Department of Computer Science and Engineering

Course File

for

Object Oriented Analysis & Design using UML

III/I Semister(R16)

Object Oriented Analysis and Design using UML

UNIT -1
SYLLABUS

- **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
- Evolution of Object Model
- Foundation of Object Model
- Elements of Object Model
- Applying the Object Model.

Chapter 1

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.

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 developers means more complex communication and hence more difficult coordination.

2. 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 on site 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.

3. 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 numbers 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.

ATTRIBUTES OFCOMPLEX SYSTEM

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 or 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.

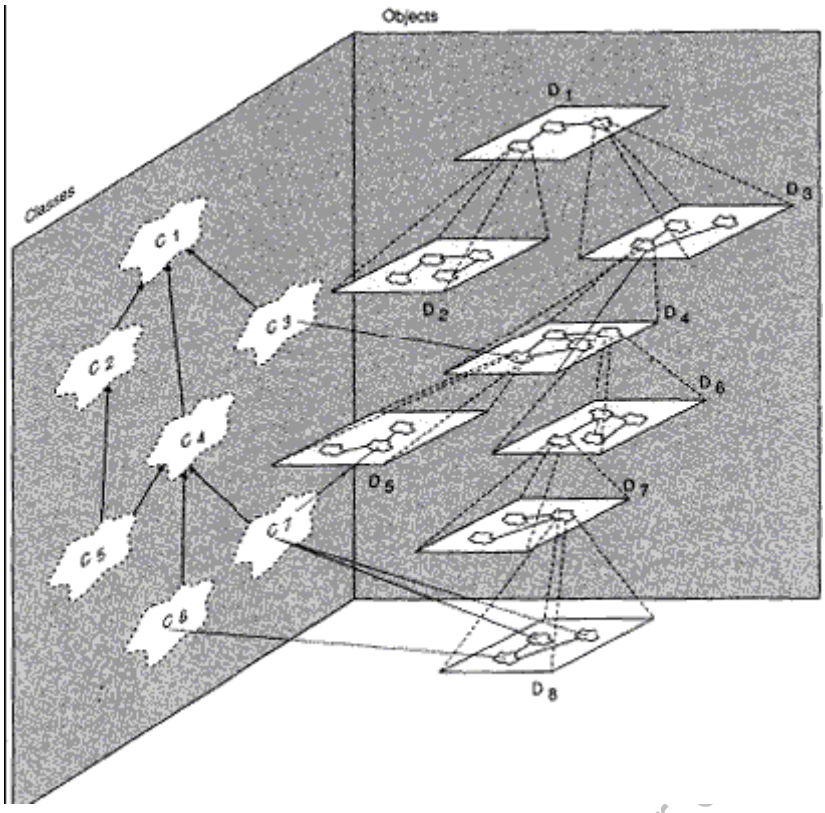


Figure 1.1 : Canonical form of a complex system

The figure 1.1 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 overall process.

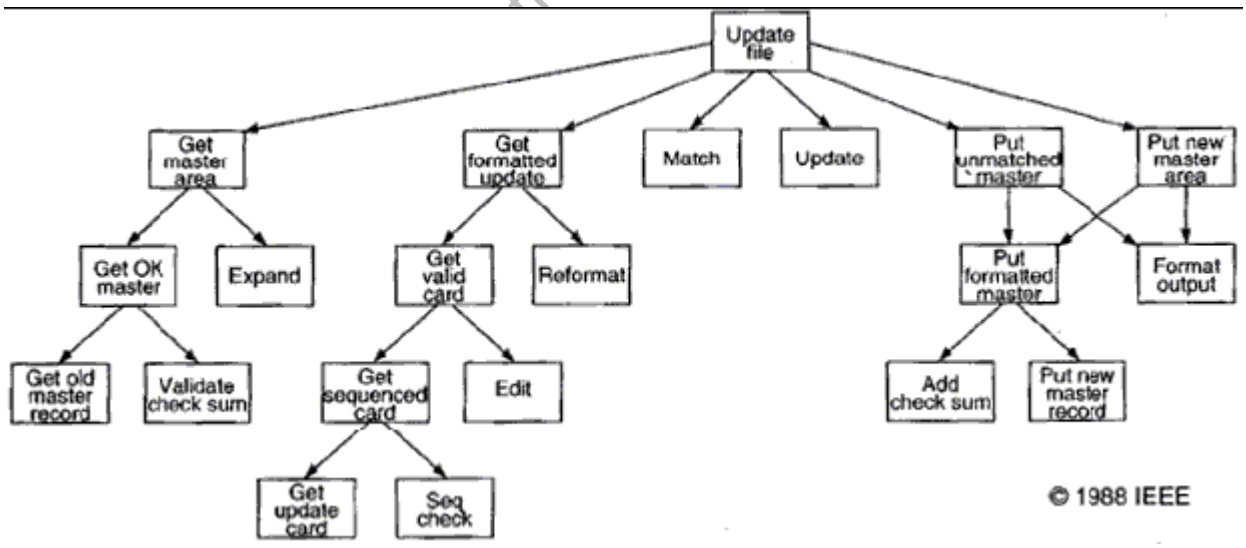


Figure 1.2: Algorithmic decomposition

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.

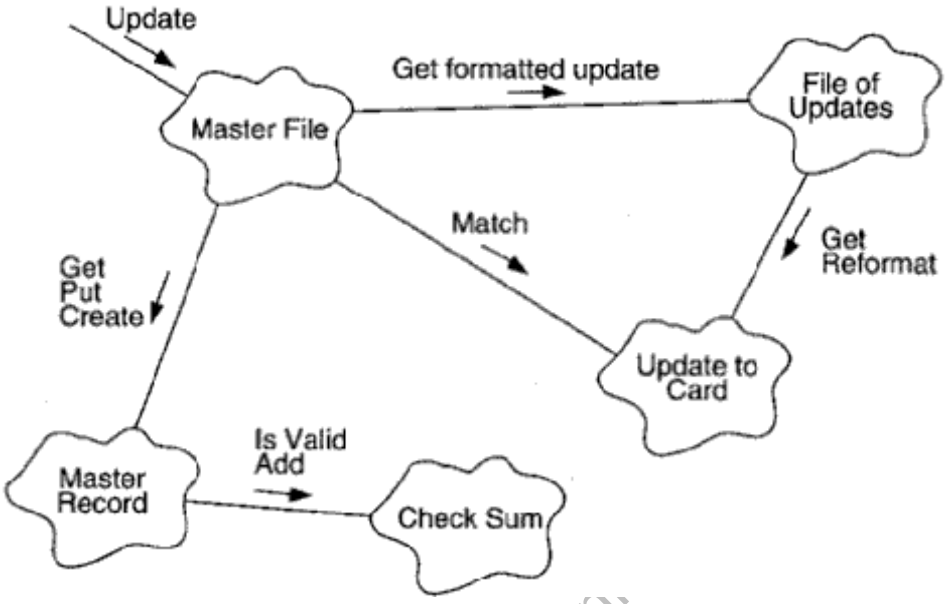


Figure 1.3: Object Oriented decomposition

Algorithmic versus object oriented decomposition: The algorithmic view highlighst 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 fordoing 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.

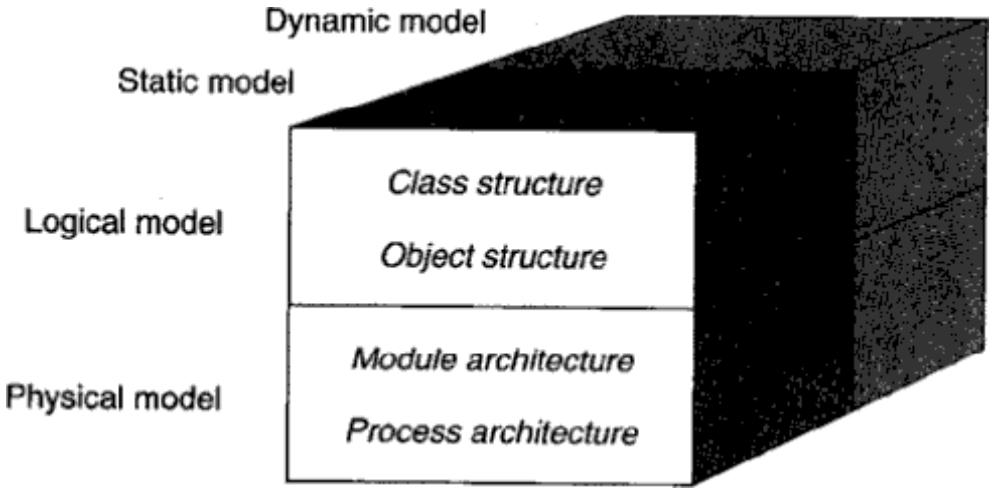


Figure 1.4: Models of object oriented development

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 Object Model

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.

The Evolution of the object Model

- The shift in focus from programming-in-the-small to programming-in-the-large.
- The evolution of high-order programming languages.
- New industrial strength software systems are larger and more complex than their predecessors.
- Development of more expressive programming languages advances the decomposition, abstraction and hierarchy.
- Wegner has classified some of more popular programming languages in generations according to the language features.

The generation of programming languages

1. First generation languages (1954 – 1958)
    - Used for specific & engineering application.
    - Generally consists of mathematical expressions.
    - For example: FORTRAN I, ALGOL 58, Flowmatic, IPLV etc.
2. Second generation languages (1959 – 1961)
    - Emphasized on algorithmic abstraction.
    - FORTRAN II - having features of subroutines, separate compilation
    - ALGOL 60 - having features of block structure, data type
    - COBOL - having features of data, descriptions, file handing
    - LISP - List processing, pointers, garbage collection
3. Third generation languages (1962 – 1970)
    - Supports data abstraction.
    - PL/1 – FORTRAN + ALGOL + COBOL
    - ALGOL 68 – Rigorous successor to ALGOL 60
    - Pascal – Simple successor to ALGOL 60
    - Simula - Classes, data abstraction
4. The generation gap (1970 – 1980)
    - C – Efficient, small executables
    - FORTRAN 77 – ANSI standardization
5. Object Oriented Boom (1980 – 1990)
    - Smalltalk 80 – Pure object oriented language
    - C++ - Derived from C and Simula
    - Ada83 – Strong typing; heavy Pascal influence
    - Eiffel - Derived from Ada and Simula
6. Emergence of Frameworks (1990 – today)

- Visual Basic – Eased development of the graphical user interface (GUI) for windows applications
- Java – Successor to Oak; designed for portability
- Python – Object oriented scripting language
- J2EE – Java based framework for enterprise computing
- .NET – Microsoft's object based framework
- Visual C# - Java competitor for the Microsoft .NET framework
- Visual Basic .NET – VB for Microsoft .NET framework

Topology of first and early second generation programming languages

- Topology means basic physical building blocks of the language & how those parts can be connected.
- Arrows indicate dependency of subprograms on various data.
- Error in one part of program effect across the rest of system.

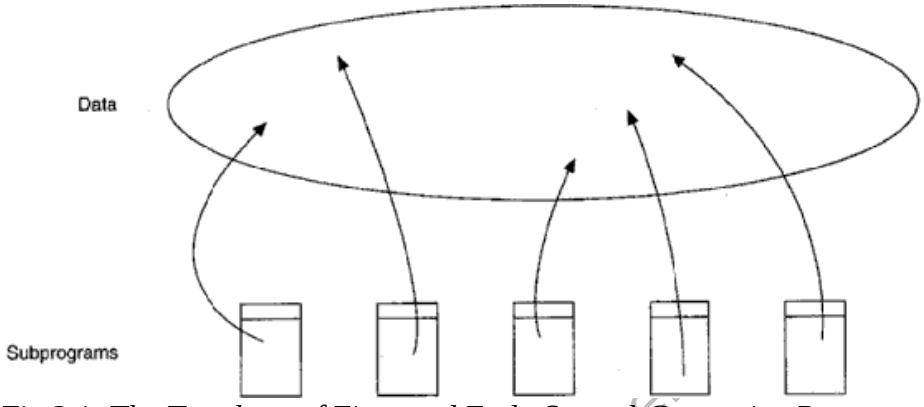


Fig 2.1: The Topology of First- and Early Second-Generation Programming Languages

Topology of late second and early third generation programming languages

Software abstraction becomes procedural abstraction; subprograms as an obstruction mechanism and three important consequences:

- Languages invented that supported parameter passing mechanism
- Foundations of structured programming were laid.
- Structured design method emerged using subprograms as basic physical blocks.

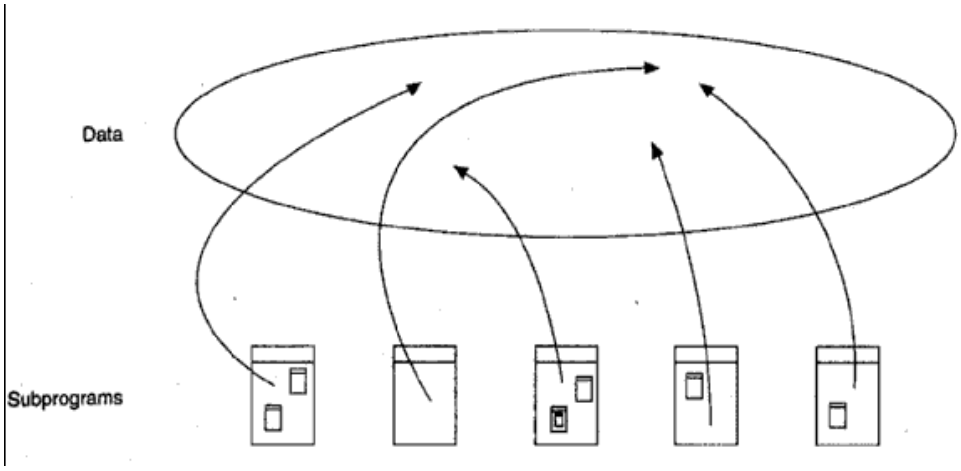


Fig 2.2: The Topology of Late Second- and Early Third-Generation Programming Languages

The topology of late third generation programming languages

- Larger project means larger team, so need to develop different parts of same program independently, i.e. complied module.
- Support modular structure.

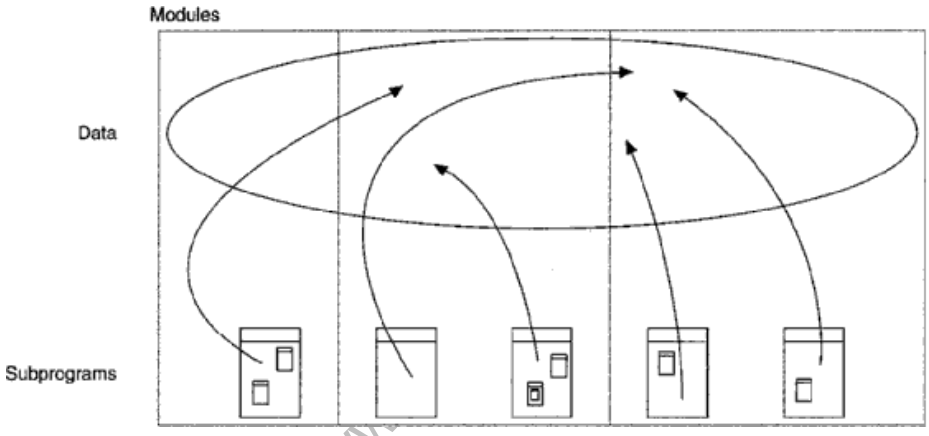


Fig 2.3: The Topology of Late Third-Generation Programming Languages

Topology of object and object oriented programming language

Two methods for complexity of problems

(i) Data driven design method emerged for data abstraction.

(ii) Theories regarding the concept of a type appeared

- Many languages such as Smalltalk, C++, Ada, Java were developed.
- Physical building block in these languages is module which represents logical collection of classes and objects instead of subprograms.
- Suppose procedures and functions are verbs and pieces of data are nouns, then
- Procedure oriented program is organized around verbs and object oriented program is organized around nouns.

Data and operations are united in such a way that the fundamental logical building blocks of our systems are no longer algorithms, but are classes and objects.

- In large application system, classes, objects and modules essential yet insufficient means of abstraction.

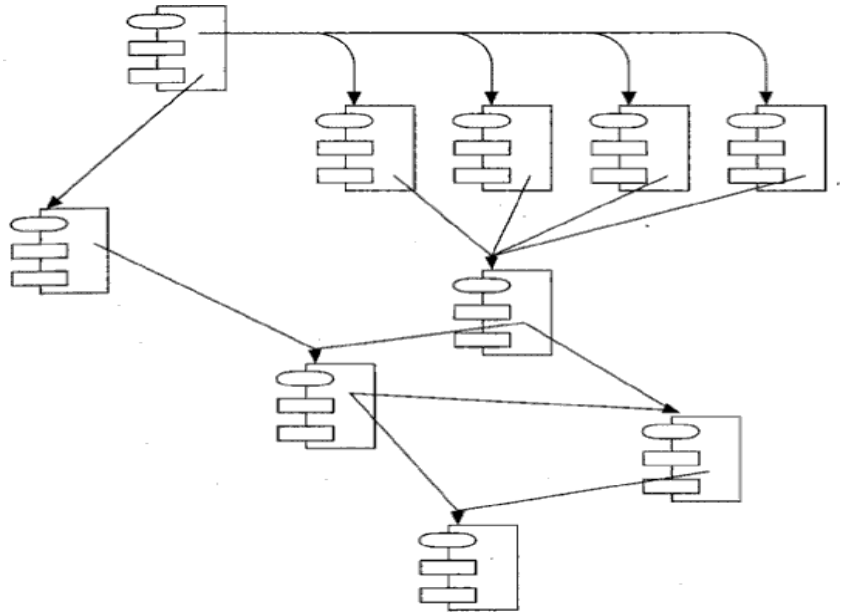


Fig 2.4: The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

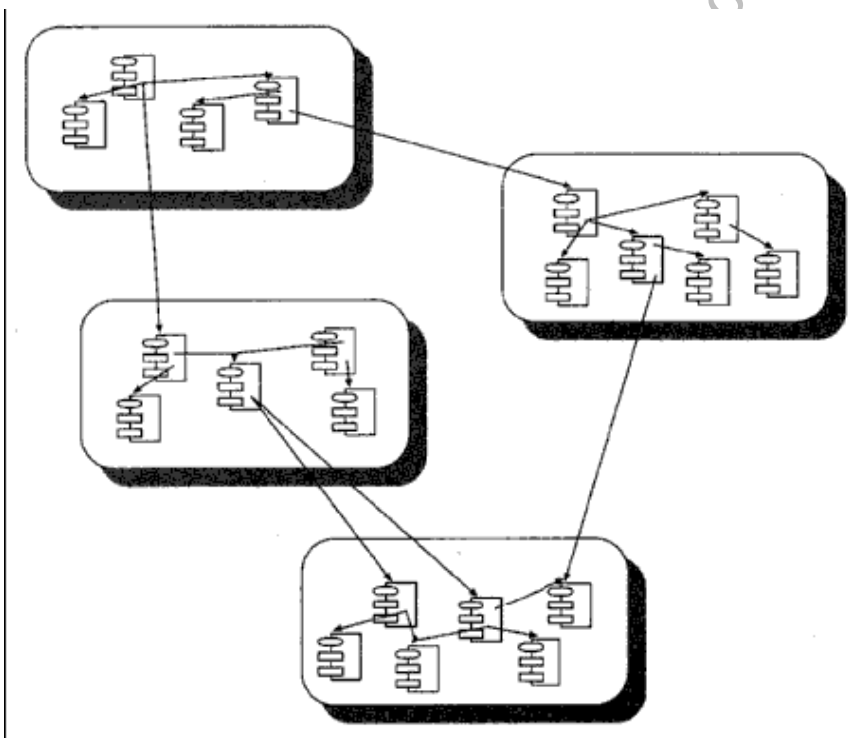


Fig 2.5: The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

Foundations of the object model

In structured design method, build complex system using algorithm as their fundamental building block. An object oriented programming language, class and object as basic building block.

Following events have contributed to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hardware support for operating systems concepts
- Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada.
- Advances in programming methodology, including modularization and information hiding.

We would add to this list three more contributions to the foundation of the object model:

- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

OOA (Object Oriented analysis)

During software requirement phase, requirement analysis and object analysis, it is a method of analysis that examines requirements from the perspective of classes and objects as related to problem domain. Object oriented analysis emphasizes the building of real-world model using the object oriented view of the world.

OOD (Object oriented design)

During user requirement phase, OOD involves understanding of the application domain and build an object model. Identify objects; it is methods of design showing process of object oriented decomposition. Object oriented design is a method of design encompassing the process of object oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

OOP (Object oriented programming)

During system implementation phase, t is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united in inheritance relationships. Object oriented programming satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have associated type (class).
- Classes may inherit attributes from supertype to subtype.

Elements of Object Model

Kinds of Programming Paradigms: According to Jenkins and Glasgow, most programmers work in one language and use only one programming style. They have not been exposed to alternate ways of thinking about a problem. Programming style is a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear.

There are five main kinds of programming styles:

1. Procedure oriented – Algorithms for design of computation
2. Object oriented – classes and objects
3. Logic oriented – Goals, often expressed in a predicate calculus
4. Rules oriented – If then rules for design of knowledge base
5. Constraint orient – Invariant relationships.

Each requires a different mindset, a different way of thinking about the problem. Object model is the conceptual frame work for all things of object oriented.

There are four **major elements** of object model. They are:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

There are three **minor elements** which are useful but not essential part of object model. Minor elements of object model are:

1. Typing
2. Concurrency
3. Persistence

Abstraction

Abstraction is defined as a simplified description or specification of a system that emphasizes some of the system details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, not so significant, immaterial.

An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries on the perspective of the viewer. An abstraction focuses on the outside view of an object, Abstraction focuses up on the essential characteristics of some object, relative to the perspective of the viewer. From the most to the least useful, these kinds of abstraction include following.

- Entity abstraction: An object that represents a useful model of a problem domain or solution domain entity.
- Action abstraction: An object that provides a generalized set of operations all of which program the same kind of function.
- Virtual machine abstractions: An object that groups together operations that are used by some superior level of control, or operations that all use some junior set of operations.
- Coincidental abstraction: An object that packages a set of operations that have no relation to each other.

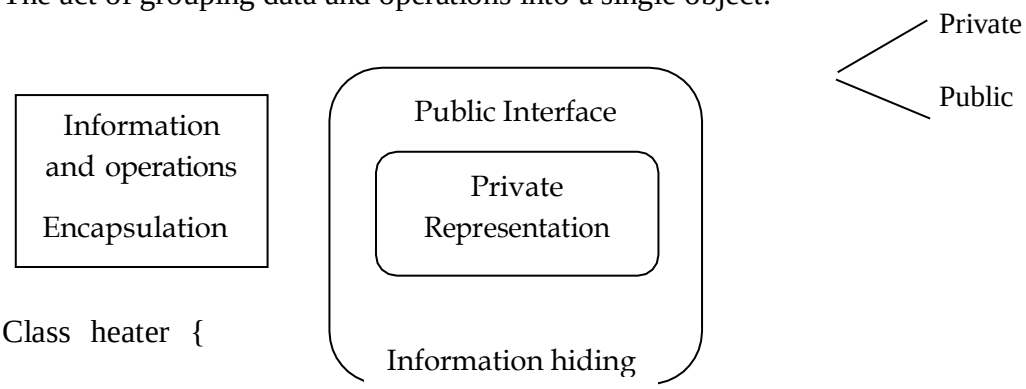
Abstraction: Temperature Sensor
Important Characteristics:
temperature location

Figure 2.6: Abstraction of a Temperature Sensor

Encapsulation

The act of grouping data and operations into a single object.

Clas



Class heater {

Public:

heater (location):

~ heater ( ):

void turnon ( );

void turnoff ( );

Boolean ison () const

private:

Abstraction: Heater
Important Characteristics: location
status

Figure 2.7: Abstraction of a Heater

Modularity

The act of partitioning a program into individual components is called modularity. It is reusable component which reduces complexity to some degree. Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program. In some languages, such as Smalltalk, there is no concept of a module, so the class forms the only physical unit of decomposition. Java has packages that contain classes. In many other languages, including Object Pascal, C++, and Ada, the module is a separate language construct and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system; we place these abstractions in modules to produce the system's physical architecture. Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- modules can be compiled separately. modules in C++ are nothing more than separately compiled files, generally called header files.
- Interface of module are files with .h extensions & implementations are placed in files with .c or .cpp suffix.
- Modules are units in pascal and package body specification in ada.
- modules Serve as physical containers in which classes and objects are declared like gates in IC of computer.
- Group logically related classes and objects in the same module.
- E.g. consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate their activities.
- A poor design is to define each message class in its own module; so difficult for users to find the classes they need. Sometimes modularization is worse than no modulation at all.
- Developer must balance: desire to encapsulate abstractions and need to make certain abstractions visible to other modules.
- Principles of abstraction, encapsulation and modularity are synergistic (having common effect)

Example of modularity

Let's look at modularity in the Hydroponics Gardening System. Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Since one

of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans (e.g., FruitGrowingPlan, GrainGrowingPlan). The implementations of these GrowingPlan classes would appear in the implementation of this module. We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

Hierarchy

Hierarchy is a ranking or ordering of abstractions Encapsulation hides company inside new of abstraction and modularity logically related abstraction & thus a set of abstractions form hierarchy. Hierarchies in complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Examples of Hierarchy: Single Inheritance

Inheritance defines a relationship among classes. Where one classes shares structure or behaviors defined in one (single in heritance) or more class (multiple inheritance) & thus represents a hierarchy of abstractions in which a subclass inherits from one or more super classes. Consider the different kinds of growing plans we might use in the Hydroponics Gardening System. An earlier section described our abstraction of a very generalized growing plan. Different kinds of crops, however, demand specialized growing plans. For example, the growing plan for all fruits is generally the same but is quite different from the plan for all vegetables, or for all floral crops. Because of this clustering of abstractions, it is reasonable to define a standard fruitgrowing plan that encapsulates the behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert that FruitGrowingPlan “is a” kind of GrowingPlan.

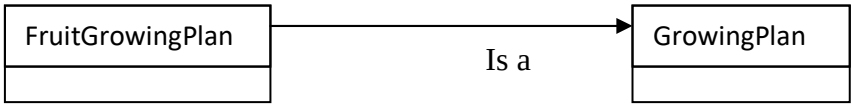


Fig 2.8: Class having one superclass (Single Inheritance)

In this case, FruitGrowingPlan is more specialized, and GrowingPlan is more general. The same could be said for GrainGrowingPlan or VegetableGrowingPlan, that is, GrainGrowingPlan “is a” kind of GrowingPlan, and VegetableGrowingPlan “is a” kind of GrowingPlan. Here, GrowingPlan is the more general superclass, and the others are specialized subclasses.

As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses. This is why we often speak of inheritance as being a generalization/specialization hierarchy. Superclasses represent generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden.

Examples of Hierarchy: Multiple Inheritance

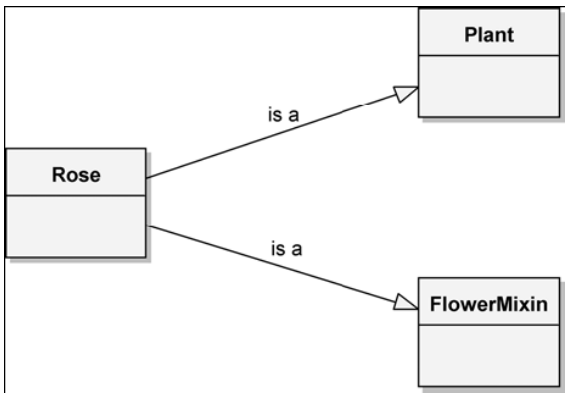


Figure 2.9: The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)

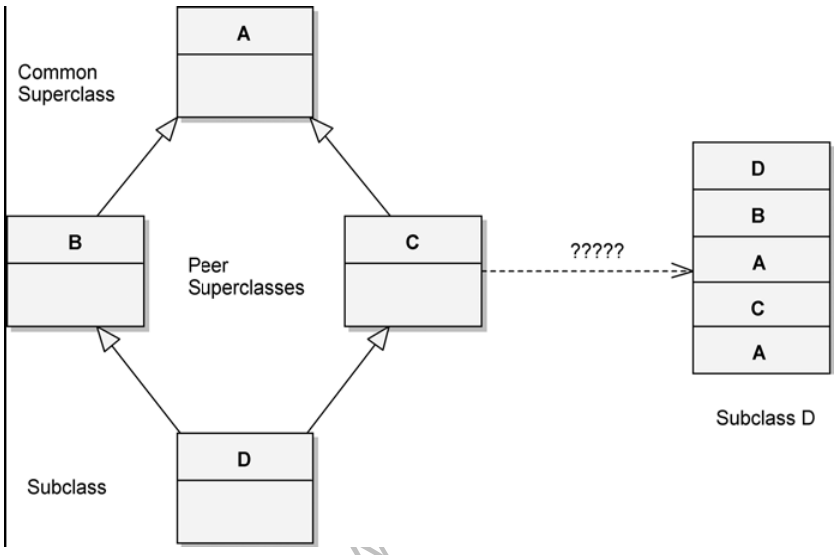


Figure 2.10: The Repeated Inheritance

Repeated inheritance occurs when two or more peer superclasses share a common superclass.

Hierarchy: Aggregation

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggregation relationships. For example, consider the abstraction of a garden. We can contend that a garden consists of a collection of plants together with a growing plan. In other words, plants are "part of" the garden, and the growing plan is "part of" the garden. This "part of" relationship is known as aggregation.

Aggregation raises the issue of ownership. Our abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted). In other words, the lifetime of a garden and its plants are independent. In contrast, we have decided that a GrowingPlan object is intrinsically associated with a Garden object and does not exist independently. Therefore, when we create an instance of Garden, we also create an instance of GrowingPlan; when we destroy the Garden object, we in turn destroy the GrowingPlan instance.

Typing

A type is a precise characterization of structural or behavioral properties which a collection of entities share. Type and class are used interchangeably class implements a type. Typing is the enforcement of the class of an object. Such that object of different types may not be interchanged. Typing implements abstractions to enforce design decisions. E.g. multiplying temp by a unit of force does not make serve but multiplying mass by force does. So this is strong typing. Example of strong and weak typing: In strong type, type conformance is strictly enforced. Operations can not be called upon an object unless the exact signature of that operation is defined in the object's class or super classes.

A given programming language may be strongly typed, weakly typed, or even untyped, yet still be called object-oriented. A strongly typed language is one in which all expressions defined in super class are guaranteed to be type consistent. When we divide distance by time, we expect some value denoting speed, not weight. Similarly, dividing a unit of force by temperature doesn't make sense, but dividing force by mass does. These are both examples of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions.

Benefits of Strongly typed languages:

- Without type checking, program can crash at run time
- Type declaration help to document program
- Most compilers can generate more efficient object code if types are declared.

Examples of Typing: Static and Dynamic Typing

Static typing (static binding/early binding) refers to the time when names are bound to types i.e. types of all variables are fixed at the time of compilation. Dynamic binding (late binding) means that types of all variables and expressions are not known until run time. Dynamic building (object pascal, C++) small talk (untyped).

Polymorphism is a condition that exists when the features of dynamic typing and inheritance interact. Polymorphism represents a concept in type theory in which a single name (such as a

variable declaration) may denote objects of many different classes that are related by some common superclass. The opposite of polymorphism is monomorphism, which is found in all languages that are both strongly and statically typed.

Concurrency

OO-programming focuses upon data abstraction, encapsulation and inheritance concurrency focuses upon process abstraction and synchronization. Each object may represent a separate thread of actual (a process abstraction). Such objects are called active. In a system based on an object oriented design, we can conceptualize the word as consisting of a set of cooperative objects, some of which are active (serve as centers of independent activity). Thus concurrency is the property that distinguishes an active object from one that is not active. For example: If two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of object being acted upon is not computed when both active objects try to update their state simultaneously. In the preserve of concurrency, it is not enough simply to define the methods are preserved in the presence of multiple thread of control.

Examples of Concurrency

Let's consider a sensor named ActiveTemperatureSensor, whose behavior requires periodically sensing the current temperature and then notifying the client whenever the temperature changes a certain number of degrees from a given setpoint. We do not explain how the class implements this behavior. That fact is a secret of the implementation, but it is clear that some form of concurrency is required.

There are three approaches to concurrency in object oriented design

- Concurrency is an intrinsic feature of languages. Concurrency is termed as task in ada, and class process in small talk. class process is used as super classes of all active objects. we may create an active object that runs some process concurrently with all other active objects.
- We may use a class library that implements some form of light weight process AT & T task library for C++ provides the classes' sched, Timer, Task and others. Concurrency appears through the use of these standard classes.
- Use of interrupts to give the illusion of concurrency use of hardware timer in active Temperature sensor periodically interrupts the application during which time all the sensor read the current temperature, then invoke their callback functions as necessary.

Persistence

Persistence is the property of an object through which its existence transcends time and or space objects continues to exist after its creator ceases to exist and/or the object's location moves from the address space in which it was created. An object in software takes up some amount of space and exists for a particular amount of time. Object persistence encompasses the followings.

- Transient results in expression evaluation
- Local variables in procedure activations

- Global variables where exists is different from their scope
- Data that exists between executions of a program

Data that exists between various versions of the program

- Data that outlines the Program.

Traditional Programming Languages usually address only the first three kind of object persistence. Persistence of last three kinds is typically the domain of database technology. Introducing the concept of persistence to the object model gives rise to object oriented databases. In practice, such databases build upon some database models (Hierarchical, network relational). Database queries and operations are completed through the programmer abstraction of an object oriented interface. Persistence deals with more than just the lifetime of data. In object oriented databases, not only does the state of an object persist, but its class must also transcend only individual program, so that every program interprets this saved state in the same way.

In most systems, an object once created, consumes the same physical memory until it classes to exist. However, for systems that execute upon a distributed set of processors, we must sometimes be concerned with persistence across space. In such systems, it is useful to think of objects that can move from space to space.

Applying the Object Model

Benefits of the Object Model: Object model introduces several new elements which are advantageous over traditional method of structured programming. The significant benefits are:

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilent to change.
- Object model appears to the working of human cognition, many people who have no idea how a computer works find the idea of object oriented systems quite natural.

Application of Object Model

OOA & Design may be in only method which can be employed to attack the complexity inherent in large systems. Some of the applications of the object model are as follows:

- Air traffic control
- Animation
- Business or insurance software
- Business Data Processing
- CAD
- Databases
- Expert Systems
- Office Automation
- Robotics
- Telecommunication
- Telemetry System etc.

UNIT 2

Classes and Objects

When we use object-oriented methods to analyze or design a complex software system, our basic building blocks are classes and objects.

An object is an abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both

- Objects have an internal state that is recorded in a set of attributes.
- Objects have a behavior that is expressed in terms of operations. The execution of operations changes the state of the object and/or stimulates the execution of operations in other objects.
- Objects (at least in the analysis phase) have an origin in a real world entity.

Classes represent groups of objects which have the same behavior and information structures.

- Every object is an instance of a single class
- Class is a kind of type, an ADT (but with data), or an 'entity' (but with methods)
- Classes are the same in both analysis and design
- A class defines the possible behaviors and the information structure of all its object instances.

The nature of an object

The ability to recognize physical objects is a skill that humans learn at a very early age. From the perspective of human, cognition, an object is any of the following.

- A tangible and/or visible thing.
- Something that may be apprehended intellectually.
- Something toward which thought or action is directed.

Informally, object is defined as a tangible entity that exhibits some well defined behavior. During software development, some objects such as inventions of design process whose collaborations with other such objects serve as the mechanisms that provide some higher level behavior more precisely.

An object represents an individual, identifiable item, until or entity either real or abstract, with a well defined role in the problem domain. E.g. of manufacturing plant for making airplane wings, bicycle frames etc. A chemical process in a manufacturing plant may be treated as an object; because it has a crisp conceptual boundary interacts with certain other objects through a well defined behavior. Time, beauty or colors are not objects but they are properties of other objects. We say that mother (an object) loves her children (another object).

An object has state, behavior and identify; the structure and behavior similar objects are defined an their common class, the terms instance and object are defined in their common class, the terms instance and object are interchangeable.

State

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when someone puts money in a slot and pushes a button to make a selection, a drink emerges

from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing because the user has violated the basic assumptions of their operation. Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says, "Correct change only," and puts in extra money. Most machines are user-hostile; they will happily swallow the excess money.

A property is a distinctive characteristic that contributes to making an object uniquely that object properties are usually static because attributes such as these are unchanging and fundamental to the nature of an object. Properties have some value. The value may be a simple quantity or it might denote another object. The fact that every object has static implies that every object has state implies that every object takes up some amount of space be it in the physical world or in computer memory.

We may say that all objects within a system encapsulate some state and that all of the state within a system is encapsulated by objects. Encapsulating the state of an object is a start, but it is not enough to allow us to capture the full intent of the abstractions we discover and invent during development

e.g. consider the structure of a personnel record in C++ as follows

```
struct personnelRecord{
char     name[100];
int      socialsecurityNumber;
char     department[10];
float    salary;
};
```

This denotes a class. Objects are as personnel Record Tom, Kaitlyn etc are all 2 distinct objects each of which takes space in memory. Own state in memory class can be declared as follows.

```
Class   personnelrecord{
public: char*employeename()const;
 int     SSN()  const;
 char*  empdept        const;
 protected:
char     name[100];
int      SSN;
 char    department[10];
 float   salary;
};
```

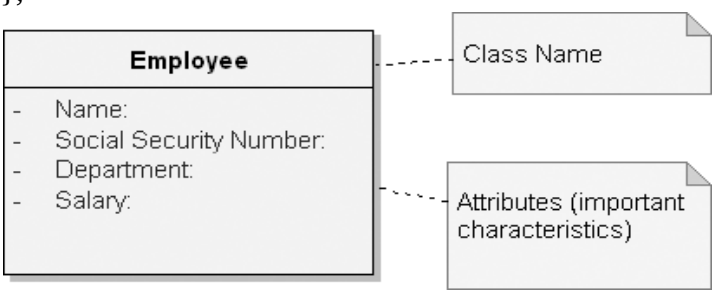



Figure 3–1 Employee Class with Attributes

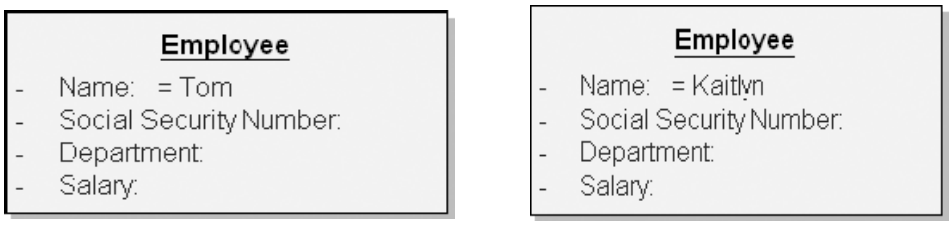


Figure 3–2 Employee Objects Tom and Kaitlyn

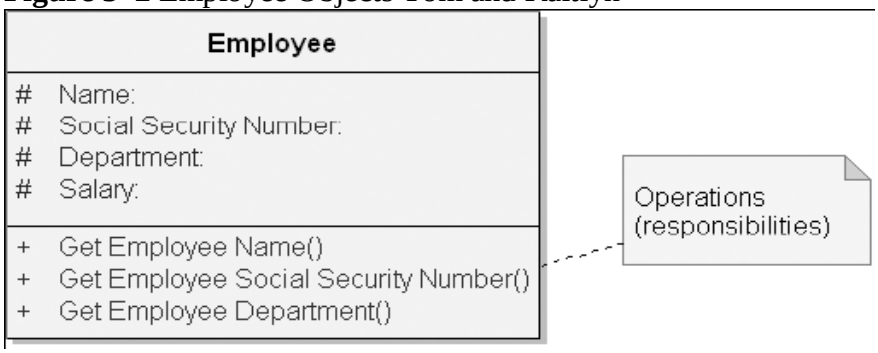


Figure 3–3 Employee Class with Protected Attributes and Public Operations

Class representation is hidden from all other outside clients. Changing class representation will not break outside source code. All clients have the right to retrieve the name, social security No and department of an employee. Only special clients (subclass) have permission to modify the values of these properties as well as salary. Thus, all objects within a system encapsulate some state.

Behavior

Behavior is how an object acts and reacts, in terms of its state changeable state of object affect its behavior. In vending machine, if we don't deposit change sufficient for our selection, then the machine will probably do nothing. So behavior of an object is a function of its state as well as the operation performed upon it. The state of an object represents the cumulative results of its behavior.

e.g. consider the declaration of queue in C++

```
Class Queue{
public:
Queue();
Queue(constQueue);
virtual ~Queue();
virtual Queue&operator = (ConstQueue);
Virtual int operator == (constQueue&)const;
int operator = (constQueue)const;
virtual voidclear();
Virtual voidappend(constvoid*);
virtual voidPOP();
virtual void remove (int at);
virtual int length();
virtual int isempty ( ) const;
```

```
virtual const void * Front ( ) const;
virtual int location (const void*);
protected..
};

queue a, b;
a. append (& Tom);
a.append (& Kaitlyn);
b = a;
a. pop( );
```

Operations

An operation denotes a service that a class offers to its clients. A client performs 5 kinds of operations upon an object.

- Modifier: An operation that alters the state of an object.
- Selector: An operation that accesses the state of an object but does not alter the state.
- Iterator: An operation that permits all parts of an object to be accessed in some well defined order. In queue example operations, clear, append, pop, remove) are modifies, const functions (length, is empty, front location) are selectors.
- Constructor: An operation that creates an object and/or initializes its state.
- Destructor: An operation that frees the state of an object and/or destroys the object itself.

Identity

Identity is that property of an object which distinguishes it from all other objects.
Consider the following declarations in C++.

```
struct point {
int x;
int y;
point ( ) : x (0), y (0){}
 point (int x value, int y value) : x (x value), (y value) {}
};

Next we provide a class that denotes a display items as follows.
Class DisplayItem{
Public: DisplayItem ();
displayitem (const point & location);
 virtual ~ displayitem ().
Virtual void draw();
Virtual void erase();
Virtual void select();
Virtual void Unselect ( );
virtual void move (const point & location);
int isselected ();
virtual void unselect ();
 virtual void move (const point & location);
```

```
 virtual void point location ( ) const;
int isunder (const point & location) const;
Protected …….
};
```

To declare instances of this class:

```
displayItem  item1;
item2 = new displayItem (point (75, 75);
Display item * item 3 = new Display Item (point (100, 100)) ;
display item * item 4 = 0
```

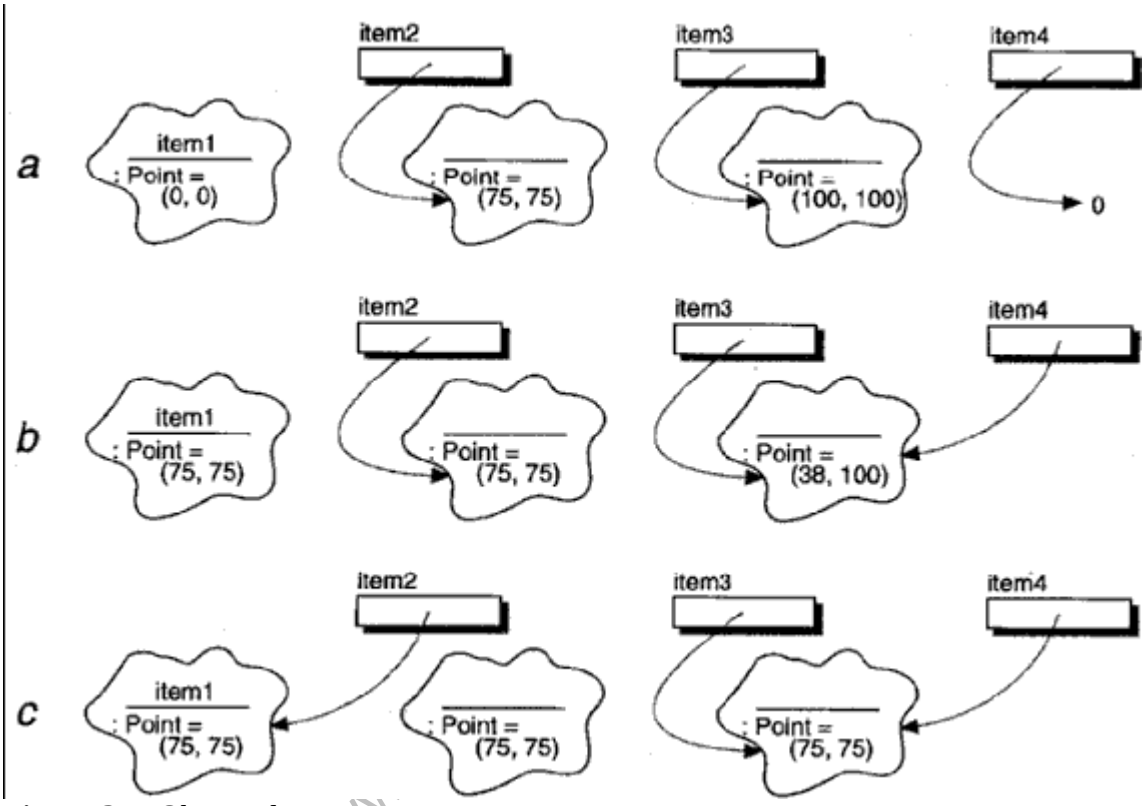


Figure 3–4 Object Identity

First declaration creates four names and 3 distinct objects in 4 diff location. Item 1 is the name of a distinct display item object and other 3 names denote a pointer to a display item objects. Item 4 is no such objects, we properly say that item 2 points to a distinct display item object, whose name we may properly refer to indirectly as * item2. The unique identity (but not necessarily the name) of each object in preserved over the lifetime of the object, even when its state is changed. Copying, Assignment, and Equality Structural sharing takes place when the identity of an object is aliased to a second name.

Object life span

The lifeline of an object extends from the time it is first created (and this first consumes space) until that space is recalled, whose purpose is to allocate space for this object and establish an

initial stable state. Often objects are created implicitly in C++ programming an object by value creates a new objection the stack that is a copy of the actual parameters.

In languages such as smalltalk, an object is destroyed automatically as part of garbage collection when all references to it have been lost. In C++, objects continuous exist and consume space even if all references to it are lost. Objects created on the stack are implicitly destroyed wherever control panels beyond the block in which the object can declared. Objects created with new operator must be destroyed with the delete operator. In C++ wherever an object is destroyed either implicitly or explicitly, its destructor is automatically involved, whose purpose is to declared space assigned to the object and its part.

Roles and Responsibilities

A role is a mask that an object wears and so defines a contract between an abstraction and its clients.

Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports.

In other words, we may say that the state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction's responsibilities.

Most interesting objects play many different roles during their lifetime such as:

- A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money. However, to a taxing authority, the account may play the role of an entity whose dividends must be reported on annually.

Objects as Machines

The existence of state within an object means that the order in which operations are invoked is important. This gives rise to the idea that each object is like a tiny, independent machine. Continuing the machine metaphor, we may classify objects as either active or passive. An active object is one that encompasses its own thread of control, whereas a passive object does not. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated on by another object. Passive objects, on the other hand, can undergo a state change only when explicitly acted on. In this manner, the active objects in our system serve as the roots of control. If our system involves multiple threads of control, we will usually have multiple active objects. Sequential systems, on the other hand, usually have exactly one active object, such as a main object responsible for managing an event loop that dispatches messages. In such architectures, all other objects are passive, and their behavior is ultimately triggered by messages from the one active object. In other kinds of sequential system architectures (such as transaction-processing systems), there is no obvious central active object, so control tends to be distributed throughout the system's passive objects.

Relationship among Objects

Objects contribute to the behavior of a system by collaborating with one another. E.g. object structure of an airplane. The relationship between any two objects encompasses the assumptions that each makes about the other including what operations can be performed. Two kinds of objects relationships are links and aggregation.

Links

A link denotes the specific association through which one object (the client) applies the services of another object (the supplier) or through which are object may navigate to another. A line between two object icons represents the existence of pass along this path. Messages are shown as

directed lines representing the direction of message passing between two objects is typically unidirectional, may be bidirectional data flow in either direction across a link.
As a participation in a link, an object may play one of three roles:

- Controller: This object can operate on other objects but is not operated on by other objects. In some contexts, the terms active object and controller are interchangeable.
- Server: This object doesn't operate on other objects; it is only operated on by other objects.
- Proxy: This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.

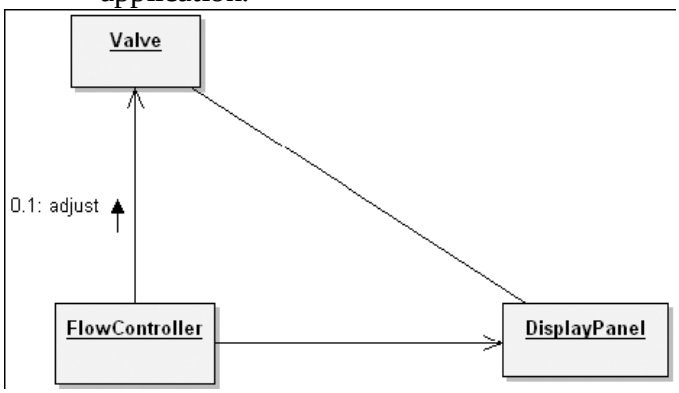


Figure 3–5 Links
In the above figure, FlowController acts as a controller object, DisplayPanel acts as a server object, and Valve acts as a proxy.

Visibility
Consider two objects, A and B, with a link between the two. In order for A to send a message to object B, B must be visible to A. Four ways of visibility

- The supplier object is global to the client
- The supplier object is a programmer to some operation of the client
- The supplier object is a part of the client object.
- The supplier object is locally declared object in some operation of the client.

Synchronization
Wherever one object passes a message to another across a link, the two objects are said to be synchronized. Active objects embody their own thread of control, so we expect their semantics to be guaranteed in the presence of other active objects. When one active object has a link to a passive one, we must choose one of three approaches to synchronization.

1. Sequential: The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
2. Guarded: The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.
3. Concurrent: The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

Aggregation

Whereas links denote peer to peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the aggregate) to its parts. Aggregation is a specialized kind of association. Aggregation may or may not denote physical containment. E.g. airplane is composed of wings, landing gear, and so on. This is a case of physical containment. The relationship between a shareholder and her shares is an aggregation relationship that doesn't require physical containment.

There are clear trade-offs between links and aggregation. Aggregation is sometimes better because it encapsulates parts as secrets of the whole. Links are sometimes better because they permit looser coupling among objects.

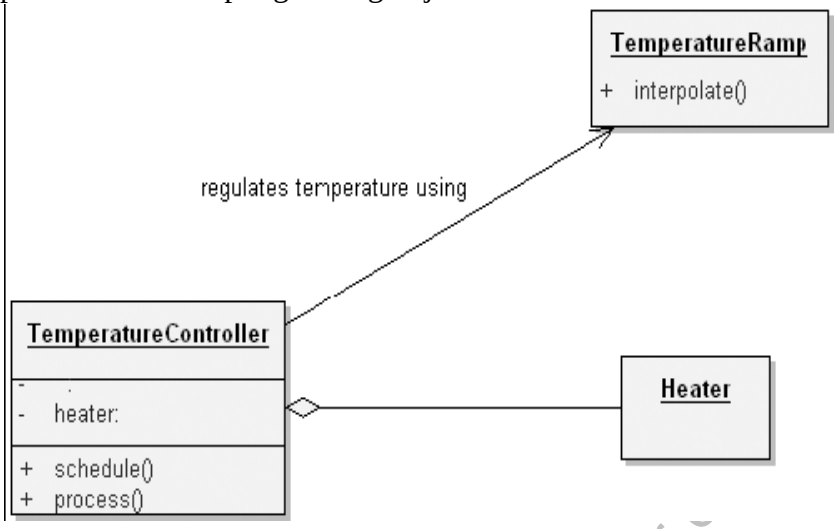


Figure 3–6 Aggregation

The Nature of the class

A class is a set of objects that share a common structure, common behavior and common semantics. A single object is simply an instance of a class. Object is a concrete entity that exists in time and space but class represents only an abstraction. A class may be an object is not a class.

Interface and Implementation: The interface of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and secrets of its behavior. The interface primarily consists of the declarations of all the operators applicable to instance of this class, but it may also include the declaration of other classes, constants variables and exceptions as needed to complete the abstraction. The implementation of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class consists of the class. Interface of the class is divided into following four parts.

- **Public**: a declaration that is accessible to all clients
- **Protected**: a declaration that is accessible only to the class itself and its subclasses
- **Private:** a declaration that is accessible only to the class itself
- **Package:** a declaration that is accessible only by classes in the same package

Relationship among Classes

We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some kind of sharing. Second, a class relationship might indicate some kind of semantic connection.

There are three basic kinds of class relationships.

- The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower.
- The second is whole/part, which denotes a "part of" relationship. A petal is not a kind of a flower; it is a part of a flower.
- The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers. As another example, roses and candles are largely independent classes, but they both represent things that we might use to decorate a dinner table.

Association

Of the different kinds of class relationships, associations are the most general. The identification of associations among classes is describing how many classes/objects are taking part in the relationship. As an example for a vehicle, two of our key abstractions include the vehicle and wheels. As shown in Figure 3–7, we may show a simple association between these two classes: the class Wheel and the class Vehicle.

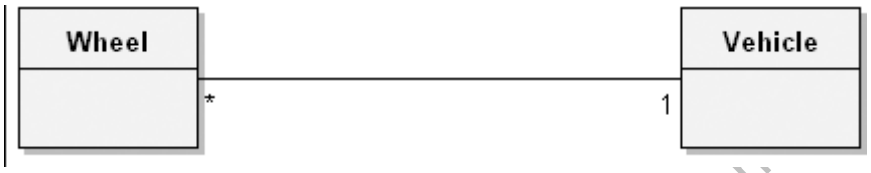


Figure 3–7 Association

Multiplicity/Cardinality

This multiplicity denotes the cardinality of the association. There are three common kinds of multiplicity across an association:

1. One-to-one
2. One-to-many
3. Many-to-many

Inheritance

Inheritance, perhaps the most semantically interesting of the concrete relationships, exists to express generalization/specialization relationships. Inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance means that subclasses inherit the structure of their superclass.

Space probe (spacecraft without people) report back to ground stations with information regarding states of important subsystems (such as electrical power & population systems) and different sensors (such as radiation sensors, mass spectrometers, cameras, detectors etc), such relayed information is called telemetry data. We can take an example for Telemetry Data for our illustration.

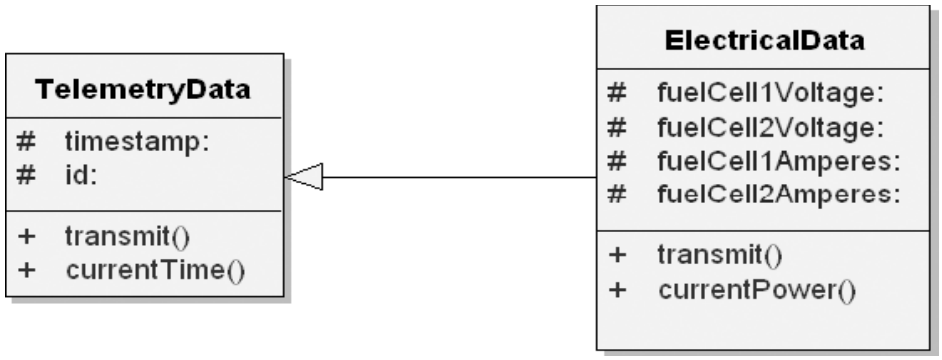


Figure 3–8 ElectricalData Inherits from the Superclass TelemetryData

As for the class ElectricalData, this class inherits the structure and behavior of the class TelemetryData but adds to its structure (the additional voltage data), redefines its behavior (the function transmit) to transmit the additional data, and can even add to its behavior (the function currentPower, a function to provide the current power level).

Single Inheritence

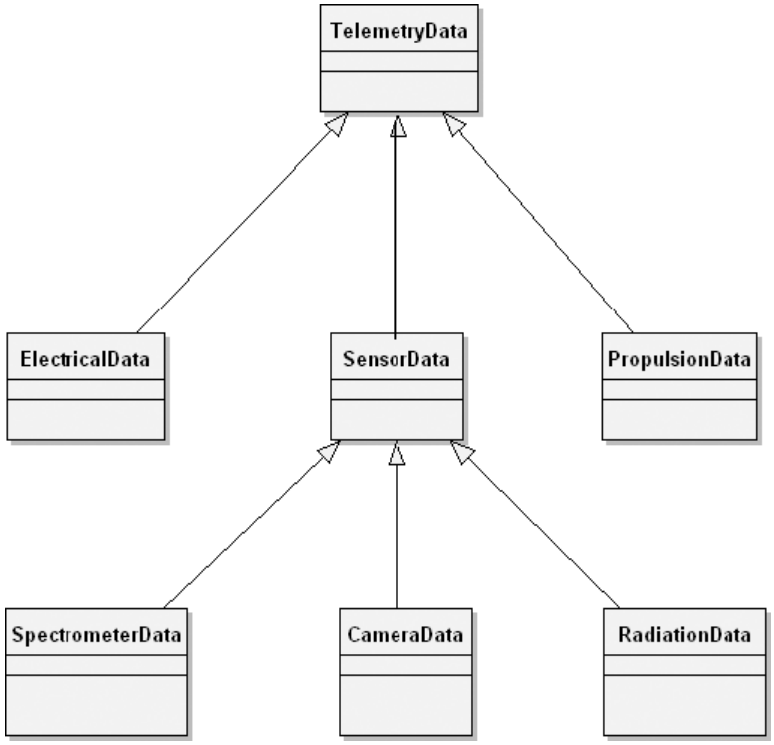


Figure 3–9 Single Inheritance

Figure 3–9 illustrates the single inheritance relationships deriving from the superclass TelemetryData. Each directed line denotes an "is a" relationship. For example, CameraData "is a" kind of SensorData, which in turn "is a" kind of TelemetryData.

Multiple Inheritence

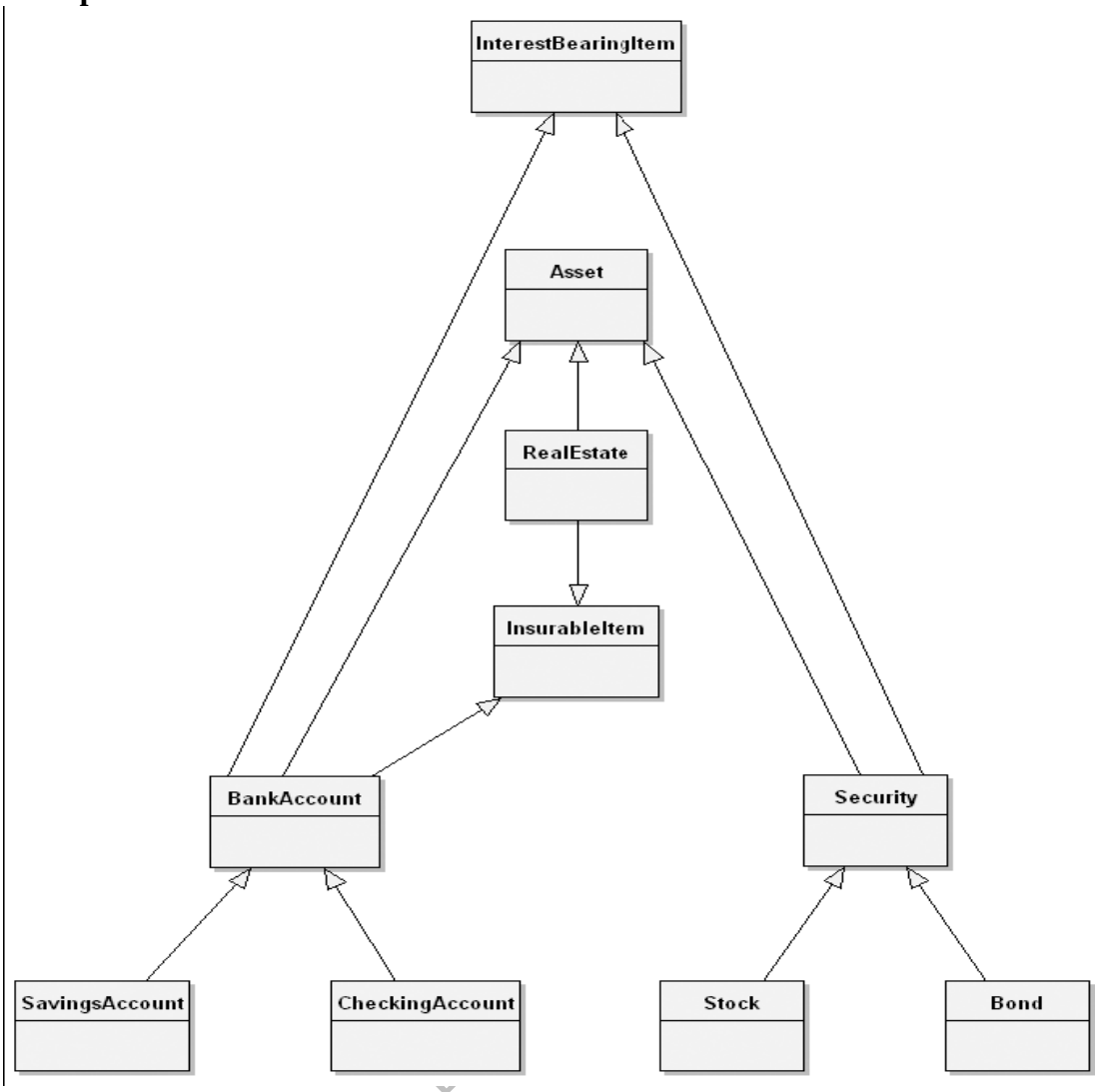


Figure 3–10 Multiple Inheritance

Consider for a moment how one might organize various assets such as savings accounts, real estate, stocks, and bonds. Savings accounts and checking accounts are both kinds of assets typically managed by a bank, so we might classify both of them as kinds of bank accounts, which in turn are kinds of assets. Stocks and bonds are managed quite differently than bank accounts, so we might classify stocks, bonds, mutual funds, and the like as kinds of securities, which in turn are also kinds of assets.

Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance. Figure 3–10 illustrates such a class structure. Here we see that the class Security is a kind of Asset as well as a kind of InterestBearingItem. Similarly, the class

BankAccount is a kind of Asset, as well as a kind of InsurableItem and InterestBearingItem.

Polymorphism

Polymorphism is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways. With polymorphism, an operation can be implemented differently by the classes in the hierarchy.

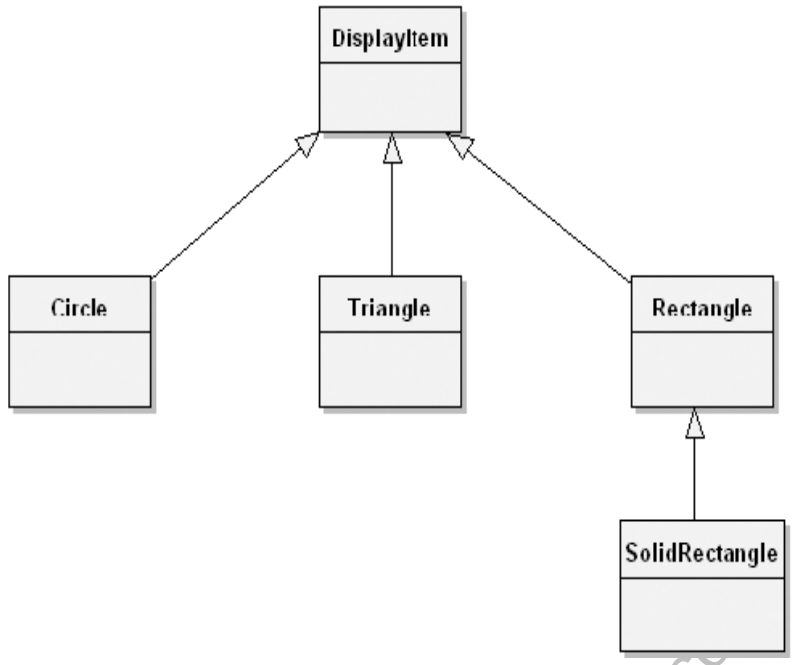


Figure 3-11 Polymorphism

Consider the class hierarchy in Figure 3–11, which shows the base class DisplayItem along with three subclasses named Circle, Triangle, and Rectangle. Rectangle also has one subclass, named SolidRectangle. In the class DisplayItem, suppose that we define the instance variable theCenter (denoting the coordinates for the center of the displayed item), along with the following operations:

- draw: Draw the item.
- move: Move the item.
- location: Return the location of the item.

The operation location is common to all subclasses and therefore need not be redefined, but we expect the operations draw and move to be redefined since only the subclasses know how to draw and move themselves.

Aggregation

We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes. As shown in Figure 3–12, the class TemperatureController denotes the whole, and the class Heater is one of its parts.

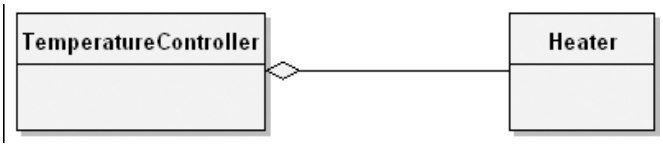


Figure 3–12 Aggregation

Physical Containment

In the case of the class TemperatureController, we have aggregation as containment by value, a kind of physical containment meaning that the Heater object does not exist independently of its enclosing TemperatureController instance. Rather, the lifetimes of these two objects are intimately connected: When we create an instance of TemperatureController, we also create an instance of the class Heater. When we destroy our TemperatureController object, by implication we also destroy the corresponding Heater object.

Using

Using shows a relationship between classes in which one class uses certain services of another class in a variety of ways. "Using" relationship is equivalent to an association, although the reverse is not necessarily true.

Clients and Suppliers

"Using" relationships among classes parallel the peer-to-peer links among the corresponding instances of these classes. Whereas an association denotes a bidirectional semantic connection, a "using" relationship is one possible refinement of an association, whereby we assert which abstraction is the client and which is the supplier of certain services.

Instantiation

The process of creating a new object (or instance of a class) is often referred to as instantiation.

Genericity

The possibility for a language to provided parameterized modules or types. E.g. List (of: Integer) or List (of: People). There are four basic ways of genericity

- Use of Macros – in earlier versions of C++, does not work well except on a small scale.
- Building heterogenous container class: used by small task and rely upon instance of some distant base class.
- By building generalized container classes as in small task, but then using explicity type checking code to enforce the convention that the contents are all of the same clam, which is asserted when the container object is created used in object Pascal, which are strongly typed support heritance but don't support any form of parameterized class.
- Using parameterized class (Also known as generic class) is one that serves as a temperature for other classes & template that may be parameterized by other classes, objects and or operations. A parameterized class must be instantiated (i.e. parameters must be filled in) before objects can be created.

Metaclass

Metaclass is a class whose instances are themselves classes. Small task and CLOS support the concept of a metaclass directly, C++ does not. A class provides an interface for the programmer to interface with the definition of objects. Programmers can easily manipulate the class.

Metaclass is used to provide class variables (which are shared by all instances of the class) and operations for initializing class variables and for creating the metaclass's single instance.

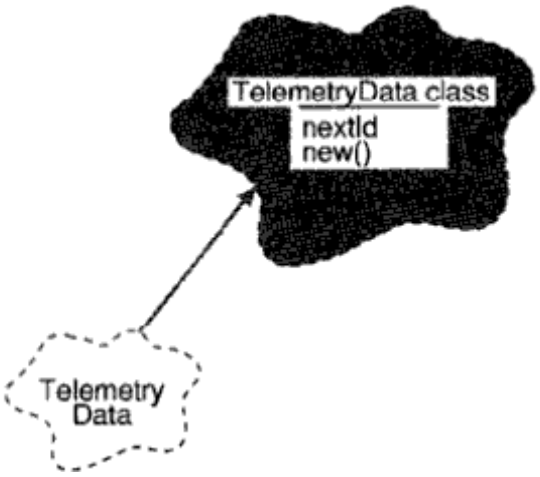


Figure 3-13 Metaclass

As shown in figure 3-13, a class variable next ID for the metaclass of telemetry data can be defined in order to assist in generating district ID's up on the creation of each instance of telemetry data. Similarly, an operation can be defined for creating new instances of the class, which perhaps generates them from some pre-allocated pool of storage. In C++, and destructors serve the purpose of metaclass creation operations. Member function and member objects as static in C++ are shared by all instances of class in C++. Static member's objects and static member function of C++ are equivalent to small task's metaclass operations.

Importance of proper classification

Classification is the means where by we order knowledge. There is no any golden path to classification. Classification and object oriented development: The identification of classes and objects is the hardest part of object oriented analysis and design, identification involves both discovery and invention. Discovery helps to recognize the key abstractions and mechanisms that form the vocabulary of our problem domain. Through invention, we desire generalized abstractions as well as new mechanisms that specify how objects collaborate discovery and inventions are both problems of classifications.

Intelligent classification is actually a part of all good science class of should be meaningful is relevant to every aspect of object oriented design classify helps us to identify generalization, specialization, and aggregation hierarchies among classes classify also guides us making decisions about modularizations.

The difficulty of classification

Examples of classify: Consider start and end of knee in leg in recognizing human speech, how do we know that certain sounds connect to form a word, and aren't instead a part of any surrounding words? In word processing system, is character class or words are class? Intelligent classification is difficult e.g. problems in classify of biology and chemistry until 18th century, organisms were arranged from the most simple to the most complex, human at top of the list. In mid 1970, organisms were classified according to genus and species. After a century later, Darwin's theory

came which was depended upon an intelligent classification of species. Category in biological taxonomy is the kingdom, increased in order from phylum, subphylum class, order, family, genus and finally species. Recently classify has been approached by grouping organisms that share a common generic heritage i.e. classify by DNA. DNA in useful in distinguishing organisms that are structurally similar but genetically very different classify depends on what you want classification to do. In ancient times, all substances were through to be sure ambulation of earth, fire, air and water. In mid 1960s – elements were primitive abstractive of chemistry in 1869 periodic law came.

The incremental and iterative nature of classification

Intelligent classification is intellectually hard work, and that it best comes about through an incremental and iterative process. Such processes are used in the development of software technologies such as GUI, database standards and programming languages. The useful solutions are understood more systematically and they are codified and analyzed. The incremental and iterative nature of classification directly impacts the construction of class and object hierarchies in the design of a complex software system. In practice, it is common to assert in the class structure early in a design and then revise this structure over time. Only at later in the design, once clients have been built that use this structure, we can meaningfully evaluate the quality of our classification. On the basis of this experience, we may decide to create new subclasses from existing once (derivation). We may split a large class into several smaller ones (factorization) or create one large class by uniting smaller ones (composition). Classify is hard because there is no such as a perfect classification (classify are better than others) and intelligent classify requires a tremendous amount of creative insight.

Identifying classes and objects

Classical and modern approaches: There are three general approaches to classifications.

- Classical categorization
- Conceptual clustering
- Prototypal theory

Classical categorizations

All the entities that have a given property or collection of properties in common forms a category. Such properties are necessary and sufficient to define the category. i.e. married people constitute a category i.e. either married or not. The values of this property are sufficient to decide to which group a particular person belongs to the category of tall/short people, where we can agree to some absolute criteria. This classification came from plato and then from Aristotle's classification of plants and animals. This approach of classification is also reflected in modern theories of child development. Around the age of one, child typically develops the concept of object permanence, shortly there after, the child acquires skill in classifying these objects, first using basic category such as dogs, cats and toys. Later the child develops more general categories (such as animals). In criteria for sameness among objects specifically, one can divide objects into disjoint sets depending upon the presence or absence of a particular property. Properties may denote more than just measurable characteristics. They may also encompass observable behaviors e.q. bird can fly but others can not is one property.

Conceptual clustering

It is a more modern variation of the classical approach and largely derives from attempts to explain how knowledge is represented in this approach, classes are generated by first formulating conceptual description of these classes and then classifying the entities according to the descriptions. e.g. we may state a concept such as "a love song". This is a concept more than a property, for the "love songness" of any song is not something that may be measured empirically. However, if we decide that a certain song is more of a love song than not, we place it in this category. thus this classify represents more of a probabilistic clustering of objects and objects may belong to one or more groups, in varying degree of fitness conceptual clustering makes absolute judgments of classify by focusing upon the best fit.

Prototype theory

It is more recent approach of classify where a class of objects is represented by a prototypical object, an object is considered to be a member of this class if and only if it resembles this prototype in significant ways. e.g. category like games, not in classical since no single common properties shared by all games, e.g. classifying chairs (beanbag chairs, barber chairs, in prototypes theory, we group things according to the degree of their relationship to concrete prototypes.

There approaches to classify provide the theoretical foundation of objected analysis by which we identify classes and objects in order to design a complex software system.

Object oriented Analysis

The boundaries between analysis and design are furry, although the focus of each is quite district. An analysis, we seek to model the world by discovering. The classes and objects that form the vocabulary of the problem domain and in design, we invent the abstractions and mechanisms that provide the behavior that this model requires following are some approaches for analysis that are relevant to object oriented system.

Classical approaches

It is one of approaches for analysis which derive primarily from the principles of classical categorization. e.g. Shlaer and Mellor suggest that classes and objects may come from the following sources:

- Tangible things, cars, pressure sensors
- Roles – Mother, teacher, politician
- Events – landing, interrupt
- Interactions – meeting

From the perspective of database modeling, ross offers the following list:

(i) People – human who carry out some function
(ii) Places – Areas set for people or thing
(iii) Things – Physical objects (tangible)
(iv) Organizations – organized collection of people resources
(v) Concepts – ideas
(vi) Events – things that happen

Coad and Yourdon suggest another set of sources of potential objects.

(i) Structure

(ii) Dences
(iii) Events remembered (historical)
(iv) Roles played (of users)
(v) Locations (office, sites)
(vi) Organizational units (groups)

Behavior Analysis

Dynamic behavior also be one of the primary source of analysis of classes and objects things can are grouped that have common responsibilities and form hierarchies of classes (including superclasses and subclasses). System behaviors of system are observed. These behaviors are assigned to parts of system and tried to understand who initiates and who participates in these behaviors. A function point is defined as one and user business functions and represents some kind of output, inquiry, input file or interface.

Domain Analysis

Domain analysis seeks to identify the classes and objects that are common to all applications within a given domain, such as patient record tracking, compliers, missile systems etc. Domain analysis defined as an attempt to identify the objects, operations and, relationships that are important to particular domain.

More and Bailin suggest the following steps in domain analysis.

i) Construct a strawman generic model of the domain by consulting with domain expert.
ii) Examine existing system within the domain and represent this understanding in a common format.
iii) Identify similarities and differences between the system by consulting with domain expert.
iv) Refine the generic model to accommodate existing systems.

Vertical domain Analysis: Applied across similar applications.

Horizontal domain Analysis: Applied to related parts of the same application domain expert is like doctor in a hospital concerned with conceptual classification.

Use case Analysis

Earlier approaches require experience on part of the analyst such a process is neither deterministic nor predictably successful. Use case analysis can be coupled with all three of these approaches to derive the process of analysis in a meaningful way. Use case is defined as a particular form pattern or exemplar some transaction or sequence of interrelated events. Use case analysis is applied as early as requirements analysis, at which time end users, other domain experts and the development team enumerate the scenarios that are fundamental to system's operation. These scenarios collectively describe the system functions of the application analysis then proceeds by a study of each scenario. As the team walks through each scenario, they must identify the objects that participate in the scenario, responsibilities of each object and how those objects collaborate with other objects in terms of the operations each invokes upon the other.

CRC cards

CRC are a useful development tool that facilitates brainstorming and enhances communication among developers. It is 3 x 5 index card (class/Responsibilities/collaborators i.e. CRC) upon

which the analyst writes in pencil with the name of class (at the top of card), its responsibilities(on one half of the card) and its collaborators (on the other half of the card). One card is created for each class identified as relevant to the scenario. CRC cards are arranged to represent generalization/specialization or aggregation hierarchies among the classes.

Informal English Description

Proposed by Abbott. It is writing an English description of the problem (or a part of a problem) and then underlining the nouns and verbs. Nouns represent candidate objects and the verbs represent candidate operations upon them. it is simple and forces the developer to work in the vocabulary of the problem space.

Structured Analysis

Same as English description as an alternative to the system, many CASE tools assists in modeling of the system. In this approach, we start with an essential model of the system, as described by data flow diagrams and other products of structured analysis. From this model we may proceed to identify the meaningful classes and objects in our problem domain in 3 ways.

- Analyzing the context diagrams, with list of input/output data elements; think about what they tell you or what they describe e.g. these make up list of candidate objects.
- Analyzing data flow domains, candidate objects may be derived from external entities, data stores, control stores, control transformation, candidate classes derive from data flows and candidate flows.
- By abstraction analysis: In structured analysis, input and output data are examined and followed inwards until they reach the highest level of abstraction.

Key abstractions and mechanisms

Identifying keyabstractions Finding key abstractions

A key abstraction is a class or object that forms part of the vocabulary of the problem domain. The primary value of identifying such abstractions is that they give boundaries to our problems. They highlight the things that are in the system and therefore relevant to our design and suppress the things that are outside of system identification of key abstraction involves two processes. Discovery and invention through discovery we come to recognize the abstraction used by domain experts. If through inventions, we create new classes and objects that are not necessarily part of the problem domain. A developer of such a system uses these same abstractions, but must also introduce new ones such as databases, screen managers, lists queues and so on. These key abstractions are artifacts of the particular design, not of the problem domain.

Refining key abstractions

Once we identify a certain key abstraction as a candidate, we must evaluate it. Programmer must focus on questions. How we objects of this class created? What operations can be done on such objects? If there are not good answers to such questions, then the problem is to be thought again and proposed solution is to be found out instead of immediately starting to code among the problems placing classes and objects at right levels of abstraction is difficult. Sometimes we may find a general subclass and so may choose to move it up in the class structure, thus increasing the degree of sharing. This is called class promotion. Similarly, we may find a class to be too

general, thus making inheritance by a subclass difficult because of the large semantic gap. This is called a grainsize conflict.
Naming conventions are as follows:

- Objects should be named with proper noun phrases such as the sensor or simply shapes.
- Classes should be named with common noun phrases, such as sensor or shapes.
- Modifier operations should be named with active verb phrases such as draw, moveleft.
- Selector operations should imply a query or be named with verbs of the form "to be" e.g. is
  open, extent of.

Identifying Mechanisms Finding Mechanism

A mechanism is a design decision about how collection of objects cooperates. Mechanisms represent patterns of behavior e.g. consider a system requirement for an automobile: pushing the accelerator should cause the engine to run faster and releasing the accelerator should cause the engine to run slower. Any mechanism may be employed as long as it delivers the required behavior and thus which mechanism is selected is largely a matter of design choice. Any of the following design might be considered.

- A mechanical linkage from the acceleration to the (the most common mechanism)
- An electronic linkage from a preserve sensor below the accelerator to a computer that controls the carburetor (a drive by wire mechanism)
- No linkage exists; the gas tank is placed on the roof of the car and gravity causes fuel to flow to the engine. Its rate of flow is regulated by a clip around the fuel the pushing on the
  accelerator pedal eases tension on the clip, causing the fuel to flow faster (a low cost mechanism)

Key abstractions reflect the vocabulary of the problem domain and mechanisms are the soul of the design. Idioms are part of a programming culture. An idiom is an expression peculiar to a certain programming language. e.g. in CLOS, no programmer use under score in function or variable names, although this is common practice in ada.
A frame work is collection of classes that provide a set of service for a particular domain. A framework exports a number of individual classes and mechanisms which clients can use.

Examples of mechanisms:
Consider the drawing mechanism commonly used in graphical user interfaces. Several objects must collaborate to present an image to a user: a window, a new, the model being viewed and some client that knows when to display this model. The client first tells the window to draw itself. Since it may encompass several subviews, the window next tells each if its subviews to draw them. Each subview in turn tells the model to draw itself ultimately resulting in an image shown to the user.

UNIT-III:

Introduction to UML:

- Why we model
- Conceptual model of UML
- Architecture
- Classes
- Relationships
- Common Mechanisms
- Class diagrams
- Object diagrams.

UNIT-III

Introduction to UML

Why we model

- Importance of Modeling
- Four principlasof modeling
- The essential BluePrints of a Software System
- Object oriented modeling

Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system. We build models to visualize and control the system's architecture. We build models to better understand the system we are building, often exposing opportunities for simplification and reuse. And we build models to manage risk.

The Importance of Modeling

If you want to build a dog house, you can pretty much start with a pile of lumber, some nails, and a few basic tools such as a hammer, saw, and tape measure. In a few hours, with little prior planning, you'll likely end up with a dog house that's reasonably functional, and you can probably do it with no one else's help. As long as it's big enough and doesn't leak too much, your dog will be happy. If it doesn't work out, you can always start over, or get a less demanding dog.

If you really want to build the software equivalent of a house or a high rise, the problem is more than just a matter of writing lots of software in fact, the trick is in creating the right software and in figuring out how to write less software. This makes quality software development an issue of architecture and process and tools. Even so, many projects start out looking like dog houses but grow to the magnitude of a high rise simply because they are a victim of their own success. There comes a time when, if there was no consideration given to architecture, process, or tools, the dog house, now grown into a high rise, collapses of its own weight. The collapse of a dog house may annoy your dog; the failure of a high rise will materially affect its tenants.

Unsuccessful software projects fail in their own unique ways, but all successful projects are alike in many ways. There are many elements that contribute to a successful software organization; one common thread is the use ofmodeling.

Modeling is a proven and well-accepted engineering technique. We build architectural models of houses and high rises to help their users visualize the final product. We may even build mathematical models to analyze the effects of winds or earthquakes on our buildings.

Modeling is not just a part of the building industry. It would be inconceivable to deploy a new aircraft or an automobile without first building models from computer models to physical wind tunnel models to full-scale prototypes. New electrical devices, from microprocessors to telephone switching

systems, require some degree of modeling in order to better understand the system and to communicate those ideas to others. In the motion picture industry, storyboarding, which is a form of modeling, is central to any production. In the fields of sociology, economics, and business management, we build models so that we can validate our theories or try out new ones with minimal risk and cost.

> *We build models so that we can better understand the system we are developing.*

Through modeling, we achieve four aims.

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

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

> *We build models of complex systems because we cannot comprehend such a system in its entirety.*

There are limits to the human ability to understand complexity. Through modeling, we narrow the problem we are studying by focusing on only one aspect at a time. This is essentially the approach of"divide-and-conquer" that Edsger Dijkstra spoke of years ago: Attack a hard problem by dividing it into a series of smaller problems that you can solve. Furthermore, through modeling, we amplify the human intellect. A model properly chosen can enable the modeler to work at higher levels of abstraction.

Principles of Modeling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling. First,

> *The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.*

In other words, choose your models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that you simply could not gain otherwise; the wrong models will mislead you, causing you to focus on irrelevant issues.
Second,

> *Every model may be expressed at different levels of precision.*

If you are building a high rise, sometimes you need a 30,000-foot viewfor instance, to help your investors visualize its look and feel. Other times, you need to get down to the level of the studsfor instance, when there's a tricky pipe run or an unusual structural element.

> The same is true with software models. Sometimes a quick and simple executable model of the user interface is exactly what you need; at other times you have to get down and dirty with the bits, such as when you are specifying cross-system interfaces or wrestling with networking bottlenecks.

. In any case, the best kinds of models are those that let you choose your degree of detail, depending

on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

Third,

The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details.

In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

Fourth,

No single model or view is sufficient. Every nontrivial system is best approached through a small set of nearly independent models with multiple viewpoints.

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans. And within any kind of model, you need multiple views to capture the breadth of the system, such as blueprints of different floors.

The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, you can study electrical plans in isolation, but you can also see how they map to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

The same is true of object-oriented software systems. To understand the architecture of such a system, you need several complementary and interlocking views: a use case view (exposing the requirements of the system), a design view (capturing the vocabulary of the problem space and the solution space), an interaction view (showing the interactions among the parts of the system and between the system and the environment), an implementation view (addressing the physical realization of the system), and a deployment view (focusing on system engineering issues). Each of these views may have structural, as well as behavioral, aspects. Together, these views represent the blueprints of software.

Object-Oriented Modeling

In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective.

The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure or function. This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones. There's nothing

inherently evil about such a point of view except that it tends to yield brittle systems. As requirements change (and they will) and the system grows (and it will), systems built with an algorithmic focus turn out to be very hard tomaintain.

The contemporary view of software development takes an object-oriented perspective. In this approach, the main building block of all software systems is the object or class. Simply put, an object is a thing, generally drawn from the vocabulary of the problem space or the solution space. A class is a description of a set of objects that are similar enough (from the modeler's viewpoint) to share a specification. Every object has identity (you can name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behavior (you can do things to the object, and it can do things to other objects as well).

A Conceptual Model of the UML

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

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML

There are four kinds of things in the UML:

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

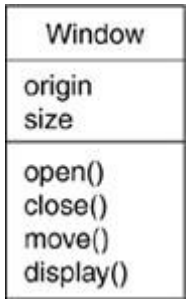
These things are the basic object-oriented building blocks of the UML. You use them to write well- formed models.

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 in Figure 2-1.

Figure 2-1. 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 in Figure 2-2.

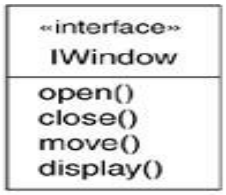


Figure 2-2. Interfaces

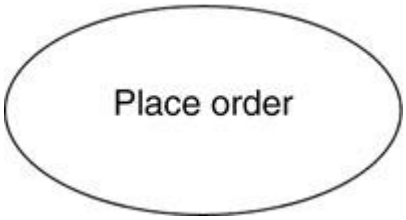
A *collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Collaborations have structural, as well as behavioral, dimensions. A given class or object might participate in 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 in Figure2-3.

Figure 2-3. Collaboration

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 in Figure 2-4.

Figure 2-4. Use Cases

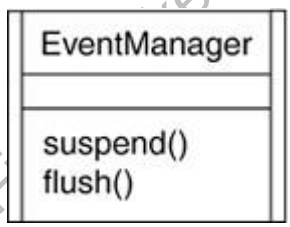


The remaining three thingsactive classes, components, and nodesare all class-like, meaning they also describe sets of entities that share the same attributes, operations, relationships, and semantics.

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 in Figure 2-5.

Figure 2-5. 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 in Figure 2-6.

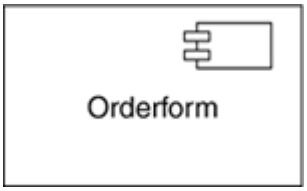


Figure 2-6. Components

The remaining two elementsartifacts and nodesare 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 in Figure 2-7.

Figure 2-7. 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 in Figure 2-8.

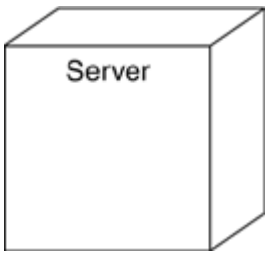


Figure 2-8. Nodes

These elementsclasses, interfaces, collaborations, use cases, active classes, components, artifacts, and nodesare 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 in Figure 2-9.

Figure 2-9. Messages

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

machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in Figure 2-10.

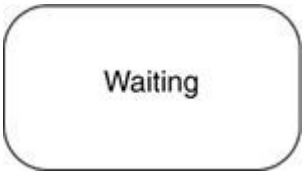
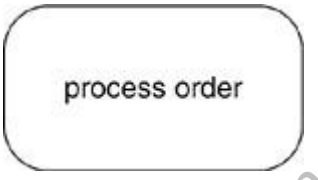


Figure 2-10. States

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 2-11. Actions



These three elementsinteractions, state machines, and activitiesare 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 in Figure 2-12.

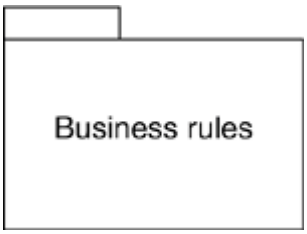


Figure 2-12. Packages

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 in Figure 2-13.

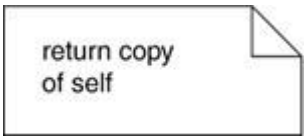


Figure 2-13. Notes

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).

Architecture

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

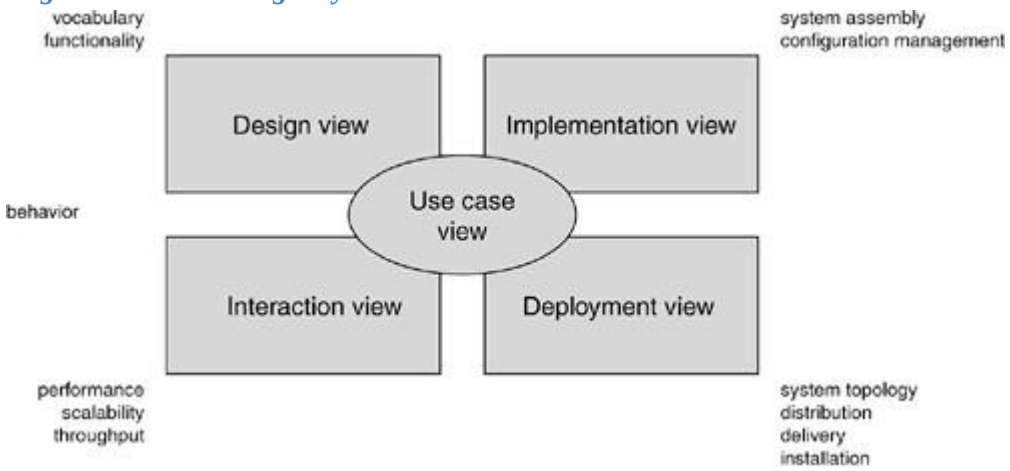
Architecture is the set of significant decisions about

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

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

As Figure 2-23 illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

Figure 2-23. Modeling a System's Architecture



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

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

The *interaction view* of a system shows the flow of control among its various parts, including possible concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that control the system and the messages that flow between them.

The *implementation view* of a system encompasses the artifacts that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent files that can be assembled in various ways to produce a running system. It is also concerned with the mapping from logical classes and components to physical artifacts. With the UML, the static aspects of this view are captured in artifact diagrams; the dynamic aspects of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

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

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another: Nodes in the deployment view hold components in the implementation view that, in turn, represent the

physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express each of these five views.

Classes

Modeling a system involves identifying the things that are important to your particular view. These things form the vocabulary of the system you are modeling. For example, if you are building a house, things like walls, doors, windows, cabinets, and lights are some of the things that will be important to you as a home owner. Each of these things can be distinguished from the other.

Each of them also has a set of properties. Walls have a height and a width and are solid. Doors also have a height and a width and are solid as well, but have the additional behavior that allows them to open in one direction. Windows are similar to doors in that both are openings that pass through walls, but windows and doors have slightly different properties. Windows are usually (but not always) designed so that you can look out of them instead of pass through them.

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

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

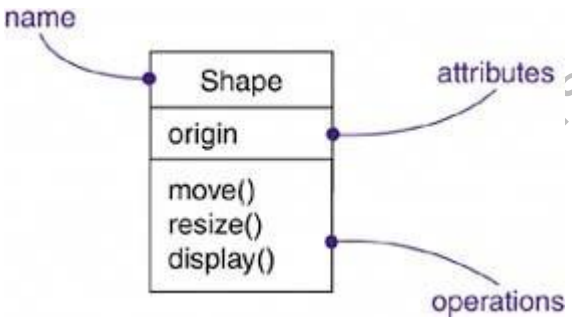


Figure 4-1. Classes

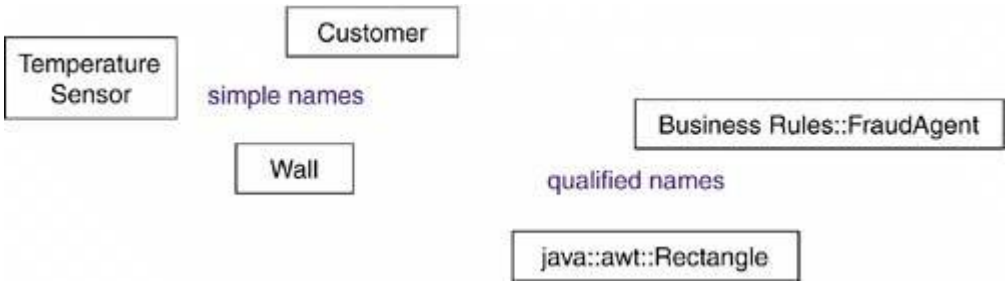
Terms and Concepts

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

Names

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a *simple name*; a *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, as Figure 4-2 shows.

Figure 4-2. Simple and Qualified Names



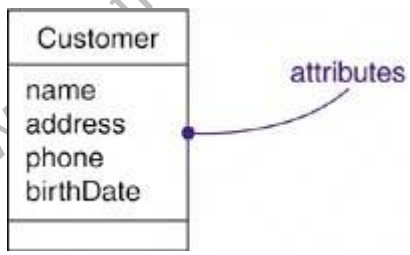
Note

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

Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth. An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass. At a given moment, an object of a class will have specific values for every one of its class's attributes. Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names, as shown in Figure 4-3.

Figure 4-3. Attributes

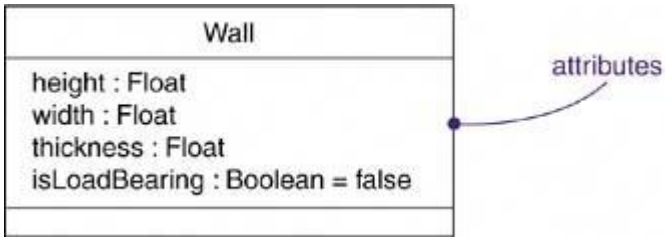


Note

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

You can further specify an attribute by stating its class and possibly a default initial value, as shown Figure 4-4.

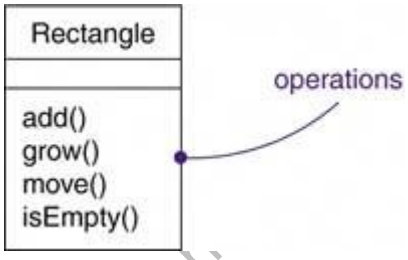
Figure 4-4. Attributes and Their Class



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 awtpackage, all objects of the class Rectangle can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure 4-5.

Figure 4-5. Operations

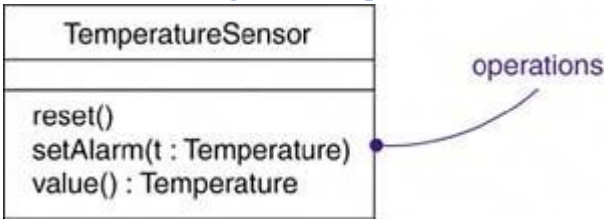


Note

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

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, as shown in Figure 4-6.

Figure 4-6. Operations and Their Signatures



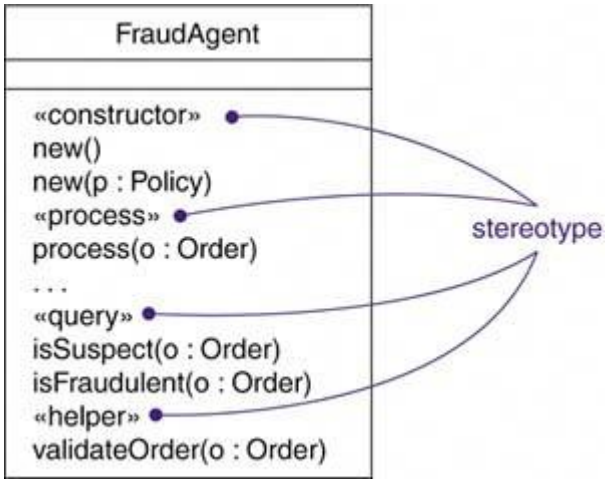
Organizing Attributes and Operations

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 ("..."). You can also suppress the compartment entirely, in which case you can't tell if there are any attributes or operations or how many there are.

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as shown in Figure 4-7.

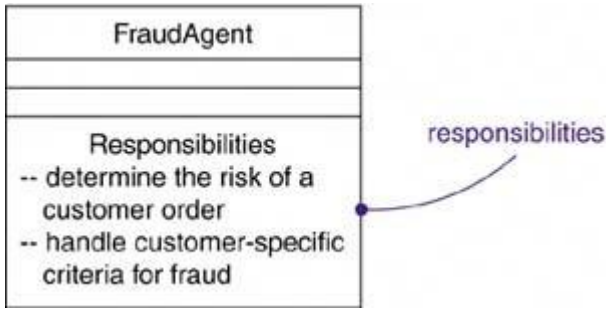
Figure 4-7. Stereotypes for Class Features



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.

When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary. Techniques like CRC cards and use case-based analysis are especially helpful here. A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful. As you refine your models, you will translate these responsibilities into a set of attributes and operations that best fulfill the class's responsibilities. Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure 4-8. Figure 4-8. Responsibilities



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 in Figure 2-14.

Figure 2-14. 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 in Figure 2-15.

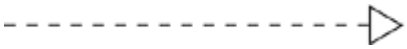
Figure 2-15. Associations

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 in Figure 2-16.

Figure 2-16. 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 in Figure 2-17.

Figure 2-17. 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.

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 is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's underlying specification. You use the UML's graphical notation to visualize a system; you use 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, Figure 2-18 shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

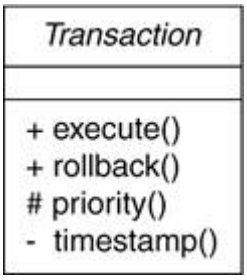


Figure 2-18. Adornments

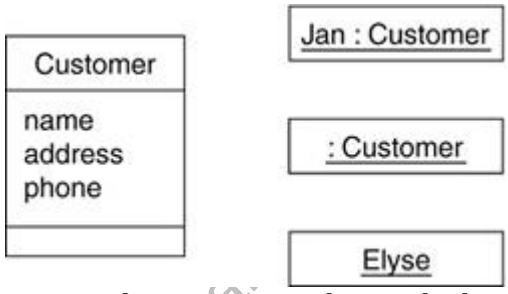
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

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 Figure 2-19. Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

Figure 2-19. 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 Figure 2-20.

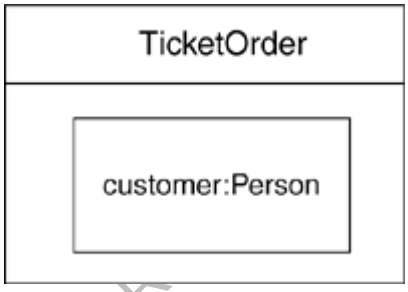
Figure 2-20. Interfaces and Implementation

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 (Figure 2-21).

Figure 2-21. 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 Overflowin 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. In Figure 2-19, for example, the class EventQueueis 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 Figure 2-22 shows, you can add a constraint that explicitly marks these for the operation add.

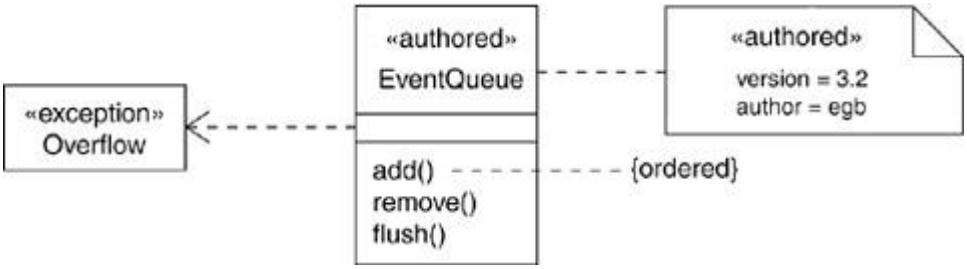


Figure 2-22. Extensibility Mechanisms

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.

Class Diagrams

Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

You use class diagrams to model the static design view of a system. For the most part, this involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas. Class diagrams are also the foundation for a couple of related diagrams: component diagrams and deployment diagrams.

Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.

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
- interface
- Dependency, generalization, and association relationships

Like all other diagrams, class diagrams may contain notes and constraints.

Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes you'll want to place instances in your class diagrams as well, especially when you want to visualize the (possibly dynamic) type of an instance.

Note

Component diagrams and deployment diagrams are similar to class diagrams, except that instead of containing classes they contain components and nodes, respectively.

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 asystem

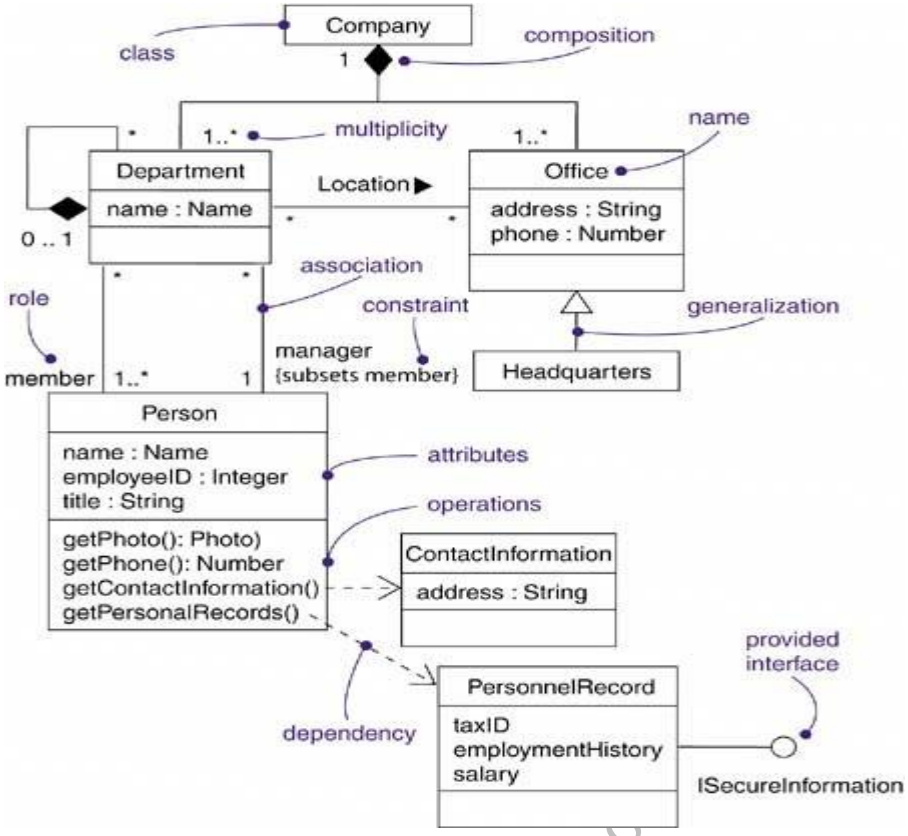
Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simplecollaborations

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 databaseschema

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 Comment

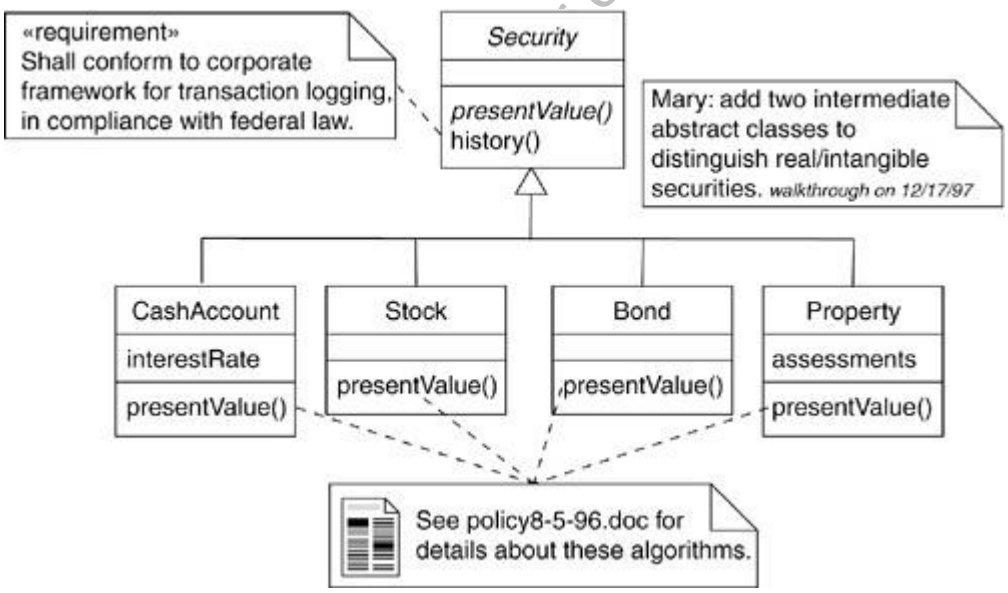
The most common purpose for which you'll use notes is to write down free-form observations, reviews, or explanations. By putting these comments directly in your models, your models can become a common repository for all the disparate artifacts you'll create during development. You can even use notes to visualize requirements and show how they tie explicitly to the parts of your model.

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.

For example, Figure 6-9 shows a model that's a work in progress of a class hierarchy, showing some requirements that shape the model, as well as some notes from a design review.

Figure 6-9. Modeling Comments



In this example, most of the comments are simple text (such as the note to Mary), but one of them (the note at the bottom of the diagram) provides a hyperlink to another document.

Modeling New Properties

The basic properties of the UML's building blocksattributes and operations for classes, the contents of packages, and so onare generic enough to address most of the things you'll want to model. However,

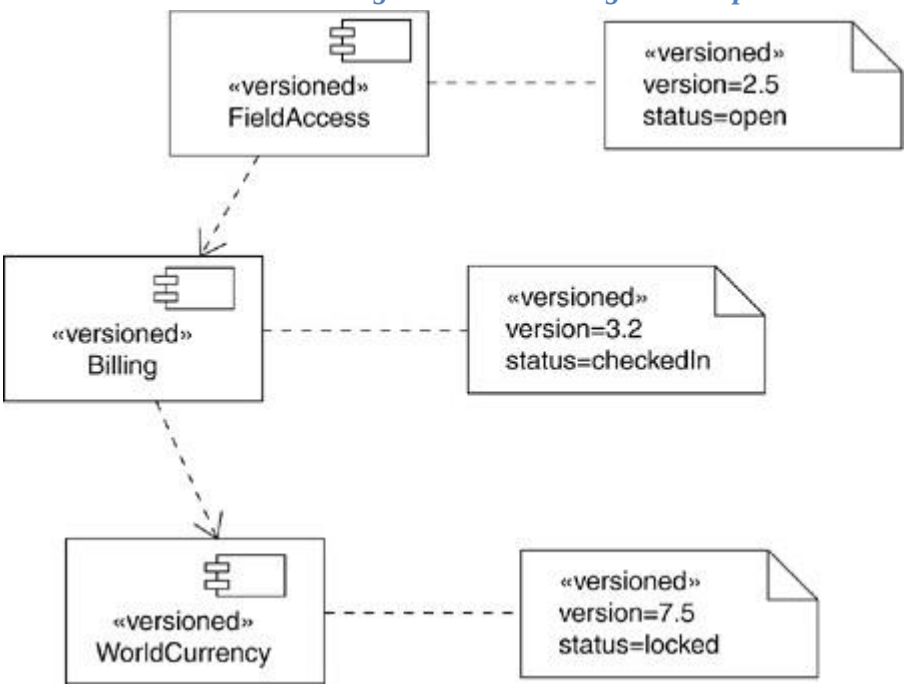
if you want to extend the properties of these basic building blocks, you need to define stereotypes and tagged values.

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.

For example, suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.
Figure 6-10 shows three subsystems, each of which has been extended with the «versioned» stereotype to include its version number and status.

Figure 6-10. Modeling New Properties



Note

The values of tags such as version and status are things that can be set by tools. Rather than setting these values in your model by hand, you can use a development environment that integrates your configuration management tools with your modeling tools to maintain these values for you.

Modeling New Semantics

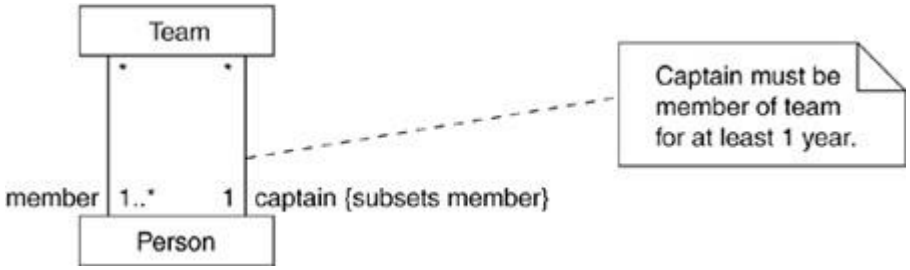
When you create a model using the UML, you work within the rules the UML lays down. That's a good thing, because it means that you can communicate your intent without ambiguity to anyone else who knows how to read the UML. However, if you find yourself needing to express new semantics about which the 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.

For example, Figure 6-11 models a small part of a corporate human resources system.

Figure 6-11. Modeling New Semantics



This diagram shows that each Person may be a member of zero or more Teams and that each Team must have at least one Person as a member. This diagram goes on to indicate that each Team must have exactly one Person as a captain and every Person may be the captain of zero or more Teams. All of these semantics can be expressed using simple UML. However, to assert that a captain must also be a member of the same team is something that cuts across multiple associations and cannot be expressed using simple UML. To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the Team, connecting the two associations with a constraint. There is also a constraint that the captain must be a member for at least 1 year.

Object Diagrams

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

You use object diagrams to model the static design view or static process view of a system. This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.

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

With the UML, you use class diagrams to visualize the static aspects of your system's building blocks. You use interaction diagrams to visualize the dynamic aspects of your system, consisting of instances of these building blocks and messages dispatched among them. An object diagram covers a set of instances of the things found in a class diagram. An object diagram, therefore, expresses the static part of an interaction, consisting of the objects that collaborate but without any of the messages passed among them. In both cases, an object diagram freezes a moment in time, as in Figure 14-1.

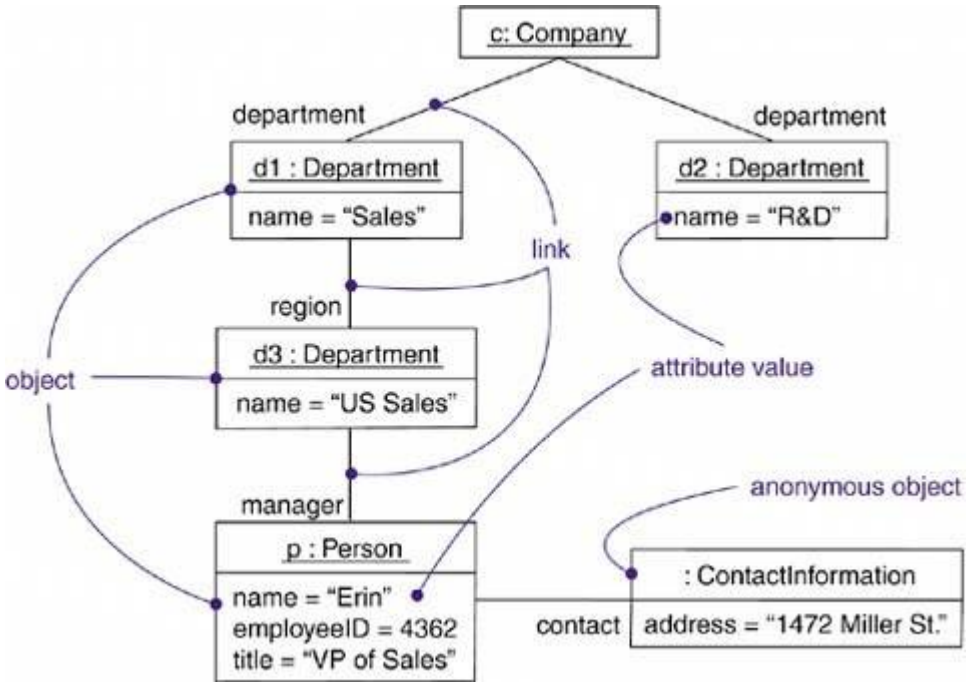


Figure 14-1. An Object Diagram

Terms and Concepts

An *object diagram* is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

Common Properties

An object diagram is a special kind of diagram and shares the same common properties as all other diagramsthat is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

Contents

Object diagrams commonly contain

- Objects
- Links

Like all other diagrams, object diagrams may contain notes and constraints.

Sometimes you'll want to place classes in your object diagrams as well, especially when you want to visualize the classes behind each instance.

Note

An object diagram correlates with a class diagram: The class diagram describes the general situation, and the instance diagram describes specific instances derived from the class diagram.An object diagram contains primarily objects and links. Deployment diagrams may also occur in generic and instance forms: General deployment diagrams describe node types, and instance deployment diagrams describe a concrete configuration of node instances described by those types.

Common Uses

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

When you model the static design view or static interaction view of a system, you typically use object diagrams to model object structures
Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another. You can show dynamic behavior and execution as a sequence of frames.

Common Modeling Techniques

Modeling Object Structures

When you construct a class diagram, a component diagram, or a deployment diagram, what you are really doing is capturing a set of abstractions that are interesting to you as a group and, in that context, exposing their semantics and their relationships to other abstractions in the group. These diagrams show only potentiality. If class A has a one-to-many association to class B, then for one instance of A there might be five instances of B; for another instance of A there might be only one instance of B. Furthermore, at a given moment in time, that instance of A, along with the related instances of B, will each have certain values for their attributes and state machines.

If you freeze a running system or just imagine a moment of time in a modeled system, you'll find a set of objects, each in a specific state and each in a particular relationship to other objects. You can use object diagrams to visualize, specify, construct, and document the structure of these snapshots.
Object diagrams are especially useful for modeling complex data structures.

When you model your system's design view, a set of class diagrams can be used to completely specify the semantics of your abstractions and their relationships. With object diagrams, however, you cannot completely specify the object structure of your system. For an individual class, there may be a multitude of possible instances, and for a set of classes in relationship to one another, there may be many times more possible configurations of these objects. Therefore, when you use object diagrams, you can only meaningfully expose interesting sets of concrete or prototypical objects. This is what it means to model an object structurean object diagram shows one set of objects in relation to one another at one moment intime.

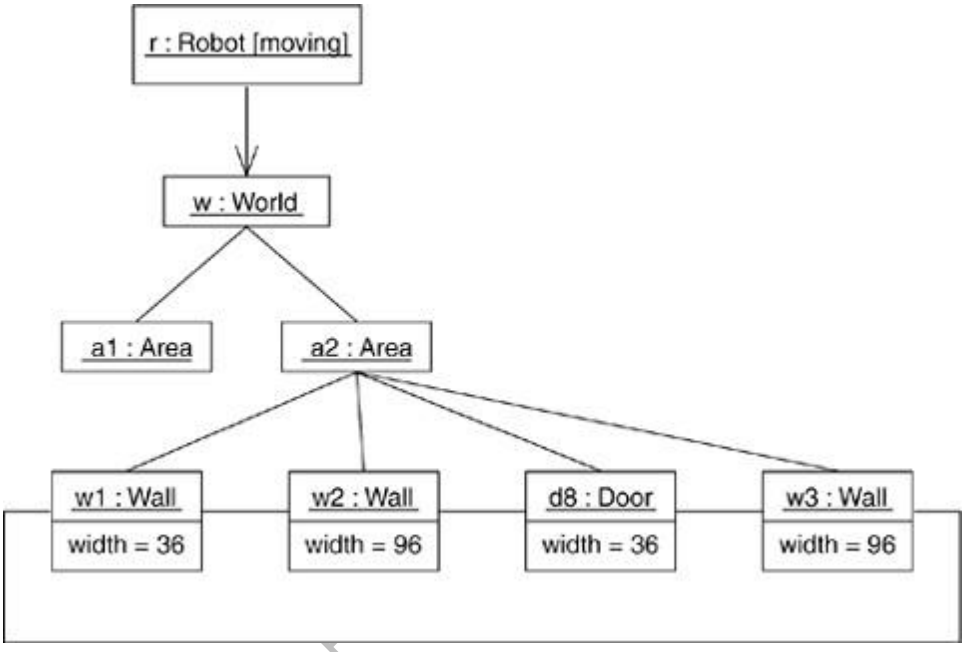
To model an object structure,

- 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 otherthings.
- Create a collaboration to describe amechanism.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things as well.

- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.

For example, Figure 14-2 shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

Figure 14-2. Modeling Object Structures



As this figure indicates, one object represents the robot itself (r, an instance of Robot), and r is currently in the state marked moving. This object has a link to w, an instance of World, which represents an abstraction of the robot's world model.

As this figure indicates, one object represents the robot itself (r, an instance of Robot), and r is currently in the state marked moving. This object has a link to w, an instance of World, which represents an abstraction of the robot's world model.

At this moment in time, w is linked to two instances of Area. One of them (a2) is shown with its own links to three Wall objects and one Door object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

Reverse Engineering

Reverse engineering (the creation of a model from code) an object diagram can be useful. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

To reverse engineer an object diagram,

- choose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time. Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.
- You will usually have to manually add or label structure that is not explicit in the target code. The missing information supplies the design intent that is only implicit in the final code.es are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

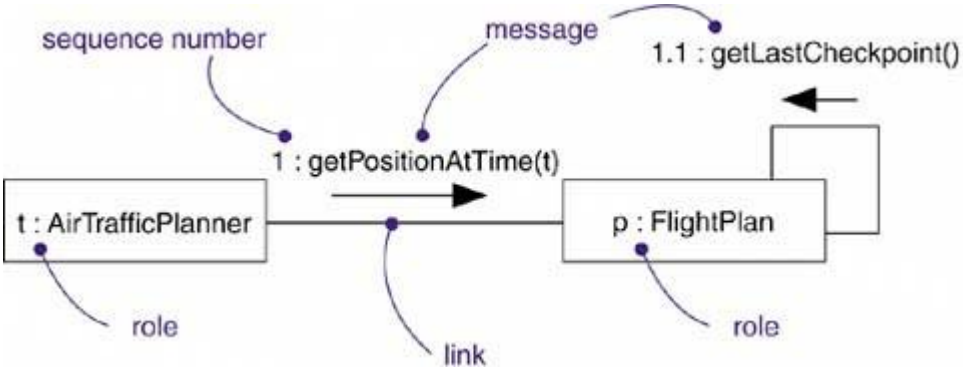
Chapter 4: Interactions

In this chapter

- Roles, links, messages, actions, and sequences
- Modeling flows of control
- Creating well-structured algorithms

In every interesting system, objects don't just sit idle; they interact with one another by passing messages. An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.

Figure 4-1. Messages, Links, and Sequencing



Terms and Concepts

An *interaction* is a behavior that comprises a set of messages exchanged among objects in a set of roles within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Objects and Roles

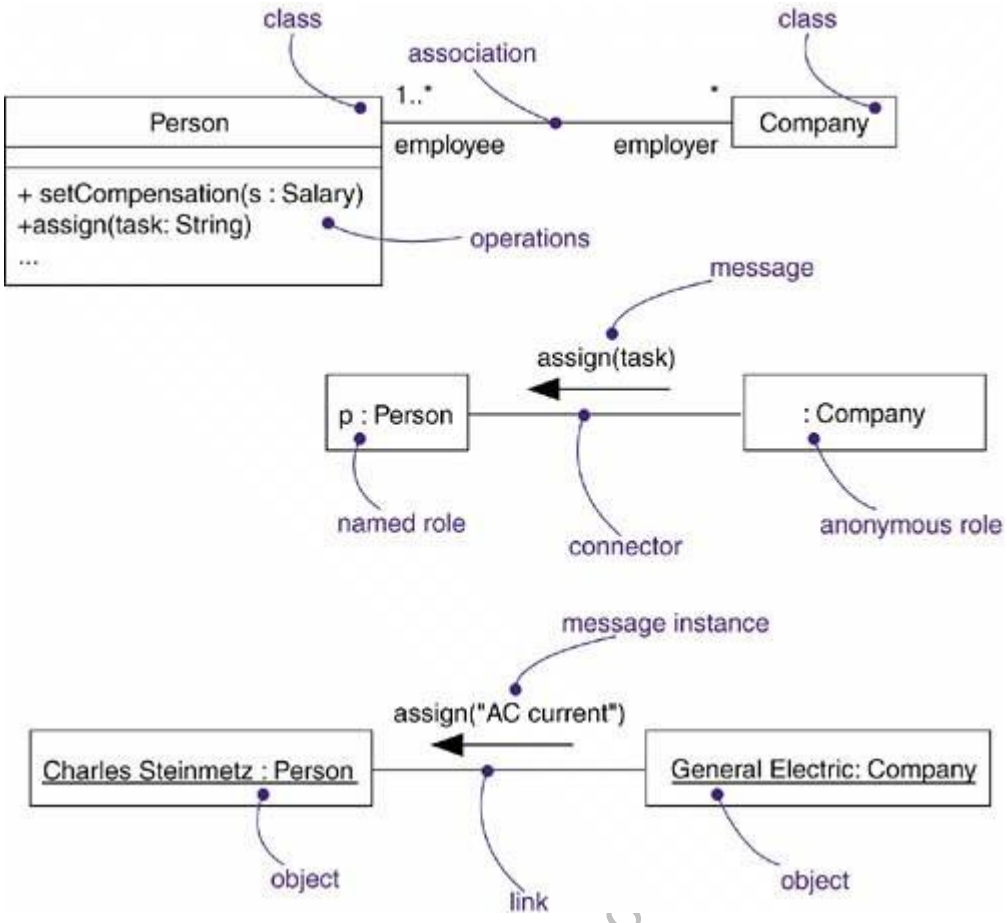
The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, `p`, an instance of the class `Person`, might denote a particular human. Alternately, as a prototypical thing, `p` might represent any instance of `Person`.

> **Note**
>
> In a collaboration, the interactors are usually prototypical things that play particular roles, not specific objects in the real world, although it is sometimes useful to describe collaborations among particular objects.

Links and Connectors

A link is a semantic connection among objects. In general, a link is an instance of an association. As shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.

Figure 4-2. Associations, Links, and Connectors



A link specifies a path along which one object can dispatch a message to another (or the same) object. Most of the time it is sufficient to specify that such a path exists. If you need to be more precise about how that path exists, you can adorn the appropriate end of the link with one of the following constraints:

- `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

In most models, we are more interested in prototypical objects and links within some context rather than individual objects and links. A prototypical object is called a *role*; a prototypical link is called a *connector*; the context is a collaboration or the internal structure of a classifier. The multiplicity of roles and connectors are defined relative to their enclosing context. For example, a multiplicity of 1 on a role means one object represents the role for each object that represents the context. A collaboration or internal structure can be used many times, just like a class declaration; each use is bound to a separate set of objects and links for the context, roles, and links.

Figure 4-2 shows an example.

Messages

A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue. The receipt of a message instance may be considered an occurrence of an event. (An *occurrence* is the UML name for an instance of an event.)

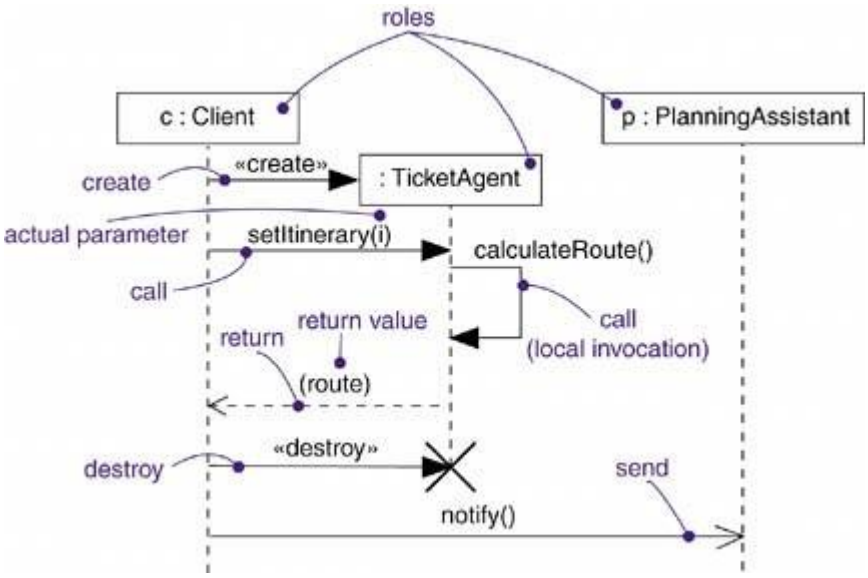
When you pass a message, an action usually results on its receipt. An action may result in a change in state of the target object and objects accessible from it.

In the UML, you can model several kinds of messages.

- 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

The UML provides a visual distinction among these kinds of messages, as Figure 4-3 shows.

Figure 4-3. Messages

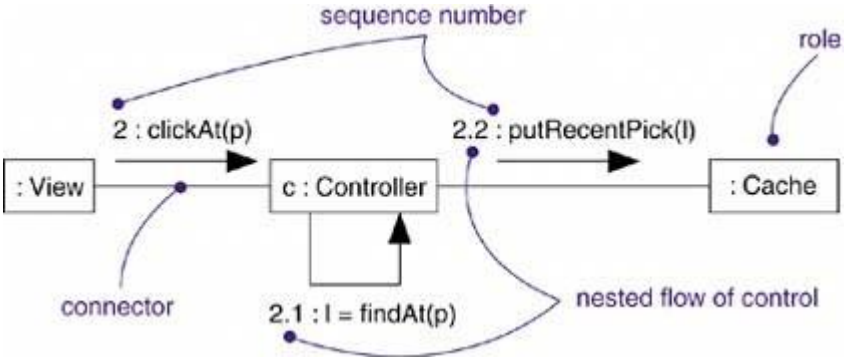


Sequencing

When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning; the start of every sequence is rooted in some process or thread. Furthermore, any sequence will continue as long as the process or thread that owns it lives. A nonstop system, such as you might find in real time device control, will continue to execute as long as the node it runs on is up.

A communication diagram shows message flow between roles within a collaboration. Messages flow along connections of the collaboration, as in Figure 4-4.

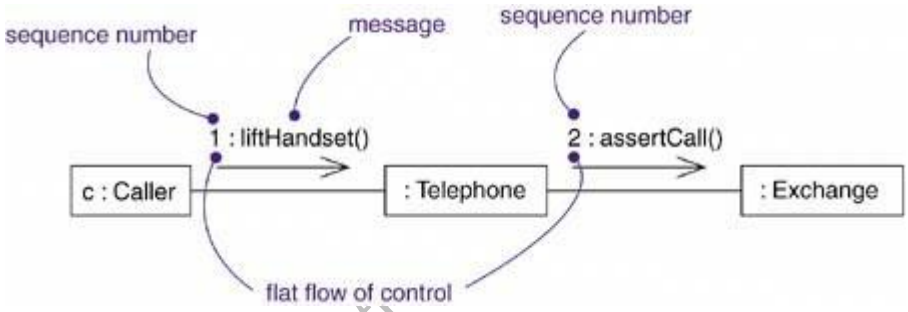
Figure 4-4. Procedural Sequence



Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure 4-4 shows. In this case, the message `findAt` is specified as the first message nested in the second message of the sequence (`2.1`).

Less common but also possible, as Figure 4-5 shows, you can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step. In this case, the message `assertCall` is specified as the second message in the sequence.

Figure 4-5. Flat Sequence



The distinction between asynchronous and procedural sequences is important in the modern concurrent computing world.

Creation, Modification, and Destruction

Most of the time the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions objects may be created (specified by a `create` message) and destroyed (specified by a `destroy` message). The same is true of links: The relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach a note to its role within a communication diagram.

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object in a sequence diagram by showing the state or the values on the lifeline.
Within a sequence diagram, the lifetime, creation, and destruction of objects or roles are explicitly shown by the vertical extent of their lifelines. Within a communication diagram, creation and destruction must be indicated using notes. Use sequence diagrams if object lifetimes are important to show.

Representation

When you model an interaction, you typically include both roles (each one representing objects that appear in an instance of the interaction) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those roles and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the roles that send and receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a communication diagram. Both sequence diagrams and communication diagrams are kinds of interaction diagrams.

Sequence diagrams and communication diagrams are similar, meaning that you can take one and transform it into the other, although they often show different information, so it may not be so useful to go back and forth. There are some visual differences. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, communication diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Control

When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.

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. Name the roles.
- If your 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. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- 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.

For example, Figure 4-6 shows a set of roles that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three roles: `p` (a `StockQuotePublisher`), `s1`, and `s2` (both instances of `StockQuoteSubscriber`). This figure is an example of a sequence diagram, which emphasizes the time order of messages.

Figure 4-6. Flow of Control by Time

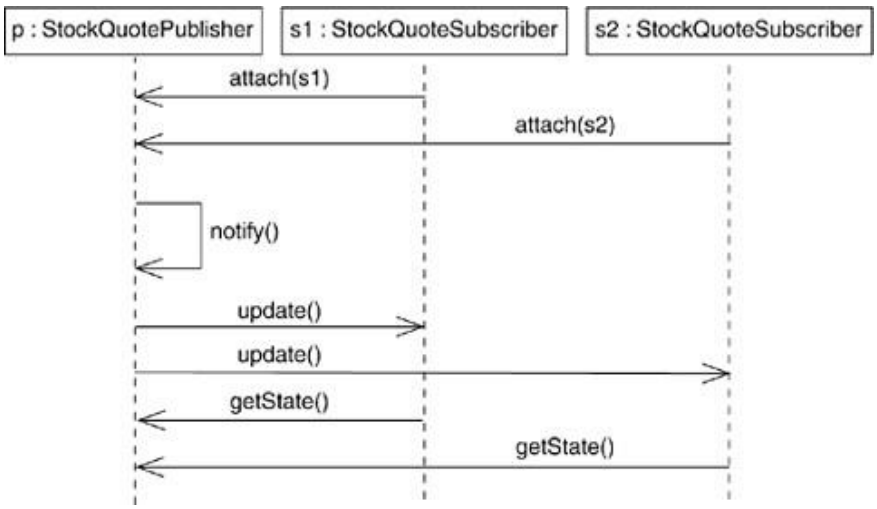
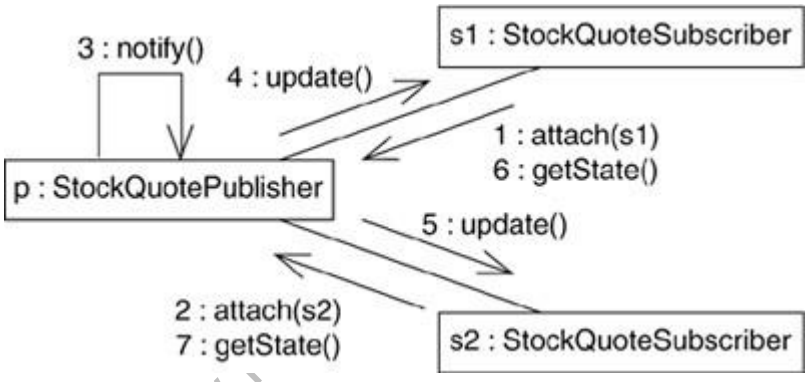


Figure 4-7 is semantically equivalent to the previous one, but it is drawn as a communication diagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure 4-7. Flow of Control by Organization



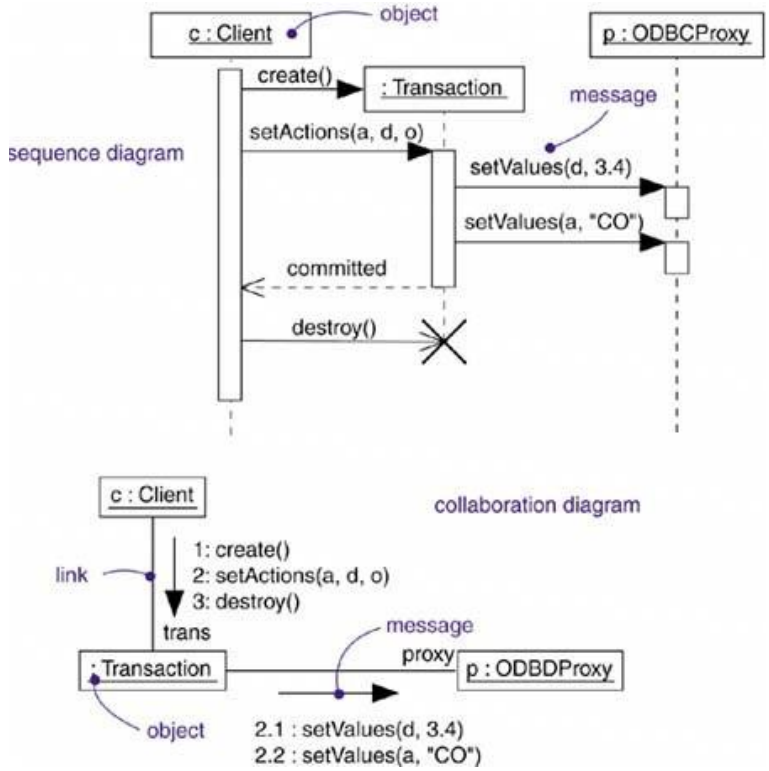
Interaction Diagrams

Sequence diagrams and communication diagrams which are collectively called interaction diagrams are two of the diagrams used in the UML for modeling the dynamic aspects of systems. An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. 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 that send and receive messages.

You use interaction diagrams to model the dynamic aspects of a system. For the most part, this involves modeling concrete or prototypical instances of classes, interfaces, components, and nodes, along with the messages that are dispatched among them, all in the context of a scenario that illustrates a behavior. Interaction diagrams may stand alone to visualize, specify, construct, and document the dynamics of a particular society of objects, or they may be used to model one particular flow of control of a usecase.

Interaction diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Figure 19-1. 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. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *communication diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a communication diagram is a collection of vertices and arcs.

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

If an object changes the values of its attributes, its state, or its roles, you can place a state icon on its lifeline at the point that the change occurs, showing those modifications.

Second, there is the focus of control. The focus of control 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. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self-operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing and control has not passed to another object, but this is rather fussy.

The main content in a sequence diagram is the messages. A message is shown by an arrow from one lifeline to another. The arrowhead points to the receiver. If the message is asynchronous, the line has a stick arrowhead. If the message is synchronous (a call), the line has a filled triangular arrowhead. A reply to a synchronous message (a return from a call) is shown by a dashed arrow with a stick arrowhead. The return message may be omitted, as there is an implicit return after any call, but it is often useful for showing return values

The ordering of time along a single lifeline is significant. Usually the exact distance does not matter; lifelines only show relative sequences, so the lifeline is not a scale diagram of time. Usually the positions of messages on separate pairs of lifelines do not imply any sequencing information; the messages could occur in any order. The entire set of messages on separate lifelines forms a partial ordering. A series of messages establishes a chain of causality, however, so that any point on another lifeline at the end of the chain must always follow the point on the original lifeline at the start of the chain.

Structured Control in Sequence Diagrams

A sequence of messages is fine for showing a single, linear sequence, but often we need to show conditionals and loops. Sometimes we want to show concurrent execution of multiple sequences. This kind of high-level control can be shown using structured control operators in sequence diagrams.

A control operator is shown as a rectangular region within the sequence diagram. It has a taga text label inside a small pentagon in the upper left cornerto tell what kind of a control operator it is. The operator applies to the lifelines that cross it. This is considered the body of the operator. If a lifeline does not apply to the operator, it may be interrupted at the top of the control operator and resumed at the bottom. The following kinds of control are the most common:

Optional execution

The tag is `opt`. The body of the control operator is executed if a guard condition is true when the operator is entered. The guard condition is a Boolean expression that may appear in square brackets at the top of any one lifeline within the body and may reference attributes of that object.

Conditional execution

The tag is `alt`. The body of the control operator is divided into multiple subregions by horizontal dashed lines. Each subregion represents one branch of a conditional. Each subregion has a guard condition. If the guard condition for a subregion is true, the subregion is executed. However, at mosT

one subregion may be executed; if more than one guard condition is true, the choice of subregion is nondeterministic and could vary from execution to execution. If no guard condition is true, then control continues past the control operator. One subregion may have a the special guard condition `[else]`; this subregion is executed if none of the other guard conditions are true.

Parallel execution

The tag is `par`. The body of the control operator is divided into multiple subregions by horizontal dashed lines. Each subregion represents a parallel (concurrent) computation. In most cases, each subregion involves different lifelines. When the control operator is entered, all of the subregions execute concurrently. The execution of the messages in each subregion is sequential, but the relative order of messages in parallel subregions is completely arbitrary. This construct should not be used if the different computations interact. There are very many real-world situations that decompose into independent, parallel activities, however, so this is a very useful operator.

Loop (iterative) execution

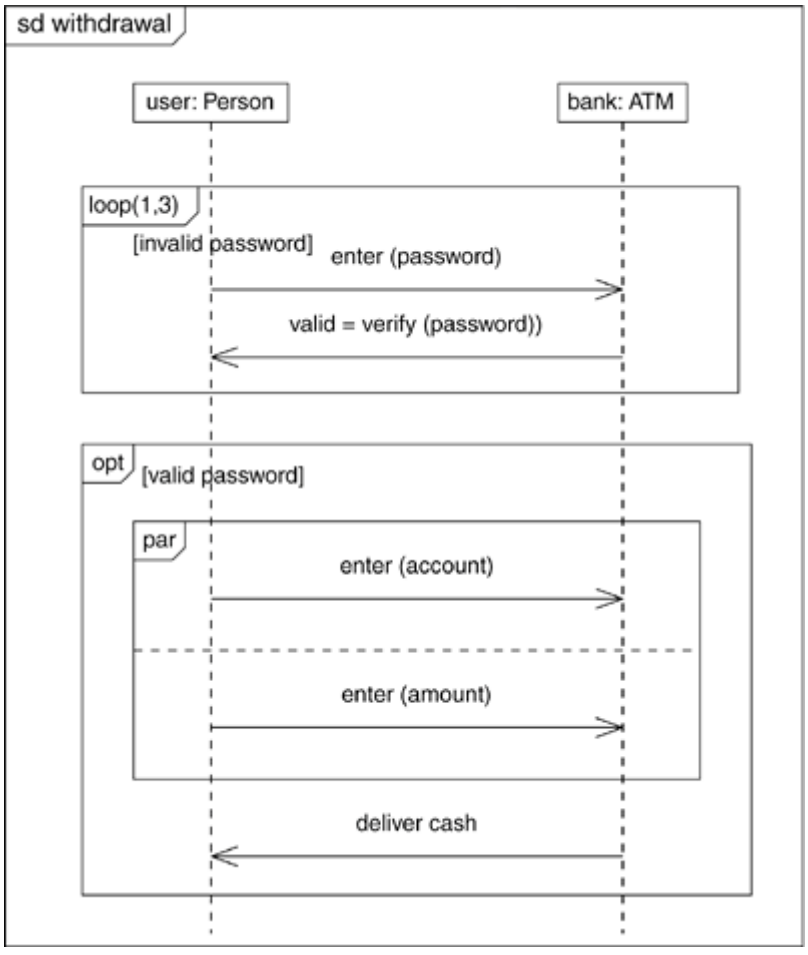
The tag is `loop`. A guard condition appears at the top of one lifeline within the body. The body of the loop is executed repeatedly as long as the guard condition is true before each iteration. When the guard condition is false at the top of the body, control passes out of the control operator.

There are many other kinds of operators, but these are the most useful.

To provide a clear indication of the boundary, a sequence diagram may be enclosed in a rectangle, with a tag in the upper left corner. The tag is `sd`, which may be followed by the name of the diagram.

Figure 19-3 shows a simplified example that illustrates some control operators. The user initiates the sequence. The first operator is a loop operator. The numbers in parentheses (1,3) indicate the minimum and maximum number of times the loop body must be executed. Since the minimum is one, the body is always executed at least once before the condition is tested. In the loop, the user enters the password and the system verifies it. The loop continues as long as the password is incorrect. However, after three tries, the loop terminates in any case.

Figure 19-3. Structured Control Operators



The next operator is an optional operator. The optional body is executed if the password is valid; otherwise the rest of the sequence diagram is skipped. The optional body contains a parallel operator. Operators can be nested as shown.

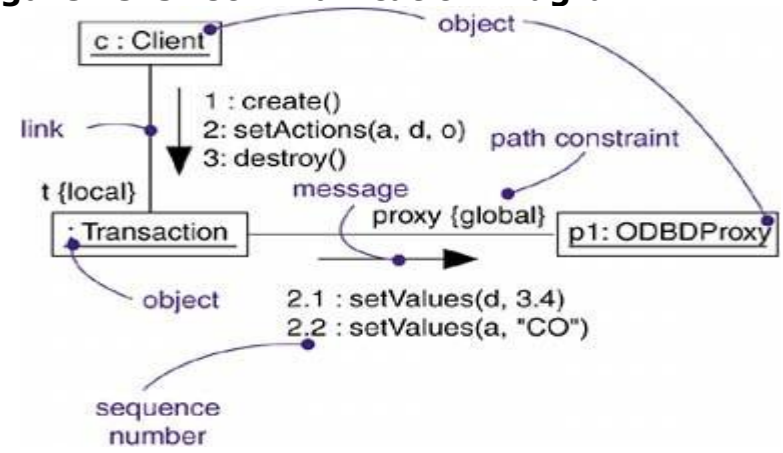
The parallel operator has two subregions: one allows the user to enter an account and the other allows the user to enter an amount. Because they are parallel, there is no required order for making the two entries; they can occur in either order. This emphasizes that concurrency does not always imply physically simultaneous execution. Concurrency really means that two actions are uncoordinated and can happen in any order. If they are truly independent actions, they can overlap; if they are sequential actions, they can occur in any order.

Once both actions have been performed, the parallel operator is complete. In the next action within the optional operator, the bank delivers cash to the user. The sequence diagram is now complete.

Communication Diagrams

A communication diagram emphasizes the organization of the objects that participate in an interaction. As Figure 19-5 shows, you form a communication diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. The links may have rolenames to identify them. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure 19-5. Communication Diagram



Communication diagrams have two features that distinguish them from sequence diagrams. First, there is the path. You render a path corresponding to an association. You also render paths corresponding to local variables, parameters, global variables, and self access. A path represents a source of knowledge to an object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence and communication diagrams, especially if the context of the diagram is an operation. For example, using the previous communication diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation `register`, attached to the `Student` class.

```
public void register() {
 CourseCollection courses = getSchedule();
 for (int i = 0; i < courses.size(); i++)
  courses.item(i).add(this);
```

points to examine the attribute values of individual objects. `this.registered = true;`

```
}
```

"Reasonably clever" means the tool would have to realize that `getSchedule` returns a `CourseCollection` object, which it could determine by looking at the operation's signature. By walking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence and communication diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by hand from a prototypical execution of the

`register` operation.

However, more interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the messages in the diagram as they were dispatched in a running system. Even better, with this tool under the control of a debugger, you could control the speed of execution, possibly setting breakpoints to stop the action at interesting

Use Cases

In this chapter

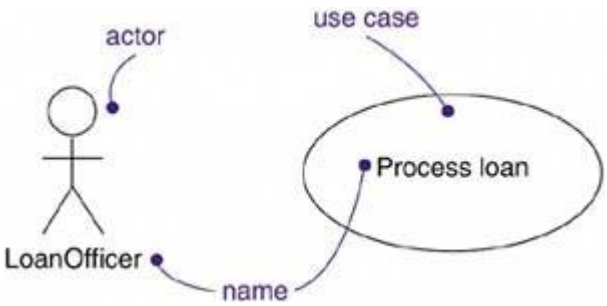
Well-structured use cases denote essential subject behaviors only, and are neither overly general nor too specific.

A use case involves the interaction of actors and the system or other subject. An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Actors can be human or they can be automated systems. For example, in modeling a bank, processing a loan involves, among other things, the interaction between a customer and a loan officer.

A use case carries out some tangible amount of work. From the perspective of a given actor, a use case does something that's of value to an actor, such as calculate a result, generate a new object, or change the state of another object. For example, in modeling a bank, processing a loan results in the delivery of an approved loan, manifest in a pile of money handed to the customer.

You can apply use cases to your whole system. You can also apply use cases to part of your system, including subsystems and even individual classes and interfaces. In each case, these use cases not only represent the desired behavior of these elements, but they can also be used as the basis of test cases for these elements as they evolve during development. Use cases applied to subsystems are excellent sources of regression tests; use cases applied to the whole system are excellent sources of integration and system tests. The UML provides a graphical representation of a use case and an actor, as Figure 17-1 shows. This notation permits you to visualize a use case apart from its realization and in context with other use cases.

Figure 17-1. Actor and Use Case



Terms and Concepts

A *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. Graphically, a use case is rendered as an ellipse.

Subject

The *subject* is a class described by a set of use cases. Usually the class is a system or subsystem. The use cases represent aspects of the behavior of the class. The actors represent aspects of other classes that interact with the subject. Taken together, the use cases describe the complete behavior of the subject.

Names

Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string. That name alone is known as a *simple name*; a *qualified name* is the use case name prefixed by the name of the package in which that use case lives. A use case is typically drawn showing only its name, as in Figure 17-2

Figure 17-2. Simple and Qualified Names

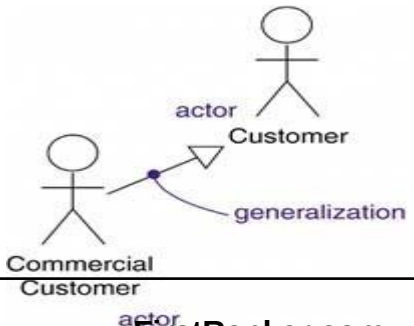
Use Cases and Actors

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. For example, if you work for a bank, you might be a `LoanOfficer`. If you do your personal banking there, as well, you'll also play the role of `Customer`. An instance of an actor, therefore, represents an individual interacting with the system in a specific way. Although you'll use actors in your models, actors are not actually part of the software application. They live outside the application within the surrounding environment.

In an executing system, actors need not exist as separate entities. One object may play the part of multiple actors. For example, one `Person` may be both a `LoanOfficer` and a `Customer`. An actor represents one aspect of an object.

As Figure 17-3 indicates, actors are rendered as stick figures. You can define general kinds of actors (such as `Customer`) and specialize them (such as `CommercialCustomer`) using generalization relationships.

Figure 17-3. Actors



Actors may be connected to use cases only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily. When you write this flow of events, you should 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.

For example, in the context of an ATM system, you might describe the use case `ValidateUser` in the following way:

> *Main flow of events*: The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.
>
> *Exceptional flow of events*: The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer*'s account.
>
> *Exceptional flow of events*: The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.
>
> *Exceptional flow of events*: If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

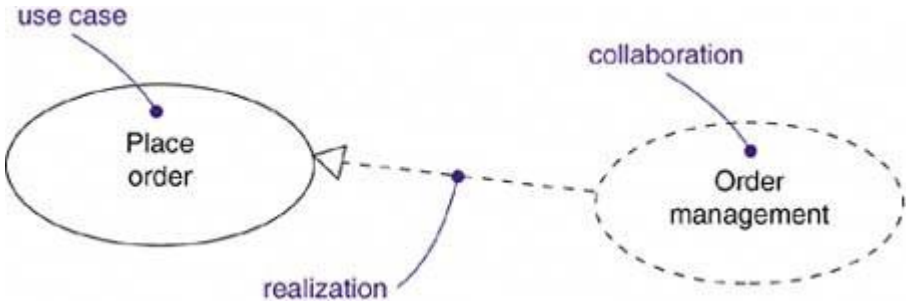
Use Cases and Scenarios

This one use case (`Hire employee`) actually describes a set of sequences in which each sequence in the set represents one possible flow through all these variations. Each sequence is called a scenario. A scenario is a specific sequence of actions that illustrates behavior. Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

A use case captures the intended behavior of the system (or subsystem, class, or interface) you developing, without having to specify how that behavior is implemented. That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out). Ultimately, however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration.
As Figure 17-4 shows, you can explicitly specify the realization of a use case by a collaboration. Most of the time, though, a given use case is realized by exactly one collaboration, so you will not need to model this relationship explicitly.

Figure 17-4. Use Cases and Collaborations

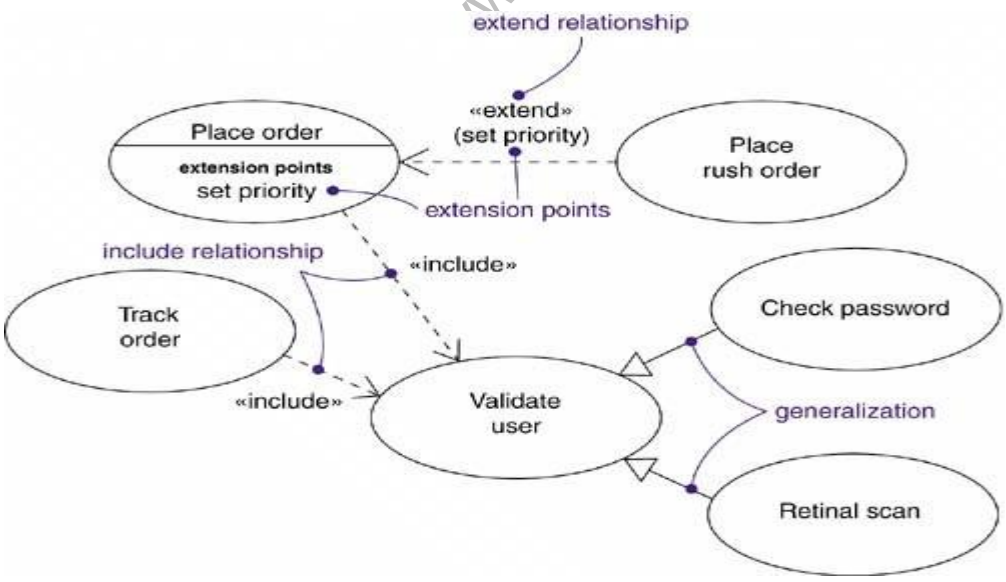


Organizing Use Cases

You can organize use cases by grouping them in packages in the same manner in which you can organize classes.
You can also organize use cases by specifying generalization, include, and extend relationships among them. You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).

Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances). For example, in a banking system, you might have the use case `Validate User`, which is responsible for verifying the identity of the user. You might then have two specialized children of this use case (`Check password` and `Retinal scan`), both of which behave just like `Validate User` and may be applied anywhere `Validate User` appears, yet both of which add their own behavior (the former by checking a textual password, the latter by checking the unique retina patterns of the user). As shown in Figure 17-5, generalization among use cases is rendered as a solid directed line with a large triangular arrowhead, just like generalization among classes.

Figure 17-5. Generalization, Include, and Extend



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. The base use case may stand alone, but under certain conditions its behavior may be extended by the behavior of another use case. This base use case may be extended only at certain points called, not surprisingly, its extension points. You can think of extend as the extension use case pushing behavior to the base use case.

You use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, you separate optional behavior from mandatory behavior. You may also use an extend relationship to model a separate subflow that is executed only under given conditions. Finally, you may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor. You may also use an extend relationship to distinguish configurable parts of an implementable system; the implication is that the system can exist with or without the various extensions.

You render an extend relationship as a dependency, stereotyped as `extend`. You may list the extension points of the base use case in an extra compartment. These extension points are just labels that may appear in the flow of the base use case. For example, the flow for `Place order` might read as follows:

Other Features

Use cases are classifiers, so they may have attributes and operations that you may render just as for classes. You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events. These objects and operations may be used in your interaction diagrams to specify the behavior of the use case.
As classifiers, you can also attach state machines to use cases. You can use state machines as yet another way to describe the behavior represented by a use case.

Use Case Diagrams

In this chapter

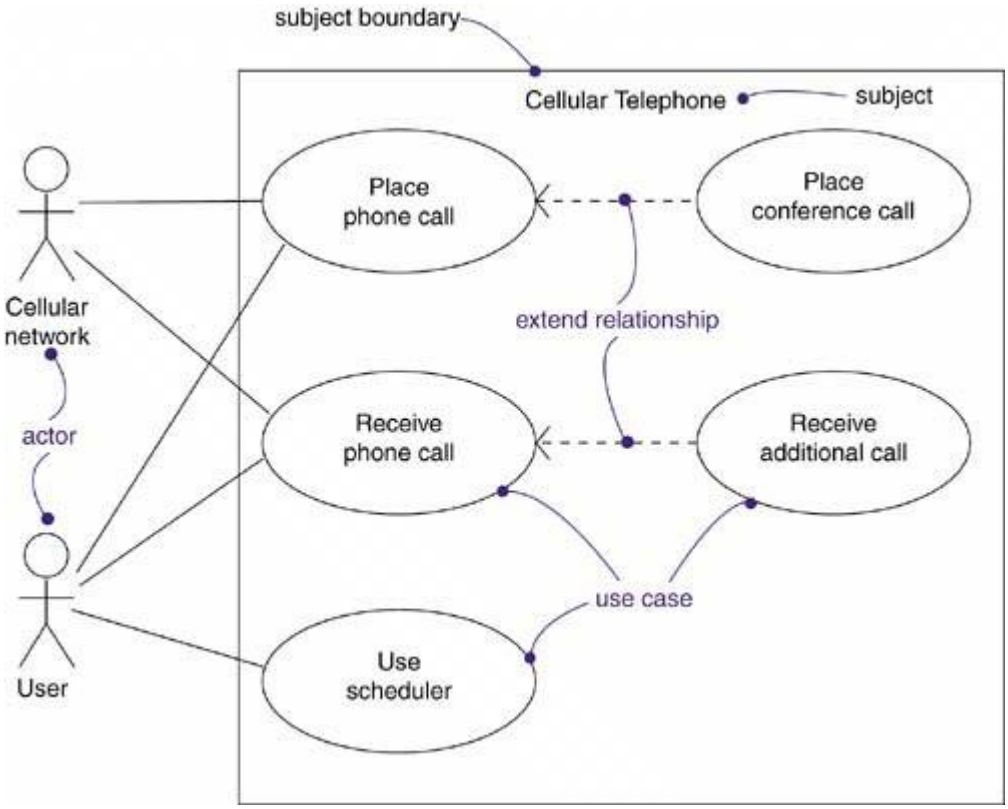
- Modeling the context of a system
- Modeling the requirements of a system
- Forward and reverse engineering

Use case diagrams are one of the diagrams in the UML for modeling the dynamic aspects of systems. (Activity diagrams, state diagrams, sequence diagrams, and communication diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems.) Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. Each one shows a set of use cases and actors and their relationships.

You apply use case diagrams to model the use case view of a system. For the most part, this involves modeling the context of a system, subsystem, or class, or modeling the requirements of the behavior of these elements.

Use case diagrams are important for visualizing, specifying, and documenting the behavior of an element. They make systems, subsystems, and classes approachable and understandable by presenting an outside view of how those elements may be used in context. Use case diagrams are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering.

Figure 18-1. A Use Case Diagram



Terms and Concepts

A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

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

Contents

Use case diagrams commonly contain

- Subject
- Use cases
- Actors
- Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints.

Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks. Occasionally, you'll want to place instances of use cases in your diagrams as well, especially when you want to visualize a specific executing system.

Notation

The subject is shown as a rectangle containing a set of use case ellipses. The name of the subject is

placed within the rectangle. The actors are shown as stick figures placed outside the rectangle; their names are placed under them. Lines connect actor icons to the use case ellipses with which they communicate. Relationships among use cases (such as extend and include) are drawn inside the rectangle.

Common Uses

You apply use case diagrams to model the use case view of a subject, such as a system. This view primarily models the external behavior of a subjectthe outwardly visible services that the subject provides in the context of its environment.
When you model the use case view of a subject, you'll typically apply use case diagrams in one of two ways.

1. To model the context of a subject

Modeling the context of a subject involves drawing a line around the whole system and asserting which actors lie outside the subject and interact with it. Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

2. To model the requirements of a subject

Modeling the requirements of a subject involves specifying what that subject should do (from a point of view of outside the subject) independent of how that subject should do it. Here, you'll apply use case diagrams to specify the desired behavior of the subject. In this manner, a use case diagram lets you view the whole subject as a black box; you can see what's outside the subject and you can see how that subject reacts to the things outside, but you can't see how that subject works on the inside.

Common Modeling Techniques

Modeling the Context of a System

Given a system-any system-some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that interact with the system constitute the system's context. This context defines the environment in which that system lives.

In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system. Deciding what to include as an actor is important because in doing so you specify a class of things that interact with the system. Deciding what not to include as an actor is equally, if not more, important because that constrains the system's environment to include only those actors that are necessary in the life of the system.

To model the context of a system,

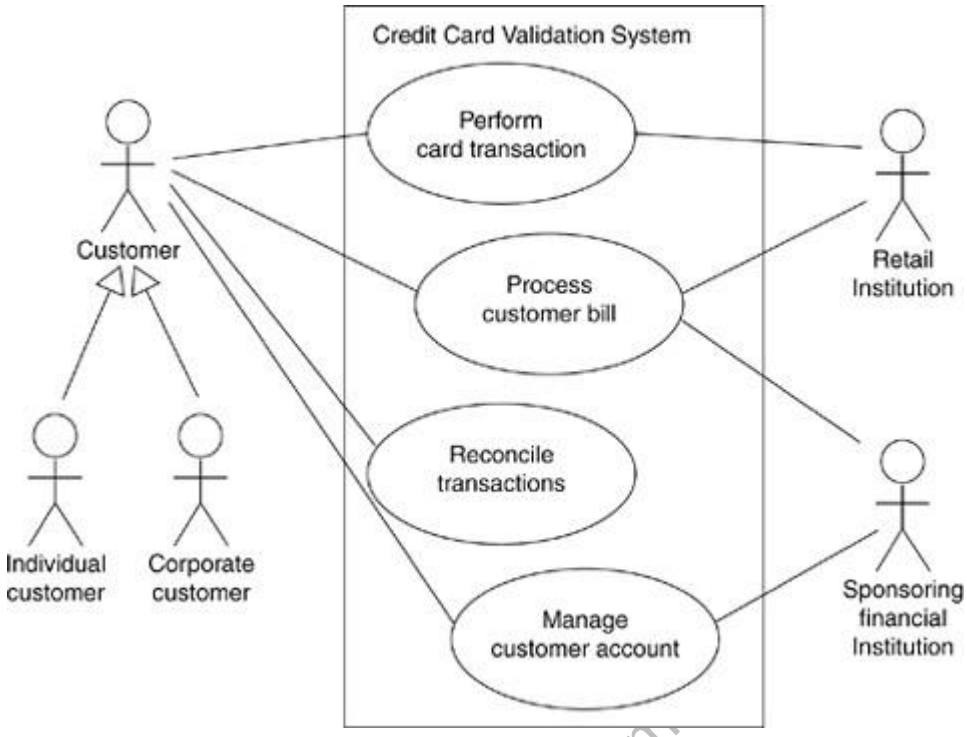
- Identify the boundaries of the system by deciding which behaviors are part of it and which are performed by external entities. This defines the subject.
- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks, which groups are needed 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.
- Where it aids understandability, 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.

For example, Figure 18-2 shows the context of a credit card validation system, with an emphasis on the actors that surround the system. You'll find `Customers`, of which there are two kinds (`Individual`

`customer` and `Corporate customer`). These actors are the roles that humans play when interacting with the system. In this context, there are also actors that represent other institutions, such as `Retail institution` (with which a `Customer` performs a card transaction to buy an item or a service) and `Sponsoring financial institution` (which serves as the clearinghouse for the credit card account). In the real world, these latter two actors are likely software-intensive systems themselves.

Figure 18-2. Modeling the Context of a System



This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

Modeling the Requirements of a System

A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do. For the most part, you don't care how the system does it, you just care *that* it does it. A well-behaved system will carry out all its requirements faithfully, predictably, and reliably. When you build a system, it's important to start with agreement about what that system should do, although you will certainly evolve your understanding of those requirements as you iteratively and incrementally implement the system. Similarly, when you are handed a system to use, knowing how it behaves is essential to using it properly.

Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between. Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system,

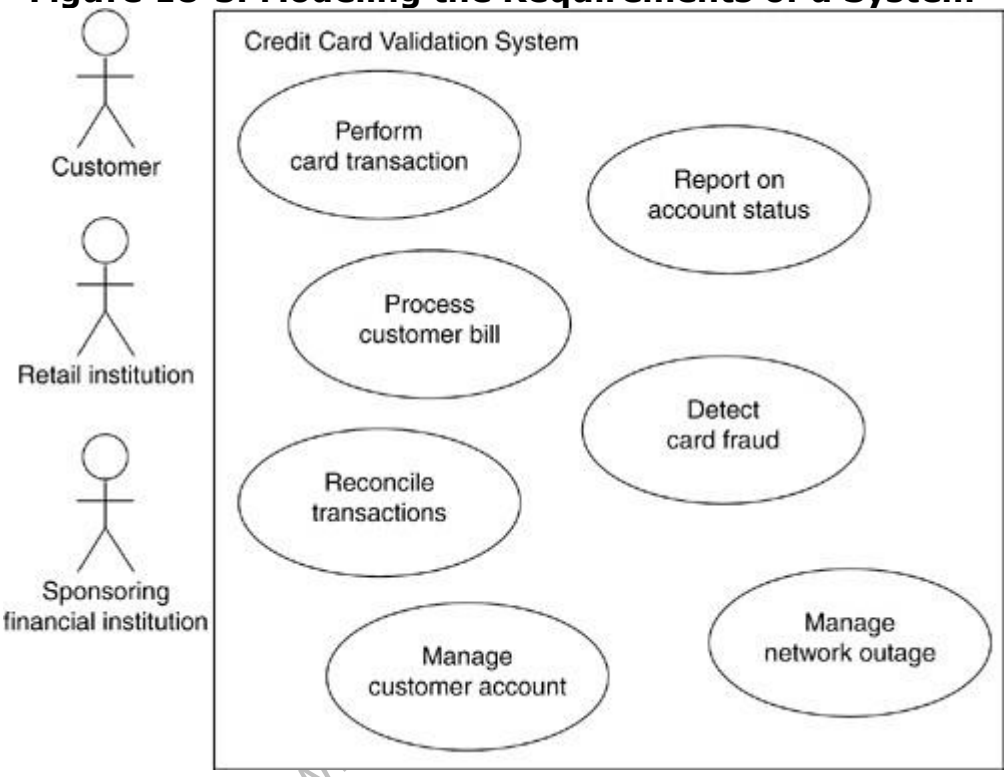
- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as usecases.
- Factor common behavior into new use cases that are used by others; factor variant 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 or constraints that assert nonfunctional requirements; you may have to attach some of these to the whole system.

Figure 18-3 expands on the previous use case diagram. Although it elides the relationships among the actors and the use cases, it adds additional use cases that are somewhat invisible to the average customer yet are essential behaviors of the system. This diagram is valuable because it offers a common starting place for end users, domain experts, and developers to visualize, specify, construct, and document their decisions about the functional requirements of this system. For example, `Detect card fraud` is a behavior important to both the `Retail institution` and the `Sponsoring financial institution`. Similarly, `Report on account status` is another behavior required of the system by the various institutions in its context.

Figure 18-3. Modeling the Requirements of a System



Common Modeling Techniques

Modeling the Behavior of an Element

The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class. When you model the behavior of these things, it's important that you focus on what that element does, not how it does it.

To model the behavior of an element,

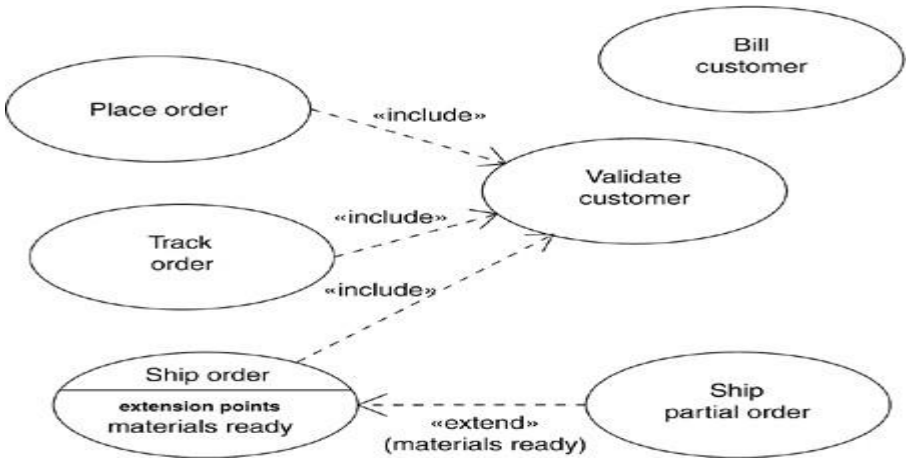
- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
- Organize actors by identifying general and more specialized roles.
- For each actor, 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 relationships to factor common behavior and distinguish exceptional behavior.

For example, a retail system will interact with customers who place and track orders. In turn, the system will ship orders and bill the customer. As Figure 17-6 shows, you can model the behavior of such a system by declaring these behaviors as use cases (`Place order`, `track order`, `Ship order`, and `Bill customer`). Common behavior can be factored out (`Validate custome`r) and variants (`Shipartial order`) can be distinguished as well. For each of these use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

Figure 17-6. Modeling the Behavior of an Element



As your models get bigger, you will find that many use cases tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of classes.

The requirement modeled by the use case `Manage network outage` is a bit different from all the others because it represents a secondary behavior of the system necessary for its reliable and continuous operation.

Once the structure of the use case is determined, you must describe the behavior of each use case. Usually you should write one or more sequence diagrams for each mainline case. Then you should write sequence diagrams for variant cases. Finally, you should write at least one sequence diagram to illustrate each kind of error or exception condition; error handling is part of the use case and should be planned along with the normal behavior.

This same technique applies to modeling the requirements of a subsystem.

Forward and Reverse Engineering

Most of the UML's other diagrams, including class, component, and state diagrams, are clear candidates for forward and reverse engineering because each has an analog in the executable system. Use case diagrams are a bit different in that they reflect rather than specify the implementation of a system, subsystem, or class. Use cases describe how an element behaves, not how that behavior is implemented, so it cannot be directly forward or reverse engineered.

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. A use case diagram can be forward engineered to form tests for the element to which it applies. Each use case in a use case diagram specifies a flow of events (and variants of those flows), and these flows specify how the element is expected to behave-that's something worthy of testing. A well-structured use case will even specify pre- and postconditions that can be used to define a test's initial state and its success criteria. For each use case in a use case diagram, you can create a test case that you can run every time you release a new version of that

element, thereby confirming that it works as required before other elements rely on it.

To forward engineer a use case diagram,

- Identify the objects that interact with the system. Try to identify the various roles that each external object may play.
- Make up an actor to represent each distinct interaction role.
- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
- Use tools to run these tests each time you release the element to which the use case diagram applies.

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. To automatically reverse engineer a use case diagram is currently beyond the state of the art, simply because there is a loss of information when moving from a specification of how an element behaves to how it is implemented. However, you can study an existing system and discern its intended behavior by hand, which you can then put in the form of a use case diagram. Indeed, this is pretty much what you have to do anytime you are handed an undocumented body of software. The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
- Render these actors and use cases in a use case diagram, and establish their relationships.

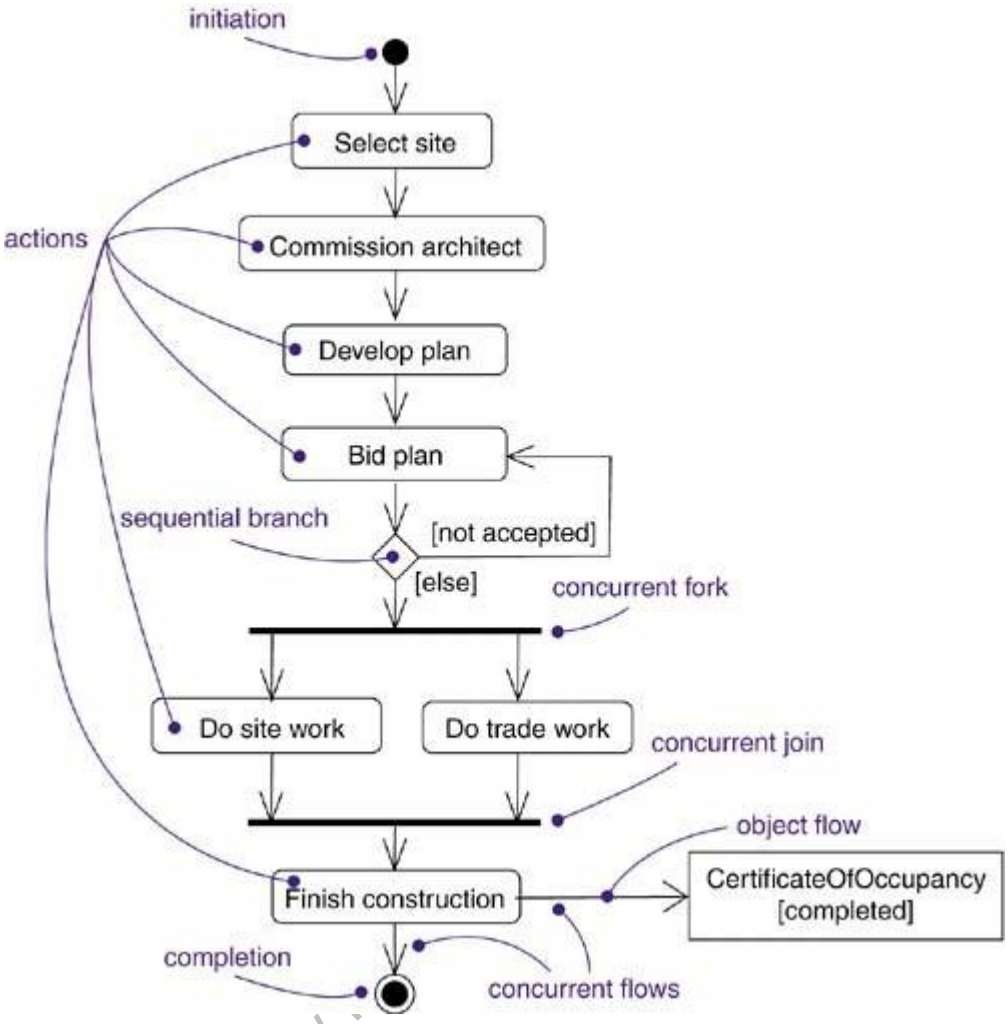
Activity Diagrams

In this chapter

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity. Unlike a traditional flowchart, an activity diagram shows concurrency as well as branches of control.

Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Figure 20-1. Activity Diagram



Terms and Concepts

An *activity diagram* shows the flow from activity to activity. An *activity* is an ongoing nonatomic execution within a state machine. The execution of an activity ultimately expands into the execution of individual *actions*, each of which may change the state of the system or communicate messages. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation such as evaluating an expression. Graphically, an activity diagram is a collection of nodes and arcs.

Common Properties

An activity diagram is a kind of diagram and shares the same common properties as do all other diagramsa name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from other kinds of diagrams is its content.

Activity diagrams commonly contain

- Actions
- Activity nodes
- Flows
- Object values

Like all other diagrams, activity diagrams may contain notes and constraints.

Actions and Activity Nodes

In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object. These executable, atomic computations are called actions. As Figure 20-2 shows, you represent an action using a rounded box. Inside that shape, you may write an expression.

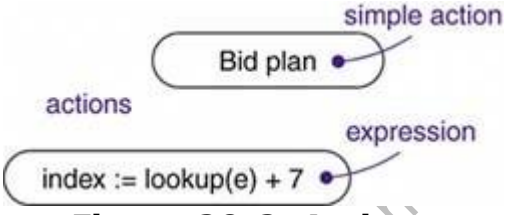


Figure 20-2. Actions

Actions can't be decomposed. Furthermore, actions are atomic, meaning that events may occur, but the internal behavior of the action state is not visible. You can't execute part of an action; either it executes completely or not at all. Finally, the work of an action state is often considered to take insignificant execution time, but some actions may have substantial duration.

An *activity node* is an organizational unit within an activity. In general, activity nodes are nested groupings of actions or other nested activity nodes. Furthermore, activity nodes have visible substructure; in general, they are considered to take some duration to complete. You can think of an action as a special case of an activity node. An action is an activity node that cannot be further decomposed. Similarly, you can think of an activity node as a composite whose flow of control is made up of other activity nodes and actions. Zoom into the details of an activity node and you'll find another activity diagram. As Figure 20-3 shows, there's no notational distinction between actions and activity nodes, except that an activity node may have additional parts, which will usually be maintained in the background by an editing tool.

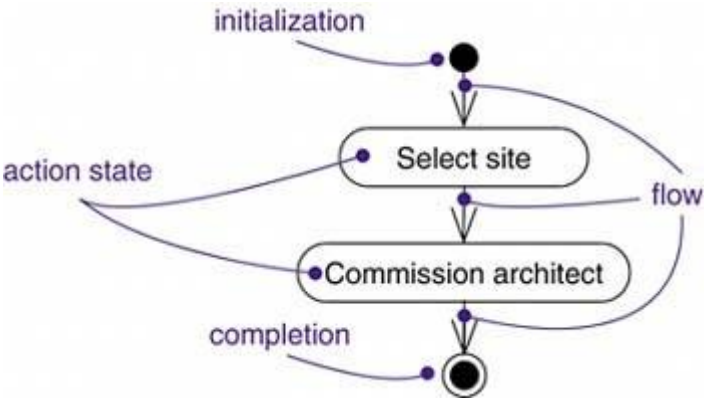
Figure 20-3. Activity Nodes

Control Flows

When an action or activity node completes execution, flow of control passes immediately to the next

action or activity node. You specify this flow by using flow arrows to show the path of control from one action or activity node to the next action or activity node. In the UML, you represent a flow as a simple arrow from the predecessor action to its successor, without an event label, as Figure 20-4 shows.

Figure 20-4. Completion Transitions

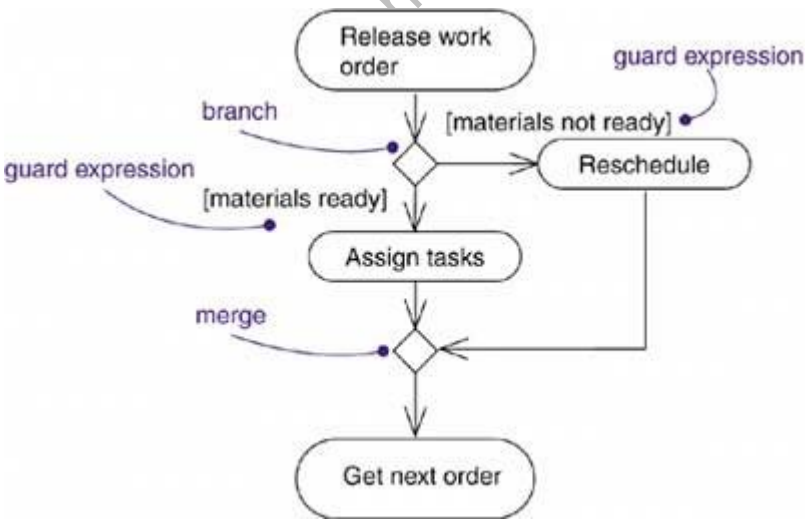


Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in which case it will have a beginning but no end). Therefore, as the figure shows, you may specify initialization (a solid ball) and completion (a solid ball inside a circle) as special symbols.

Branching

Simple, sequential flows are common, but they aren't the only kind of path you'll need to model a flow of control. As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression. As Figure 20-5 shows, you represent a branch as a diamond. A branch may have one incoming and two or more outgoing flows. On each outgoing flow, you place a Boolean expression, which is evaluated on entering the branch. The guards on the outgoing flows should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).

Figure 20-5. Branching



As a convenience, you can use the keyword `else` to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.

When two paths of control merge back together, you can also use a diamond symbol with two input arrows and one output arrow. No guards are necessary on merge.
You can achieve the effect of iteration by using one action that sets the value of an iterator, another action that increments the iterator, and a branch that evaluates if the iteration is finished. The UML includes node types for loops, but these may often be expressed more easily in text than in graphics.

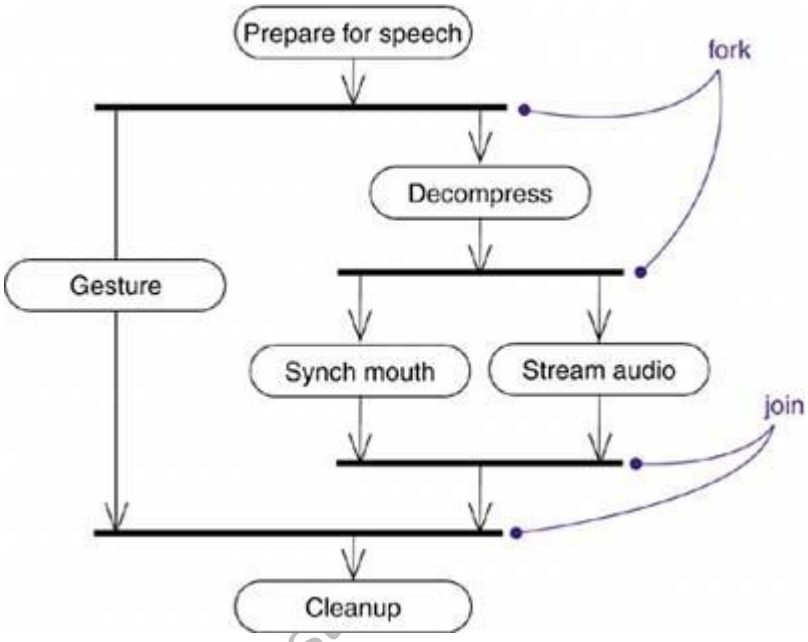
Forking and Joining

Simple and branching sequential transitions are the most common paths you'll find in activity

diagrams. Howeverespecially when you are modeling workflows of business processesyou might encounter flows that are concurrent. In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures. As Figure 20-6 shows, a fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continues in parallel. Conceptually, the activities of each of these flows are truly parallel, although, in a running system, these flows may be either truly concurrent (in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.

Figure 20-6. Forking and Joining

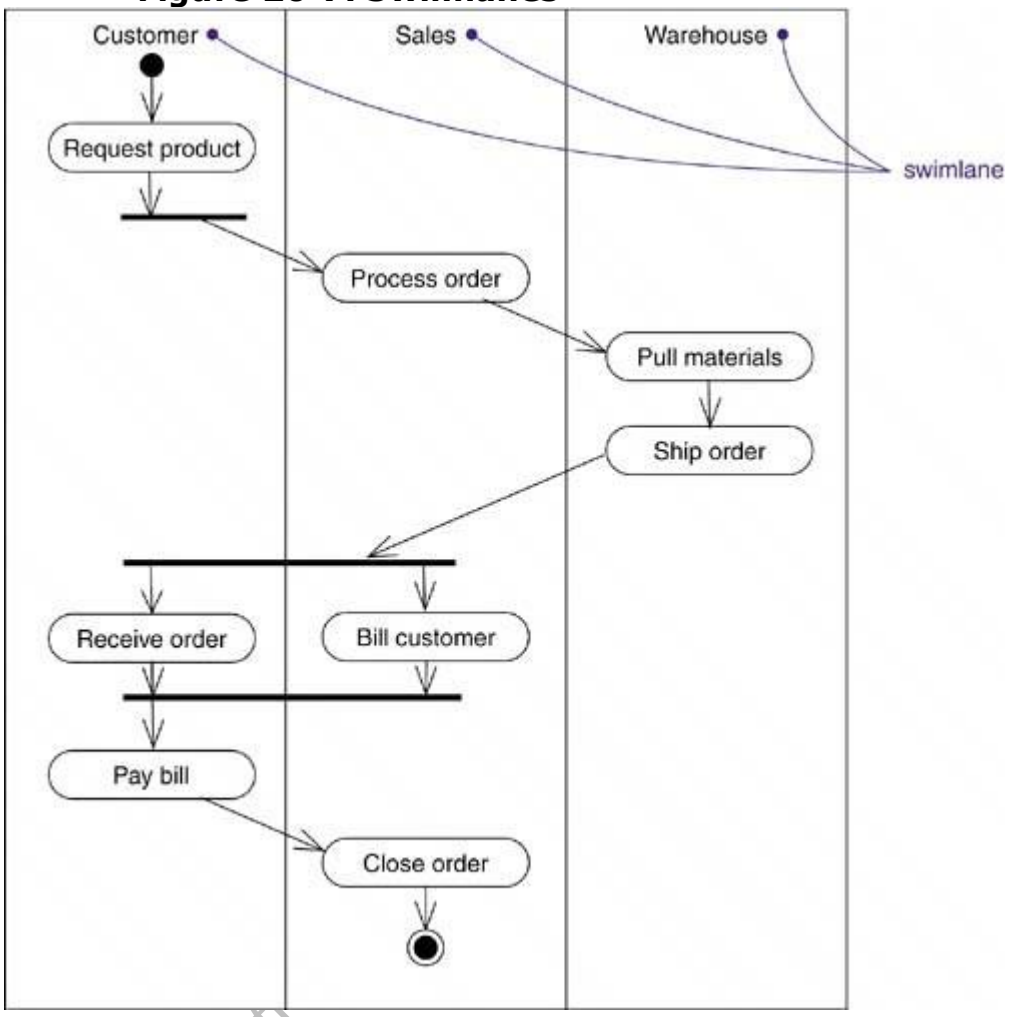


As the figure also shows, a join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continues in parallel. At the join, the concurrent 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.

Swimlanes

You'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities. In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line, as shown in Figure 20-7. A swimlane specifies a set of activities that share some organizational property.

Figure 20-7. Swimlanes



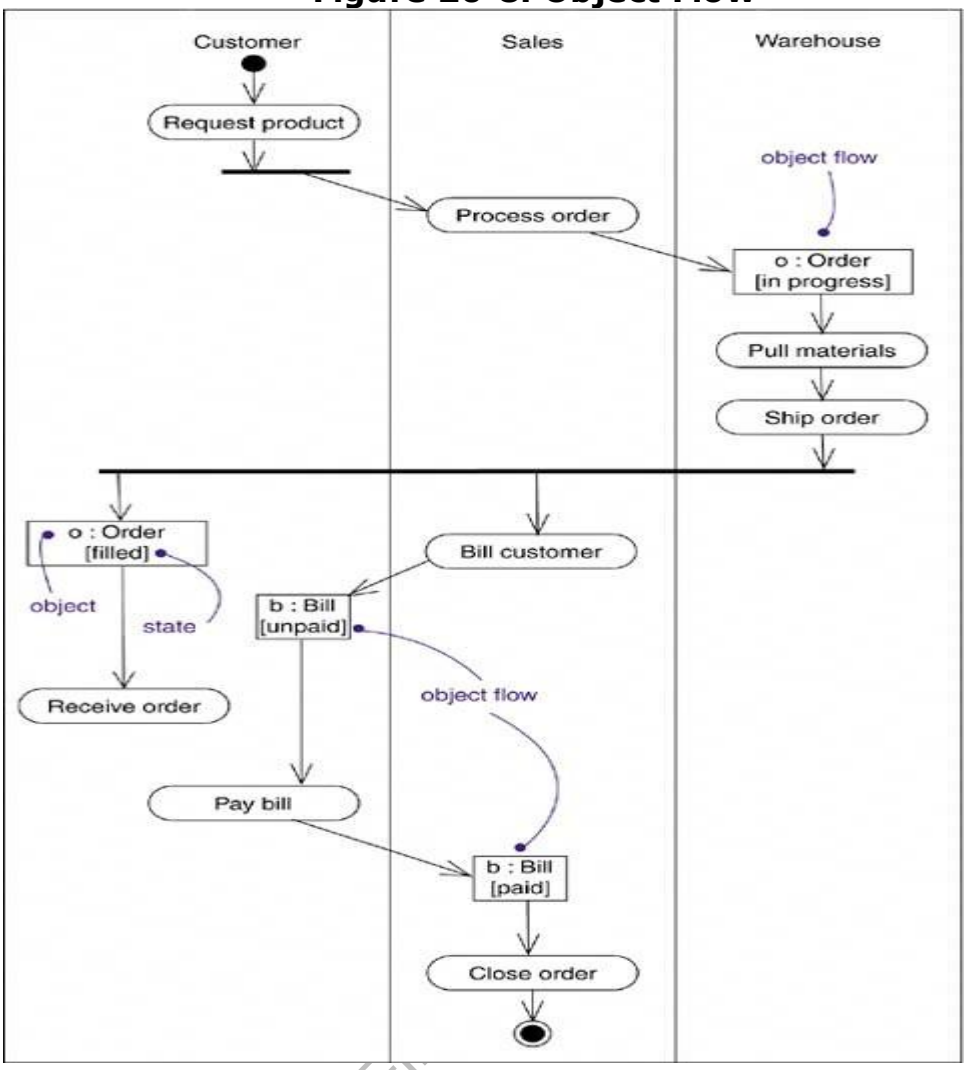
Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity, such as an organizational unit of a company. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

Object Flow

Objects may be involved in the flow of control associated with an activity diagram. For example, in the workflow of processing an order as in the previous figure, the vocabulary of your problem space will also include such classes as `Order` and `Bill`. Instances of these two classes will be produced by certain activities (`Process order` will create an `Order` object, for example); other activities may use or modify these objects (for example, `Ship order` will change the state of the `Order` object to `filled`).

As Figure 20-8 shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected by arrows to the actions that produce or consume them.

Figure 20-8. Object Flow



This called an object flow because it represents the flow of an object value from one action to another. An object flow inherently implies control flow (you can't execute an action that requires a value without the value!), so it is unnecessary to draw a control flow between actions connected by object flows.

In addition to showing the flow of an object through an activity diagram, you can also show how its state changes. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name.

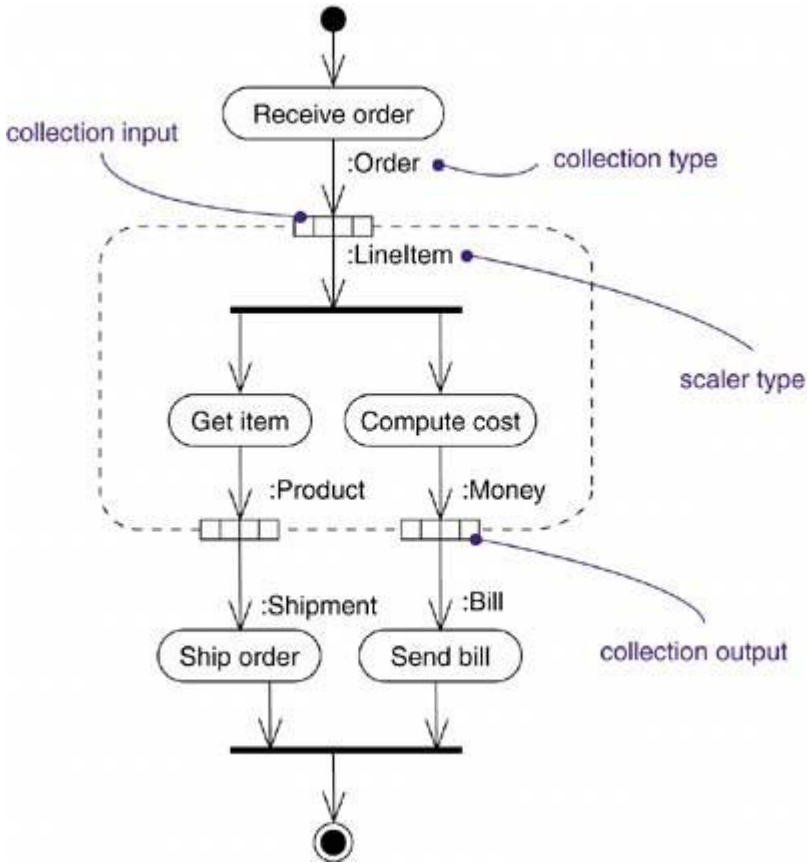
Expansion Regions

An expansion region represents a activity model fragment that is performed on the elements of a list or set. It is shown in an activity diagram by drawing a dashed line around a region in the diagram. The inputs to the region and the outputs from the region are collections of values, such as the line items in an order. Collection inputs and outputs are shown as a row of small squares joined together (to suggest an array of values). When an array value arrives at a collection input on an expansion region from the rest of the activity model, it is broken apart into the individual values. The execution region is executed once for each element in the array. It is unnecessary to model the iteration; it is implicit in the expansion region. The different executions can be performed concurrently, if possible. When each execution of the expansion region completes, its output value (if any) is placed into an output array in the same order as the corresponding input. In other words, an expansion region performs a "forall" operation on the elements of an array to create a new array.

Expansion regions allow operations on collections and operations on individual elements of the collections to be shown on the same diagram, without the need to show all of the detailed but straightforward iteration machinery.

Figure 20-9 shows an example of an expansion region. In the main body of the diagram, an order is received. This produces a value of type `Order`, which consists of an array of `LineItem` values. The `Order` value is the input to an expansion region. Each execution of the expansion region works on one element from the `Order` collection. Therefore, inside the region the type of input value corresponds to one element of the `Order` array, namely a `LineItem`. The expansion region activity forks into two actions: one action finds the `Product` and adds it to the shipment, and the other action computes the cost of that item. It is not necessary that the `LineItems` be taken in order; the different executions of the expansion region can proceed concurrently. When all executions of the expansion region are complete, the Items are formed into a `Shipment` (a collection of `Products`) and the charges are formed into a `Bill` (a collection of `Money` values). The `Shipment` value is the input to the `ShipOrder` action and the `Bill` value is the input to the `SendBill` action.

Figure 20-9. Expansion region



Common Uses

You use activity diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use an activity diagram to model some dynamic aspect of a system, you can do so in the context of virtually any modeling element. Typically, however, you'll use activity diagrams in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach activity diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you'll typically use activity diagrams in two ways.

1. To model a workflow

Here you'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity

diagrams, modeling object flow is particularly important.

2. To model an operation

Here you'll use activity diagrams as flowcharts to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the operation and its local objects.

Common Modeling Techniques

Modeling a Workflow

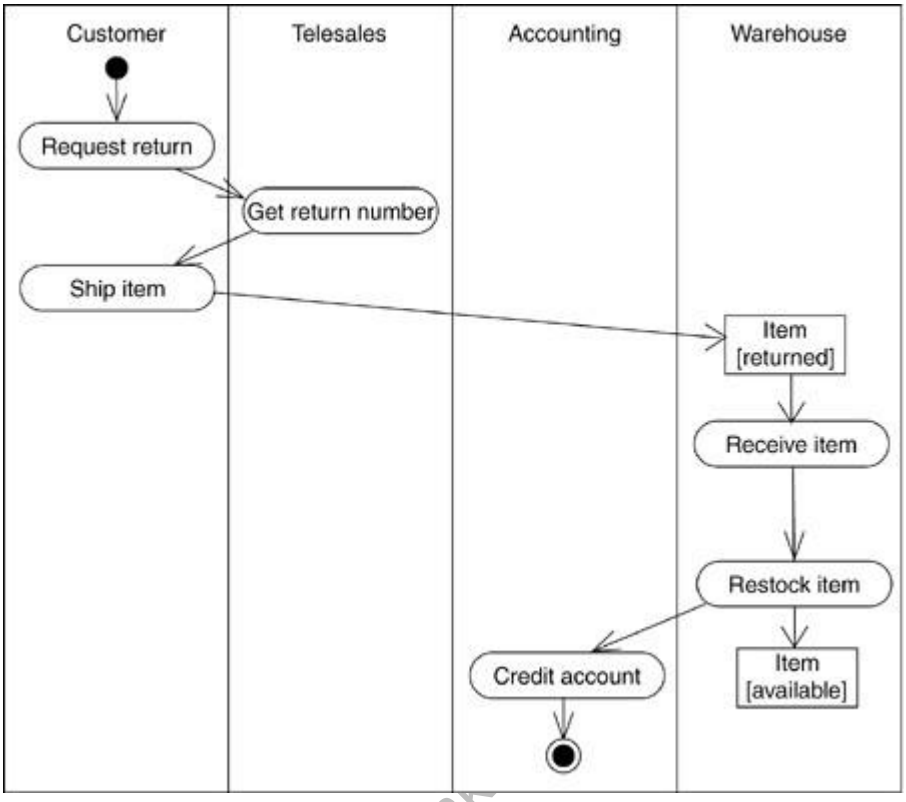
No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system. Especially for mission-critical enterprise software, you'll find automated systems working in the context of higher-level business processes. These business processes are kinds of workflows because they represent the flow of work and objects through the business. For example, in a retail business, you'll have some automated systems (for example, point-of-sale systems that interact with marketing and warehouse systems), as well as human systems (the people that work at each retail outlet, as well as the telesales, marketing, buying, and shipping departments). You can model the business processes for the way these various automated and human systems collaborate by using activity diagrams.

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 or organization.
- 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 actions that take place over time and render them in the activity diagram.
- For complicated actions or for sets of actions that appear multiple times, collapse these into calls to a separate activity diagram.
- Render the flows that connect these actions and activity nodes. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important object values 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 20-10 shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order. Work starts with the `Customer` action `Request return` and then flows through `Telesales` (`Get return number`), back to the `Customer` (`Ship item`), then to the `Warehouse` (`Receive item` then `Restock item`), finally ending in `Accounting` (`Credit account`). As the diagram indicates, one significant object (an instance of `Item`) also flows the process, changing from the `returned` to the `available` state.

Figure 20-10. Modeling a Workflow



In this example, there are no branches, forks, or joins. You'll encounter these features in more complex workflows.

Modeling an Operation

An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior. You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations. The most common element to which you'll attach an activity diagram is an operation.

Used in this manner, an activity diagram is simply a flowchart of an operation's actions. An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model. For example, any other operation or signal that an action state references can be type-checked against the class of the target object.

To model an operation,

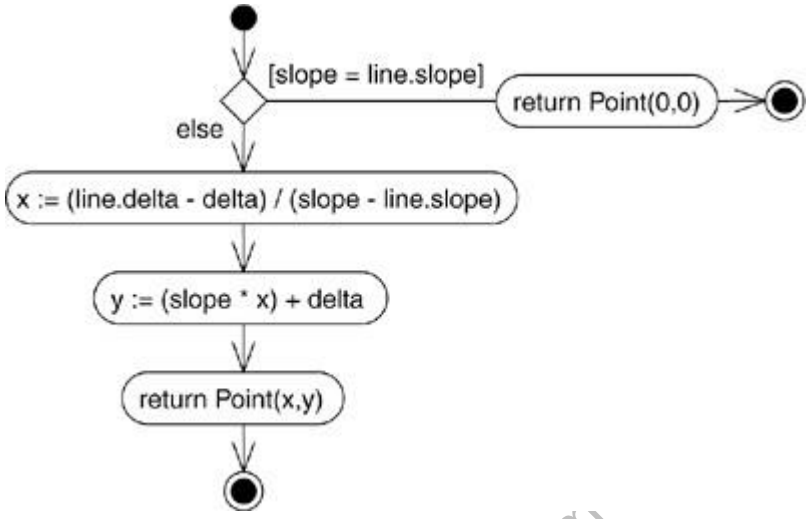
- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the postconditions at 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.

For example, in the context of the class `Line`, Figure 20-11 shows an activity diagram that specifies the algorithm of the operation `intersection`, whose signature includes one parameter (`line`, of the class `Line`) and one return value (of the class `Point`). The class `Line` has two attributes of interest: `slope` (which holds the slope of the line) and `delta` (which holds the offset of the line relative to the origin).

Figure 20-11. Modeling an Operation



The algorithm of this operation is simple, as shown in the following activity diagram. First, there's a guard that tests whether the `slope` of the current line is the same as the `slope` of parameter `line`. If so, the lines do not intersect, and a `Point` at `(0,0)` is returned. Otherwise, the operation first calculates an `x` value for the point of intersection, then a `y` value; `x` and `y` are both objects local to the operation. Finally, a `Point` at `(x,y)` is returned.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation. For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation `intersection`.

```
Point Line::intersection (line : Line) {
 if (slope == line.slope) return Point(0,0);
 int x = (line.delta - delta) /
                 (slope - line.slope);
 int y = (slope * x) + delta;
 return Point(x, y);
}
```

There's a bit of cleverness here, involving the declaration of the two local variables. A less-sophisticated tool might have first declared the two variables and then set their values.

Reverse engineering (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation. In particular, the previous diagram could have been generated from the implementation of the class `Line`.

More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the action states in the diagram as they were dispatched in a running system. Even better, with this tool also under the control of a debugger, you could control the speed of execution, possibly

setting breakpoints to stop the action at interesting points in time to examine the attribute values of individual objects.

Common Modeling Techniques

Modeling the Behavior of an Element

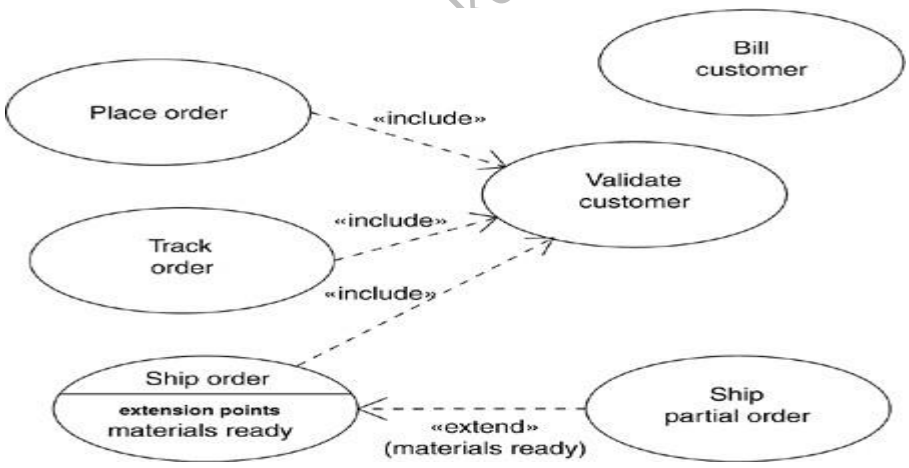
The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class. When you model the behavior of these things, it's important that you focus on what that element does, not how it does it.

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 relationships to factor common behavior and distinguish exceptional behavior.

For example, a retail system will interact with customers who place and track orders. In turn, the system will ship orders and bill the customer. As Figure 17-6 shows, you can model the behavior of such a system by declaring these behaviors as use cases (`Place order`, `track order`, `Ship order`, and `Bill customer`). Common behavior can be factored out (`Validate custome`r) and variants (`Shipartial order`) can be distinguished as well. For each of these use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

Figure 17-6. Modeling the Behavior of an Element



As your models get bigger, you will find that many use cases tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of classes.

5.Events and Signals

In this chapter

- Signal events, call events, time events, and change events
- Modeling a family of signals
- Modeling exceptions
- Handling events in active and passive objects

In the real world, things happen. Not only do things happen, but lots of things may happen at the same time, and at the most unexpected times. "Things that happen" are called events, and each one represents the specification of a significant occurrence that has a location in time and space.

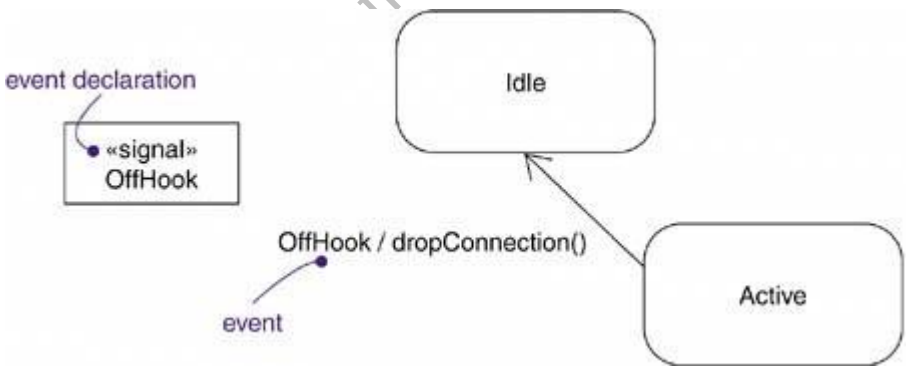
In the context of state machines, you use events to model the occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

Events may be synchronous or asynchronous, so modeling events is wrapped up in the modeling of processes and threads.

In the UML, each thing that happens is modeled as an event. An event is the specification of a significant occurrence that has a location in time and space. A signal, the passing of time, and a change of state are asynchronous events, representing events that can happen at arbitrary times. Calls are generally synchronous events, representing the invocation of an operation.

The UML provides a graphical representation of an event, as Figure 21-1 shows. This notation permits you to visualize the declaration of events (such as the signal `OffHook`) as well as the use of events to trigger a state transition (such as the signal `OffHook`, which causes a transition from the `Active` to the `Idle` state as well as the execution of the `dropConnection` action).

Figure 21-1. Events



Terms and Concepts

An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *signal* is a kind of event that represents the specification of an asynchronous message communicated between instances.

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and a message from a collision sensor are both

examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.
In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

Signals

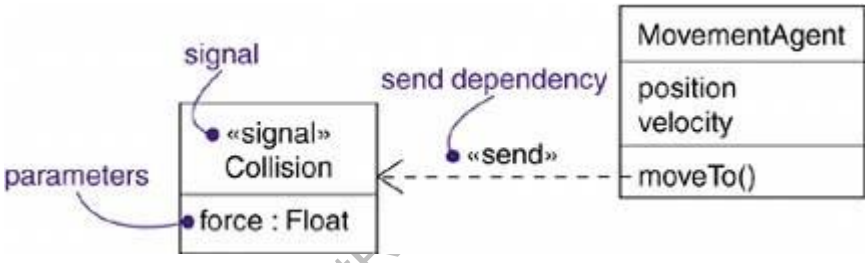
A message is a named object that is sent asynchronously by one object and then received by another. A signal is a classifier for messages; it is a message type.

Signals have a lot in common with plain classes. For example, signals may have instances, although you don't generally need to model them explicitly. Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general (for example, the signal `NetworkFailure`) and some of which are specific (for example, a specialization of `NetworkFailure` called `WarehouseServerFailure`). Also as for classes, signals may have attributes and operations. Before it has been sent by one object or after it is received by another, a signal is just an ordinary data object.

A signal may be sent by the action of a transition in a state machine. It may be modeled as a message between two roles in an interaction. The execution of a method can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send.

In the UML, as Figure 21-2 shows, you model signals as stereotyped classes. You can use a dependency, stereotyped as `send`, to indicate that an operation sends a particular signal.

Figure 21-2. Signals



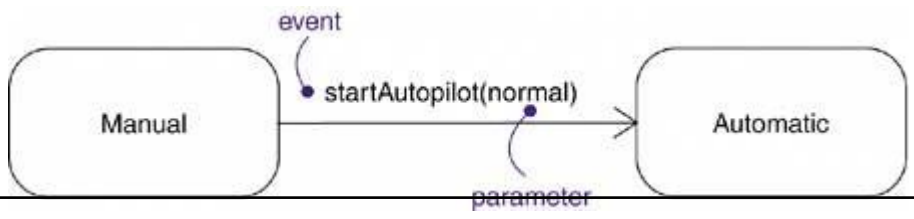
Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the receipt by an object of a call request for an operation on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object. The choice is specified in the class definition for the operation.

Whereas a signal is an asynchronous event, a call event is usually synchronous. This means that when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender. In those cases where the caller does not need to wait for a response, a call can be specified as asynchronous.

As Figure 21-3 shows, modeling a call event is indistinguishable from modeling a signal event. In both cases, you show the event, along with its parameters, as the trigger for a state transition.

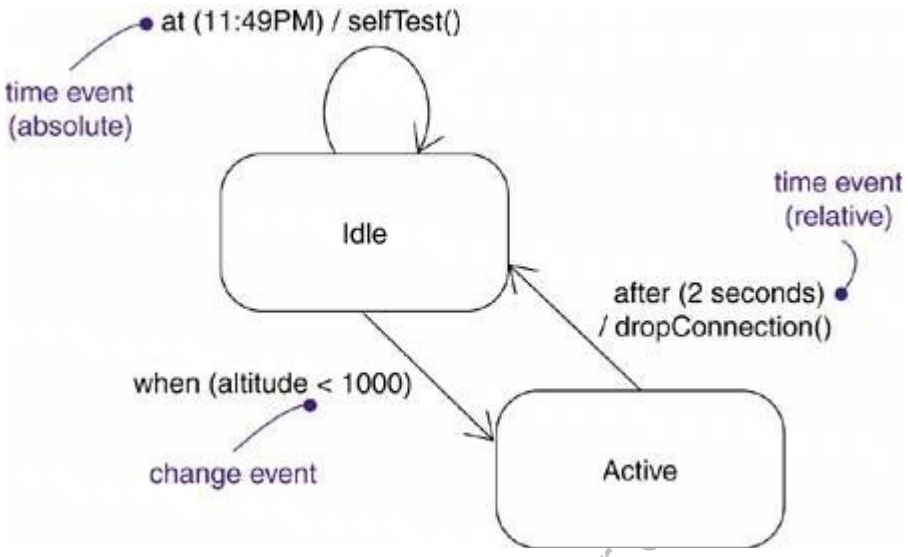
Figure 21-3. Call Events



Time and Change Events

A time event is an event that represents the passage of time. As Figure 21-4 shows, in the UML you model a time event by using the keyword `after` followed by some expression that evaluates to a period of time. Such expressions can be simple (for example, `after 2 seconds`) or complex (for example, `after 1 ms since exiting Idle`). Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state. To indicate a time event that occurs at an absolute time, use the keyword `at`. For example, the time event `at (1 Jan 2005, 1200 UT)` specifies an event that occurs on noon Universal Time on New Year's Day 2005.

Figure 21-4. Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure 21-4 shows, in the UML you model a change event by using the keyword `when` followed by some Boolean expression. You can use such expressions for the continuous test of an expression (for example, `when altitude < 1000`).

A change event occurs once when the value of the condition changes from false to true. It does not occur when the value of the condition changes from true to false. The event does not recur while the event remains true.

Sending and Receiving Events

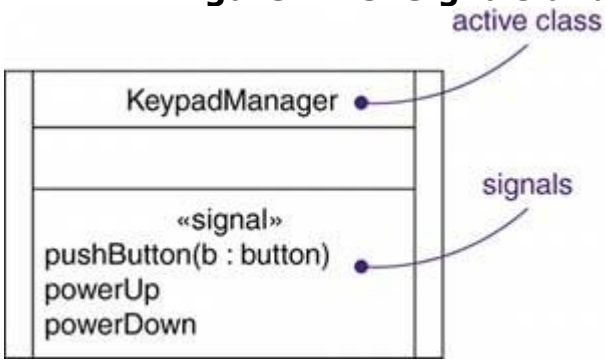
Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.

Any instance of any class can send a signal to or invoke an operation of a receiving object. 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. For example, if an actor interacting with an ATM system sends the signal `pushButton`, the actor may continue along its way independent of the system to which the signal was sent. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for the receiver to reply. For example, in a trading system, an instance of the class `trader` might invoke the operation `confirmTransaction` on some instance of the class `TRade`, thereby affecting the state of the `TRade` object. If this is a synchronous call, the `trader` object will wait until the operation is finished.

Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then

the sender and the receiver are in a rendezvous for the duration of the operation. This means that the flow of control of the sender suspends until the execution of the operation completes. If this is a signal, then the sender and receiver do not rendezvous: The sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost (if no response to the event is specified), it may trigger the receiver's state machine (if there is one), or it may just invoke a normal method call.
In the UML, you model the call events that an object may receive as operations on the class of the object. In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in Figure 21-5.

Figure 21-5. Signals and Active Classes



Common Modeling Techniques

Modeling a Family of Signals

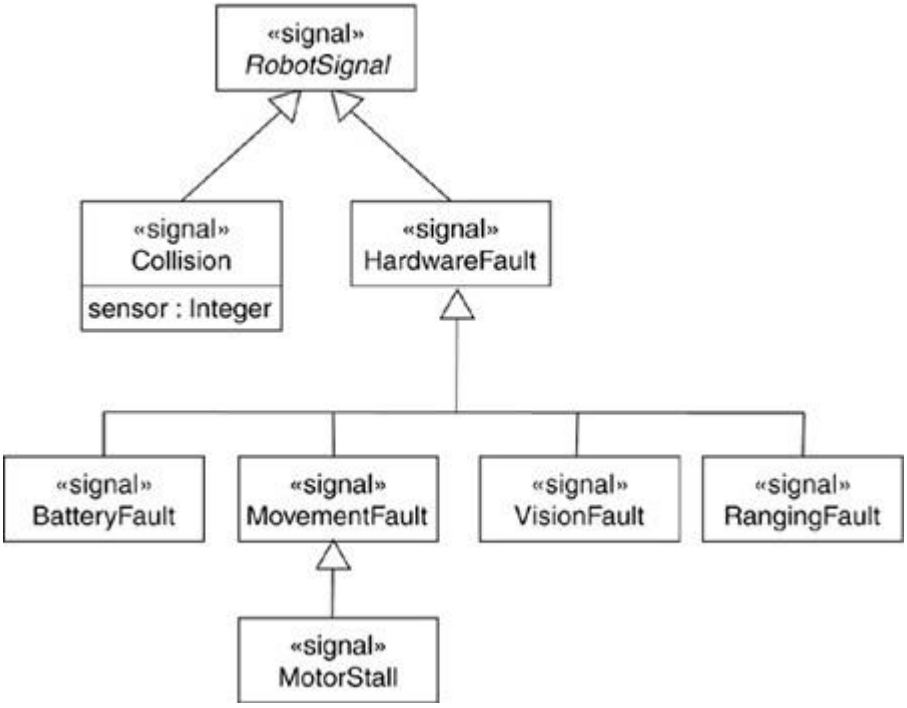
In most event-driven systems, signal events are hierarchical. For example, an autonomous robot might distinguish between external signals, such as a `Collision`, and internal ones, such as a `HardwareFault`. External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations. For example, `HardwareFault` signals might be further specialized as `BatteryFault` and `MovementFault`. Even these might be further specialized, such as `MotorStall`, a kind of `MovementFault`.

By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a `MotorStall`. As a leaf signal in this hierarchy, the transition can be triggered only by that signal, so it is not polymorphic. In contrast, suppose you modeled the state machine with a transition triggered by the receipt of a `HardwareFault`. In this case, the transition is polymorphic and can be triggered by a `HardwareFault` or any of its specializations, including `BatteryFault`, `MovementFault`, and `MotorStall`.

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 21-6 models a family of signals that may be handled by an autonomous robot. Note that the root signal (`RobotSignal`) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (`Collision` and `HardwareFault`), one of which (`HardwareFault`) is further specialized. Note that the `Collision` signal has one parameter.



igure 21-6. Modeling Families of Signals

Modeling Abnormal Occurrences

An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the abnormal occurrences that its operations can produce. If you are handed a class or an interface, the operations you can invoke will be clear, but the abnormal occurrences that each operation may raise will not be clear unless you model them explicitly.

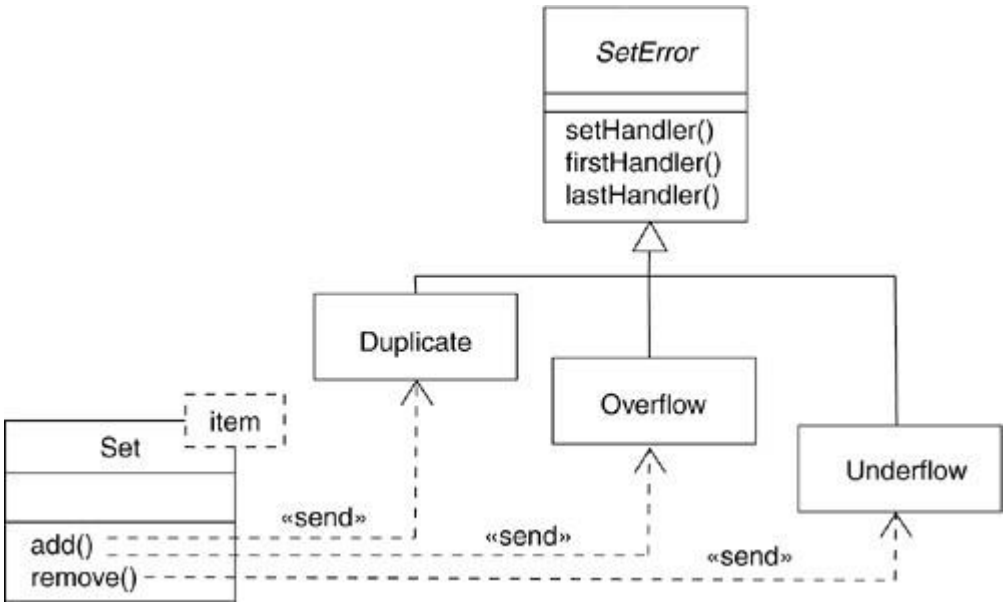
In the UML, abnormal occurrences are just additional kinds of events that can be modeled as signals. Error events may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model abnormal occurrences primarily to specify the kinds of abnormal occurrences that an object may produce.

To model abnormal occurrences

- For each class and interface, and for each operation of such elements, consider the normal things that happen. Then think of things that can go wrong and model them as signals among objects.
- Arrange the signals in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions as necessary.
- For each operation, specify the abnormal occurrence signals that it may raise. You can do so explicitly (by showing `send` dependencies from an operation to its signals) or you can use sequence diagrams illustrating various scenarios.

Figure 21-7 models a hierarchy of abnormal occurrences that may be produced by a standard library of container classes, such as the template class `Set`. This hierarchy is headed by the abstract signal `Error` and includes three specialized kinds of errors: `Duplicate`, `Overflow`, and `Underflow`. As shown, the `add` operation may produce `Duplicate` and `Overflow` signals, and the `remove` operation produces only the `Underflow` signal. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which signals each operation may send, you can create clients that use the `Set` class correctly.

Figure 21-7. Modeling Error Conditions



State Machines

In this chapter

- States, transitions, and activities
- Modeling the lifetime of an object
- Creating well-structured algorithms

Using an interaction, you can model the behavior of a society of objects that work together. Using a state machine, you can model the behavior of an individual object. 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.

You use state machines to model the dynamic aspects of a system. For the most part, this involves specifying the lifetime of the instances of a class, a use case, or an entire system. These instances may respond to such events as signals, operations, or the passing of time. When an event occurs, some effect will take place, depending on the current state of the object. An *effect* is the specification of a behavior execution within a state machine. Effects ultimately resolve into in the execution of actions that change the state of an object or return values. A *state* of an object is a period of time during which it satisfies some condition, performs some activity, or waits for some event.

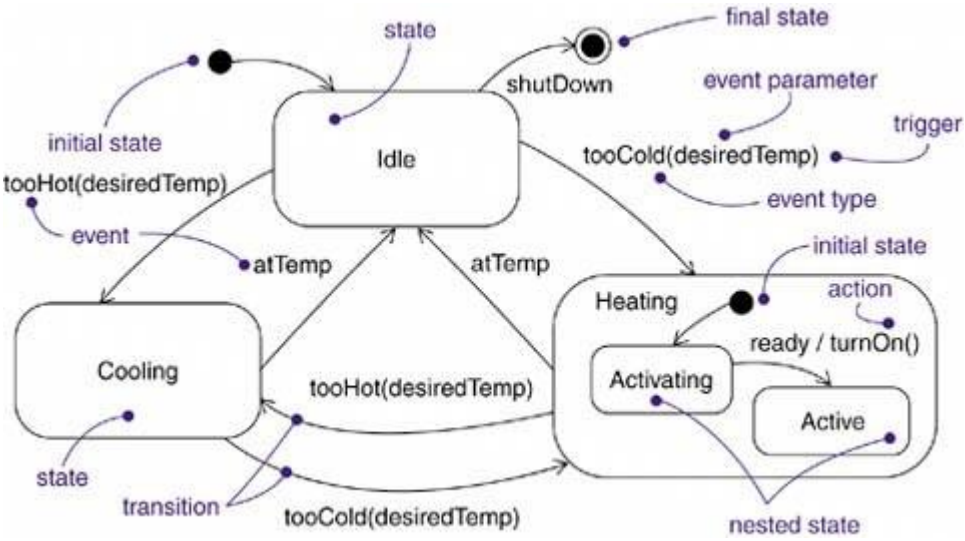
You can visualize the dynamics of execution in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams) or by emphasizing the potential states of the objects and the transitions among those states (using state diagrams).

Well-structured state machines are like well-structured algorithms: They are efficient, simple, adaptable, and understandable.
In the UML, you model the dynamic aspects of a system by using state machines. Whereas an interaction models a society of objects that work together to carry out some action, a state machine models the lifetime of a single object, whether it is an instance of a class, a use case, or even an entire system. In the life of an object, it may be exposed to a variety of events, such as a signal, the invocation of an operation, the creation or destruction of the object, the passing of time, or the change in some condition. In response to these events, the object performs some action, which is a computation, and then it changes its state to a new value. The behavior of such an object is therefore affected by the past, at least as the past is reflected in the current state. An object may receive an event, respond with an action, then change its state. An object may receive another event and its response may be different, depending on its current state in response to the previous event.

You use state machines to model the behavior of any modeling element, most commonly a class, a use case, or an entire system. State machines may be visualized using state diagrams. You can focus on the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

The UML provides a graphical representation of states, transitions, events, and effects, as Figure 22-1 shows. This notation permits you to visualize the behavior of an object in a way that lets you emphasize the important elements in the life of that object.

Figure 22-1. State Machines



Terms and Concepts

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. 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 events. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event 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 nonatomic execution within a state machine. An *action* is an executable computation that results in a change in state of the model or the return of a value. Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line or path from the original state to the new state.

Context

Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist. In between, an object may act on other objects (by sending them messages) as well as be acted on (by being the target of a message). In many cases, these messages will be simple, synchronous operation calls. For example, an instance of the class `Customer` might invoke the operation `getAccountBalance` on an instance of the class `BankAccount`. Objects such as these don't need a state machine to specify their behavior because their current behavior does not depend on their past.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous messages communicated between instances. For example, a cellular phone must respond to random phone calls (from other phones), keypad events (from the customer initiating a phone call), and to events from the network (when the phone moves from one call to another). Similarly, you'll encounter objects whose current behavior depends on their past behavior. For example, the behavior of an air-to-air missile guidance system will depend on its current state, such as `NotFlying` (it's not a good idea to launch a missile while it's attached to an aircraft that's still sitting on the ground) or `Searching` (you shouldn't arm the missile until you have a good idea what it's going to hit).

The behavior of an object that must respond to asynchronous messages or whose current behavior

depends on its past is best specified by using a state machine. This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no transition for that signal in its current state and does not defer the signal in that state will simply ignore that signal. In other words, the absence of a transition for a signal is not an error; it means that the signal is not of interest at that point. You'll also use state machines 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` (waiting for a command to start heating the house), `Activating` (its gas is on, but it's waiting to come up to temperature), `Active` (its gas and blower are both on), and `ShuttingDown` (its gas is off but its blower is on, flushing residual heat from the system).

When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of `Heater` might be `Idle` or perhaps `ShuttingDown`.

A state has several parts:

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit effects	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving nonorthogonal (sequentially active) or orthogonal (concurrently active) substates
5. 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

As Figure 22-2 shows, you represent a state as a rectangle with rounded corners.

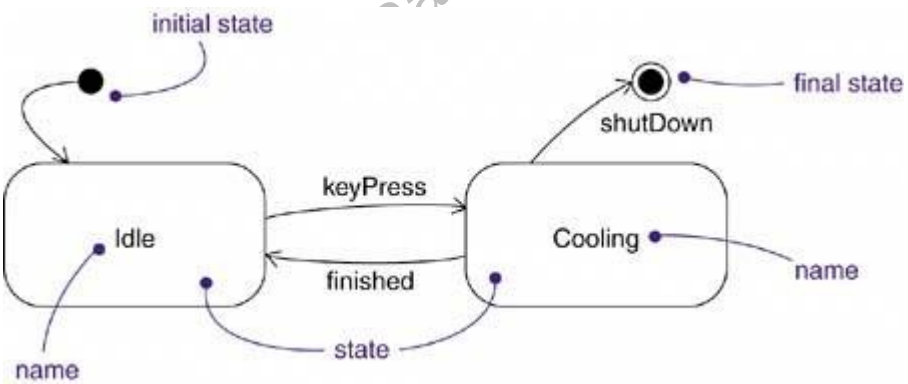


Figure 22-2. States

Initial and Final States

As the figure shows, there are two special states that may be defined for an object's state machine. First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle. Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle (a bull's eye).

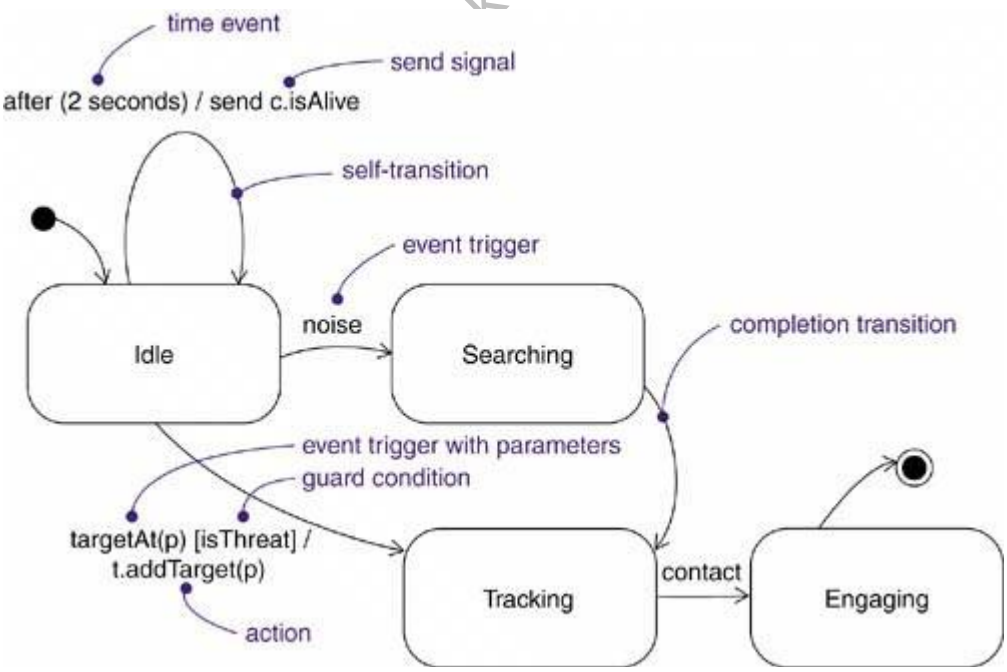
A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state. For example, a `Heater` might transition from the `Idle` to the `Activating` state when an event such as `tooCold` (with the parameter `desiredTemp`) occurs.

A transition has five parts.

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied
2. Event trigger	The event whose recognition by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
3. Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates true, the transition is eligible to fire; if the expression evaluates false, the transition does not fire, and if there is no other transition that could be triggered by that same event, the event is lost
4. Effect	An executable behavior, such as an action, that may act on the object that owns the state machine and indirectly on other objects that are visible to the object
5. Target state	The state that is active after the completion of the transition

As Figure 22-3 shows, a transition is rendered as a solid directed line from the source to the target state. A self-transition is a transition whose source and target states are the same.

Figure 22-3. Transitions



An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. As shown in the previous figure, events may include signals, calls, the passing of time, or a change in state. A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.
It is also possible to have a completion transition, represented by a transition with no event trigger. A completion transition is triggered implicitly when its source state has completed its behavior, if any.

Guard Condition

As the previous figure shows, 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. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

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. Within the Boolean expression, you can include conditions about the state of an object (for example, the expression `aHeater in Idle`, which evaluates true if the `Heater` object is currently in the `Idle` state). If the condition is not true when it is tested, the event does not occur later when the condition becomes true. Use a change event to model that kind of behavior.

Effect

An effect is a behavior that is executed when a transition fires. Effects may include inline computation, operation calls (to the object that owns the state machine as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object. To indicate sending a signal you can prefix the signal name with the keyword `send` as a visual cue.

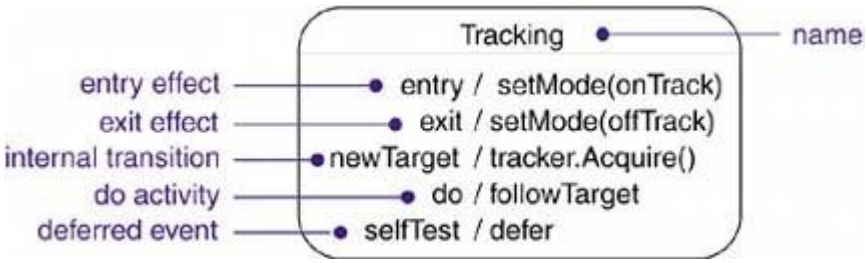
Transitions only occur when the state machine is quiescent, that is, when it is not executing an effect from a previous transition. The execution of the effect of a transition and any associated entry and exit effects run to completion before any additional events are allowed to cause additional transitions. This is in contrast to a do-activity (described later in this chapter), which may be interrupted by events.

Advanced States and Transitions

You can model a wide variety of behavior using only the basic features of states and transitions in the UML. Using these features, you'll end up with flat state machines, which means that your behavioral models will consist of nothing more than arcs (transitions) and vertices (states).

UML state machines have a number of features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit effects, internal transitions, do-activities, and deferred events. These features are shown as text strings within a text compartment of the state symbol, as shown in Figure 22-4.

Figure 22-4. Advanced States and Transitions



Entry and Exit Effects

In a number of modeling situations, you'll want to perform some setup action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to

perform some cleanup action no matter which transition led you away. For example, in a missile guidance system, you might want to explicitly announce the system is `onTrack` whenever it's in the `tracking` state, and `offTrack` whenever it's out of the state. Using flat state machines, you can achieve this effect by putting those actions on every entering and exiting transition, as appropriate. However, that's somewhat error prone; you have to remember to add these actions every time you add a new transition. Furthermore, modifying this action means that you have to touch every neighboring transition.

As Figure 22-4 shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry effect (marked by the keyword `entry`) and an exit effect (marked by the keyword `exit`), each with its appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Entry and exit effects may not have arguments or guard conditions, but the entry effect at the top level of a state machine for a class may have parameters for the arguments that the machine receives when the object is created.

Internal Transitions

Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called internal transitions, and they are subtly different from self-transitions. In a self-transition, such as you see in Figure 21-3, an event triggers the transition, you leave the state, an action (if any) is performed, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition executes the state's exit action, then it executes the action of the self-transition, and finally, it executes the state's entry action.

However, suppose you want to handle the event but don't want to execute the state's entry and exit actions. The UML provides a shorthand for this idiom using an internal transition. An *internal transition* is a transition that responds to an event by performing an effect but does not change state. In Figure 21-4, the event `newTarget` labels an internal transition; if this event occurs while the object is in the `tracking` state, action `tracker.acquire` is executed but the state remains the same, and no entry or exit actions are executed. You indicate an internal transition by including a transition string (including an event name, optional guard condition, and effect) inside the symbol for a state instead of on a transition arrow. Note that the keywords `entry`, `exit`, and `do` are reserved words that may not be used as event names. Whenever you are in the state and an event labeling an internal transition occurs, the corresponding effect is performed without leaving and then reentering the state. Therefore, the event is handled without invoking the state's exit and then entry actions.

Do-Activities

When an object is in a state, it generally sits idle, waiting for an event to occur. Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event. For example, if an object is in the `tracking` state, it might `followTarget` as long as it is in that state. As Figure 21-4 shows, in the UML you use the special `do` transition to specify the work that's to be done inside a state after the entry action is dispatched. You can also specify a behavior, such as a sequence of actionsfor example, do / op1(a); op2(b); op3(c). If the occurrence of an event causes a transition that forces an exit from the state, any ongoing do-activity of the state is immediately terminated.

Deferred Events

Consider a state such as `tracking`. As illustrated in Figure 21-3, suppose there's only one transition leading out of this state, triggered by the event `contact`. While in the state `TRacking`, any events other than `contact` and other than those handled by its substates will be lost. That means that the event may occur, but it will be ignored and no action will result because of the presence of that event.

In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out. However, in some modeling situations, you'll want to accept some events but postpone a response to them until later. For example, while in the `TRacking` state, you may want to postpone a response to signals such as `selfTest`, perhaps sent by some maintenance agent in the system.

In the UML, you can specify this behavior by listing deferred events. A deferred event is an event whose processing in the state is postponed until another state becomes active; if the event is not

deferred in that state, the event is handled and may trigger transitions as if it had just occurred. If the state machine passes through a sequence of states in which the event is deferred, it is preserved until a state is finally encountered in which the event is not deferred. Other nondeferred events may occur during the interval. As you can see in Figure 21-4, you can specify a deferred event by listing the event with the special action `defer`. In this example, `selfTest` events may happen while in the `TRacking` state, but they are held until the object is in the `Engaging` state, at which time it appears as if they just occurred.

Submachines

A state machine may be referenced within another state machine. Such a referenced state machine is called a *submachine*. They are useful in building large state models in a structured manner. See the *UML Reference Manual* for details.

Substates

These advanced features of states and transitions solve a number of common state machine modeling problems. However, there's one more feature of the UML's state machinessubstatesthat does even more to help you simplify the modeling of complex behaviors. A substate is a state that's nested inside another one. For example, a `Heater` might be in the `Heating` state, but also while in the `Heating` state, there might be a nested state called `Activating`. In this case, it's proper to say that the object is both `Heating` and `Activating`.

A simple state is a state that has no substructure. A state that has substatesthat is, nested statesis called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (nonorthogonal) substates. In the UML, you render a composite state just as you do a simple state, but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level.

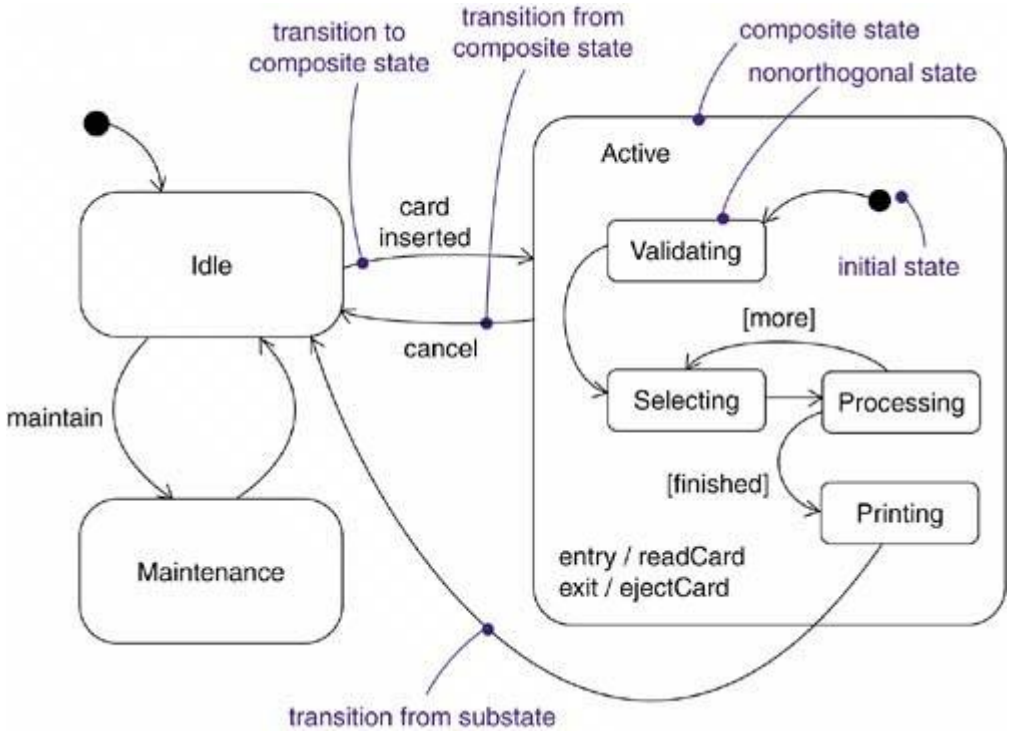
Nonorthogonal Substates

Consider the problem of modeling the behavior of an ATM. This system might be in one of three basic states: `Idle` (waiting for customer interaction), `Active` (handling a customer's transaction), and `Maintenance` (perhaps having its cash store replenished). While `Active`, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the `Idle` state. You might represent these stages of behavior as the states `Validating`, `Selecting`, `Processing`, and `Printing`. It would even be desirable to let the customer select and process multiple transactions after `Validating` the account and before `Printing` a final receipt.

The problem here is that, at any stage in this behavior, the customer might decide to cancel the transaction, returning the ATM to its `Idle` state. Using flat state machines, you can achieve that effect, but it's quite messy. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the `Active` sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many such interrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using nested substates, there's a simpler way to model this problem, as Figure 22-5 shows. Here, the `Active` state has a substructure, containing the substates `Validating`, `Selecting`, `Processing`, and `Printing`. The state of the ATM changes from `Idle` to `Active` when the customer enters a credit card in the machine. On entering the `Active` state, the entry action `readCard` is performed. Starting with the initial state of the substructure, control passes to the `Validating` state, then to the `Selecting` state, and then to the `Processing` state. After `Processing`, control may return to `Selecting` (if the customer has selected another transaction) or it may move on to `Printing`. After `Printing`, there's a completion transition back to the `Idle` state. Notice that the `Active` state has an exit action, which ejects the customer's credit card.

Figure 22-5. Sequential Substates



Notice also the transition from the `Active` state to the `Idle` state, triggered by the event `cancel`. In any substate of `Active`, the customer might cancel the transaction, and that returns the ATM to the `Idle` state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the `Active` state, no matter what caused a transition out of that state). Without substates, you'd need a transition triggered by `cancel` on every substructure state.

Substates such as `Validating` and `Processing` are called nonorthogonal, or disjoint, substates. Given a set of nonorthogonal substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates (or the final state) at a time. Therefore, nonorthogonal substates partition the state space of the composite state into disjoint states.

From a source outside an enclosing composite state, a transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after performing its entry action, if any. If its target is the nested state, control passes to the nested state after performing the entry action (if any) of the composite state and then the entry action (if any) of the substate.

A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state (and its exit action, if any, is executed), then it leaves the composite state (and its exit action, if any, is executed). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine. The completion transition of a composite state is taken when control reaches the final substate within the composite state.

History States

A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

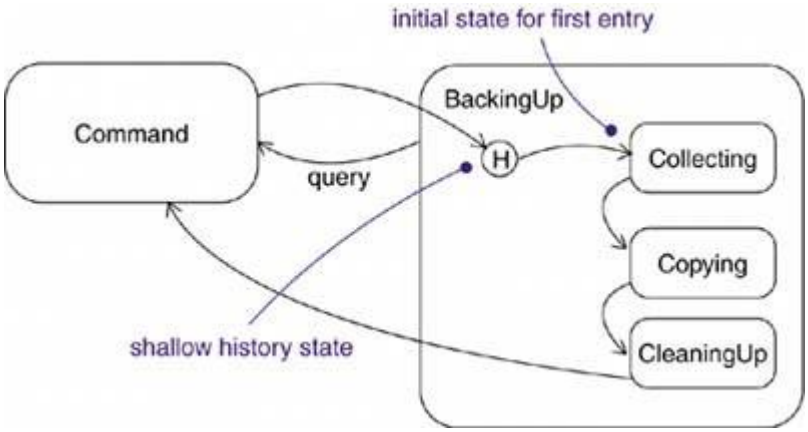
Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at its initial state (unless, of course, the transition targets a substate directly). However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. For example, in modeling the behavior of an agent that does an unattended backup of computers across a network, you'd like it to remember where it was in the process if it ever gets interrupted by, for example, a query from the operator.

Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd

need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. In this way, leaving the composite state would cause the last substate to be remembered; entering the composite state would transition to the proper substate. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.

In the UML, a simpler way to model this idiom is by using history states. A history state allows a composite state that contains nonorthogonal substates to remember the last substate that was active in it prior to the transition from the composite state. As Figure 22-6 shows, you represent a shallow history state as a small circle containing the symbol `H`.

Figure 22-6. History State



If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such as `Collecting`. The target of this transition specifies the initial state of the nested state machine the first time it is entered. Continuing, suppose that while in the `BackingUp` state and the `Copying` state, the `query` event is posted. Control leaves `Copying` and `BackingUp` (dispatching their exit actions as necessary) and returns to the `Command` state. When the action of `Command` completes, the completion transition returns to the history state of the composite state `BackingUp`. This time, because there is a history to the nested state machine, control passes back to the `Copying` statethus bypassing the `Collecting` statebecause `Copying` was the last substate active prior to the transition from the state `BackingUp`.
In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

Orthogonal Substates

Nonorthogonal substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify orthogonal regions. These regions let you specify two or more state machines that execute in parallel in the context of the enclosing object.

For example, Figure 22-7 shows an expansion of the `Maintenance` state from Figure 21-5. `Maintenance` is decomposed into two orthogonal regions, `Testing` and `Commanding`, shown by nesting them in the `Maintenance` state but separating them from one another with a dashed line. Each of these orthogonal regions is further decomposed into substates. When control passes from the `Idle` to the `Maintenance` state, control then forks to two concurrent flowsthe enclosing object will be in both the `Testing` region and the `Commanding` region. Furthermore, while in the `Commanding` region, the enclosing object will be in the `Waiting` or the `Command`state.

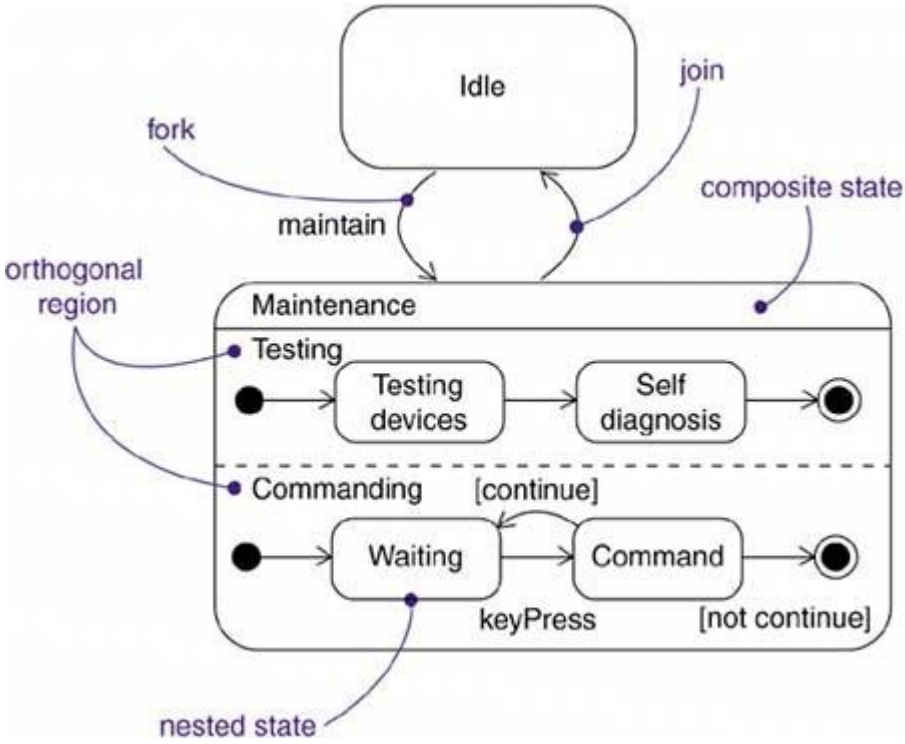


Figure 22-7. Concurrent Substates

Execution of these two orthogonal regions continues in parallel. Eventually, each nested state machine reaches its final state. If one orthogonal region reaches its final state before the other, control in that region waits at its final state. When both nested state machines reach their final states, control from the two orthogonal regions joins back into one flow.

Whenever there's a transition to a composite state decomposed into orthogonal regions, control forks into as many concurrent flows as there are orthogonal regions. Similarly, whenever there's a transition from a composite substate decomposed into orthogonal regions, control joins back into one flow. This holds true in all cases. If all orthogonal regions reach their final states, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Fork and Join

Usually, entry to a composite state with orthogonal regions goes to the initial state of each orthogonal region. It is also possible to transition from an external state directly to one or more orthogonal states. This is called a fork, because control passes from a single state to several orthogonal states. It is shown as a heavy black line with one incoming arrow and several outgoing arrows, each to one of the orthogonal states. There must be at most one target state in each orthogonal region. If one or more orthogonal regions have no target states, then the initial state of those regions is implicitly chosen. A transition to a single orthogonal state within a composite state is also an implicit fork; the initial states of all the other orthogonal regions are implicitly part of the fork.

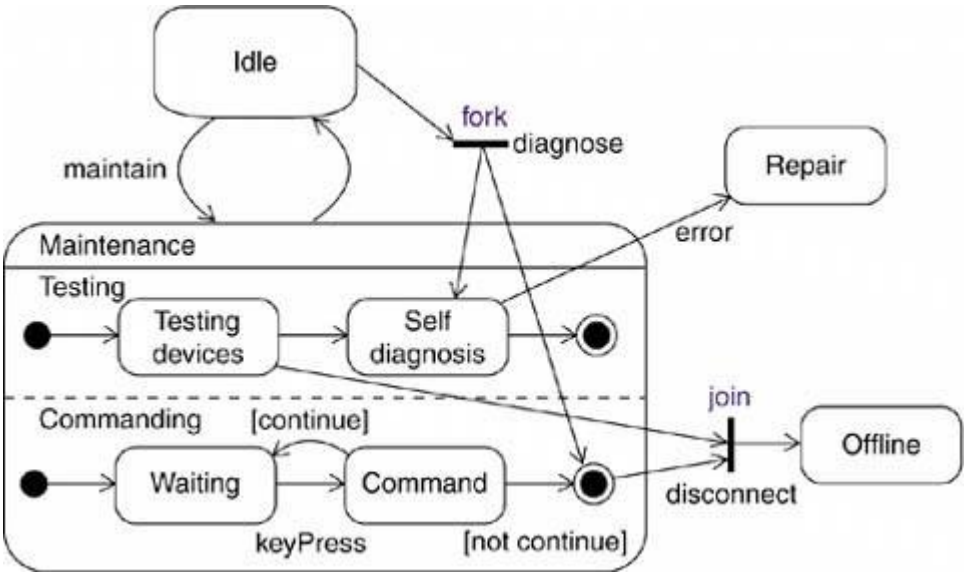
Similarly, a transition from any state within a composite state with orthogonal regions forces an exit from all the orthogonal regions. Such a transition often represents an error condition that forces termination of parallel computations.

A join is a transition with two or more incoming arrows and one outgoing arrow. Each incoming arrow must come from a state in a different orthogonal region of the same composite state. The join may have a trigger event. The join transition is effective only if all of the source states are active; the status of other orthogonal regions in the composite state is irrelevant. If the event occurs, control leaves all of the orthogonal regions in the composite state, not just the ones with arrows from them.

Figure 22-8 shows a variation on the previous example with explicit fork and join transitions. The transition `maintain` to the composite state `Maintenance` is still an implicit fork into the default initial states of the orthogonal regions. In this example, however, there is also an explicit fork from `Idle` into the two nested states `Self diagnose` and the final state of the `Commanding` region. (A final state is a real state and can be the target of a transition.) If an error event occurs while the `Self diagnose` state is active, the implicit join transition to `Repair` fires; both the `Self diagnose` state and whatever state is active in the `Commanding` region are exited. There is also an explicit join transition to the `Offline`

state. This transition fires only if the `disconnect` event occurs while the `Testing devices` state and the final state of the `Commanding` region are both active; if both states are not active, the event has no effect.

Figure 22-8. Fork and join transitions



Active Objects

Another way to model concurrency is by using active objects. Thus, rather than partitioning one object's state machine into two (or more) concurrent regions, you could define two active objects, each of which is responsible for the behavior of one of the concurrent regions. If the behavior of one of these concurrent flows is affected by the state of the other, you'll want to model this using orthogonal regions. If the behavior of one of these concurrent flows is affected by messages sent to and from the other, you'll want to model this using active objects. If there's little or no communication between the concurrent flows, then the approach you choose is a matter of taste, although most of the time, using active objects makes your design decisions more obvious.

Common Modeling Techniques

Modeling the Lifetime of an Object

The most common purpose for which you'll use state machines is to model the lifetime of an object, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state machine models the behavior of a single object over its lifetime, such as you'll find with user interfaces, controllers, and devices.

When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior. Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction.

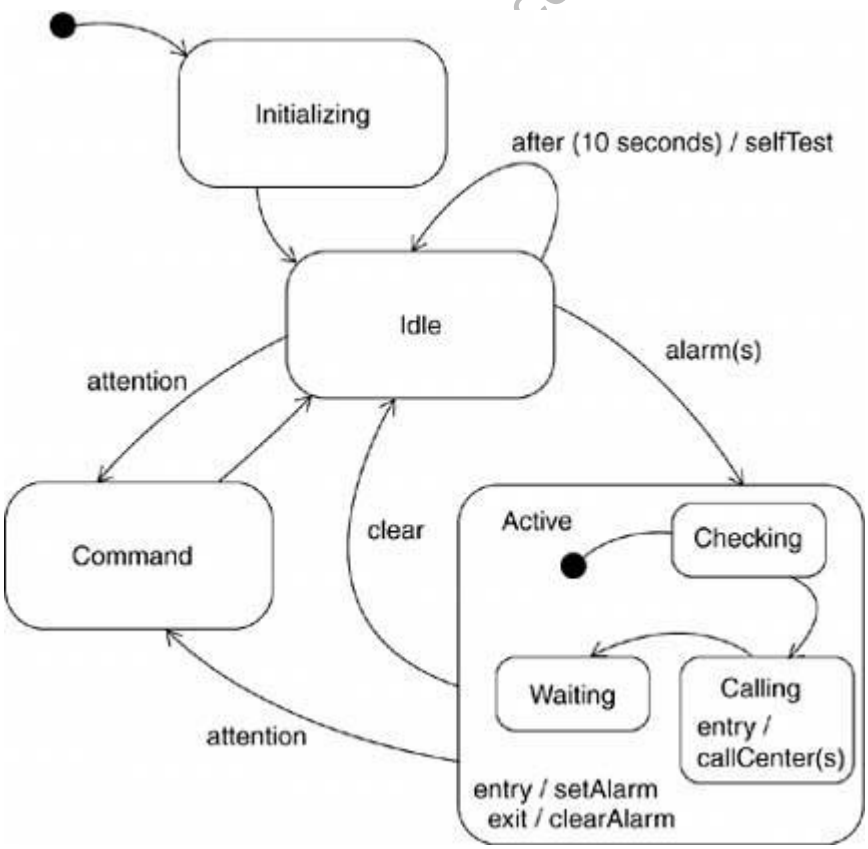
To model the lifetime of an object,

- 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, find 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 22-9 shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house.

Figure 22-9. Modeling the Lifetime of An Object



In the lifetime of this controller class, there are four main states: `Initializing` (the controller is starting up), `Idle` (the controller is ready and waiting for alarms or commands from the user), `Command`

(the controller is processing commands from the user), and `Active` (the controller is processing an alarm condition). When the controller object is first created, it moves first to the `Initializing` state and then unconditionally to the `Idle` state. The details of these two states are not shown, other than the self-transition with the time event in the `Idle` state. This kind of time event is commonly found in embedded systems, which often have a heartbeat timer that causes a periodic check of the system's health.

Control passes from the `Idle` state to the `Active` state on receipt of an `alarm` event (which includes the parameter `s`, identifying the sensor that was tripped). On entering the `Active` state, `setAlarm` is performed as the entry action, and control then passes first to the `Checking` state (validating the alarm), then to the `Calling` state (calling the alarm company to register the alarm), and finally to the `Waiting` state. The `Active` and `Waiting` states are exited only upon `clear`ing the alarm or by the user signaling the controller for `attention`, presumably to issue a command.

Notice that there is no final state. That, too, is common in embedded systems, which are intended to run indefinitely.

Processes and Threads

In this chapter

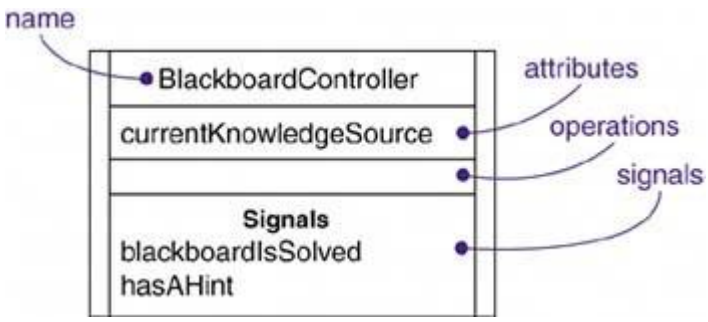
- Active objects, processes, and threads
- Modeling multiple flows of control
- Modeling interprocess communication
- Building thread-safe abstractions

Not only is the real world a harsh and unforgiving place, but it is a very busy place as well. Events happen and things take place all at the same time. Therefore, when you model a system of the real world, you must take into account its process view, which encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.

In the UML, you model each independent flow of control as an active object that represents a process or thread that can initiate control activity. 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.

Building abstractions so that they work safely in the presence of multiple flows of control is hard. In particular, you have to consider approaches to communication and synchronization that are more complex than for sequential systems. You also have to be careful to neither over-engineer your process view (too many concurrent flows and your system ends up thrashing) nor under-engineer it (insufficient concurrency does not optimize the system's throughput.

Figure 23-1. Active Class



Terms and Concepts

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. 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. Graphically, an active class is rendered as a rectangle with double lines for left and right sides. Processes and threads are rendered as stereotyped active classes (and also appear as sequences in interaction diagrams).

Flow of Control

In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another. Even if there are concurrent things happening among the actors outside the system, a sequential program will process only one event at a time, queuing or discarding any concurrent external events.

This is why it's called a flow of control. If you trace the execution of a sequential program, you'll see the locus of execution flow from one statement to another, in sequential order. You might see actions that branch, loop, and jump about, and if there is any recursion or iteration, you see the flow circle back on itself. Nonetheless, in a sequential system, there would be a single flow of execution.

In a concurrent system, there is more than one flow of controlthat is, more than one thing can take place at a time. In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread. If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

In the UML, you use an active class to represent a process or thread that is the root of an independent flow of control and that is concurrent with all peer flows of control.

Classes and Events

Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow. In contrast to active classes, plain classes are implicitly called passive because they cannot independently initiate control activity.

You use active classes to model common families of processes or threads. In technical terms, this means that an active objectan instance of an active classreifies (is a manifestation of) a process or thread. By modeling concurrent systems with active objects, you give a name to each independent flow of control. 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.

Active classes share the same properties as all other classes. Active classes may have instances. Active classes may have attributes and operations. Active classes may participate in dependency, generalization, and association (including aggregation) relationships. Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints. Active classes may be the realization of interfaces. Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines. Active classes may participate in collaborations.

In your diagrams, active objects may appear wherever passive objects appear. You can model the collaboration of active and passive objects by using interaction diagrams (including sequence and collaboration diagrams). An active object may appear as the target of an event in a state machine. Speaking of state machines, both passive and active objects may send and receive signal events and call events.

Communication

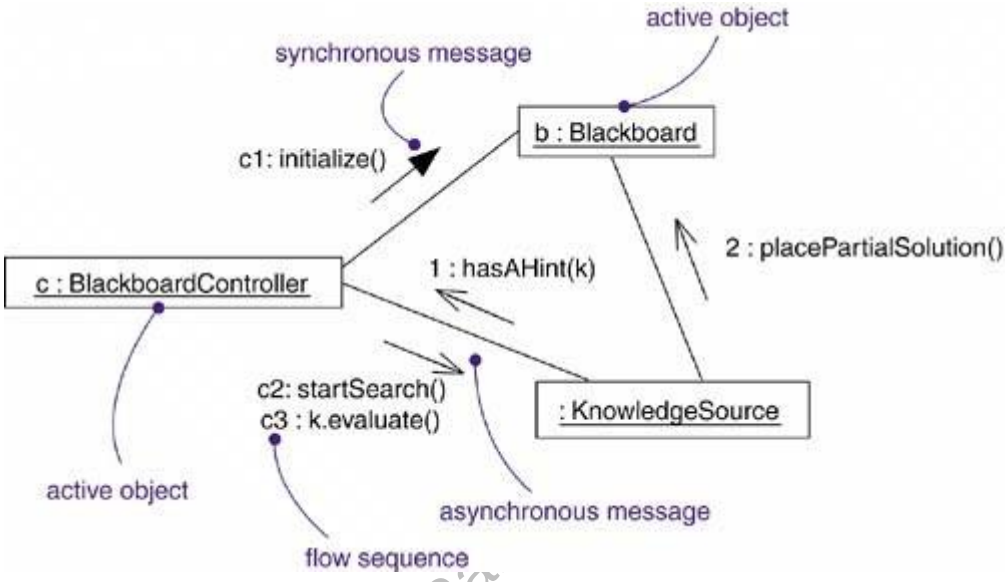
When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider.

First, a message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.

Second, a message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication. First, one active object might synchronously call an operation of another. That kind of communication has rendezvous semantics, which means that the caller calls the operation; the caller waits for the receiver to accept the call; the operation is invoked; a method is chosen for execution based on the operation and the class of the receiver object; the method is executed; a return object (if any) is passed back to the caller; and then the two objects continue on their independent paths. For the duration of the call, the two flows of control are in lock step. Second, one active object might asynchronously send a signal or call an operation of another object. That kind of communication has mailbox semantics, which means that the caller sends the signal or calls the operation and then continues on its independent way. In the meantime, the receiver accepts the signal or call whenever it is ready (with intervening events or calls queued) and continues on its way after it is done. This is called a mailbox because the two objects are not synchronized; rather, one object drops off a message for the other.

In the UML, you render a synchronous message with a solid (filled) arrowhead and an asynchronous message as a stick arrowhead, as in Figure 23-2.

Figure 23-2. Communication



Third, a message may be passed from an active object to a passive object. A potential conflict arises if more than one active object at a time passes its flow of control through one passive object. It is an actual conflict if more than one object writes or reads and writes the same attributes. In that situation, you have to model the synchronization of these two flows very carefully, as discussed in the next section.

Fourth, a message may be passed from a passive object to an active one. At first glance, this may seem illegal, but if you remember that every flow of control is rooted in some active object, you'll understand that a passive object passing a message to an active object has the same semantics as an active object passing a message to an active object.

Synchronization

Visualize for a moment the multiple flows of control that weave through a concurrent system. When a flow passes through an operation, we say that at a given moment, the locus of control is in the operation. If that operation is defined for some class, we can also say that at a given moment, the locus of control is in a specific instance of that class. You can have multiple flows of control in one operation (and therefore in one object), and you can have different flows of control in different operations (but still result in multiple flows of control in the one object).

The problem arises when more than one flow of control is in one object at the same time. If you are not careful, more than one flow might modify the same attribute, corrupting the state of the object or losing information. This is the classical problem of mutual exclusion. A failure to deal with it properly yields all sorts of race conditions and interference that cause concurrent systems to fail in mysterious and unrepeatable ways.

The key to this problem is serialization of access to the critical object. There are three approaches,

each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. 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.
2. Guarded — The semantics and integrity of the object are 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 execute on the object, reducing this to sequential semantics. There is a danger of deadlock if care is not taken.
3. Concurrent — The semantics and integrity of the object are guaranteed in the presence of multiple flows of control because multiple flows of control access disjoint sets of data or only read data. This situation can be arranged by careful design rules.

Some programming languages support these constructs directly. Java, for example, has the `synchronized` property, which is equivalent to the UML's `concurrent` property. In every language that supports concurrency, you can build support for all these properties by constructing them out of semaphores.

As Figure 23-3 shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation. Note that concurrency must be asserted separately for each operation and for the entire object. Asserting concurrency for an operation means that multiple invocations of that operation can execute concurrently without danger. Asserting concurrency for an object means that invocations of different operations can execute concurrently without danger; this is a more stringent condition.

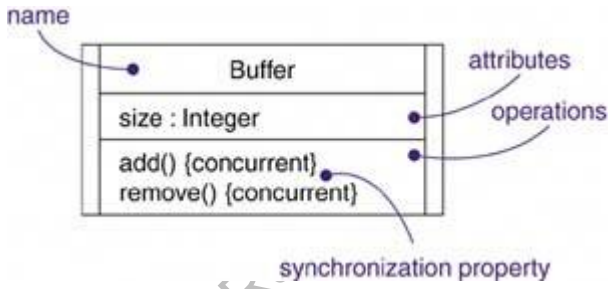


Figure 23-3. Synchronization

Common Modeling Techniques

Modeling Multiple Flows of Control

Building a system that encompasses multiple flows of control is hard. Not only do you have to decide how best to divide work across concurrent active objects, but once you've done that, you also have to devise the right mechanisms for communication and synchronization among your system's active and passive objects to ensure that they behave properly in the presence of these multiple flows. For that reason, it helps to visualize the way these flows interact with one another. You can do that in the UML by applying class diagrams (to capture their static semantics) and interaction diagrams (to capture their dynamic semantics) containing active classes and objects.
To model multiple flows of control,

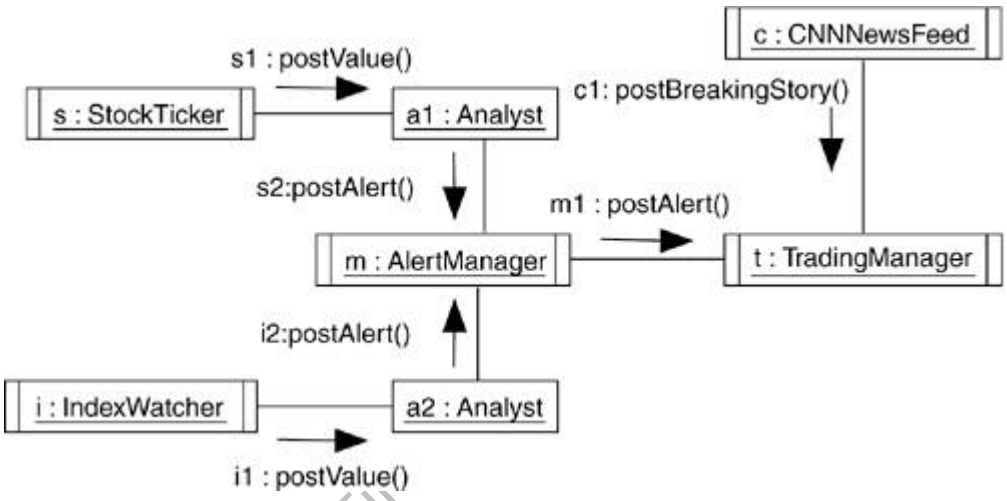
- Identify the opportunities for concurrent execution and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over-engineer your system by introducing unnecessary concurrency.
- Consider a balanced distribution of responsibilities among 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.

For example, Figure 23-4 shows part of the process view of a trading system. You'll find three objects that push information into the system concurrently: a `StockTicker`, an `IndexWatcher`, and a `CNNNewsFeed` (named `s`, `i`, and `c`, respectively). Two of these objects (`s` and `i`) communicate with their own `Analyst` instances (`a1` and `a2`). At least as far as this model goes, the `Analyst` can be designed under the simplifying assumption that only one flow of control will be active in its instances at a time. Both `Analyst` instances, however, communicate simultaneously with an `AlertManager` (named `m`). Therefore, `m` must be designed to preserve its semantics in the presence of multiple flows. Both `m` and `c` communicate simultaneously with `t`, a `tradingManager`. Each flow is given a sequence number that is distinguished by the flow of control that owns it.

Figure 23-4. Modeling Flows of Control



In diagrams such as this, it's also common to attach corresponding state machines, with orthogonal states showing the detailed behavior of each active object.

Modeling Interprocess Communication

As part of incorporating multiple flows of control in your system, you also have to consider the mechanisms by which objects that live in separate flows communicate with one another. Across threads (which live in the same address space), objects may communicate via signals or call events, the latter of which may exhibit either asynchronous or synchronous semantics. Across processes

(which live in separate address spaces), you usually have to use different mechanisms.

The problem of interprocess communication is compounded by the fact that, in distributed systems, processes may live on separate nodes. Classically, there are two approaches to interprocess communication: message passing and remote procedure calls. In the UML, you still model these as asynchronous or synchronous events, respectively. But because these are no longer simple in-process calls, you need to adorn your designs with further information.

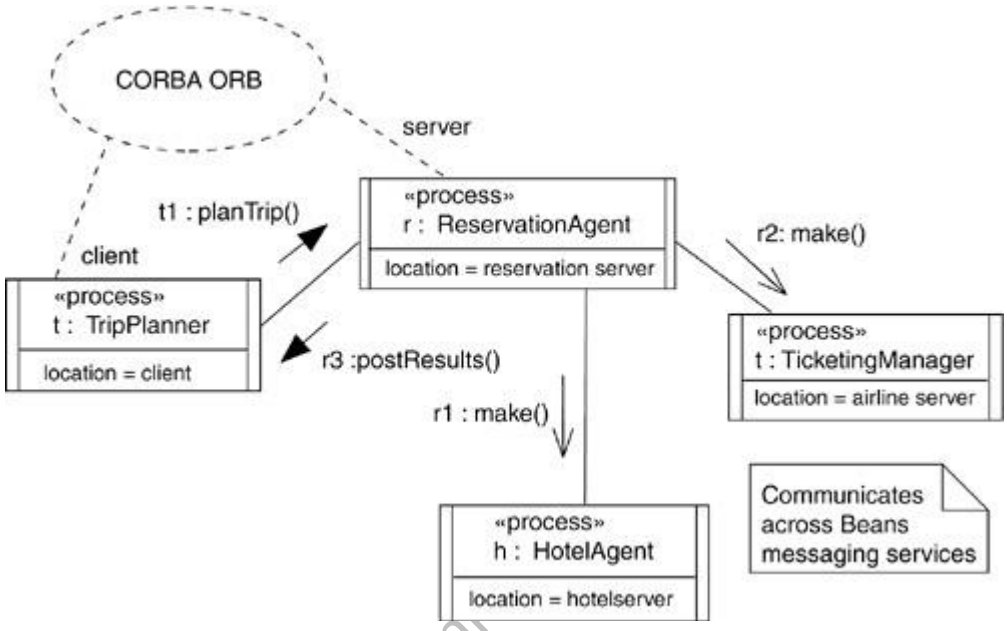
To model interprocess communication,

- Model the multiple flows of control.

- 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 23-5 shows a distributed reservation system with processes spread across four nodes. Each object is marked using the `process` stereotype. Each object is marked with a `location` attribute, specifying its physical location. Communication among the `ReservationAgent`, `TicketingManager`, and `HotelAgent` is asynchronous. Communication is described in a note as building on a Java Beans messaging service. Communication between the `tripPlanner` and the `ReservationSystem` is synchronous. The semantics of their interaction is found in the collaboration named `CORBA ORB`. The `TRipPlanner` acts as a `client`, and the `ReservationAgent` acts as a `server`. By zooming into the collaboration, you'll find the details of how this server and client collaborate.

Figure 23-5. Modeling Interprocess Communication



Time and Space

- Time, duration, and location
- Modeling timing constraints
- Modeling the distribution of objects
- Modeling objects that migrate
- Dealing with real time and distributed systems

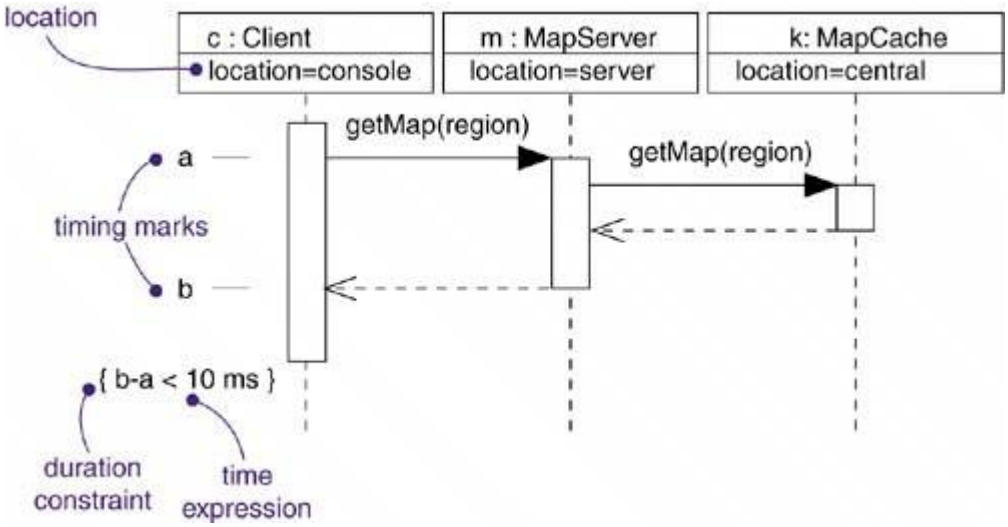
The real world is a harsh and unforgiving place. Events may happen at unpredictable times yet demand specific responses at specific times. A system's resources may have to be distributed around the worldsome of those resources might even move aboutraising issues of latency, synchronization, security, and quality ofservice.

Modeling time and space is an essential element of any real time and/or distributed system. You use a number of the UML's features, including timing marks, time expressions, constraints, and tagged values, to visualize, specify, construct, and document these systems.

Dealing with real time and distributed systems is hard. Good models reveal the properties of a system's time and space characteristics.

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, as Figure 24-1 shows.

Figure 24-1. Timing Constraints and Location



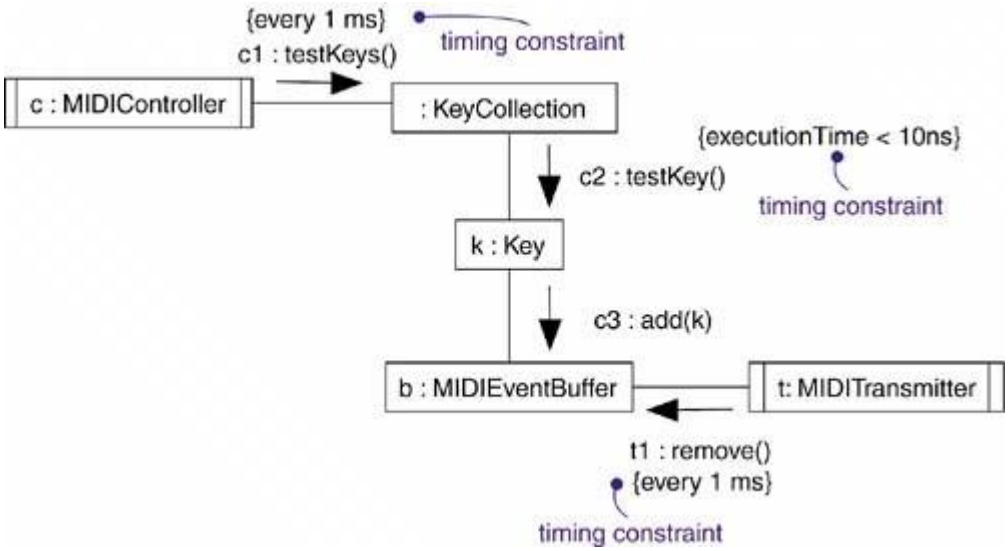
Terms and Concepts

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, by their very name, 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, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used in time expressions. Messages in an interaction are usually not given names. They are mainly rendered with the name of an event, such as a signal or a call. However, you can give them names to write a time expression because the same event may trigger different messages. If the designated message is ambiguous, use the explicit name of the message in an expression to designate the message you want to mention in a time expression. Given a message name, you can refer to any of three functions of that messagethat is, `sendTime`, `receiveTime`, and `transmissionTime`. (These are our suggested functions, not official UML functions. A real-time system might have even more functions.) For synchronous calls, you can also reference the round-trip message time with `executionTime` (again our suggestion). You can then use these functions to specify arbitrarily complex time expressions, perhaps even using weights or offsets that are either constants or variables (as long as those variables can be bound at execution time). Finally, as shown in Figure 24-2, you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure 24-2. Time



Location

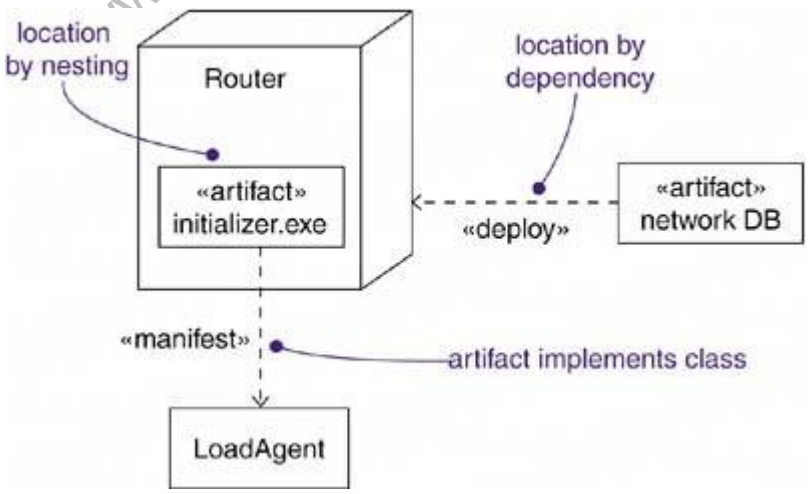
Distributed systems, by their nature, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

Components are discussed in Chapter 15; nodes are discussed in Chapter 27; deployment diagrams are discussed in Chapter 31; the class/object dichotomy is discussed in Chapters 2 and 13; classes are discussed in Chapters 4 and 9.

In the UML, you model the deployment view of a system by using deployment diagrams that represent the topology of the processors and devices on which your system executes. Artifacts such as executables, libraries, and tables reside on these nodes. Each instance of a node will own instances of certain artifacts, and each instance of an artifact will be owned by exactly one instance of a node (although instances of the same kind of artifact may be spread across different nodes).

Components and classes may be manifested as artifacts. For example, as Figure 24-3 shows, class `LoadAgent` is manifested by the artifact `initializer.exe` that lives on the node of type `Router`.

Figure 24-3. Location



As the figure illustrates, you can model the location of an artifact in two ways in the UML. First, as shown for the `Router`, you can physically nest the element (textually or graphically) in a extra compartment in its enclosing node. Second, you can use a dependency with the keyword `«deploy»` from the artifact to the node that contains it.

Common Modeling Techniques

Modeling Timing Constraints

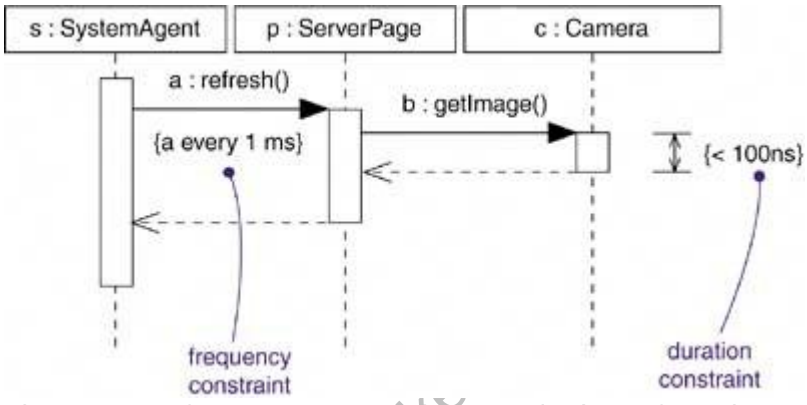
Modeling the absolute time of an event and modeling the relative time between events are the primary time-critical properties of real time systems for which you'll use timing constraints.

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.

For example, as shown in Figure 24-4, the left-most constraint specifies the repeating start time for the call event `refresh`. Similarly, the right timing constraint specifies the maximum duration for calls to `getImage`.

Figure 24-4. Modeling Timing Constraint



Often, you'll choose short names for messages so that you don't confuse them with operation names.

Modeling the Distribution of Objects

When you model the topology of a distributed system, you'll want to consider the physical placement of both nodes and artifacts. If your focus is the configuration management of the deployed system, modeling the distribution of nodes is especially important in order to visualize, specify, construct, and document the placement of physical things such as executables, libraries, and tables. If your focus is the functionality, scalability, and throughput of the system, modeling the distribution of objects is what's important.

Deciding how to distribute the objects in a system is a difficult problem, and not just because the problems of distribution interact with the problems of concurrency. Naive solutions tend to yield profoundly poor performance, and over-engineered solutions aren't much better. In fact, they are probably worse because they usually end up being brittle.

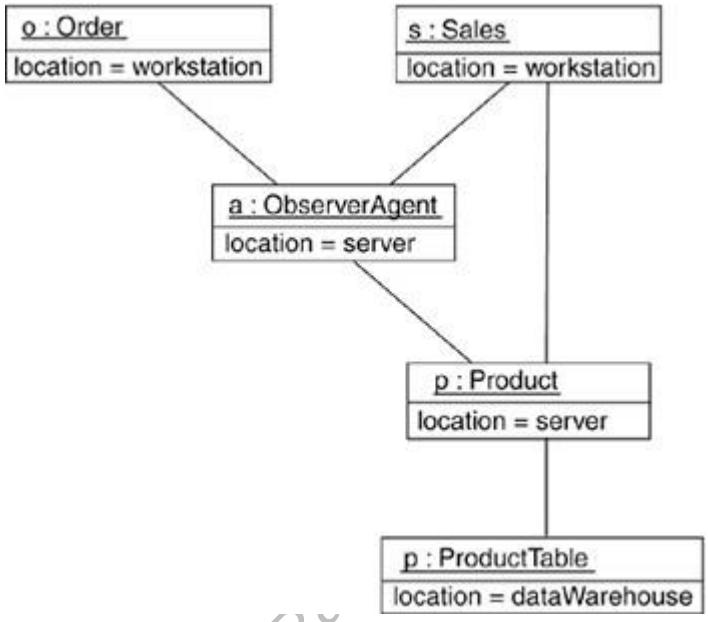
To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects. Colocate sets of objects that have high degrees of interaction, to reduce the cost of communication. 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 artifacts so that tightly coupled objects are on the same artifact.
- Assign artifacts 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 artifacts to the same node.

Figure 24-5 provides an object diagram that models the distribution of certain objects in a retail system. The value of this diagram is that it lets you visualize the physical distribution of certain key objects. As the diagram shows, two objects reside on a `Workstation` (the `Order` and `Sales` objects), two objects reside on a `Server` (the `ObserverAgent` and the `Product` objects), and one object resides on a `DataWarehouse` (the `ProductTable` object).

Figure 24-5. Modeling the Distribution of Objects



State Diagrams

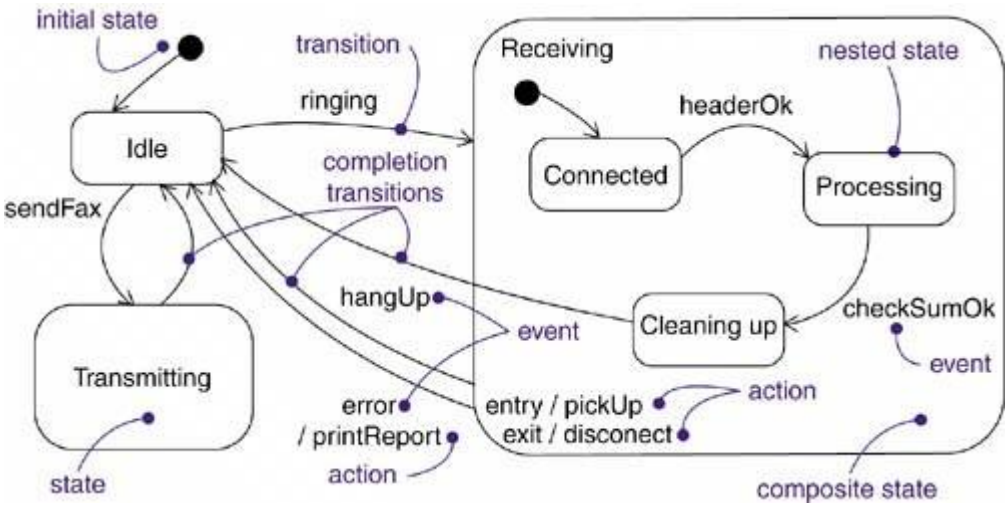
- Modeling reactive objects
- Forward and reverse engineering

State diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. A state diagram shows a state machine. Both activity and state diagrams are useful in modeling the lifetime of an object. However, whereas an activity diagram shows flow of control from activity to activity across various objects, a state diagram shows flow of control from state to state within a single object.

You use state diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the behavior of reactive objects. A reactive object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object has a clear lifetime whose current behavior is affected by its past. State diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, construct, and document the dynamics of an individual object.

State diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
In the UML, you model the event-ordered behavior of an object by using state diagrams. As Figure 25-1 shows, a state diagram is simply a presentation of a state machine, emphasizing the flow of control from state to state.

Figure 25-1. State Diagram



Terms and Concepts

A *state diagram* shows a state machine, emphasizing the flow of control from state to state. 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. 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 *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event 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* specifies an ongoing execution within a state machine. An *action* specifies a primitive executable computation that results in a change in state of the model or the return of a value. Graphically, a state diagram is a collection of nodes and arcs.

Common Properties

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

Contents

State diagrams commonly contain

- Simple states and composite states
- Transitions, events, and actions

Like all other diagrams, state diagrams may contain notes and constraints.

Common Uses

You use state diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the event-ordered behavior of any kind of object in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use a state diagram to model some dynamic aspect of a system, you do so in the context of virtually any modeling element. Typically, however, you'll use state diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach state diagrams to use cases (to model a scenario).

When you model the dynamic aspects of a system, a class, or a use case, you'll typically use state diagrams to model reactive objects.

A reactiveor event-drivenobject is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the next event. For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

Common Modeling Techniques

Modeling Reactive Objects

The most common purpose for which you'll use state diagrams is to model the behavior of reactive objects, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state diagram models the behavior of a single object over its lifetime. Whereas an activity diagram models the flow of control from activity to activity, a state diagram models the flow of control from event to event.

When you model the behavior of a reactive object, you essentially specify three things: the stable states in which that object may live, the events that trigger a transition from state to state, and the actions that occur on each state change. Modeling the behavior of a reactive object also involves modeling the lifetime of an object, starting at the time of the object's creation and continuing until its destruction, highlighting the stable states in which the object may be found.

A stable state represents a condition in which an object may exist for some identifiable period of time. When an event occurs, the object may transition from state to state. These events may also trigger self- and internal transitions, in which the source and the target of the transition are the same state. In reaction to an event or a state change, the object may respond by dispatching an action.

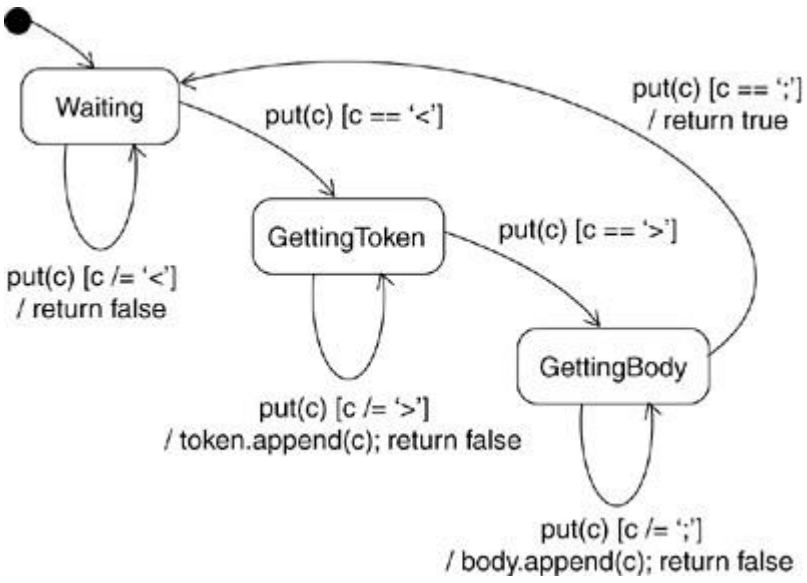
To model a reactive object,

- 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.

For example, Figure 25-2 shows the state diagram for parsing a simple context-free language, such as you might find in systems that stream in or stream out messages to XML. In this case, the machine is designed to parse a stream of characters that match the syntax

```
message : '<' string '>' string ';'
```

Figure 25-2. Modeling Reactive Objects



The first string represents a tag; the second string represents the body of the message. Given a stream of characters, only well-formed messages that follow this syntax may be accepted.

As the figure shows, there are only three stable states for this state machine: `Waiting`, `GettingToken`, and `GettingBody`. This state is designed as a Mealy machine, with actions tied to transitions. In fact, there is only one event of interest in this state machine, the invocation of `put` with the actual parameter `c` (a character). While `Waiting`, this machine throws away any character that does not designate the start of a token (as specified by the guard condition). When the start of a token is received, the state of the object changes to `GettingToken`. While in that state, the machine saves any character that does not designate the end of a token (as specified by the guard condition). When the end of a token is received, the state of the object changes to `GettingBody`. While in that state, the machine saves any character that does not designate the end of a message body (as specified by the guard condition). When the end of a message is received, the state of the object changes to `Waiting`, and a value is returned indicating that the message has been parsed (and the machine is ready to receive another message).

Note that this state specifies a machine that runs continuously; there is no final state.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for state diagrams, especially if the context of the diagram is a class. For example, using the previous state diagram, a forward engineering tool could generate the following Java code for the class `MessageParser`.

```
class MessageParser {
public
 boolean put(char c) {
  switch (state) {
   case Waiting:
    if (c == '<') {
```

```
      state = GettingToken;
      token = new StringBuffer();
      body = new StringBuffer();
     }
     break;
    case GettingToken :
     if (c == '>')
      state = GettingBody;
     else
      token.append(c);
     break;
    case GettingBody :
     if (c == ';')
      state = Waiting;
     else
      body.append(c);
     return true;
   }
   return false;
  }
  StringBuffer getToken() {
   return token;
  }
  StringBuffer getBody() {
   return body;
  }
private
  final static int Waiting = 0;
  final static int GettingToken = 1;
  final static int GettingBody = 2;
  int state = Waiting;
  StringBuffer token, body;
}
```

This requires a little cleverness. The forward engineering tool must generate the necessary private attributes and final static constants.

Reverse engineering (the creation of a model from code) is theoretically possible but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful state diagrams. More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the states in the diagram as they were reached in the running system. Similarly, the firing of transitions could be animated, showing the receipt of events and the resulting dispatch of actions. Under the control of a debugger, you could control the speed of execution, setting breakpoints to stop the action at interesting states to examine the attribute values of individual objects.

6.Components

- Components, interfaces, and realization
- Internal structure, ports, parts, and connectors
- Wiring subcomponents together
- Modeling an API

A component is a logical, replaceable part of a system that conforms to and provides the realization of a set of interfaces.

Good components define crisp abstractions with well-defined interfaces, making it possible to easily replace older components with newer, compatible ones.

Interfaces bridge your logical and design models. For example, you may specify an interface for a class in a logical model, and that same interface will carry over to some design component that realizes it.

Interfaces allow you to build the implementation of a component using smaller components by wiring ports on the components together.

Terms and Concepts

An *interface* is a collection of operations that specify a service that is provided by or requested from a class or component.

A *component* is a replaceable part of a system that conforms to and provides the realization of a set of interfaces.

A *port* is a specific window into an encapsulated component accepting messages to and from the component conforming to specified interfaces.

Internal structure is the implementation of a component by means of a set of parts that are connected together in a specific way.

A *part* is the specification of a role that composes part of the implementation of a component. In an instance of the component, there is an instance corresponding to the part.

A *connector* is a communication relationship between two parts or ports within the context of a component.

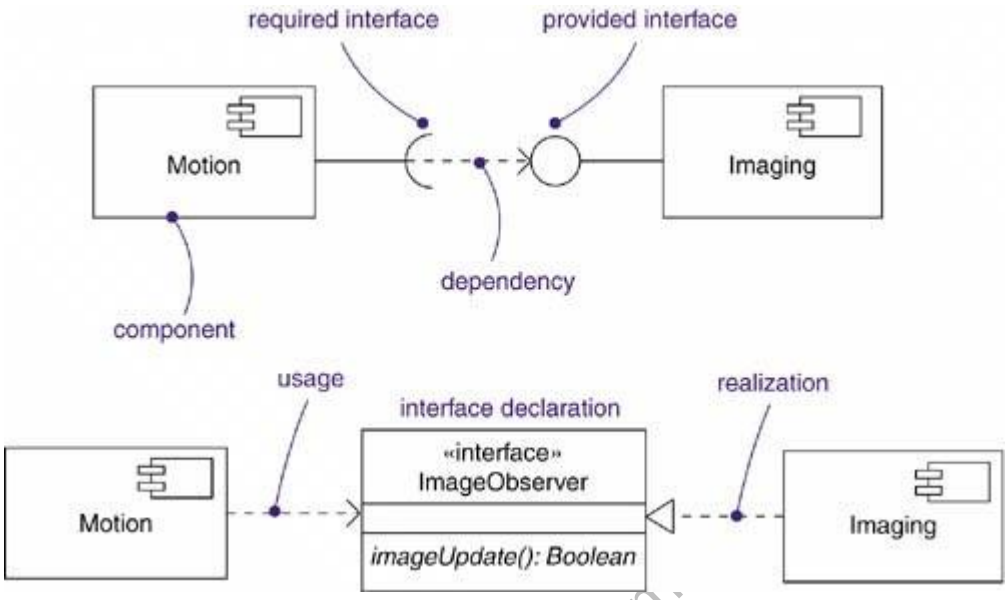
Components and Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.
To construct a system based on components, you decompose your system by specifying interfaces that represent the major seams in the system. You then provide components that realize the interfaces, along with other components that access the services through their interfaces. This mechanism permits you to deploy a system whose services are somewhat location-independent and, as discussed in the next section, replaceable.

An interface that a component realizes is called a *provided interface*, meaning an interface that the component provides as a service to other components. A component may declare many provided interfaces. The interface that a component uses is called a *required interface*, meaning an interface that the component conforms to when requesting services from other components. A component may conform to many required interfaces. Also, a component may both provide and require interfaces.

As Figure 15-1 indicates, a component is shown as a rectangle with a small two-pronged icon in its upper right corner. The name of the component appears in the rectangle. A component can have attributes and operations, but these are often elided in diagrams. A component can show a network of internal structure, as described later in this chapter.

Figure 15-1. Components and Interfaces



You can show the relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form. A provided interface is shown as a circle attached to the component by a line (a "lollipop"). A required interface is shown as a semicircle attached to the component by a line (a "socket"). In both cases, the name of the interface is placed next to the symbol. The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship. The component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

A given interface may be provided by one component and required by another. The fact that this interface lies between the two components breaks the direct dependency between the components. A component that uses a given interface will function properly no matter what component realizes that interface. Of course, a component can be used in a context if and only if all its required interfaces are realized as provided interfaces of other components.

Replaceability

The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable artifacts. This means that you can design a system using components and then implement those components using artifacts. You can then evolve that system by adding new components and replacing old ones, without rebuilding the system. Interfaces are the key to making this happen. In the executable system, you can use any artifacts that implement a component conforming to or providing that interface. You can extend the system by making the components provide new services through other interfaces, which, in turn, other components can discover and use. These semantics explain the intent behind the definition of components in the UML. A component conforms to and provides the realization of a set of interfaces and enables substitutability both in the logical design and in the physical implementation based on it.

A component is *replaceable*. A component is substitutableit is possible to replace a component with another that conforms to the same interfaces. At design time, you choose a different component. Typically, the mechanism of inserting or replacing an artifact in a run time system is transparent to the component user and is enabled by object models (such as COM+ and Enterprise Java Beans) that require little or no intervening transformation or by tools that automate the mechanism.

A component is *part of a system*. A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used. A component is logically and physically cohesive and thus denotes a meaningful structural and/or behavioral chunk of a larger system. A component may be reused across many systems. Therefore, a component represents a fundamental building block on which systems can be designed and composed. This definition is recursivea system at one level of abstraction may simply be a component at a higher level of abstraction.

Finally, as discussed in the previous section, a component *conforms to and provides the realization of a set of interfaces*.

Organizing Components

You can organize components by grouping them in packages in the same manner in which you organize classes.

You can also organize components by specifying dependency, generalization, association (including aggregation), and realization relationships among them.

Components can be built from other components. See the discussion on internal structure later in this chapter.

Ports

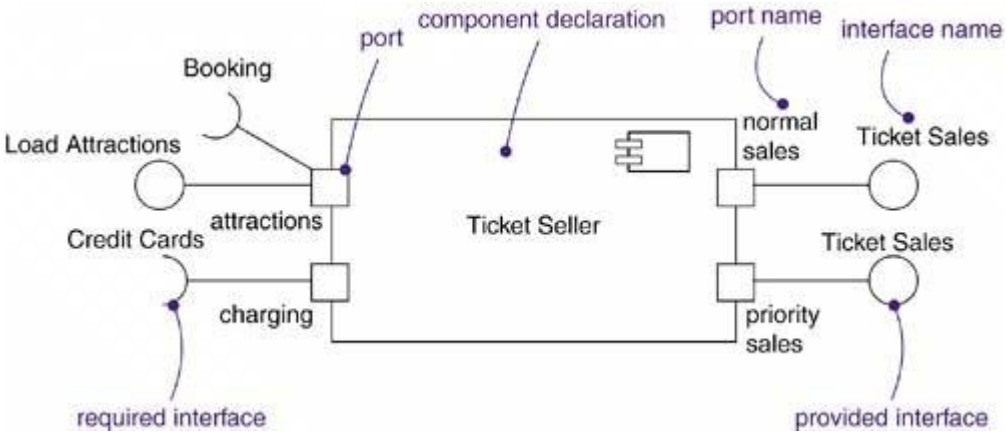
Interfaces are useful in declaring the overall behavior of a component, but they have no individual identity; the implementation of the component must merely ensure that all the operations in all of the provided interfaces are implemented. To have greater control over the implementation, ports can be used.

A *port* is an explicit window into an *encapsulated component*. In an encapsulated component, all of the interactions into and out of the component pass through ports. The externally visible behavior of the component is the sum of its ports, no more and no less. In addition, a port has identity. Another component can communicate with the component through a specific port. The communications are described completely by the interfaces that the port supports, even if the component supports other interfaces. In the implementation, the internal parts of the component may interact through a specific external port, so that each part can be independent of the requirements of the other parts. Ports permit the interfaces of a component to be divided into discrete packets and used independently. The encapsulation and independence provided by ports permit a much greater degree of encapsulation and substitutability.

A port is shown as a small square straddling the border of a componentit represents a hole through the encapsulation boundary of the component. Both provided and required interfaces may be attached to the port symbol. A provided interface represents a service that can be requested through that port. A required interface represents a service that the port needs to obtain from some other component. Each port has a name so that it can be uniquely identified given the component and the port name. The port name can be used by internal parts of the component to identify the port through which to send and receive messages. The component name and port name together uniquely identify a specific port in a specific component for use by other components.

Ports are part of a component. Instances of ports are created and destroyed along with the instance of the component to which they belong. Ports may also have multiplicity; this indicates the possible number of instances of a particular port within an instance of the component. Each port on a component instance has an array of port instances. Although the port instances in an array all satisfy the same interface and accept the same kinds of requests, they may have different states and data values. For example, each instance in an array might have a different priority level, with the higher-priority port instances being served first.
Figure 15-2 shows the model of a `Ticket Seller` component with ports. Each port has a name and, optionally, a type to tell what kind of a port it is. The component has ports for ticket sales, attractions, and credit card charging.

Figure 15-2. Ports on a Component



There are two ports for ticket sales, one for normal customers and one for priority customers. They both have the same provided interface of type `Ticket Sales`. The credit card processing port has a required interface; any component that provides the specified services can satisfy it. The attractions port has both provided and required interfaces. Using the `Load Attractions` interface, a theater can enter shows and other attractions into the ticket database for sale. Using the `Booking` interface, the ticket seller component can query the theaters for the availability of tickets and actually buy the tickets.

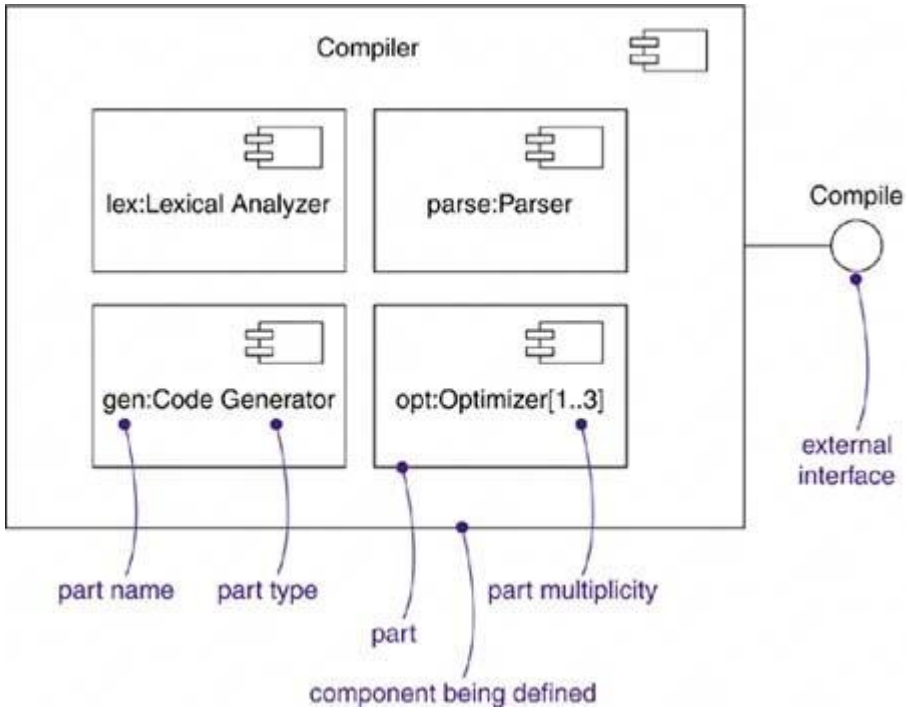
Internal Structure

A component can be implemented as a single piece of code, but in larger systems it is desirable to be able to build large components using smaller components as building blocks. The internal structure of a component is the parts that compose the implementation of the component together with the connections among them. In many cases, the internal parts can be instances of smaller components that are wired together statically through ports to provide the necessary behavior without the need for the modeler to specify extra logic.

A *part* is a unit of the implementation of a component. A part has a name and a type. In an instance of the component, there is one or more instance corresponding to each part having the type specified by the part. A part has a multiplicity within its component. If the multiplicity of the part is greater than one, there may be more than one part instance in a given component instance. If the multiplicity is something other than a single integer, the number of part instances may vary from one instance of the component to another. A component instance is created with the minimum number of parts; additional parts can be added later. An attribute of a class is a kind of part: it has a type and a multiplicity, and each instance of the class has one or more instance of the given type.

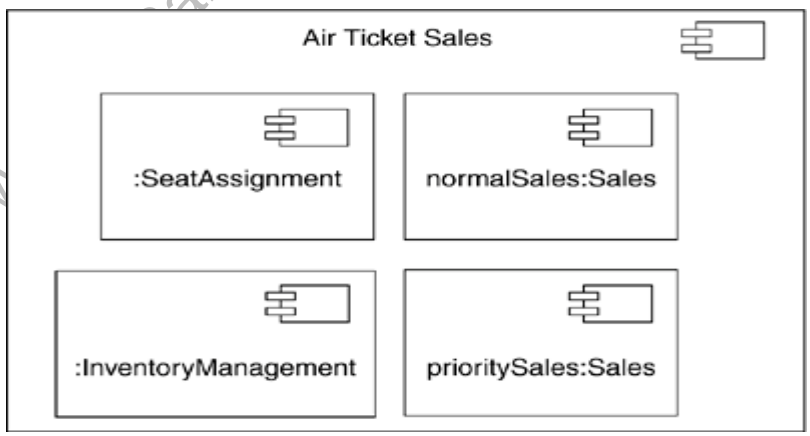
Figure 15-3 shows a compiler component built from four kinds of parts. There is a lexical analyzer, a parser, a code generator, and one to three optimizers. More complete versions of the compiler can be configured with different levels of optimization; within a given version, the appropriate optimizer can be selected at run time.

Figure 15-3. Parts Within a Component



Note that a part is not the same as a class. Each part is potentially distinguishable by its name, just like each attribute in a class is distinguishable. There can be more than one part of the same type, but you can tell them apart by their names, and presumably they have distinct functions within the component. For example, in Figure 15-4 an `Air Ticket Sales` component might have separate `Sales` parts for frequent fliers and for regular customers; they both work the same, but the frequent-flier part is available only to special customers and involves less chance of waiting in line and may provide additional perks. Because these components have the same type, they must have names to distinguish them. The other two components of types `SeatAssignment` and `InventoryManagement` do not require names because there is only one of each type within the `Air Ticket Sales` component.

Figure 15-4. Parts of the Same Type

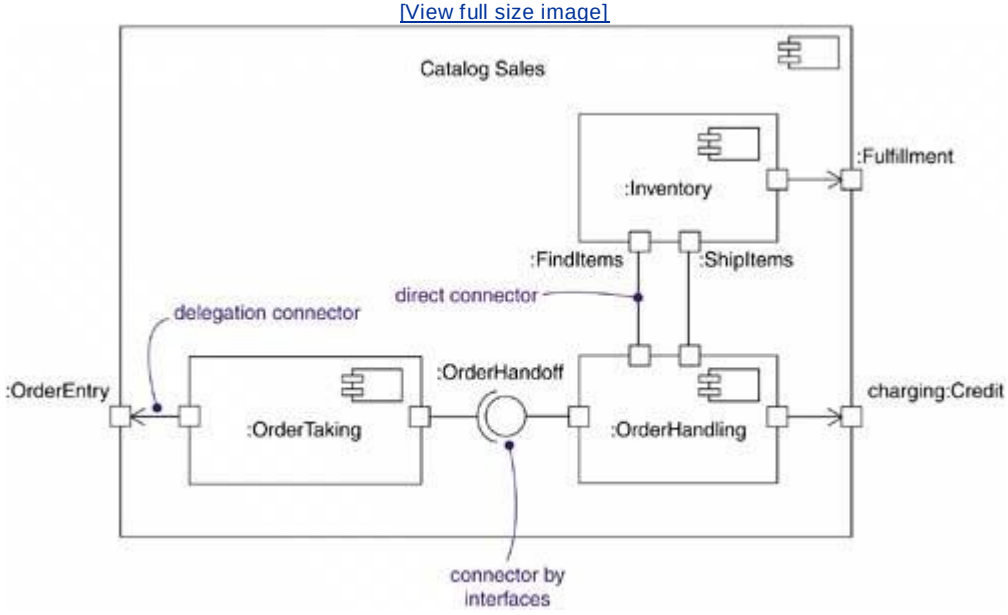


If the parts are components with ports, you can wire them together through their ports. The rule is simple: Two ports can be connected together if one provides a given interface and the other requires the interface. Connecting the ports means that the requiring port will invoke the providing port to obtain services. The advantage of ports and interfaces is that nothing else needs to be known; if the interfaces are compatible, the ports can be connected. A tool could automatically generate calling code from one component to another. They can also be reconnected to other components that provide the same interfaces, if new components become available. A wire between two ports is called a *connector*. In an instance of the overall component, it represents a link or a transient link. A link is an instance of an ordinary association. A transient link represents a usage relationship between two components. Instead of an ordinary association, it might be supplied by a procedure parameter or a local variable that serves as the target of an operation. The advantage of ports and interfaces is that the two components don't have to know about each other at design time, as long as their interfaces are compatible.

You can show connectors in two ways (Figure 15-5). If two components are explicitly wired together, either directly or through ports, just draw a line between them or their ports. On the other hand, if two components are connected because they have compatible interfaces, you can use a ball-and-socket notation to show that there is no inherent relationship between the components, although they are connected inside this component. You could substitute some other component that satisfies the interface.

Figure 15-5. Connectors

[View full size image]



You can also wire internal ports to external ports of the overall component. This is called a delegation connector, because messages on the external port are delegated to the internal port. This is shown by an arrow from an internal port to an external port. You can think of this in two ways, whichever you prefer: In the first approach, the internal port *is* the same as the external port; it has been moved to the boundary and allowed to peek through. In the second approach, any message to the external port is transmitted immediately to the internal port, and vice versa. It really doesn't matter; the behavior is the same in either case.

Figure 15-5 shows an example with internal ports and different kinds of connectors. External requests on the `OrderEntry` port are delegated to the internal port of the `OrderTaking` subcomponent. This component in turn sends its output to its `OrderHandoff` port. This port is connected by a ball-and-socket symbol to the `OrderHandling` subcomponent. This kind of connection implies that there is no special knowledge between the two components; the output could be connected to any component that obeys the `OrderHandoff` interface. The `OrderHandling` component communicates with the Inventory component to find the items in stock. This is shown as a direct connector; because no interfaces are shown, this tends to suggest that the connection is more tightly coupled. Once the items are found in stock, the `OrderHandling` component accesses an external `Credit` service; this is shown

by the delegation connector to the external port called `changing`. Once the external credit service responds, the `OrderHandling` component communicates with a different port `ShipItems` on the `Inventory` component to prepare the order for shipment. The `Inventory` component accesses an external `Fulfillment` service to actually perform the shipment.

Note that the component diagram shows the structure and potential message paths of the component. The component diagram itself does not show the sequencing of messages through the component. Sequencing and other kinds of dynamic information can be shown using interaction diagrams.

Common Modeling Techniques

Modeling Structured Classes

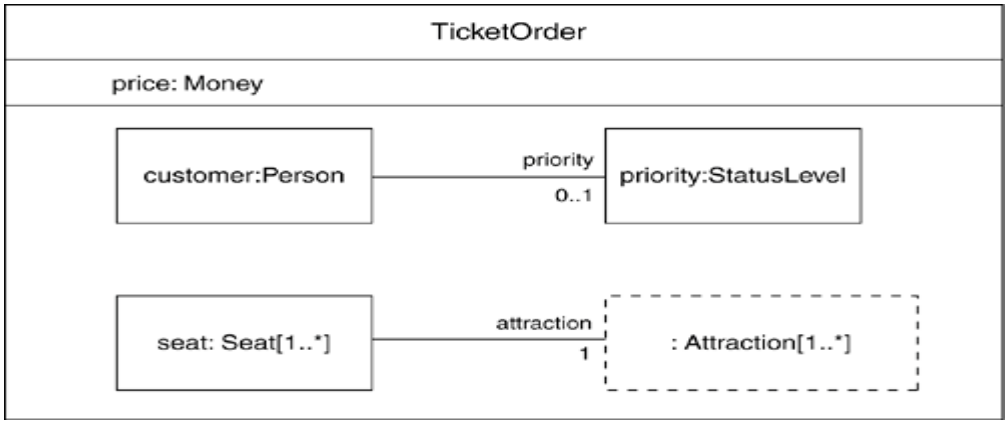
A structured class can be used to model data structures in which the parts have contextual connections that apply only within the class. Ordinary attributes or associations can define composite parts of a class, but the parts cannot be related to each other on a plain class diagram. A class whose internal structure is shown with parts and connectors avoids this problem.

To model a structured class,

- Identify the internal parts of the class and their types.
- Give each part a name that indicates its purpose in the structured class, not its generic type.
- Draw connectors between parts that communicate or have contextual relationships.
- Feel free to use other structured classes as the types of parts, but remember that you can't make connections to parts inside another structured class; connect to its external ports.

Figure 15-6 shows the design of the structured class `TicketOrder`. This class has four parts and one ordinary attribute, `price`. The `customer` is a `Person` object. The `customer` may or may not have a `priority` status, so the `priority` part is shown with multiplicity 0..1; the connector from `customer` to `priority` also has the same multiplicity. There are one or more seats reserved; `seat` has a multiplicity value. It is unnecessary to show a connector from `customer` to `seats` because they are in the same structured class anyway. Notice that `Attraction` is drawn with a dashed border. This means that the part is a reference to an object that is not owned by the structured class. The reference is created and destroyed with an instance of the `TicketOrder` class, but instances of *Attraction* are independent of the `TicketOrder` class. The `seat` part is connected to the `attraction` reference because the order may include seats for more than one `attraction`, and each `seat` reservation must be connected to a specific `attraction`. We see from the multiplicity on the connector that each `Seat` reservation is connected to exactly one `Attraction` object.

Figure 15-6. Structured Class



Modeling an API

If you are a developer who's assembling a system from component parts, you'll often want to see the application programming interfaces (APIs) that you use to glue these parts together. APIs represent the programmatic seams in your system, which you can model using interfaces and components.

An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some* component realizes it. From a system configuration management perspective, though, these realizations are important because you need to ensure that when you publish an API, there's some realization available that carries out the API's obligations. Fortunately, with the UML, you can model both perspectives.

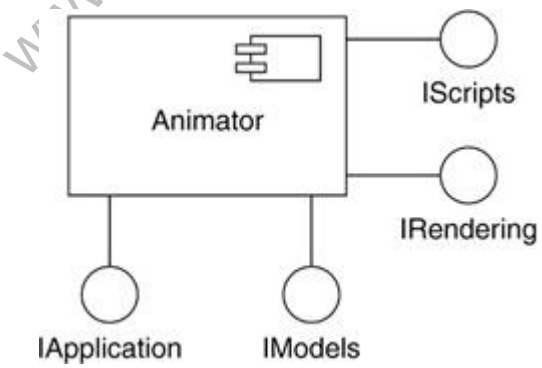
The operations associated with any semantically rich API will be fairly extensive, so most of the time you won't need to visualize all these operations at once. Instead, you'll tend to keep the operations in the backplane of your models and use interfaces as handles with which you can find these sets of operations. If you want to construct executable systems against these APIs, you will need to add enough detail so that your development tools can compile against the properties of your interfaces. Along with the signatures of each operation, you'll probably also want to include uses cases that explain how to use each interface.

To model an API,

- Identify the seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge.
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

Figure 15-7 exposes the APIs of an animation component. You'll see four interfaces that form the API: `IApplication`, `IModels`, `IRendering`, and `IScripts`. Other components can use one or more of these interfaces as needed.

Figure 15-7. Modeling an API



Deployment

In this chapter

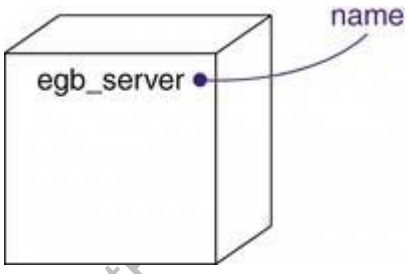
- Nodes and connections
- Modeling processors and devices
- Modeling the distribution of artifacts
- Systems engineering

Nodes, just like artifacts, live in the material world and are an important building block in modeling the physical aspects of a system. 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.

You use nodes to model the topology of the hardware on which your system executes. A node typically represents a processor or a device on which artifacts may be deployed. Good nodes crisply represent the vocabulary of the hardware in your solution domain.

The UML provides a graphical representation of node, as Figure 27-1 shows. This canonical notation permits you to visualize a node apart from any specific hardware. Using stereotypesone of the UML's extensibility mechanismsyou can (and often will) tailor this notation to represent specific kinds of processors and devices.

Figure 27-1. Nodes



Terms and Concepts

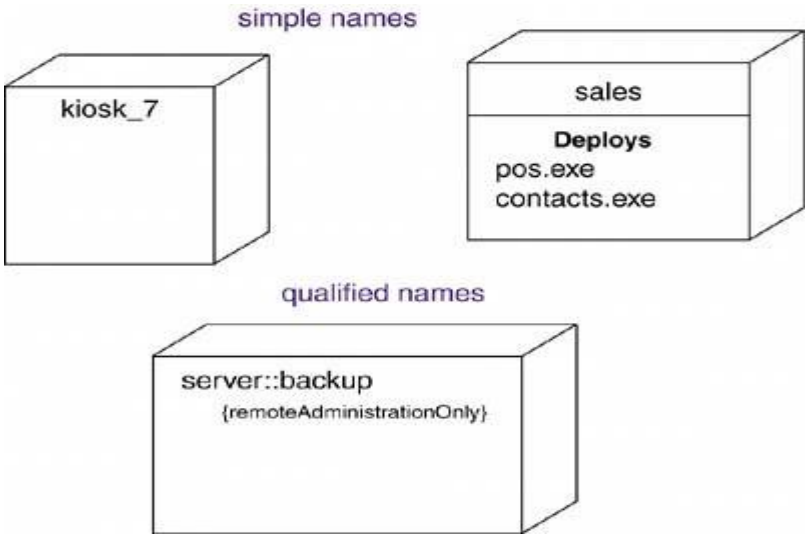
A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.

Names

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

A node is typically drawn showing only its name, as in Figure 27-2. Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.

Figure 27-2. Nodes with Simple and Qualified Names



Nodes and Artifacts

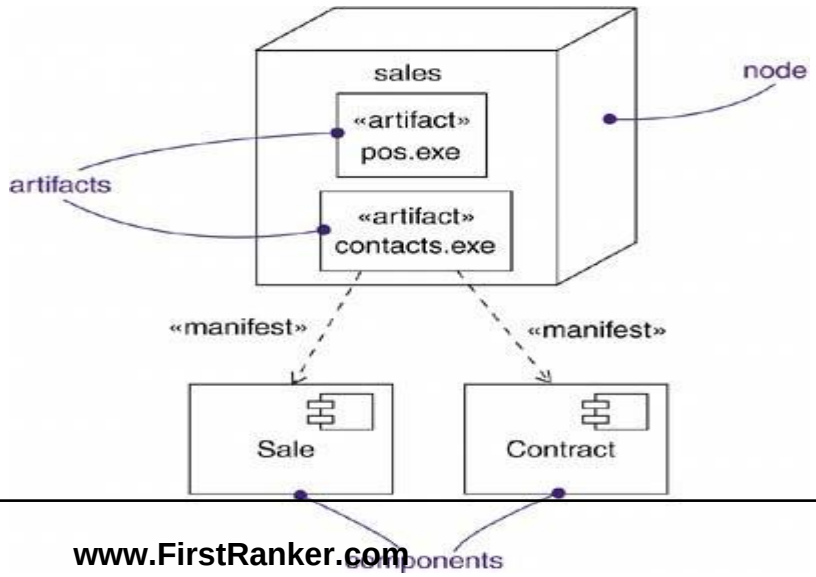
In many ways, nodes are a lot like artifacts: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between nodes and artifacts.

- Artifacts are things that participate in the execution of a system; nodes are things that execute artifacts.
- Artifacts represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of artifacts.

This first difference is the most important. Simply put, nodes execute artifacts; artifacts are things that are executed by nodes.

The second difference suggests a relationship among classes, artifacts, and nodes. In particular, an artifact is the manifestation of a set of logical elements, such as classes and collaborations, and a node is the location upon which artifacts are deployed. A class may be manifested by one or more artifacts, and, in turn, an artifact may be deployed on one or more nodes. As Figure 27-3 shows, the relationship between a node and the artifacts it deploys can be shown explicitly by using nesting. Most of the time, you won't need to visualize these relationships graphically but will indicate them as a part of the node's specification, for example, using a table.

Figure 27-3. Nodes and Artifacts



A set of objects or artifacts that are allocated to a node as a group is called a *distribution unit*.

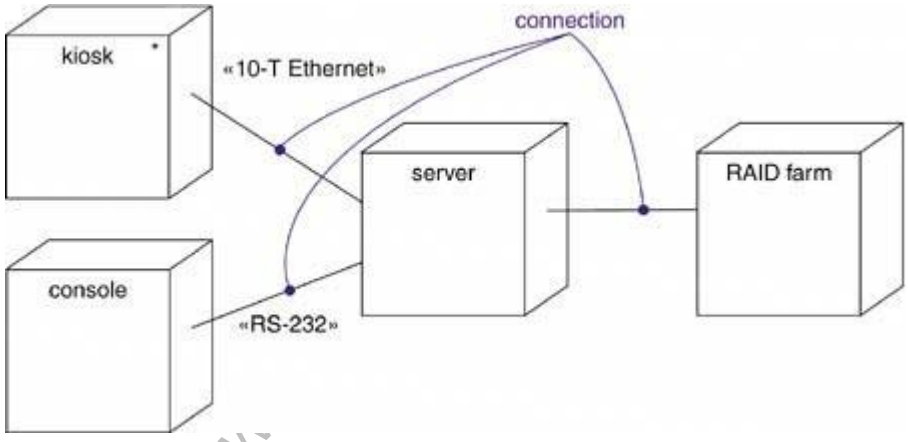
Organizing Nodes

You can organize nodes by grouping them in packages in the same manner in which you can organize classes and artifacts.
You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship you'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus, as Figure 27-4 shows. You can even use associations to model indirect connections, such as a satellite link between distant processors.

Figure 27-4. Connections



Because nodes are class-like, you have the full power of associations at your disposal. This means that you can include roles, multiplicity, and constraints. As in the previous figure, you should stereotype these associations if you want to model new kinds of connectionsfor example, to distinguish between a 10-T Ethernet connection and an RS-232 serial connection.

Common Modeling Techniques

Modeling Processors and Devices

Modeling the processors and devices that form the topology of a stand-alone, embedded, client/server, or distributed system is the most common use of nodes.

Because all of the UML's extensibility mechanisms apply to nodes, you will often use stereotypes to specify new kinds of nodes that you can use to represent specific kinds of

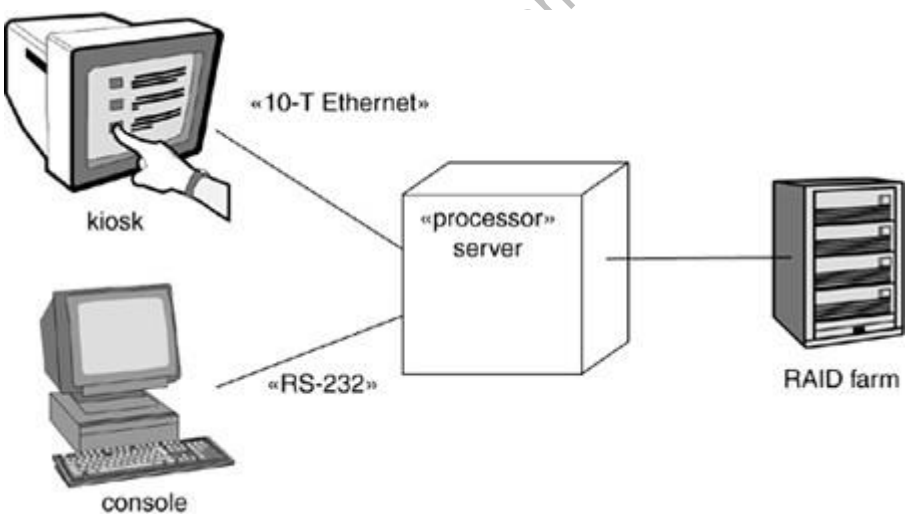
processors and devices. A *processor* is a node that has processing capability, meaning that it can execute a artifact. A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

To model processors and devices,

- Identify the computational elements of your system's deployment view and model each as a node.
- If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
- As with class modeling, consider the attributes and operations that might apply to each node.

For example, Figure 27-5 takes the previous diagram and stereotypes each node. The `server` is a node stereotyped as a generic processor; the `kiosk` and the `console` are nodes stereotyped as special kinds of processors; and the `RAID farm` is a node stereotyped as a special kind of device.

Figure 27-5. Processors and Devices



Modeling the Distribution of Artifacts

When you model the topology of a system, it's often useful to visualize or specify the physical distribution of its artifacts across the processors and devices that make up the system.

To model the distribution of artifacts,

- For each significant artifact in your system, allocate it to a given node.

- Consider duplicate locations for artifacts. It's not uncommon for the same kind of artifact (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.

1. Don't make the allocation visible, but leave it as part of the backplane of your modelthat is, in each node'sspecification.
2. Using dependency relationships, connect each node with the artifacts it deploys.
3. List the artifacts deployed on a node in an additional compartment.

Using the third approach, Figure 27-6 takes the earlier diagrams and specifies the executable artifacts that reside on each node. This diagram is a bit different from the previous ones in that it is an object diagram, visualizing specific instances of each node. In this case, the `RAID farm` and `kiosk` instances are both anonymous and the other two instances are named (`c` for the `console` and `s` for the `server`). Each processor in this figure is rendered with an additional compartment showing the artifact it deploys. The `server` object is also rendered with its attributes (`processorSpeed` and `memory`) and their values visible. The deployment compartment can show a text list of artifact names, or it can show nested artifact symbols.

Figure 27-6. Modeling the Distribution of Artifacts