UNIT 1

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.

1.1 The Structure of Complex systems

A complex system is a system composed of many components which may interact with each other. In many cases it is useful to represent such a system as a network where the nodes represent the components and the links their interactions. Examples of complex systems are Personal Computer, Plants, and Education System.

The structure of personal computer

A personal computer is a device of moderate complexity. Most are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a CD or DVD drive and hard disk drive. We may take any one of these parts and further decompose it. For example, a CPU typically encompasses primary memory, an arithmetic/logic unit (ALU), and a bus to which peripheral devices are attached. Each of these parts may in turn be further decomposed: An ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

Central Processing Unit (CPU) — *Executes programs*

- Arithmetic & Logic Unit (ALU)
  - Registers
    - NAND gate
      - CMOS Gate
      - Interconnect
    - Inverter
  - Control Logic
- Primary Memory
- Bus — *peripheral devices*

- Hard disk drive — *Persistent storage for data*
- Monitor — *Outputs data*
- Keyboard — *Inputs data*
- Secondary storage device (DVD drive / USB) — *Removable storage for data*

Complex systems hierarchies

- Each Level of Hierarchy represents a Layer of Abstraction
- Each Layer
  - Is built on top of other layers: *CMOS Gates —> NAND Gates—> Registers*
  - (in turn) Supports other layers: *CMOS Gates —> NAND Gates—> Registers*

    - Is independently understandable: *CPU*
    - Works independently with clear separation of concerns: *ALU, Memory*
- Common services / properties are shared across Layers: *Same power-tree feeds the components of CPU*
- Layers together show **Emergent Behavior**
- Behavior of the whole is greater than the sum of its parts
- Systems demonstrate cross-domain commonality: *Cars have processors, memory, display*

1.2 The Inherent Complexity of Software

There are two types of software are there in terms of complexity

- Simple Software
- Industrial-strength software

Simple Software

Simple Software is software with limited set of behaviors and not very complex. It is specified, constructed, maintained, and used by the same person or a small group 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. Can be thrown away and replaced with entirely new software rather than attempt to reuse them, repair them, or extend their functionality

Industrial-strength software

Industrial-strength software applications exhibits a very rich set of behaviors, 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 these tend to have a long life span, and over time, many users come to depend on their proper functioning. Complexity of Industrial-Strength Software systems exceeds the human intellectual capacity

Why Software Is Inherently Complex?

Inherent complexity derives from four elements:

1. The complexity of the problem domain,
2. The difficulty of managing the development process,
3. The flexibility possible through software,
4. The problems of characterizing the behavior of discrete systems.

The complexity of the problem domain

Domains are difficult to understand. Consider the requirements for the electronic system of a multiengine aircraft, a cellular phone switching system, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend. The functional requirements

complex to master and often are competing, even contradictory. Non-functional Requirements (usability, performance, cost, survivability, and reliability) are often Implicit and difficult to justify in the budget.

This external complexity usually springs from the ―communication gap‖ that exists between the users of a system and its developers. Users cannot express; developers cannot understand due to lack of expertise across domains.

The difficulty of managing the development process

The task of the software development team is to engineer the illusion of simplicity. We strive to write less code by inventing clever and powerful mechanisms. Today, it is not unusual to find delivered systems whose size is measured in hundreds of thousands or even millions of lines of code. The complexity of software goes well beyond the comprehension of a single (or small group of) individual/s, even with meaningful decomposition. Hence, we need more developers and more teams. The key management challenge is to maintain a unity and integrity of management.

The flexibility possible through software

Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction. This flexibility turns out to be an incredibly seductive property, however, because it also forces the developer to craft virtually all the primitive building blocks on which these higher-level abstractions stand. While the construction industry has uniform building codes and standards for the quality of raw materials, few such standards exist in the software industry. As a result, software development remains a labor-intensive business.

Problems of characterizing the behavior of discrete systems

Software is built and executed on digital computers. Hence, they are Discrete Systems. A large application would have hundreds or even thousands of variables and more than one (often several) thread of control. A state is entire collection of variables, the current values of variables, the current address of each process, the calling stack of each process. Software has finite number of states. Yet, many of these the states are often intractable and influenced by external factors. This change is not deterministic (unlike continuous system) and needs extensive Testing.

1.3 Attributes of Complex System

1. Hierarchic Structure
2. Relative Primitives
3. Separation of Concerns
4. Common Patterns
5. Stable Intermediate Forms

Hierarchic Structure

- All **systems** are composed of interrelated **sub-systems**
- Sub-systems are composed of sub-sub-systems, and so on

- Lowest level sub-systems are composed of **elementary components**
- All systems are parts of larger systems
- The value added by a system must come from the relationships between the parts, not from the parts per se
- **We can understand only those systems that have a hierarchic structure**

Relative Primitives

- **Subjective Choice** — strongly dependent on the experience and expertise of the designer
- What is primitive for one observer may be at a much higher level of abstraction for another.
- **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**

Separation of Concerns

- Hierarchic systems are:
  - **decomposable** — *can be divided into identifiable parts*
  - **nearly decomposable** — *the parts are not completely independent*
- Difference between intra- and intercomponent interactions provides a clear Separation of Concerns among the various parts of a system — helps the analysis and design in isolation

Common Patterns

- Complex systems have **Common Patterns**
- Complex systems are composed of only a few different kinds of subsystems in various combinations and arrangements (cells found in both plants and animals etc.)
- Common Patterns are a major source of reuse in OOAD. Examples include Design Patterns, STL in C++, Data Structures in Python etc.

Stable Intermediate Forms

- It is extremely difficult to design a complex system correctly in one go
- Start with a simple system and then refine (Iterative Refinement)
- Objects, once considered complex, become the primitive objects on which more complex systems are built
- The system matures from one intermediate form to the next
- **A complex system that works is invariably found to have evolved from a simple system that worked**

1.4 Organized and Disorganized Complexity

The Canonical Form of a Complex System: The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex system. Because we can apply knowledge of (gained from) one system to other system (e.g., If you can drive one type of car, it‘s easier for you to drive another type of car)

Hierarchies are of multiple types

- Decomposition — part—of or HAS—A
- Abstraction — IS—A

It is essential to view system from both perspectives. ‑Part of‖ hierarchy is also known as Class Structure. ‑IS A‖ hierarchy is also known as Object Structure. 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.

In Figure 1–1 we see the two orthogonal hierarchies of the system: its class structure and its object structure. Each hierarchy is layered, with the more abstract. Classes and objects built on more primitive ones. What class or object is chosen as? Primitive is relative to the problem at hand. Looking inside any given level reveals yet another level of complexity. Especially among the parts of the object structure, there are close collaborations among objects at the same level of abstraction.

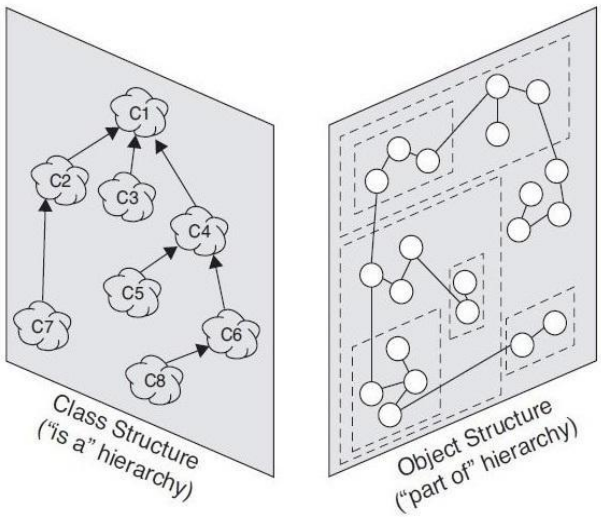


Figure 1–1 The Key Hierarchies of Complex Systems

The Limitations of the Human Capacity for Dealing with Complexity

Maximum number of chunks of information that an individual can simultaneously comprehend is on the order of seven, plus or minus two. Human channel capacity is related to the capacity of short-term memory. The Processing speed is a limiting factor — it takes the mind about five seconds to accept a new chunk of information. This leads to **Disorganized Complexity**

1.5 Bringing Order to Chaos

Principles that will provide basis for development

- Decomposition
- Abstraction
- Hierarchy

The Role of Decomposition

Decomposition techniques are divided into two types they are

- Algorithmic Decomposition
- Object-Oriented Decomposition

Algorithmic 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.

Object-Oriented Decomposition: Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem. We view the world as a set of autonomous agents that collaborate to perform some higher level behavior. Calling one operation creates another object. In this manner, each object 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.

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: Increase the semantic content of individual chunks of information by explicitly recognizing the class and object hierarchies. Object structure illustrates how different objects collaborate with one another through patterns of interaction that we call mechanisms Class structure highlights common structure and behavior within a system.

1.6 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: The purpose of design is to construct a system that:

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

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.

1.7 Evolution of Object Model

The elements of the object oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy,

typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

Generations of Programming Languages

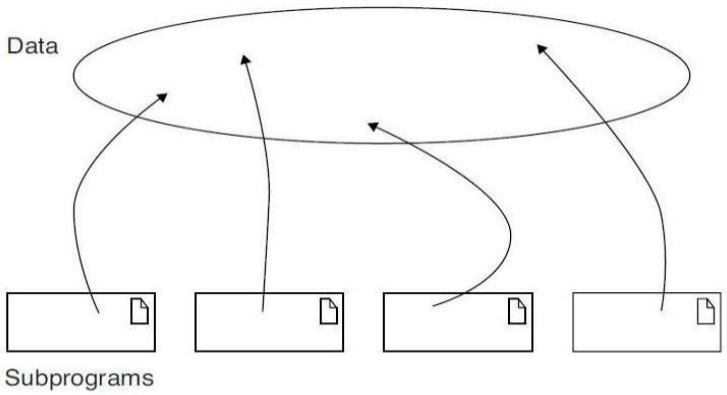
- First-generation Languages (1954-1958)
- Second-generation Languages (1959-1961)
- Third-generation Languages (1962-1970)
- Fourth-generation Languages (1970-1980)
- Object-orientation Boom (1980-1990)
- Emergence of Frameworks (1990—today)

First-generation Languages (1954-1958)

- *Major Features*
  - **Formula Translation**
  - **Mathematical Computation**
  - **Notions of Programming:**
    - Variable and Literal
    - Expressions
    - Unconditional & Conditional Flow
- *Languages*
  - FORTRAN I (Mathematical expressions)
  - ALGOL 58 (Mathematical expressions)
  - Flowmatic (Mathematical expressions)
  - IPL V (Mathematical expressions)

Topology of First-generation Languages

- Applications in these languages exhibit:
  - Flat physical structure
  - Only of global data and subprograms
  - Error in one part of a program can have effect across the rest of the system



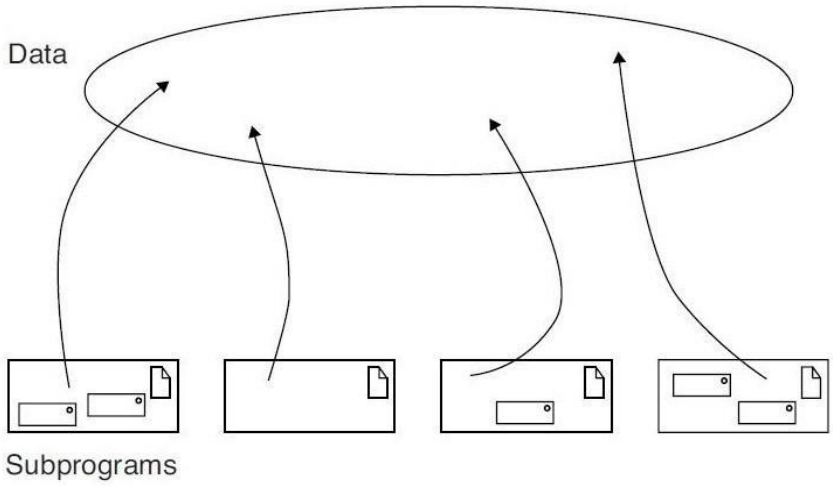
The Topology of First- and Early Second-Generation Programming Languages

Second-generation Languages (1959-1961)

- *Major Features*
  - **Blocks and Subprograms**
  - **Notions of Programming:**
    - Data Type & Description
    - Statements e Parameter Passing Mechanisms
    - Files
    - Pointers
    - List
    - Data Structures
- *Languages*
    - FORTRAN II (Subroutines, separate compilation)
    - ALGOL 60 (Block structure, data types)
    - COBOL (Data description, file handling)
    - Lisp (List processing, pointers, garbage collection)

Topology of Second-generation Languages

- Applications in these languages exhibit:
  - Foundations of structured programming - nested subprograms, control structures and scope and visibility of declarations
  - Support parameter-passing mechanisms
  - Fails to address the problems of programming-in-the-large and data design



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

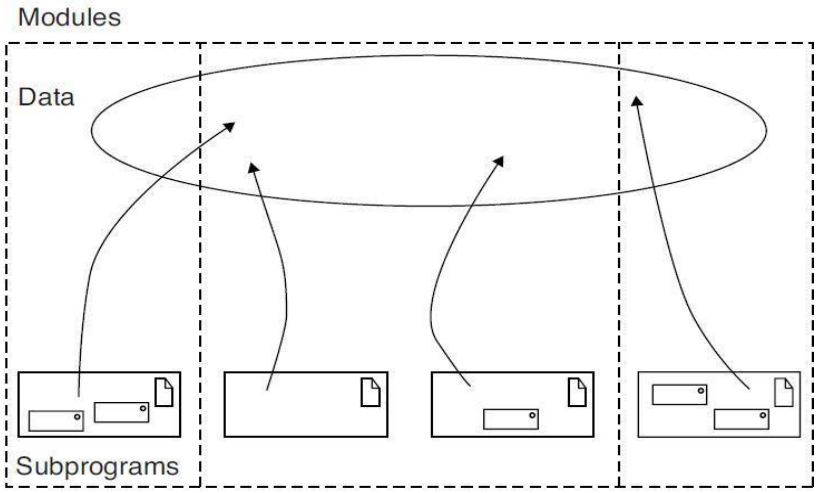
Third-generation Languages (1962-1970)

- *Major Features*
  - **Structured Programming**
  - **Modularity**

        - **Abstraction**
        - **Notions of Programming:**
            - Control Constructs
            - Class
        - **One-language-fits-it-all**
- *Languages*
    - PL/1 (FORTRAN + ALGOL + COBOL)
    - ALGOL 68 (Rigorous successor to ALGOL 60)
    - Pascal (Simple successor to ALGOL 60)
    - Simula (Classes, data abstraction)

Topology of Third-generation Languages

- Applications in these languages exhibit:
    - Support separately compiled module (group subprograms that were most likely to change together)
    - Dismal support for data abstraction and strong typing, hence such errors could be detected only during execution of the program



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

Fourth-generation Languages (1970-1980)

- Major Features
    - **Relational Models for large data handling**
    - **Notions of Programming:**
        - Low-level access for Systems Programming
        - Efficiency
    - **Moving away from "One-language-fits-it-all"**
- Languages
    - C (Efficient, small executable)

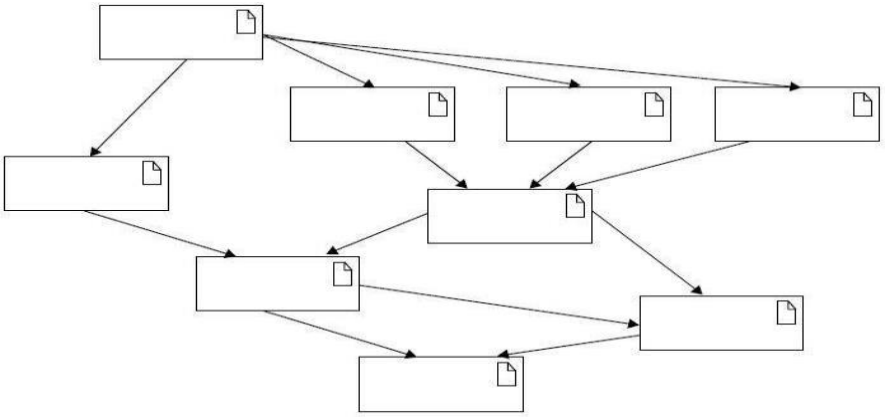
- FORTRAN 77 (ANSI standardization)
- SQL (Structured Query Language)

Object-orientation Boom (1980-1990)

- Major Features
  - **Object Models**
  - **Notions of Programming:**
    - Strong Typing
    - Safety
    - Exception
- Languages
  - Smalltalk 80 (Pure object-oriented language)
  - C++ (Derived from C and Simula)
  - Ada83 (Strong typing; heavy Pascal influence)
  - Eiffel (Derived from Ada and Simula)

Topology of Small to Medium Sized Applications Using OB and OOP Languages

- Applications in Object Based (OB) and Object Oriented Programming (OOP) languages exhibit:
  - Fundamental logical building blocks are no longer algorithms, but instead are classes and objects
  - Little or no global data
  - Classes, objects, and modules provide an essential yet insufficient means of abstraction



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

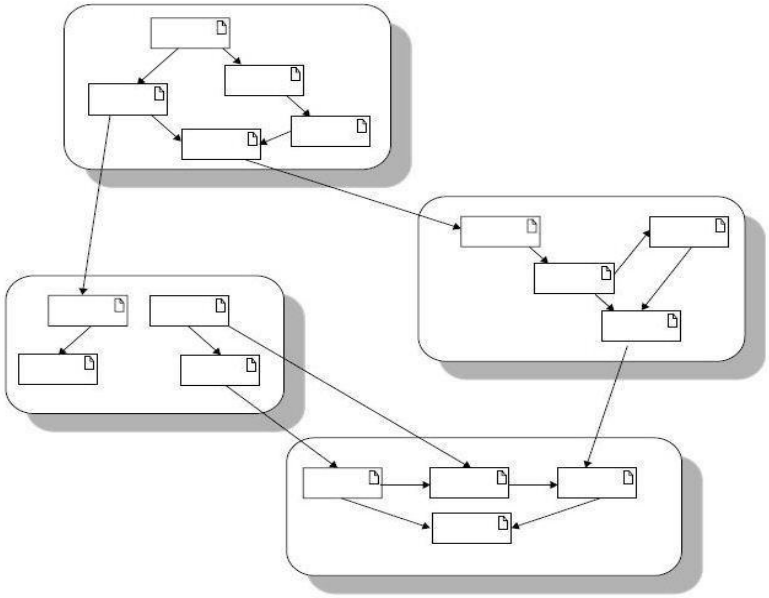
Emergence of Frameworks (1990—today)

- *Major Features*
  - **Programming Frameworks**
  - **Use of OOAD for Large Scale Systems**

- **Notions of Programming:**
  - Dynamic Typing
  - Portability
  - Threads
- **Frameworks:**
    - J2SE, J2EE, J2ME (Java-based frameworks for standard, enterprise & mobile computing)
    - .NET (Microsoft's object-based framework)

- *Languages*
  - Visual Basic (Development GUI for Windows applications)
  - Java (Successor to Oak; designed for portability)
  - Python (00 scripting, portable, dynamically typed)
  - Visual C# (Java competitor for .NET Framework)
  - VB.NET (Visual Basic for NET Framework)

- Applications in these language frameworks exhibit:
  - Programming-in-the-large
  - At any given level of abstraction, meaningful collections of objects achieve some higher-level behavior



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

1.8 Foundation of Object Model

Structured design methods evolved to guide developers who use procedural language and algorithms as their fundamental building blocks to build complex system.

Similarly, object-oriented design methods have evolved to help developers who exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.
With Algorithmic Decomposition alone, only limited amount of complexity can be handled — hence we turn to Object-Oriented Decomposition. The following events have contributed to the evolution:

- Advances in computer architecture
- Advances in programming languages
- Advances in programming methodology
- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

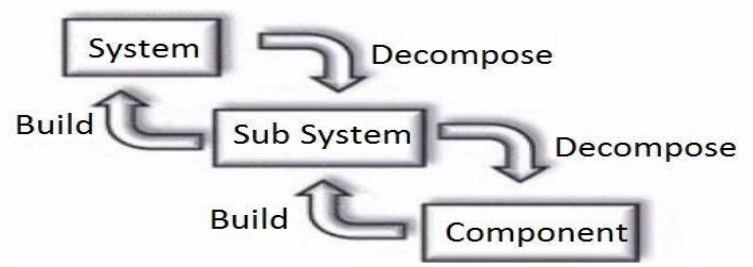
Object Oriented Analysis (OOA)

Object-Oriented Analysis is a method of analysis that examines ***requirements*** from the perspective of the ***classes*** and ***objects*** found in the ***vocabulary of the problem domain***
The stages for Object-Oriented Analysis are:

- Identifying Objects
- Identifying Structure
- Identifying Attributes
- Identifying Associations
- Defining Services

Object Oriented Design (OOD)

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 state and dynamic models of the system under design.
There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.



Object-oriented Decomposition

Object-Oriented Programming

What is object-oriented programming (OOP)? We define it as follows:

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

There are three important parts to this definition:

(1) Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks

(2) Each object is an instance of some class; and

(3) Classes may be related to one another via inheritance relationships

1.9 Elements of Object Model

There are four major (mandatory and essential) elements of the object model:

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

In addition, there are three minor (useful but not essential) elements of the object model:

1. Typing
2. Concurrency
3. Persistence

Abstraction

Abstraction refers to the process of presenting essential features without including any complex background details. Focus attention on only one aspect of the problem and ignore irrelevant details. It is also called as Model Building. Abstraction focuses on the outside view of an object. Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Abstraction: Example — Hydroponics Gardening System

- Hydroponics farm:
  - Plants are grown in a nutrient solution, without sand, gravel, or other soils
  - Control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations
  - Design an automated system that constantly monitors and adjusts these elements

- An automated gardener should efficiently carry out growing plans for the healthy production of multiple crops

- Key Abstractions:
  - Sensors
  - Heaters
  - Growing Plans

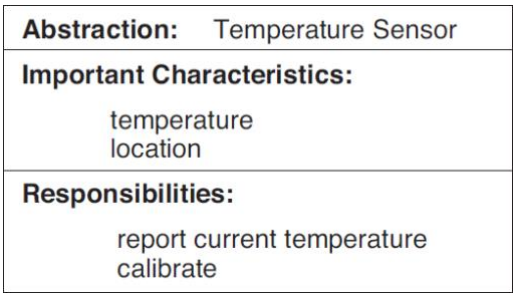


Figure: Abstraction of a Temperature Sensor

Encapsulation

Wrapping up of code and data together into a single unit is known as encapsulation. Encapsulation hides the details of the implementation of an object. Encapsulation compartmentalizes the elements of an abstraction that constitute:

- Structure
- Behavior

Encapsulation Example - Hydroponics Gardening System

A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running. All a client needs to know about the class Heater is its available interface (i.e., the responsibilities that it may execute at the client‘s request).

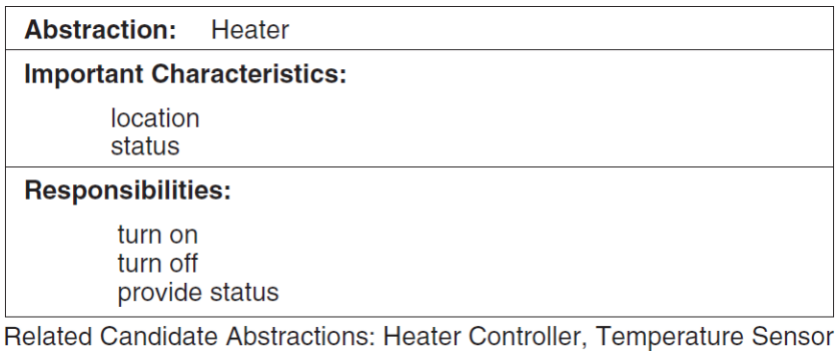


Figure: Abstraction of a Heater

Modularity

Modularity refers to partition a program into individual components to manage complexity. The divided modules compiled separately or together, but Modules connected with the other module. Because the solution may not be known when the design stage starts, decomposition into smaller modules may be quite difficult. Modularization should follow the semantic grouping of the common and interrelated functionality of the system.

Objectives of Modularity

- Modular Decomposition
  - Systematic mechanism to divide problem into individual components
- Modular Composability
  - Enable reuse of existing components
- Modular Understandability
  - Understand module as a unit
- Modular Protection
  - Errors are localized and do not spread to other modules

Modularization: Example 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, a follow the progress of currently active ones. The Key Abstractions are Growing plan, User Interface.

Growing plan

- Create a module whose purpose is to collect all of the classes associated with individual growing plans (for example, FruitGrowingPlan, GrainGrowingPlan)
- The implementations of these GrowingPlan classes would appear in the implementation of this module a

User Interface

- Another 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. Two most important hierarchies in a complex system are

- Class structure or is-a hierarchy (Abstraction or IS—A)
- Object structure or part-of hierarchy (Decomposition or HAS—A)

The common structure & behavior are migrated to the superclass. Superclass represents a generalized abstraction and subclasses represent specializations. A subclass can add, modify & hide methods from the superclass.

Examples of Hierarchy: Single Inheritance

Inheritance is the most important –is a‖ hierarchy, and it is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more super classes. Typically, a subclass augments or redefines the existing structure and behavior of its super classes.

For example, a bear –is a‖ kind of mammal, a house –is a‖ kind of tangible asset, and a quick sort –is a‖ particular kind of sorting algorithm.

Typing

The concept of a type derives primarily from the theories of abstract data types. A type is a precise characterization of structural or behavioral properties which a collection of entities all share. For our purposes, we will use the terms type and class interchangeably. Although the concepts of a type and a class are similar, we include typing as a separate element of the object. A Programming Language may be

- Statically typed
- Dynamically typed
- Strongly typed
- Weakly typed
- Untyped

Polymorphism

Single method exhibiting different behaviours in the same class or different classes is known as polymorphism. Polymorphism is the most powerful feature of object-oriented programming languages next to the support for abstraction.

Concurrency

Concurrency is the property that distinguishes an active object from one that is not active. It allows multiple tasks to execute, interact and collaborate at the same time to achieve the global functionality. Concurrency is critical for Client-Server Model of computation. Concurrency is of two types

- Heavyweight and
- Lightweight Concurrency

Heavyweight Concurrency

A heavyweight process is typically independently managed by the target operating system and so encompasses its own address space. Communication among heavyweight processes is expensive and involves inter-process communication. It is commonly called a Process.

Lightweight Concurrency

A lightweight process lives within a single OS process along with other lightweight processes and shares the same address space. Communication among lightweight processes is less expensive and often involves shared data. It is commonly called a Thread.

1.10 Applying the Object Model

The object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming. The object model offers a number of significant benefits that other models simply do not provide. Most

importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems hierarchy, relative primitives (i.e., multiple levels of abstraction), separation of concerns, patterns, and stable intermediate forms.

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

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilient 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.

UNIT-2
OOAD

Importance Of Modeling:

There are many elements that contribute to successful software. Software organization one common thread is the use of modeling.

Modeling is proven and well accepted engineering technique. We build architectural models of houses to help their users visualize the final product. We also build the models in the fields of sociology, economics, business management, automobiles and new electrical devices.

"**Model is a simplification of Reality**:"

A Model provides the blue prints of a system. Model may encompass detailed plans. Every system may described from different aspects using different models, and each model is therefore a semantically closed abstraction of the system.

A model may be structural, emphasizing the organization of the system, (or) it may be behavioral, emphasizing the dynamics of the system.

"We build models so that we can better understand the system we are developing"

We achieve 4 aims through the modeling, when are

1. Models help us to visualize the system
2. Models permit us to specify the structure (or) behavior at a system
3. Model gives us a template that guides us in constructing the system.
4. Model document the decision we have made.

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

Through modeling, we narrow the problem we are studying by focusing on only one aspect at a time. This is essentially the approach of "divide-and-conquer" , Attack a hard problem by dividing it into a series of smaller problems that you can solve.

Principles of modeling:

There are 4 basic principles of modeling.

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

In other words, choose your models well. The right models will brilliantly illuminate the development problems. The wrong models will mislead you.

In software the models you choose can greatly affect your world view. If you build a system through the eyes of a data base developer, you will likely focus on entity relationship models

If you build a system through the eyes of object oriented developer, you will focus on architecture, is centered on the classes and patterns of their interaction.

ii) "Every model may be expressed at different levels of precision."

In any case the best kinds of models are those that let you choose your degree of detail, depending on "WHO" is doing the viewing and why they need to view it.
An analyst "(or) end user will want to focus on issues of what. A developer will want to focus on issues of" HOW".

iii)" The best models are connected to reality".

It is best to have models that have a clear connection to reality and where connection is weak, to know exactly how those models are divorced from the real world, all models simplify reality.

iv) "No single model is sufficient. Every non trivial system is best approached through a small set of nearly independent models."

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans. The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. To understand the architecture of object oriented software systems, you need several complementary and inter locking views.

Object oriented modeling:

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

The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure (or) function. This view leads developers to focus on issues to control and the decomposition of algorithms into smaller ones.
As requirements change and the system grows, system built with an algorithmic focus turn out to be very hard to maintain.

The contemporary views of software development take and object oriented perspective. In this approach, the main building block of software system is the object (or) class, an object is a thing. A class is a description of a set of common objects. Every object has identity, state, and behavior for example, in the data base you will found concrete objects, such as tables representing entities from problem domain.

Unified modeling language (UML):

The UML is a language for visualizing. Specifying, constructing and documenting the artifacts of a software intensive system.

A conceptual model of the UML:

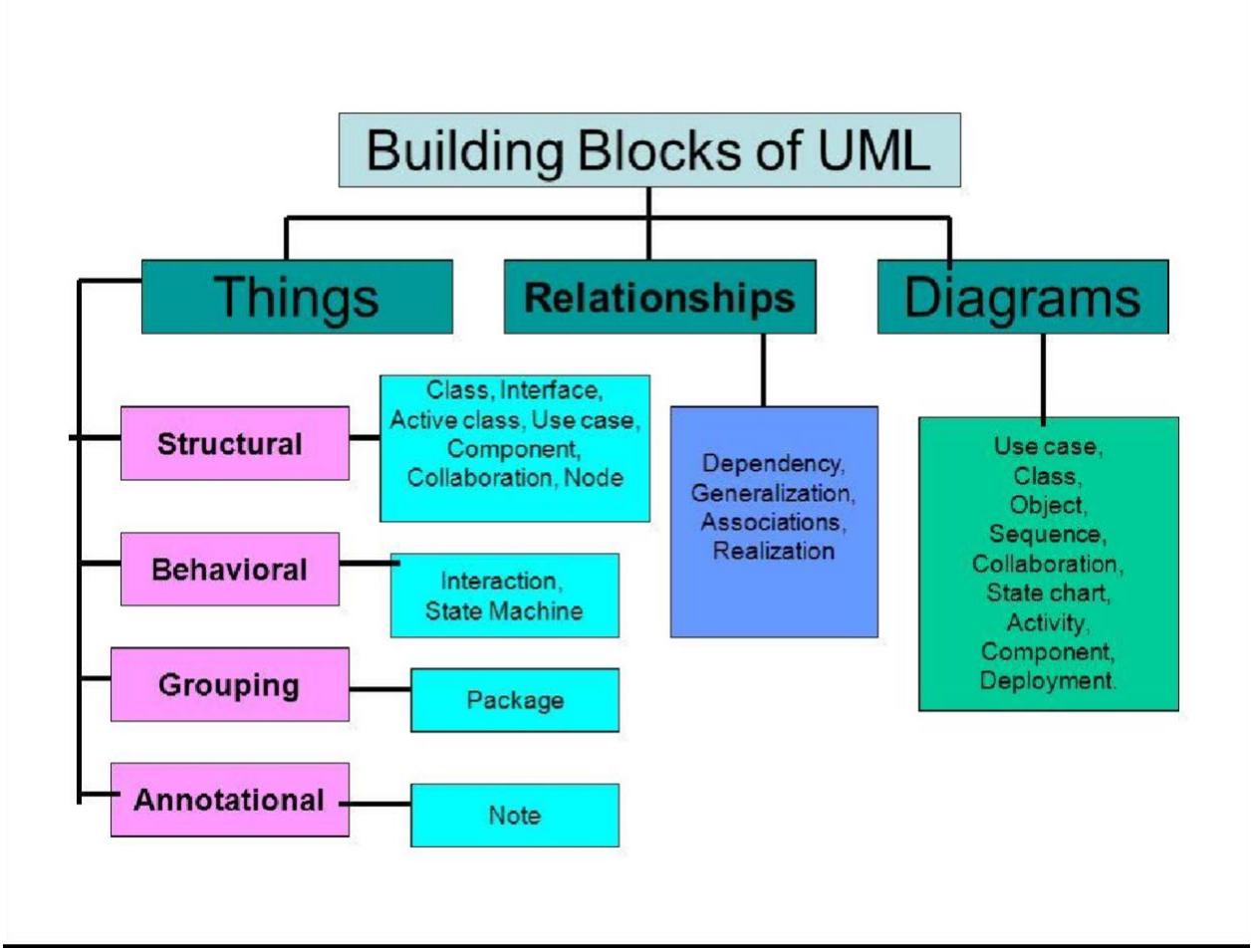
To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's **basic building blocks**, the **rules** and some **common mechanisms**

Building blocks of the UML:

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

‘Things’ are the abstractors that are first class citizens in models.
‘Relationships’ tie these things together.

‘Diagrams’ group intersecting collections of things.



Things:

There are 4 kinds of things in the UML
1. Structural things
2. Behavioral things
3. Grouping things
4. An notational things

1. Structural things:

Structural things are nouns of UML models these are mostly static parts of a model. There are 7 kinds of structural things.

i) Class:

Class is a description of a set of objects that share the same attributes, operations, relationships and semantics. It implements one (or) more interfaces. Graphically a class is rendered as a rectangle, usually including its name, attributes and operations.

Window
Origin size
Open() Close () Move() Display ()

Person
Name Phonenum Email
Purchase( )

ii) Interface:

An interface is a collection of operations that specify a service of a class (or) component. An interface might represent the complete behavior of a class (or) component (or)only a part of behavior. Graphically it is rendered as a circle together with its name.

iii) Collaboration:

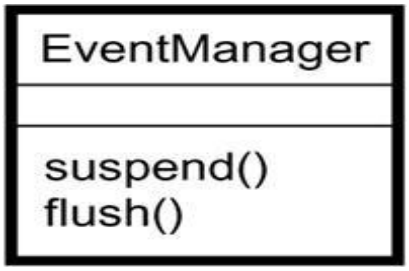
It defines an interaction and is a society of roles and other element that work together to provide some cooperative behavior. A given class might participate in several collaborations. Graphically a collaboration is rendered as an ellipse with solid line including its name.

iv) Use case:

Use case is a description at set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is realized by collaboration. Graphically a usecase is rendered as an ellipse with solid line, including its name.

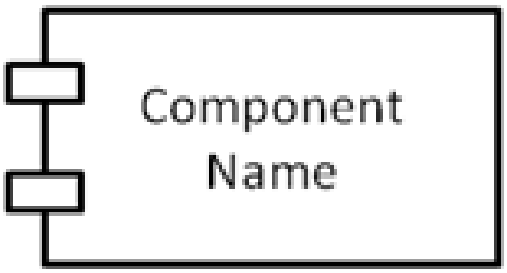
v) Active class:

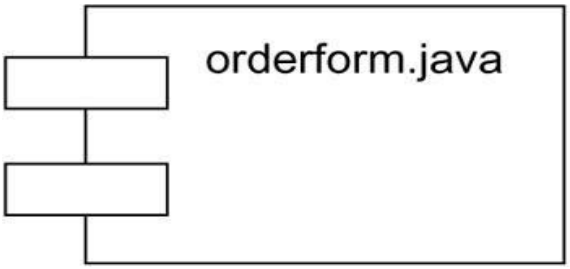
An active class is a class whose objects own one (or) more processes (or) threads and therefore can initiate control activity. Graphically it is rendered just like a class but with heavy lines, including its name, attributes and operations.



vi) Component:

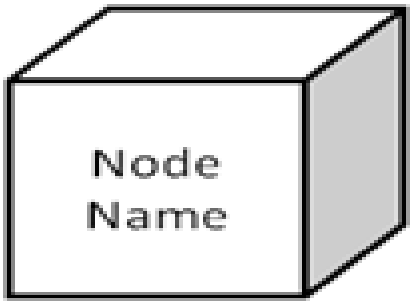
A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components Java Beans, as well as source code files.Graphically it is represented as a rectangle with tabs including its name.

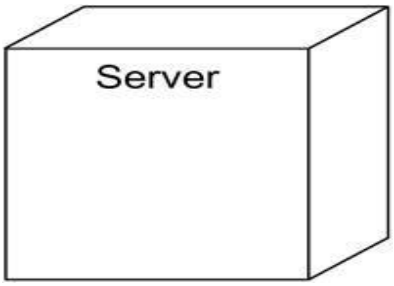




vii) Node:

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





2. Behavioral things:

Behavioral things are the dynamic parts of UML models representing over time and space. There are 2 primary kinds of behavioral things.

i) Interaction:

Interaction is a behavior that comprises a set of massages exchanged among set of objects with in a particular context to accomplish a specific purpose. Graphically a message is rendered as a directed line almost always including the name of its operations.

ii) State machine:

A state machine is a behavior that specifies the sequences of states an object (or) an interaction goes through during its life time in response to events together with its response to those events. Graphically a state is rendered as a rounded rectangle, including its name

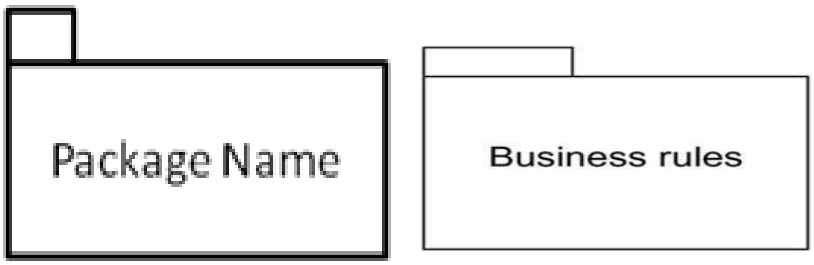
3.Grouping things:

Graphing things are the organizational parts of UML models. The primary kind of graphing thing is packages.

i)package:

It is a general purpose mechanism for organizing elements into groups. Structural things, behavioral thing and even other things may be placed in a package.

Graphically a package is rendered as a tabbed folder including its name and sometimesit's' contents.

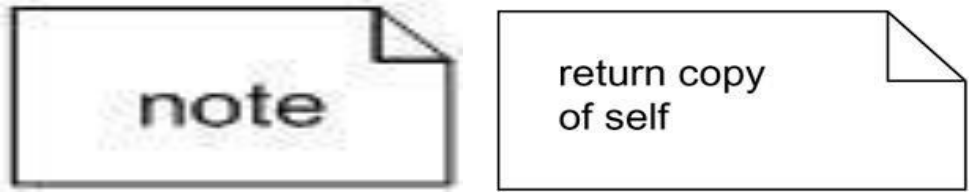


4.An notational things:

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe.

i)NOTE:

A Note is simply a symbol for rendering constraints and comments attached ot an element (or) a collection of elements.

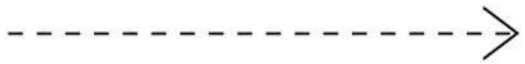


Relationships:

There are 4 kinds of relationships in the UML
i) Dependency
ii) Association
iii) Generalization
iv) Realization

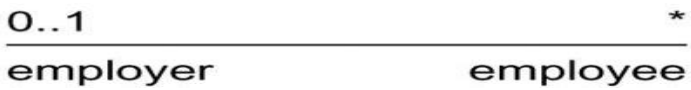
i) Dependency:

It is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing. Graphically it is rendered as a dashed line possibly directed and including a label.



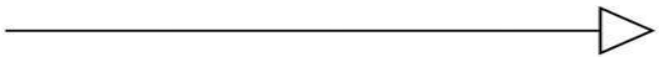
ii) Association:

It is a structure relationship that describes a set of links, a link being a connection among objects. Graphically an association is rendered as a solid line, possibly directed, including its, name, and containing multiplicity and role name.



iii) Generalization:

it is a specialization / generalization relationship in which objects of the specialized element (child) are substitutable for objects of the generalized element (parent) the child shares the structure and the behavior of the parent. Graphically it is rendered as solid line with narrow arrow head painting to parent.



IV**) Realization:**

It is a semantic relationship between classifiers, where in one classifier serapes a contract that another classifier guarantees to carryout it is used in between interfaces and the class (or) graphically it is rendered as a cross between a generalization and dependency.

Diagrams:

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and Ares (relations ships). UML include 9 such diagrams.

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

1. class diagram:

It shows a set of classes, interfaces and collaborations and their relationships. Class diagram address the static design view of a system.

2.Object diagram:

It shows a set of objects and their relationships .this diagram address the static design view of a system.

3.Use case diagram:

It shows a set of use cases and actors and their relationships. It address the static use case view of a system.

4. Sequence diagram:

It is an interaction diagram that emphasizes the time ordering of messages. It consist set of objects and relationships. It addresses the dynamic view of the system.

5.Collaboration diagram:

It is also an interaction diagram that emphasizes the structural organization of the objects that send and receive in messages. It addresses the dynamic view of the system.

6.State chart diagram:

It shows a state machine consisting of states, transitions, events, and activities. It address the dynamic view of a system

7.Activity diagram:

It is a special kind of state chart diagram that shows the flow from activity to activity within a system. It addresses the dynamic view of a system.

8.component diagram:

It shows the organizations and dependencies among a set of components. It addresses the static implementation view of a system.

9.Deployment Diagram:

It shows the configuration of run time processing hoes and the component that live on them. It addresses the static deployment view of architecture.

Rules of the UML:

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 relatedmodels.
The UML has 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

Common mechanisms in the UML:

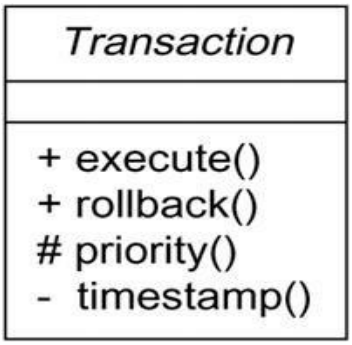
There 4 common mechanisms in the UML
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

1.Specifications:

UML is more than just a graphical language. Rather specification that provides a textual statements of the syntax and semantics of that building block.
For ex: behind a class icon is a specification that provides the full set of attributes, operations,

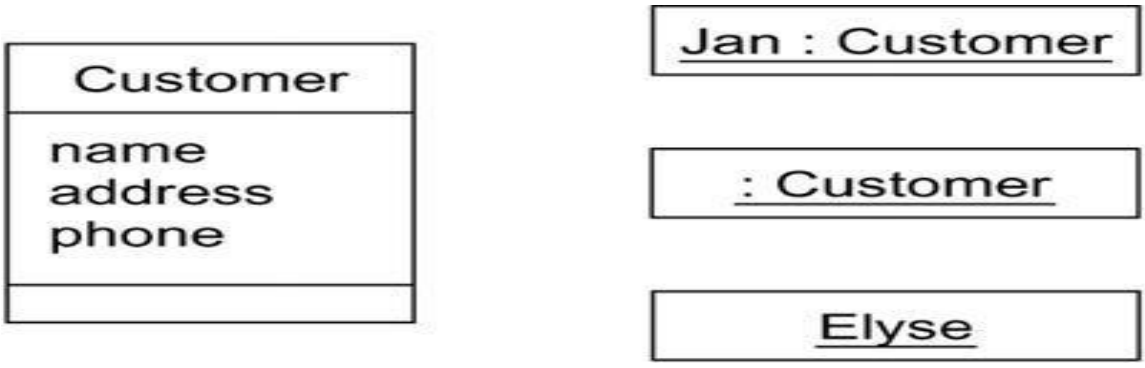
2.Adornments:

A class's specification may include other details such as whether it is abstract (or) the visibility of its attributes and operations. A many of these details can be rendered as graphical (or) textual adornments to the class basic rectangular notation
For ex: the bellow fig shows a class, adorned to indicated that is an abstract class with two public one protected and one private operation.

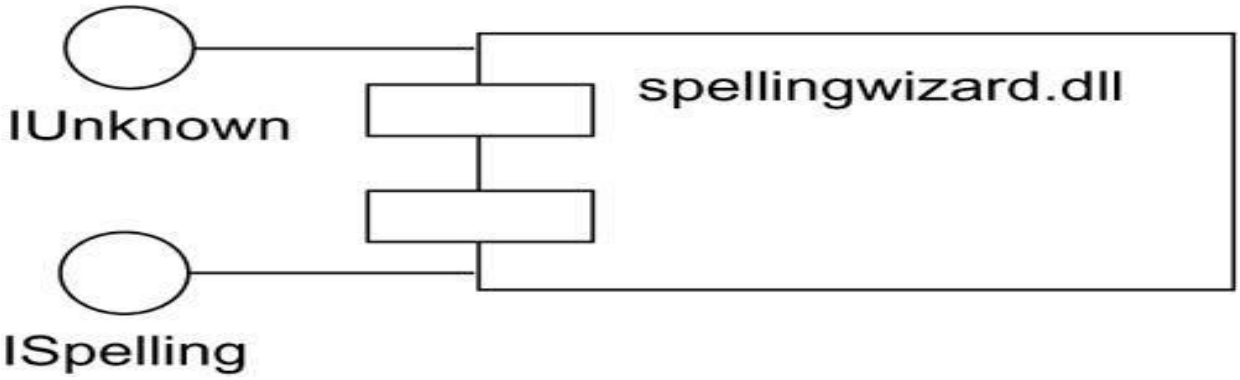


3. **Common divisions:**

In modelingObject oriented systems; the world often gets divided in at least a couple of 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 above figure, there is one class named customer together with three objects
Jan(which is marked explicitly as being a Customer object), :Customer(an anonymous Customerobject).
Second, there is a separation of interface and implementation.



In above fig, there is a one component named ‘spelling wizard.dll’ that implements two interfaces I unknown and I spelling.

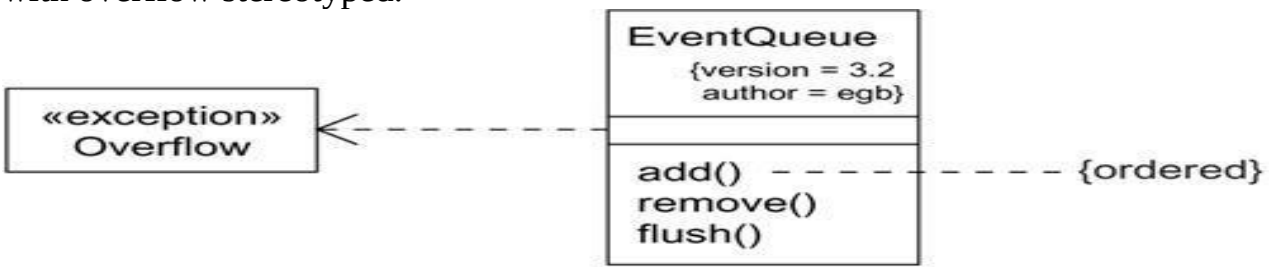
4.Extensibility mechanisms:

The UML'S extensibility mechanisms include

- Stereotypes
- Tagged values
- Constraints

Stereotypes:

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, In java (or) c++ the exceptions are modeled as special classes. It is marked with overflow stereotyped.



Tagged value:

A tagged value extends the properties of a UML building block, allowing you create new information in that element's specification. For example. The class 'Event queue' is extended by marking its version and author explicitly.

Constraint:

It extends the semantics of UML building blocks, allowing you to add new rules (or) modify for example:
In 'Event Queue' class all addition are done in 'order'

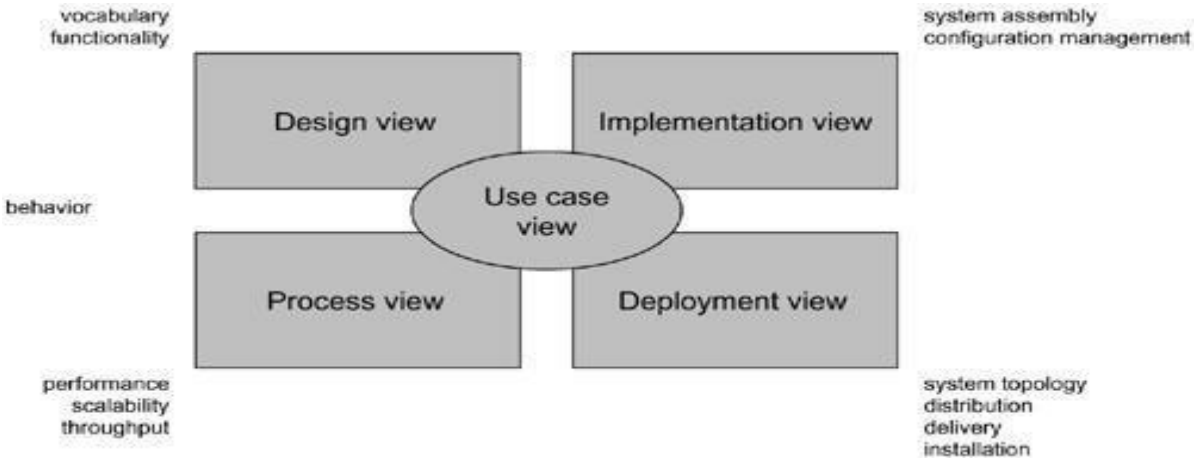
Architecture:

Architecture is the set of significant decision about

- The organization of a software system

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

The bellow fig illustrates, the architecture of a software intensive system can best be described by five interlocking views.



The use case view Of a system encompasses the use cases that describes the behavior of the system as seen by its end users, analysts and testers. The static aspects of this view are captured in use case diagrams, the dynamic aspects of this view are captured in interaction diagrams.

The design view Of a system encompasses the classes, interfaces and collaborations that form the vocabulary of the problem and its solution. Which supports the functional requirements of a system. The static and dynamic aspects of this view is captured in class, object and interaction diagrams respectively.

The process view Of the system encompasses the threads and processes that form the system's concurrency and synchronization mechanism. This view addresses the performance, scalability and throughput of the system.
The static and dynamic aspects of this view is captured in class, object and interaction diagrams respectively.

The implementation view Of a system encompasses the components and files that are used to assemble and releases the physical system. This view addresses the

configuration management of the system releases. The static aspects of this view are captures in component diagrams, the dynamic aspects of this view captured in interaction diagrams.

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

Software development life cycle:

In UML, you should consider a process that is

- Use case driven
- Architecture centric
- Interactive and incremental

Use case driven means that use cases are used as a primary artifact for establishingthe desired behavior of the system, for verifying and validating the systems' architecture.

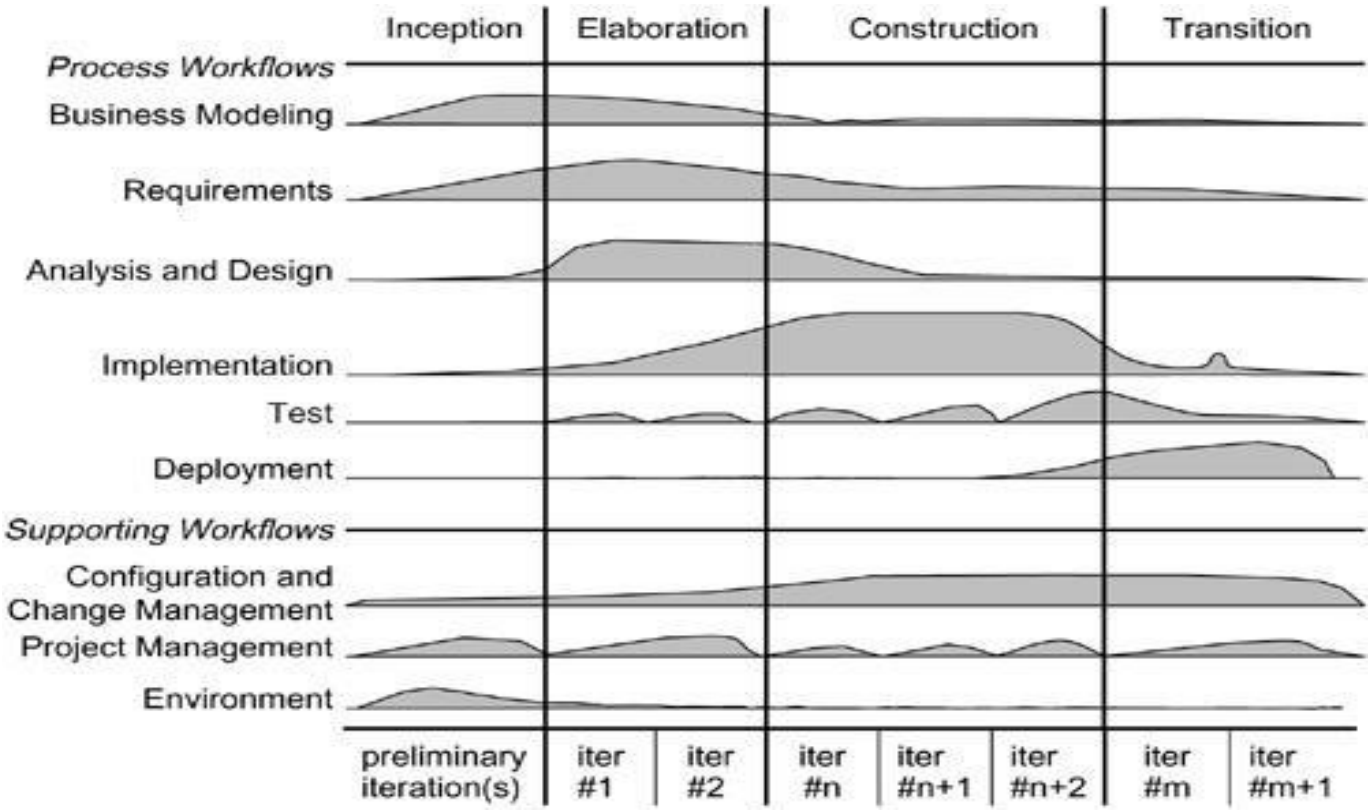
Architecture- centric means that a system's architecture is used as a primary and evolving the system under development.

An **iterative process** is one that involves managing a stream of executable releases. An incremental process is one that involves the continuous integration of the systems architecture to produce these releases.
This use case driven, architecture – centric and iterative / incremental process can be broken into phases.

A phase is the span of time between two major milestones of the process, when a well-defined set of objectives are met, artifacts are completed and decisions are made.

The bellow fig shows, 4 phases in s/w development life cycle.

- Inception
- Elaboration
- Construction
- Transition



Inception is the first phase of the process the goals of inception phase are:

- Establish business case for the project
- Establish project scope and boundary conditions
- Outline one (or) more candidate architecture.
- Identify risks.
- Prepare a project schedule and cost estimate.
- Feasibility.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized and base lined.

Construction is the third phase During the construction phase, all remaining components and application features are developed and integrated into the product, and all features are thoroughly tested.
The outcome of the construction phase is a product ready to put in hands of its end-users. At minimum, it consists of:

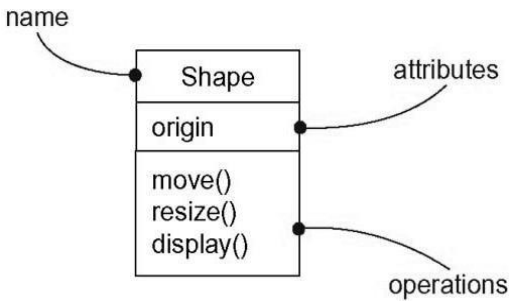
- The software product integrated on the adequate platforms.
- The user manuals.
- A description of the current release.

Transition is the fourth phase of the process, when the software is turned in to the hands of the user community. For even during this phase, the system is continuously improved, bugs are dictated and features that did not make an earlier release are added.

chapter– II Structural Modeling

Classes

- 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.



Graphical Representation of Class in UML

Terms and Concepts

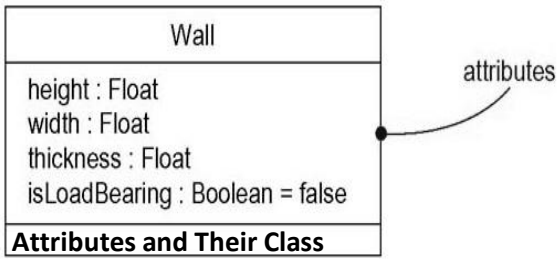
Names
Every class must have a name that distinguishes it from other classes. A name is a textual string that name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives.

Simple Name

Path Name

Attributes

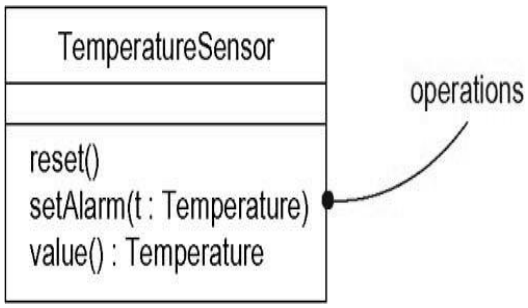
- An attribute is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of thing you are modeling that is shared by all objects of that class
- You can further specify an attribute by stating its class and possibly a default initial value



Attributes and Their Class

Operations

- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- You can specify an operation by stating its signature, covering the name, visibility, type, and default value of all parameters and a return type

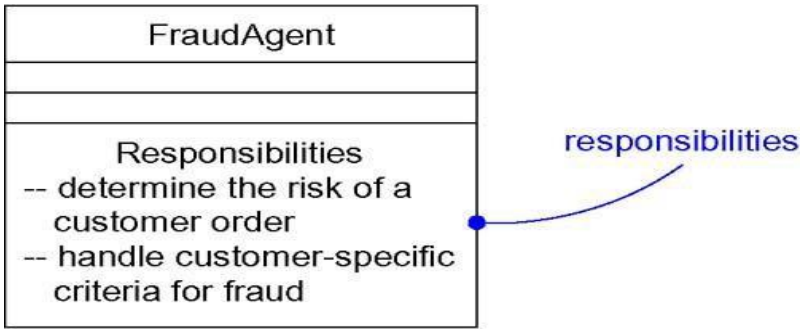


Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. You can elide a class, meaning that you can choose to show only some
or none of a class's attributes and operations. An empty compartment doesn't necessarily mean there are no attributes or operations, just that you didn't choose to show them.

Responsibilities

- A Responsibility is a contract or an obligation of a class
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.



- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon

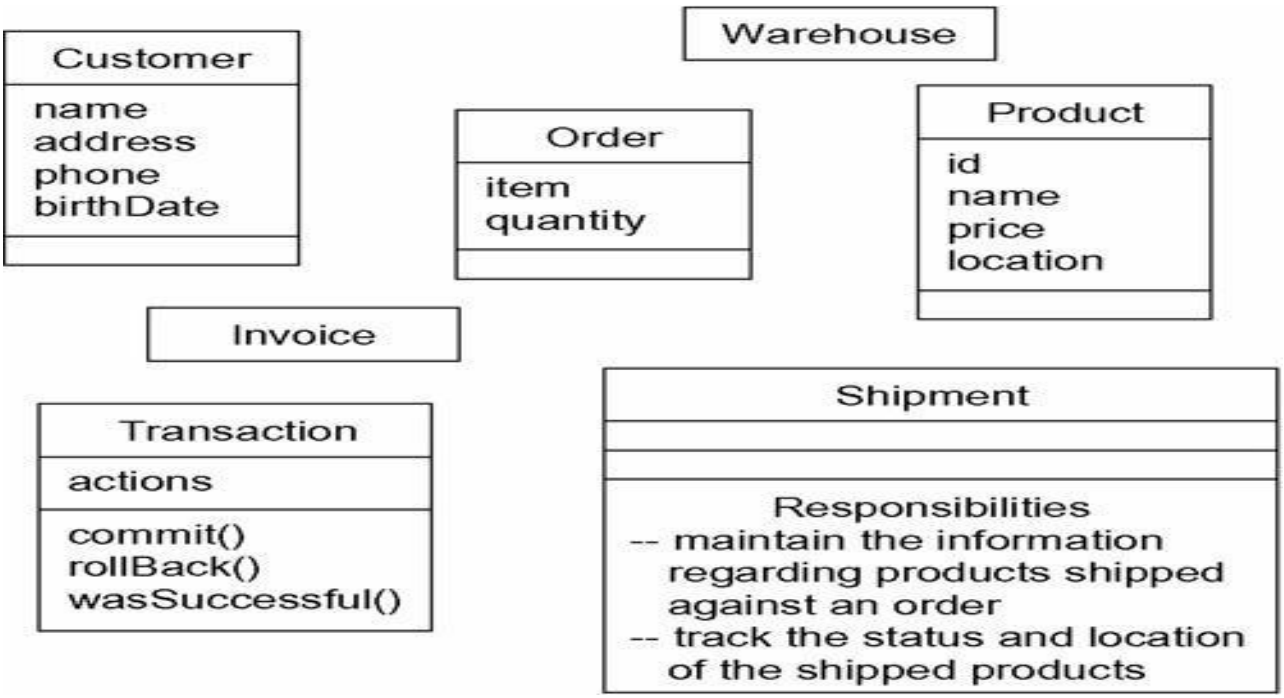
Common Modeling Techniques

Modeling 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.

- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

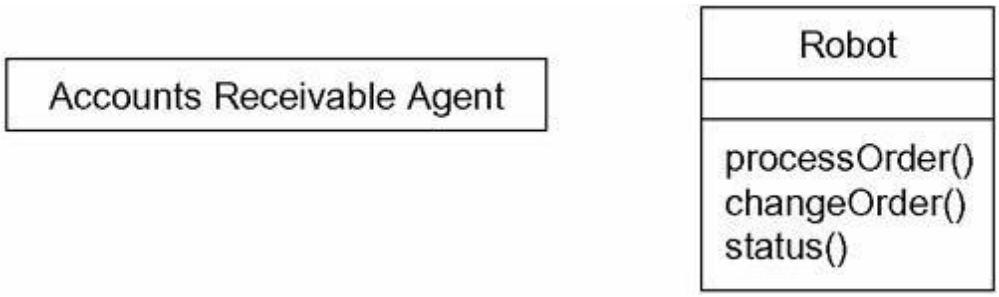


Modeling the Distribution of Responsibilities in a System

- 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.

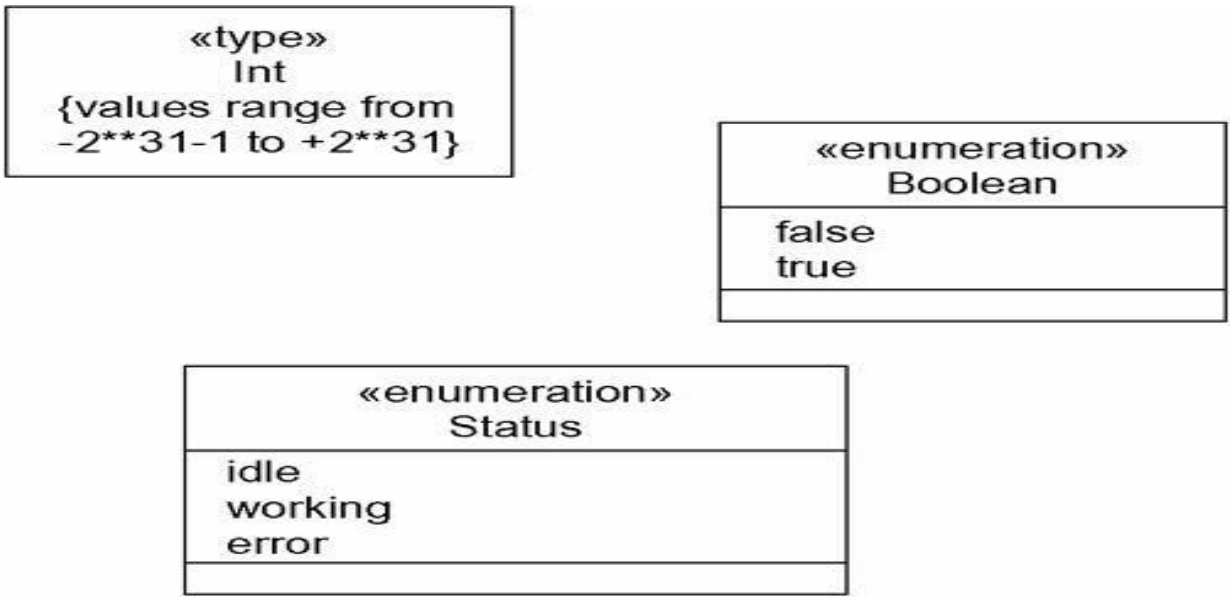
Modeling Nonsoftware Things

- 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

- Model the thing you are abstracting as a type 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.



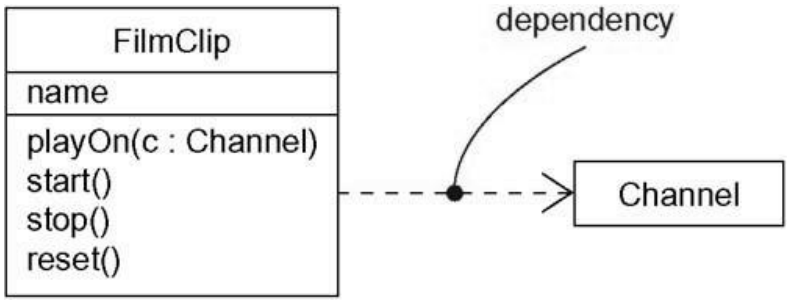
Relationships

In object-oriented modeling, there are three kinds of relationships that are most important:
Dependency
Generalization
Association

Dependency

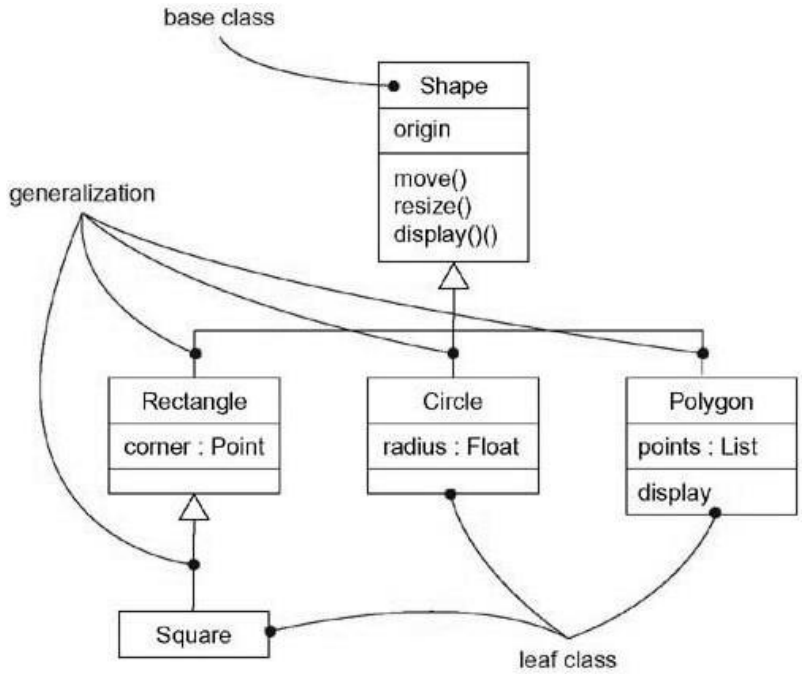
A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse. Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.

Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation. if the used class changes, the operation of the other class may be affected, as well, because the used class may now present a different interface or behaviour.



Generalization

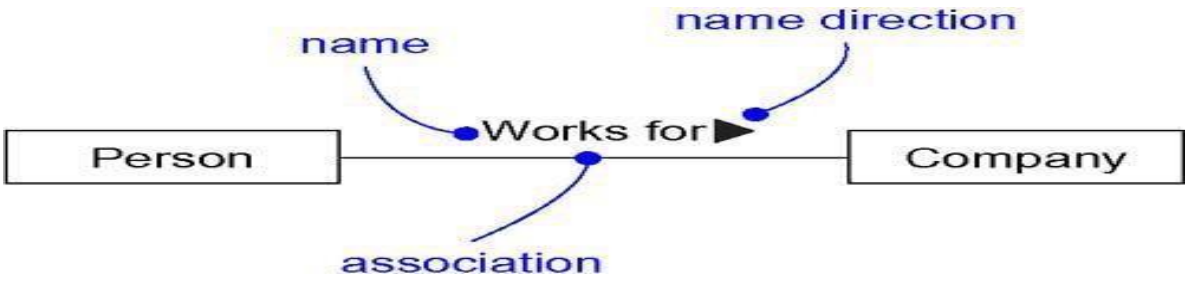
A generalization is a relationship between a general thing (called the super class or parent) and a more specific kind of that thing (called the subclass or child). Generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Generalization is sometimes called an **"is-a-kind-of"** relationship.
Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent.



Association

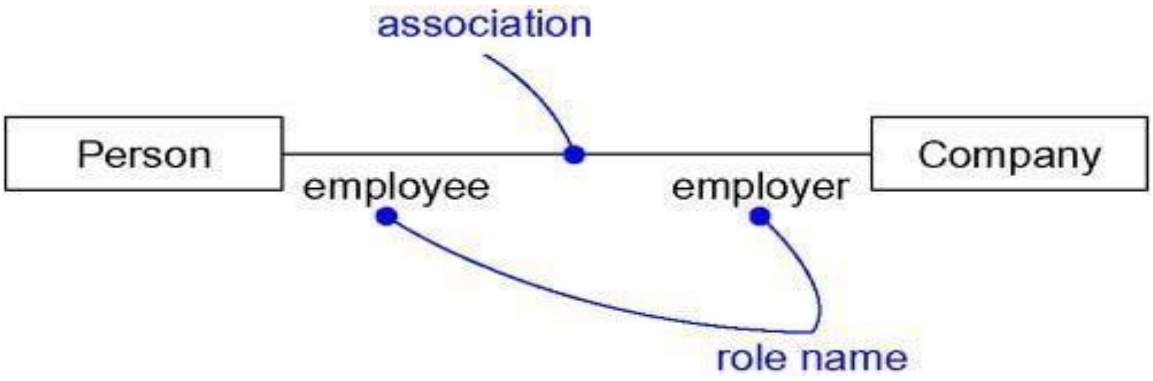
An association is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa.
Graphically, an association is rendered as a solid line connecting the same or different classes.

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



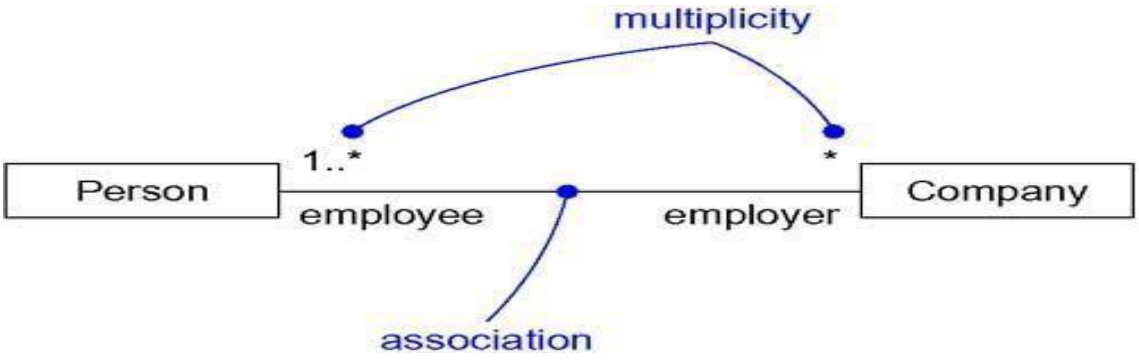
Role

When a class participates in an association, it has a specific role that it plays in that relationship; The same class can play the same or different roles in other associations.
An instance of an association is called a link.



Multiplicity

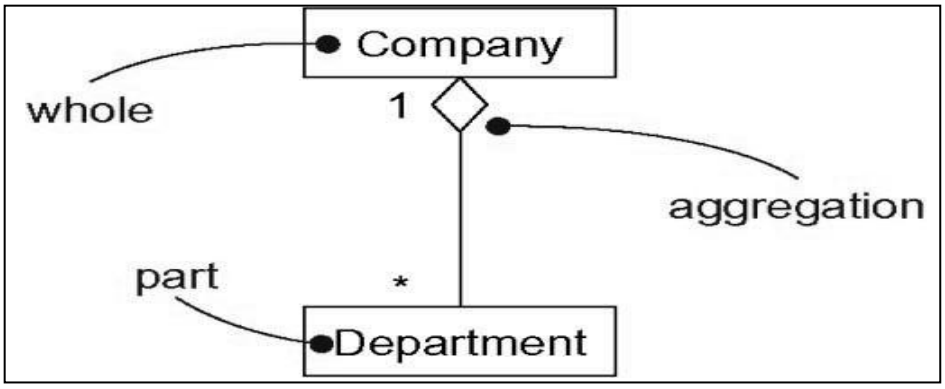
In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association, This "**how many**" is called the multiplicity of an association's role.
You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).



Aggregation

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

Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end

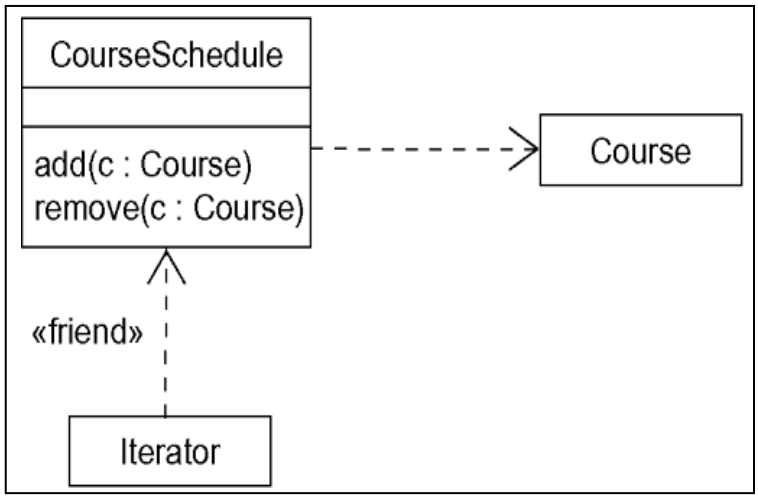


Common Modeling Techniques

Modeling Simple Dependencies

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

  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

- 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.

- 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

- 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 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 it helps 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

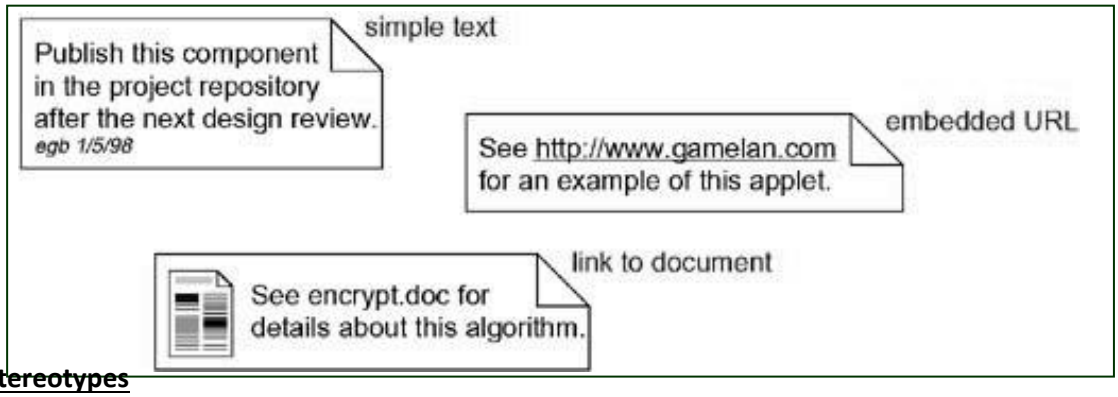
Common Mechanisms

Note

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.

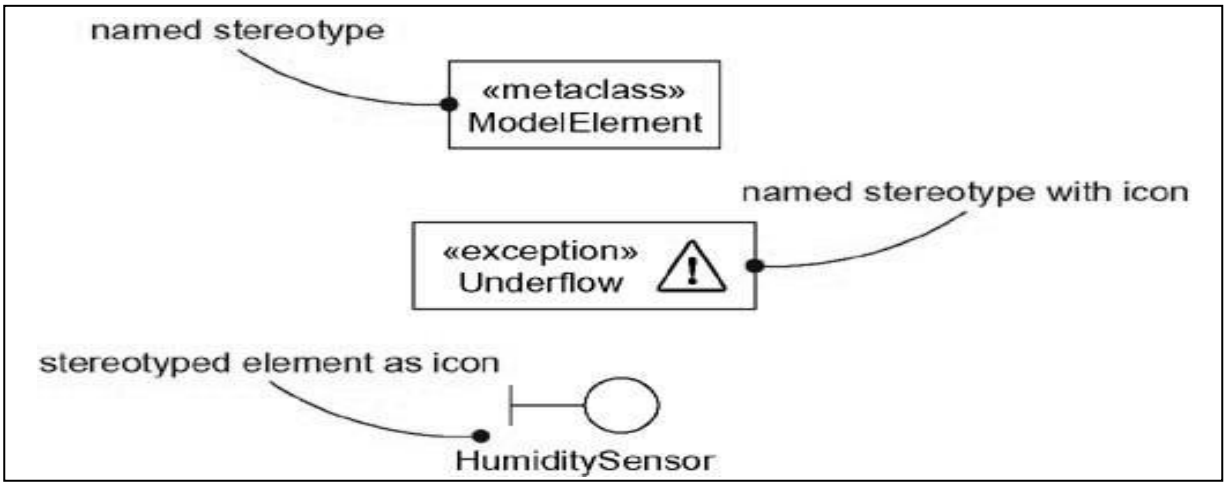
A note may contain any combination of text or graphics, you can put a live URL inside a note, or even link to or embed another document.



Stereotypes

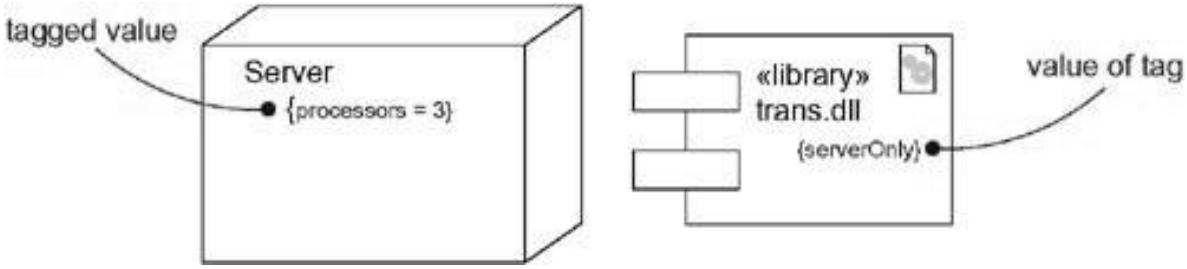
A stereotype is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem.

Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element



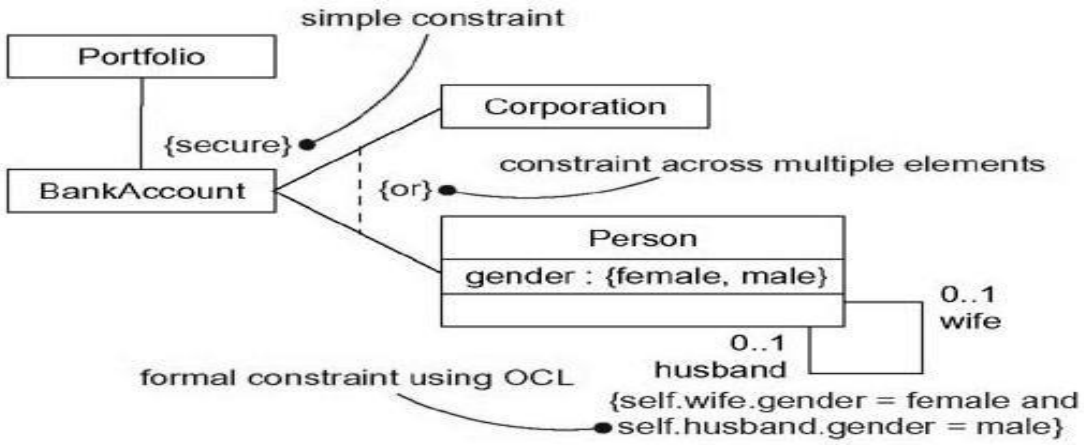
Tagged Values

- Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- A tagged value is an extension of the properties of a UML element, allowing you to create new information in that element's specification.
- Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.



Constraints

- A constraint specifies conditions that must be held true for the model to be well-formed.
- A constraint is rendered as a string enclosed by brackets and placed near the associated element.
- 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.



Common Modeling Techniques

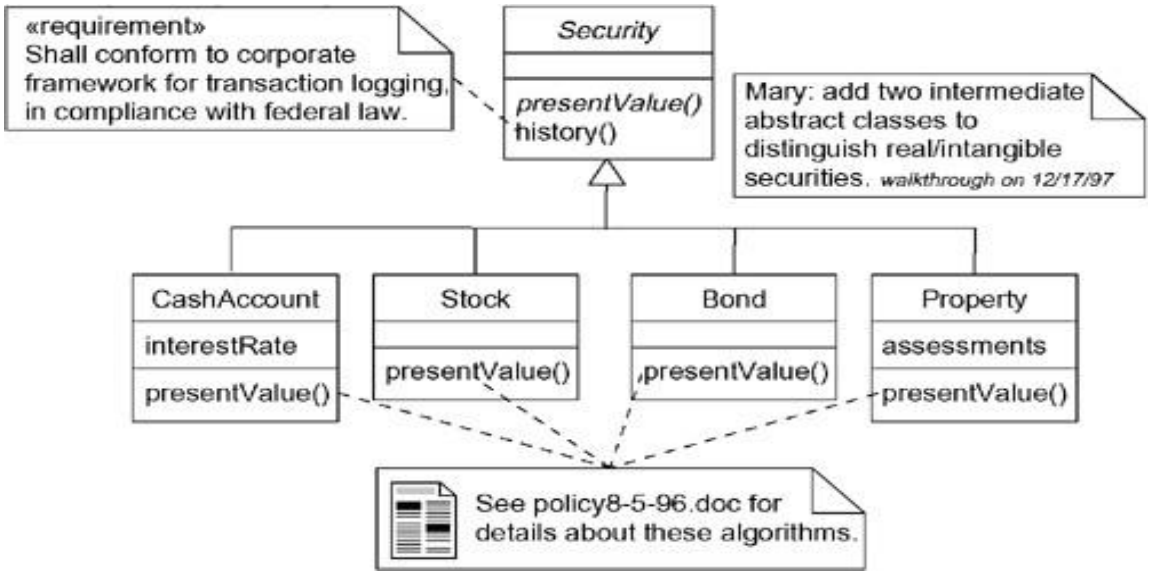
Modeling Comments

- 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.



Modeling New Building Blocks

- Make sure there's not already a way to express what you want by using basic UML. If you have a common modelling problem, chances are there's already some standard stereotype that will do what you want.

- If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model and define a new stereotype for that thing.

- Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.

- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype

Modeling New Properties

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.

- If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype.

Modeling New Semantics

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.

- If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers.

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

Diagrams

System

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

SubSystem

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

Model

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

View

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

Diagram

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- A diagram is just a graphical projection into the elements that make up a system.

Structural Diagrams

- The UML's four structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.
- The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

  - Class diagram : Classes, interfaces, and collaborations
  - Object diagram : Objects
  - Component diagram : Components
  - Deployment diagram : Nodes

Class Diagram

- We use class diagrams to illustrate the static design view of a system.
- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams that include active classes are used to address the static process view of a system.

Object Diagram

- Object diagrams address the static design view or static process view of a system
- An object diagram shows a set of objects and their relationships.
- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

- We use component diagrams to illustrate the static implementation view of a system.
- A component diagram shows a set of components and their relationships.
- Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- We use deployment diagrams to illustrate the static deployment view of an architecture.
- A deployment diagram shows a set of nodes and their relationships.
- Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

- The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

Use case diagram : Organizes the behaviors of the system
Sequence diagram : Focused on the time ordering of messages
Collaboration diagram : Focused on the structural organization of objects that send and

receive messages

Statechart diagram : Focused on the changing state of a system driven by events

Activity diagram : Focused on the flow of control from activity to activity

Use Case Diagram

- A use case diagram shows a set of use cases and actors and their relationships.
- We apply use case diagrams to illustrate the static use case view of a system.
- Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Sequence Diagram

- We use sequence diagrams to illustrate the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.

Collaboration Diagram

- We use collaboration diagrams to illustrate the dynamic view of a system.
- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.

***** Sequence and collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.

Statechart Diagram

- We use statechart diagrams to illustrate the dynamic view of a system.
- They are especially important in modeling the behavior of an interface, class, or collaboration.
- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.

Activity Diagram

- We use activity diagrams to illustrate the dynamic view of a system.
- Activity diagrams are especially important in modeling the function of a system.
- Activity diagrams emphasize the flow of control among objects.
- An activity diagram shows the flow from activity to activity within a system.
- An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.

Common Modeling Techniques

Modeling Different Views of a System

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project.

- For each of these views, decide which artifacts you need to create to capture the essential details of that view.

- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.

- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

Modeling Different Levels of Abstraction

- Consider the needs of your readers, and start with a given model.

- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction which means that they'll need to reveal a lot of detail. If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction which means that they'll hide a lot of detail.

- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

  Building blocks and relationships:
  - Hide those that are not relevant to the intent of your diagram or the needs of your reader.

  Adornments:
  - Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

  Flow:
  - In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.

  Stereotypes:
  - In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

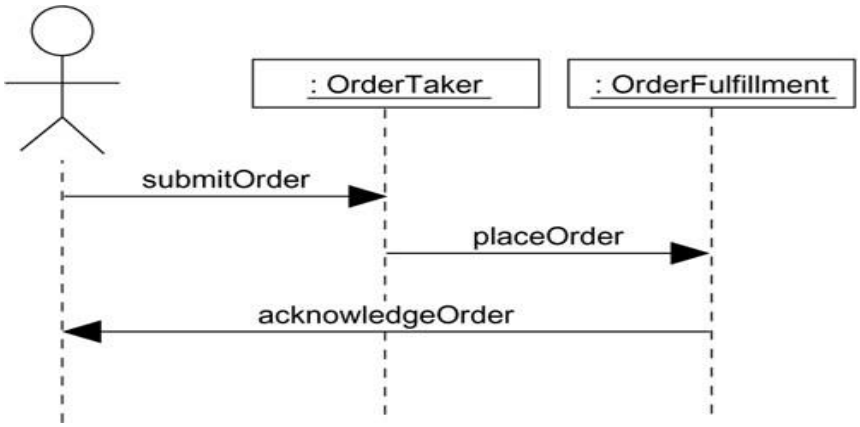
To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.

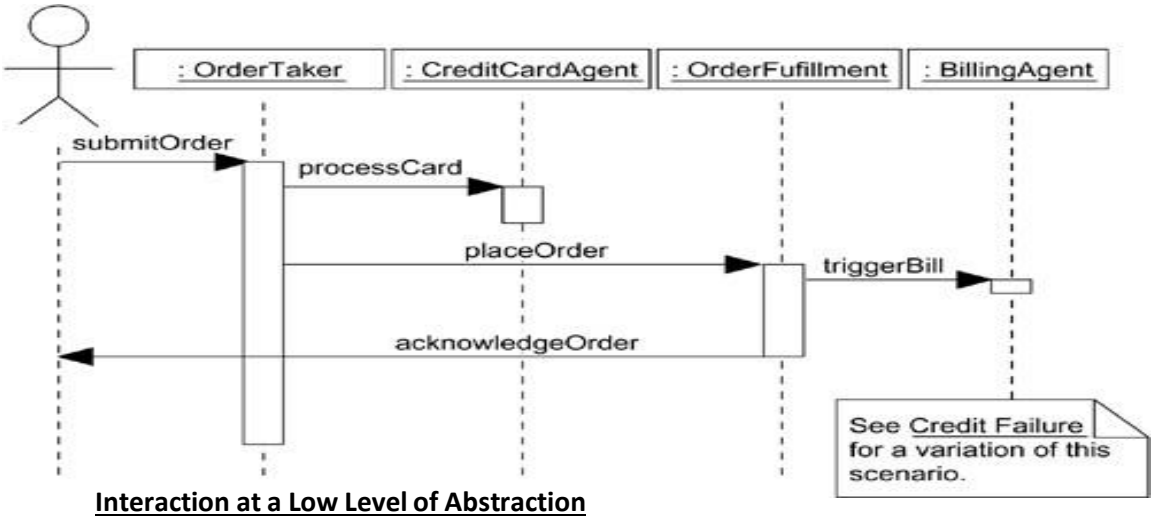
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.

- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

Use cases and their realization:
Use cases in a use case model will trace to collaborations in a design model.
Collaborations and their realization:
Collaborations will trace to a society of classes that work together to carry out the collaboration.
Components and their design:
Components in an implementation model will trace to the elements in a design model.
Nodes and their components:
Nodes in a deployment model will trace to components in an implementation model.



Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

* Both of these diagrams work against the same model, but at different levels of detail.

Modeling Complex Views

- First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.

- If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher level collaborations, then render only those packages or collaborations in your diagram.

- If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.

- If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

_____ ********** _____

UNIT-3

Clases and Object Diagrams

Classifiers

When you model, you'll discover abstractions that represent things in the real world and things in your solution.
The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however. The UML provides a number of other kinds of classifiers to help you model.

Interface - A collection of operations that are used to specify a service of a class or a Component.

Datatype- A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean).

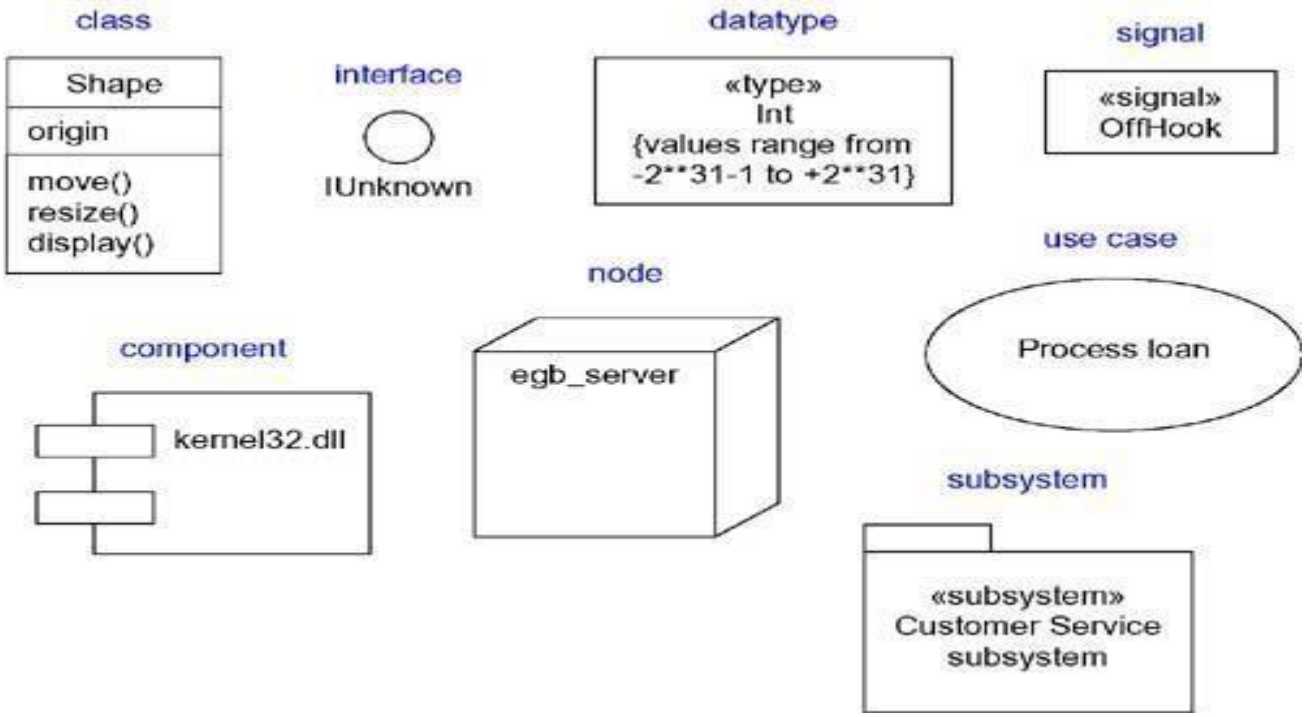
Signal- The specification of an asynchronous stimulus communicated between instances.

Component- A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

Node- A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability.

Use case- A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor.

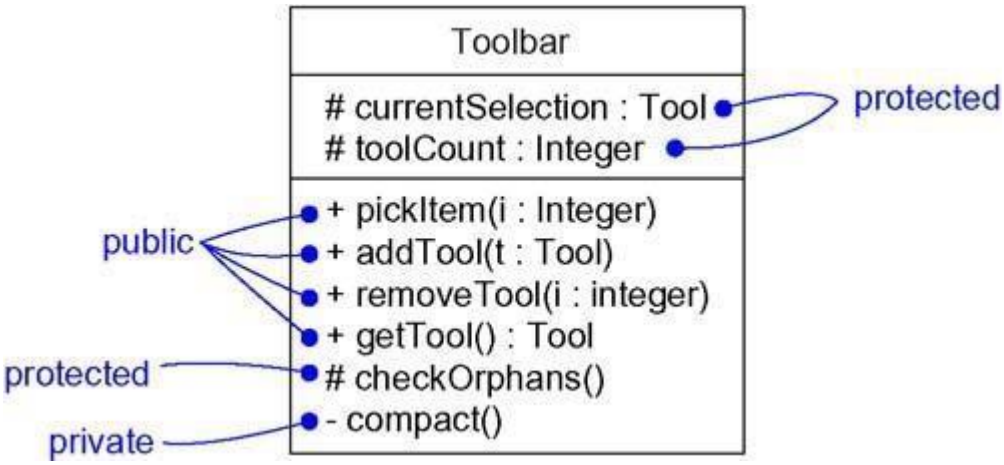
Subsystem- A grouping of elements of which some constitute a specification of the behaviour offered by the other contained elements.



Visibility

One of the most important details you can specify for a classifier's attributes and operations isits visibility. The visibility of a feature specifies whether it can be used by other classifiers.In the UML, you can specify any of three levels of visibility.

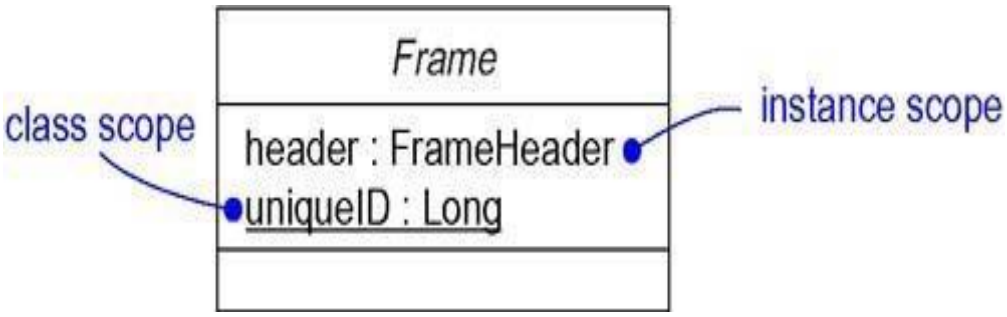
1. public- Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +.
2. protected - Any descendant of the classifier can use the feature; specified by prependingthe symbol #
3. private- Only the classifier itself can use the feature; specified by prepending thesymbol -



Scope

Another important detail you can specify for a classifier's attributes and operations is its owner scope.In the UML, you can specify two kinds of owner scope.

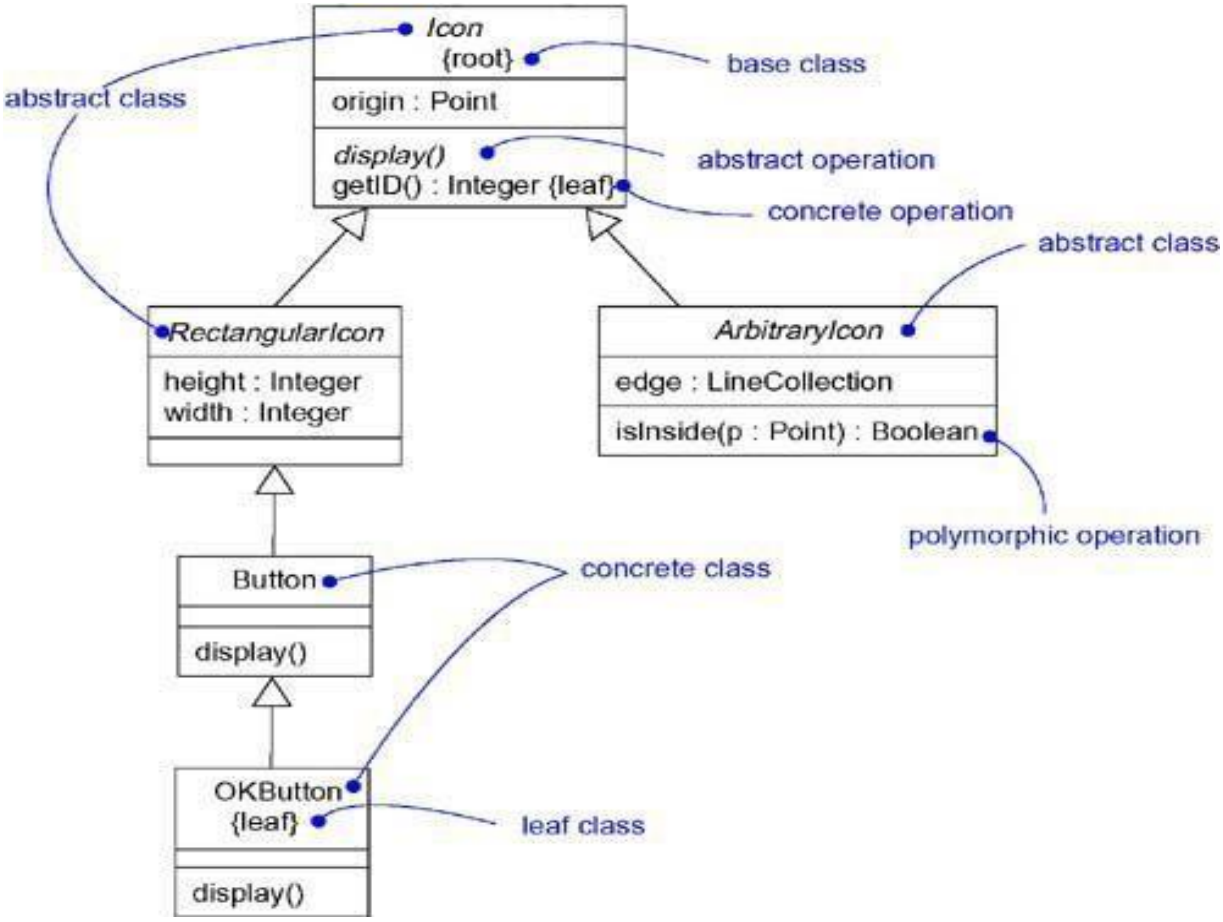
1. **instance-** Each instance of the classifier holds its own value for the feature.
2. **classifier**- There is just one value of the feature for all instances of the classifier.



Abstract, Root, Leaf, and Polymorphic Elements:

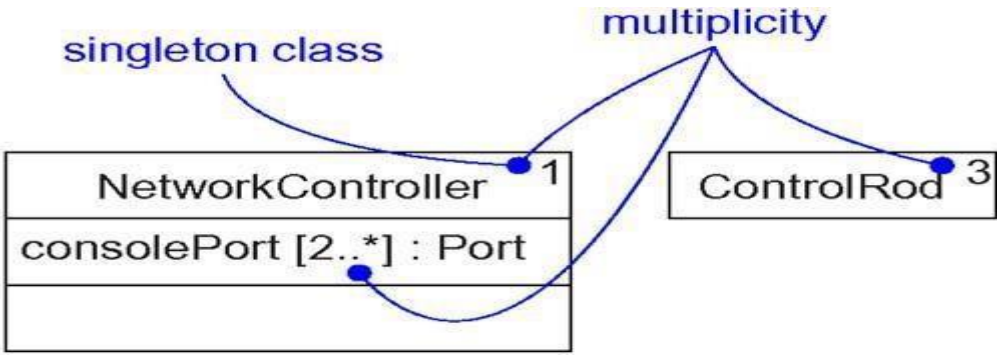
In hierarchies, it's common to specify that certain classes are abstract, meaning that they may not have any direct instances. In the UML, you specify that a class is abstract by writing its name in italics. For example, bellow fig shows, Icon, RectangularIcon, and ArbitraryIcon
are all abstract classes. By contrast, a concrete class (such as Button and OKButton) is onethat may have direct instances.

you can also specify that a class may have no children. Such an element is called a leaf class and is specified in the UML by writing the property leaf below the class's name. A class mayhave no parents. Such an element is called a root class, and is specified in the UML by writing the property root below the class's name.
An operation is polymorphic, which means that, in a hierarchy of classes, you can specifyoperations with the same signature at different points in the hierarchy. The child classes override the behaviour of parent classes.



Multiplicity:

The number of instances a class may have is called its multiplicity. In the UML, you can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon.



Attributes

You can also specify the visibility, scope, and multiplicity of each attribute. There's stillmore. You can also specify the type, initial value, and changeability of each attribute.

In its full form, the syntax of an attribute in the UML is

[visibility] name [multiplicity] [: type] [= initial-value] [{property-string}]

For example,

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

There are three defined properties that you can use with attributes.

1. changeable- There are no restrictions on modifying the attribute's value.

2. addOnly- For attributes with a multiplicity greater than one, additional values may beadded, but once created, a value may not be removed or altered.

3. frozen- The attribute's value may not be changed after the object is initialized.

Operations:

You can also specify the visibility and scope of each operation. There's still more: You can also specify the parameters, return type, concurrency semantics, and other properties of eachoperation.
In its full form, the syntax of an operation in the UML is

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

For example,
+set(n : Name, s : String)- Visibility, Name and
parametersgetID( ) : Integer- Name and return type
restart( ) {guarded}- Name and property

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

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

Direction may be any of the following values:

In- An input parameter; may not be modified
Out- An output parameter; may be modified to communicate information to the callerInout- An input parameter; may be modified

There are four defined properties that you can use with operations.

1. isQuery- Execution of the operation leaves the state of the system unchanged. In otherwords, the operation is a pure function that has no side effects.

2.sequential- Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semanticsand integrity of the object cannot be guaranteed.

3. guarded- The semantics and integrity of the object is guaranteed in the presence of

multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invokedon the object, reducing this to sequential semantics.

4.concurrent- The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple callsfrom concurrent flows of control may occur simultaneously to one object onany concurrent operation, and all may proceed concurrently with correct semantics;

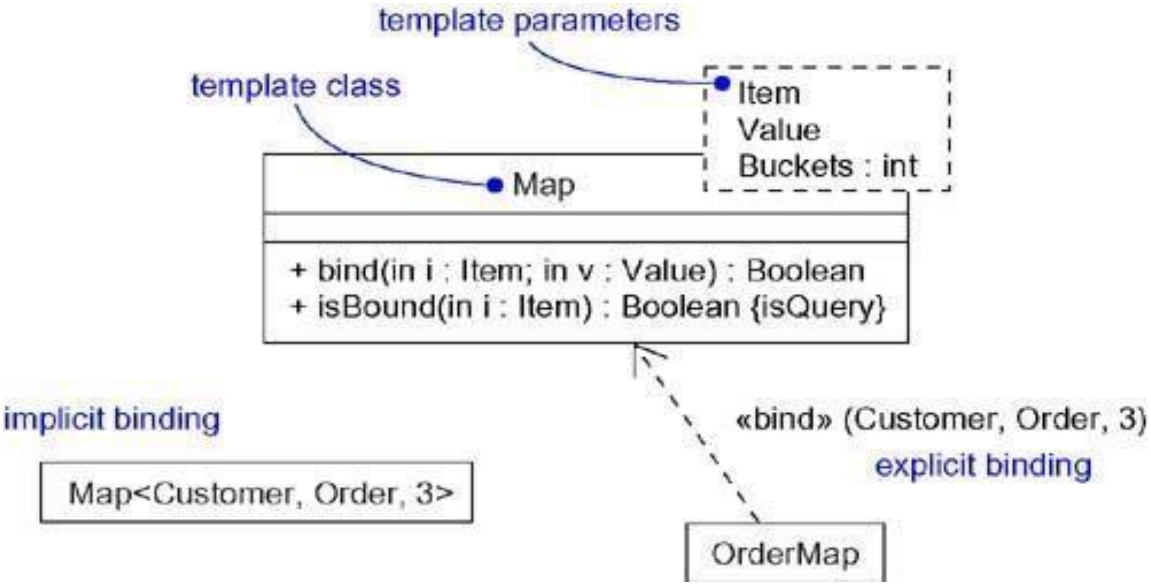
Template Classes:

A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes. A template includes slots for classes, objects, and values, and these slots serve as the template's parameters. You can't usea template directly; you have to instantiate it first. Instantiation involves binding these formaltemplate parameters to actual ones.

```
template<class Item, class Value, int Buckets>
class Map {
public:
virtual Boolean bind(const Item&, const
Value&);virtual Boolean isBound(const Item&)
const;
...
};
```

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

```
objects.m : Map<Customer, Order, 3>;
```



The UML defines four standard stereotypes that apply to classes.

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

Modeling the Semantics of a Class:

- Specify the responsibilities of the class. A responsibility is a contract or obligation of atype or class and is rendered in a note (stereotyped as responsibility) attached to the class, or in an extra compartment in the class icon.

- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as semantics) attached to the class.

- Specify the body of each method using structured text or a programming language,rendered in a note attached to the operation by a dependency relationship.

- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as precondition, postcondition, and invariant) attached to the operation or class by a dependency relationship.

- Specify a state machine for the class. A state machine is a behavior that specifies thesequences of states an object goes through during its lifetime in response to events, together with its responses to those events.

- Specify a collaboration that represents the class. A collaboration is a society of rolesand other elements that work together to provide some cooperative behavior that'sbigger than the sum of all the elements. A collaboration has a structural part, as wellas a dynamic part, so you can use collaborations to specify all dimensions of a class'ssemantics.

- Specify the pre- and post-conditions of each operation, plus the invariants of the classas a whole, using a formal language such as OCL.

Advanced Relationships

Dependency:

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line.

- There are 17 such stereotypes, all of which can be organized into six groups.

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

1	**Bind**	Specifies that the source instantiates the target template using the givenactual parameters
2	**Derive**	Specifies that the source may be computed from the target
3	**Friend**	Specifies that the source is given special visibility into the target
4	**instanceOf**	Specifies that the source object is an instance of the target classifier
5	**instantiate**	Specifies that the source creates instances of the target
6	**powertype**	Specifies that the target is a powertype of the source; a powertype is aclassifier whose objects are all the children of a given parent
7	**refine**	Specifies that the source is at a finer degree of abstraction than the target
8	**use**	Specifies that the semantics of the source element depends on the semantics of the public part of the target

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

9	**access**	Specifies that the source package is granted the right to reference theelements of the target package
10	**import**	A kind of access that specifies that the public contents of the targetpackage enter the flat namespace of the source, as if they had been declared in the source

* Two stereotypes apply to dependency relationships among use cases:

11	**extend**	Specifies that the target use case extends the behavior of the source
12	**include**	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

* There are three stereotypes when modeling interactions among objects.

13	**become**	Specifies that the target is the same object as the source but at a laterpoint in time and with possibly different values, state, or roles
14	**call**	Specifies that the source operation invokes the target operation
15	**copy**	Specifies that the target object is an exact, but independent, copy of the source

* One stereotype you'll encounter in the context of state machines is

16	**?send**	Specifies that the source operation sends the target event

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

17	**?trace**	Specifies that the target is an historical ancestor of the source

Generalization:

A generalization is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing(called the subclass or child).

The UML defines one stereotype and four constraints that may be applied to generalization relationships.

1	**?implementation**	Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability

Next, there are four standard constraints that apply to generalization relationships

1	**complete**	Specifies that all children in the generalization have been specified inthe model and that no additional children are permitted
2	**incomplete**	Specifies that not all children in the generalization have been specified(even if some are elided) and that additional children are permitted
3	**disjoint**	Specifies that objects of the parent may have no more than one of thechildren as a type
4	**overlapping**	Specifies that objects of the parent may have more than one of the children as a type

Association:

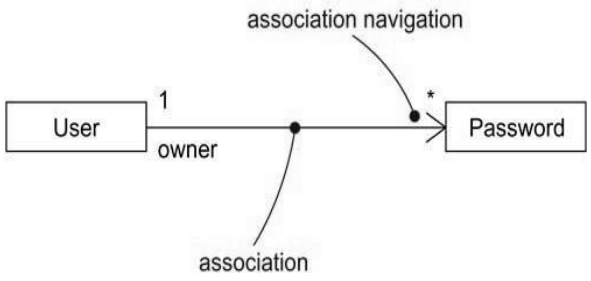
An association is a structural relationship, specifying that objects of one thing are connected to objects of another.

For advanced uses, there are a number of other properties you can use to model subtle details, such as

Navigation
Qualification
various flavors of aggregation.

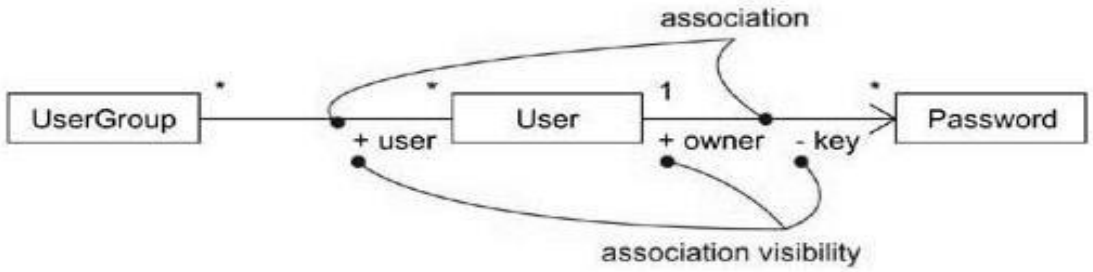
Navigation

- unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified,navigation across an association is bidirectional.
- However, there are some circumstances in which you'll want to limit navigation to justone direction.



Visibility

- However, there are circumstances in which you'll want to limit the visibility acrossthat association relative to objects outside the association.
- In the UML, you can specify three levels of visibility for an association end, just asyou can for a class's features by appending a visibility symbol to a role name the visibility of a role is public.
- Private visibility indicates that objects at that end are not accessible to any objectsoutside the association.
- Protected visibility indicates that objects at that end are not accessible to any objectsoutside the association, except for children of the other end.

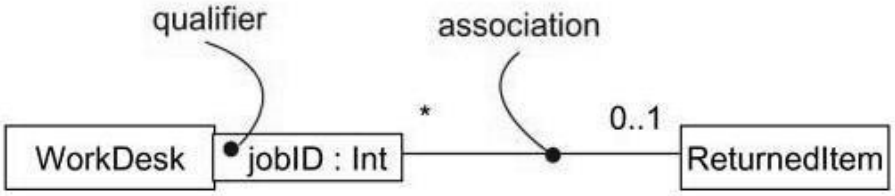


Qualification

Given an object at one end of an association, how do you identify an object or setof objects at the other end?

In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association.

You render a qualifier as a small rectangle attached to the end of an association,placing the attributes in the rectangle

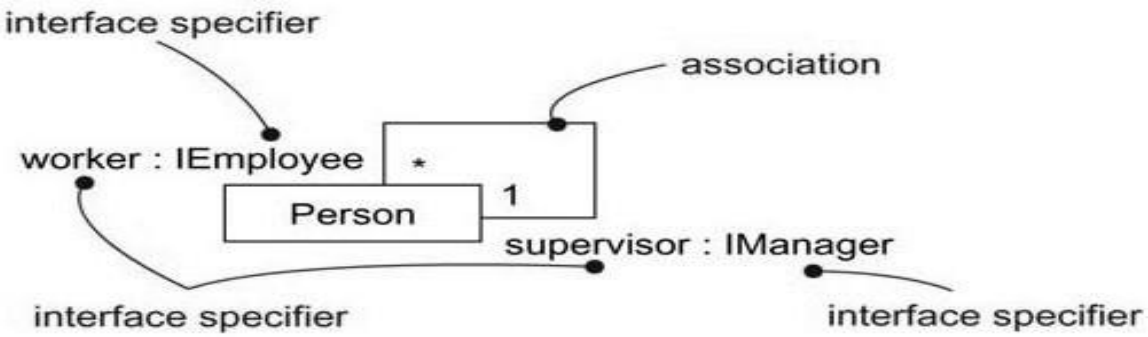


Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or a component

Collectively, the interfaces realized by a class represent a complete specification ofthe behavior of that class.

However, in the context of an association with another target class, a source classmay choose to present only part of its face to the world


interface specifier
association
worker : IEmployee
*
Person
1
supervisor : IManager
interface specifier
interface specifier

a Person class may realize many interfaces: IManager, IEmployee, IOfficer, and so on you can model the relationship between a supervisor and her workers with a one-to-manyassociation, explicitly labeling the roles of this association as supervisor and worker.

- In the context of this association, a Person in the role of supervisor presents only the IManager face to the worker; a Person in the role of worker presents only the IEmployee face to the supervisor. As the figure shows, you can explicitly show the type of role using the syntax rolename : iname, where iname is some interface of the other classifier.

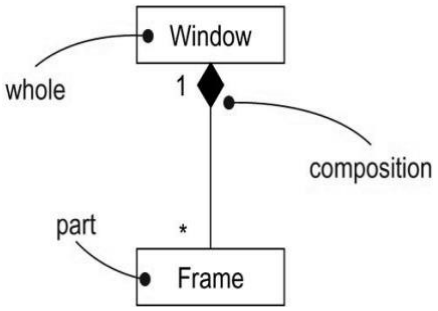
Composition

Composition is a form of aggregation, with strong ownership and coincident lifetime as partof the whole.

Parts with non-fixed multiplicity may be created after the composite itself, but once createdthey live and die with it. Such parts can also be explicitly removed before the death of the composite.

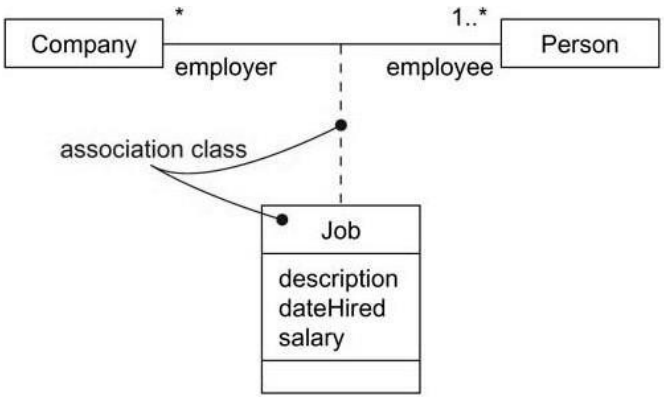
This means that, in a composite aggregation, an object may be a part of only one compositeat a time.

In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts.



Association Classes

* In an association between two classes, the association itself might have properties.
* An association class can be seen as an association that also has class properties, or as aclass that also has association properties.
* We render an association class as a class symbol attached by a dashed line to anassociation



Association Classes

Constraints

* UML defines five constraints that may be applied to association relationships.

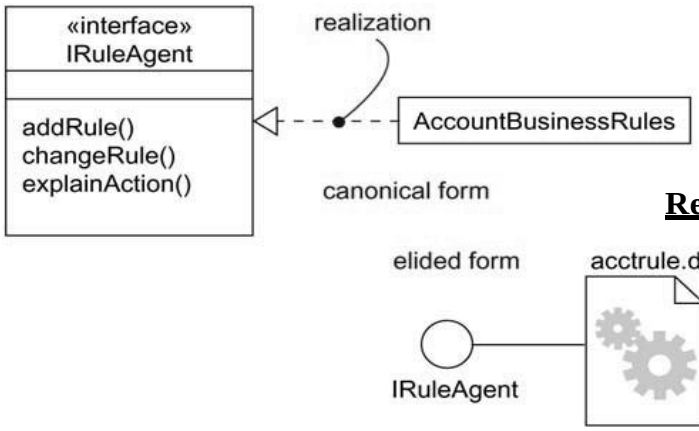
1	**implicit**	Specifies that the relationship is not manifest but, rather, is only conceptual
2	**ordered**	Specifies that the set of objects at one end of an association are in an explicit order
3	**changeable**	Links between objects may be added, removed, and changed freely
4	**addOnly**	New links may be added from an object on the opposite end of the association
5	**frozen**	A link, once added from an object on the opposite end of the association, may not be modified or deleted

Finally, there is one constraint for managing related sets of associations:

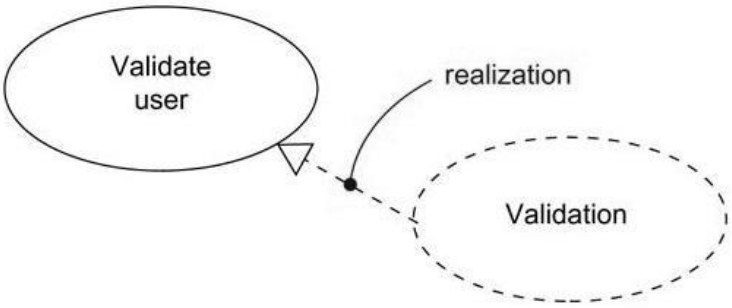
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

1. Realization is sufficiently different from dependency, generalization, and associationrelationships that it is treated as a separate kind of relationship.
2. A realizationis a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
3. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
4. You'll use realization in two circumstances: in the context of interfaces and in the contextof collaborations
5. Most of the time, you'll use realization to specify the relationship between an interfaceand the class or component that provides an operation or service for it.



Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

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

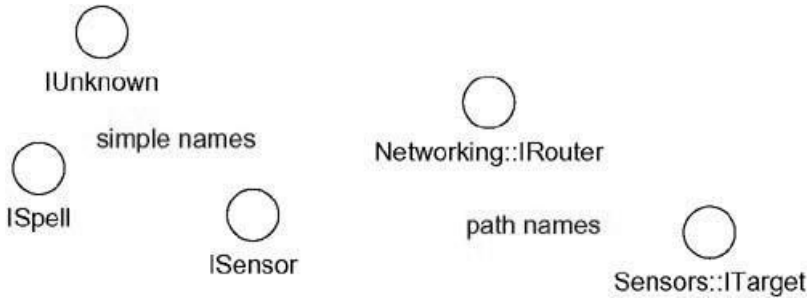
Interfaces, type and roles

Interface

- An interface is a collection of operations that are used to specify a service of a class or acomponent

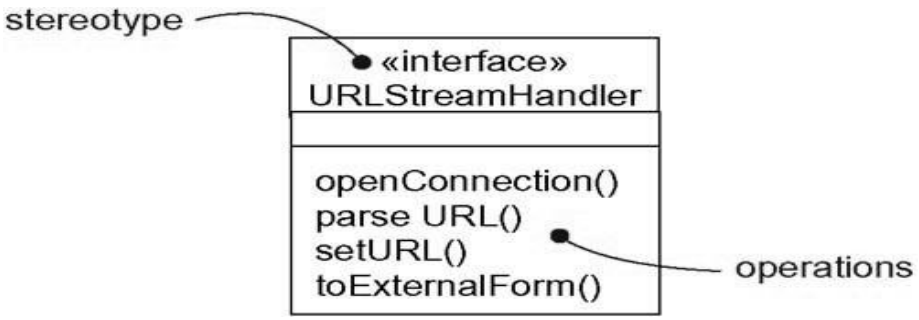
Names

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



Operations

- Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation
- These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- You can render an interface as a stereotyped class, listing its operations in the appropriatecompartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties



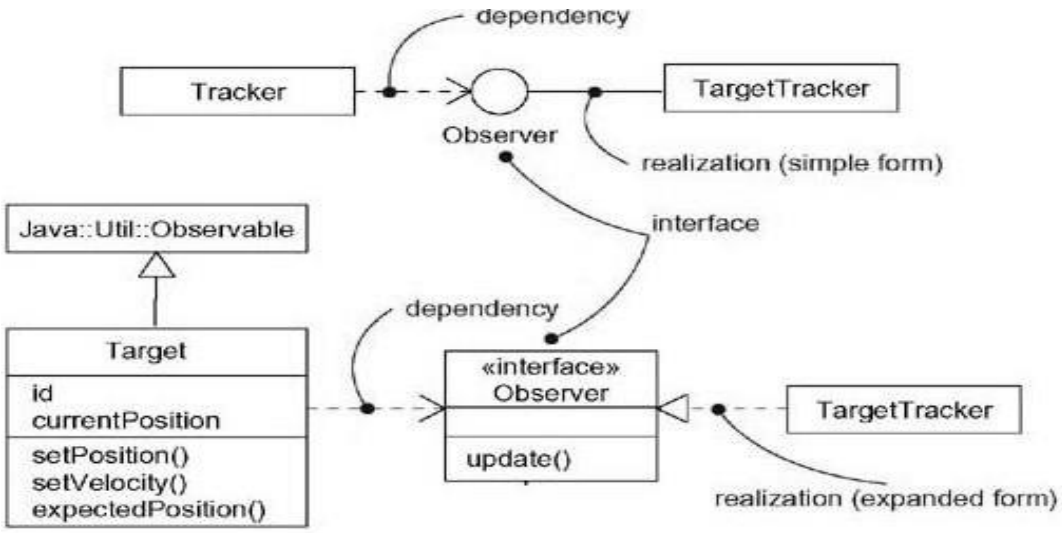
Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces

We can show that an element realizes an interface in two ways.

First, you can use the simple form in which the interface and its realization relationship arerendered as a lollipop sticking off to one side of a class or component.

Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.



Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.

- First, you may attach pre- and post-conditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.

- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.

- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

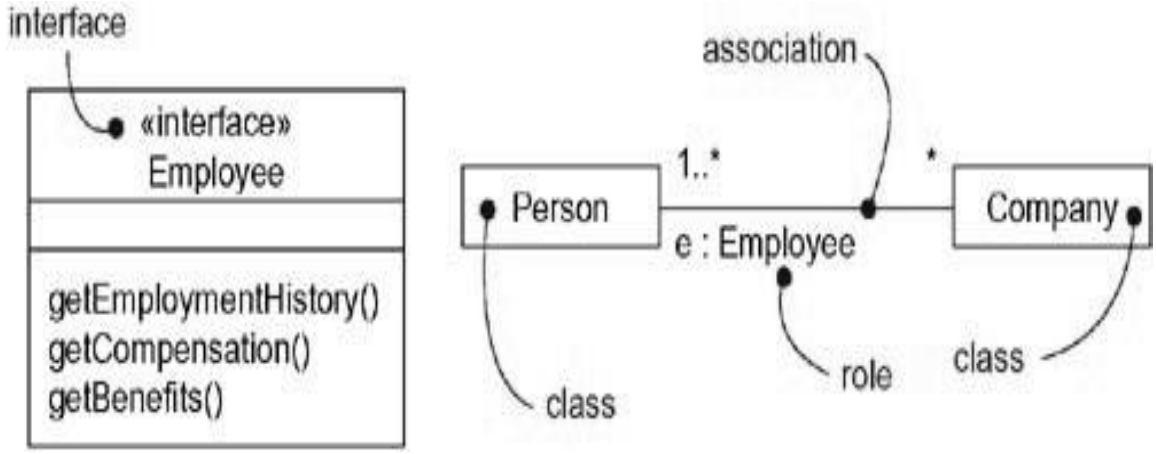
Types and Roles

A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.

an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.



Roles

Common Modeling Techniques

➢ Modeling the Seams in a System

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.

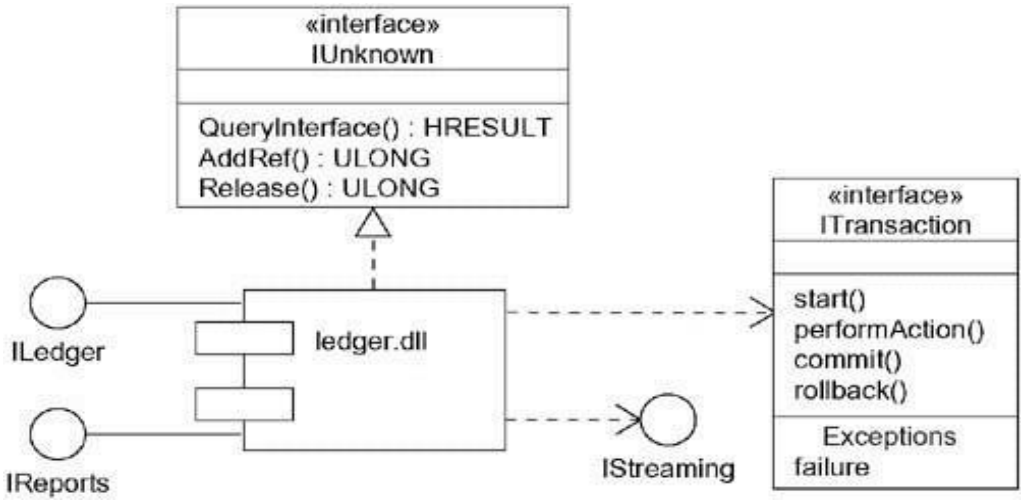
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.

- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.

- Package logically related sets of these operations and signals as interfaces.

- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.

- For each such interface in your system, document its dynamics by using pre- and post-conditions for each operation, and use cases and state machines for the interface as a whole.



- **Modeling Static and Dynamic Typeseling Static and Dynamic Types**

Specify the different possible types of that object by rendering each type as a class stereotyped as type (if the abstraction requires structure and behavior) or as interface (if the abstraction require only behavior).

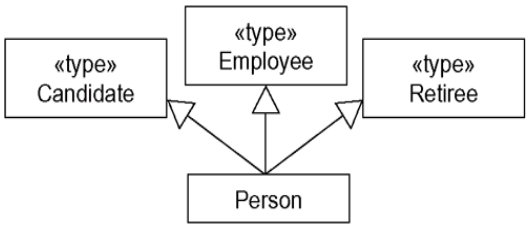
Model all the roles the of the object may take on at any point in time. You can do so in two ways:

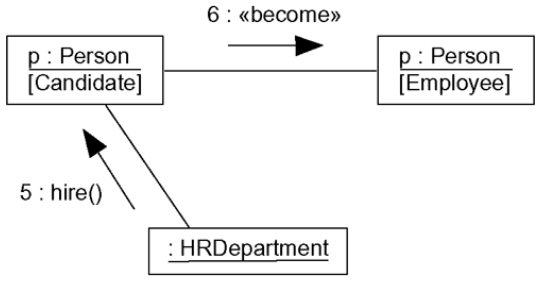
1.) First, in a class diagram, explicitly type each role that the class plays in its association with other classes. Doing this specifies the face instances of that class put on in the context of the associated object.

2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.

To show the change in role of an object, render the object once for each role it plays in theinteraction, and connect these objects with a message stereotyped as become.





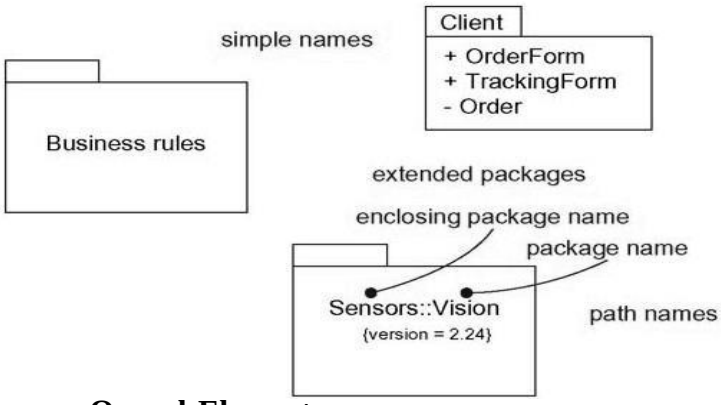
Modeling Static Types **Modeling Dynamic Types**

Packages

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

Names

- Every package must have a name that distinguishes it from other packages. A name is atextual string.
- That name alone is known as a simple name; a path name is the package name prefixedby the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments toexpose their details.



Simple and Extended Package

Owned Elements

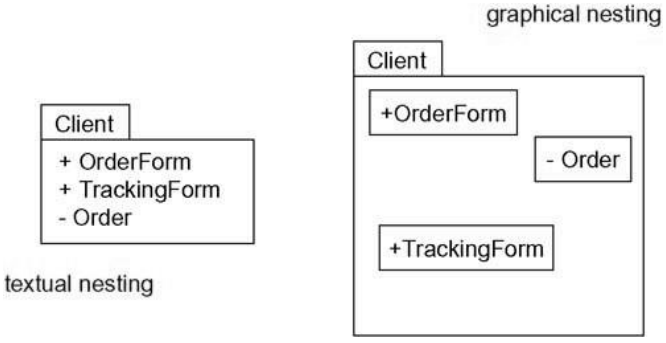
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.

Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquelyowned by exactly one package.

Elements of different kinds may have the same name within a package. Thus, you can have a class named Timer, as well as a component named Timer, within the same package.

Packages may own other packages. This means that it's possible to decomposeyour models hierarchically.

We can explicitly show the contents of a package either textually or graphically.



Visibility

You can control the visibility of the elements owned by a package just as you cancontrol the visibility of the attributes and operations owned by a class.

Typically, an element owned by a package is public, which means that it is visibleto the contents of any package that imports the element's enclosing package.

Conversely, protected elements can only be seen by children, and private elementscannot be seen outside the package in which they are declared.

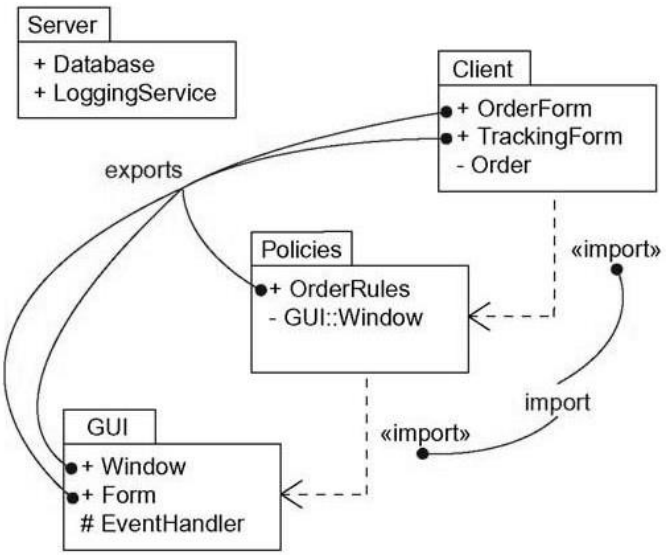
We specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol.

Importing and Exporting

In the UML, you model an import relationship as a dependency adorned with the stereotype import.

Actually, two stereotypes apply here—import and access— and both specify that the source package has access to the contents of the target. Import adds the contents of the target to the source's namespace. Access does not add the contents of the target. The public parts of a package are called its exports.

The parts that one package exports are visible only to the contents of those packages that explicitly import the package. Import and access dependencies are not transitive

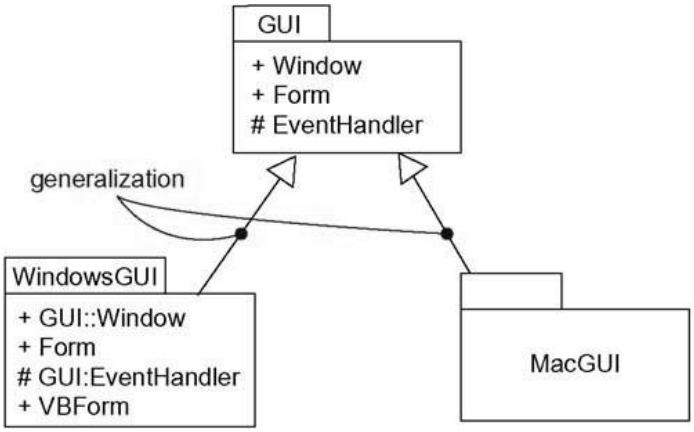


Importing and Exporting

Generalization

There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages.

A specialized package (such as WindowsGUI) can be used anywhere a moregeneral package (such as GUI) is used



Generalization Among Packages

Standard Elements

The UML defines five standard stereotypes that apply to packages

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

Common Modeling Techniques

Modeling Groups of Elements

- Scan the modeling elements in a particular architectural view and look for clumpsdefined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hidethe element.
- Explicitly connect packages that build on others via import dependencies.
- In the case of families of packages, connect specialized packages to their moregeneral part via generalizations.

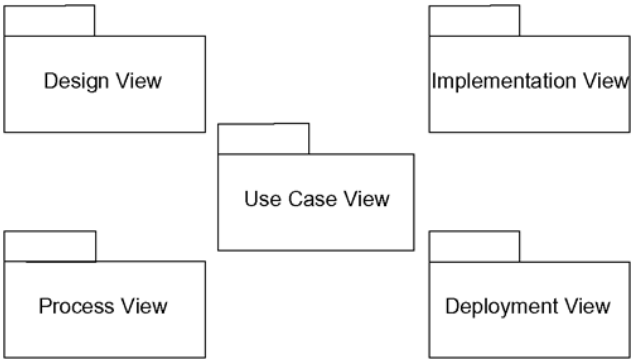
Modeling Architectural Views

Identify the set of architectural views that are significant in the context of your problem. Inpractice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.

Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.

As necessary, further group these elements into their own packages.

There will typically be dependencies across the elements in different views. So, in general,let each view at the top of a system be open to all others at that level.



Modeling Architectural Views

Class Diagrams

Terms and Concepts

A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.
Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships
- Class diagrams may also contain packages or subsystems, both of which are used togroup elements of your model into larger chunks.

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 thesystem 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.

  - To model the vocabulary of a system
  - To model simple collaborations
  - To model a logical database schema

To Model a Simple 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.

To Model Logical Database Schema:

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

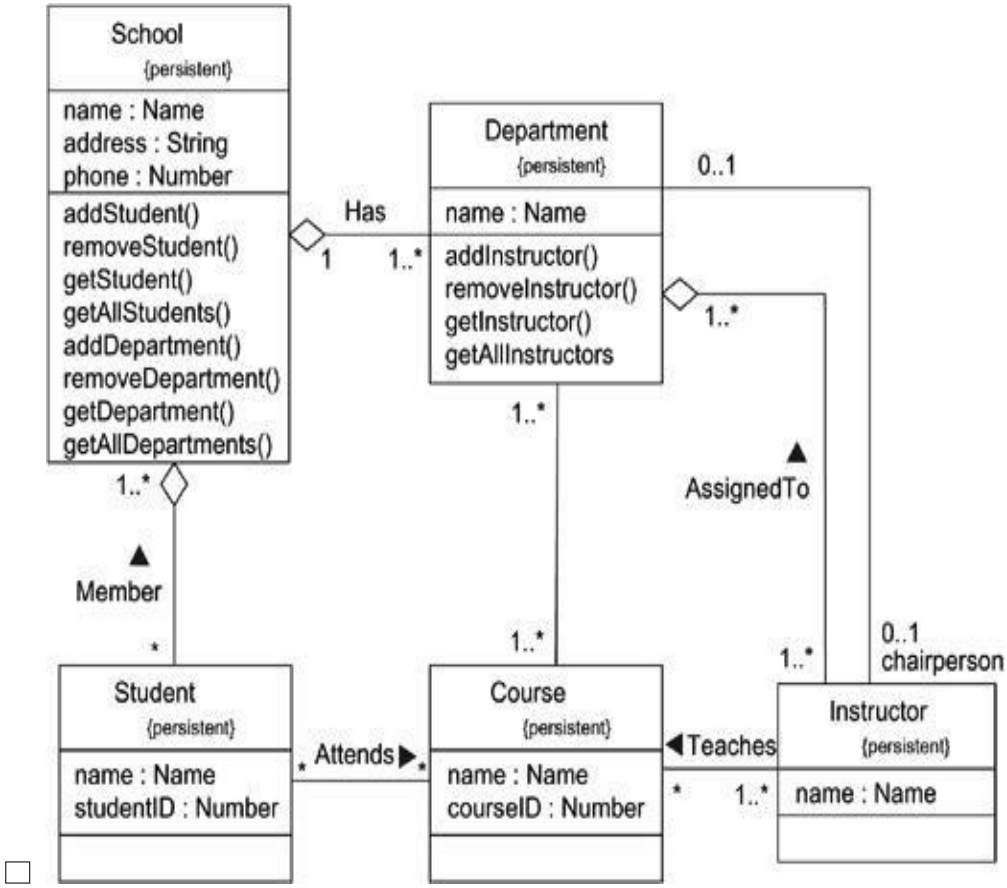
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.

- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities thatstructure these classes.

- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary 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.

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



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 theUML 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:

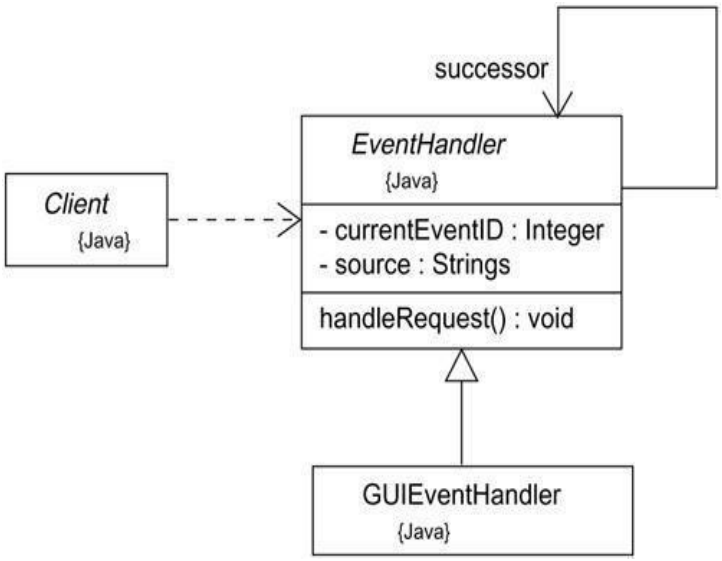
- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

- Depending on the semantics of the languages you choose, you may have 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 either choose to prohibit developers from modeling with multiple inheritance (which makesyour models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping

more complex).

- Use tagged values to specify 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 forward engineer your models.



All of these classes specify a mapping to Java, as noted in their tagged value. 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.
- 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 awhole.

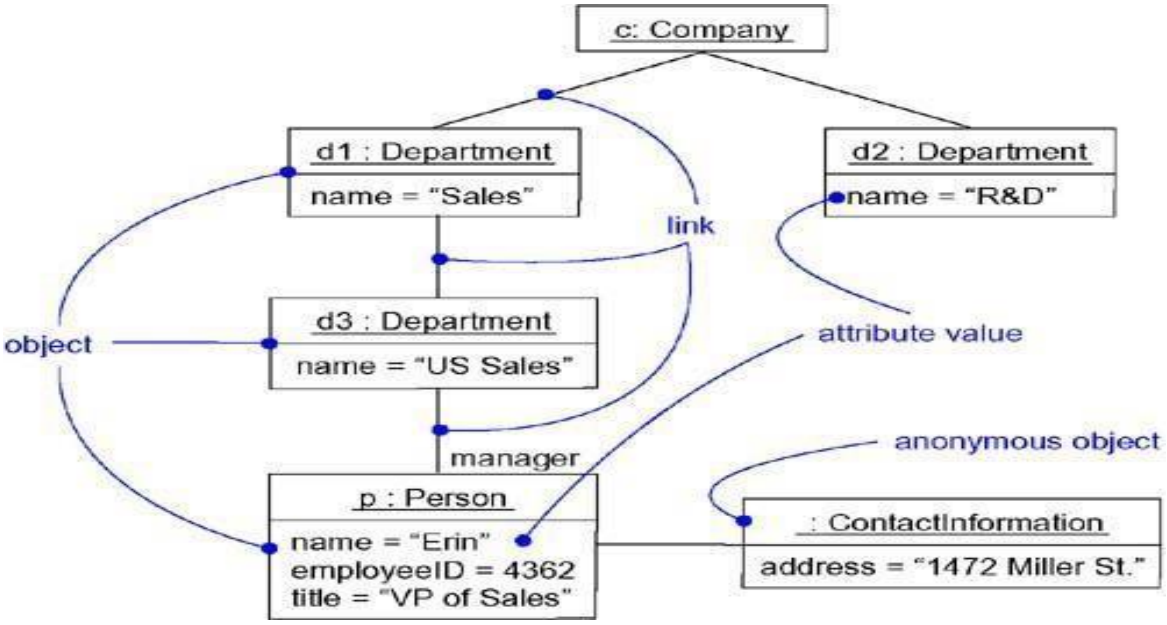
- Using a tool, point to the code you'd like to reverse engineer. Use your tool togenerate a new model or modify an existing one that was previously forwardengineered.

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

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.

- Contents: Object diagrams commonly
  containObjects
  Links



Common Modeling

Techniques Modeling Object

Structures:

- 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 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 istheoretically possible but pragmatically of limited value.
- In an object-oriented system, instances are things that are created and destroyed bythe application during run time.
- In these cases, component instances and node instances are things that live outside therunning system and are amenable to some degree of forward engineering.
- Reverse engineering (the creation of a model from code) an object diagram is a verydifferent thing.
- In fact, while you are debugging your system, this is something that you or your toolswill do all the time.

To reverse engineer an object diagram:

- Chose the target you want to reverse engineer. Typically, you'll set your contextinside an operation or relative to an instance of one particular class.

- Using a tool or simply walking through a scenario, stop execution at a certainmoment in time.

- Identify the set of interesting objects that collaborate in that context and render themin 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 theseobjects.

- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

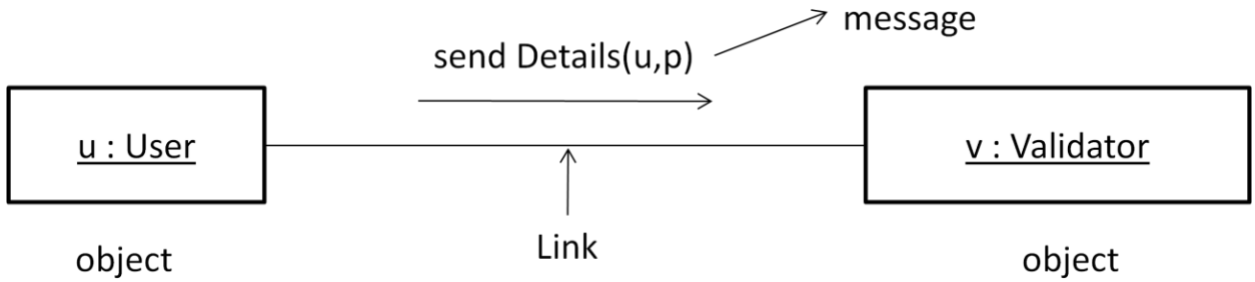
4.0 Basic Behavioral Modeling: 4.1 Interactions, 4.2 Interaction diagrams, 4.3 Use cases, 4.4 Use case Diagrams, 4.5 Activity Diagrams.

4.1 Interactions

Introduction:

In UML, the dynamic aspects of a system can be modeled using interactions. Interactions contain messages that are exchanged between objects. A message can be an invocation of an operation or a signal. The messages may also include creation and destruction of other objects.

We can use interactions to model the flow of control within an operation, a class, a component, a use case or the system as a whole. Using interaction diagrams, we can model these flows in two ways: one is by focusing on how the messages are dispatched across time and the second is by focusing on the structural relationships between objects and then consider how the messages are passed between the objects. Graphically a message is rendered as a directed line with the name of its operation as show below:



Interaction (Definition):

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

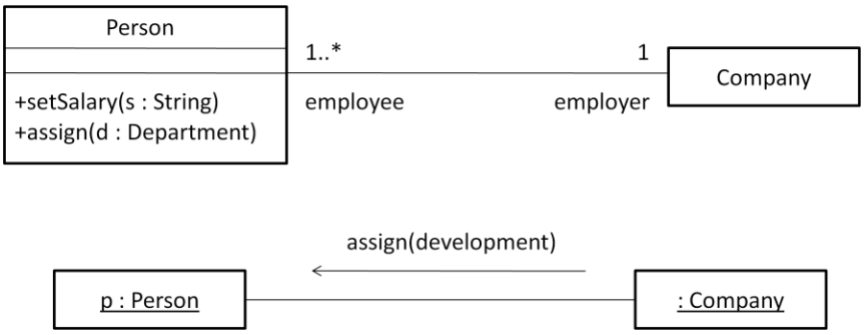
Objects and Roles:

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

Links:

A link is a semantic connection among objects. In general, a link is an instance of association. Wherever, a class has an association with another class, there may be a link between the instances of the two classes. Wherever there is a link between two objects, one object can send messages to another object. We can adorn the appropriate end of the link with any of the following standard stereotypes:

association	Specifies that the corresponding object is visible by association
Self	Specifies that the corresponding object is visible as it is the dispatcher of the operation
Global	Specifies that the corresponding object is visible as it is in an enclosing scope
Local	Specifies that the corresponding object is visible as it is in local scope
parameter	Specifies that the corresponding object is visible as it is a parameter

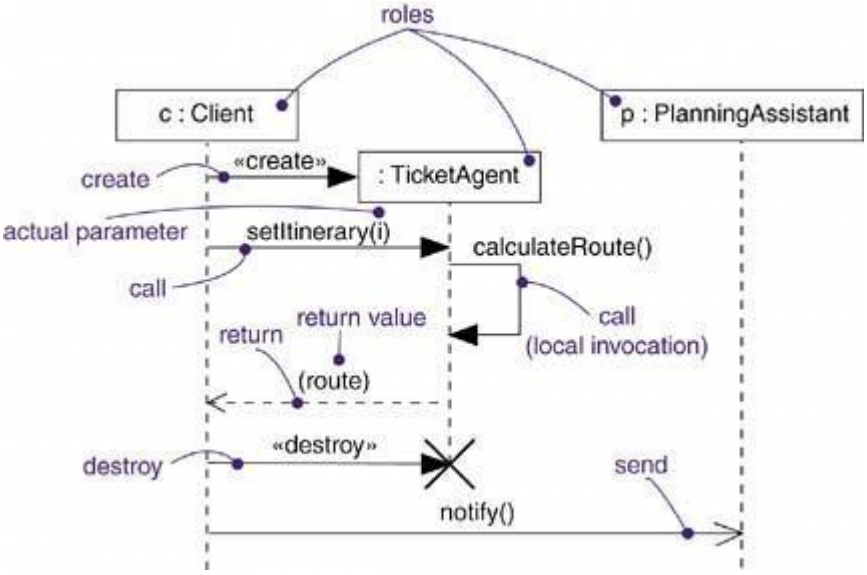


Messages

A message is the specification of communication among objects that conveys information with the expectation that activity will succeed. The receipt of a message instance may be considered an instance of an event.

When a message is passed, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change of state. In UML, we can model several kinds of actions like:

Call	Invokes an operation on an object
Return	Returns a value to the caller
Send	Sends a signal to the object
Create	Creates an object
Destroy	Destroys an object

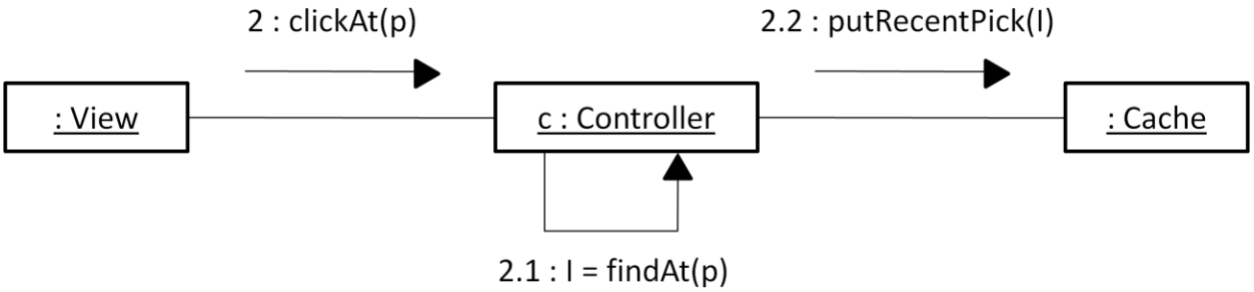


Sequencing:

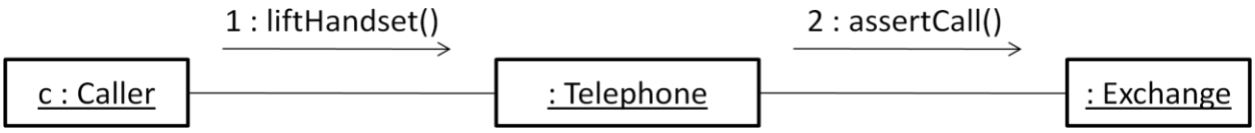
When an object passes a message to another object, the receiving object might in turn send a message to another object, which might send a message to yet a different object and so on. This stream of messages forms a sequence. So, we can define a sequence as a stream of messages. Any sequence must have a beginning. The start of every sequence is associated with some process or thread.

To model the sequence of a message, we can explicitly represent the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

Most commonly, we can specify a procedural or nested flow of control, rendering using a filled solid arrowhead. Less common but also possible, you can specify a flat flow of control, rendered using a stick arrowhead. We will use flat sequences only when modeling interactions in the context of use cases that involve the system as a whole, together with actors outside the system. In all other cases, we will use procedural sequences, because they represent ordinary, nested operation calls of the type we find in most programming languages.



(a) Procedural Sequence



(b) Flat Sequence

Representation:

When we model an interaction, we typically include both objects and messages. We can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of messages and by emphasizing the structural organization of the objects that send and receive messages. In UML, the first kind of representation is called a sequence diagram and the second kind of representation is called a collaboration diagram. Both sequence and collaboration diagrams are known as interaction diagrams.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other without loss of information. Sequence diagram lets us to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time. A collaboration diagram lets us to model the structural links that may exist among the objects in the interaction.

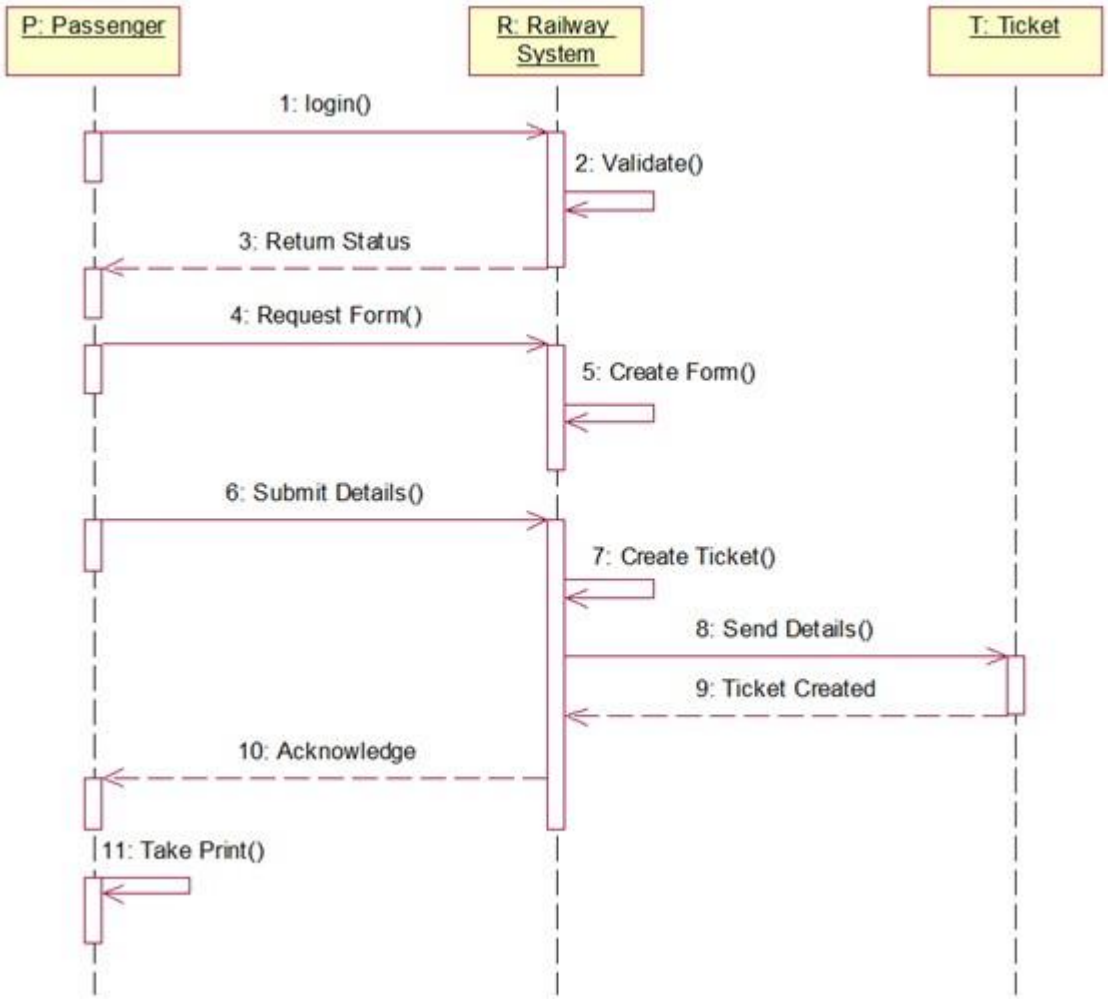
Common Modeling Techniques

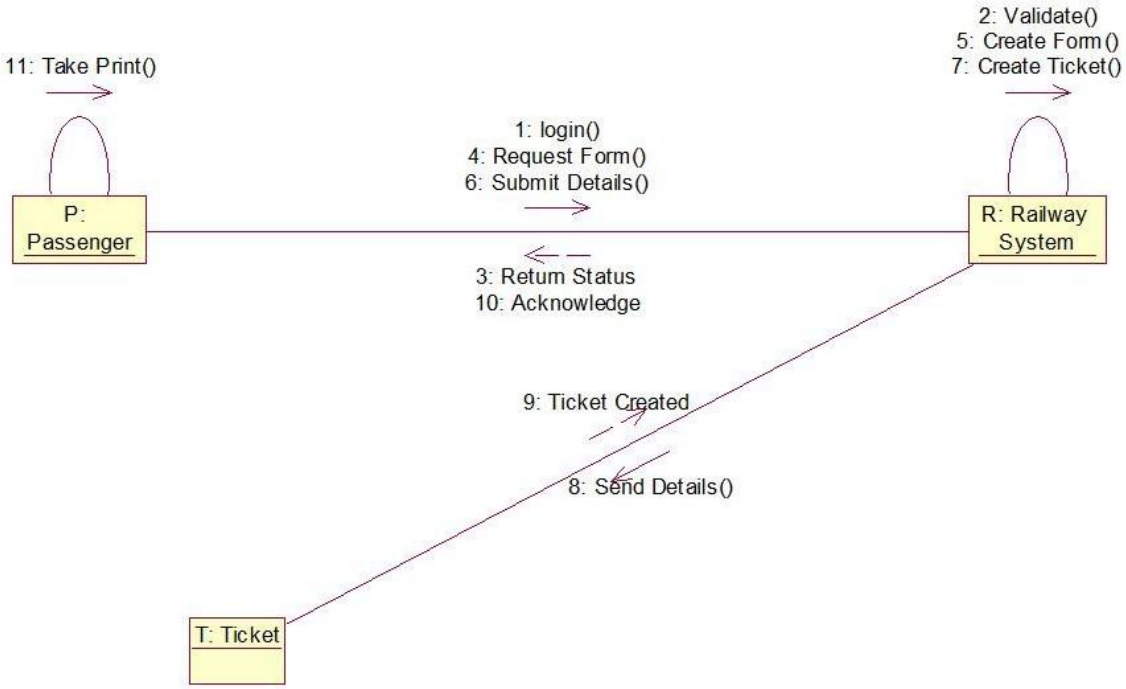
Modeling a flow of control:

To model a flow of control,

1. Set the context for the interaction, whether it is the system as a whole, a class or an individual operation.
2. Identify the objects and their initial properties which participate in the interaction.
3. Identify the links between objects for communication through messages.
4. In time order, specify the messages that pass from object to object. Use parameters and return values as necessary.
5. To add semantics, adorn each object at every moment in time with its state and role.

Consider the following example of railway reservation system's sequence and collaboration diagrams:





4.2 Interaction Diagrams

Introduction:

An interaction diagram represents an interaction, which contains a set of objects and the relationships between them including the messages exchanged between the objects. A sequence diagram is an interaction diagram in which the focus is on time ordering of messages. Collaboration diagram is another interaction diagram in which the focus is on the structural organization of the objects. Both sequence diagrams and collaboration diagrams are isomorphic diagrams.

Common Properties:

Interaction diagrams share the properties which are common to all the diagrams in UML. They are: a name which identifies the diagram and the graphical contents which are a projection into the model.

Contents:

Interaction diagrams commonly contain:

1. Objects
2. Links
3. Messages

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

Sequence Diagrams:

A sequence diagram is one of the two interaction diagrams. The sequence diagram emphasizes on the time ordering of messages. In a sequence diagram, the objects that participate in the interaction are arranged at the top along the x-axis. Generally, the object which initiates the interaction is placed on the left and the next important object to its right and so on. The messages dispatched by the objects are arranged from top to bottom along the y-axis. This gives the user the detail about the flow of control over time.

Sequence diagram has 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. Most of the objects are alive throughout the interaction. Objects may also be created during the interaction with the receipt of the message stereotyped with create. Objects may also be destroyed during the interaction with the receipt of the message stereotyped with destroy.

Second, there is focus of control which is represented as a thin rectangle over the life line of the object. The focus of control represents the points in time at which the object is performing an action. We can also represent recursion by using a self message.

Collaboration Diagrams:

A collaboration diagram is one of the two interaction diagrams. The collaboration diagram emphasizes on the structural organization of the objects in the interaction. A collaboration diagram is made up of objects which are the vertices and these are connected by links. Finally, the messages are represented over the links between the objects. This gives the user the detail about the flow of control in the context of structural organization of objects that collaborate.

Collaboration diagram has two features that distinguish them from the sequence diagrams. First, there is a path which indicates one object is linked to another. Second, there is a sequence number to indicate the time ordering of a message by prefixing the message with a number. We can use Dewey decimal numbering system for the sequence numbers. For example a message can be numbered as 1 and the next messages in the nested sequence can be numbered 1.1 and so on.

Common Uses:

We use interaction diagrams to model the dynamic aspects (interactions) of the system. When we use an interaction diagram to model some dynamic aspect of a system, we do so in the context of the system as a whole, a subsystem, an operation or a class. We typically use the interaction diagrams in two ways:

1. To model flows of control by time ordering
2. To model flows of control by organization

Common Modeling Techniques

Modeling flow of control by time ordering:

To model a flow of control by time ordering,

1. Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
2. Identify the objects that take part in the interaction and lay them out at the top along the x-axis in a sequence diagram.
3. Set the life line for each object.
4. Layout the messages between objects from the top along the y-axis.
5. To visualize the points at which the object is performing an action, use the focus of control.
6. To specify time constraints, adorn each message with the time and space constraints.
7. To specify the flow of control in a more formal manner, attach pre and post conditions to each message.

Modeling flow of control by organization:

To model a flow of control by organization,

1. Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
2. Identify the objects that take part in the interaction and lay them out in a collaboration diagram as the vertices in a graph.
3. Set the initial properties of each of these objects.
4. Specify the links among these objects.
5. Starting with the messages that initiate the interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Use Dewey numbering system to specify nested flow of control.
6. To specify time constraints, adorn each message with the time and space constraints.
7. To specify the flow of control in a more formal manner, attach pre and post conditions to each message.

4.3 Use Cases

Introduction:

Use case modeling describes what a system does or what is the functionality provided by the system to benefit the users. Use case modeling was created by Ivar Jacobson. More than any other diagrams in UML, use case diagrams allow us to quickly gather the requirements of the software system. The primary components of a use case model are use cases, actors or roles and the system being modeled also known as the subject.

The primary purpose of use cases are:

1. To describe the functional requirements of the system, resulting in an agreement between the stakeholders and the software developers who are developing the system.
2. To give a clear and consistent description of what the system should do.
3. To provide a basis for conducting system tests to verify whether the system works appropriately or not.
4. To provide the ability to transform functional requirements into classes and operations in the system.

Use Case:

A use case represents the functionality provided by the system to the user. A use case is defined as "a set of actions performed by the system, which produces an observable result that is, typically, of some value to one or more actors or other stakeholders of the system". The actions can include communicating with other actors or systems as well as performing calculations inside the system. The characteristics of a use case are:

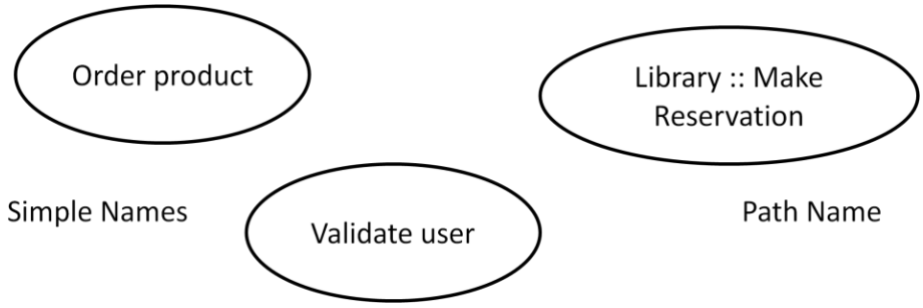
1. A use case is always initiated by an actor.
2. A use case provides value to an actor.
3. A use case is complete.

Use cases are connected to actors through associations, which are sometimes referred to as communication associations. Associations represent which actors the use case is communicating with. The association should always be binary, implying a dialog between the actor and system.

A use case is a classifier, not an instance. A use case represents the functionality as a whole with possible alternatives, errors and exceptions that can occur during the execution of the use case. An instance of the use case is known as a scenario, which represents a specific execution path through the system.

Example: Consider a online shopping system,
Use case: Purchase Product
Scenario: A user named surya purchases a product, gaming console by using debit card payment option.

The graphical representation of a use case is a solid ellipse with the use case name specified inside the ellipse. A use case generally placed inside the system boundary and is always connected to an actor using an association relationship. The name of the use case is generally a phrase rather a one word label. The name can be a simple name or a path name as shown in the below example:

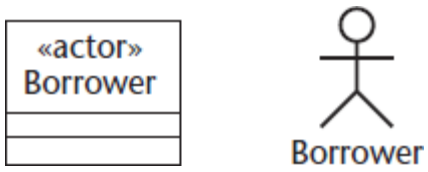


Actors:

An actor is one which interacts or uses the system (subject). An actor sends or receives messages from the system. Actor can be either a person or another system (computer or application). An actor is a classifier, not an instance. An actor represents a role, not an individual user of the system. For example, in an online shopping system, if ramesh wants to purchase a product, his role will be buyer.

An actor has a name, and the name should reflect the actor's role. The name should not represent an instance of the actor or the functionality of an actor. A use case is always initiated by an actor that sends a message to it. This message is also known as stimulus. Actors may be of two types: active actors and passive actors. Active actors or those which initiates a use case and passive actors are those which participate in a use case but never initiate it.

Actor can be represented with a class symbol stereotyped as <<actor>>. Actor has a standard stereotype icon known as the "stickman". An actor can have both attributes and behavior.



Flow of Events:

A use case describes what a system does but it does not specify how it does that. We can specify the behavior of a use case by describing the flow of events in text clearly enough for an outsider to understand it easily. When we write the flow of events, we must specify when the use case starts and ends, what objects are exchanged between the system and the actor, the basic/main flow and alternate flows of events. For example in an ATM system, we can describe the use case "Validate User" in the following way:

Main flow of events:

The use case starts when the system prompts the customer for a PIN number. The customer can now enter a PIN number via the keypad. The customer commits the entry by pressing the Enter button. The system then checks this PIN number to see if its valid or not. If the PIN is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

Customer can cancel the transaction at any point by pressing the Cancel button thereby restarting the use case. No changes are made to the customer's state.

Exceptional flow of events:

Customer can clear the PIN number anytime before confirming it and reenter a new PIN number again.

Exceptional flow of events:

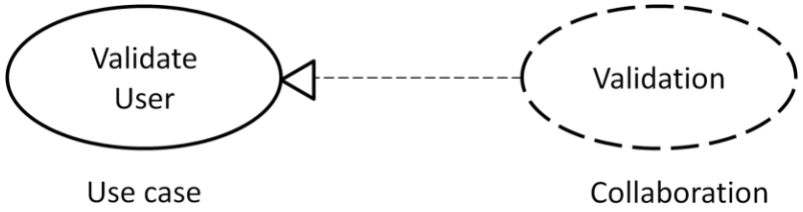
If the Customer enters an invalid PIN number, the use case restarts. If this happens, three times in a row, the system cancels the transaction, preventing the Customer from interacting with the ATM for 60 seconds.

Scenarios:

A scenario is a specific sequence of actions that illustrates behavior. Scenario is an instance of a use case like objects are instances of classes. For example, in an online shopping system consider the use case "Purchase Product". For this use case, a scenario can be: user Ramesh purchases a laptop. Another scenario can be user Mahesh purchases a washing machine etc..

Collaborations:

A use case specifies what the system does but does not specify how it is implemented. A use case is implemented by creating a collection of classes and other elements that work together to achieve the behavior of the use case. This collection of elements, including both its static and dynamic structure, is modeled in UML as collaboration.

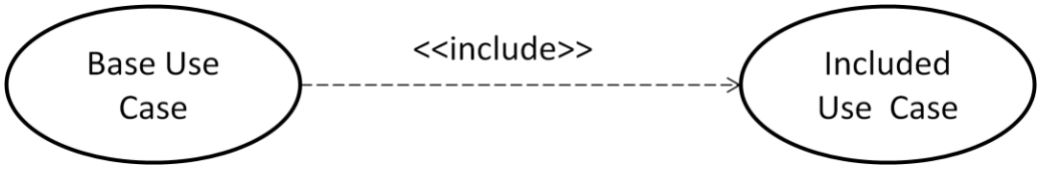


Organizing Use Cases:

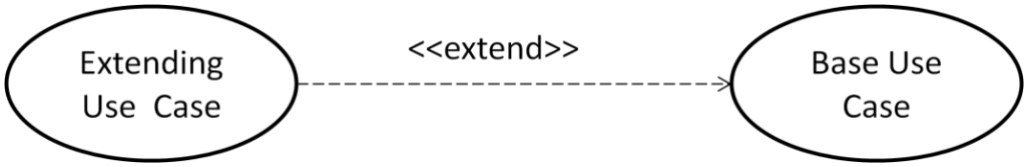
We can organize the use cases by grouping them in packages in the same manner in which we can organize classes. We can also organize use cases by specifying generalization, include and extend relationships. We apply these relationships in order to factor common behavior and in order to factor variants (alternatives).

Generalization among use case represents that the child use case inherits the behavior from the parent use case. For example in an ATM system, the behavior of the use case "Validate User" is to check whether the user is a valid user or not. To implement this, system might ask for a PIN number or might ask for a retinal scan of the eye or may ask for finger scan. All these three, PIN validation, Retinal scan and Finger scan are specialized ways of checking the validity of a user and can be applied at any place where the "Validate User" use case appears.

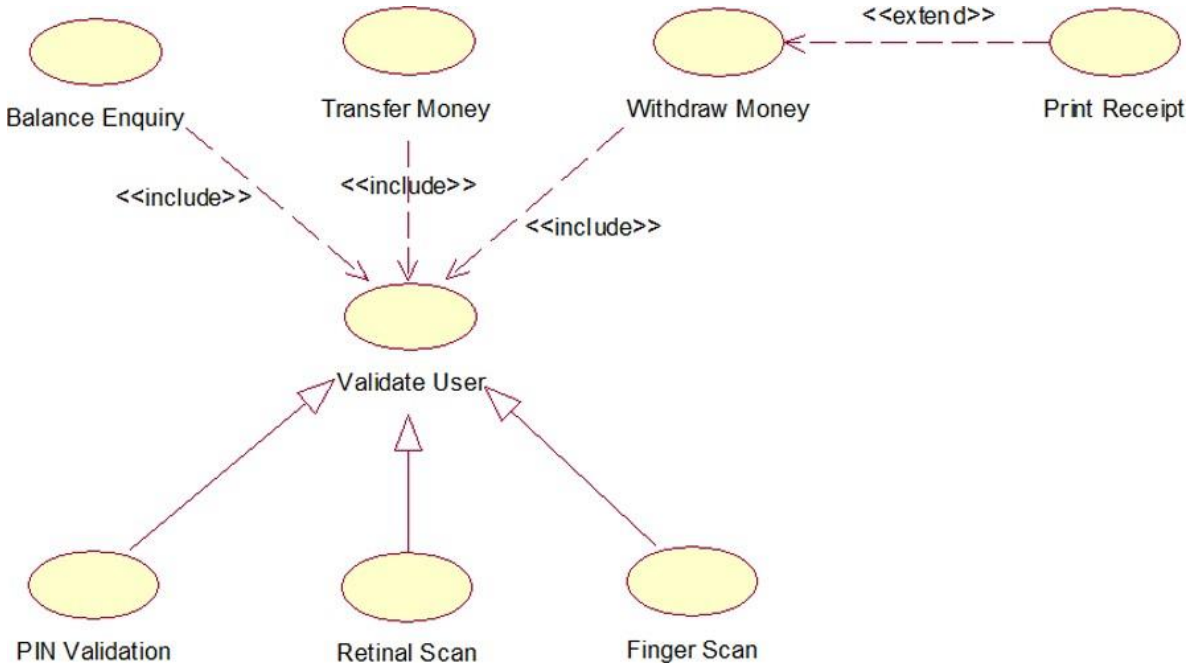
An include relationship between use cases means that the base use case explicitly incorporates the behavior of the included use case. Such relationship is represented as a dependency stereotyped with include. In an include relationship, the base use case cannot exist without the included use case. The include relationship is used to separate the common behavior.



An extend relationship between use cases means that the extending use case extends the behavior of the base use case. Such relationship is graphically represented as dependency stereotyped with exclude. In a extend relationship, the base use case may exist on its own but at certain points its behavior may be extended by the behavior of another use case. The extend relationship is used to separate the optional behavior.



Consider the following which illustrates the use of generalization, include and extend relationships between use cases:

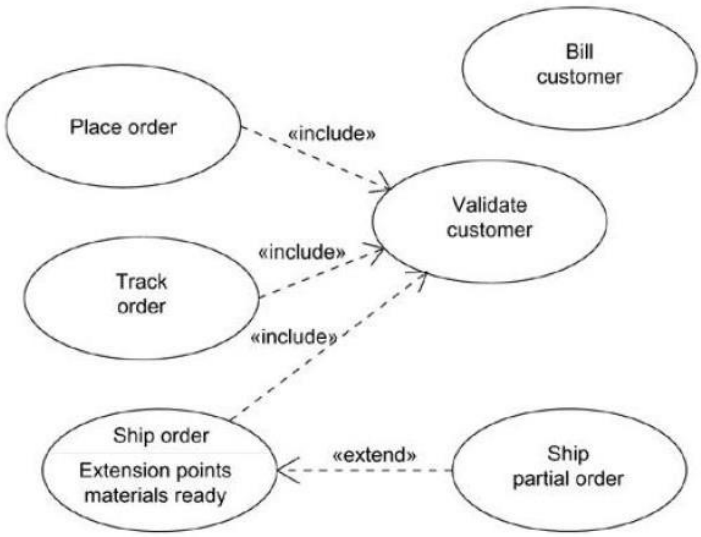


Common Modeling Techniques:

Modeling the behavior of an element:

To model the behavior of an element:

1. Identify the actors that interact with the element.
2. Organize the actors by identifying the general and more specialized roles.
3. For each actor, consider the primary ways in which the actor interacts with the element.
4. Consider also the alternative ways in which the actor interacts with the element.
5. Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.



4.4 Use Case Diagrams

Use case diagrams are one of the five behavioral diagrams for modeling the dynamic aspects of the system. A use case diagram specifies what the system does (behavior) for the actors. Use case diagram plays a key role in modeling the behavior of a system or subsystem or a class. Each use case diagram consists of a set of use cases, actors and their relationships.

Common Properties:

Use case diagrams shares some common properties with the rest of the UML diagrams like a name which identifies a use case diagram and the graphical content which is a projection into a model.

Content:
Use case diagrams commonly contain:

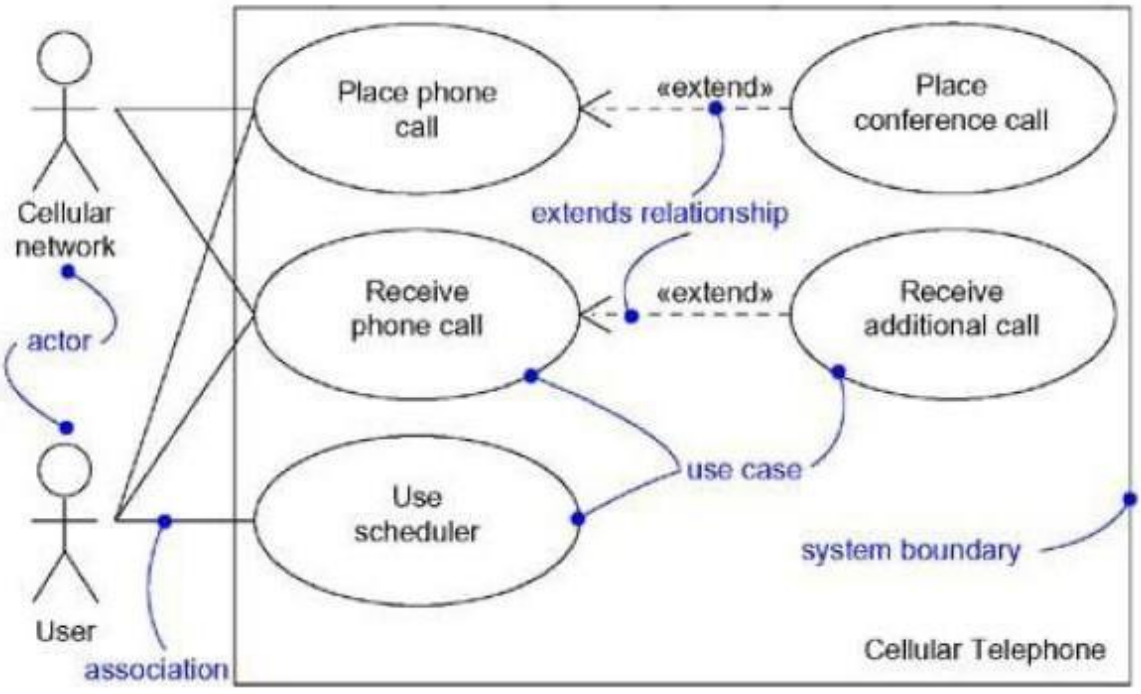
- Use cases
- Actors
- Dependency, generalization and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints. They may also contain packages or subsystems, which are used to group elements of your model into larger chunks.

Common Uses:

When we model the static use case view of a system, we apply use case diagrams in one of the two ways:

- To model the context of a system
- To model the requirements of a system



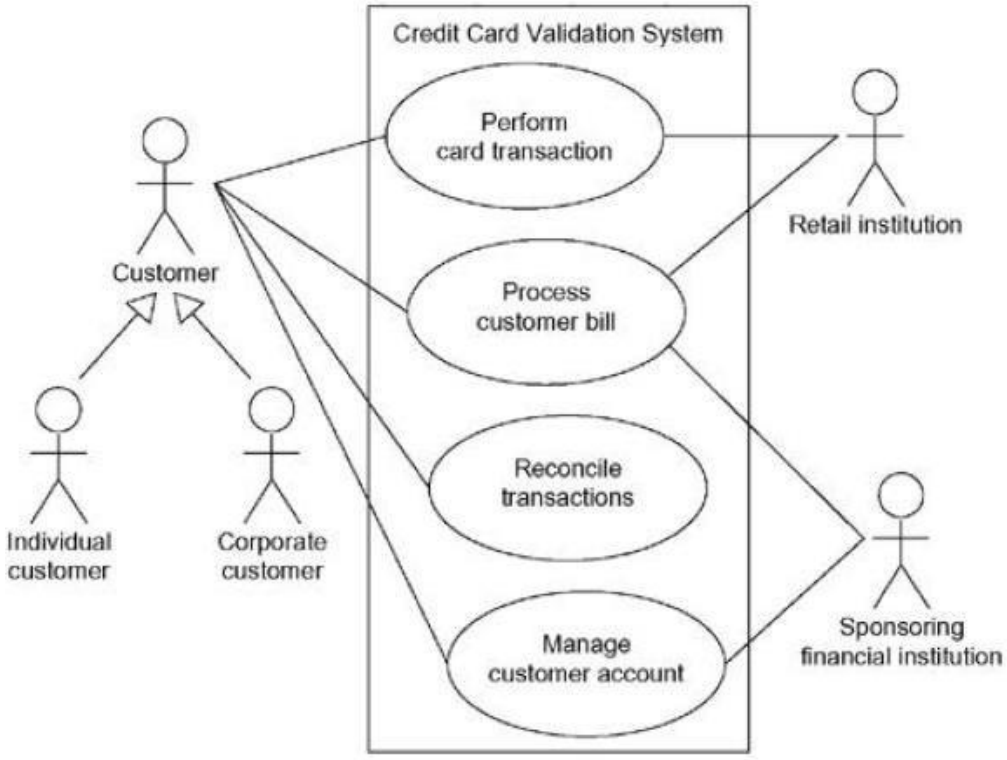
Common Modeling Techniques

Modeling the context of a system:

All the things (actors) that live outside the system and interact with the system constitute the system's context. This context defines the environment in which the system lives.

To model the context of a system:

1. Identify the actors that surround the system.
2. Organize actors that are similar to one another in a generalization-specialization hierarchy.
3. For better semantics, provide a stereotype for each actor.
4. Create a use case diagram with these actors and specify the paths of communication for each actor to the system's uses cases using association relationships.

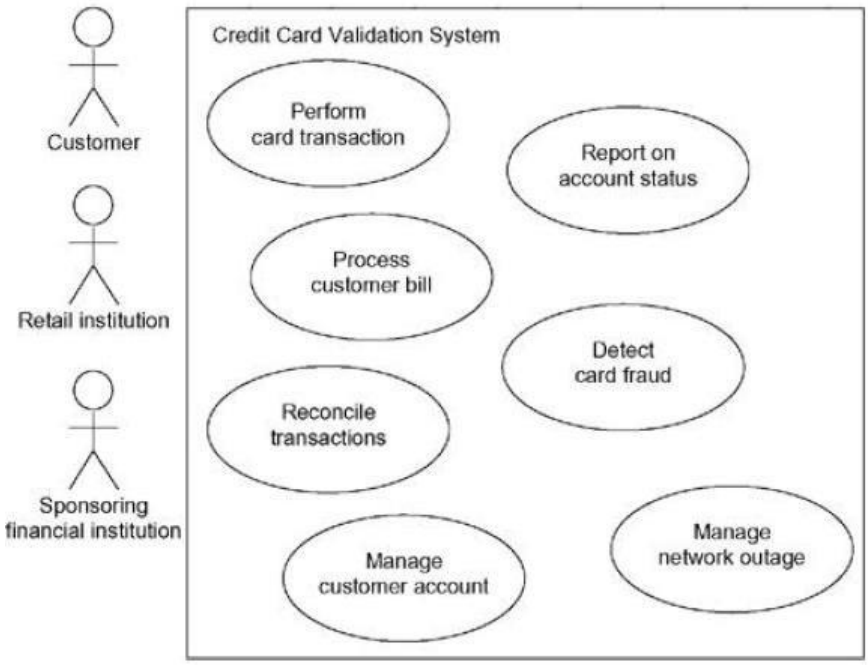


Modeling the requirements of a system:

A system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system:

1. Establish the context of the system by identifying the actors that surround (use) the system.
2. For each actor, identify the behavior that the actor expects from the system or requires the system to provide.
3. Name these common behaviors as use cases.
4. Use include and extend relationships to factor common behavior and optional behavior.
5. Model these use cases, actors and their relationships in a use case diagram.
6. Use notes to provide details about the use cases and other non-functional requirements.



Forward and reverse engineering:

To forward engineer a use case diagram:

1. For each use case, identify the flow of events and its exceptional flow of events.
2. Generate a test script for each flow, using the flow's preconditions as the test's initial state and its post conditions as its success criteria.
3. As necessary, simulate test runs to represent each actor that interacts with the use case.
4. Use tools to run these tests each time you release the element to which the use case diagram applies.

To reverse engineer a use case diagram:

1. Identify each actor that interacts with the system.
2. For each actor, consider how the actor interacts with the system.
3. Trace the flow of events in the executable system relative to each actor.
4. Cluster (group) related flows by declaring a corresponding use case. Use include and extend relationships as necessary.
5. Model these actors and use cases in a use case diagram, and establish their relationships.

4.5 Activity Diagrams

Introduction:

An activity diagram is like a flowchart, representing flow of control from activity to activity, whereas, the interaction diagrams focus on the flow of control from object to object. Activities result in some action, which is made up of executable atomic computations that result in a change of state of the system or the return of a value. Actions involve calling another operation, sending a signal, creating or destroying an object or some pure computation, such as evaluating an expression.

Common Properties:

An activity diagram shares the same common properties as do all other UML diagrams like a name which is used to uniquely identify the diagram and the graphical content which is a projection into the model.

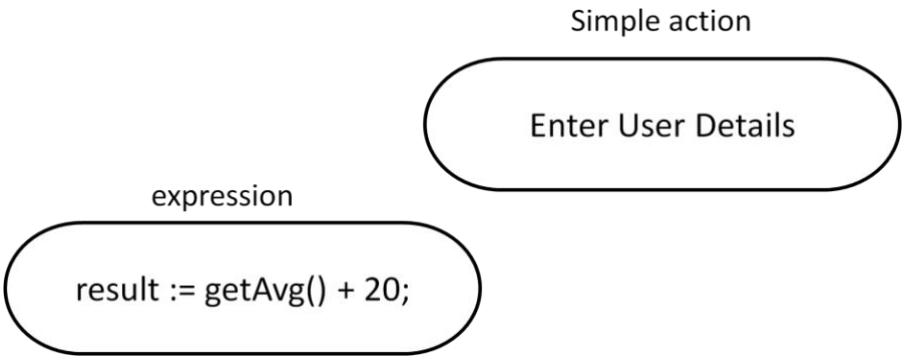
Content:

Activity diagrams mainly contain:

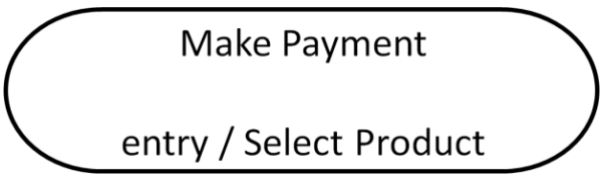
- Activity states and action states
- Transitions
- Objects

Action States and Activity States:

Activity diagram may represent actions which are atomic computations. These atomic computations are called action states because they are states of the system, each representing the execution of an action. In UML, an action states is represented using a lozenge symbol (rounded rectangle) as shown below:

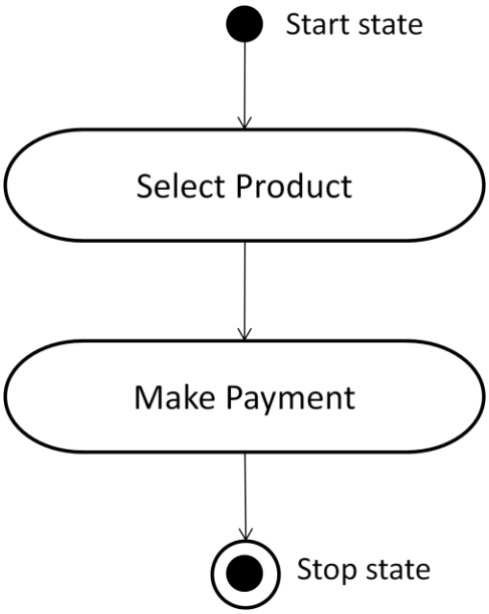


In contrast, activity states can be further decomposed, their activity being represented by other activity diagrams. Activity states are non-atomic and can be interrupted, considered to take some duration to complete. There is no difference in the notation of activity states and action states except that the activity states may have additional information like entry and exit actions as shown below:



Transitions:

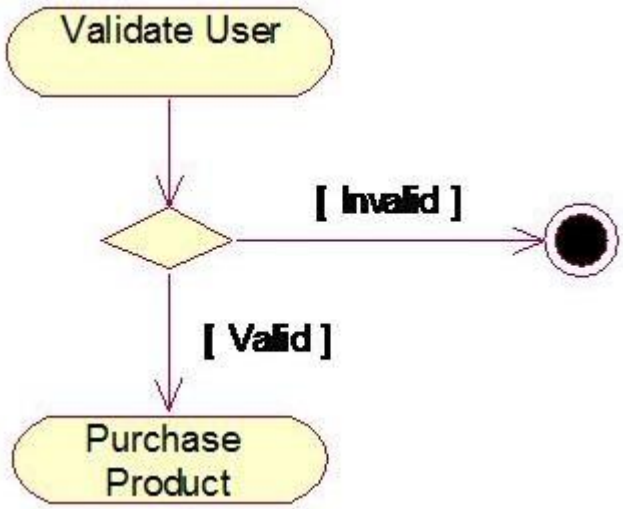
When the action or activity completes, the flow of control passes immediately to the next action or activity state. This flow is represented as transitions to show the path from one action or activity state to the next action or activity state. In UML, a transition is represented as a simple directed line.



A flow of control has to start and end someplace. Therefore, as shown in the above figure, we can specify the initial state (solid ball) and stop state (a solid ball inside a hollow circle).

Branching:

Like in flowcharts, we can include a branch, which specifies alternate paths taken based on some Boolean expression which is also known as a guard condition. In UML, a branch is represented as a diamond. A branch may have one incoming transition and two or more outgoing transitions. Across all these outgoing transitions, guards should not overlap, but they should cover all possibilities.

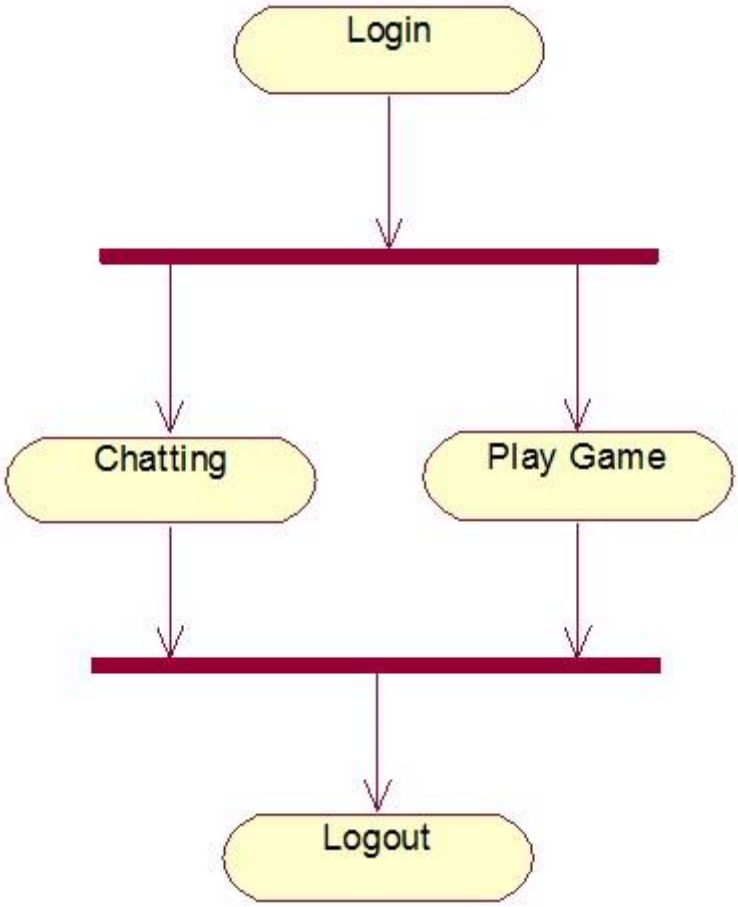


Forking and Joining:

While modeling flow of control, it is common to encounter flows that are concurrent or parallel. In UML, a synchronization bar is used to specify the forking and joining of these parallel flows of control. A synchronization bar is represented as a thick vertical or horizontal line.

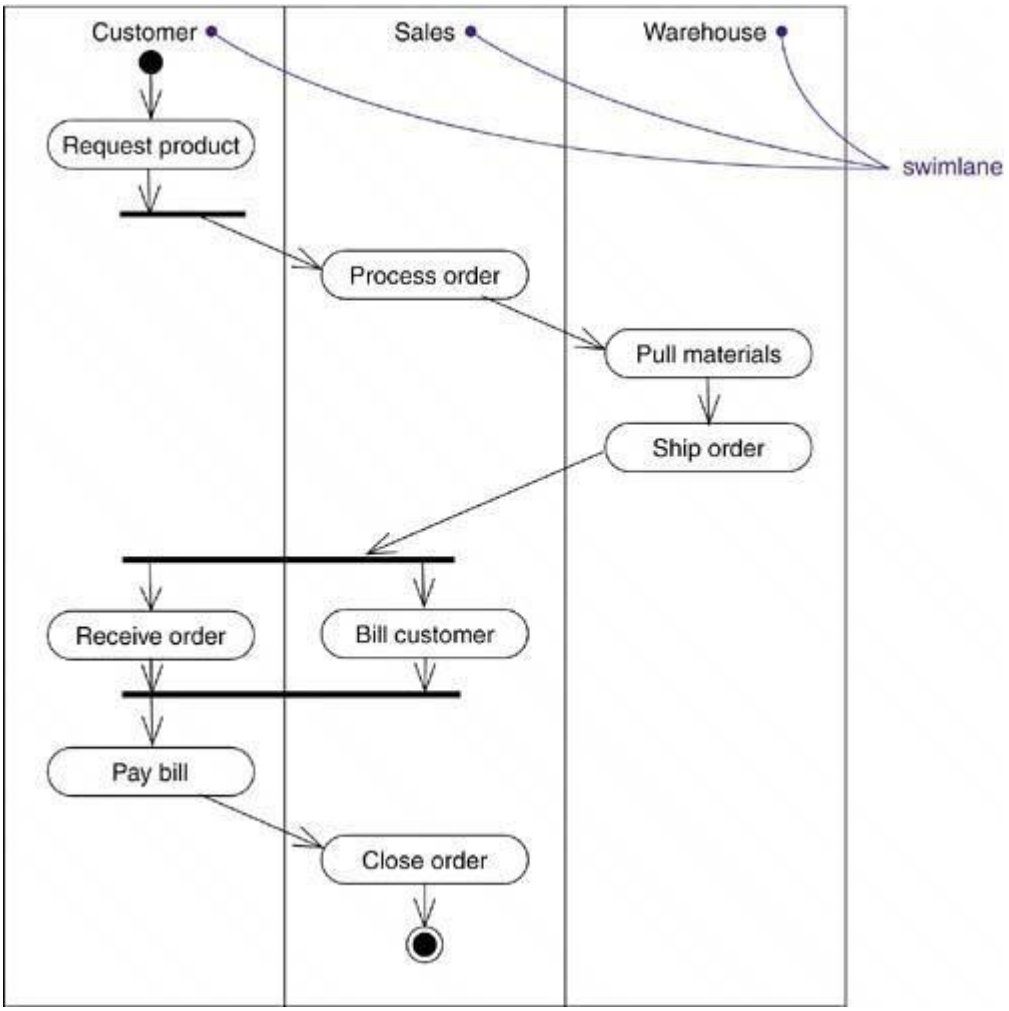
A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities in each flow carry out in parallel.

A join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition. At the join, all the concurrent flows synchronize, meaning that each flow waits for the other to join and continues on below the join.



Swimlanes:

In activity diagrams, the activity states can be divided into logical groups, each group representing the object responsible for the activities. In UML, each group is known as a swimlane because, visually, each group is divided from its neighbor by a vertical solid line as shown in the below figure. Each swimlane has its own name. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.



Common Uses:

When modeling the dynamic aspects of a system, you'll use activity diagrams in two of the following ways:

1. To model a workflow
2. To model an operation

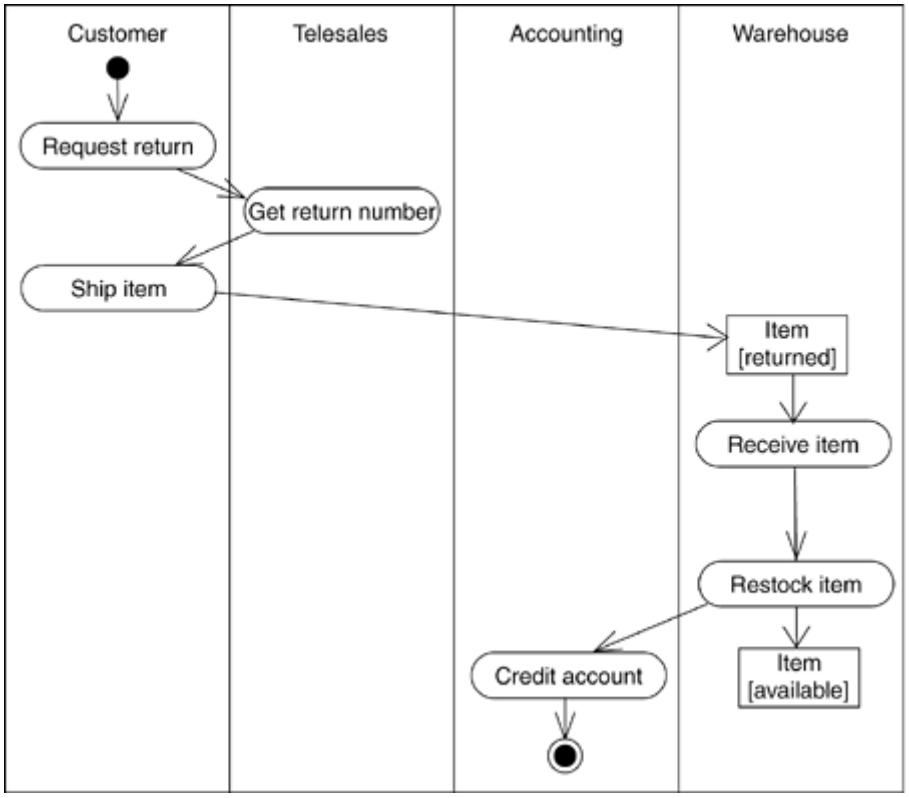
Common Modeling Techniques

Modeling a Workflow:

To model a workflow:

1. Establish a focus for the workflow.
2. Select the objects that have the high-level responsibilities for parts of the overall workflow. Create a swimlane for each of these objects.
3. Identify the pre-conditions of the workflow's initial state and the post-conditions of the workflow's final state.
4. Starting at the workflow's initial state layout the actions and activities that take place and render them as actions states or activity states.
5. For complex actions, or for sets of actions that appear multiple times, collapse them into activity states and provide a separate activity diagram for them.
6. Connect the action and activity states by transitions. Consider branching and then consider forking and joining.

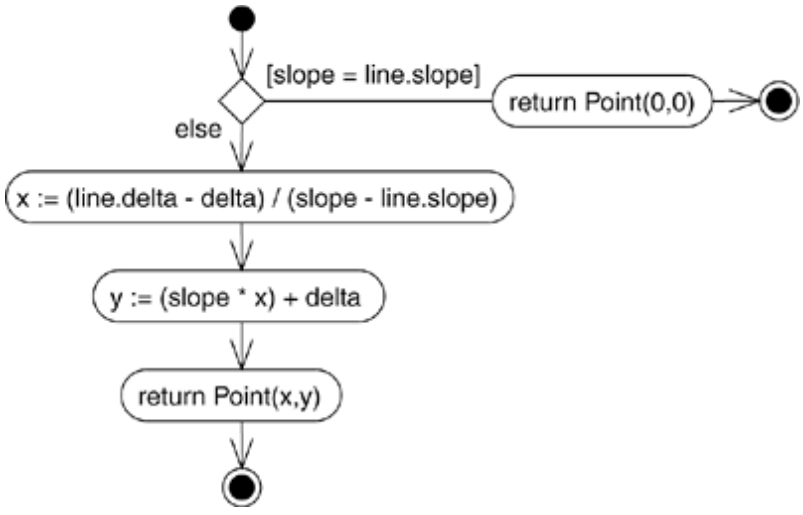
7. If there are important objects that are involved in the workflow, render them in the activity diagram as well.



Modeling an operation:

To model an operation:

1. Collect the abstractions involved in an operation like: parameters, attributes of the enclosing class and the neighboring class.
2. Identify the operation's pre-conditions and the operation's post-conditions.
3. Beginning at the operation's initial state, define the actions and activities and render them as action states and activity states respectively.
4. Use branching as necessary to specify conditional paths and iteration.
5. If this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.



UNIT – 5 : Advanced Behavioral Modeling

Syllabus :Events and signals,statemachines,processes and Threads ,time and space chart diagrams, Component, Deployment, Component Diagrams and Deployment diagrams

1.Events and Signals

Terms and Concepts

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

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

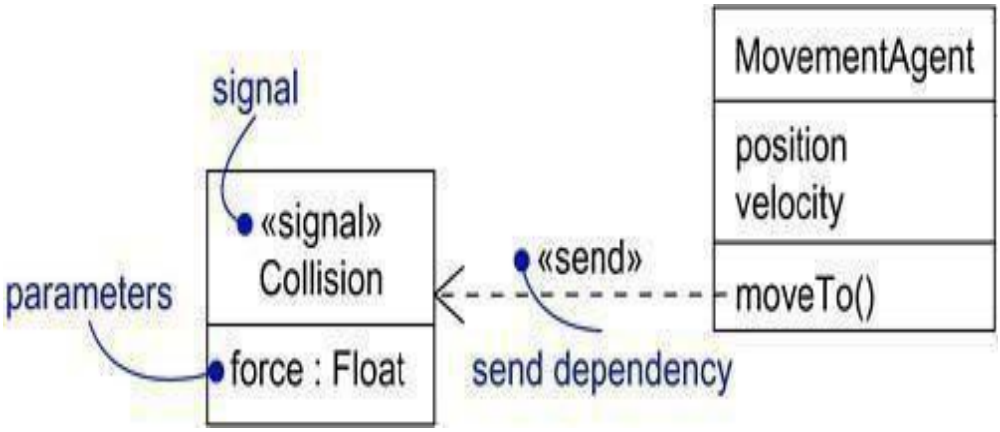
In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

Signals

- A signal represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another.
- Exceptions are supported by most contemporary programming languages and are the most common kind of internal signal that you will need to model.
- Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general and some of which are specific
- Also as for classes, signals may have attributes and operations.
- A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction. The execution of an operation can also send signals
- In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send.

- We model signals (and exceptions) as stereotyped classes. We can use a dependency, stereotyped as send, to indicate that an operation sends a particular signal

Figure Signals

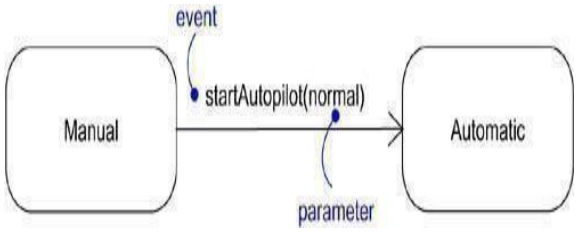

MovementAgent
position
velocity
moveTo()
signal
«signal»
Collision
force : Float
parameters
«send»
send dependency

Call Events

- Just as a signal event represents the occurrence of a signal, a **call** event represents the dispatch of an operation. In both cases, the event may trigger a state transition in a state machine
- Whereas a signal is an asynchronous event, a call event is synchronous
- This means that when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender.
- Modeling a call event is indistinguishable from modeling a signal event. In both cases, you show the event, along with its parameters, as the trigger for a state transition.

.

Figure Call Events



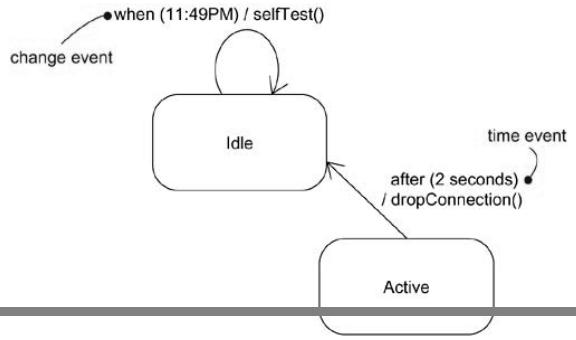
Time and Change Events

Call Events

Time and Change Events

- **A time event** is an event that represents the passage of time
- in the UML you model a time event by using the keyword after followed by some expression that evaluates to a period of time.
- Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.

- **A change event** is an event that represents a change in state or the satisfaction of some condition
- In the UML you model a change event by using the keyword when followed by some Boolean expression
- You can use such expressions to mark an absolute time (such as when time = 11:59) or for the continuous test of an expression



Sending and Receiving Events

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

Sending and Receiving Events

- Signal events and call events involve at least two objects:
    - The object that sends the signal or invokes the operation
    - The object to which the event is directed.
- Any instance of any class can send a signal to or invoke an operation of a receiving object.
- When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.
- Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous(assignation) for the duration of the operation.
- This means that the flow of control of the sender is put in lock step with the flow of control of the receiver until the activity of the operation is carried out.
- If this is a signal, then the sender and receiver do not rendezvous: the sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost
- In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class

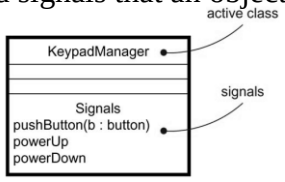
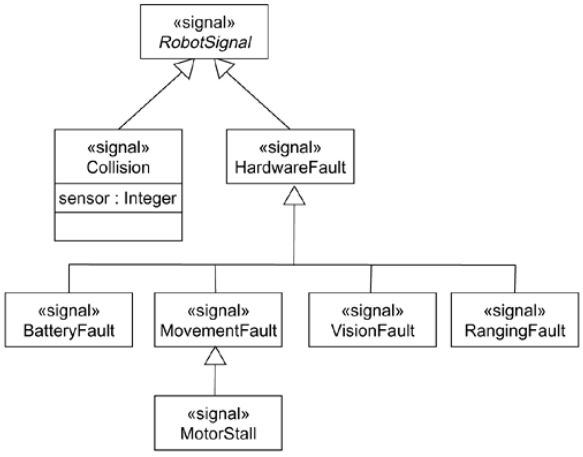


Fig:Signals and Active Classes

Common Modeling Techniques

Modeling a Family of Signals

- In most event-driven systems, signal events are hierarchical.
- External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations
- 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.



Modeling Abnormal Occurrence(Exceptions)

- An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the exceptions that its operations can raise.
- In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions may be attached to specification operations.
- Modeling exceptions is somewhat the inverse of modeling a general family of signals.
- We model a family of signals primarily to specify the kinds of signals an active object may receive
- We model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations

To model Abnormal Occurrence(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.

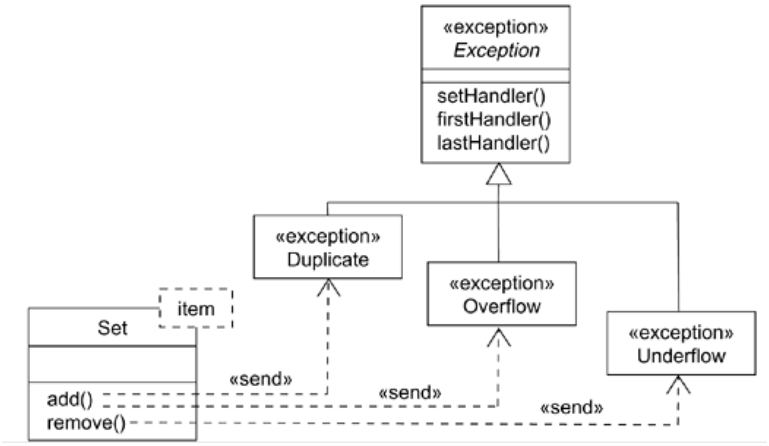


Fig: Modeling Exceptions

2. 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, together with its responses to those events
- **A state** is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- **An event** is the specification of a significant occurrence that has a location in time and space.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition
- **A transition** is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- **An activity** is ongoing nonatomic 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

Context

- Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist.
- In between, an object may act on other objects (by sending them messages), as well as be acted on (by being the target of a message).
- In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous stimuli communicated between instances.
- The behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past is best specified by using a state machine
- This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no state machine will simply ignore that signal.
- We'll also use state machines to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time.

- When an object's state machine is in a given state, the object is said to be in that state. A state has several parts.

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit actions	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates
5. Deferred events	A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

We represent a state as a rectangle with rounded corners.

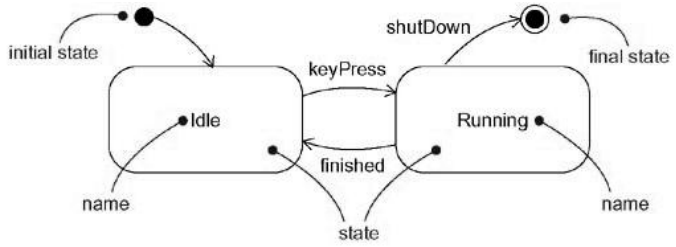


Fig: **States**

Initial and Final States

- There are two special states that may be defined for an object's state machine.
- **initial state**, which indicates the default starting place for the state machine or substate.
- An initial state is represented as a filled black circle
- **final state**, which indicates that the execution of the state machine or the enclosing state has been completed.
- A final state is represented as a filled black circle surrounded by an unfilled circle.

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.
- On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state.
- A transition has five parts

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied

2. Event trigger	The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
3. Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates True, the transition is eligible to fire; if the expression evaluates False, the transition does not fire and if there is no other transition that could be triggered by that same event, the event is lost
4. 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
5. Target state	The state that is active after the completion of the transition

- A transition is rendered as a solid directed line from the source to the target state.
- A self-transition is a transition whose source and target states are the same.

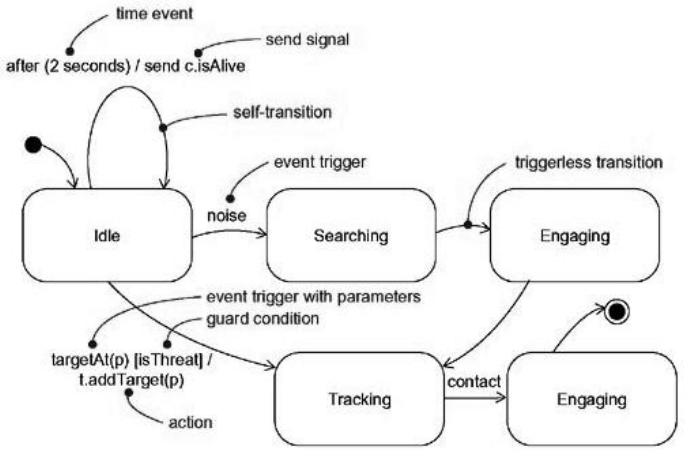


Fig:Transitions

Event Trigger

- An event is the specification of a significant occurrence that has a location in time and space.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- It is also possible to have a triggerless transition, represented by a transition with no event trigger. A triggerless transition—also called a completion transition—is triggered implicitly when its source state has completed its activity.

Guard

- A guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event.
- A guard condition is evaluated only after the trigger event for its transition occurs. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object

Action

- An action is an executable atomic computation. Actions may include operation calls), the creation or destruction of another object, or the sending of a signal to an object.
- An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion.

Advanced States and Transitions

- However, the UML's state machines have a number of advanced features that help you to manage complex behavioral models.
- These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines.
- Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events

Entry and Exit Actions

- In a number of modeling situations, you'll want to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to dispatch the same action no matter which transition led you away.
- UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry action (marked by the keyword event entry) and an exit action (marked by the keyword event exit), together with an appropriate action.
- Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

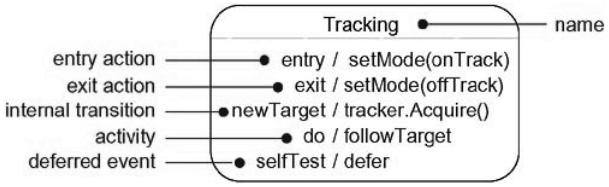


Fig:Advanced States and Transitions

Internal Transitions

- Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called **internal transitions**, and they are subtly different from self-transitions.
- In a self-transition an event triggers the transition, you leave the state, an action (if any) is dispatched, and then you reenter the same state.
- UML provides a shorthand for this idiom, as well (for example, for the event newTarget). In the symbol for the state, you can include an internal transition (marked by an event).

- Whenever you are in the state and that event is triggered, the corresponding action is dispatched without leaving and then reentering the state.
- Therefore, the event is handled without dispatching the state's exit and then entry actions.

Activities

- When an object is in a state, it generally sits idle, waiting for an event to occur.
- Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event.
- in the UML, you use the special do transition to specify the work that's to be done inside a state after the entry action is dispatched. The activity of a **do** transition might name another state machine
- You can also specify a sequence of actions—for example, do / op1(a); op2(b); op3(c).
- Actions are never interruptible, but sequences of actions are. In between each action (separated by the semicolon), events may be handled by the enclosing state, which results in transitioning out of the state.

Deferred Events

- In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out.
- However, in some modeling situations, you'll want to recognize some events but postpone a response to them until later.

In the UML, you can specify this behavior by using 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.

Substates

- There's one more feature of the UML's state machines—substates—that does even more to help you simplify the modeling of complex behaviors
- A simple state is a state that has no substructure.
- A substate is a state that's nested inside another one.
- A state that has substates— that is, nested states—is called a composite state.
- A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates.
- In the UML, you render a composite state just as you do a simple state, but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level

Sequential Substates

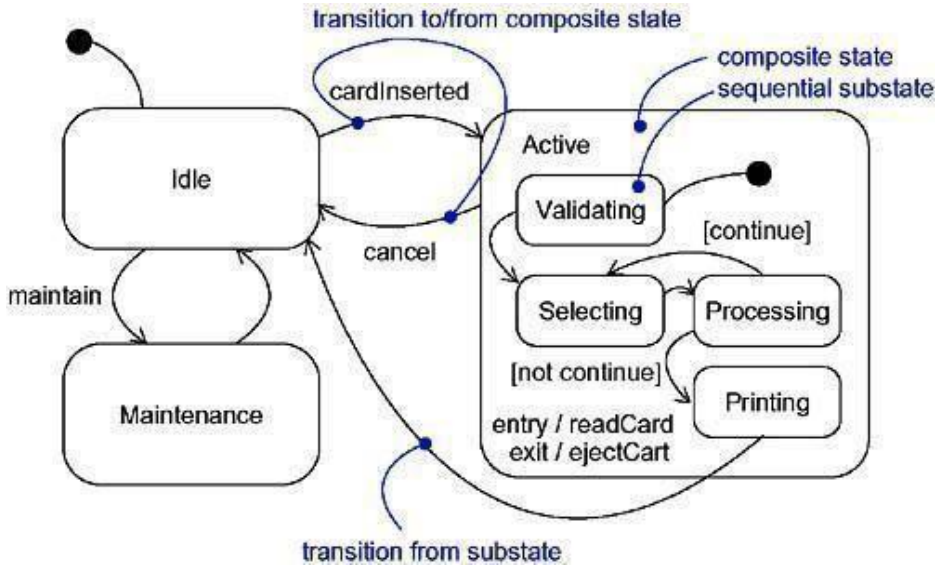
Consider the problem of modeling the behavior of an ATM. There are three basic states in which this system might be: **Idle** (waiting for customer interaction), **Active** (handling a customer's transaction), and **Maintenance** (perhaps having its cash store replenished). While **Active,** the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the **Idle** state. You might represent these stages of behavior as the states **Validating,**

Selecting,Processing, and **Printing.** It would even be desirable to let the customer select andprocess multiple transactions after **Validating** the account and before **Printing** a final receipt.

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

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

Figure SequentialSubstates



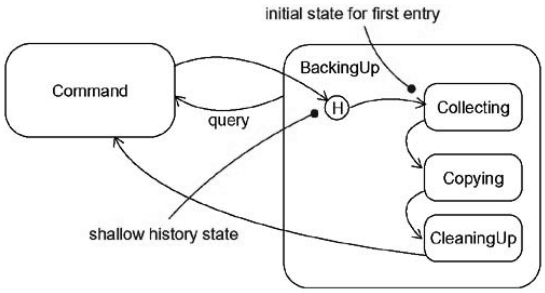
Notice also the transition from the **Active** state to the **Idle** state, triggered by the event **cancel.** In any substate of **Active,** the customer might cancel the transaction, and that returns the ATM to the **Idle** state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the **Active** state, no matter what caused a transition out of that

state). Without substates, you'd need a transition triggered by **cancel** on every substructure state.

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

History States

- A state machine describes the dynamic aspects of an object whose current behavior depends on its past.
- A state machine in effect specifies the legal ordering of states an object may go through during its lifetime
- When a transition enters a composite state, the action of the nested state machine starts over again at its initial state.
- However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.
- In the UML, a simpler way to model this idiom is by using history states.
- **A history state** allows a composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state
- As Figure shows, you represent a shallow history state as a small circle containing the symbol H.

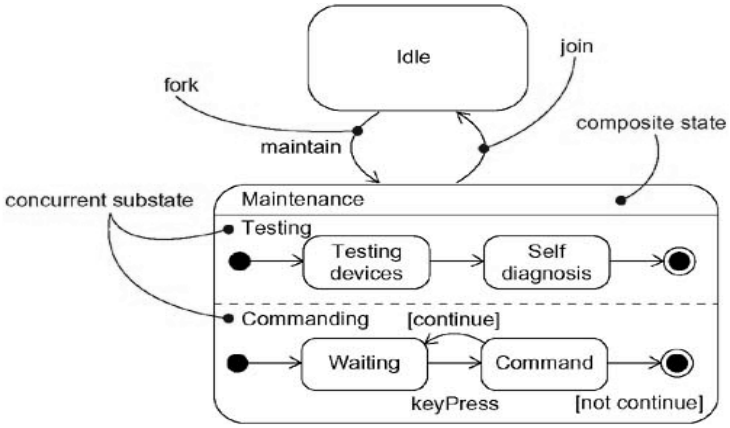


History State

- If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history.
- In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

Concurrent Substates(non orthogonal states)

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



Concurrent Substates

- Execution of these two concurrent substates continues in parallel.
- Eventually, each nested state machine reaches its final state. If one concurrent substate reaches its final state before the other, control in that substate waits at its final state.
- When both nested state machines reach their final state, control from the two concurrent substates joins back into one flow.
- Whenever there's a transition to a composite state decomposed into concurrent substates, control forks into as many concurrent flows as there are concurrent substates. Similarly, whenever there's a transition from a composite substate decomposed into concurrent substates, control joins back into one flow.
- This holds true in all cases. If all concurrent substates reach their final state, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Note: A nested concurrent state machine does not have an initial, final, or history state. However, the sequential substates that compose a concurrent state may have these features.

Common Modeling Techniques

Modeling the Lifetime of an Object

- When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior.

- Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction
- To model the lifetime of an object,
    - Set the context for the state machine, whether it is a class, a use case, or the system as a whole.

        1. 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.

        2. 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).

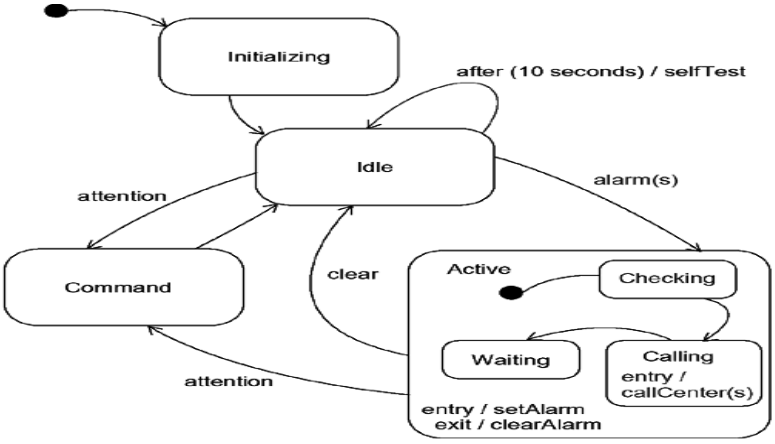
    - Expand these states as necessary by using substates.

    - 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.



3.Processes and Threads

- **An active object** is an object that owns a process or thread and can initiate control activity.
- **An active class** is a class whose instances are active objects
- **A process** is a heavyweight flow that can execute concurrently with other processes.
- **A thread** is a lightweight flow that can execute concurrently with other threads within the same process.
- Graphically, an active class is rendered as a rectangle with thick lines.
- Processes and threads are rendered as stereotyped active classes

Flow of Control

- In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another.
- A sequential program will process only one event at a time, queuing or discarding any concurrent external events.
- In a concurrent system, there is more than one flow of control—that is, more than one thing can take place at a time.
- In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread.
- If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

Note: You can achieve true concurrency in one of three ways:

- By distributing active objects across multiple nodes
- By placing active objects on nodes with multiple processors
- By a combination of both methods.

Classes and Events

- Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow.
- You use active classes to model common families of processes or threads.
- By modeling concurrent systems with active objects, you give a name to each independent flow of control.
    - When an active object is created, the associated flow of control is started;
    - When the active object is destroyed, the associated flow of control is terminated.
- Active classes share the same properties as all other classes.
- Active classes may have instances.
- Active classes may have attributes and operations.
- Active classes may participate in dependency, generalization, and association relationships.
- Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints.
- Active classes may be the realization of interfaces.
- Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines

Standard Elements

- All of the UML's extensibility mechanisms apply to active classes. Most often, you'll use tagged values to extend active class properties.
- The UML defines two standard stereotypes that apply to active classes.

1. process	Specifies a heavyweight flow that can execute concurrently with other processes
2. thread	Specifies a lightweight flow that can execute concurrently with other threads within the same process

- A process is heavyweight, Each program runs as a process in its own address space
- Processes are never nested inside one another. If the node has multiple processors, then true concurrency on that node is possible.
- If the node has only one processor, there is only the illusion of true concurrency, carried out by the underlying operating system.

- A thread is lightweight. the threads that live in the context of a process are peers of one another, contending for the same resources accessible inside the process.
- Threads are never nested inside one another.

Communication

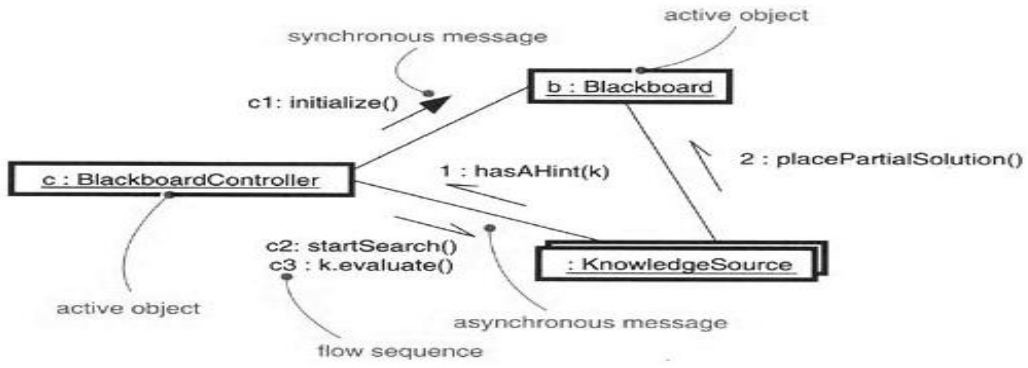
- When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider
    1. A message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.
    2. A message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication

        1.) One active object might synchronously call an operation of another.

        2.) One active object might asynchronously send a signal or call an operation of another object

        That kind of communication has mailbox semantics

- In the UML, we render a synchronous message as a full arrow and an asynchronous message as a half arrow



Communication

3. A message may be passed from an active object to a passive object. A difficulty arises if more than
   one active object at a time passes their flow of control through one passive object. In that situation,
   you have to model the synchronization of these two flows very carefully.

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

Synchronization

- The problem arises when more than one flow of control is in one object at the same time. If you are not careful, anything more than one flow will interfere with another, corrupting the state of the object
- This is the classical problem of mutual exclusion. The key to solving this problem in object-oriented systems is by treating an object as a critical region.
- There are three alternatives to this approach, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. Sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
2. Guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
3. Concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic

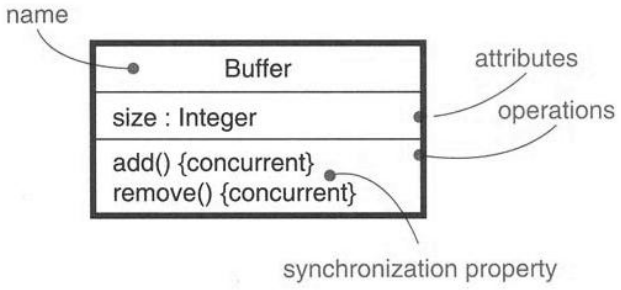


FIG:Synchronization

Process Views

- Active objects play an important role in visualizing, specifying, constructing, and documenting a system's process view.
- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view—that is, class diagrams, interaction diagrams, activity diagrams, and statechart diagrams,
- But with a focus on the active classes that represent these threads and processes.

Common Modeling Techniques

Modeling Multiple Flows of Control

Building a system that encompasses multiple flows of control is hard. Not only do you have to decide how best to divide work across concurrent active objects, but once you've done that, you also have to devise the right mechanisms for communication and synchronization among your system's active and passive objects to ensure that they behave properly in the presence of these multiple flows

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.

1. 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.
2. Capture these static decisions in class diagrams, explicitly highlighting each active class.
3. 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.
4. Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
5. Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

For example :find three objects that push information into the system concurrently: a StockTicker, an IndexWatcher, and aCNNNewsFeed(named s, i, and c, respectively). Two of these objects (sandi)communicate with their ownAnalystinstances (a1anda2).

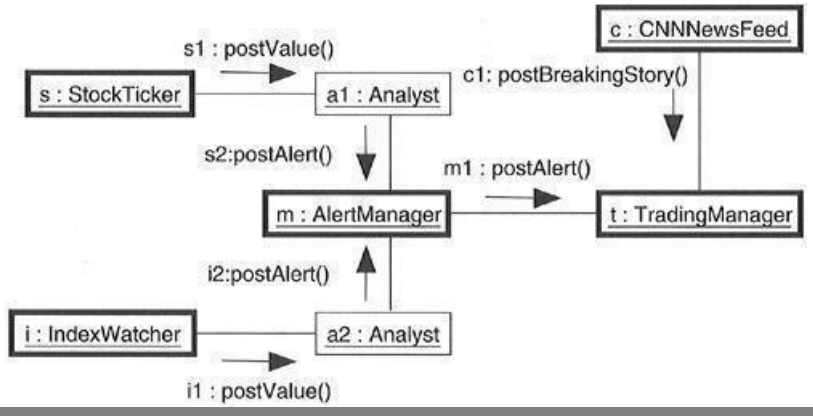


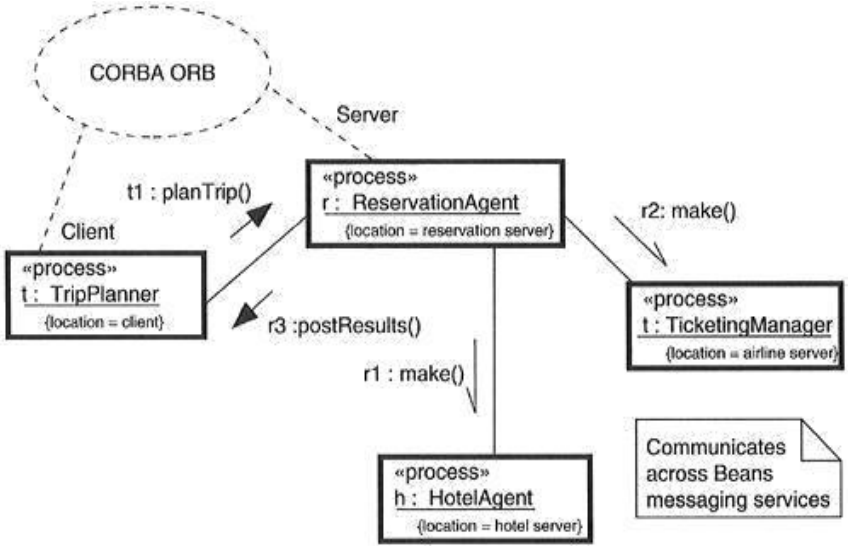
Figure Modeling Flows of Control

Modeling Inter process Communication:

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

To model interprocess communication,

1. Model the multiple flows of control.
2. Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
3. Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
4. Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations



4.Time and Space

Terms and Concepts

A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is formed as an expression from the name given to the message (which is typically different from the name of the action dispatched by the message). A *time expression* is an expression that evaluates to an absolute or relative value of time. A *timing*

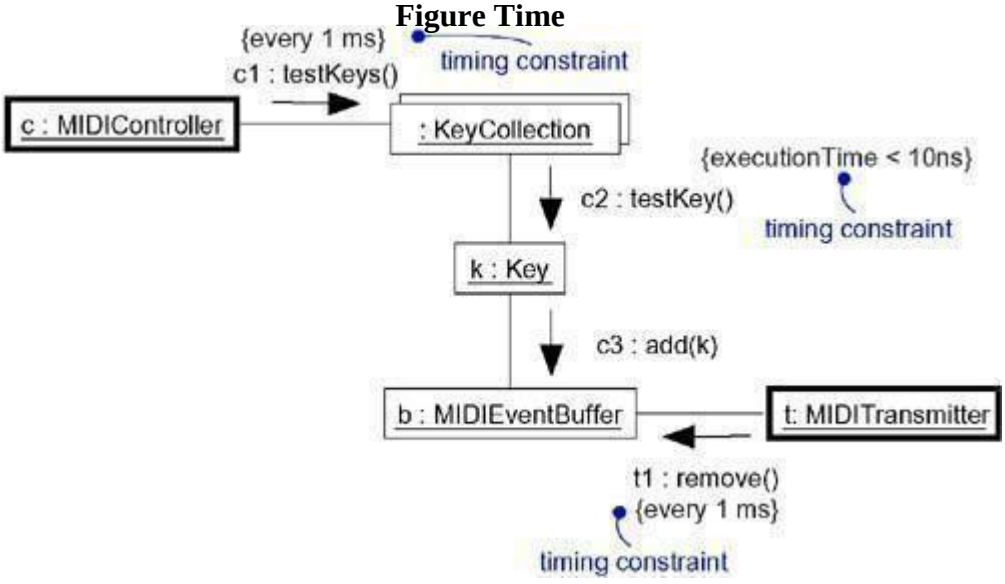
constraint is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is **rendered as for any constraint• that is, a string enclosed by bracke**ts and generally connected to an element by a

dependency relationship. *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged **value•** that is, a string enclosed by brackets and placed below an element's name, or as the nesting of components inside nodes.

Time

- Real time systems are, by their very name, time-critical systems.
- Events may happen at regular or irregular times;
- the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.
- The passing of messages represents the dynamic aspect of any system, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used as a timing mark.
- If the designated message is ambiguous, use the explicit name of the message in a timing mark to designate the message you want to mention in a time expression.
- A timing mark is nothing more than an expression
- formed from the name of a message in an interaction.
- Given a message name, you can refer to any of three functions **of that message• that is, startTime**,**stopTime**, and**executionTime.** You can then use these functions to specify
- arbitrarilycomplex time expressions,
- you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure Time



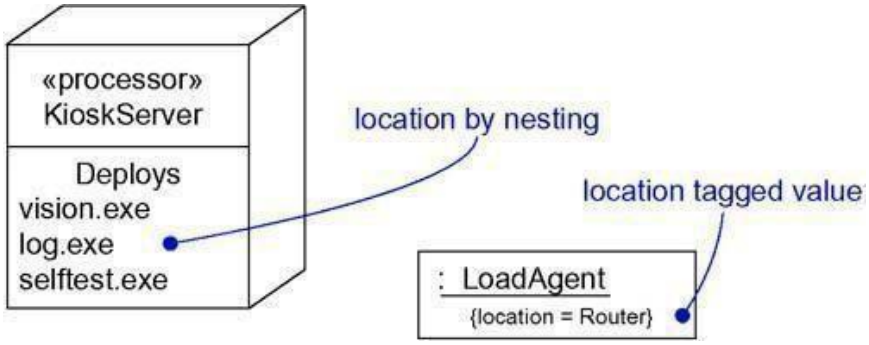
Location

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

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

For example, as Figure shows, the executable component **vision.exe** may reside on the node named**KioskServe**r.

Figure Location



You'll typically use the first form when it's important for you to give a visual cue in your diagrams about the spatial separation and grouping of components.

Similarly, you'll use the second form when modeling the location of an element is important, but secondary, to the diagram at hand, such as when you want to visualize the passing of messages among instances.

Common Modeling Techniques

Modeling Timing Constraints

Modeling the absolute time of an event, modeling the relative time between events, and modeling the time it takes to carry out an action are the three primary time-critical properties of real time systems for which you'll use timing constraints.

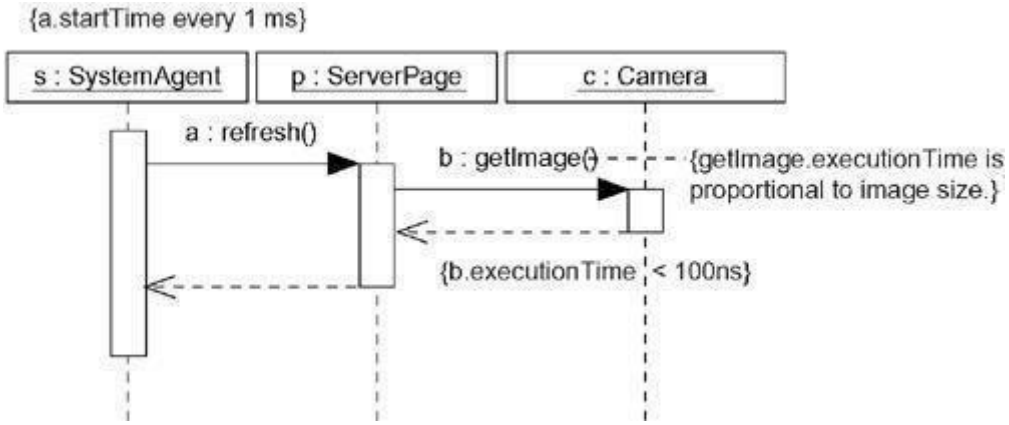
To model timing constraints,

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

- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.
- For each time critical operation in each class, consider its time complexity. Model those semantics as timing constraints on the operation.

For example, as shown in Figure the left-most constraint specifies the repeating start time the call event **refresh.** Similarly, the center timing constraint specifies the maximum duration for calls to **getImage.** Finally, the right-most constraint specifies the time complexity of the call event **getImage.**

Figure Modeling Timing Constraint



Modeling the Distribution of Objects

When you model the topology of a distributed system, you'll want to consider the physical placement of both components and class instances.
Deciding how to distribute the objects in a system is a wicked problem, and not just because the problems of distribution interact with the problems of concurrency.

To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words,

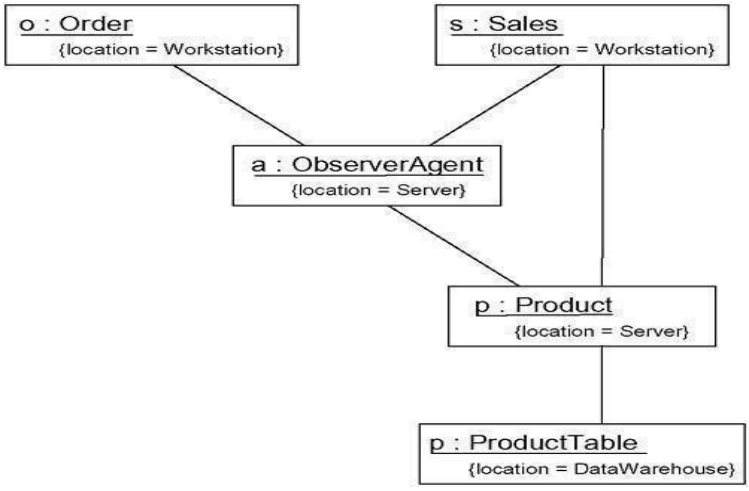
  consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus, there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.

- Next consider patterns of interaction among related sets of objects. Co-locate sets of objects that have high degrees of interaction, to reduce the cost of communication. Partition sets of objects that have low degrees of interaction.

- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Render this allocation in one of two ways:
    1. By nesting objects in the nodes of a deployment diagram
    2. By explicitly indicating the location of the object as a tagged value

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

Figure Modeling the Distribution of Objects



5.Statechart Diagrams

- A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state. A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- An *activity* is ongoing nonatomic 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. Graphically, a statechart diagram is a collection of vertices and arcs.

Common Properties

Statechart diagrams commonly contain

- Simple states and composite states
- Transitions, including events and actions distinguishes an activity diagram from a statechart diagram is that an activity diagram is basically a projection of the elements found in an activity graph, a special case of a state machine in which all or most states are activity states and in which all or most transitions are triggered by completion of activities in the source state.

You use statechart diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the event- ordered behavior of any kind of object in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.We use statechart diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach statechart diagrams to use cases (to model a scenario).

Common Modeling Technique

- To model reactive objects

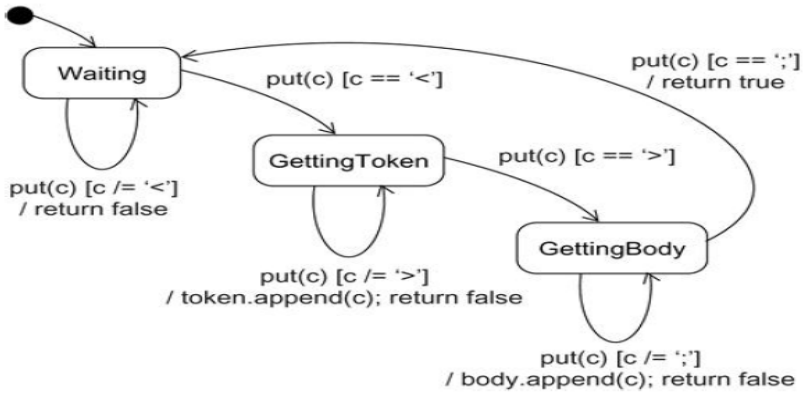
A reactive — or event-driven — object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the

next event. For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

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.

Figure Modeling Reactive Objects



Forward and Reverse Engineering

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

MessageParser.

```
classMessageParser {
  public boolean put(char
  c) { switch (state)
     {    case
      Waiti
      ng:   if
      (c   ==
      '<') {
        state = GettingToken;
        token = new
        StringBuffer(); body =
        new StringBuffer();
      }
      break;
    caseGettingToken : if
      (c == '>') state =
      GettingBody; else
      token.append(c);
    break;
    caseGettingBody :
      if (c == ';') state =
      Waiting; else
          body.append(c); return true;
  }
  return false;
 }
 StringBuffergetToken() { return token;
 }
 StringBuffergetBody() { return body;
 }
private
 final static int Waiting = 0; final static
 intGettingToken = 1; final static
 intGettingBody = 2; int state = Waiting;
 StringBuffer token, body;
}
```

Reverse engineering (the creation of a model from code) is theoretically possible, but

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