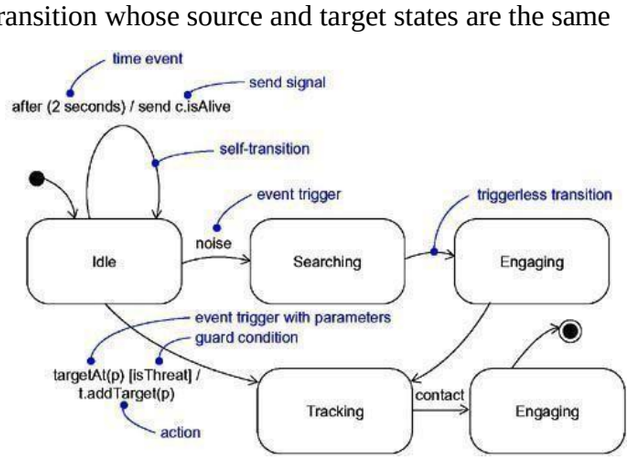


Figure: Transitions

Event Trigger

 An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.

 events may include signals, calls, the passing of time, or a change in state.

 An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.

 An event trigger may be polymorphic

Guard condition

 a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event

 A guard condition is evaluated only after the trigger event for its transition occurs

 A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

Action

 An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.

 Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object

An activity may be interrupted by other events.

Advanced States and Transitions

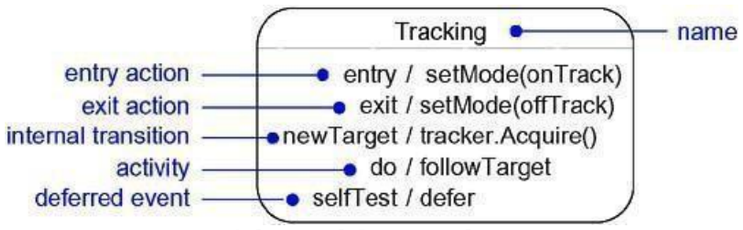


Figure: Advanced States and Transitions

Entry and Exit Actions

 Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event ‘entry’ with an appropriate action

 Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event ‘exit’, together with an appropriate action

Internal Transitions

 Internal Transitions are events that should be handled internally without leaving the state.

 Internal transitions may have events with parameters and guard conditions.

Activities

Activities make use of object’s idle time when inside a state. ‘do’ transition is used to specify the work that’s to be done inside a state after the entry action is dispatched.

Deferred Events

A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action ‘defer’.

Substates

 A substate is a state that’s nested inside another one.

 A state that has substates is called a composite state.

 A composite state may contain either *concurrent (orthogonal) or sequential (disjoint) sub states*.

 Substates may be nested to any level Sequential Substates

 Sequential Substates are those sub states in which an event common to the composite states can easily be exercised by each states inside it at any time

 sequential substates partition the state space of the composite state into disjoint states

 A nested sequential state machine may have at most one initial state and one final state

History States

 A history state allows composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.

 a shallow history state is represented as a small circle containing the symbol H

 The first time entry to a composite state doesn't have any history

 the symbol H designates a *shallow history*, which remembers only the history of the immediate nested state machine.

 the symbol H* designates *deep history*, which remembers down to the innermost nested state at any depth.

 When only one level of nesting, shallow and deep history states are semantically equivalent.

Concurrent Substates

 concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object

 Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow

 A nested concurrent state machine does not have an initial, final, or history state

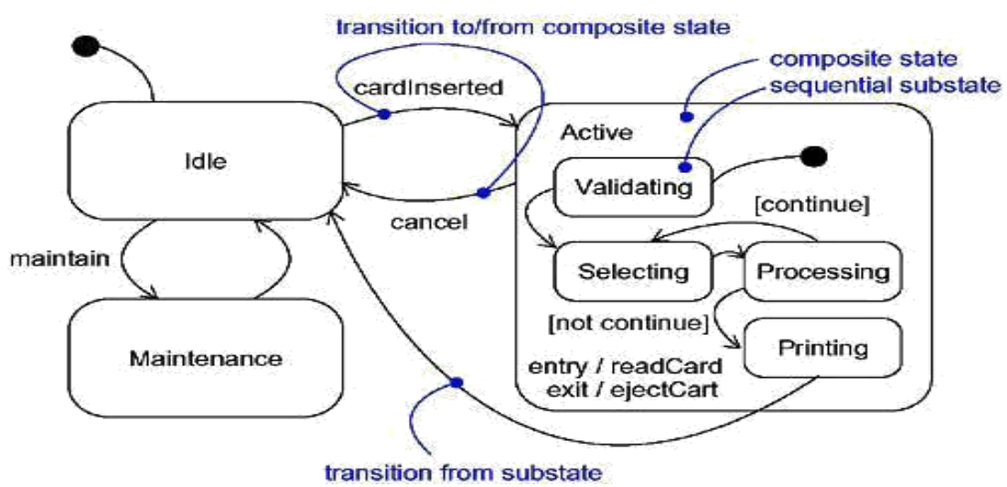


Figure : Sequential Substates

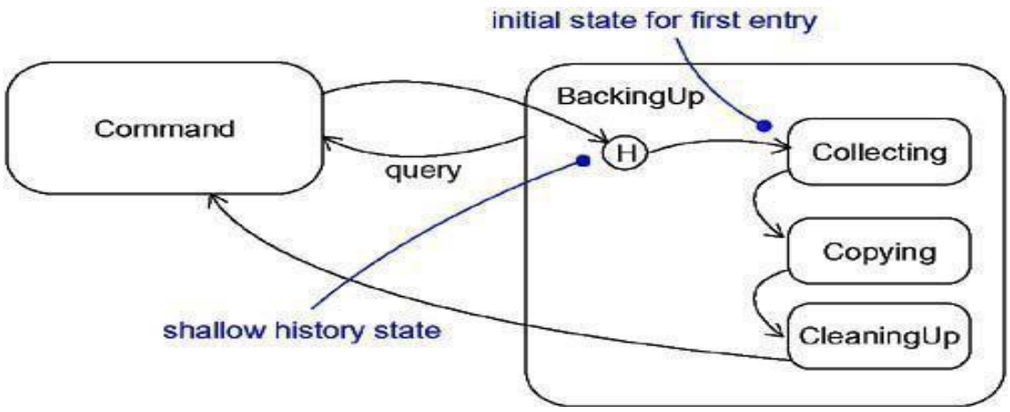


Figure: History State

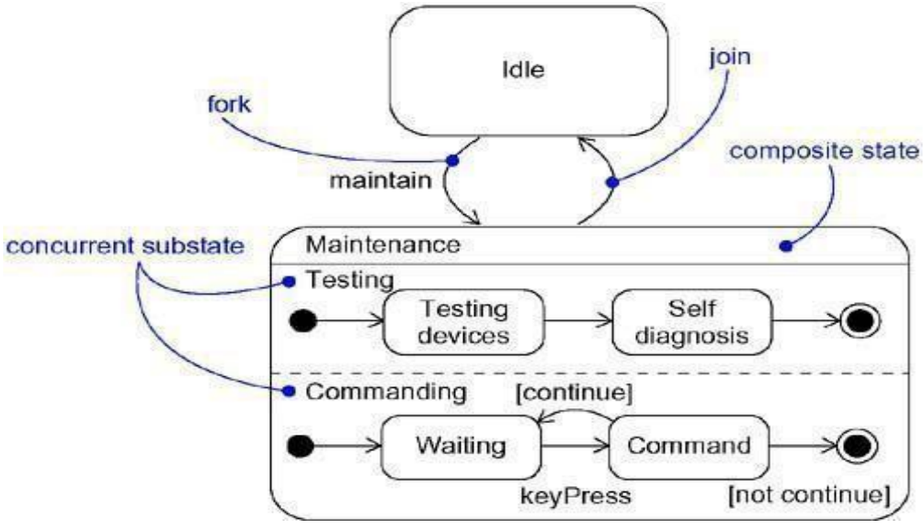


Figure: Concurrent Sub states

Common Modeling Techniques Modeling the Lifetime of an Object

To model the lifetime of an object,

 Set the context for the state machine, whether it is a class, a use case, or the system as a whole.

 If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.

If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.

 Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.

 Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.

 Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.

 Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).

 Expand these states as necessary by using substates.

 Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.

 Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.

 Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.

 After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure shows the state machine for the controller in a home security system

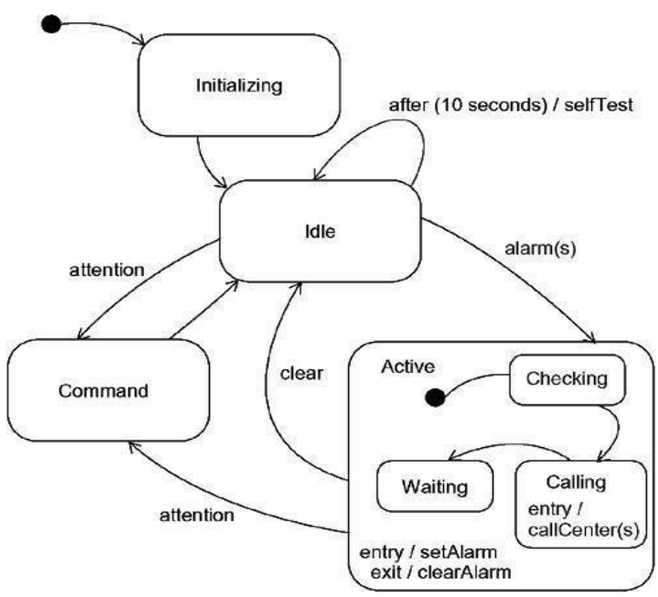


Figure: Modeling the Lifetime of an Object

PROCESSES AND THREADS

 A process is a heavyweight flow that can execute concurrently with other processes.

 A thread is a lightweight flow that can execute concurrently with other threads within the same process.

 An active object is an object that owns a process or thread and can initiate control activity.

 An active class is a class whose instances are active objects.

 Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes.

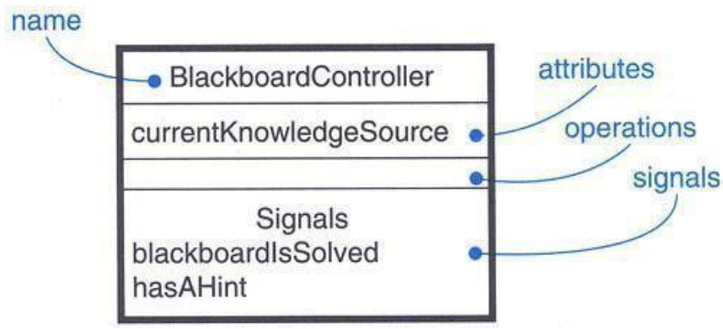


Figure: Active Class

Flow of Control

In a sequential system, there is a single flow of control. i.e, one thing, and one thing only, can take place at a time. *In a concurrent system*, there is multiple simultaneous flow of control i.e, more than one thing can take place at a time.

Classes and Events

- Active classes are just classes which represents an independent flow of control
- Active classes share the same properties as all other classes.
- When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated. *Two standard stereotypes* that apply to active classes are, <<***process***>> – Specifies a heavyweight flow that can execute concurrently with other processes. (heavyweight means, a thing known to the OS itself and runs in an independent address space) <<***thread***>> – Specifies a lightweight flow that can execute concurrently with other threads within the same process (lightweight means, known to the OS itself.)
- All the threads that live in the context of a process are peers of one another.

Communication

- In a system with both active and passive objects, there are *four possible combinations of interaction*
- *First*, a message may be passed from one passive object to another
- *Second*, a message may be passed from one active object to another
- In *inter-process communication* there are two possible styles of communication. *First*, one active object might synchronously call an operation of another. *Second*, one active object might asynchronously send a signal or call an operation of another object
- a synchronous message is rendered as a full arrow and an asynchronous message is rendered as a half arrow
- *Third*, a message may be passed from an active object to a passive object
- *Fourth*, a message may be passed from a passive object to an active one

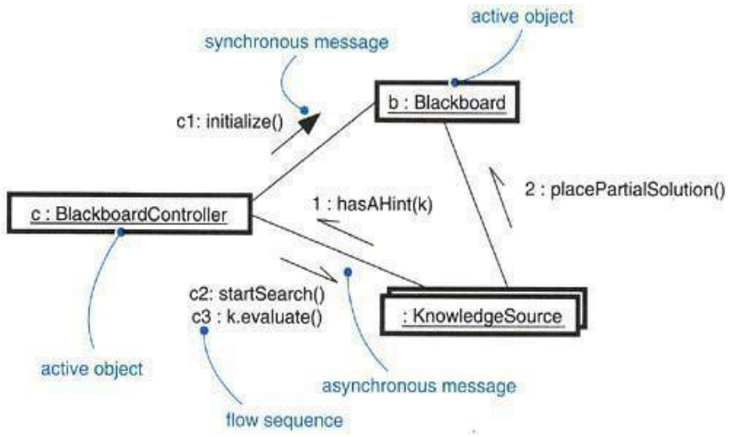


Figure: Communication

Synchronization

 Synchronization means arranging the flow of controls of objects so that mutual exclusion will be guaranteed.

 in object-oriented systems these objects are treated as a critical region

 three approaches are there to handle synchronization:

 Sequential – Callers must coordinate outside the object so that only one flow is in the object at a time

 Guarded – multiple flow of control is sequentialized with the help of object's guarded operations. in effect it becomes sequential.

 Concurrent – multiple flow of control is guaranteed by treating each operation as atomic

 synchronization are rendered in the operations of active classes with the help of constraints

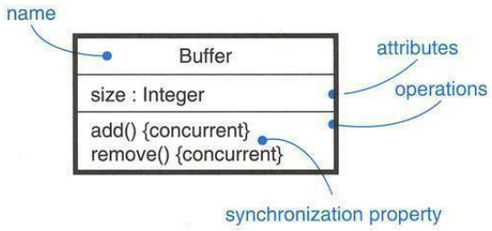


Figure: Synchronization

Process Views

 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.

Common Modeling Techniques Modeling Multiple Flows of Control

To model multiple flows of control,

 Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over engineer the process view of your system by introducing too much concurrency.

 Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that

each has the right set of attributes, operations, and signals.

 Capture these static decisions in class diagrams, explicitly highlighting each active class.

 Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.

 Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.

 Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

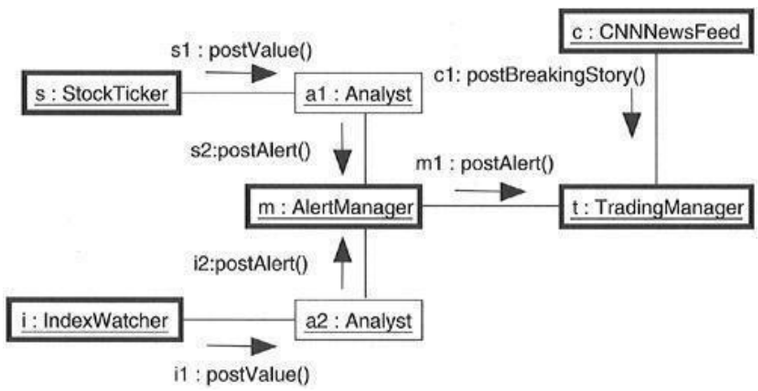


Figure: Modeling Flows of Control

Modeling InterProcess Communication

To model InterProcess communication,

 Model the multiple flows of control.

 Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.

 Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.

 Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

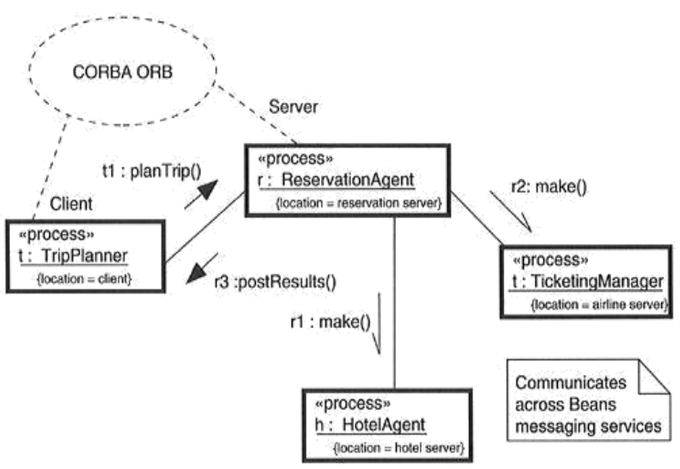


Figure: Modeling Inter process Communication

TIME AND SPACE

A distributed system is one in which components may be physically distributed across nodes. These nodes may represent different processors physically located in the same box, or they may

even represent computers that are located half a world away from one another.

To represent the modeling needs of real time and distributed systems, the UML provides a graphic representation for timing marks, time expressions, timing constraints, and location.

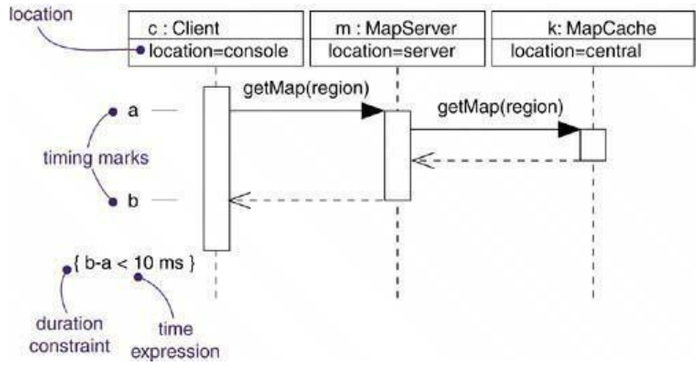


Figure: Timing Constraints and Location

A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is depicted as a small hash mark (horizontal line) on the border of a sequence diagram.

A *time expression* is an expression that evaluates to an absolute or relative value of time. A time expression can also be formed using the name of a message and an indication of a stage in its processing, for example, request.sendTime or request.receiveTime.

A *timing constraint* is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint-that is, a string enclosed by brackets and generally connected to an element by a dependency relationship.

Location is the placement of a component on a node. Location is an attribute of an object.

Time

Real time systems are, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

The passing of messages represents the dynamic aspect of any system, They are mainly rendered with the name of an event, such as a signal or a call.

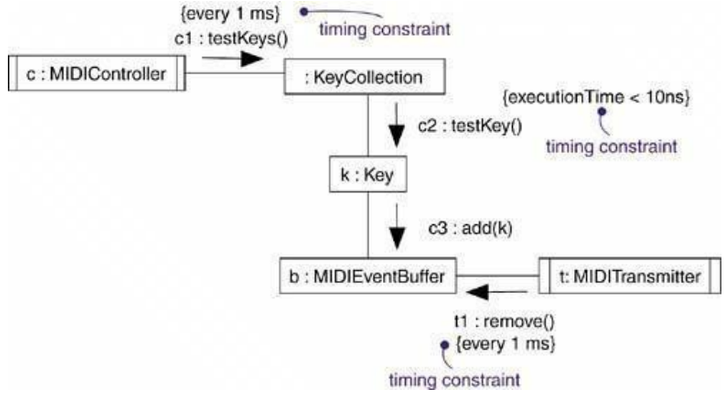


Figure: Time

Location

Distributed systems, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system;

in other systems, components may migrate from node to node.

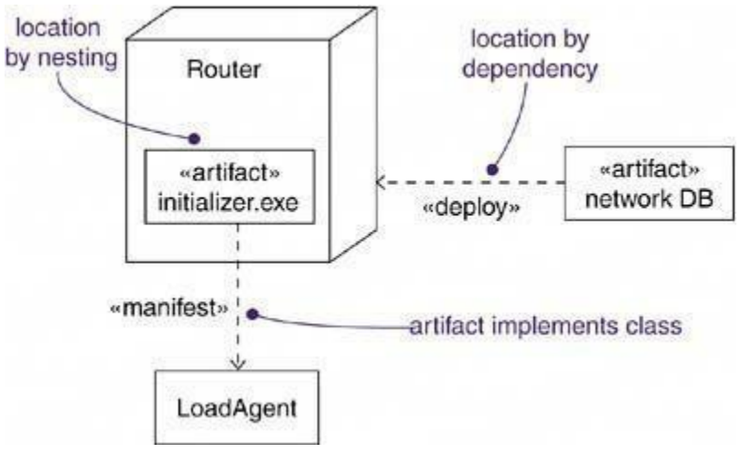


Figure: Location

Common Modeling Techniques *Modeling Timing Constraints*

To model timing constraints,

 For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.

 For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

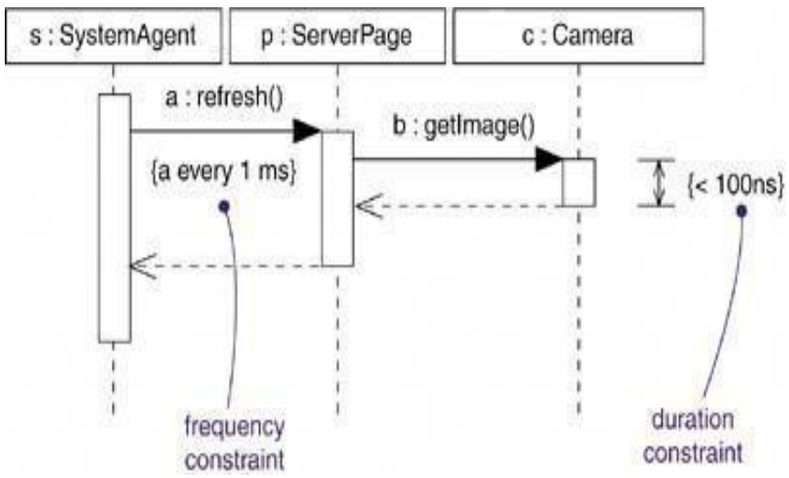


Figure: Modeling Timing Constraint

Modeling the Distribution of Objects

To model the distribution of objects,

 For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbours and their locations. A tightly coupled locality will have neighbouring objects close by; a loosely coupled one will have distant objects.

 Tentatively allocate objects closest to the actors that manipulate them.

 Next consider patterns of interaction among related sets of objects.

 Partition sets of objects that have low degrees of interaction.

 Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.

 Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.

 Assign objects to components so that tightly coupled objects are on the same component. Assign

 components to nodes so that the computation needs of each node are within capacity.

Add additional nodes if necessary.

 Balance performance and communication costs by assigning tightly coupled components to the same node.

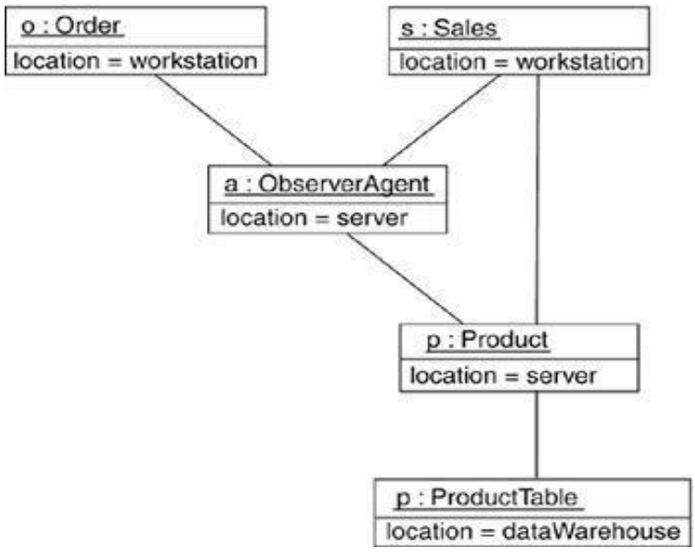


Figure: Modeling the Distribution of Objects

STATE CHART DIAGRAMS

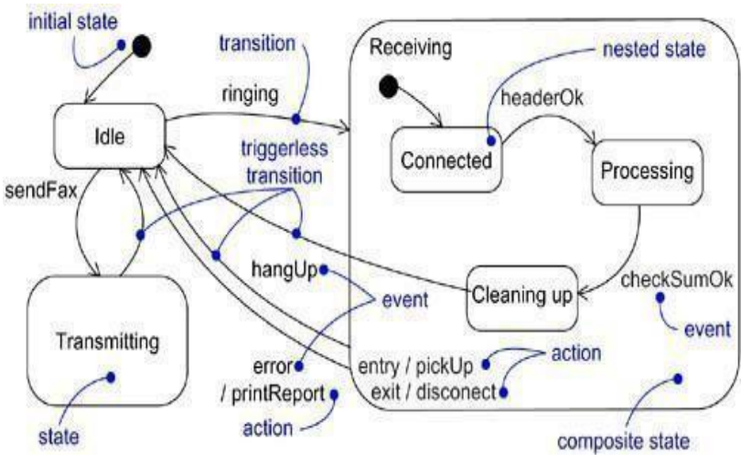


Figure: State chart Diagram

 State chart diagram is simply a presentation of a state machine which shows the flow of control from state to state.

 State chart diagrams are important for constructing executable systems through forward and reverse engineering.

 State chart diagrams are useful in modeling the lifetime of an object

 State chart diagrams commonly contain – Simple states and composite states, Transitions-including events and actions

 It is one of the five diagrams in UML for modeling the dynamic aspects of systems.

 Graphically, a state chart diagram is a collection of vertices and arcs.

A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. *An even*t in the context of state machines is an occurrence of a stimulus that can trigger a state transition. *A transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. *An activity* is ongoing non atomic execution within a state machine. An action is an executable atomic computation that results in a change in state of the model or the return of a value. *A reactive or event-driven object* is one whose behavior is best characterized by its response to events dispatched from outside its context

Modeling Reactive Objects

To model a reactive object,

 Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.

 Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.

 Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.

 Decide on the meaningful partial ordering of stable states over the lifetime of the object. Decide

 on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.

- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead end from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

The first string represents a tag; the second string represents the body of the message.

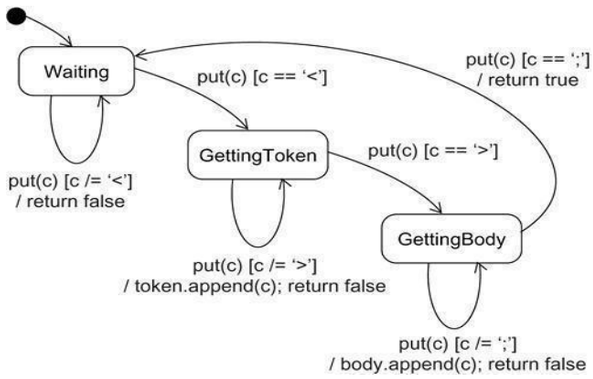


Figure: Modeling Reactive Object