UNIT-I

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing Complex Systems, Evolution of Object Model, Foundation of Object Model, Elements of Object Model, Applying the Object Model.

COMPLEXITY

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

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

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

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

THE STRUCTURE OF COMPLEX SYSTEMS

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

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

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

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

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

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

THE INHERENT COMPLEXITY OF SOFTWARE

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

Some software systems are simple.

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

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

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

Some software systems are complex.

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

Why Software is inherently Complex

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

1. The complexity of the problem domain

- Complex requirements
- Decay of system

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

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

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

2. The Difficulty of Managing the Development Process

- Management problems
- Need of simplicity

The second reason is the complexity of the software development process. Complex software intensive systems cannot be developed by single individuals. They require teams of developers. This adds extra overhead to the process since the developers have to communicate with each other about the intermediate artifacts they produce and make them interoperable with each other. This complexity often gets even more difficult to handle if the teams do not work in one location

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

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

3. The flexibility possible through software

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

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

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

4. The problem of characterizing the behavior of discrete systems

- Numerous possible states
- Difficult to express all states

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

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

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

THE FIVE ATTRIBUTES OF A COMPLEX SYSTEM

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

1. Hierarchical and interacting subsystems

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

2. Arbitrary determination of primitive components

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

3. Stronger intra-component than inter-component link

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

4. Combine and arrange common rearranging subsystems

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

5. Evolution from simple to complex systems

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

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

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

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

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

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

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

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

ORGANIZED AND DISORGANIZED COMPLEXITY

Simplifying Complex Systems

- Usefulness of abstractions common to similar activities

e.g. driving different kinds of motor vehicle

- Multiple orthogonal hierarchies

e.g. structure and control system

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

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

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

Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other. In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, and electrical). In object-orientation, the class structure and the object structure relationship is the simplest form of related hierarchy. It forms a canonical representation for object oriented analysis.

The canonical form of a complex system – the discovery of common abstractions and mechanisms greatly facilitates are standing of complex system. For example, if a pilot already knows how to fly a given aircraft, it is easier to know how to fly a similar one. May different hierarchies are present within the complex system. For example an aircraft may be studied by

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

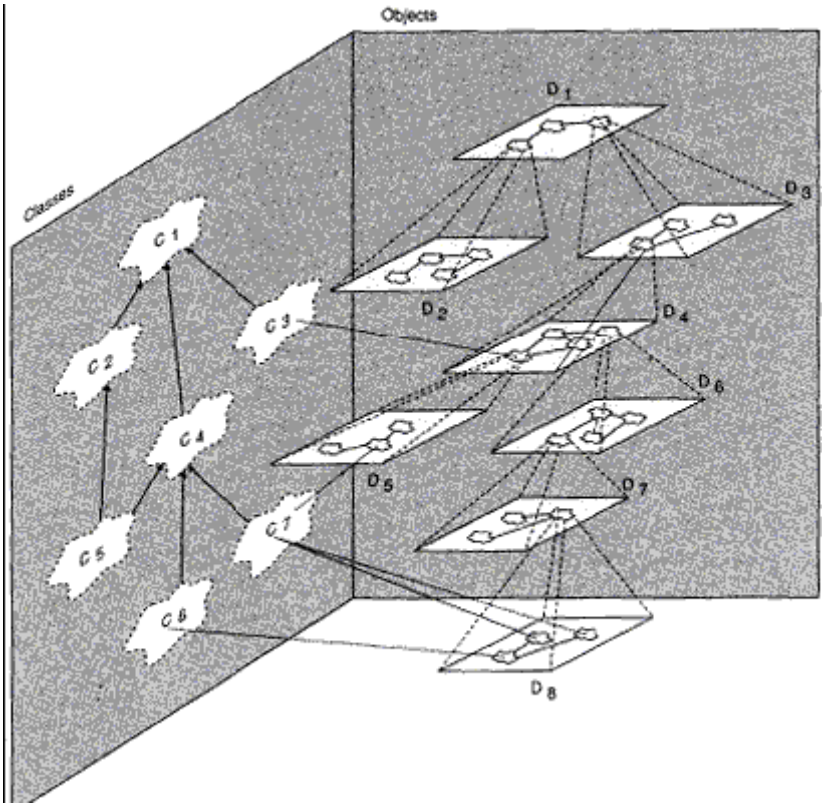


Figure: Canonical form of a complex system

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

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

APPROACHING A SOLUTION

Hampered by human limitations

- dealing with complexities
- memory
- communications

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

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

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

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

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

BRINGING ORDER TO CHAOS

Principles that will provide basis for development

- Abstraction
- Hierarchy
- Decomposition

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

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

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

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

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

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

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

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

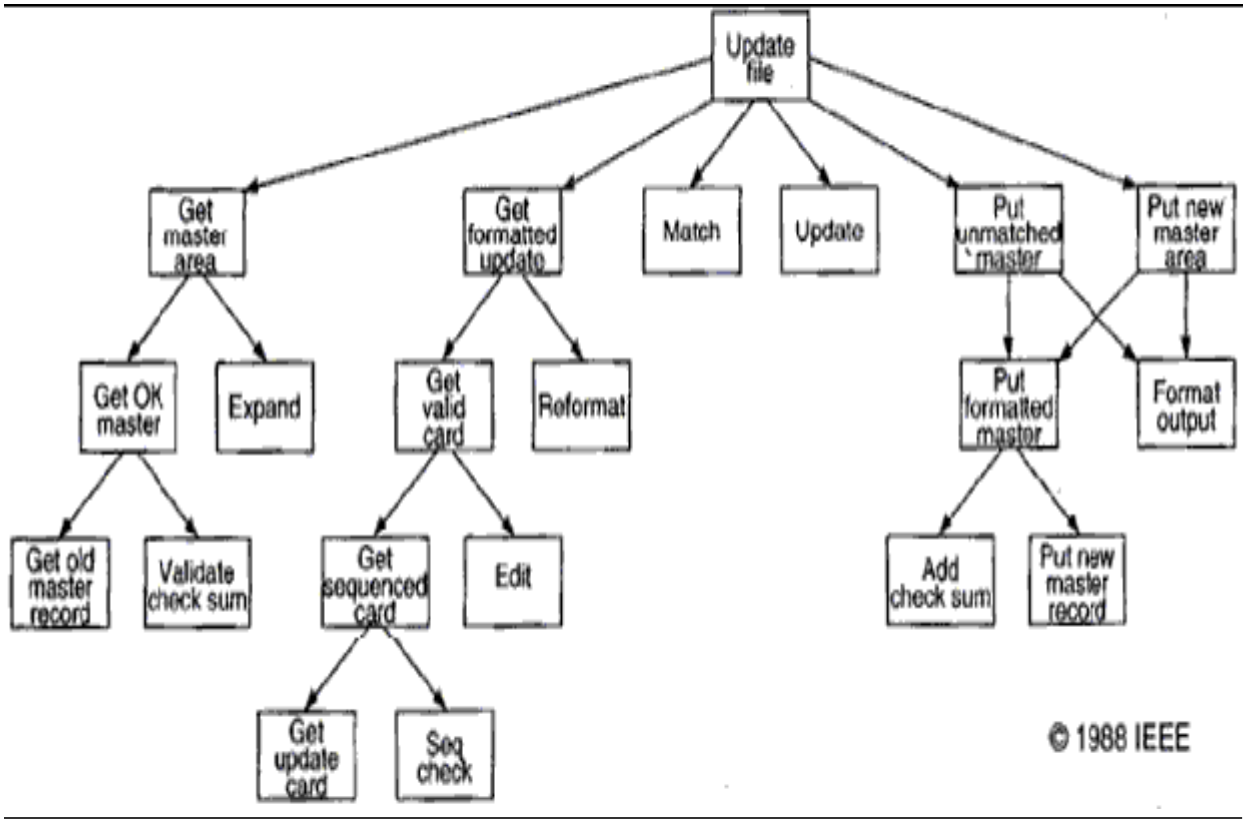


Figure: Algorithmic decomposition

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

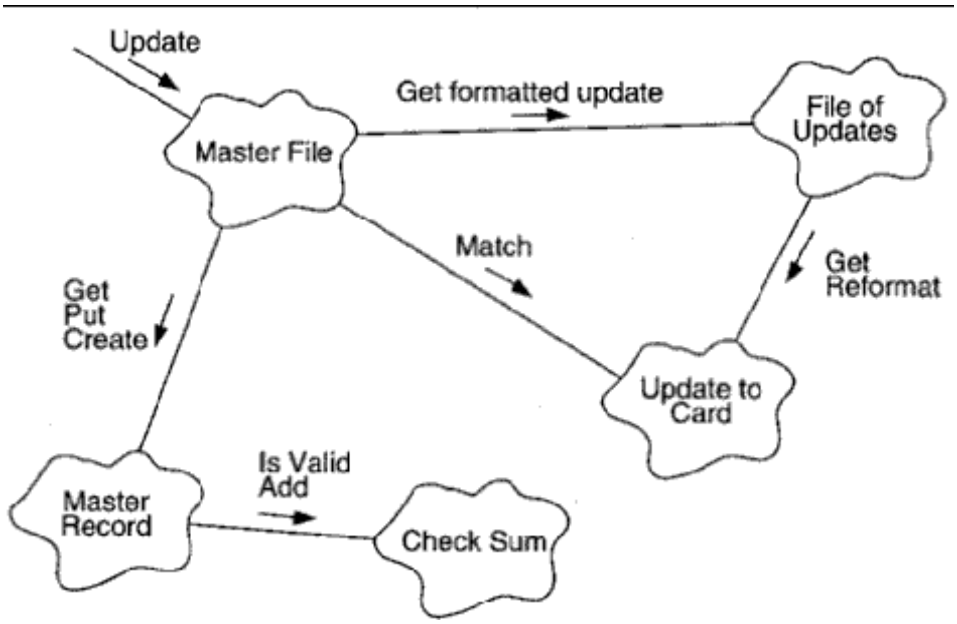


Figure: Object oriented decomposition

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

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

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

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

ON DESIGNING COMPLEX SYSTEMS

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

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

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

for doing the design.

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

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

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

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

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

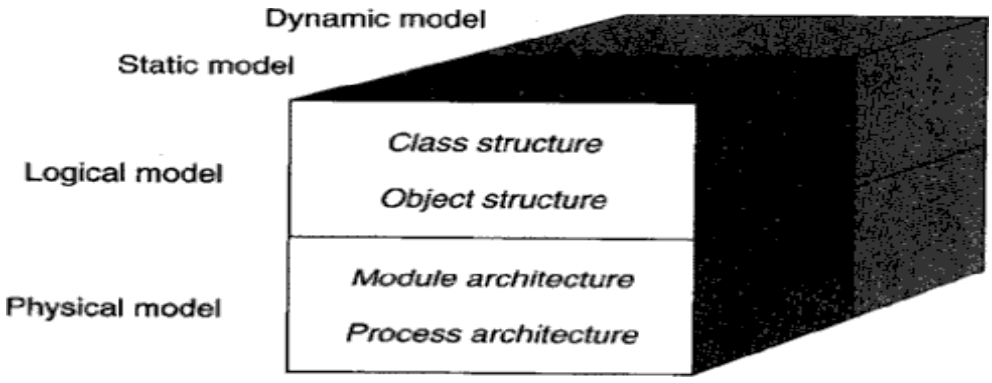


Figure: Models of object oriented development

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

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

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

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.

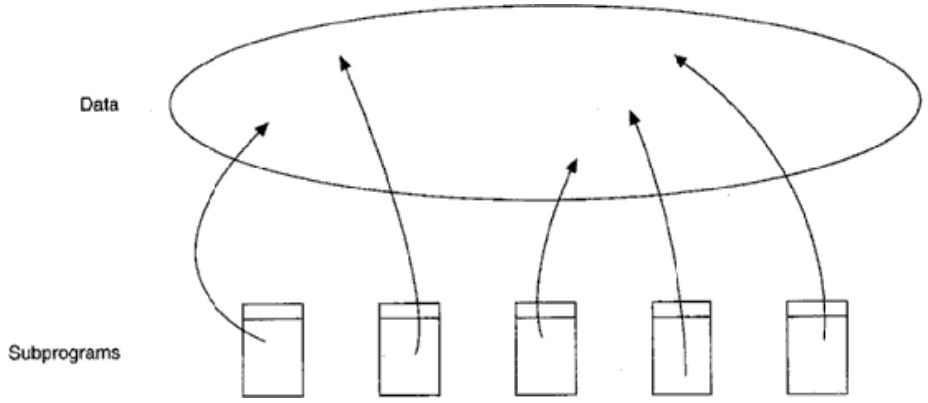


Fig: The Topology of First- and Early Second-Generation Programming Languages

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.

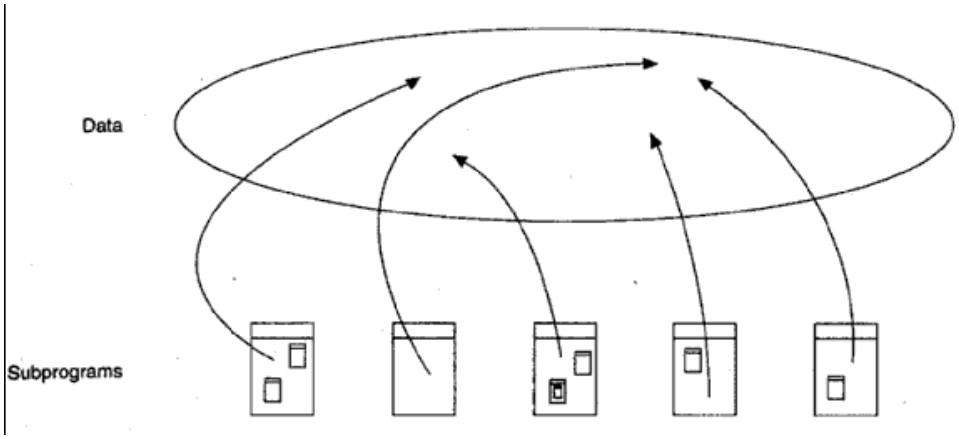


Fig: The Topology of Late Second- and Early Third-Generation Programming Languages

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.

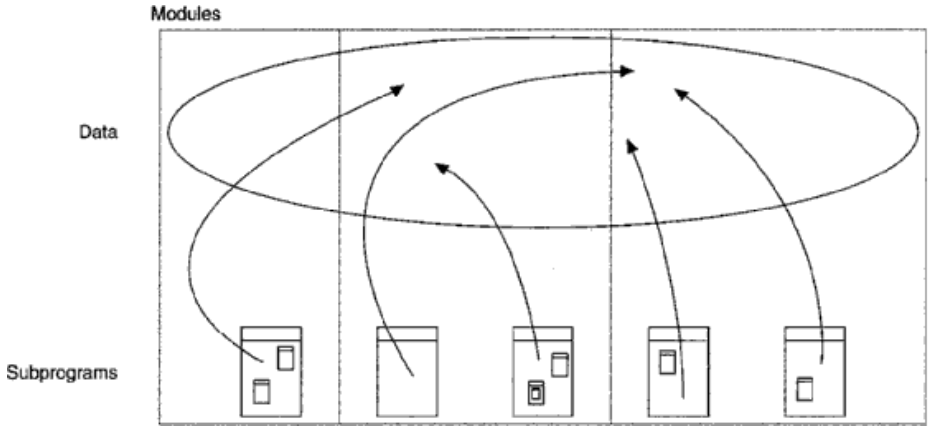


Fig: The Topology of Late Third-Generation Programming Languages

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.

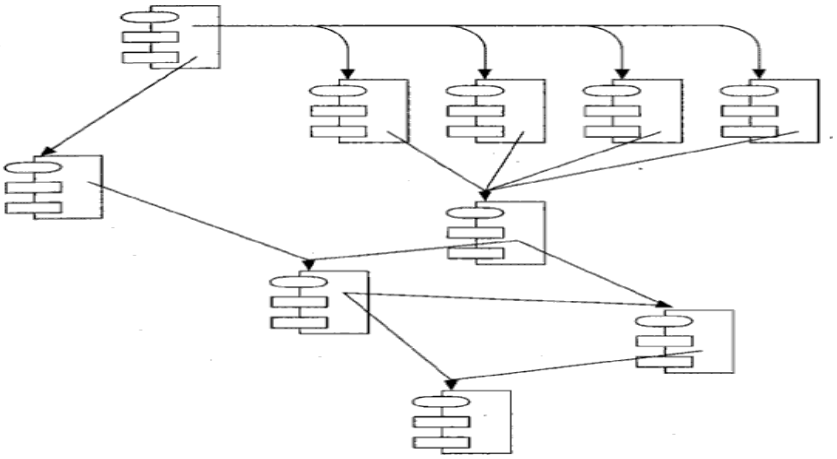


Fig: The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages.

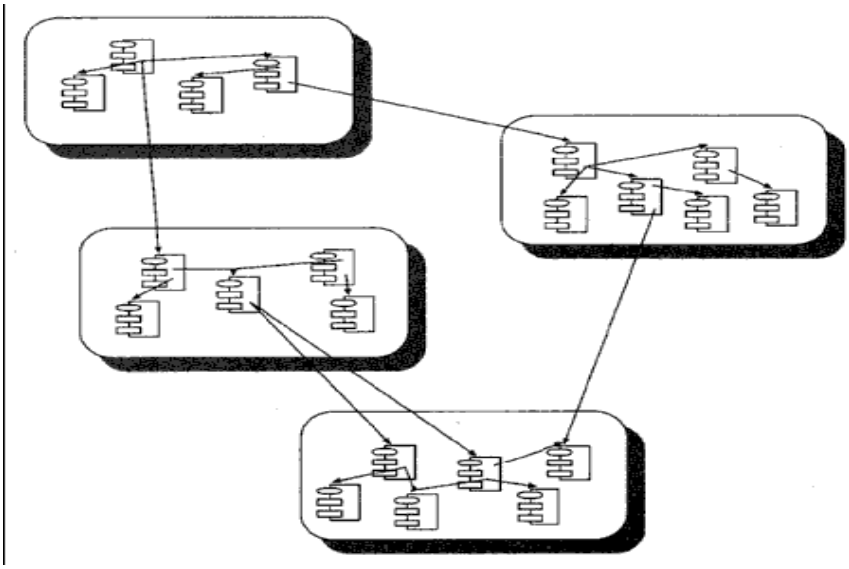


Fig: The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

FOUNDATION OF 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.

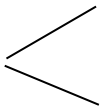
- Entity abstraction: An object that represents a useful model of a problem domain or solution domain entity.
- Action abstraction: An object that provides a generalized set of operations all of which program the same kind of function.
- Virtual machine abstractions: An object that groups together operations 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: Abstraction of a Temperature Sensor

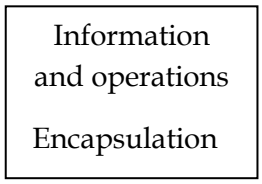
Encapsulation

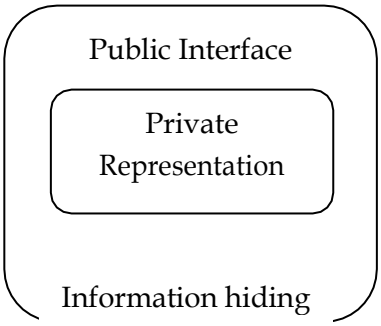
The act of grouping data and operations into a single object.



Private Public

Class





Class heater { Public:

heater (location):

~ heater ( ): void turnon ( ); void turnoff ( );

Boolean ison () const private:

Abstraction: Heater
Important Characteristics : location
status

Figure: 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.

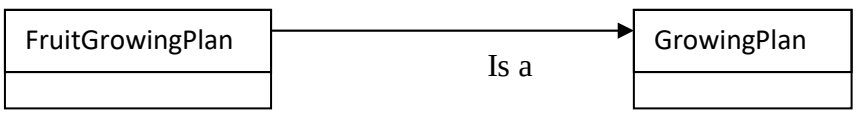


Fig: Class having one superclass (Single Inheritance)

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

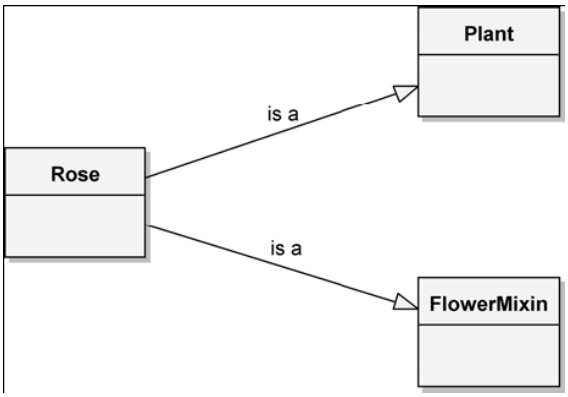


Figure: The Rose Class, Which Inherits from Multiple Superclasses (Multiple Inheritance)

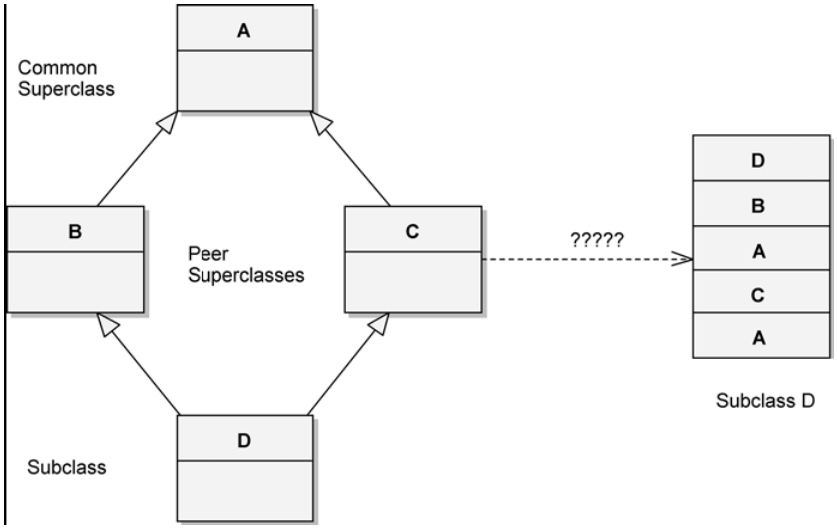


Figure: The Repeated Inheritance

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 Growing Plan 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 Growing Plan; when we destroy the Garden object, we in turn destroy the Growing Plan 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 cannot 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 i.e. 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 mis used.
- 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.

IMPORTANT QUESTIONS

1. Discuss about the object oriented modelling in detailed.
2. Explain the structure of the complex system? What are the attributes of the complex system?
3. Explain the designing of the complex system?
4. Explain evolution and foundation of the object model?

UNIT – II

Classes and Objects: Nature of object, Relationships among objects, Nature of a Class, Relationship among Classes, Interplay of Classes and Objects, Identifying Classes and Objects, Importance of Proper Classification, Identifying Classes and Objects, Key abstractions and Mechanisms.

INTRODUCTION

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 THE 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 in 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;};
```

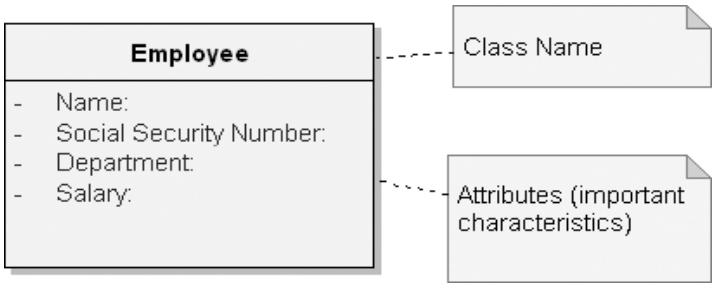


Figure: Employee Class with Attributes

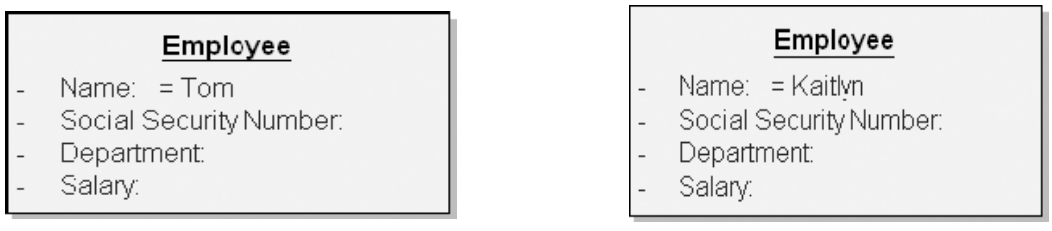


Figure: Employee Objects Tom and Kaitlyn

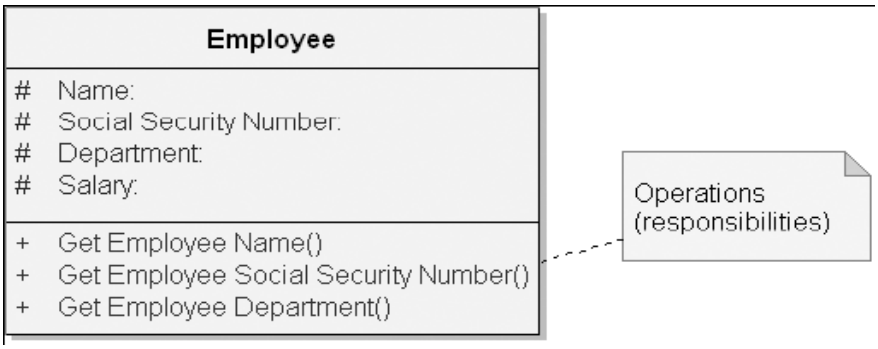


Figure: Employee Class with Protected Attributes and Public Operations

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

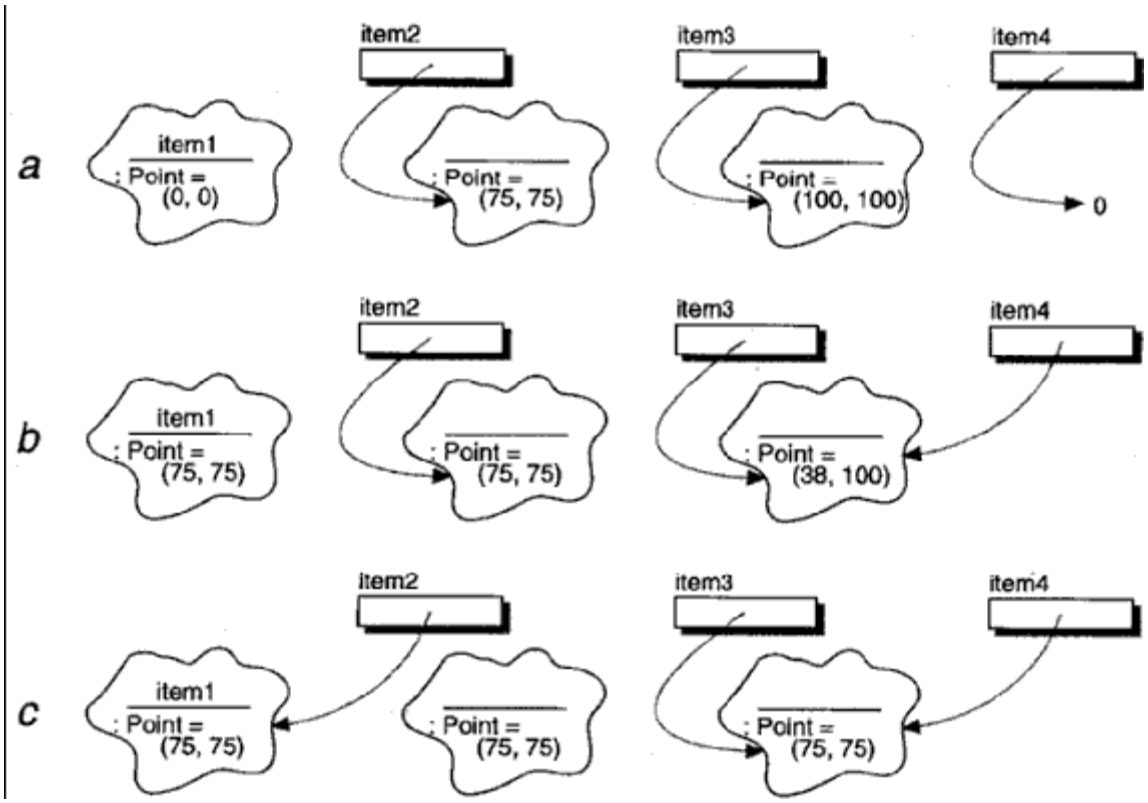


Figure: Object Identity

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.

RELATIONSHIPS 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 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.

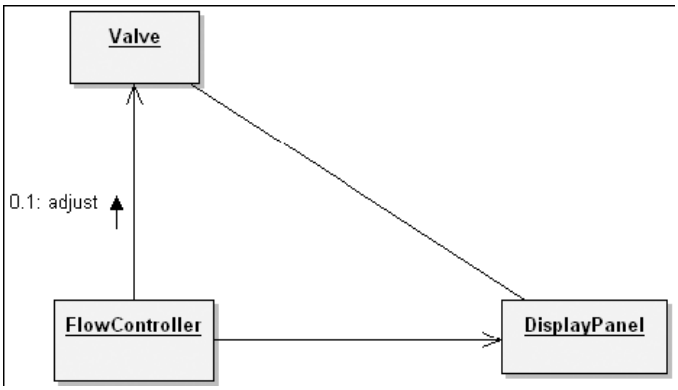


Figure: Links

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.

1. 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.

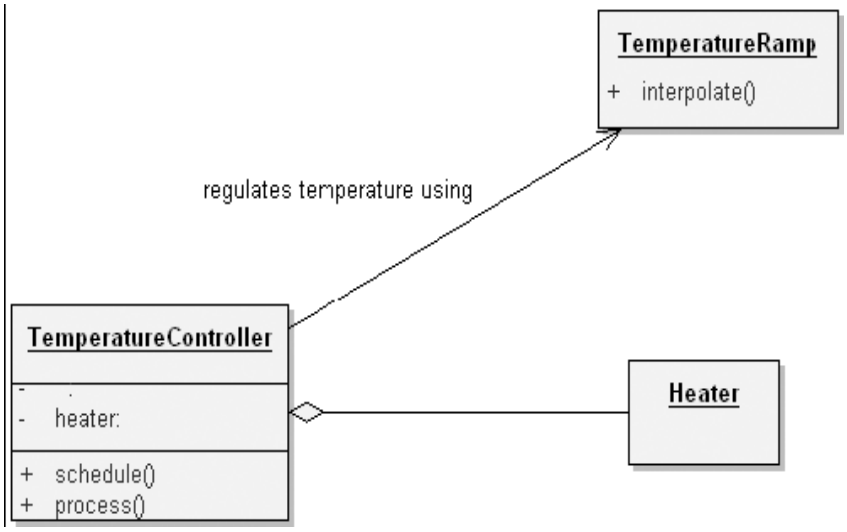


Figure: Aggregation

THE NATURE OF A 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 it's 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

RELATIONSHIPS 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 the following figure, we may show a simple association between these two

classes: the class Wheel and the class Vehicle.

Figure: 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.

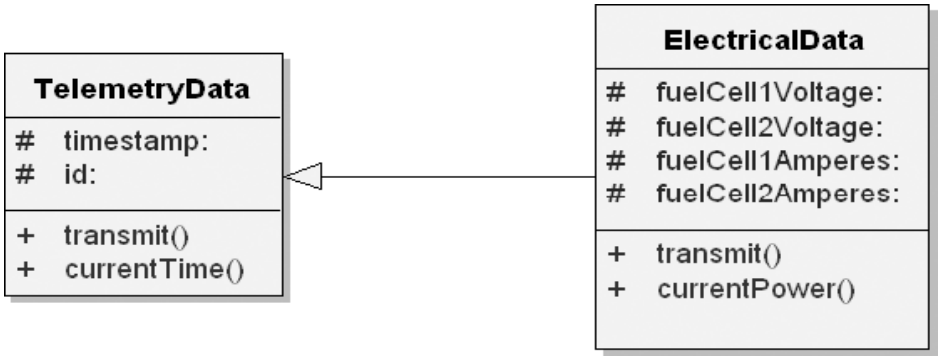


Figure: ElectricalData Inherits from the Superclass TelemetryData

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

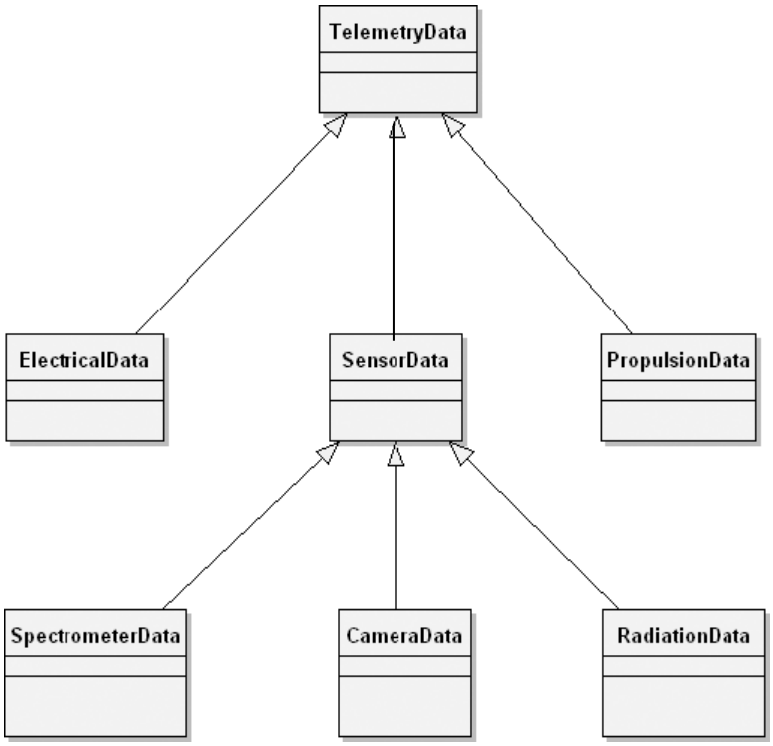


Figure: Single Inheritance

The above figure 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

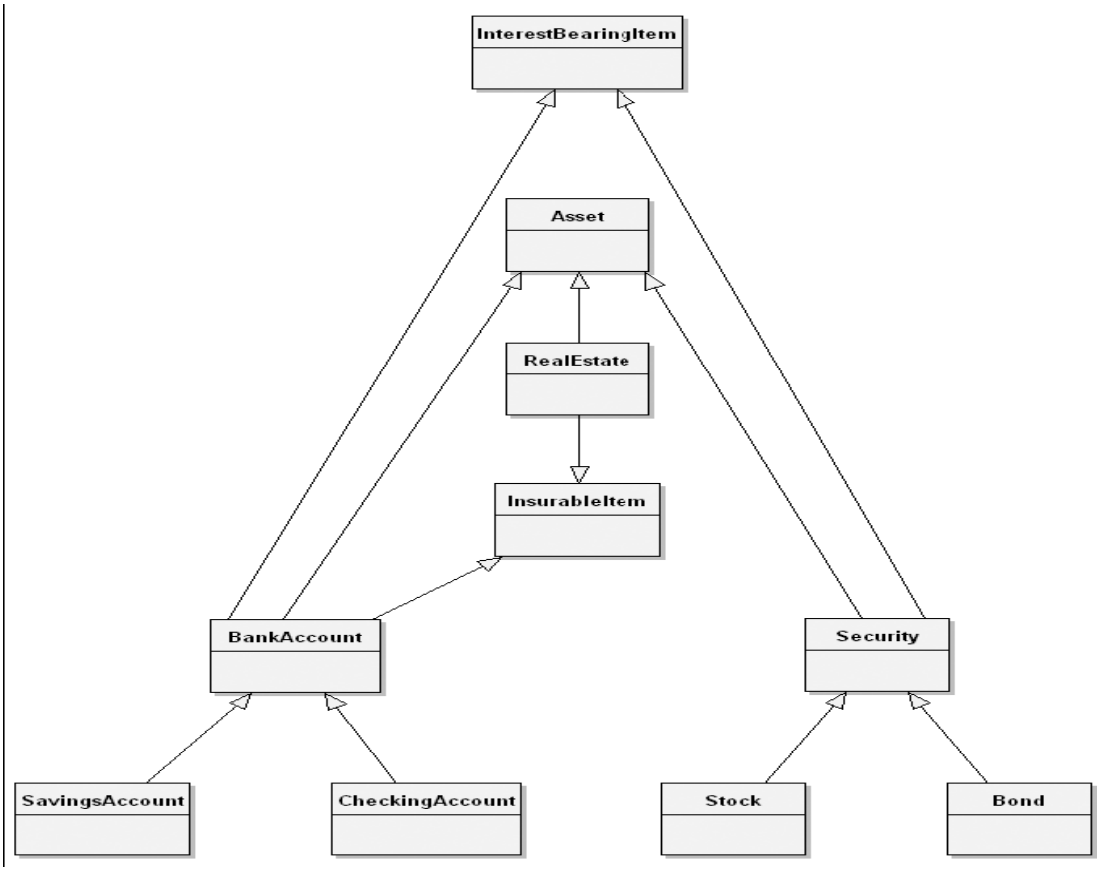


Figure: Multiple Inheritance

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.

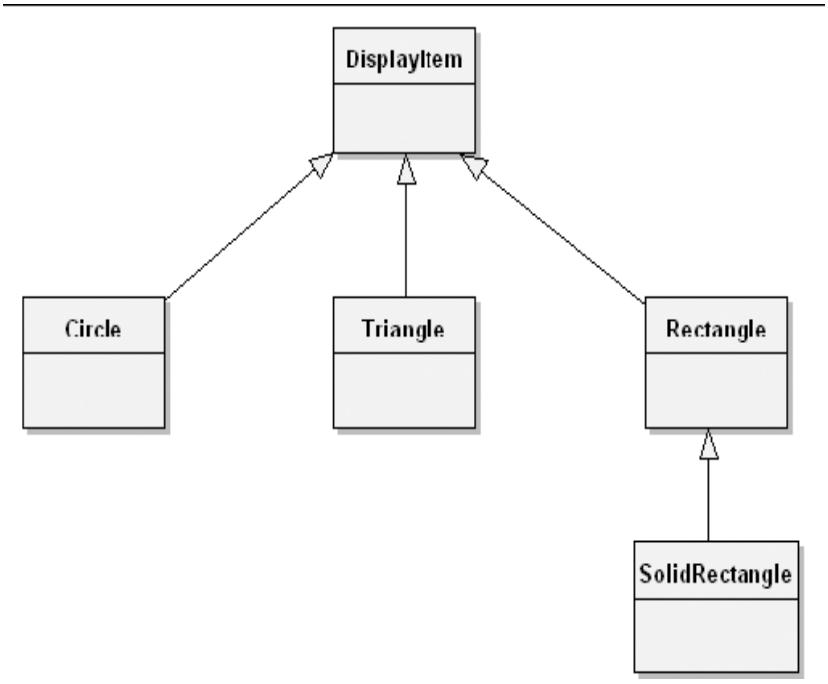


Figure: Polymorphism

Consider the class hierarchy in the above figure, 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.

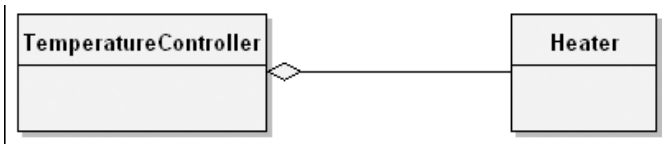


Figure: Aggregation

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.

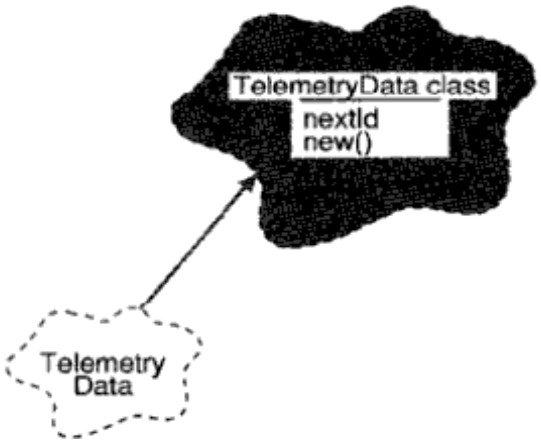


Figure: Metaclass

As shown in the above figure, 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 meta class operations.

IMPORTANCE OF PROPER CLASSIFICATION

Classification is the means whereby 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 cannot 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 super classes 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 systems 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 key abstractions 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 grain size 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 sub views, the window next tells each if its sub views to draw them. Each sub view in turn tells the model to draw itself ultimately resulting in an image shown to the user.

IMPORTANT QUESTIONS

1. Explain the relationships among objects?
2. Explain the relationships among classes?
3. Explain the importance of proper classification?
4. Describe key abstractions and mechanisms?

UNIT-III

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

WHY WE MODEL

A model is a simplification at some level of abstraction

1. Importance of Modeling:

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

To help us visualize

To specify structure or behaviour

To provide template for building system

To document decisions we have made

2. Principles of Modeling:

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

The best models are connected to reality

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

UML is a visual modeling language

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

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

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

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

language for software blueprints."

Usages of UML: UML is used to

i. document designs

design patterns / frameworks

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

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

Object Oriented Modeling:

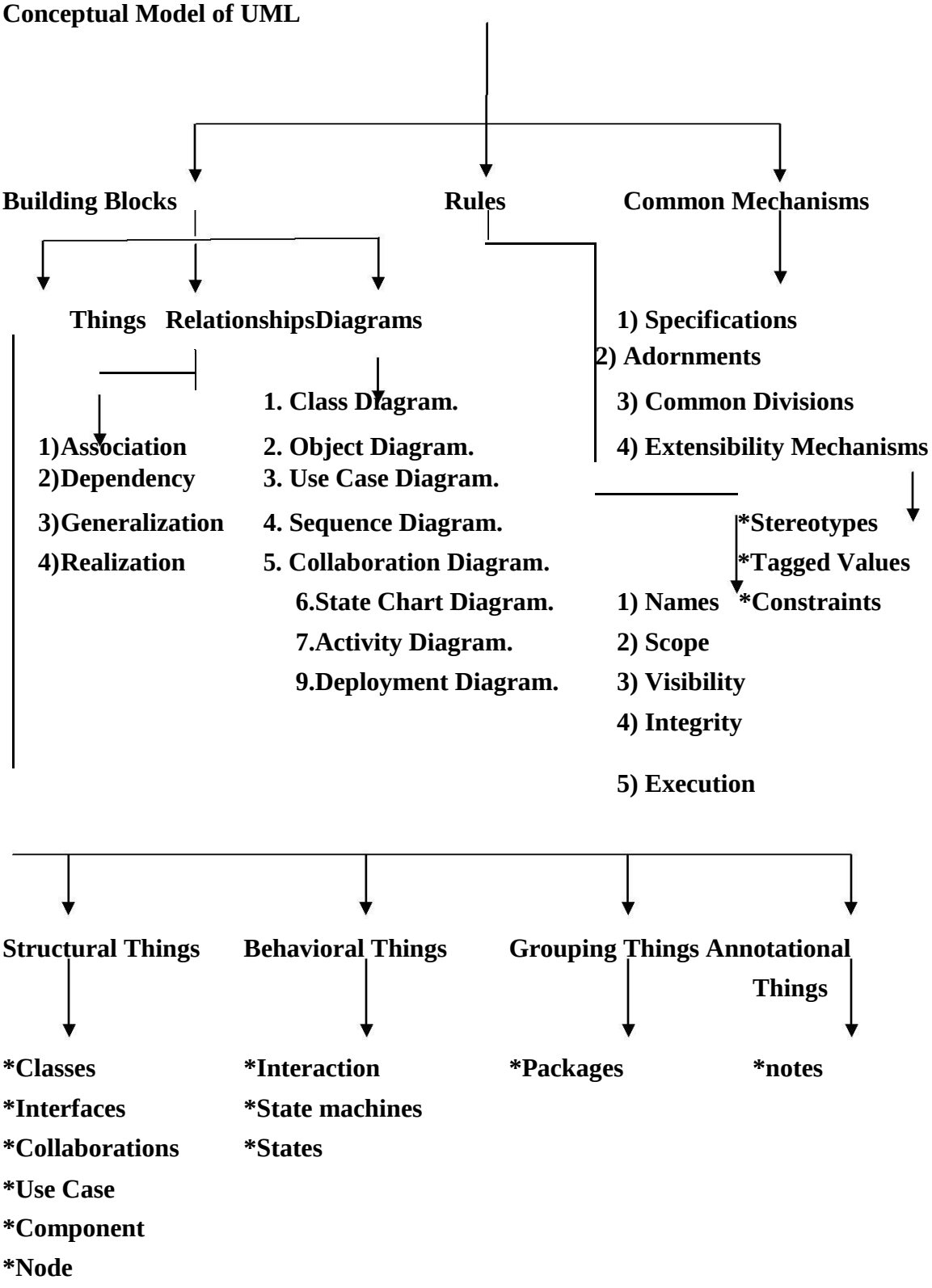
Traditionally two approaches to modeling a software system

Algorithmically – becomes hard to focus on as the requirements change

Object-oriented – models more closely real world entities

CONCEPTUAL MODEL OF THE UML

Conceptual Model of UML



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

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:
1. Things
2. Relationships
3. Diagrams

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

Things in the UML

There are four kinds of things in the UML:

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

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

Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. Collectively, the structural things are called *classifiers*.

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations, as shown in the following figure.

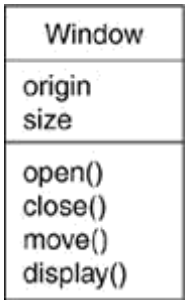
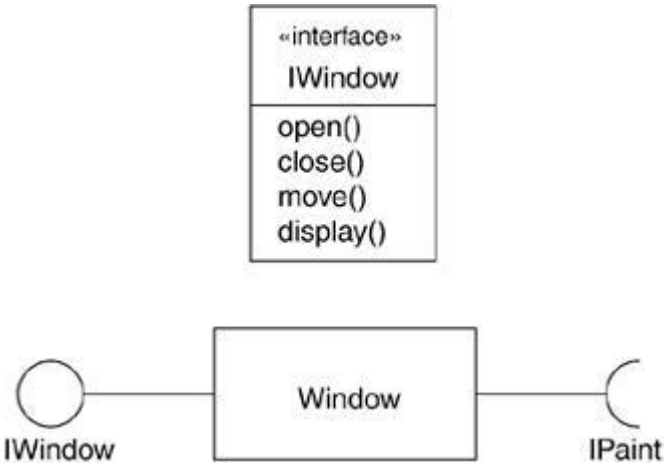


Figure: Classes

An *interface* is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. The declaration of an interface looks like a class with the keyword «interface» above the name; attributes are not relevant, except sometimes to show constants. An interface rarely stands alone, however. An interface provided by a class to the outside world is shown as a small circle attached to the class box by a line. An interface required by a class from some other class is shown as a small semicircle attached to the class box by a line, as shown in the following figure.

Figure: Interfaces



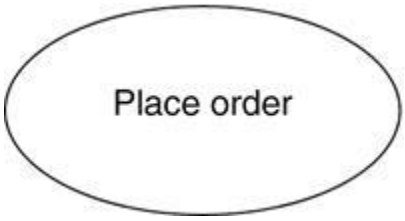
A *collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Collaborations have structural, as well as behavioral, dimensions. A given

class or object might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, sometimes including only its name, as shown in the following figure.

Figure: Collaborations

A *use case* is a description of sequences of actions that a system performs that yield observable results of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name, as shown in the following figure.

Figure: Use Cases



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

An *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered as a class with double lines on the left and right; it

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

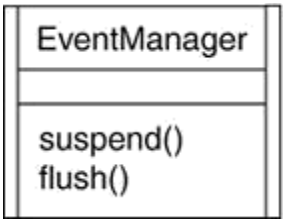
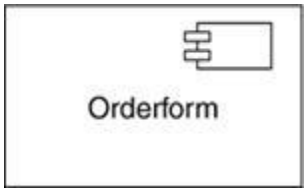


Figure: Active Classes

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

Figure: Components



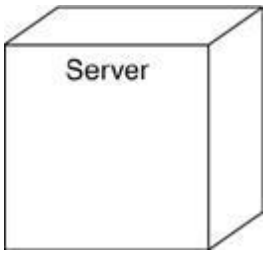
The remaining two elementsartifacts and nodesare also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

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 shown in the following figure.

Figure: Artifacts

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

Figure: Nodes



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

Behavioral Things

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

First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects or roles within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, actions, and connectors (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as shown in the following figure.

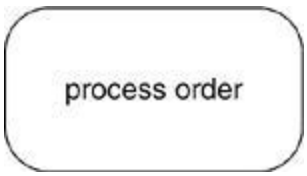
Figure: Messages

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

Figure: States

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

Figure: Actions



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

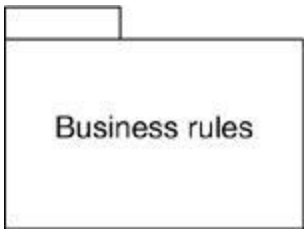
Grouping Things

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

A *package* is a general-purpose mechanism for organizing the design itself, as opposed to classes, which organize implementation constructs. Structural things, behavioral things, and even

other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents, as shown in the following figure.

Figure: Packages

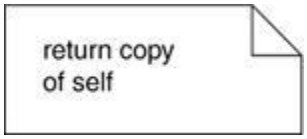


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as shown in the following figure.

Figure: Notes



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

Relationships in the UML

There are four kinds of relationships in the UML:

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

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

well-formed models.

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

Figure: Dependencies

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

Figure: Associations

Third, a *generalization* is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as shown in the following figure.

Figure: Generalizations

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

Figure: Realizations

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

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

Diagrams in the UML

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

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

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

These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system. Component diagrams are variants of class diagrams.

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

the perspective of real or prototypical cases.

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

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

Both sequence diagrams and communication diagrams are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages; a *communication diagram* is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages. Sequence diagrams and communication diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts. Sequence diagrams emphasize temporal ordering, and communication diagrams emphasize the data structure through which messages flow. A timing diagram (not covered in this book) shows the actual times at which messages are exchanged.

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

An *activity diagram* shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. A node typically hosts one or more artifacts.

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

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

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

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

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

Rules of the UML

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

The UML has syntactic and semantic rules for

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

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

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

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

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

Common Mechanisms in the UML

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

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

Specifications

The UML 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.

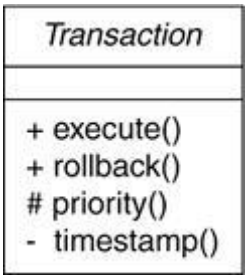
Adornments

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, the following figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

Figure: Adornments



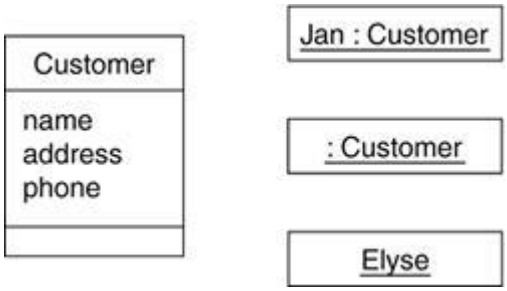
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

Common Divisions

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

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

Figure: Classes and Objects



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

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

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

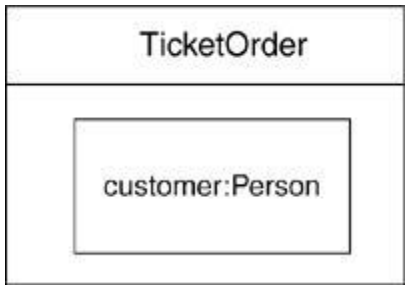
Figure: Interfaces and Implementations

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

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

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

Figure: Part with role and type



Extensibility Mechanisms

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

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

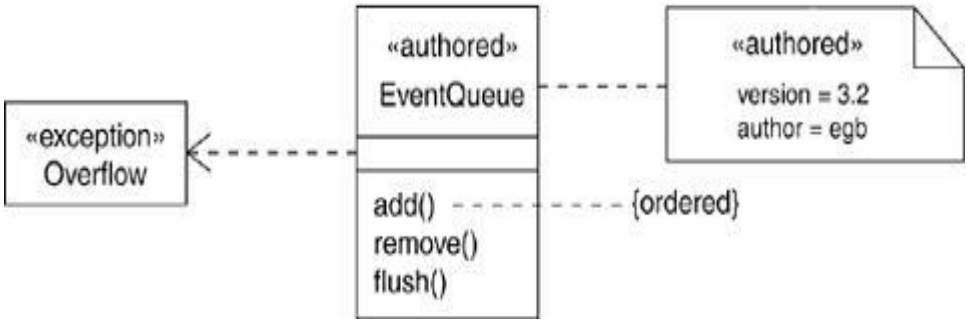
- Stereotypes
- Tagged values
- Constraints

A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your modelsmeaning that they are treated like basic building blocksby marking them with an appropriate stereotype, as for the class Overflow in Figure 2-19.

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

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

Figure: Extensibility Mechanisms



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

ARCHITECTURE

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

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

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

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

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

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

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

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

Software Development Life Cycle:

The Unified Software Development Process

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

Basic properties:

• use case driven

• architecture centric

• iterative and incremental

Use case Driven

Use cases

• capture requirements of the user,

• divide the development project into smaller subprojects,

• are constantly refined during the whole development process

• are used to verify the correctness of the implemented software

Architecture Centric:

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

• understandable,

• maintainable,

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

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

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

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

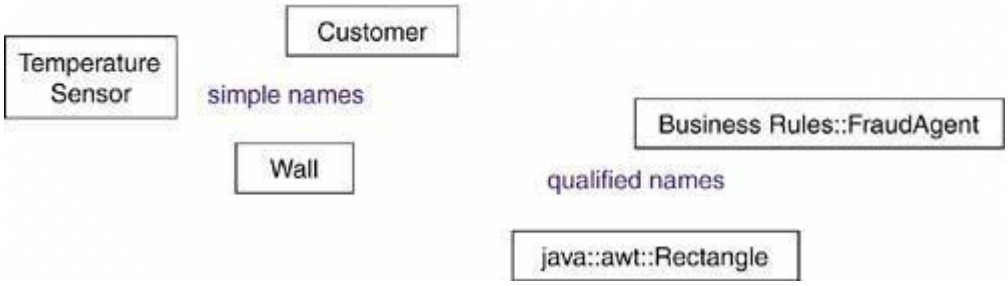
Transition contains beta testing, performance tuning and user training.

CLASSES

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

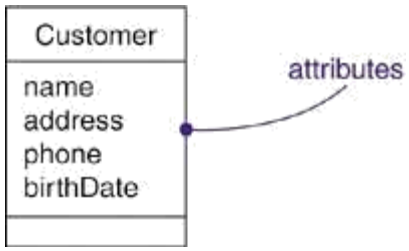
Names

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a simple name; a qualified name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name



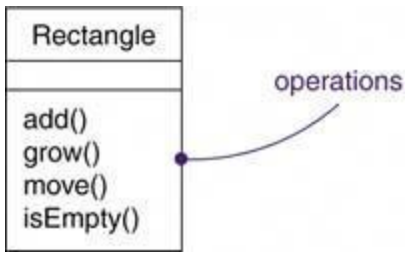
Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth.

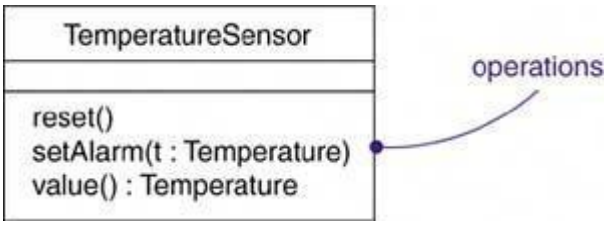


Operations

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

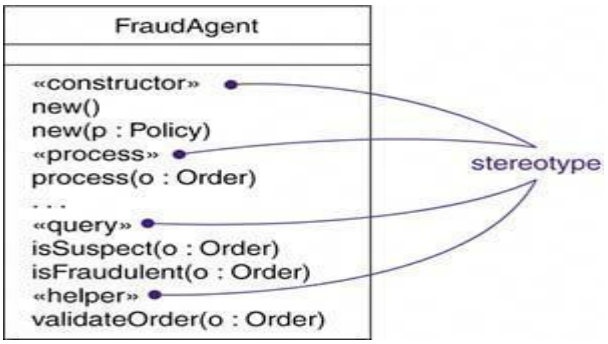


You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type.



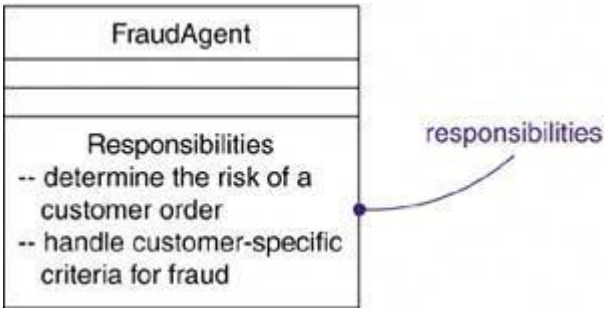
Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. You can indicate that there are more attributes or properties than shown by ending each list with an ellipsis ("...").



Responsibilities

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



Common Modeling Techniques

Modeling the Vocabulary of a System

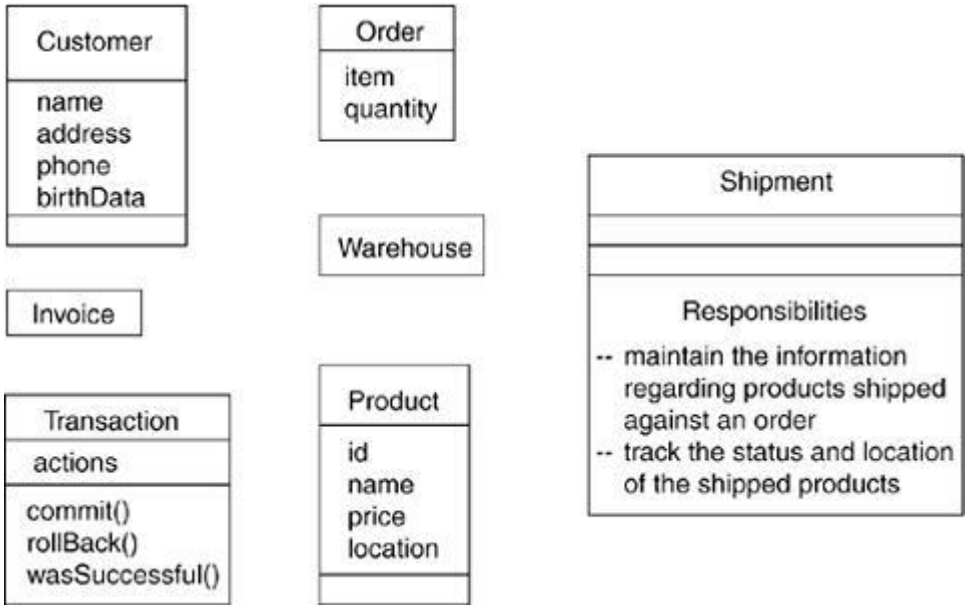
You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Each of these abstractions is a part of the vocabulary of your system, meaning that, together, they represent the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

To model the vocabulary of a system,

Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.

For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes. Provide the attributes and operations that are needed to carry out these responsibilities for each class.

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



Modeling the Distribution of Responsibilities in a System

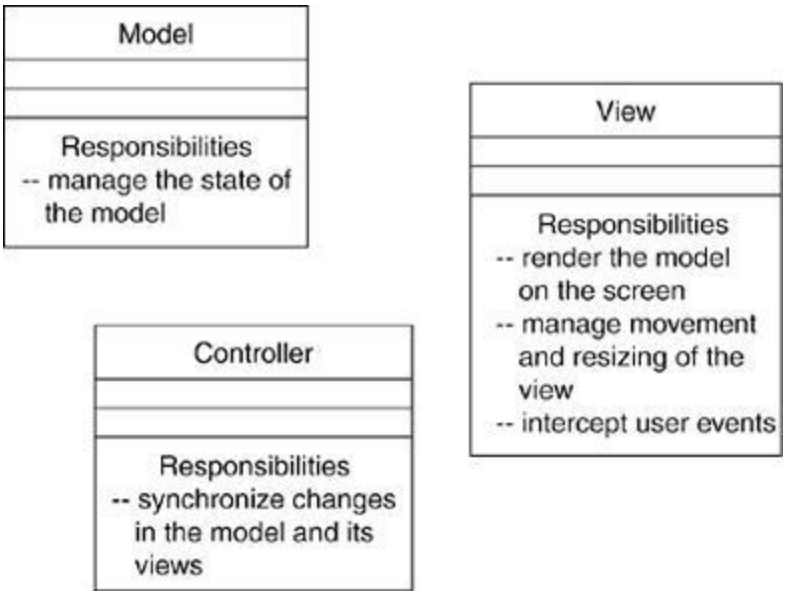
Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.

To model the distribution of responsibilities in a system,

Identify a set of classes that work together closely to carry out some behavior. Identify a set of responsibilities for each of these classes.

Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.

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



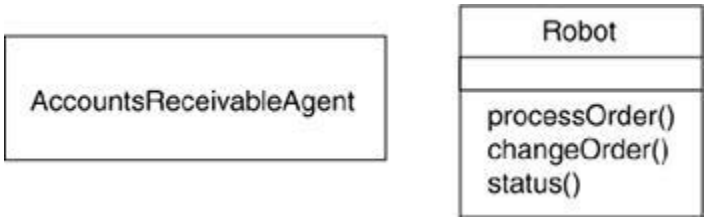
Modeling Nonsoftware Things

To model nonsoftware things,

Model the thing you are abstracting as a class.

If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.

If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node as well, so that you can further expand on its structure.

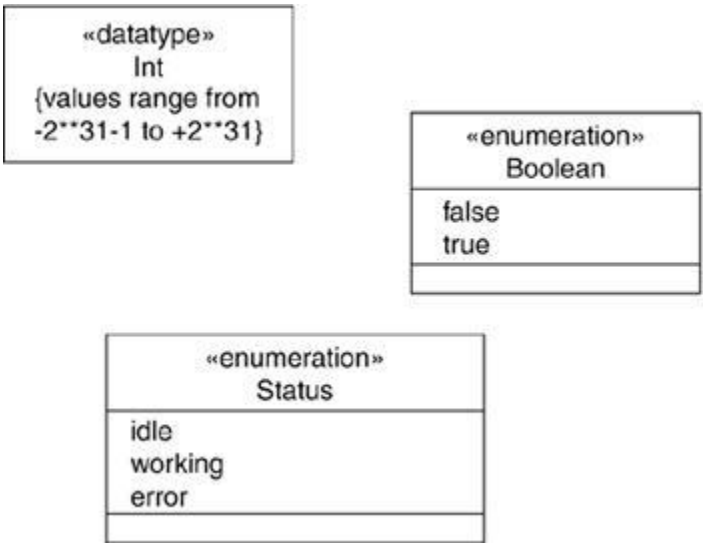


Modeling Primitive Types

To model primitive types,

Model the thing you are abstracting as a class or an enumeration, which is rendered using class notation with the appropriate stereotype.

If you need to specify the range of values associated with this type, use constraints.

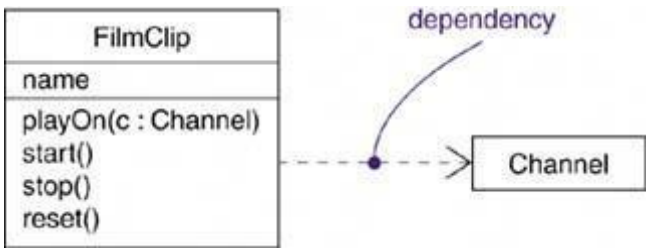


RELATIONSHIPS

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

Dependencies

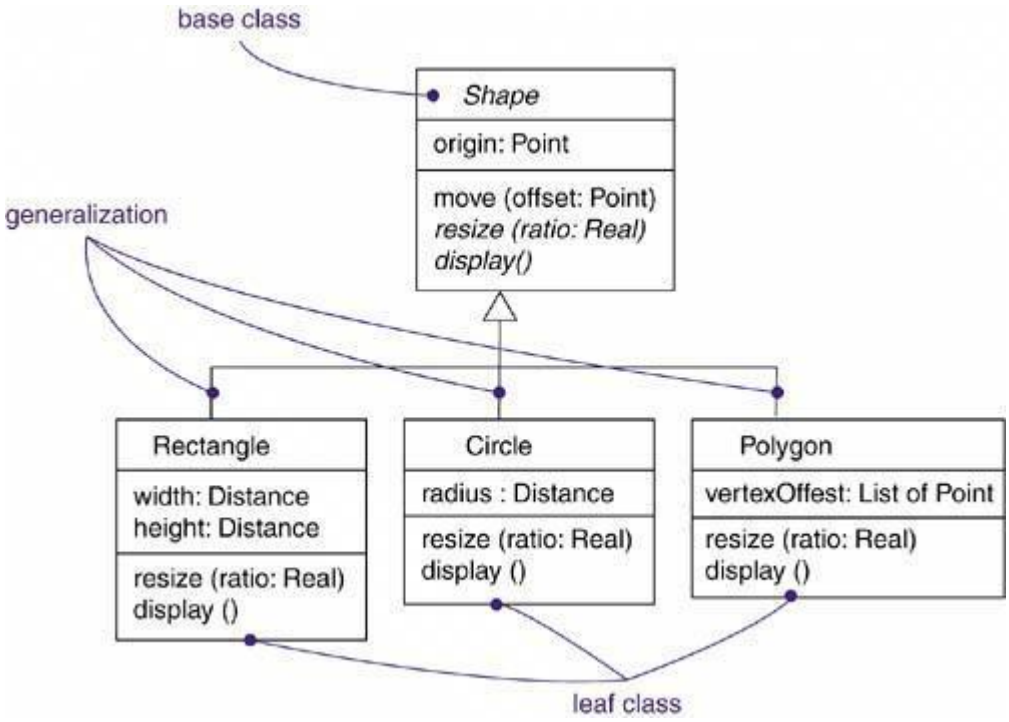
A *dependency* is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended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



Generalizations

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

or parent) and a more specific kind of thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). An objects of the child class may be used for a variable or parameter typed by the parent, but not the reverse



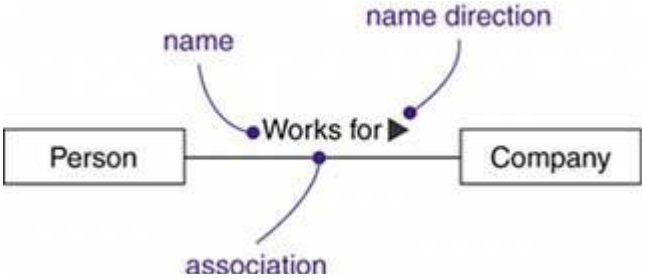
Associations

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can relate objects of one class to objects of the other class. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called *n-ary associations*.

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

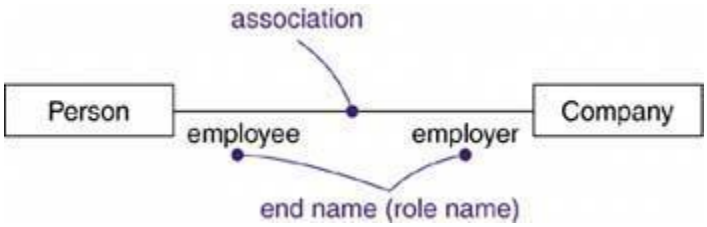
Name

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



Role

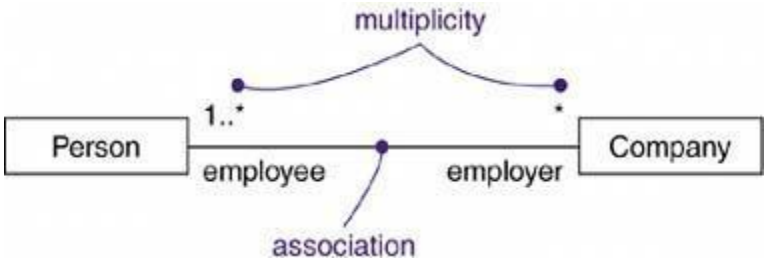
When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association. You can explicitly name the role a class plays in an association. The role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



Multiplicity

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

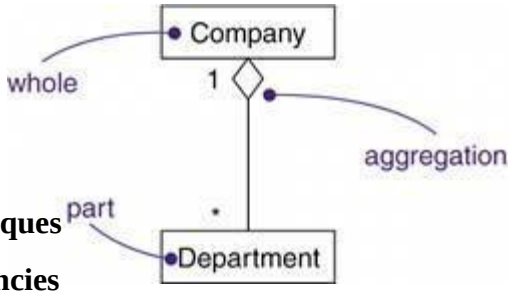
The number of objects must be in the given range. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can give an integer range (such as 2..5). You can even state an exact number (for example, 3, which is equivalent to 3..3).



Aggregation

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than

the other. Sometimes you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship



Common Modeling Techniques

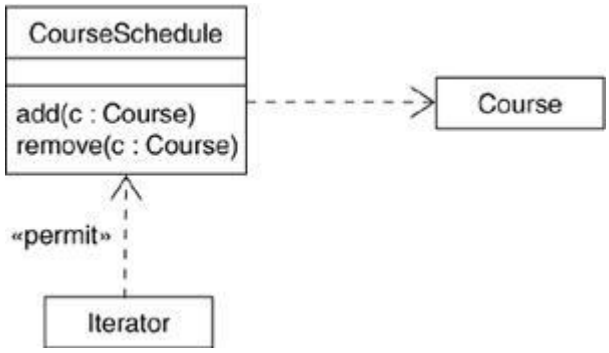
Modeling Simple Dependencies

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

To model this using relationship,

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

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



Modeling Single Inheritance

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

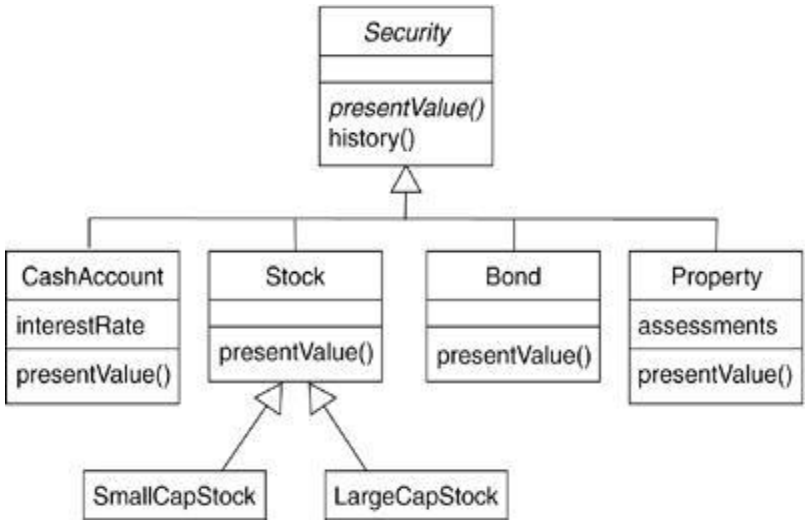
To model inheritance relationships,

Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.

Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about

introducing too many levels).

Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



Modeling Structural Relationships

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

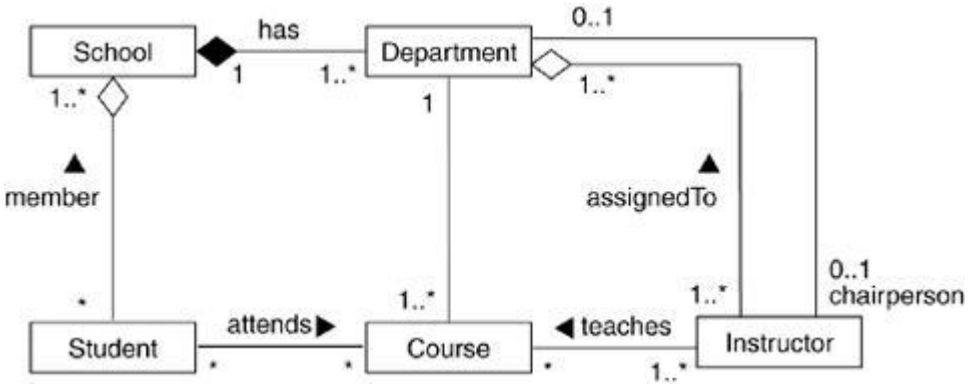
To model structural relationships,

For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.

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

For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if they help to explain the model).

If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole with a diamond.



COMMON MECHANISMS

A *note* is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

A *stereotype* is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem. Graphically, a stereotype is rendered as a name enclosed by guillemets (French quotation marks of the form « »), placed above the name of another element.

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

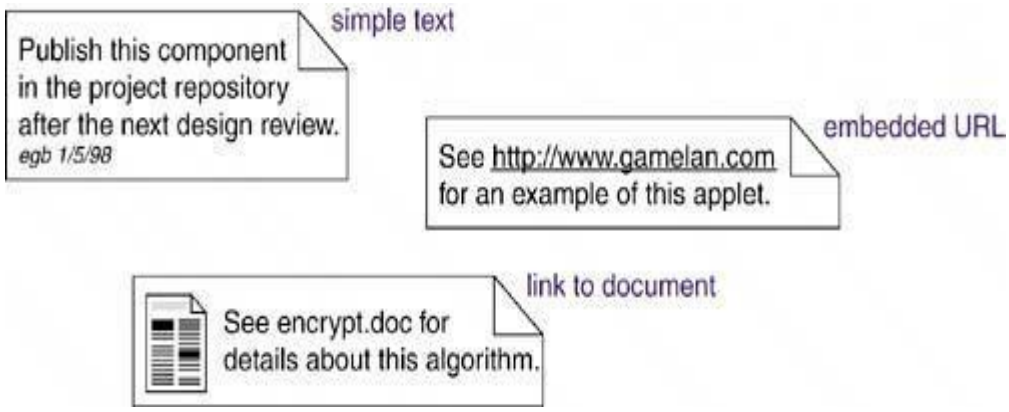
A *tagged value* is a property of a stereotype, allowing you to create new information in an element bearing that stereotype. Graphically, a tagged value is rendered as a string of the form name = value within a note attached to the object.

A *constraint* is a textual specification of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. As an alternative, you can render a constraint in a note.

Notes

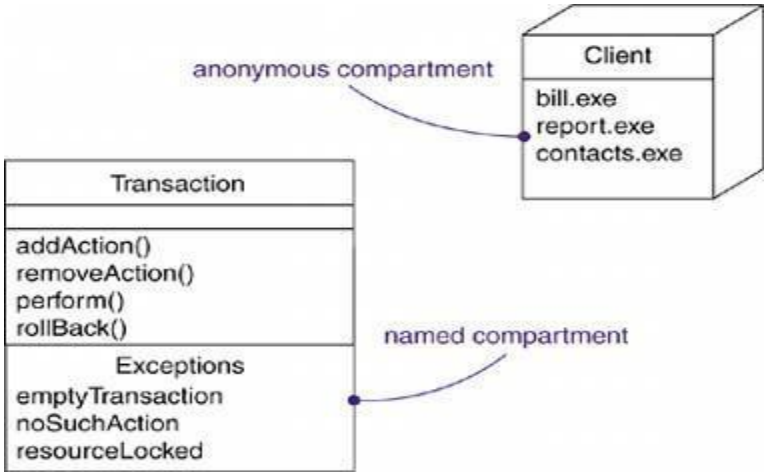
A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

A note may contain any combination of text or graphics



Other Adornments

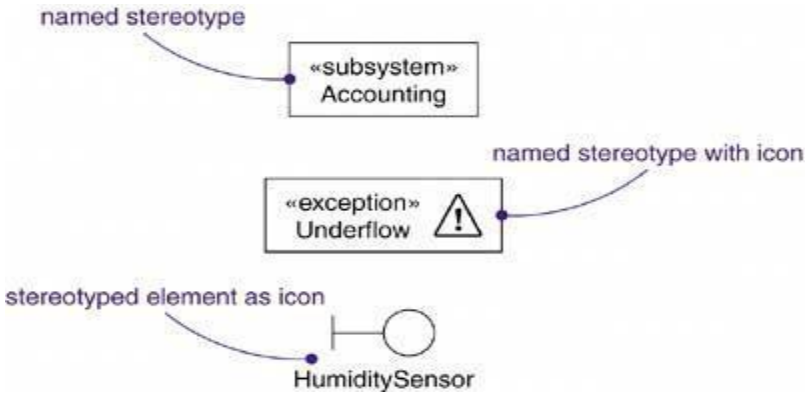
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification.



Stereotypes

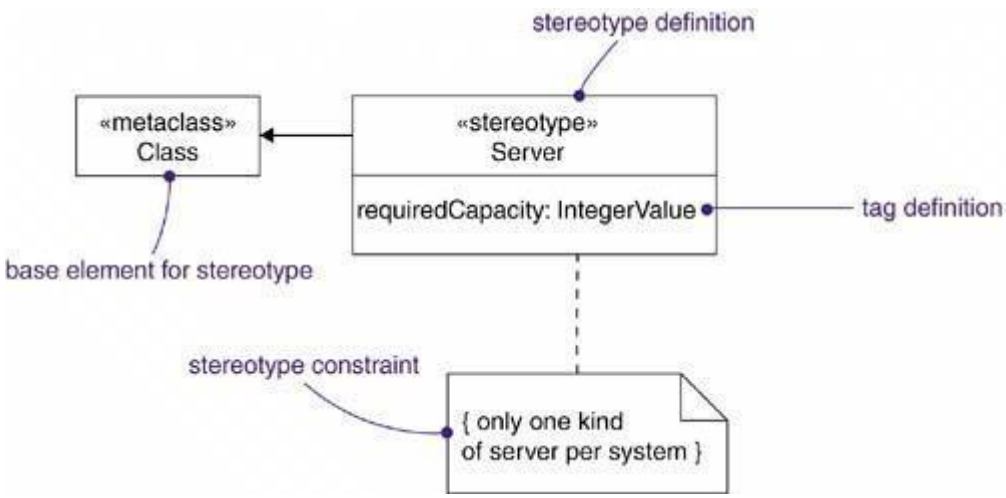
The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model.

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



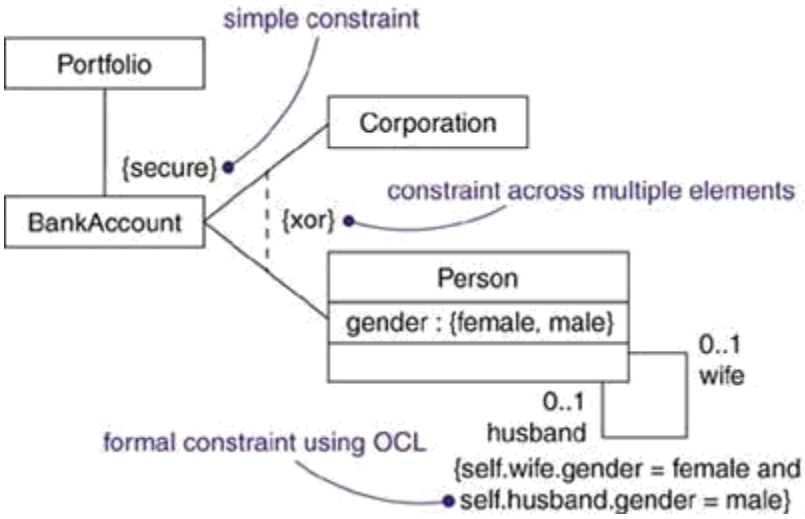
Tagged Values

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



Constraints

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



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

Common Modeling Techniques

Modeling Comments

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

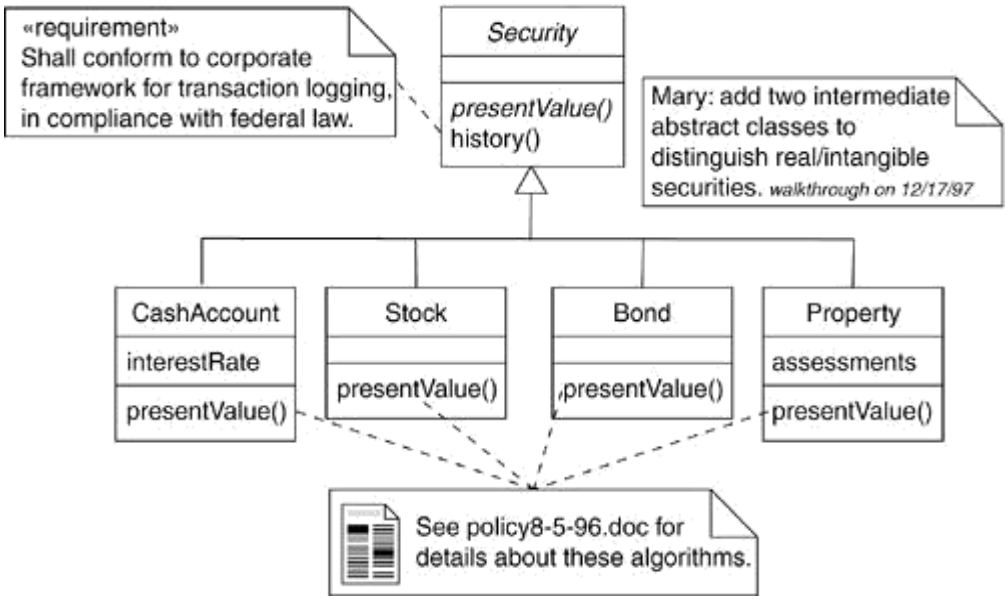
To model a comment,

Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.

Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.

If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.

As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and unless they are of historic interest discard the others.



Modeling New Properties

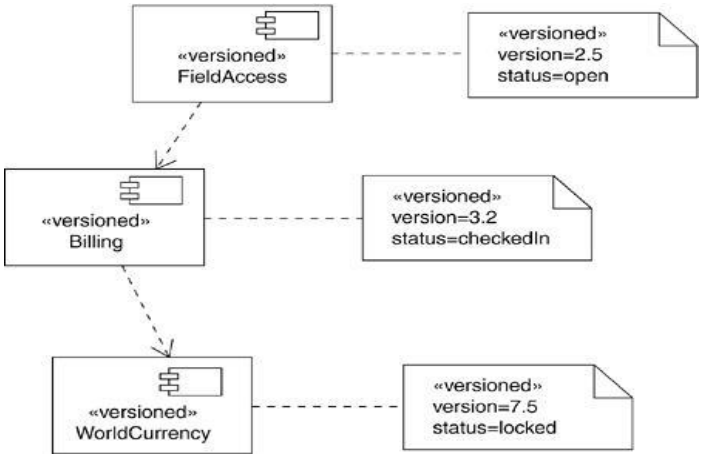
The basic properties of the UML's building blocksattributes and operations for classes, the contents of packages

To model new properties,

First, make sure there's not already a way to express what you want by using basic UML.

If you re convinced there's no other way to express these semantics, define a stereotype and add the new properties to the stereotype. The rules of generalization applytagged values defined for one kind of stereotype apply to its children.

the model itself, andunless they are of historic interestdiscard the others.



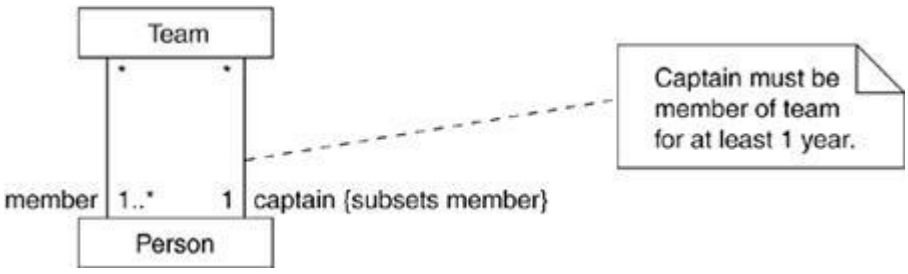
Modeling New Semantics

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

To model new semantics,

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

If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



CLASS DIAGRAMS

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 diagrams a name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

Contents

Class diagrams commonly contain the following things:

Classes

Interfaces

Dependency, generalization, and association relationships

Common Uses

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

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

1. To model the vocabulary of a system

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

2. To model simple collaborations

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

3. To model a logical database schema

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

Common Modeling Techniques

Modeling Simple Collaborations

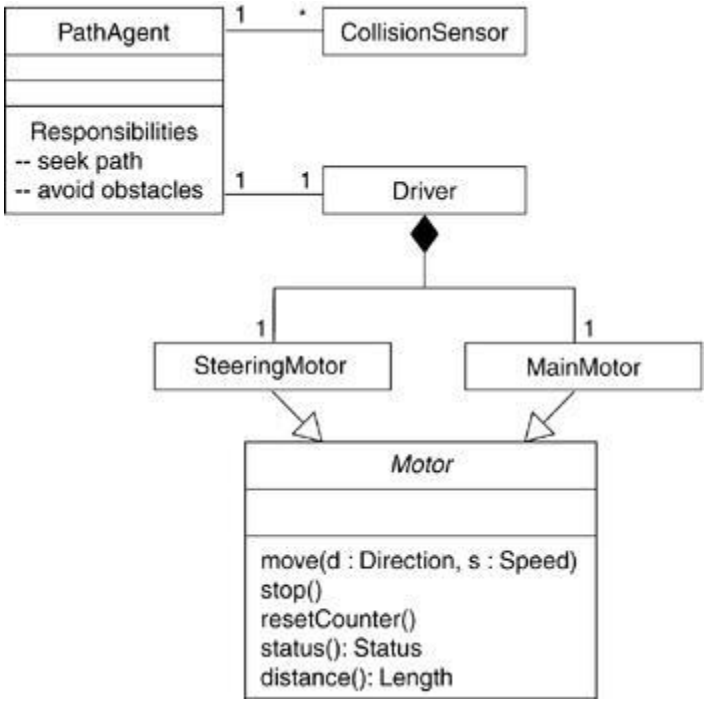
To model a collaboration,

Identify the mechanism you'd like to model. A mechanism represents some function or

behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things as well. Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.

Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.



Modeling a Logical Database Schema

To model a schema,

Identify those classes in your model whose state must transcend the lifetime of their applications.

Create a class diagram that contains these classes. You can define your own set of stereotypes and tagged values to address database-specific details.

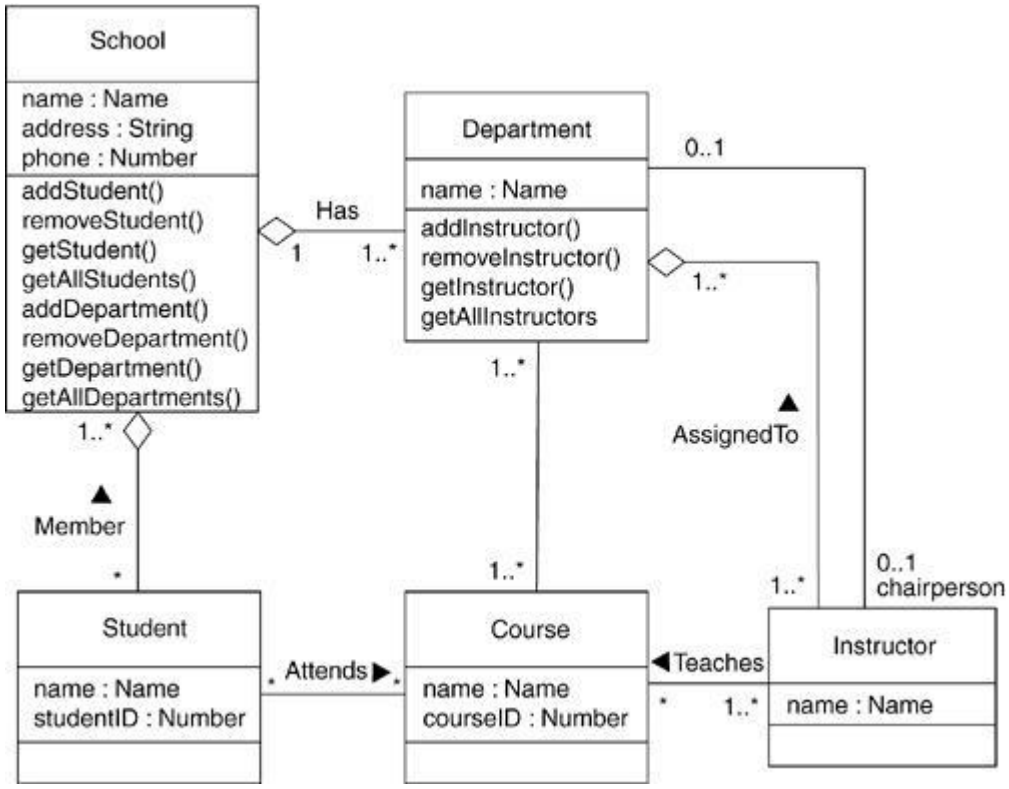
Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their multiplicities that relate these classes.

Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to

simplify your logical structure.

Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.

Where possible, use tools to help you transform your logical design into a physical design.



Forward and Reverse Engineering

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

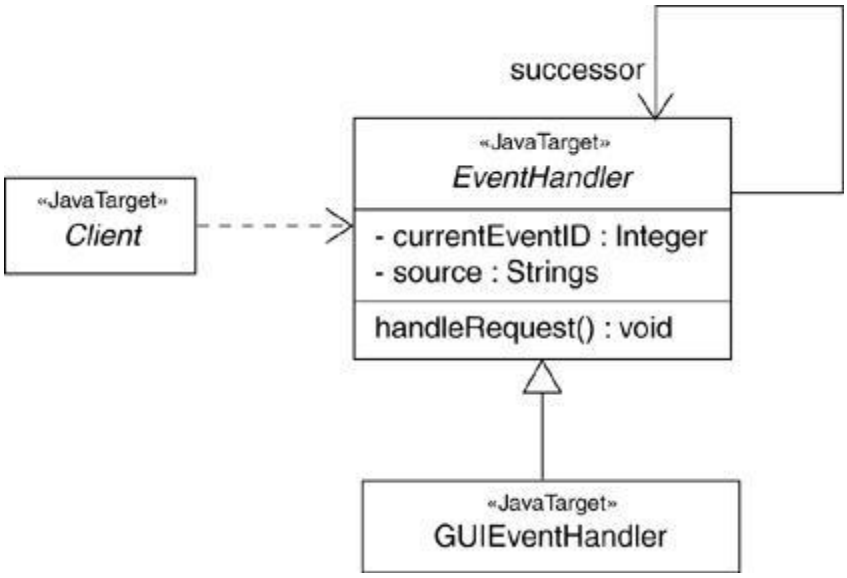
To forward engineer a class diagram,

Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole. Depending on the semantics of the languages you choose, you may want to constrain your use of certain

UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent), or you can develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).

Use tagged values to guide implementation choices in your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.

Use tools to generate code.



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

```
public abstract class EventHandler {
EventHandler successor; private Integer currentEventID; private String source;
EventHandler() {}
public void handleRequest() {}
}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering

models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. You need to select portion of the code and build the model from the bottom.

Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighbouring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

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.

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 diagrams that 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.

Common Uses

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

When you model the static design view or static process view of a system, you typically use object diagrams in one way:

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.

Common Modeling Techniques

Modeling Object Structures

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 other things.

• Create a collaboration to describe a mechanism.

• 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.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value. In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you cannot exactly instantiate these objects from the outside.

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,

•Chose 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.

IMPORTANT QUESTIONS

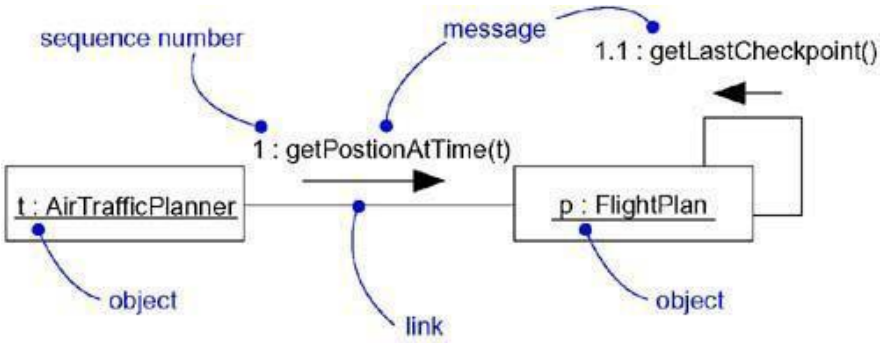
1. Discuss about conceptual model in UML and building blocks of UML.
2. What are the common mechanisms of UML and explain in detailed.
3. Define UML and explain how the architecture of UML meets the requirements of modeling?
4. Explain in detail about the extensibility mechanisms in UML.
5. Explain the various relationships with UML notation.
6. Enumerate the steps to model logical database schema. Give all example class diagrams.
7. Enumerate the steps to model different views of a system.

UNIT – IV

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

INTERACTIONS

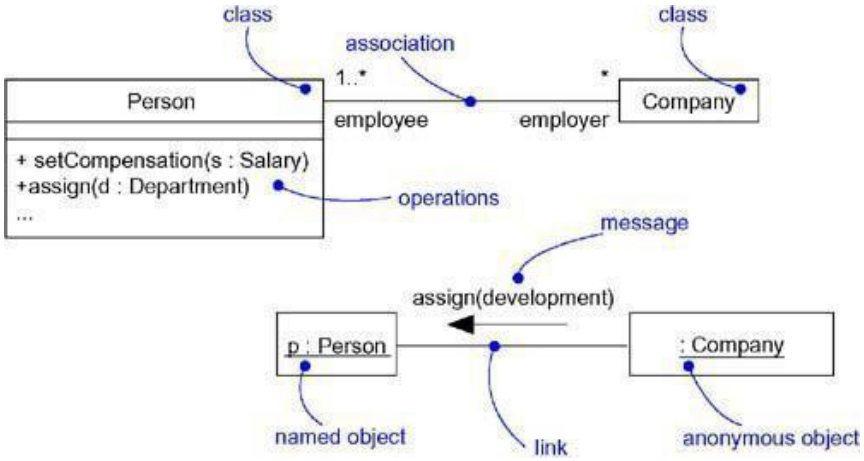
- An interaction is a behavior that is composed of a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- A message specifies the communication between objects for an activity to happen. It has following parts: its name, parameters (if any), and sequence number.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



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

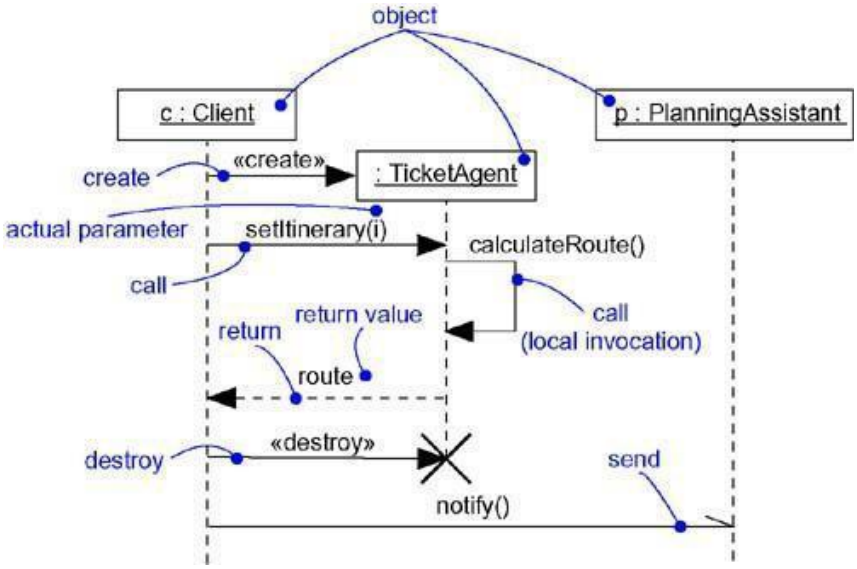
<<association>> – Specifies that the corresponding object is visible by association

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



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

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



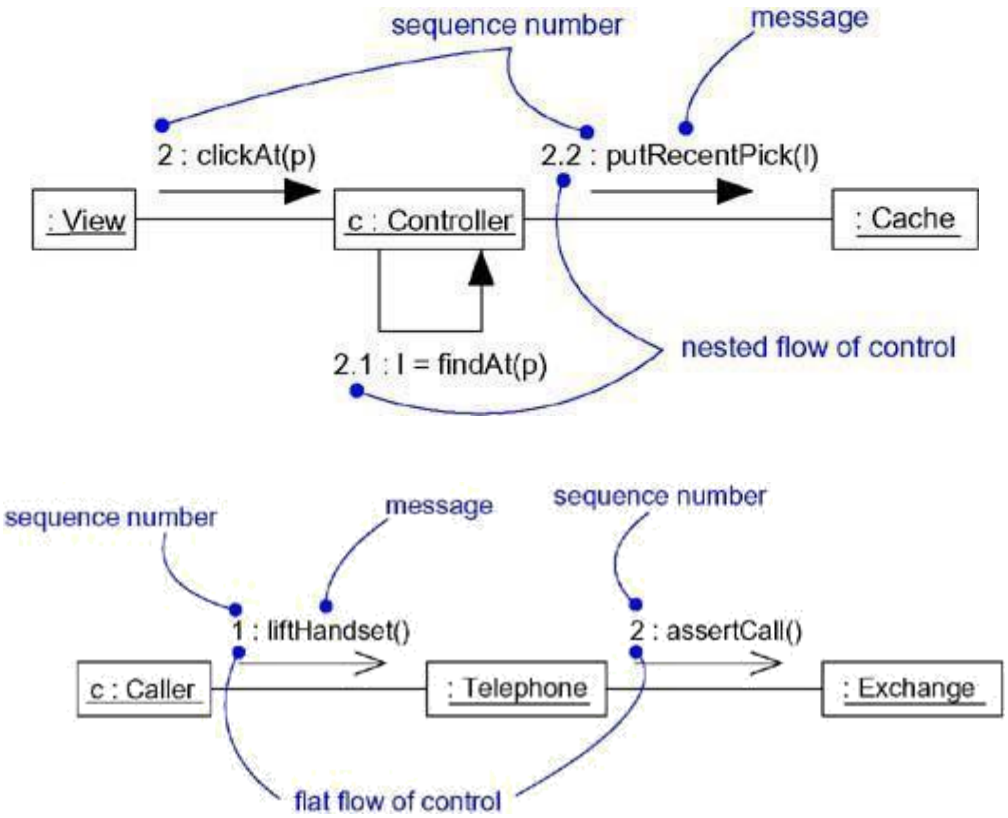
Sequencing

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

Flow of control (2 types)

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

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



Creation, Modification, and Destruction of links

Enabled by *adding the following constraints to the element*

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

Representation of interactions

interaction goes together with objects and messages.

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

Common Modeling Techniques

Modeling a Flow of Control

To model a flow of control,

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

individual operation

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

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

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

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

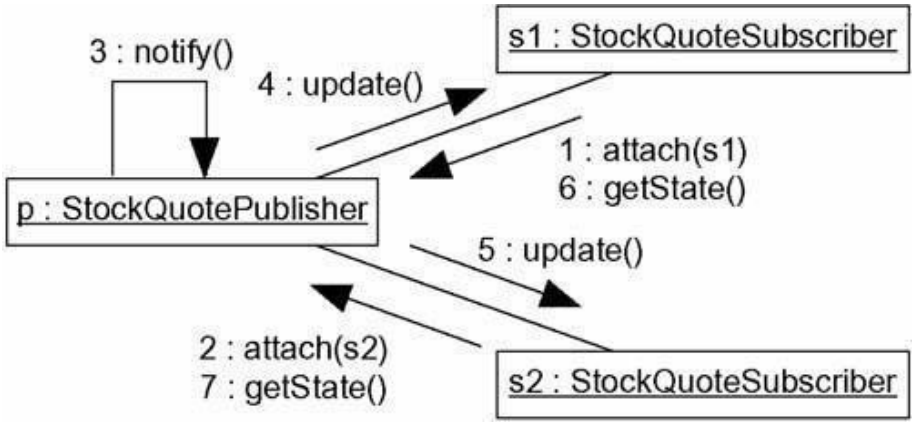


Figure: Flow of Control by time

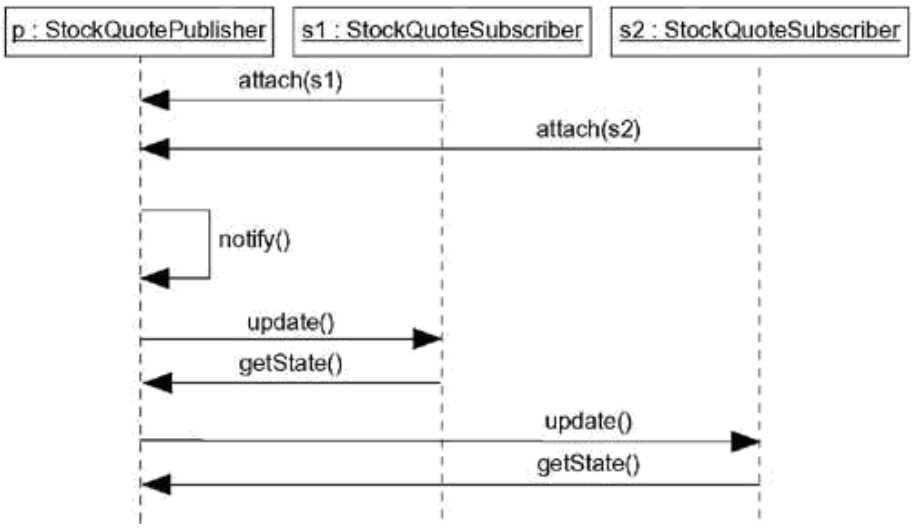


Figure: Flow of Control by Organization

INTERACTION DIAGRAMS

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

 Interaction diagrams commonly contain Objects, Links, Messages

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

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

 It may contain notes and constraints

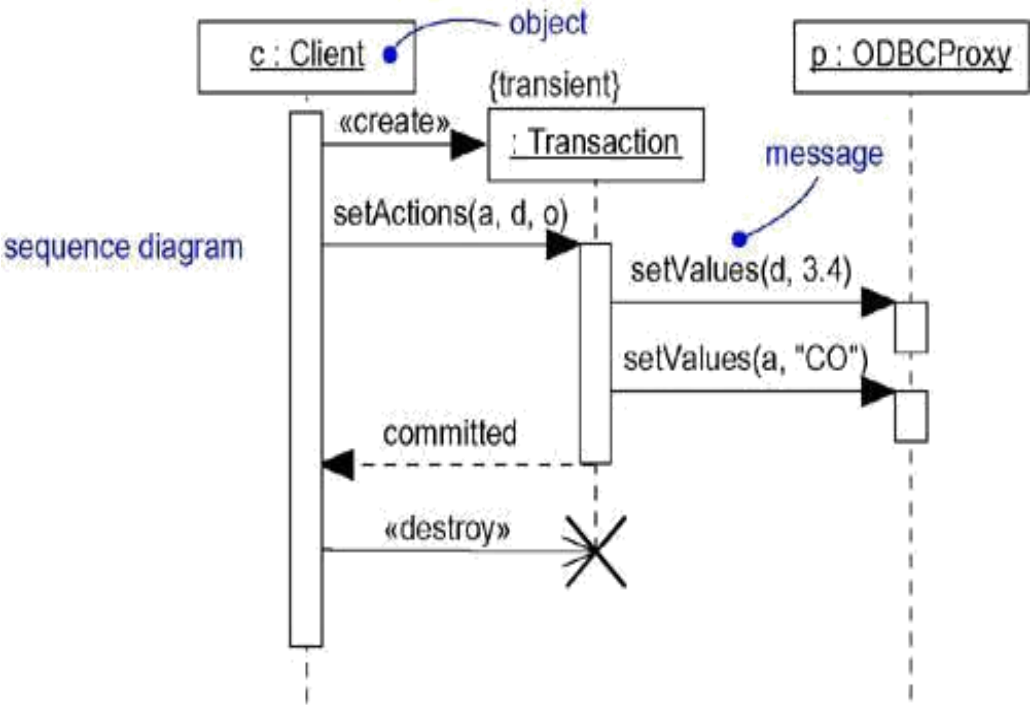
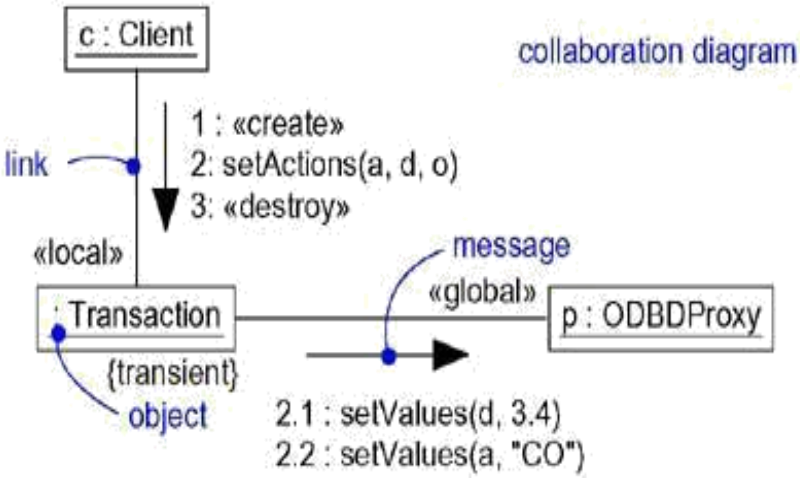


Fig: Interaction Diagrams



Sequence Diagrams

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

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

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

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

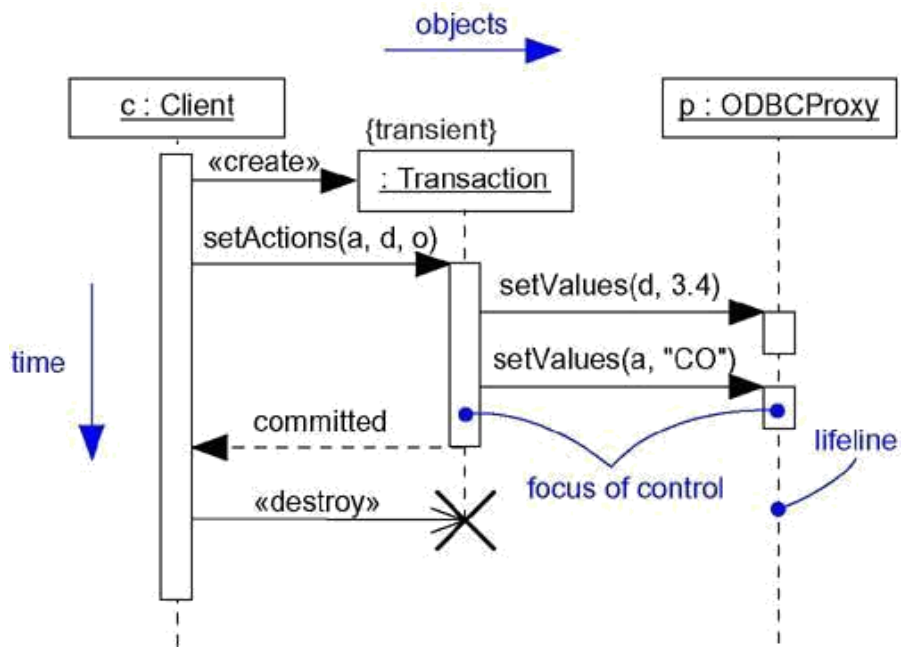


Figure: Sequence Diagram

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

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

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

Collaboration Diagrams

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

 Graphically it is a collection of vertices and arcs

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

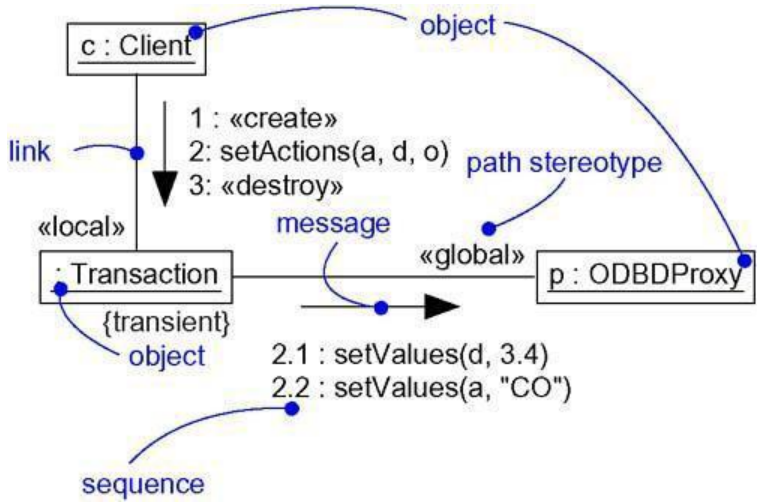


Figure: Collaboration Diagram

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

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

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

Semantic Equivalence

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

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

To model a flow of control by time ordering,

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

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

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

 Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties .

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

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

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

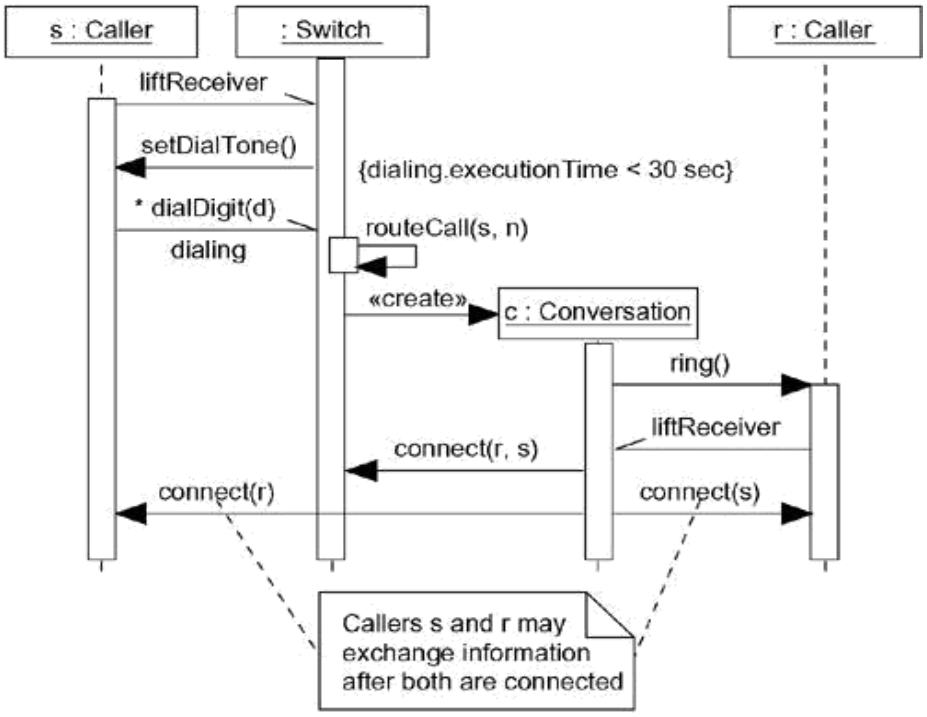


Figure: Modeling Flows of Control by Time Ordering

Modeling Flows of Control by Organization

To model a flow of control by organization

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

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

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

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

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

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

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

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

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

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

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

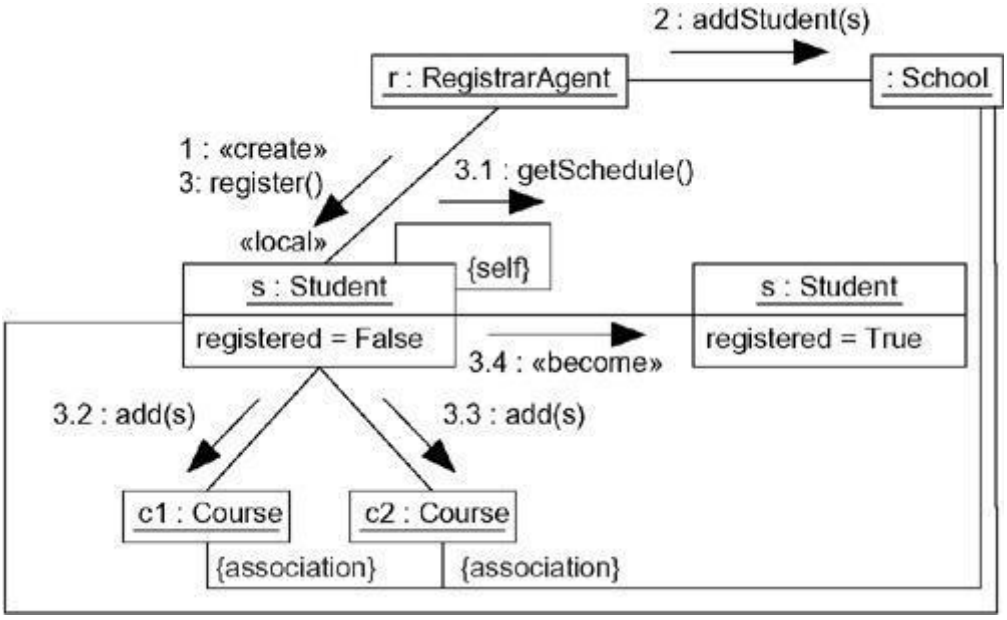


Figure: Modeling Flows of Control by Organization

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

So the purposes of interaction diagram can be describes as:

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

USE CASES

 use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor

 use case captures the intended behavior of the system (or subsystem, class, or interface) without having to specify how that behavior is implemented

 a use case is represented as an ellipse

 Every use case must have a name that distinguishes it from other use cases: simple name and path name Figure 2: Simple and Path Names

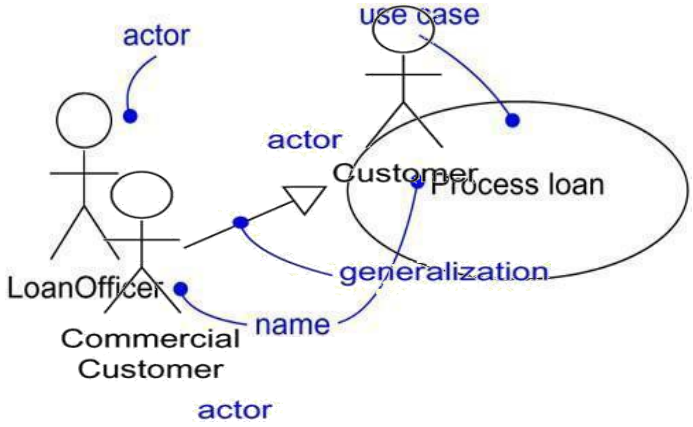


Figure 1: Actors and Use Cases

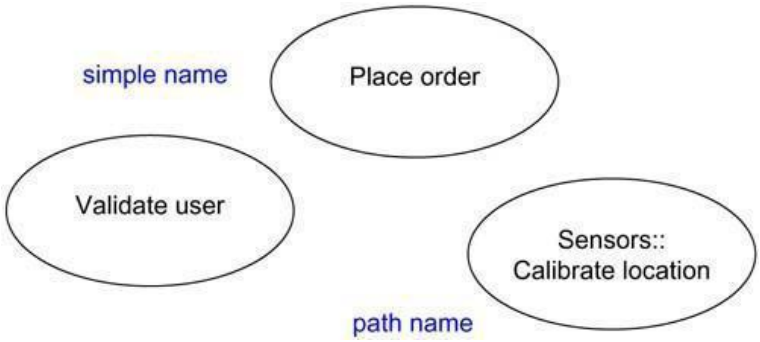


Figure 2: Simple and Path Names

Actors

 actor represents a coherent set of roles that users of use cases play when interacting with these use cases

 an actor represents a role that a human, a hardware device, or even another system plays with a system Figure 3: Actors

Actors may be connected to use cases by association

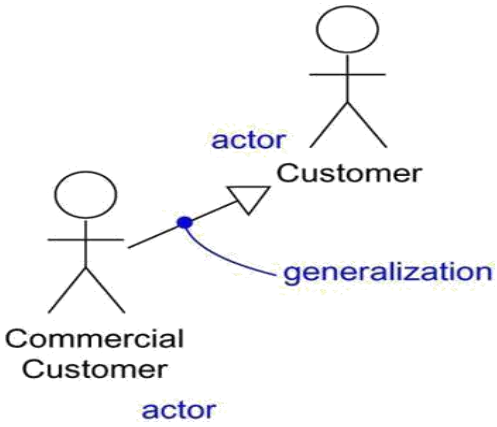


Figure 3: Actors

Use Cases & Flow of Events

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

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

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

Use Cases and Scenarios

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

Use Cases and Collaborations

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

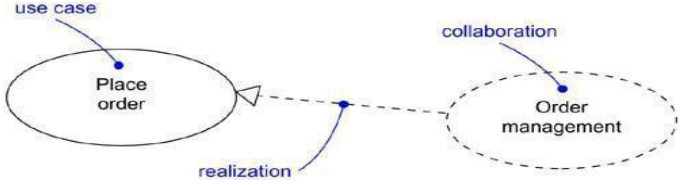


Figure 4: Use Cases and Collaborations

Organizing Use Cases

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

Generalization, Include and Extend

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

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

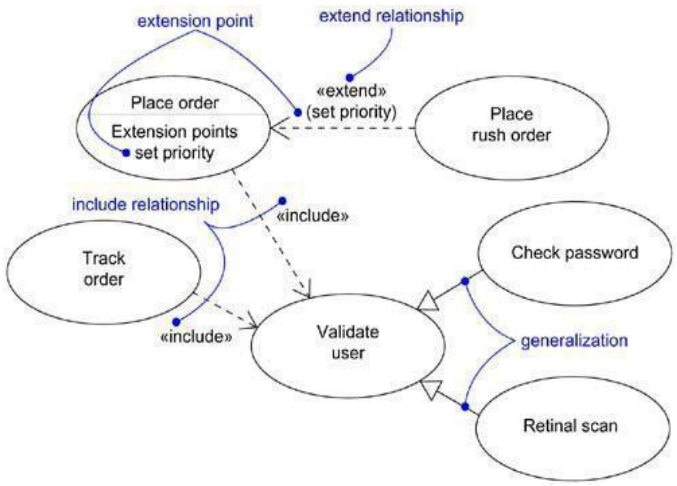


Figure 5: Generalization, Include and Extend

Modeling the Behavior of an Element

To model the behavior of an element,

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

 Organize these behaviors as use cases, applying include and extend

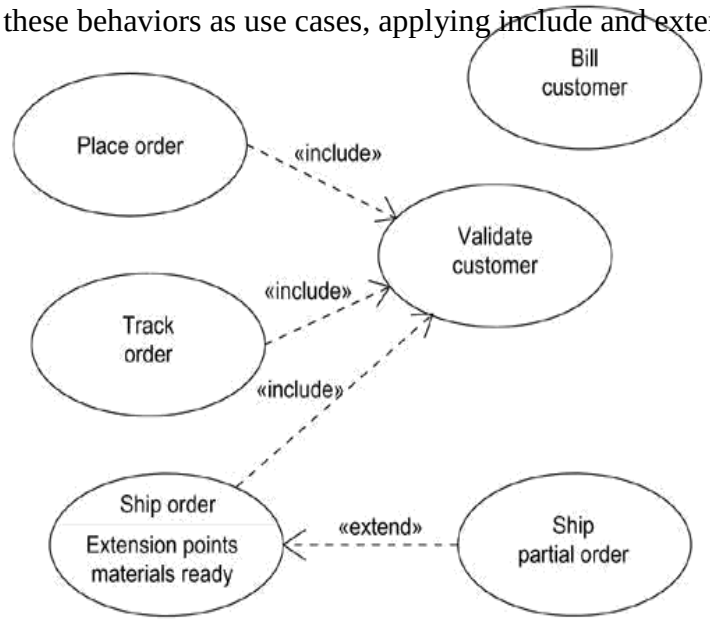


Figure 6: Modeling the Behavior of an Element

USE CASE DIAGRAMS

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

 Use case diagrams commonly contain Use cases, Actors, Dependency, generalization, and association relationships

 use case diagrams may contain packages, certain times instances of use cases, notes and constraints

 apply use case diagrams to model the static use case view of a system by modeling the context of a system and by modeling the requirements of a system

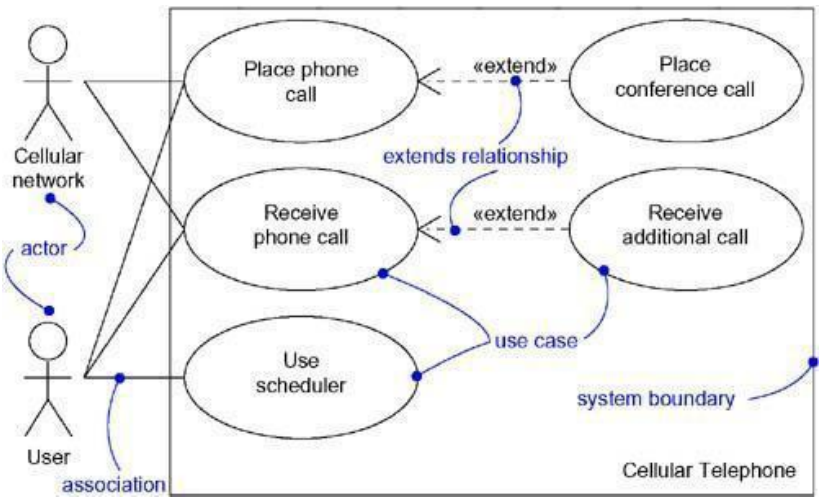


Figure 1: A Use Case Diagram

Modeling the Context of a System

To model the context of a system,

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

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

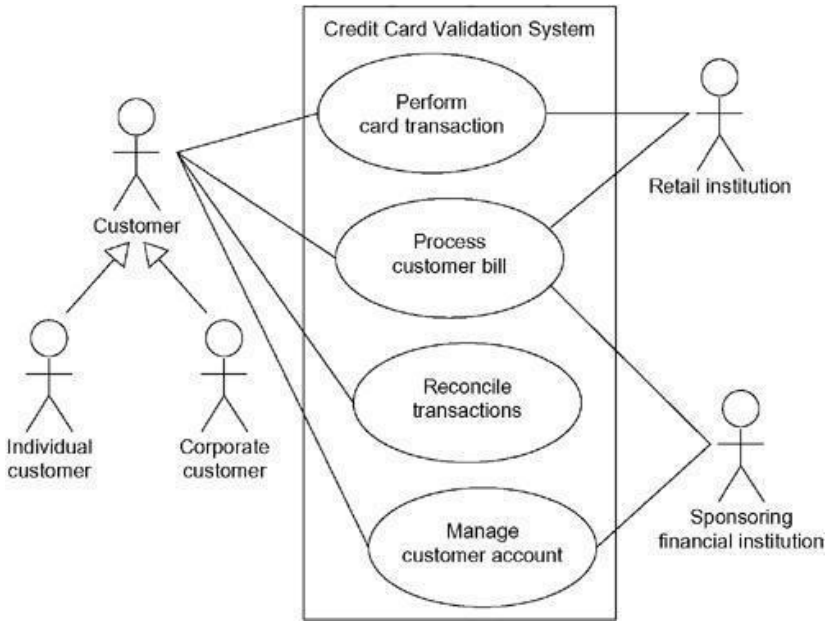


Figure 2: Modeling the Context of a System

Common Modeling Techniques

Modeling the Requirements of a System

To model the requirements of a system,

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

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

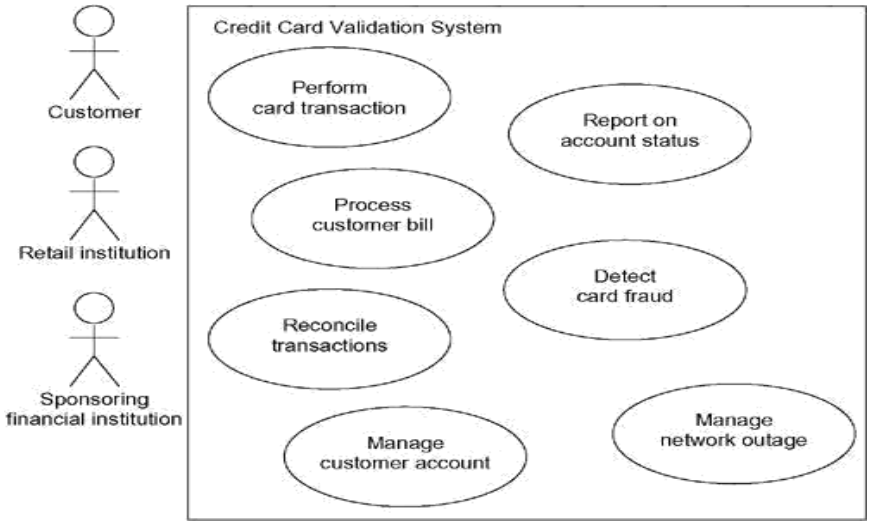


Figure 3: Modeling the Requirements of a System

ACTIVITY DIAGRAMS

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

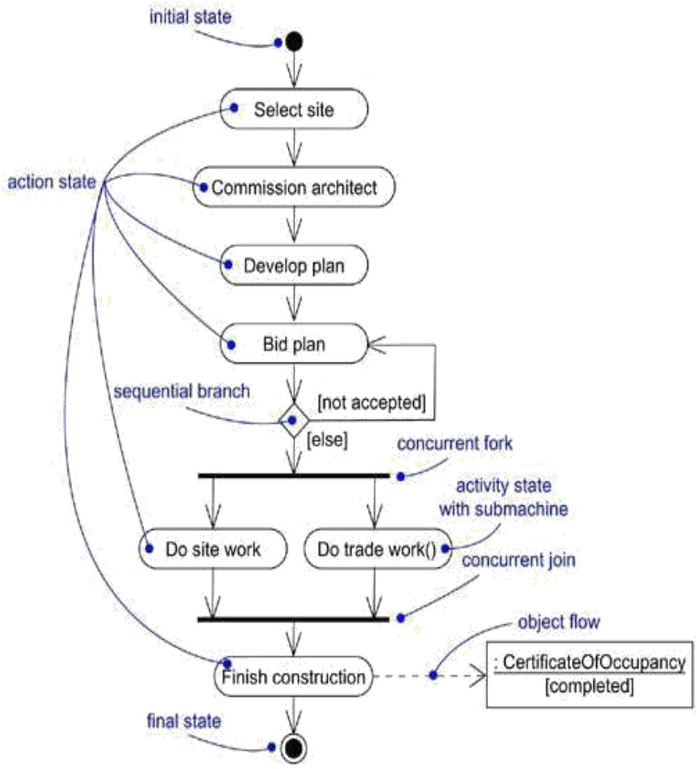


Figure 1: activity diagram

Action States

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

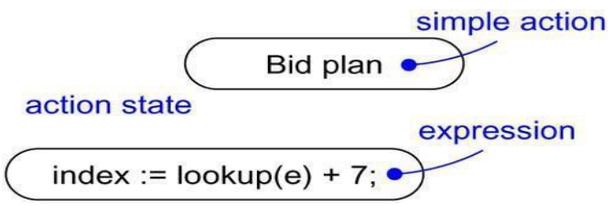


Figure 2: Action States

Activity States

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

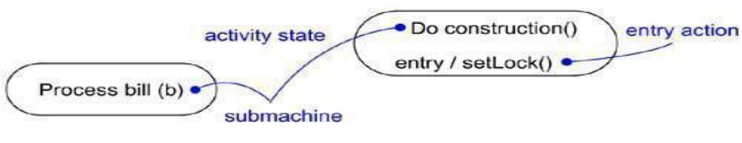


Figure 3: Activity States

Transitions

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

Triggerless Transitions

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

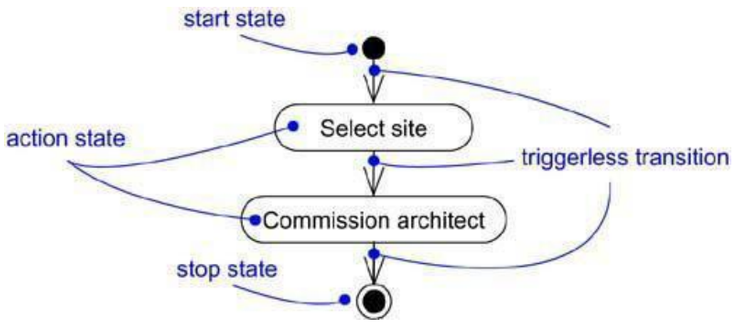


Figure 4: Triggerless Transitions

Branching

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

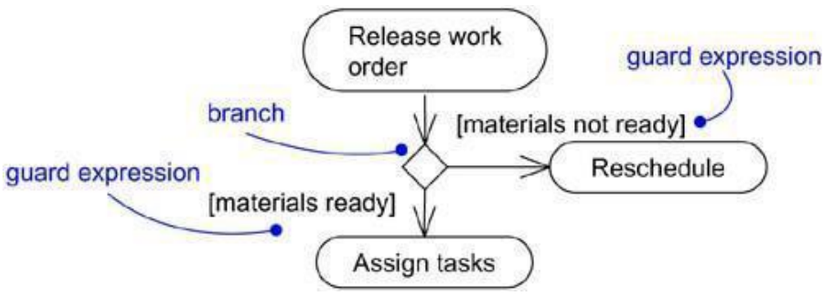


Figure 5: Branching

Forking and Joining

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

bar

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

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

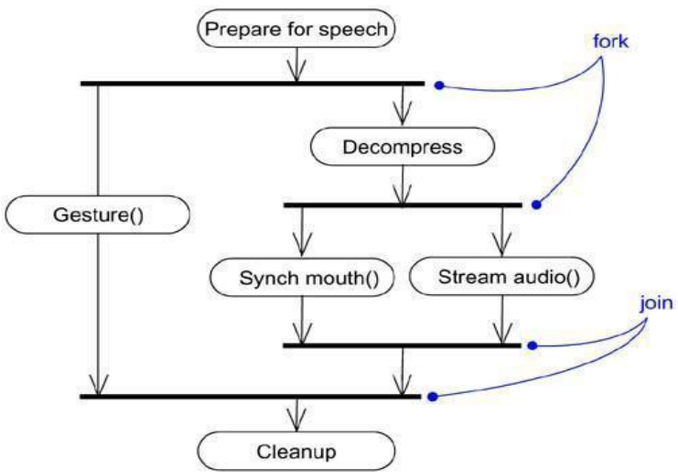


Figure 6: Forking and Joining

Swimlanes

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

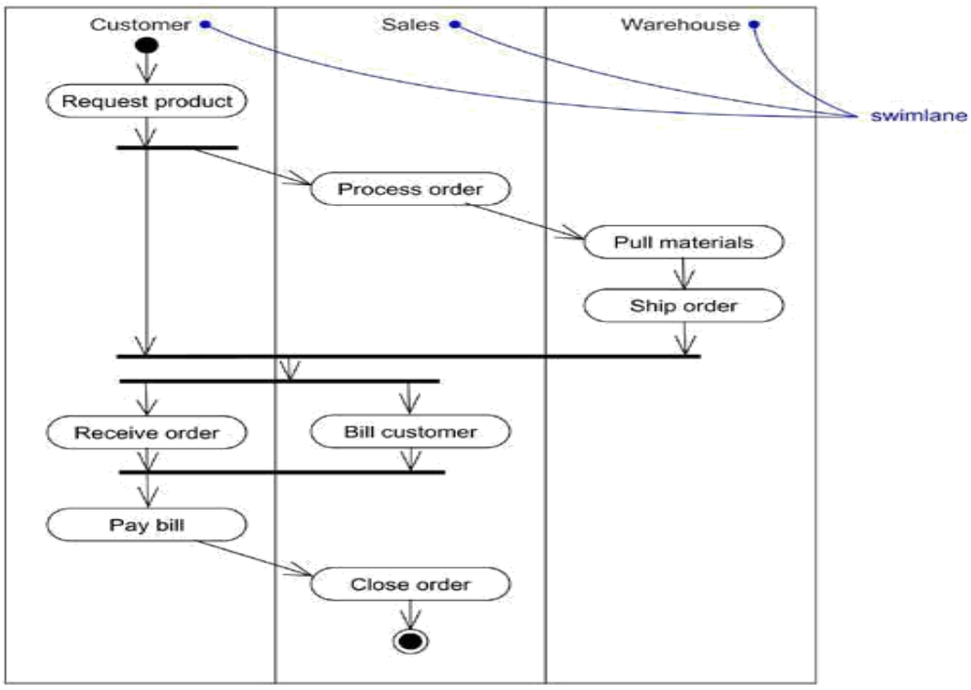


Figure 7: Swimlanes

Object Flow

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

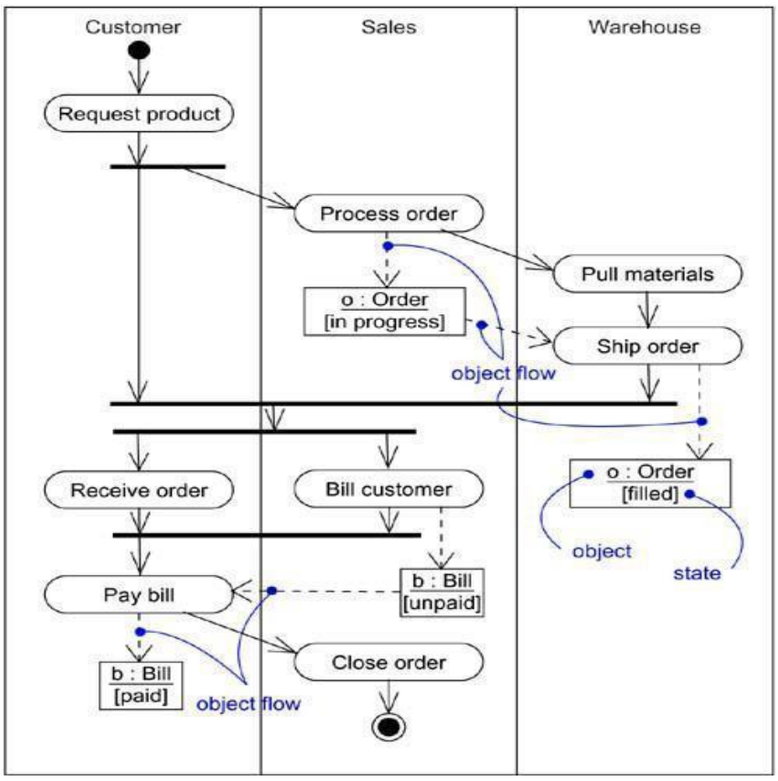


Figure 8: Object Flow

Common Modeling Techniques

Modeling a Workflow

To model a workflow,

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

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

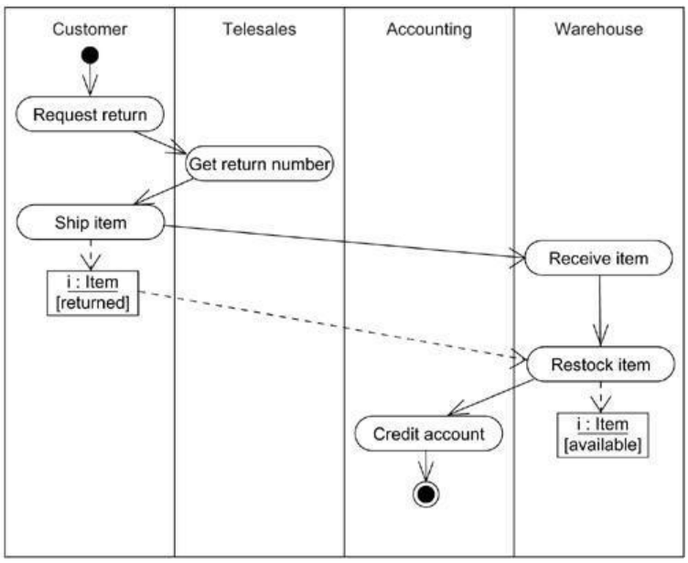


Figure: Modeling a Workflow

Modeling an Operation

To model an operation,

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

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

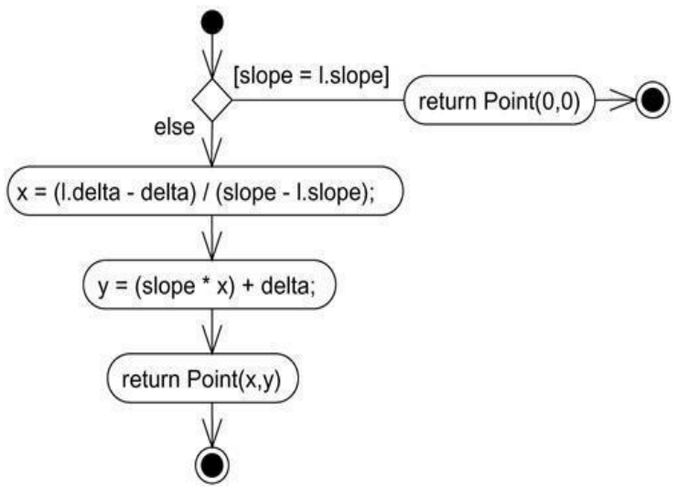


Figure 10: Modeling an Operation

IMPORTANT QUESTIONS

1. Explain the differences between collaboration and sequence diagram with example.
2. Draw the use case diagram for online reservation system with example.
3. Explain the various relationships possible among use cases. Illustrate in UML notation.
4. What are swim lanes? Explain with an activity diagram.
5. What are the various parts of a state? Explain briefly?
6. Describe the various parts of a transition.
7. Consider modeling a student information system. Consider the use case “student registers for a course". Draw a sequence diagram and explain briefly.
8. Draw a use case diagram that depicts the context of a credit card validation system. Explain briefly.

UNIT – V

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

EVENTS AND SIGNALS

Events

 An event is the specification of a significant occurrence that has a location in time and space.

 Anything that happens is modeled as an event in UML.

 In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition

 four kinds of events – signals, calls, the passing of time, and a change in state.

 Events may be external or internal and asynchronous or synchronous.

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

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

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

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

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

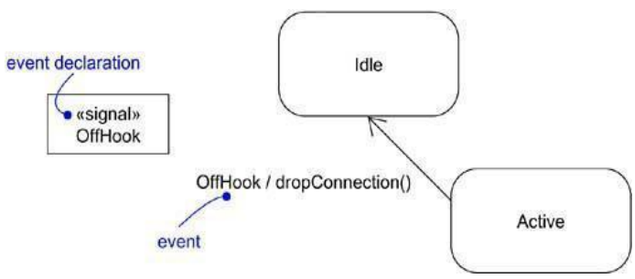


Figure 1: Events

kinds of events

1.Signal Event

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

 a signal event is an asynchronous event

 signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters

 A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction

 signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send.

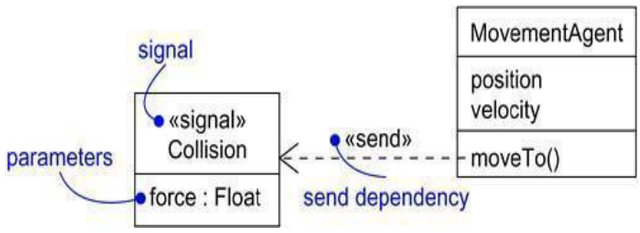


Figure: Signals

2. Call Event

 a call event represents the dispatch of an operation

 a call event is a synchronous event

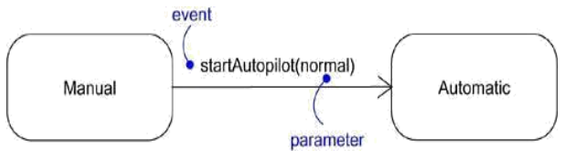


Figure: Call Events

3. Time and Change Events

 A *time event* is an event that represents the passage of time.

 modeled by using the keyword 'after' followed by some expression that evaluates to a period of time which can be simple or complex.

 A *change event* is an event that represents a change in state or the satisfaction of some condition

 modeled by using the keyword 'when' followed by some Boolean expression

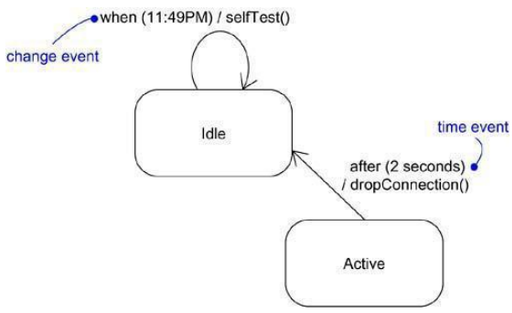


Figure: Time and Change Events

4. Sending and Receiving Events

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

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

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

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

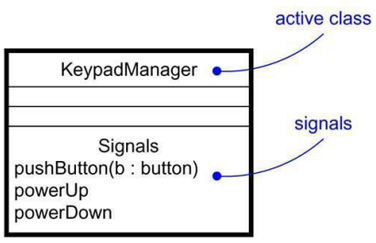


Figure: Signals and Active Classes

Common Modeling Techniques

Modeling family of signals

To model a family of signals,

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

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

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

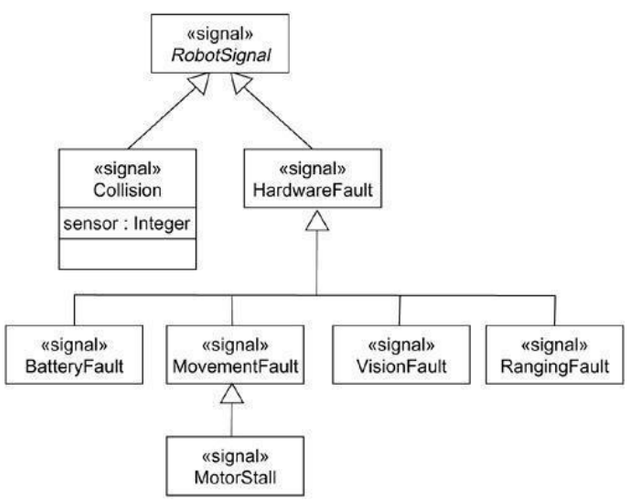


Figure: Modeling Families of Signals

Modeling Exceptions

To model exceptions,

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

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

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

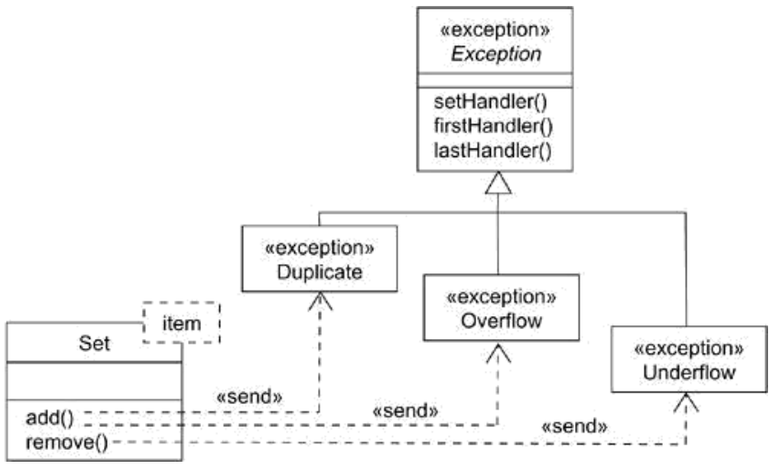


Figure: Modeling Exceptions

STATE MACHINES

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

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

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

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

States

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

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

 a state name must be unique within its enclosing state

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

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

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

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

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

Initial and final states are pseudo-states

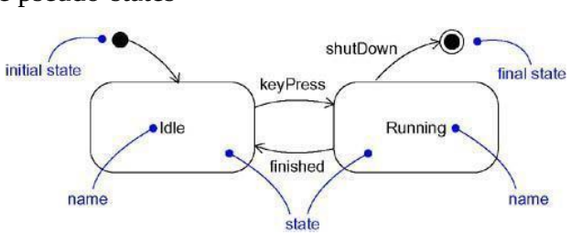


Figure: States

Transitions

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

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

 A transition has five parts:

Source state – The state affected by the transition,

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

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

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

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

 A transition may have multiple sources as well as multiple targets

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

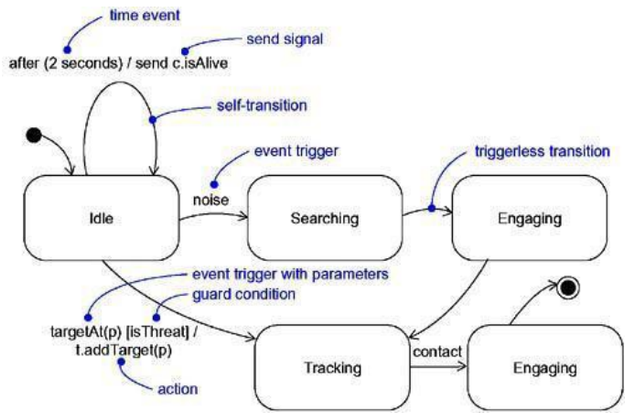


Figure: Transitions

Event Trigger

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

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

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

 An event trigger may be polymorphic

Guard condition

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

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

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

Action

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

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

An activity may be interrupted by other events.

Advanced States and Transitions

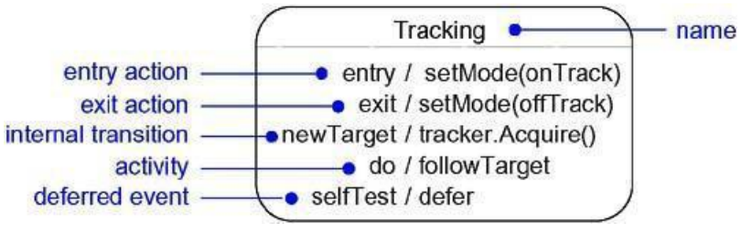


Figure: Advanced States and Transitions

Entry and Exit Actions

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

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

Internal Transitions

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

 Internal transitions may have events with parameters and guard conditions.

Activities

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

Deferred Events

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

Substates

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

 A state that has substates is called a composite state.

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

 Substates may be nested to any level

Sequential Substates

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

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

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

History States

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

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

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

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

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

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

Concurrent Substates

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

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

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

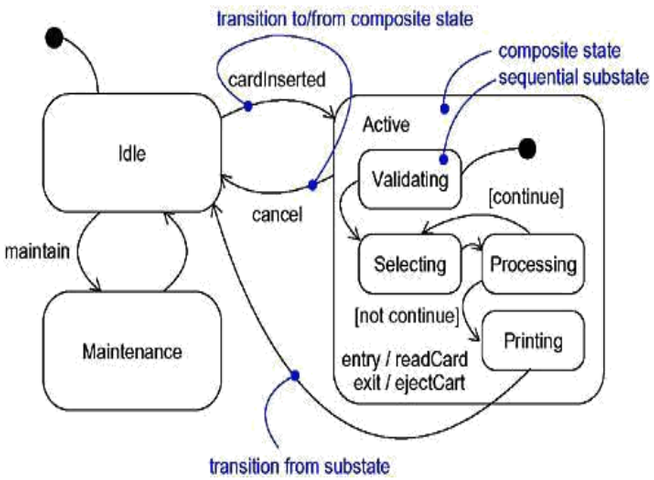


Figure : Sequential Substates

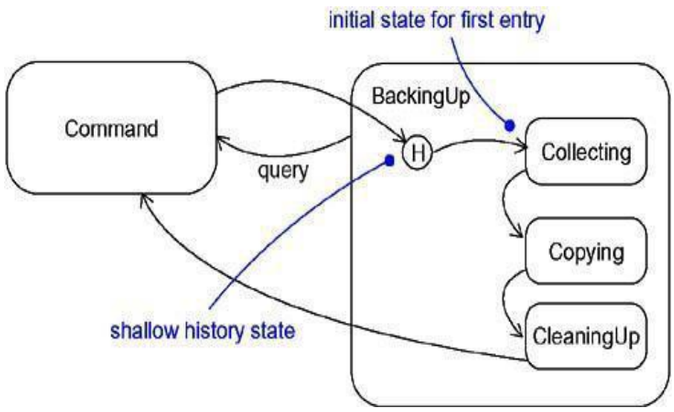


Figure: History State

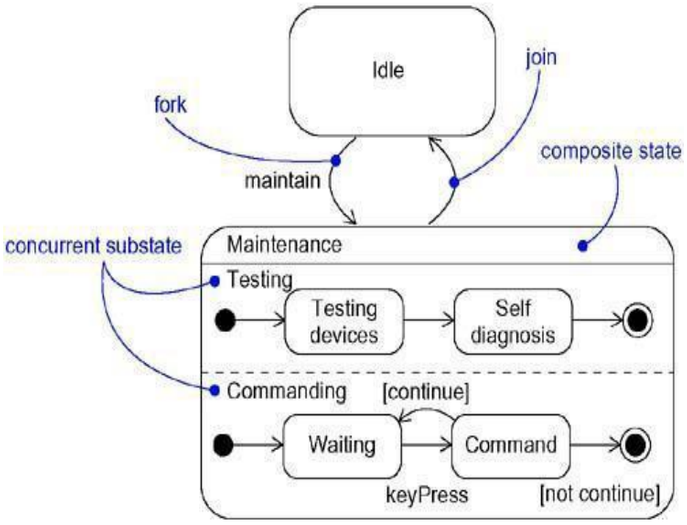


Figure: Concurrent Sub states

Common Modeling Techniques

Modeling the Lifetime of an Object

To model the lifetime of an object,

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

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

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

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

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

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

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

 Expand these states as necessary by using substates.

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

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

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

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

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

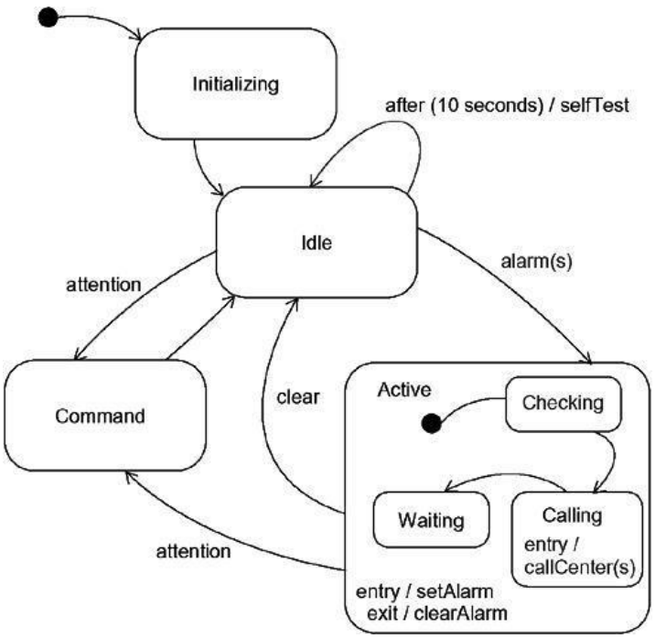


Figure: Modeling the Lifetime of an Object

PROCESSES AND THREADS

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

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

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

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

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

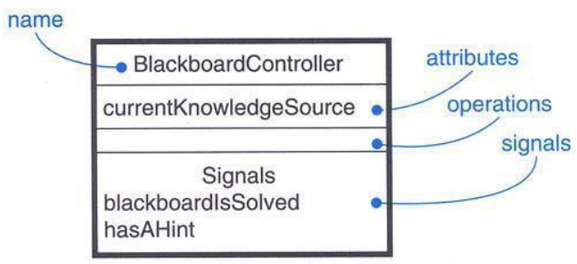


Figure: Active Class

Flow of Control

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

Classes and Events

 Active classes are just classes which represents an independent flow of control

 Active classes share the same properties as all other classes.

 When an active object is created, the associated flow of control is started; when the active

object is destroyed, the associated flow of control is terminated

 Two standard stereotypes that apply to active classes are, **<<process>>** – Specifies a heavyweight flow that can execute concurrently with other processes. (heavyweight means, a thing known to the OS itself and runs in an independent address space) **<<thread>>** – Specifies a lightweight flow that can execute concurrently with other threads within the same process (lightweight means, known to the OS itself.)

 All the threads that live in the context of a process are peers of one another

Communication

 In a system with both active and passive objects, there are *four possible combinations of interaction*

 First, a message may be passed from one passive object to another

 Second, a message may be passed from one active object to another

 In *inter-process communication* there are two possible styles of communication. *First*, one active object might synchronously call an operation of another. *Second*, one active object might asynchronously send a signal or call an operation of another object

 a synchronous message is rendered as a full arrow and an asynchronous message is rendered as a half arrow

 Third, a message may be passed from an active object to a passive object

 Fourth, a message may be passed from a passive object to an active one

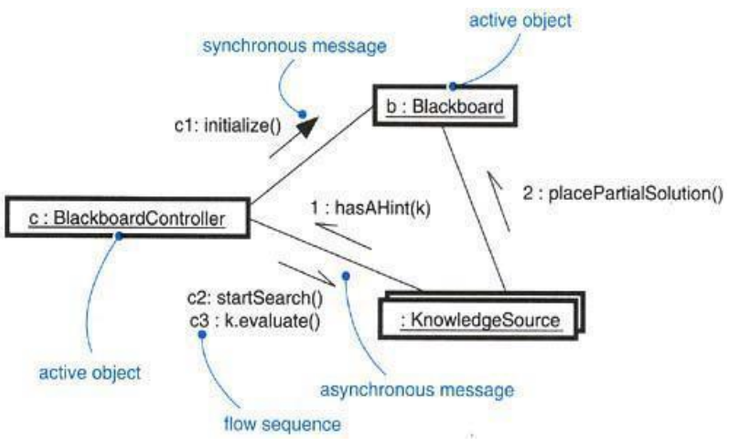


Figure: Communication

Synchronization

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

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

 three approaches are there to handle synchronization:

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

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

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

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

Figure: Synchronization

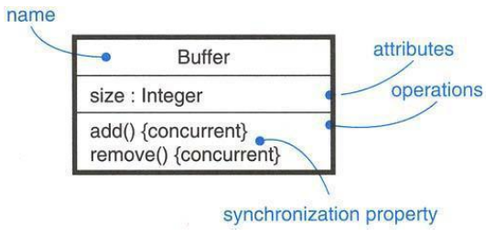


Figure: Synchronization

Process Views

 The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.

 This view primarily addresses the performance, scalability, and throughput of the system.

Common Modeling Techniques

Modeling Multiple Flows of Control

To model multiple flows of control,

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

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

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

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

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

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

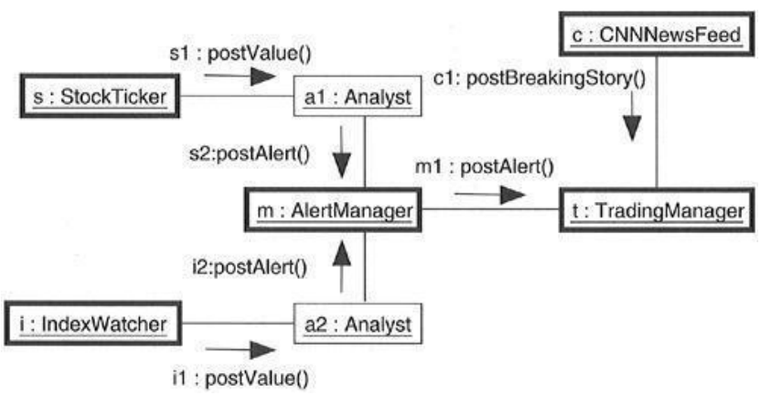


Figure: Modeling Flows of Control

Modeling InterProcess Communication

To model interprocess communication,

 Model the multiple flows of control.

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

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

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

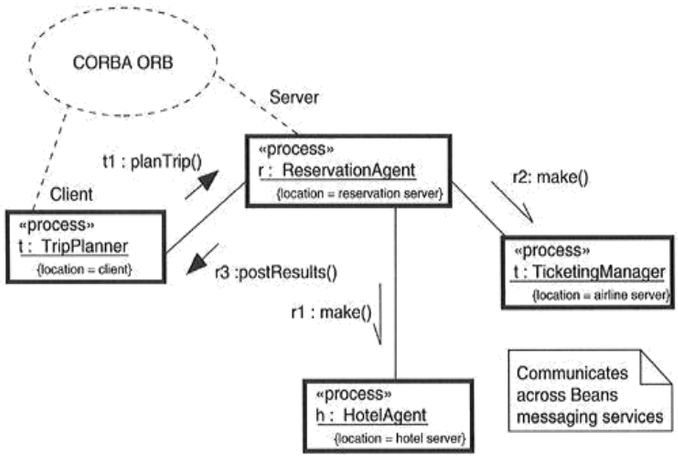


Figure: Modeling Inter process Communication

TIME AND SPACE

A distributed system is one in which components may be physically distributed across nodes. These nodes may represent different processors physically located in the same box, or they may even represent computers that are located half a world away from one another.

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

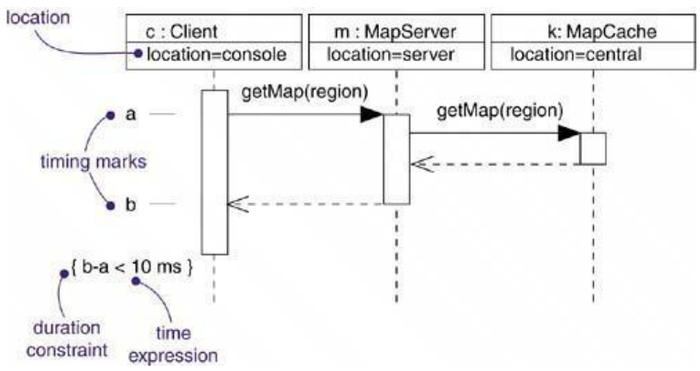


Figure: Timing Constraints and Location

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

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

A *timing constraint* is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint-that is, a string enclosed by

brackets and generally connected to an element by a dependency relationship.

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

Time

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

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

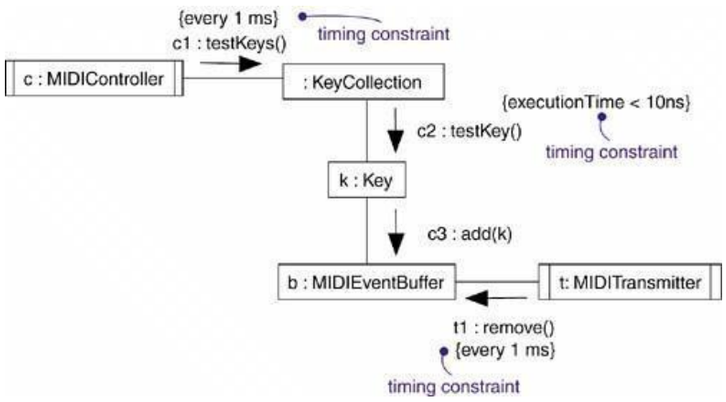


Figure: Time

Location

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

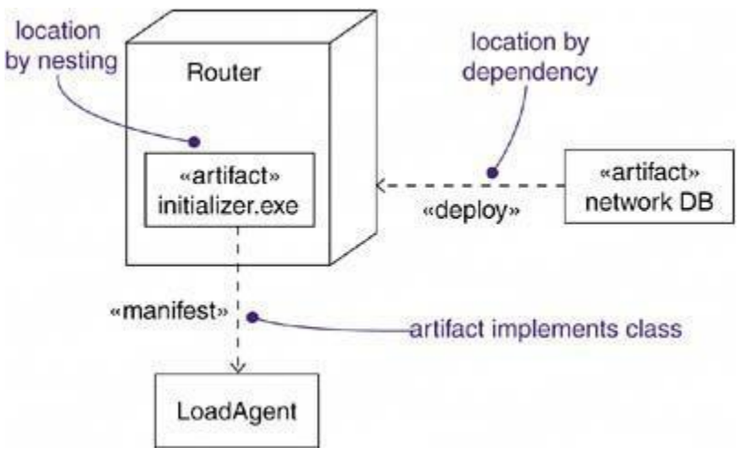


Figure: Location

Common Modeling Techniques

Modeling Timing Constraints

To model timing constraints,

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

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

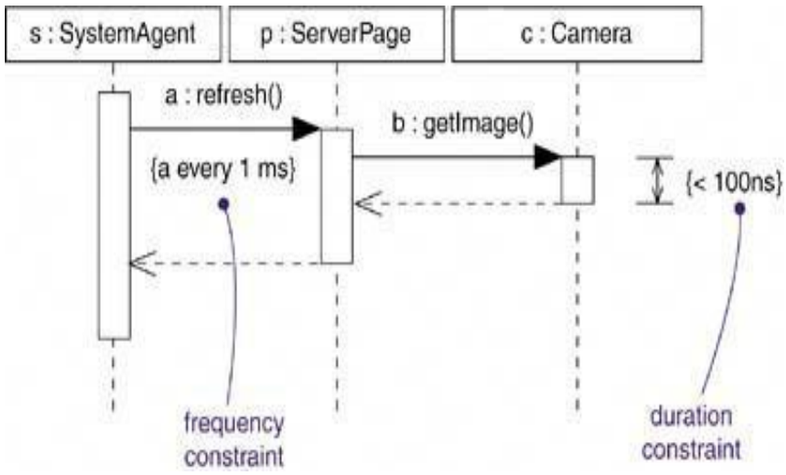


Figure: Modeling Timing Constraint

Modeling the Distribution of Objects

To model the distribution of objects,

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

 Tentatively allocate objects closest to the actors that manipulate them.

 Next consider patterns of interaction among related sets of objects.

 Partition sets of objects that have low degrees of interaction.

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

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

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

 Assign components to nodes so that the computation needs of each node are within capacity. Add additional nodes if necessary.

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

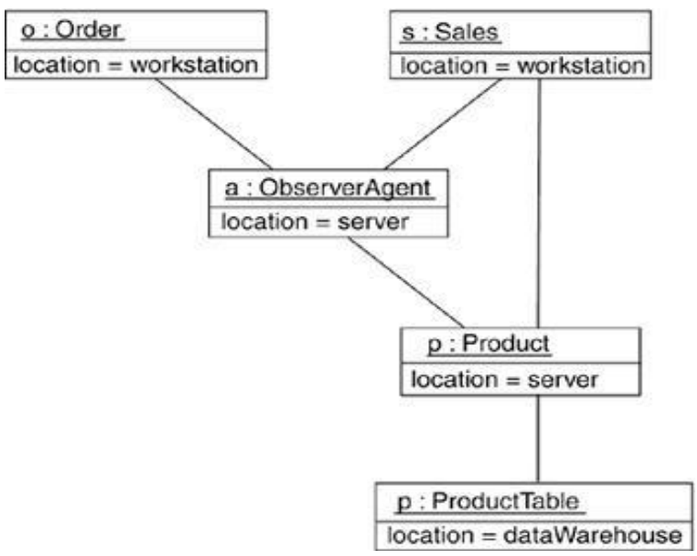


Figure: Modeling the Distribution of Objects

STATE CHART DIAGRAMS

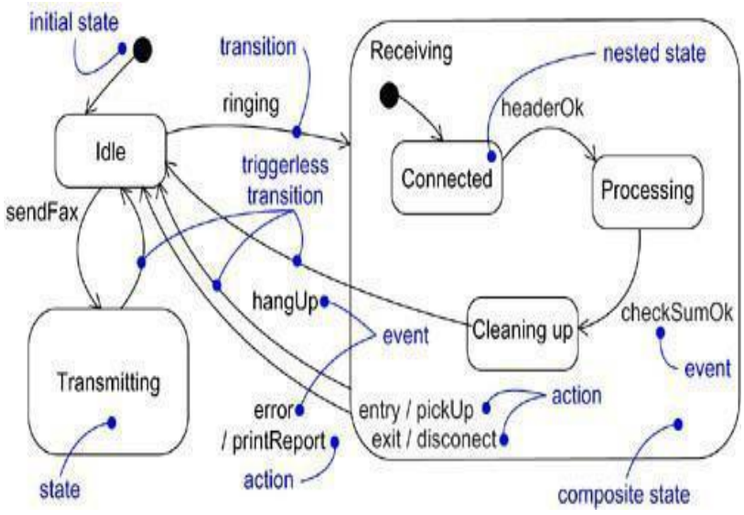


Figure: State chart Diagram

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

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

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

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

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

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

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

Modeling Reactive Objects

To model a reactive object,

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

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

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

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

 Decide on the events that may trigger a transition from state to state. Model these events as

triggers to transitions that move from one legal ordering of states to another.

 Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).

 Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.

 Check that all states are reachable under some combination of events.

 Check that no state is a dead end from which no combination of events will transition the object out of that state.

 Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

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

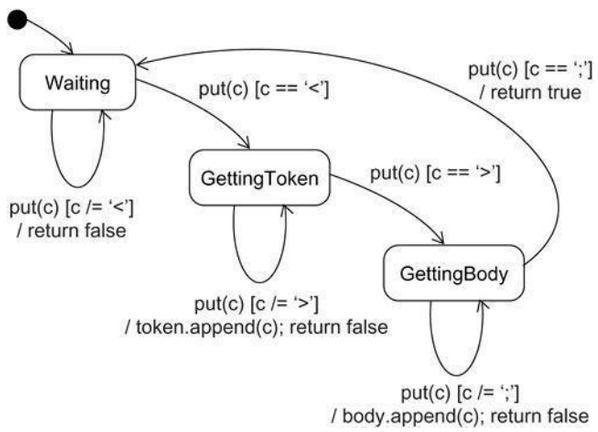


Figure: Modeling Reactive Objects

IMPORTANT QUESTIONS

1. Discuss about the state chart diagrams with example in detailed.
2. How to model distribution of responsibilities of a system?
3. What is an event and explain the type of events with examples?
4. Explain in detail about the concept of state machine?
5. Explain in detail about the concept of processes and threads?

UNIT – VI

Architectural Modelling: Component, Deployment, Component diagrams and Deployment diagrams. Case Study: The Unified Library application.

COMPONENT

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

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

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.
- 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 shown in the following figure 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.

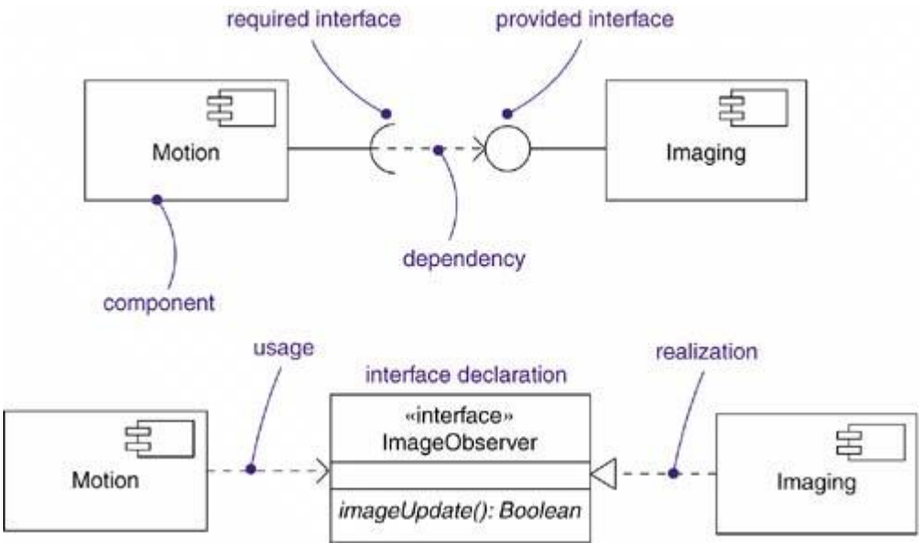


Figure: Components and Interfaces

Note

Interfaces apply at multiple levels just like other elements. The design-level interface you find used or realized by a component will map to an implementation-level interface used or realized by the artifact that implements the component.

Replaceability

A component is *replaceable*. A component is substitutable it 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 recursive a system at one level of abstraction may simply be a component at a higher level of abstraction.

Finally, 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. We 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

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.

A port is shown as a small square straddling the border of a component it 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.

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.

The following figure 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.

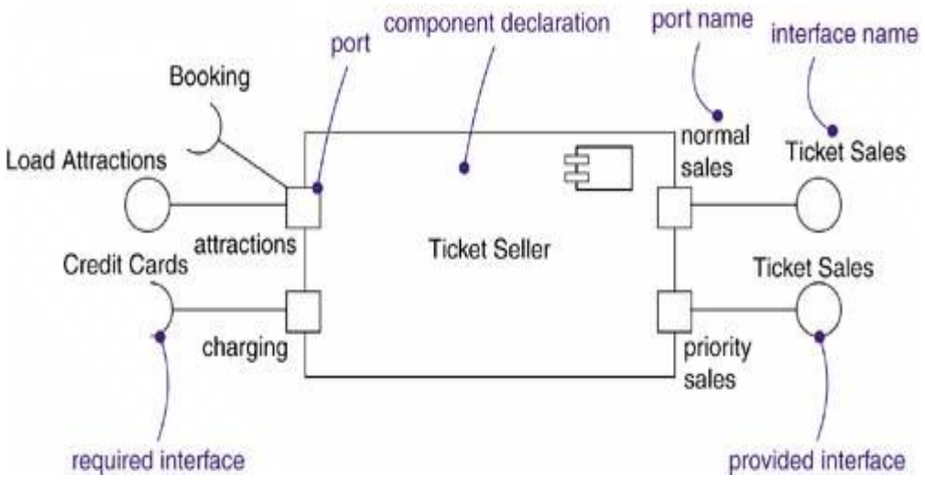


Figure: Ports on a Component

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 theatre 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

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 behaviour 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.

The following figure 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.

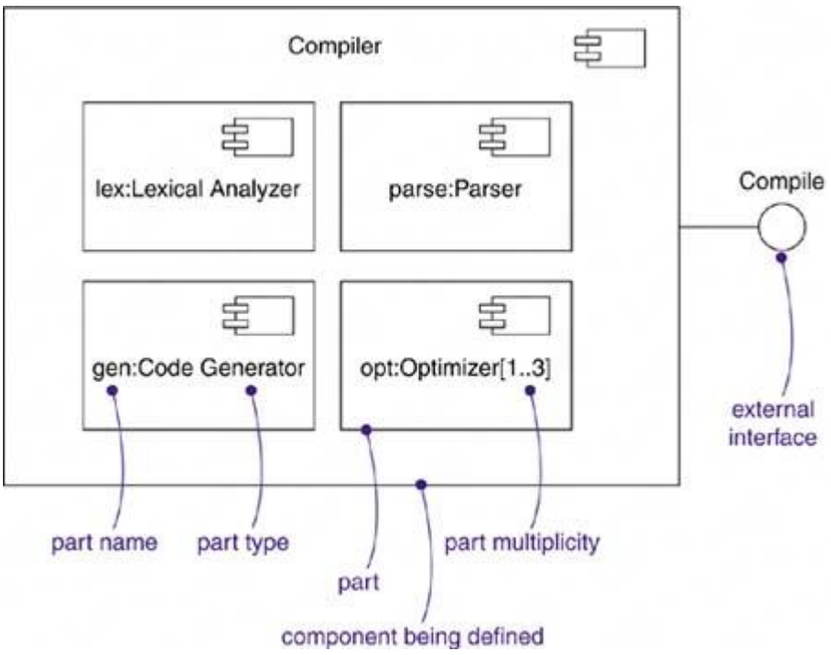


Figure: Parts With in a Component

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 the following figure, 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.

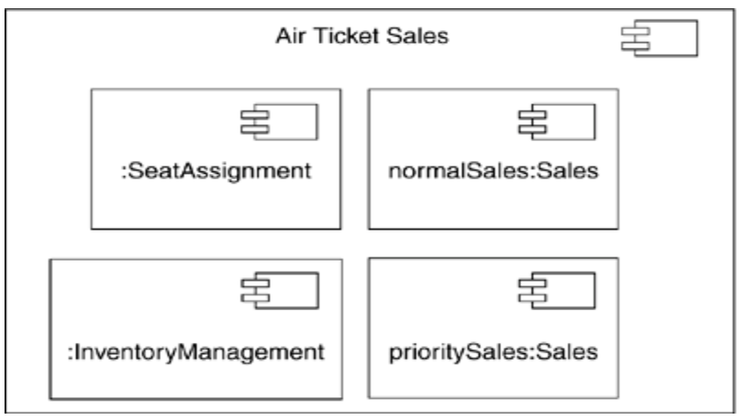


Figure: Parts of the Same Type

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*.

We can show connectors in two ways. 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.

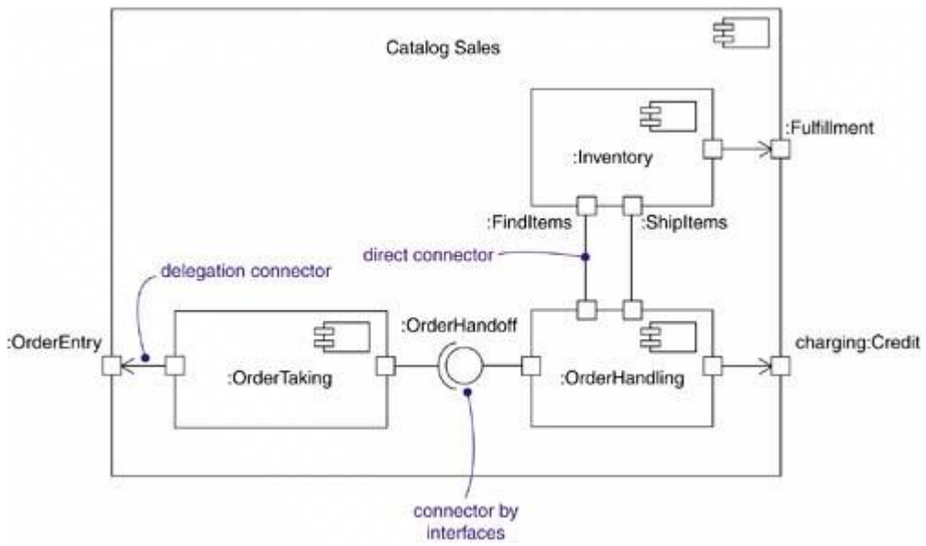


Figure: Connectors

We 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.

The following figure 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 OrderHandoffinterface. 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 Fulfilment 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

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.

The following figure 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.

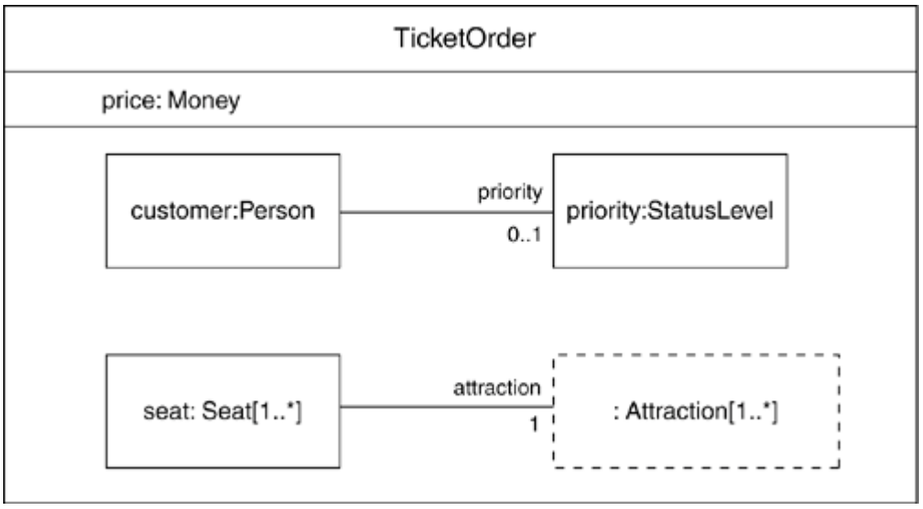


Figure: Structured Class

Modeling an API

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.

The following figure 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.

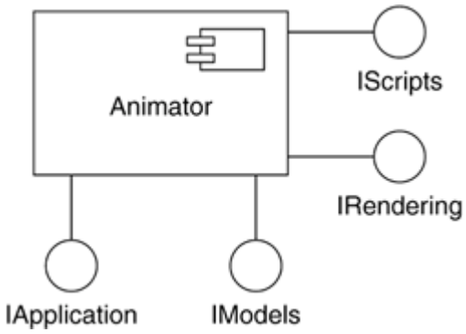


Figure: Modeling an API

DEPLOYMENT

The UML provides a graphical representation of node. This canonical notation permits you to visualize a node apart from any specific hardware. Using stereotypes one of the UML's extensibility mechanisms you can (and often will) tailor this notation to represent specific kinds of processors and devices.

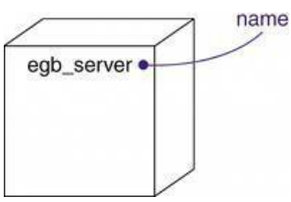


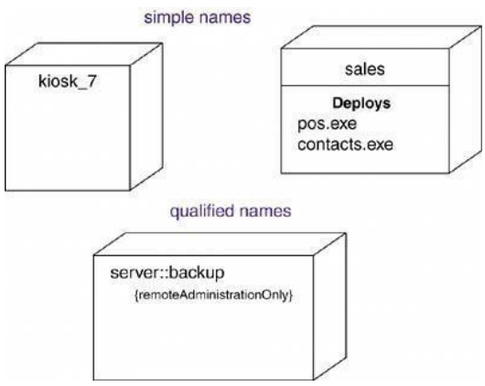
Figure: Nodes

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.

Figure Nodes with Simple and Qualified Names



Nodes and components

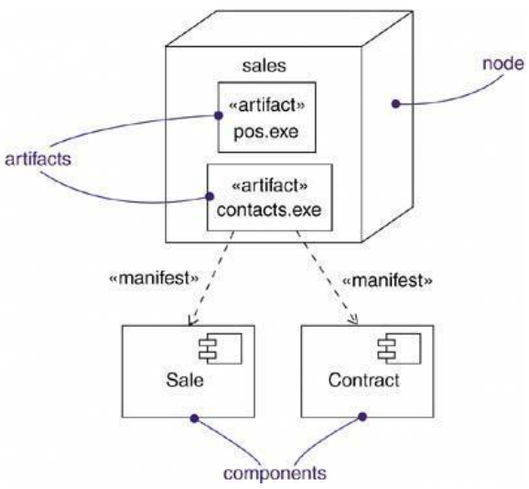
Nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. significant differences between nodes and components are.

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

This first difference is, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. A component is the manifestation of a set of logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be manifested by one or more components, and, in turn, an component may be deployed on one or more nodes.

Figure: Nodes and Components



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

Organizing Nodes

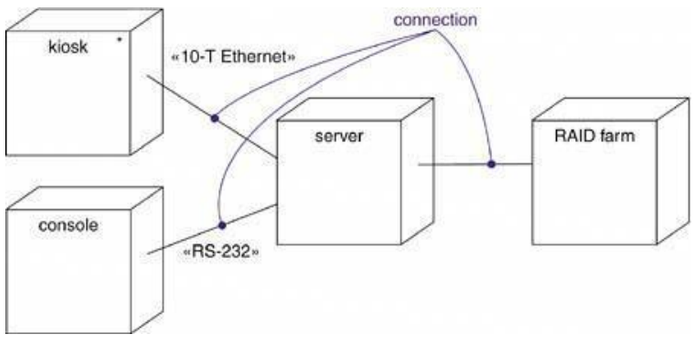
- You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.
- You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship 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.

We can include roles, multiplicity, and constraints.

Figure: Connections



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.

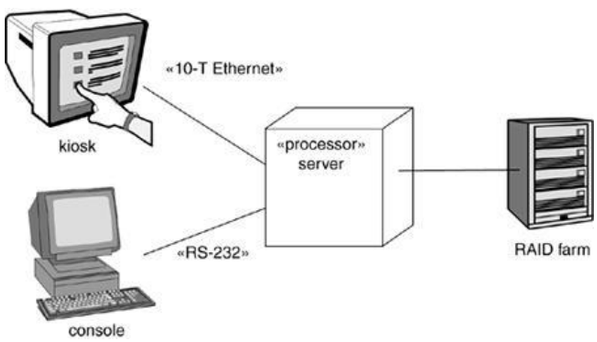
A *processor* is a node that has processing capability, meaning that it can execute a component.

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.

Figure: Processors and Devices



Modeling the Distribution of Components

To model the distribution of components,

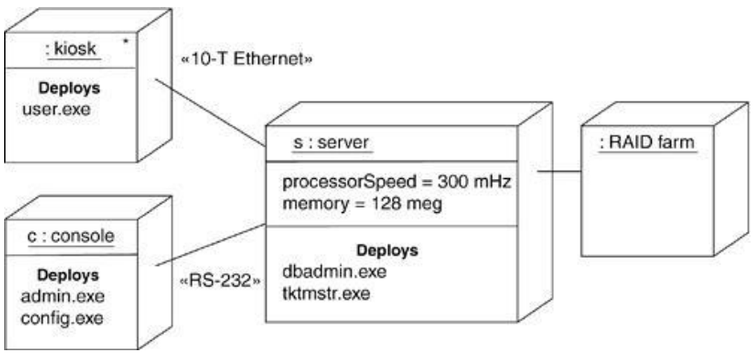
- For each significant component in your system, allocate it to a given node.
- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.

1. Don't make the allocation visible, but leave it as part of the backplane of your modelthat is, in

each node's specification.

2. Using dependency relationships, connect each node with the components it deploys.
3. List the components deployed on a node in an additional compartment.

Figure: Modeling the Distribution of Components



COMPONENT DIAGRAMS

- Component diagrams are used in modeling the physical aspects of object-oriented systems.
- A component diagram shows the organization and dependencies among a set of components.
- Component diagrams are used to model the static implementation view of a system.
- Component diagrams are essentially class diagrams that focus on a system's components.
- Graphically, a Component diagram is a collection of vertices and arcs.
- Component diagrams are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- Component diagrams commonly contain Components, Interfaces and Dependency, generalization, association, and realization relationships. It may also contain notes and constraints.

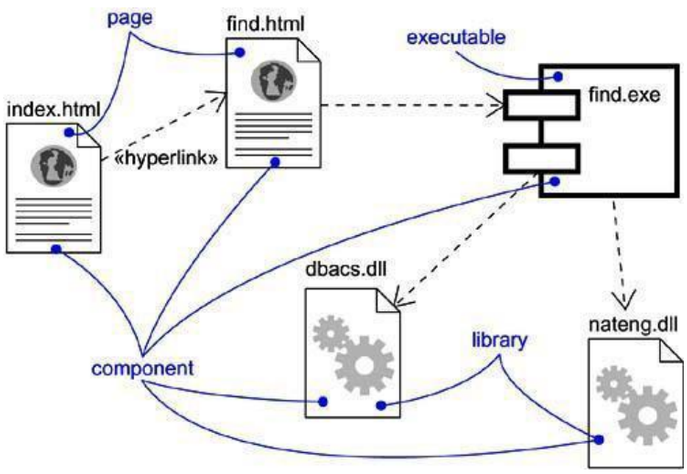


Figure: Component Diagram

Common Modeling Techniques

Modeling Source Code

To model a system's source code,

- Either by forward or reverse engineering identifies the set of source code files of interest and model them as components stereotyped as files.

- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

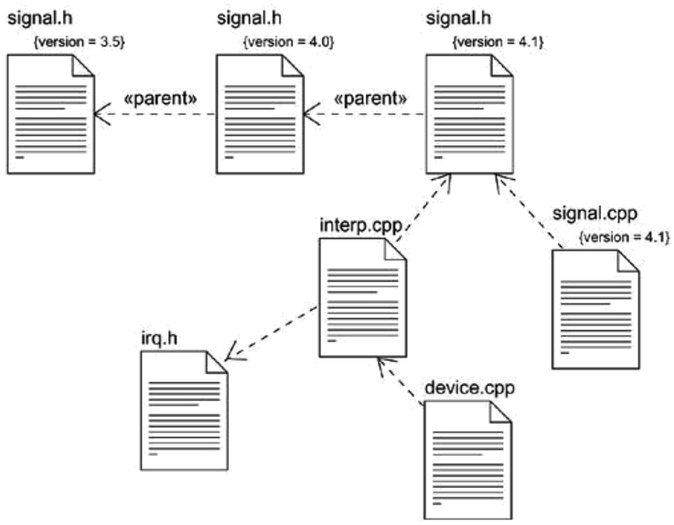


Figure: Modeling Source Code

Modeling an Executable Release

To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

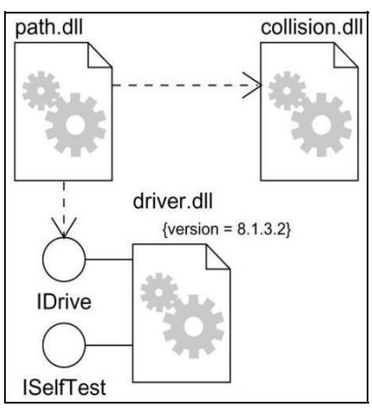


Figure: Modeling an Executable Release

Modeling a Physical Database

To model a physical database,

- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.

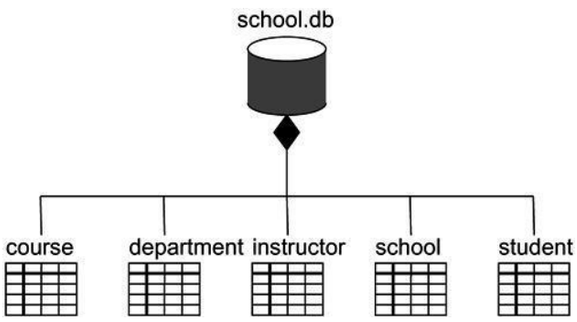


Figure: Modeling a Physical Database

Modeling Adaptable Systems

To model an adaptable system,

- Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

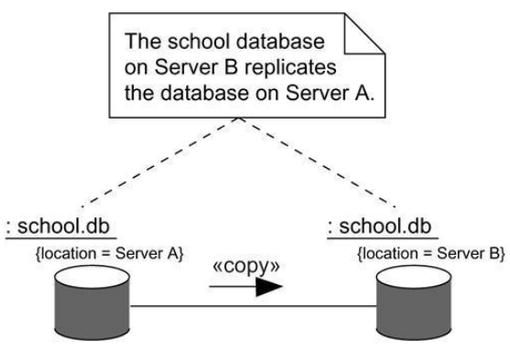


Figure: Modeling Adaptable Systems

DEPLOYMENT DIAGRAMS

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.

- Used to model the static deployment view of a system (topology of the hardware)
- A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.
- Graphically, a deployment diagram is a collection of vertices and arcs.
- Deployment diagrams commonly contain Nodes and Dependency & association relationships. It may also contain notes and constraints.
- Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

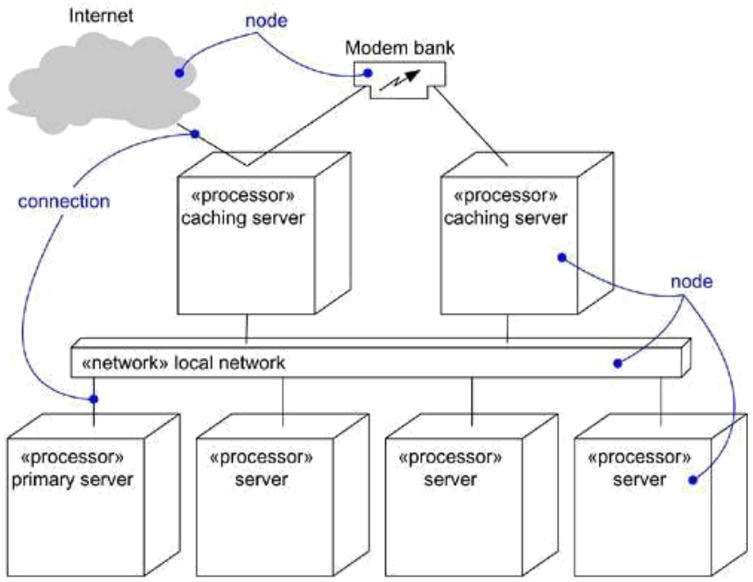


Figure: Deployment Diagram

Common Modeling Techniques

Modeling an Embedded System

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

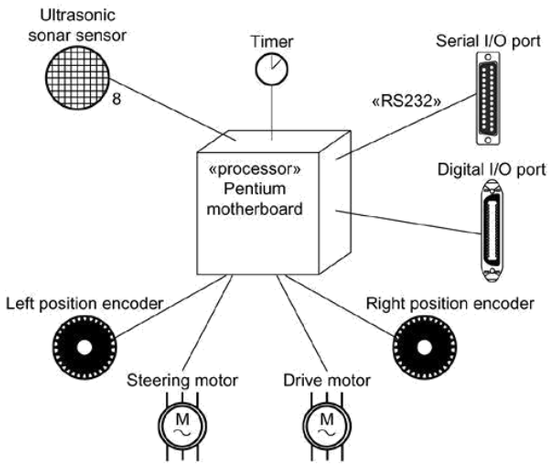


Figure: Modeling in Embedded System

Modeling a Fully Distributed System

To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems. If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.\
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.

If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

- When modeling a fully distributed system, it's common to reify the network itself as an node.

eg:- Internet, LAN, WAN as nodes

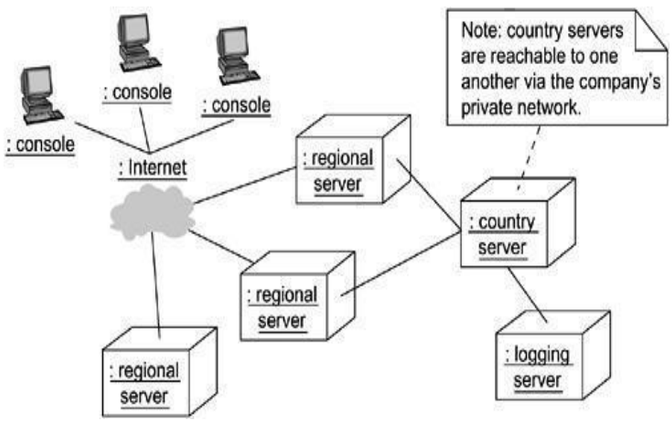


Figure: Modeling a Fully Distributed System

Modeling a Client/Server System

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are

likely to be architecturally significant.

- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

Figure 3 shows the topology of a human resources system, which follows a classical client/server architecture.

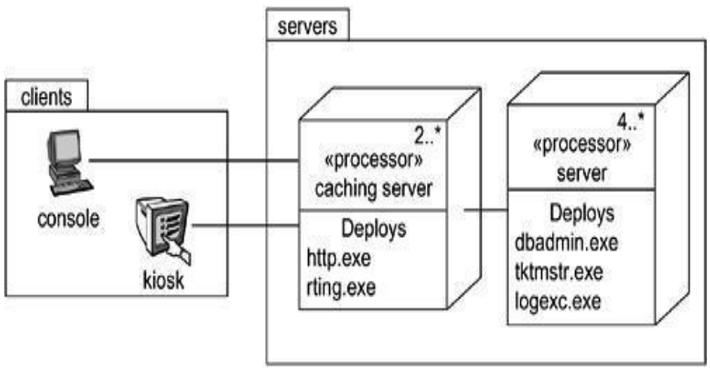


Figure: Modeling a Client/ Server System

IMPORTANT QUESTIONS

1. Discuss about differences of component and deployment diagrams with example
2. Discuss about modeling techniques for component diagrams with example.
3. What are nodes, components and explain the use of component & deployment diagrams in UML?
4. Discuss about modeling techniques for deployment diagrams with example.