SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES, CHITTOOR
(AUTONOMOUS)

Approved by AICTE, New Delhi, Affiliated to JNTUA, Ananthapuramu.

LECTURE NOTES

Subject Name : OBJECT ORIENTED ANALYSIS AND DESIGN
Year / Branch :3-1 SEMESTER /COMPUTER SCIENCE AND ENGINEERING
Regulation :R23
Prepared BY: K. GEETHAVANI

COURSE OUTCOMES:

- To Understand the importance of modelling in UML.
- To Compare and contrast the object-oriented model with the E-R and EER models.
- To Design use case diagram. Design an application using deployment diagram.
- To Apply UML diagrams to build library application.
- To Design and interpret artifact diagrams to visualize software architecture and deployment.

Syllabus

UNIT -1: INTRODUCTION TO UML (9)

Introduction to UML: Importance of modelling, principles of modelling, object-oriented modelling, conceptual model of the UML, Architecture, Software Development Life Cycle.

UNIT -2: BASIC STRUCTURAL MODELLING (9)

Basic Structural Modelling: Classes, Relationships, common Mechanisms, and diagrams. Advanced Structural Modelling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages. Class & Object Diagrams: Terms, concepts, modelling techniques for Class & Object Diagrams

UNIT -3: BASIC BEHAVIOURAL MODELLING (9)

Basic Behavioural Modelling-I: Interactions, Interaction diagrams. Basic Behavioural Modelling-11: Use cases, Use case Diagrams, Activity Diagrams.

UNIT -4: ADVANCED BEHAVIORAL MODELLING (9)

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

UNIT -5: PATTERNS AND FRAMEWORKS (9)

Patterns and Frameworks, Artifact Diagrams. Case Study: The Unified Library application.

UNIT – I

Introduction

- Software development means developing a complete software system i.e., not simply coding or programming but it covers all possible problems domains it covers all levels of complexity
- Main aim of software development is illusion simplicity
- Complex systems are decomposed into smaller one during design process using algorithm and object oriented decomposition techniques
- Object model contain OOD & OOA
- Elements of this model are encapsulation, abstraction, modularity, hierarchy, typing, concurrency and persistence
- Software systems are of two types - simple, complex
- Many systems are complex by nature
- Simple systems have: limited purpose, short life span, used by same person, constructed for specific purpose, if any changes we use new software rather repairing, more difficult to develop
- Complex systems: exhibit rich set of behaviors, long life span, many uses operating systes, difficult to develop, can't handle by same person

Terminology used in object model:

- **Object Oriented Analysis(OOA):** It is a method of analysis that examine the requirements from problem domain. Analysis is process of extracting needs of system, what system must do to satisfy requirements
- **Object Oriented Design(OOD):** Aim of OOD is to design classes identified during analysis phase and user interaction

Applications of object model:

- It help you to exploit expressive power of object oriented language.
- Encourage reuse not only software but of entire design
- Use of object model produce system that are built upon stable intermediate forms
- Object model is applicable to wide variety of problem domains are given below
    - Air traffic control
    - Databases, Animation
    - Banking and insurance software
    - VLSI Design, Robotics
    - Image Recognition
    - Office Automation, Music Composition
- A Operating system, space craft & air craft simulation

Introduction to UML:

- The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.
- It is a graphical language which provides a vocabulary and set of semantics and rules
- The UML focuses on the conceptual and physical representation of the systems
- It captures the decisions and understanding about systems that must be constructed
- It is used to understand, design, configure, maintain and control information about systems
- It also captures information about the static structure and dynamic behavior of system
- Even though UML is not a programming language, its tools can provide code generators from UML into various object oriented programming languages

Object orientation:

- Object orientation is one of important concepts
- There are few basic principles behind all object oriented principles
- Key idea in any OOP is object
- Characteristics of object are:
    - It is closed or encapsulated; that is, it is a "thing in itself" separate from other things, revealing itself by its external behavior (which includes properties such as size, shape, color, motion, sound etc)
    - The internal structure of object is accessible only if it is "cut open" in some way
    - An object may be composed of other objects which determine how it behaves. Ex: car is composed of smaller objects
    - Although each object is unique, it is usually similar to the other objects that we can put them into same category or class. A 10 rupee note is unique from another 10 rupee note only by few parameters such as date, location etc. The particular rupee can be considered as instance of its class
- The general principles of object - orientation can be summarized as follows, based on the object properties discussed above.

Object identity:

- Every object has its own identity
- Object identity is the property by which each object (regardless of class and its present state) can be identified and treated as a distinct software entity
- An object retains its identity even if some or all the values of variables or definitions of methods change over time
- Object identity is strong notion of identity that typically found in programming languages not based on object orientation
- Several forms of identity are:

  Value: A data value is used for identity (ex, primary key of a table in a relational db)

  Name: A user supplied name is used for identifying. **ex.** var p1:person Here p1 is user defined name for an object

  Built in value: Object identity is typically implemented via a unique system generated OID(object identity which is also known as handle). The value of OID is not visible to external user, but is used internally by the system to identify each object uniquely and to create and manage inter object references

  Ex. var p1:person=person new

  The right side of this statement creates a new object of class person and this object is given a num, say some 420723 for identity. The handle (OID) is the identifier attached to an object when it is created

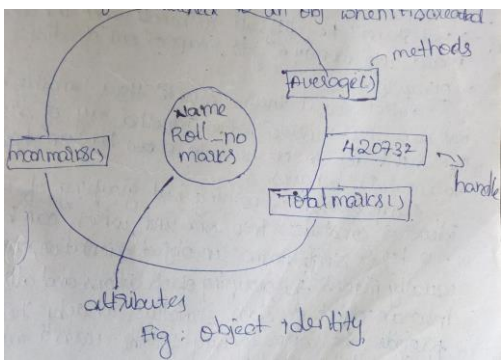


Fig : object identity

Two rules used for handles:

1. Same handle remains with object for entire life

2. No two objects have same handle. When an object is created a new handle is assigned to it or if a handle not being used currently by any object, then it can be assigned to new one.

var p1:person=person new

here p1 is used to hold the handle of the object created on right hand side

Here p1 is same as a pointer. P1 holds OID of an object. Even physical memory addresses of objects can be used as it handles. But it will be difficult to handle when the object is mould into memory or gets swapped out of disk

Encapsulation: Object reveal their "outside" through tangible behavior but keep their "inside" hidden. Another way of saying this is that we do not need to know how an object works to know how it behaves. Encapsulation is nothing but grouping of related ideas or constructs into one unit, which can be referred to by a single name. In object orientation, encapsulation usually means the grouping of operations and attributes into an object or class structure, whereby the operation provide the sole facility for the access or modification of the attribute values.

Information Hiding: An encapsulation technique whereby the encapsulated units externally visible interface suppresses certain information available within the unit

Polymorphism: The facility by which a single operation name may be defined upon more than one class and may denote different implementations(methods) in each of those classes

Genericity: It is the property of generic class which is an incomplete class specification with the formal parameter list the generic class is also called as parametrized class also in order to create an instantiable class matching set of actual parameters must be supplied to fill in the missing details of the class discussion.

1.1 Importance of Modeling:

- In software engineering we use models for visualizing and specifying the software product or applications so the designers use UML for this purposes
- We may also develop models from existing software to enable easier user understanding
- So in short, models can be used to specify, visualize, constructing and/or documenting a system
- Ex: 1) if we want to build dog house we can start with pile of lumber, some nails and a few endup with dog house. Unless house doesn't leak dog will be happy. If it doesn't work out we can always start over.
  2) If we want to build house if not enough start with lumber and nails. Here we need detailed planning before we lay foundation so we prepare some blue prints
  3) If we want to build high rise office building it is stupid to start with lumber, nails. Because here we use people's money and they will demand to have input into size, shape and style of building. So we need extensive planning because cost of failure is high
- There are many elements that contribute to a successful software organization. One common thread is use of modelling
- Modelling is proven and well accepted engineering technique. Sometimes we build mathematical models in order to analyze the effect wind (or) earthquakes on buildings
- Modelling differs from area to area. For ex in motion picture industry story boarding which is form modelling is center to any products. In field of sociology, economics we build models that we can validate our theories or try out new ones.
- All models have some set of common properties:
  - Model is simplification of reality
  - Model can be represented at different level of precision
  - We can choose which details to represent in a model and which to ignore

- A single model is not sufficient and we have set of nearly independent models

Model:

A model is a simplification of reality. A model provides the blueprints of a system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

Why do we model

- We build models so that we can better understand the system we are developing.
- Through modeling, we achieve four aims.
    1. Models help us to visualize a system as it is or as we want it to be.
    2. Models permit us to specify the structure or behavior of a system.
    3. Models give us a template that guides us in constructing a system.
    4. Models document the decisions we have made.
- We build models of complex systems because we cannot comprehend such a system in its entity.
- Software is complex and is composed of multiple interacting module and complexity can be handled using higher language and meta languages like XML
- Models are used to reduce complexity. Some of modeling techniques are flowcharts, state diagrams, E R diagrams. But they do not reflect object oriented. UML supports object oriented thinking
- While developing software it is divided into modules
- This follows divide and conquer approach

1.2. Principles of Modeling

There are four basic principles of model

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

   Choose models well wrong models will mislead you causing focus on irrelevant issues. If problem is in Quantum physics choose different model other than calculus so they use fennymann diagram. If you are constructing a new building and concern about how it behaves in high winds, construct a physical model and subject to tunnel tests.

   If we build through eyes of

   Database develop - focus on ER models and push to stored procedures

   Structured analyst - End up with models that are algorithmic centric

   Object oriented development - centred

   Ex: there might be different approaches to built a system through a DBA's view system analysts view, object oriented developers view

   Any of these approaches might be right for an application and developed culture
2. Every model may be expressed at different levels of precision.

   Sometimes model is just what you need other times we get down and dirty with bits. So the best kinds of models are those which let us choose any degree detail depending on who is viewing and why
3. The best models are connected to reality.

   All the models should have clear connection to reality. All models simplify reality. It is very important that simplification (or) simulation should not mark (or) hide important details
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

Ex: In building constructing, a number of blueprint like floor plans, elevation, electrical plans, plumbing plans etc., reveal all details.

1.3. Object Oriented Modeling

- In software, there are several ways to approach a model. The two most common ways are
    1. Algorithmic perspective
    2. Object-oriented perspective

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

2. Object-oriented perspective: In this the main building block of all software systems is the object or class object is thing and a class is a description of a set of common objects. Every object has an identity or name, state and behavior

Ex: Consider 3 tier architecture for billing system involving user interface, middleware and database. In user interface we have concrete objects such as button, menu and dialog box. In database we have concrete objects such as table represent entities. In middle layer we have objects such as Transitions and business rules. Most contemporary languages, OS and tools are object oriented in same fashion, to view world in terms of object.

An Overview of UML

- The Unified Modeling Language is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.
- The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

- **Visualizing** Through UML we see or visualize the existing system and ultimately we visualize how the system is going to be after. Unless we think or visualize we can't implement. UML helps to visualize how the components of system communicate and interacts with each other. Each and everything about system can be visualized through UML. Visualizing through UML makes model recreatable and easily interpretable even if one developer replaces another. The UML is more than just a bunch of graphical symbols.
- **Specifying** means building models that are precise, unambiguous, and complete. UML addresses the specification of all the important analysis, design, implementation decisions that must be made in developing and deploying a software system
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages through mapping a model from UML to a programming language like JAVA or C++ or VB. Forward and Reverse Engineering are possible through mapping that is, coding from a UML model into a programming language and reconstructing a model from an implementation into UML respectively. Apart from this UML permits direct execution of models, system simulation and the instrumentation of running systems.

- **Documenting** the deliverables of a project apart from coding are some artifacts which are critical in controlling, measuring, and communicating about system during its development via., requirements, architecture, design, source code, project plans, tests, prototypes, releases etc.,

Applications of UML:

Mainly used for software intensive systems

Enterprise information system, Banking & Financial Services, Telecommunications, Transportation, Defence, aerospace, Retail, Medical Electronics, Scientific, Distributed web based service.

1.4. Conceptual Model of UML:

- UML is a standard language for writing software blue prints
- There are three major elements in the conceptual model of the UML:
  1. UML's basic building blocks
  2. Rules that dictate how these building blocks may interact with each other
  3. Some common mechanisms that apply throughout the UML

1. Building blocks of UML:

- The building blocks of UML are further divided into three parts.
  - a. Things
  - b. Relationships: tie things together
  - c. Diagrams: group interesting collection of things

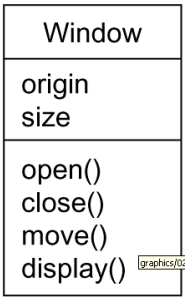
a. Things in the UML

- Things are first class citizens in model
- There are four kinds of things in the UML:
  - Structural things
  - Behavioral things
  - Grouping things
  - Annotational things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

1. Classes
2. Interfaces
3. Collaborations
4. Use cases
5. Active classes
6. Components
7. Nodes

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



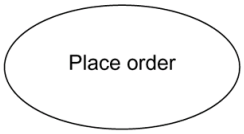
Interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specification(signatures) and not a set of operation implementations. An interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

Ispelling

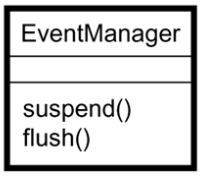
Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several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

Usecase

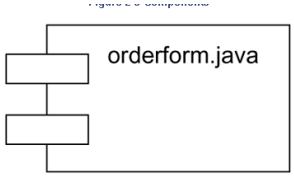
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse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



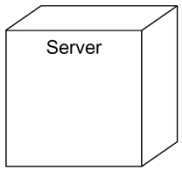
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs



Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of component may be in one node can move from one node to other. A set of components may reside on a node. Graphically, a node is rendered as a cube, usually including only its name



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

i. Interaction
ii. state machine

i. Interaction

Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. Behavior of objects are specified within the interaction. An interaction involves a number of other elements, including messages, action sequences and links. Graphically a message is rendered as a directed line, almost always including the name of its operation

ii. State Machine

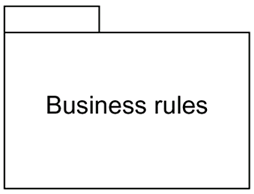
State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. Behavior of Single class or group of classes may be specified with state machine. State machine involves a number of other elements, including states, transitions, events and activities. Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

Grouping Things:-

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

Package:-

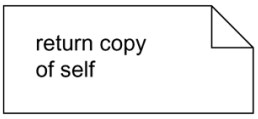
A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Package is purely conceptual. Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model. Annotational thing is note.

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment



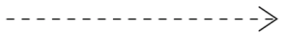
Relationships in the UML: There are four kinds of relationships in the UML:

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

Dependency:-

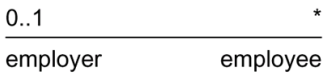
Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing

Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label



Association is a structural relationship that describes a set of links, a link being a connection among objects.

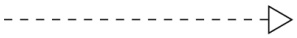
Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names. By using association you can navigate from an object of one class to an object of another class



Aggregation is a special kind of association, representing a structural relationship between a whole and its parts.

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

Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Realization relationship is in 2 places a. between interface and classes (or) components b. Between usecases and collaboration. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship



Diagrams in the UML

- When we model something, we create simplification of reality.
- So that we can better understand the system being developed.
- Using the UML, we build our models from basic building blocks, such as interfaces, collaborations, components, nodes dependencies, generalization and association.
- **Diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- Diagrams are used to visualize the system from different perspectives
- The UML defines a number of diagrams so that different aspects of system can be focused individually.
- In theory, a diagram may contain any combination of things and relationships.
- The static or structural part of system are viewed using one of the four following diagrams.

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram

- The dynamic or behavioral parts of a system are viewed using one of the following
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

Class diagram

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

Object diagram

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

Use case diagram

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

Interaction Diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. Interaction diagrams address the dynamic view of a system.

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

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

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other

Statechart diagram

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

Activity diagram

An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects. Activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.

Component diagram

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

Deployment diagram

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

Rules of the UML:

Building blocks are not enough. UML has number of rules that specify what a well formed model should look like. A well formed model is one i.e., semantically self consistent and an harmony with all its related models.

The UML has semantic rules for

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

A Model 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

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

Rules of UML encourage us but do not force us to address most important analysis, design and implementation over time.

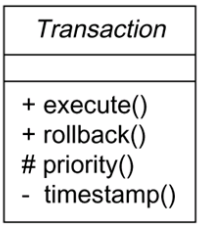
Common Mechanisms

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

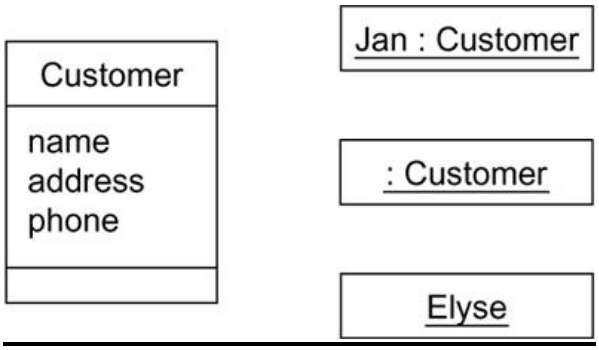
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specification: UML is a graphical language, for every graphical notion there is specification that provides textual statement of syntax and semantic of building blocks. Behind class icon is specification that provides full set of attributes, operations and behavior of that class. It is possible to build a model increment by drawing diagrams then adding semantics to the model specifications. UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. UML's diagram are simply visual properties into backplane.

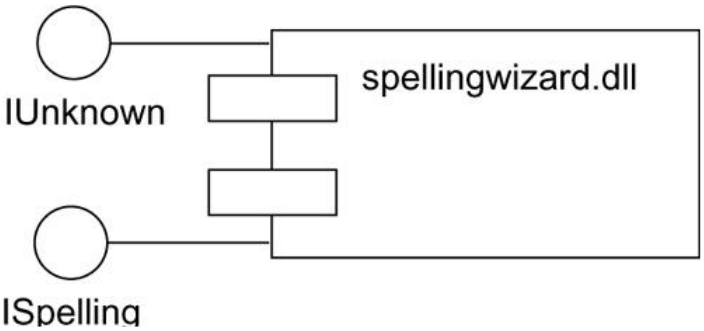
Adornments Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.



Common Divisions In modelling object oriented systems would gets divide in atleast. First division of class & object
Class - abstraction
Object - concrete manifestation of abstraction
We can model class as well as objects



We have same symbol for object and class but we orderly object name. There is separation of interface & implementation.



Extensibility Mechanisms

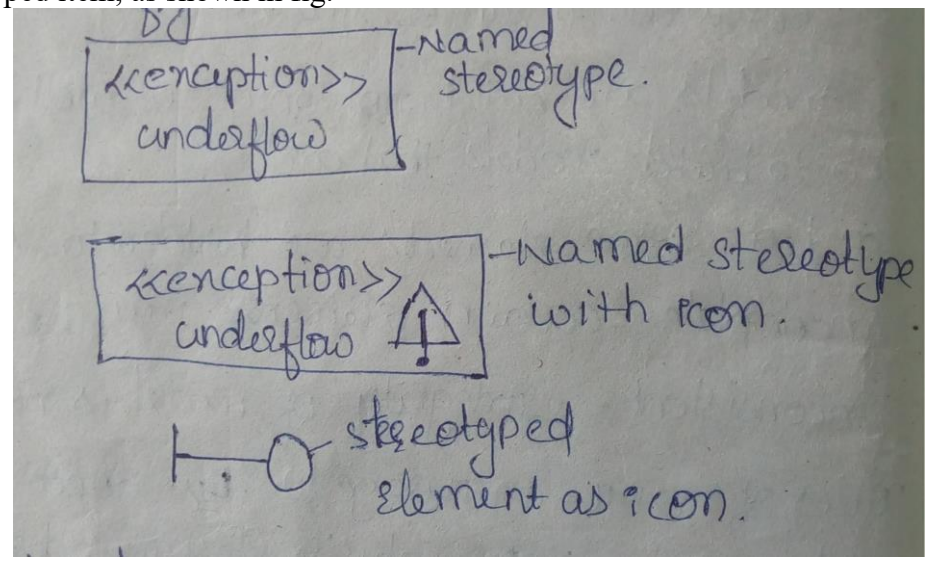
UML provide standard language for writing software blueprints. UML is open ended making it possible to extend in all ways. The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

Stereotype(new building blocks)

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. Stereotype is represented as a name enclosed by gulliments (<<name>>) and place over another element. It can be

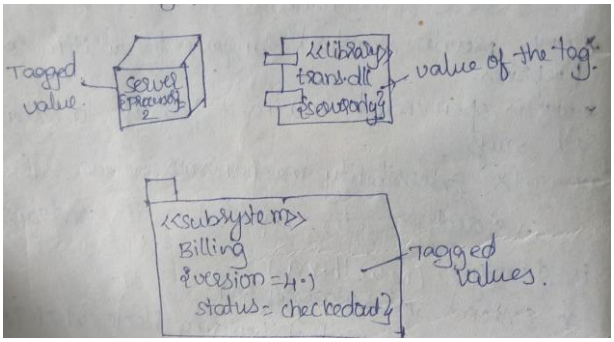
defined as icon, as a visual cue to the right of its name, or the icon can be used as the basic symbol of the stereotyped item, as shown in fig.



Tagged Values:(new property)

With stereotype new things can be added but with tagged values new properties can be added. For ex, when a project is released it is important to keep track version no. Here tagged values can be used to add this information to the models. The tag value applies to the element itself and not its instances. The fig. illustrates the concepts of tagged values. Tagged values are used to specify properties related to the coding. A tagged value is represented as a string enclosed by brackets and placed below the name of other element. The string is syntax is:

Name(tag), a separator(=), value



Constraints(new rules):

This is an extension of the semantics of a UML element, which allows the addition of new rules or modification of existing areas. In order for a system model to be well formed the conditions specified by the constraints should hold true.

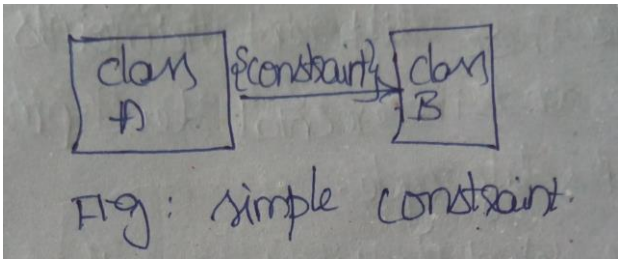
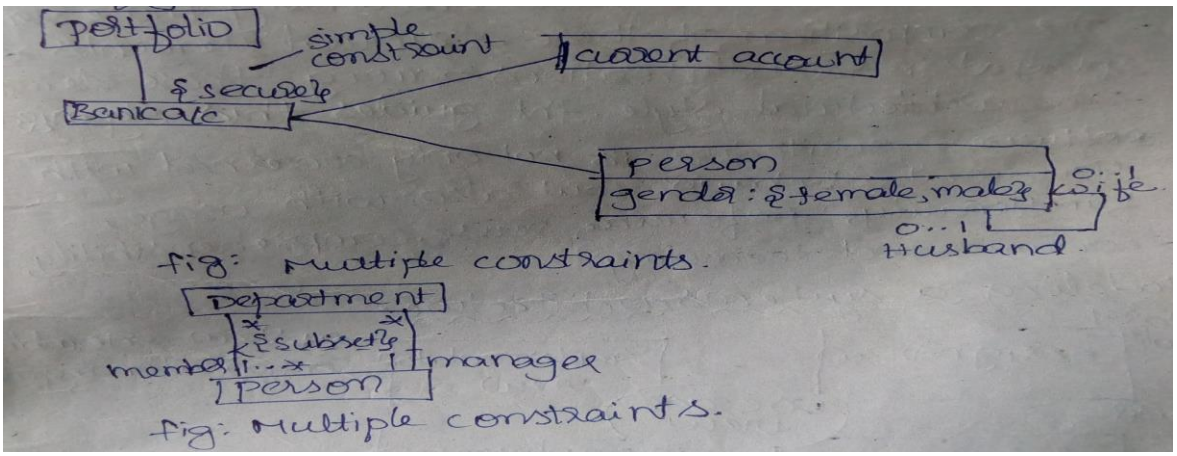


Fig: simple constraint.

When more precise specification of semantics is needed, the UML's Object Constraint Language (OCL) is used. As constraint is represented as a string enclosed by brackets and placed near the association, as shown in fig. below

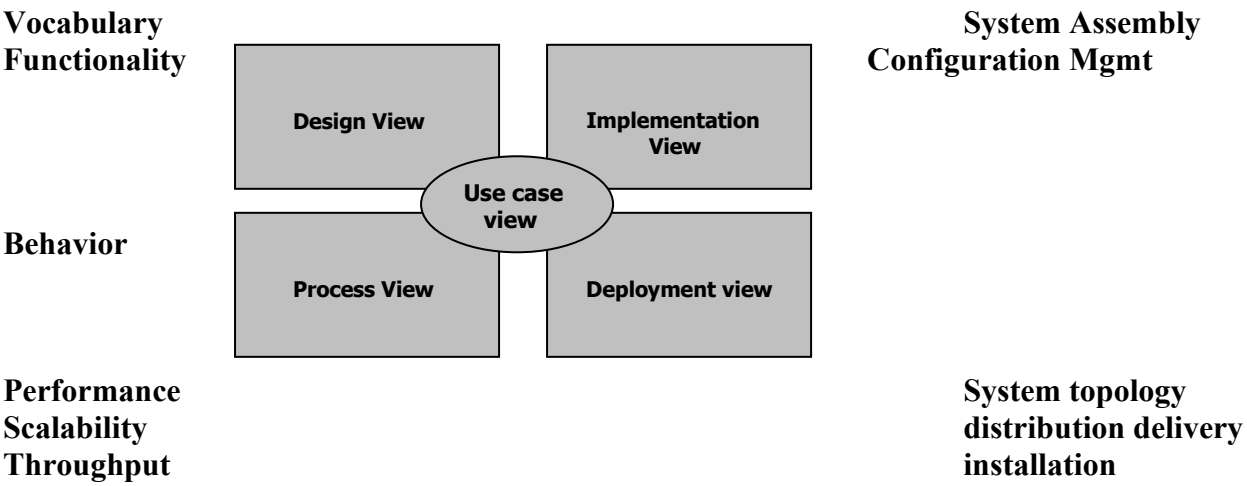


1.5. Architecture

For visualizing, specifying, constructing and documenting a software interactive system is viewed in number of perspectives. Different people (analyst, developer, system integration) bring different agendas to project. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle. Architecture is the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition. tware architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

Modeling a System's Architecture



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

Use case view

The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. It specifies the forces that shape the systems architecture. Doesn't really specify organization of software. Static views are captured in usecase diagram. The dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Design View

The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. Static view are captured in class & object diagram and dynamic view are captured by interaction, statechart, activity diagram.

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. Static and dynamic views are captured in same diagram.

Implementation View

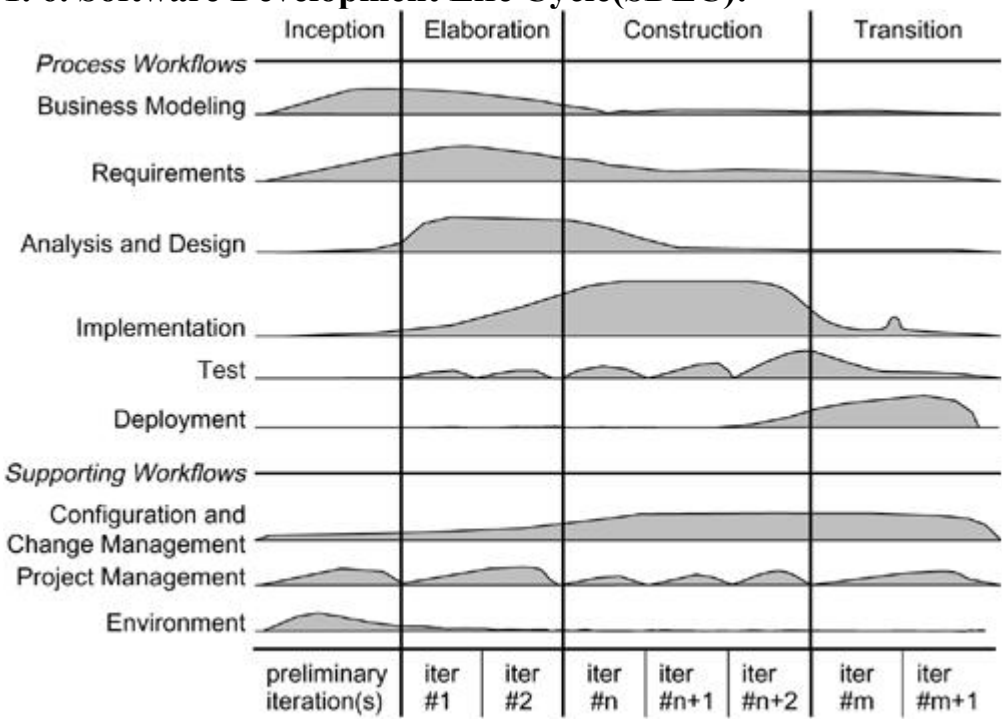
The implementation view of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. Static views are captured in component diagram and dynamic view are captured in state chart, interaction, activity diagram.

Deployment view

The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and

installation of the parts that make up the physical system. Static views are captured in deployment diagram and dynamic view are captured in interaction, statechart and activity diagram.
5 views interact with one another
Usecase - shape systems architecture
Design - support for requirements
Process - focus on performance, scalability, throughput
Implementation - focus on configuration management
Deployment - focuses on installation, distribution and delivery of parts.

1. 6. Software Development Life Cycle(SDLC):



UML is not tied to any particular SDLC but we should consider a process i.e., it is process independent. The three main aspects of this process are:

- Usecase driven
- Architecture centric
- Iterative & Incremental - risk driven

Usecase driven:
Here usecases are used as primary artifact which is used for verifying and validating system architecture, testing and for communicating among stakeholders
Architecture centric:
System architecture is used as primary artifact, for constructing, managing and evolve system under development.
Iterative and Incremental:
Managing stream of executable releases. Involves continuous integration of system to provide releases. Risk driven because each new release is focused on attacking and reducing most significant risk to the success of project. All these driven can be broken into phases. A phase is span of time between two major milestones of process. There are four phases in SDLC

- Inception

- Elaboration
- Construction
- Transition

Inception:
First phase of process. Basic idea is brought up for development.
Elaboration:
Second phase of process - when product division and architecture are defined. Here system requirement are articulated, prioritized and baselined. Requirement may range from general vision statement to precise evaluation criteria.
Construction:
Third phase of process. When software is brought from an executable architecture baseline to bring ready to the user community. Requirements and its evaluation criteria are constantly reexamined against business needs of the project and resources are allocated appropriately to attack risks to the project.
Transition:
Fourth phase of process. When software goes into the hands of user community. The software development process does not end here, the system is continuously improved, bugs are removed and few features are added.

The SDLC involves a continuous stream of executable releases of the system's architecture.

Questions Section

a) Short Questions (2 Marks)

1. Define a class in UML.
2. What is an object?
3. Differentiate between a class and an object.
4. Define association.
5. What is aggregation?
6. Define composition.
7. What is dependency?
8. Define generalization.
9. What is realization?
10. What are common mechanisms in UML?
11. What are stereotypes?
12. What are tagged values?
13. What are constraints?
14. What is an interface?

b) Essay Type Questions (10 Marks Level)

1. Define UML and explain its objectives, features, and applications in software engineering with suitable examples.
2. Explain the importance of modeling in software development. Discuss how modeling improves software quality, communication, and maintenance.
3. Describe the principles of modeling in UML with appropriate examples.

4. Explain object-oriented modeling and discuss its advantages over traditional structured modeling approaches.
5. Describe the conceptual model of UML. Explain its three major components with neat illustrations.
6. Explain the building blocks of UML and discuss how they contribute to software system development.
7. Discuss the architecture of UML and explain the role of each architectural view in software design.
8. Explain the Software Development Life Cycle (SDLC). Discuss how UML supports each phase of the SDLC.
9. Compare object-oriented modeling with conventional software development methodologies. Highlight their advantages and limitations.
10. Explain the role of UML as a standard modeling language in object-oriented software development.
11. Discuss the importance of UML in requirement analysis, system design, implementation, testing, and maintenance.
12. Explain how UML improves communication among developers, designers, customers, and stakeholders during software development.

C) Case Study-Based Questions (10 Marks Level)

Case Study 1 – Online Banking System

A bank intends to develop an online banking application that allows customers to create accounts, transfer funds, pay bills, and check account balances.

Questions

1. Explain how UML can be used throughout the SDLC for developing the system.
2. Identify the UML diagrams suitable for requirement analysis and system design.
3. Explain the importance of modeling before implementation.
4. Discuss how object-oriented modeling benefits this application

Case Study 2 – Hospital Management System

A hospital plans to computerize patient registration, doctor scheduling, billing, pharmacy, and laboratory services.

Questions

1. Explain the need for UML in developing the system.
2. Identify the SDLC phases involved.
3. Describe how the conceptual model of UML helps represent the system.
4. Justify the importance of modeling in reducing software complex

D) Design-Oriented / Higher Order Thinking Skills (HOTS) Questions

1. Design a UML-based development strategy for an Online Examination System by identifying the appropriate UML diagrams required at each SDLC phase.
2. Analyze a manual library system and propose how UML can be used to transform it into an object-oriented software system.
3. Compare object-oriented modeling with structured modeling and recommend the most suitable approach for developing a Hospital Management System.
4. A software company plans to develop a Smart City Management System. Design the UML modeling process from requirement analysis to deployment and justify the UML diagrams selected.

5. Evaluate the importance of modeling in reducing project risks and improving software quality using a real-world software project as an example.
6. A startup wants to develop a Food Delivery Application within a short development cycle. Recommend an SDLC model and explain how UML can support each development phase.
7. Analyze the role of UML in improving communication among developers, testers, customers, and project managers during software development.
8. Develop a UML-based software development roadmap for an Airline Reservation System and justify the use of different UML diagrams.
9. Examine the challenges faced in software development without modeling and propose UML-based solutions to overcome them.

References:

1. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modelling Language User Guide, Pearson Education 2nd Edition.

2. J Object-Oriented Analysis and Design with the Unified Process By John W Satzinger, Robert B Jackson and Stephen D Burd, Cengage Learning.

Expected Outcomes

a) Promote Conceptual Understanding

- Understand the fundamentals of Unified Modeling Language (UML) and its significance in software engineering.
- Explain the importance of modeling, object-oriented modeling concepts, the conceptual model of UML, UML architecture, and the Software Development Life Cycle (SDLC).
- Recognize the role of UML as a standard language for visualizing, specifying, constructing, and documenting software systems.

b) Encourage Analytical and Critical Thinking

- Analyze software development problems and identify appropriate UML concepts for representing system requirements.
- Compare object-oriented modeling with traditional software development approaches and evaluate their effectiveness.
- Assess the role of UML in improving software quality, maintainability, communication, and project management.

c) Align with Bloom's Taxonomy

- **Remember:** Recall UML terminology, principles of modeling, SDLC phases, and UML architecture.
- **Understand:** Explain the conceptual model of UML and the importance of modeling in software development.
- **Apply:** Select appropriate UML concepts and diagrams for different phases of software development.
- **Analyze:** Compare various software development approaches and examine the role of UML in system analysis and design.
- **Evaluate:** Assess the effectiveness of UML in addressing real-world software engineering challenges.

- **Create:** Propose UML-based solutions and modeling strategies for software development projects.

d) Support Semester-End Examination Preparation

- Provide comprehensive explanations of fundamental UML concepts supported by illustrations, examples, and summary tables.
- Include short-answer questions, descriptive questions, case studies, and Higher Order Thinking Skills (HOTS) questions aligned with university examination patterns.
- Strengthen students' conceptual understanding and problem-solving skills for semester-end examinations.

e) Integrate Theory with Practical Relevance

- Demonstrate the application of UML concepts through real-world case studies such as Online Banking, Hospital Management, Library Management, College Management, and E-Commerce Systems.
- Enable students to relate theoretical concepts of UML and object-oriented modeling to practical software development scenarios across different phases of the SDLC.
- Prepare students to apply UML techniques effectively in software analysis, design, documentation, and project development.

UNIT – II

Basic Structural Modeling & Advanced Structural Modeling

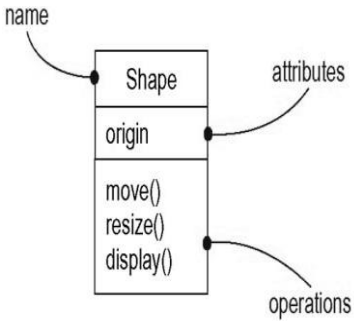
Basic Structural Modeling: Introduction:

- Basic structural modeling deals with the selection of structural elements and their interface by which system is composed
- There structural elements include, classes, relationships, common mechanisms and various diagrams
- More advanced structural modeling provides more in-depth information about the classes, relationships, interfaces, roles, packages and instances
- Common modeling techniques of structural element gives how to model these elements under various situations.

Class

- Classes are important building block of any object oriented system.

- A Class is a description of set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces.
- Graphically, a class is rendered as a rectangle
- Classes are used to capture vocabulary of a system to represent software things and hardware things
- A class is abstraction of things that are part of your vocabulary
- Class is not individual object but represent whole set of objects



Graphical Representation of Class in UML

Terms and Concepts

Names

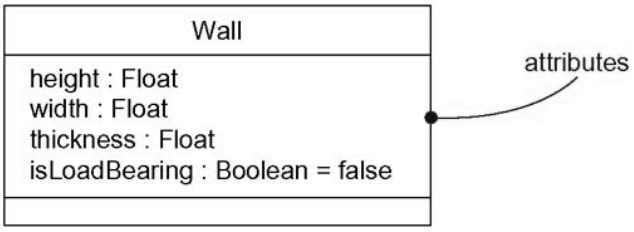
Every class must have a name that distinguishes it from other classes.

A name is a textual string and can be a simple name or a path name which is 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
- An attribute can be given by stating its class and default initial value
- Graphically attributes are listed just below class name

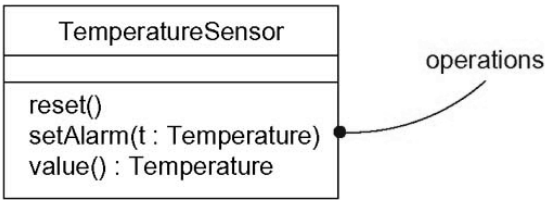


Attributes and Their Class

Operations

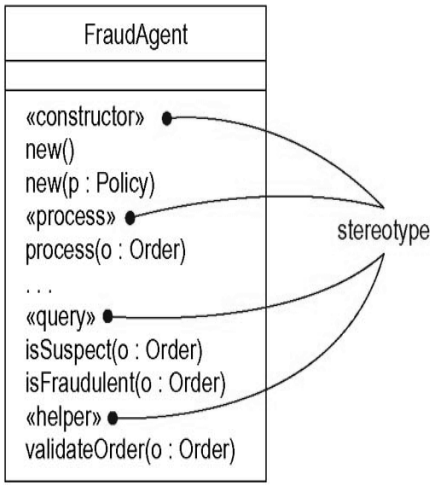
- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.

- 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, type, and default value of all parameters and a return type



Organizing Attributes and Operations:

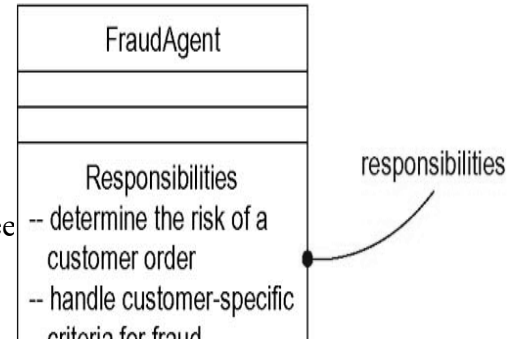
- No need to show every attribute and every operation at once.
- So we can elide(show only some (or) none of class attributes and operations) a class
- An empty compartment doesn't means that there are no operations it is just we didn't choose to show
- We can explicitly specify that there are more attributes and are shown by ending each list with ellipsis ("...")
- To better organize long lists of attributes and operations, prefix each group using stereotypes



Stereotypes for class features

Responsibilities

- A Responsibility is a contract(duty) or an obligation of a class. All object of class has some kind of behavior and state.
  Ex: Fraud agent - responsible for processing orders
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may have at least one responsibility or any number of responsibilities
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon



Other features:

- Attribute, operations and responsibilities are most common features are useful in building models of your classes.
- We could even specify other features such as visibility of individual attributes and operations, whether language is polymorphic (or) constant, exceptions that object of class might handle.

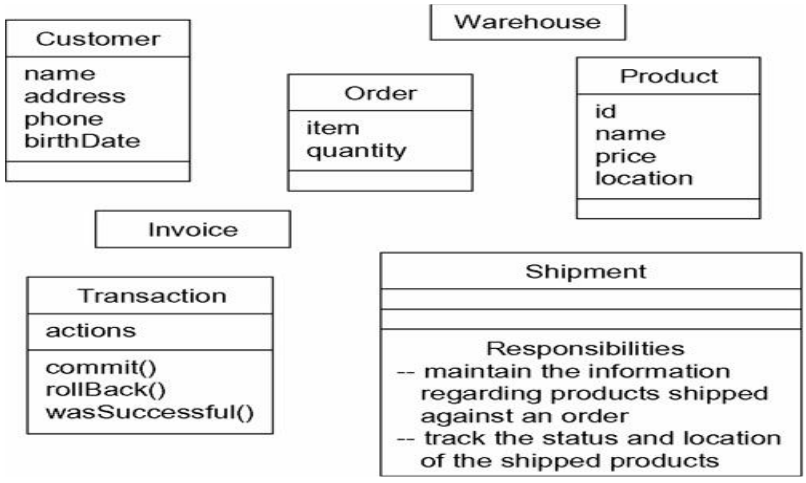
Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly used to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- Each of these abstractions is part of vocabulary of system

To model the vocabulary of a system

- Identify those **things** that users or implementers use to describe the problem or solution.
- For each abstraction, identify a **set of responsibilities** and there should be good responsibility to balance them among classes
- Provide the **attributes and operations** that are needed to carry out these responsibilities for each class.



Modeling the Distribution of Responsibilities in a System

- Once you start modeling, you should see your abstractions provide a balanced set of responsibilities.

To model the distribution of responsibilities in a system

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole,
  i) Split classes that have too many responsibilities into smaller abstractions

ii) Collapse tiny classes that have trivial responsibilities into larger ones, and
iii) 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 so that no class within collaboration does too much or too little.

Ex: Distribution of responsibilities among model, view and controller classes. Here all classes work together.

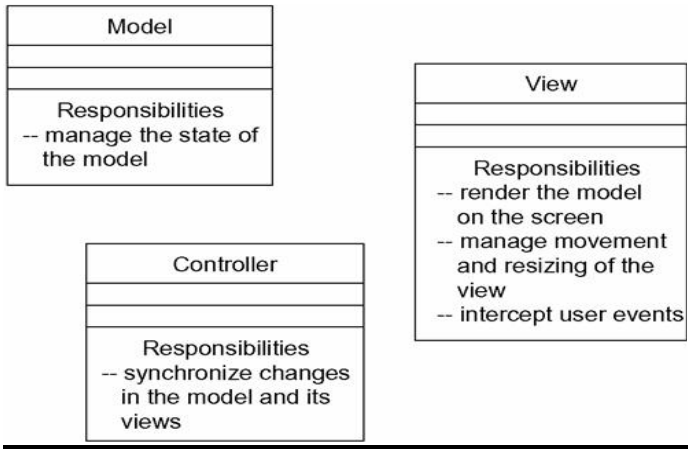


Fig. Modeling the distribution of responsibilities in a system

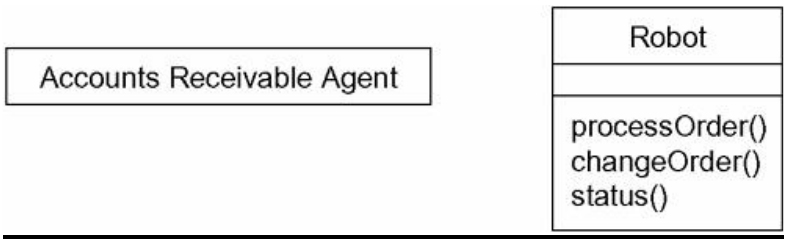
Modeling Nonsoftware Things

- Sometimes, the things you model may never have an analog in software
- For ex, who send invoices and robots that automatically package order for shipping from a warehouse which is a part of real system.
- Your application might not have any software that represents them

To model nonsoftware things

- Model the thing you are abstracting as a class.
- 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.

Ex:



Modeling Primitive Types

- Things you model may be drawn directly from the programming language you are using to implement a solution.
- Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types

To model primitive types

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

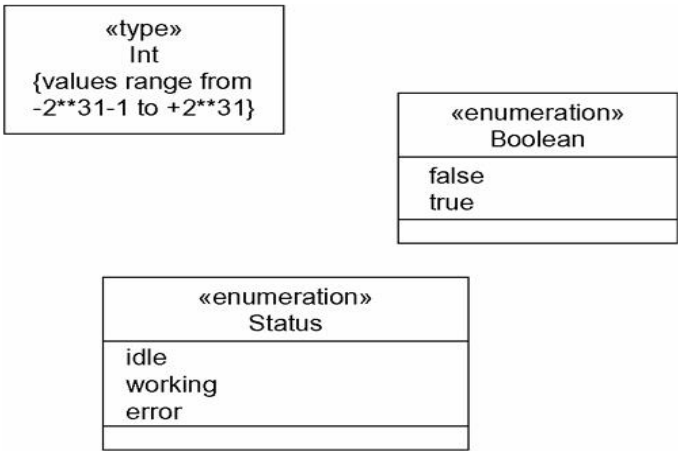


Fig. Modeling primitive types

- Here things are represented just like classes but are explicitly marked via stereotypes
- Things like integer is represented by class Int, Boolean and states are modeled a enumerates with their individual value provided as attributes

Relationships

- In the UML, things can connect to one another using relationships.
- When a system is modeled we identify vocabulary of system. Then we observe how the elements are interconnected to one another.

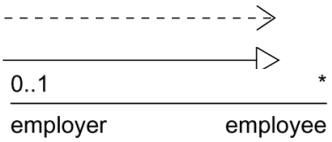
Terms and Concepts:

- Relationship is a connection among things
- The relationship describe how things collaborate with each other
- A relationship illustrated as path with different types of lines to indicate kind of relationship
- There are three most important relationship are:

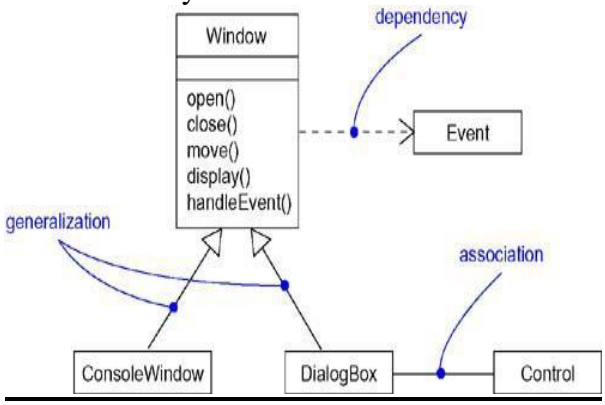
o Dependencies

o Generalizations

o Associations



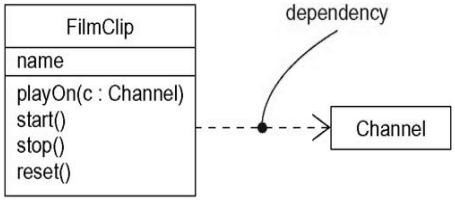
Ex: In the example any change in class event may effect class window.



Dependency

- A dependency is a **using** relationship that states that a change 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.

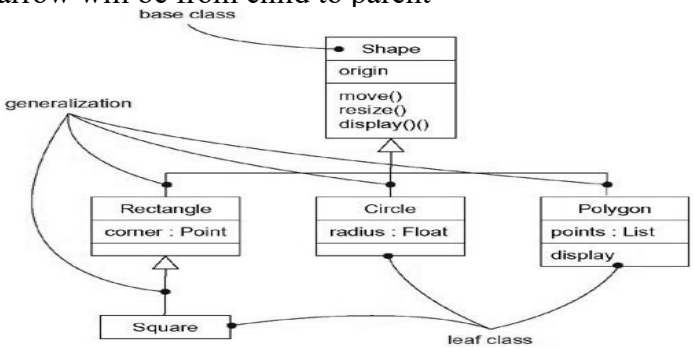
- Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

Generalization (is - a - kind of")

- 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
- A class may have zero, (or) one or more parents
- A class that has no parents and one (or) more children is called root class (or) base class
- A class with no children is called leaf class
- A class with single parent is called single inheritance
- A class with more parents is called multiple inheritance
- Generalization is used to show inheritance relation
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent
- The direction of arrow will be from child to parent

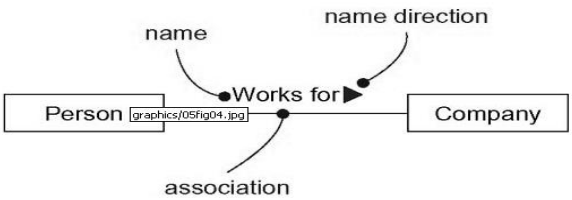


Association

- An associa ng are connected to objects of
- Associatio e class and object of other cl
- An associa :
- An associations that connect more than two classes; these are called n-ary associations
- Graphically, an association is rendered as a solid line connecting the same or different classes.
- Beyond this basic form, there are four adornments that apply to associations

a. Name

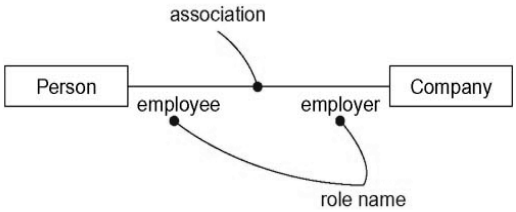
- An association can have a name, and you use that name to describe the nature of the relationship so that there is no ambiguity
- We can give a direction to name by providing a direction triangle those points in direction, we intend to read name



Association Names

b. Role

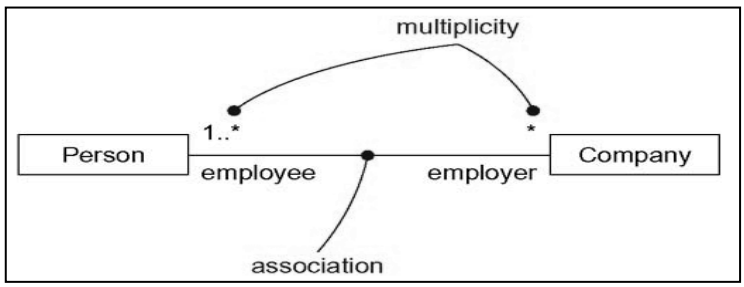
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

c. Multiplicity

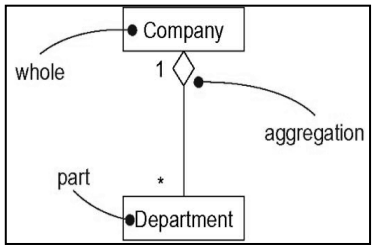
- It's important to state how many objects may be connected across an instance of an association. This count is known as multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..* or simply *), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

d. Aggregation (or) "has-a"

- Aggregation is a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").
- Aggregation is a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

Common Modeling Techniques

Modeling Simple Dependencies

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

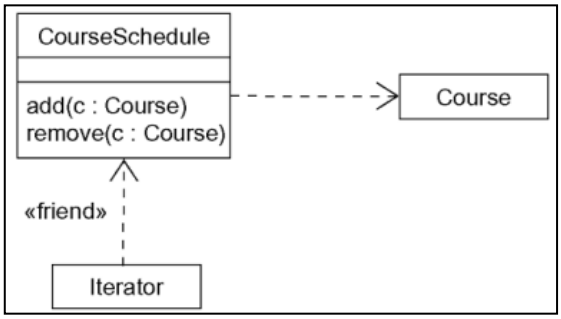
To model this dependencies

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

The following figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university.

This figure shows a dependency from CourseSchedule to Course, because Course is used in both add and remove operations of CourseSchedule.

The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.



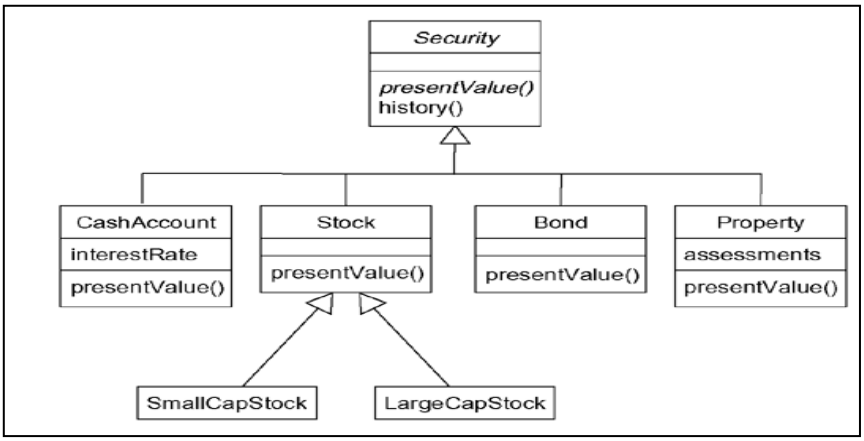
Dependency Relationships

Modeling Single Inheritance

When we model vocabulary of system you will often come across several classes. Classes may have similar behavior to other classes. So a better way is to model class from which specialized one inherits.

To model single inheritance relationship the following should be seen:

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Refine these common responsibilities, attributes, and operations to a more general class. If necessary or 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.



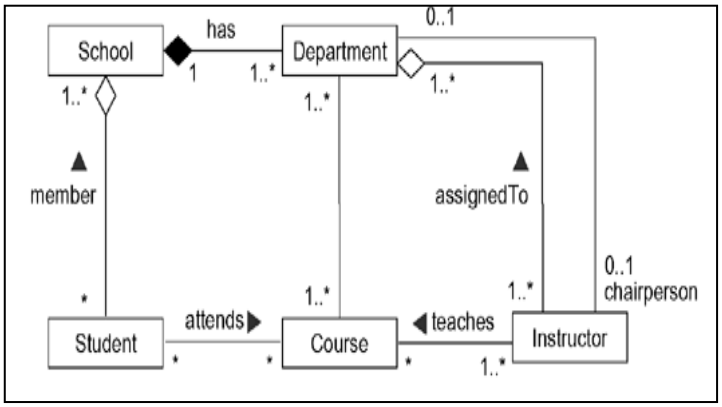
Inheritance Relationships

Modeling Structural Relationships

- Dependency and generalization relationships are one-sided.
- Associations are, bidirectional
- Given an association between two classes, you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

To model structural relationships

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association.
- 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.
- For each of these associations, specify a multiplicity, as well as role names.
- 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



Structural Relationships

There's an association between Student and Course, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students. The relationships between School and the classes Student and Department are a bit different. Here you'll see aggregation relationships. A school has zero or more students, each student may be a registered member of one or more schools, a school has one or more departments, and each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that School is a whole and that Student and Department are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from their school. You'll also see that there are two associations between Department and Instructor. One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally, departments are at

a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

Common Mechanisms

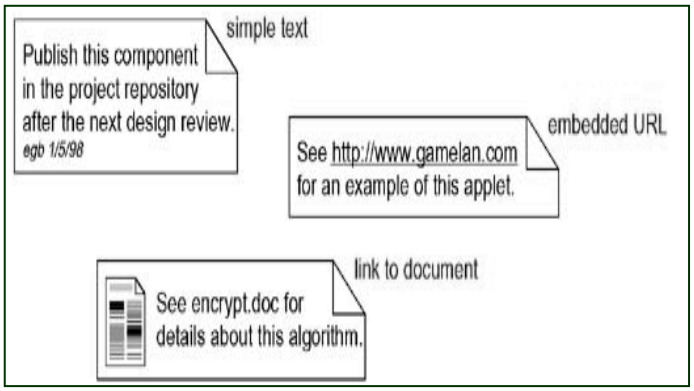
- Modeling is all about communication
- Usually in human languages, vocabulary extends itself as time passes, therefore dictionaries are printed everywhere.
- Similarly in UML when system is designed all concepts and terms can't be expressed then languages so some mechanisms are provided to cover details in the form of comments and constraints. So that we can better understand the system.

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



Notes

- Note has no semantic impact i.e., it do not alter meaning of model
- Notes are used to specify requirements review and explanations
- Note may contain any combination of text (or) graphics
- We can extend one note to another document

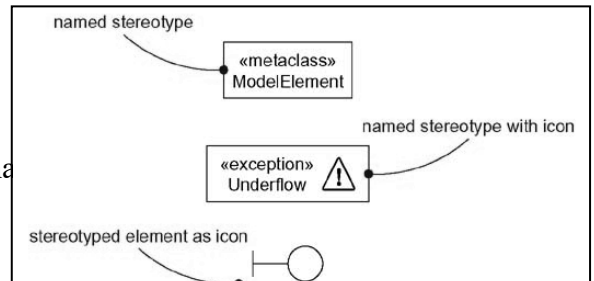
Other adornments:

- Adornments are textual or graphical item that are added to element basic notation and we need to visualize details from elements specification.(adding extra details to the basic notation)
- Example, basic notation for association is line but this may be adorned with details such as roles and multiplicity of each end.
- Most adornment are rendered by placing text near element or by adding graphic symbol to notation some time we can add extra compartment to provide the information.

Stereotypes(new building blocks)

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.

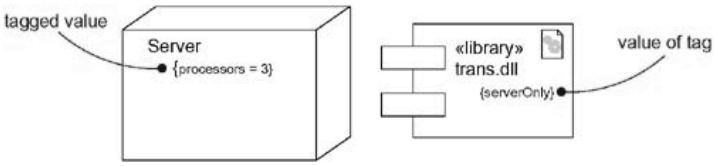
Graphically, a stereotype is rendered as a name enclosed by guillemets(<< >>) and placed above the name of another element



Stereotypes

Tagged Values(new properties)

- Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.
- A tagged value as metadata because its value applies to the element itself, not its instances.
- A tagged value is an extension of the properties of a UML element, allowing you to create new information in that element's specification.
- In its simplest form, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- That string includes a name (the tag), a separator (the symbol =), and a value (of the tag).

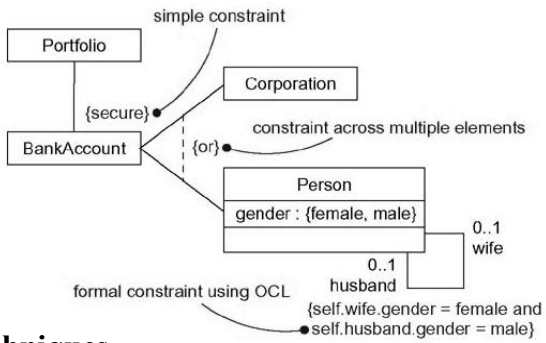


Constraints

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

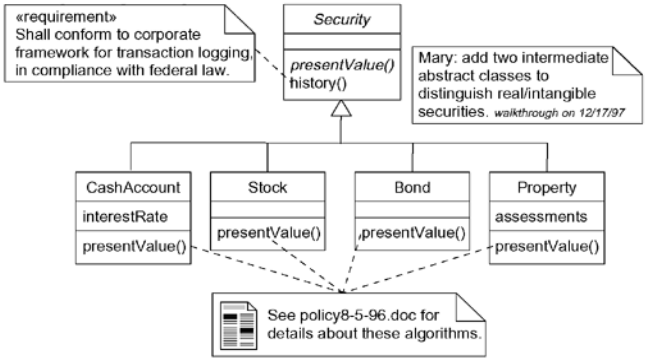
The most common purpose for which you'll use notes is to write down free-form observations, reviews, or explanations.

To model a comment,

Put your comment as text in a note and place it adjacent to the element to which it refers

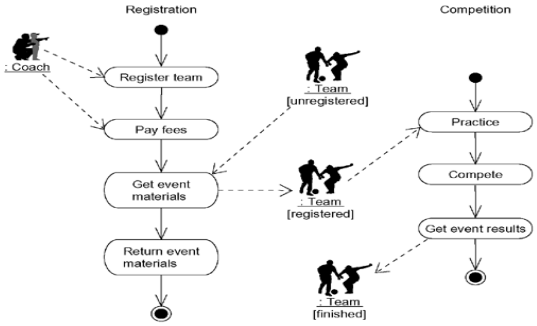
Remember that you can hide or make visible the elements of your model as you see fit.

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



Modeling New Building Blocks

- The UML's building blocks—classes, interfaces, collaborations, components, nodes, associations, and so on—are generic enough to address most of the things you'll want to model.
- However, if you want to extend your modeling vocabulary or give distinctive visual cues to certain kinds of abstractions that often appear in your domain, you need to use stereotypes
- **To model new building blocks,**
  - Make sure there's no way to express what you want by using basic UML
  - 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



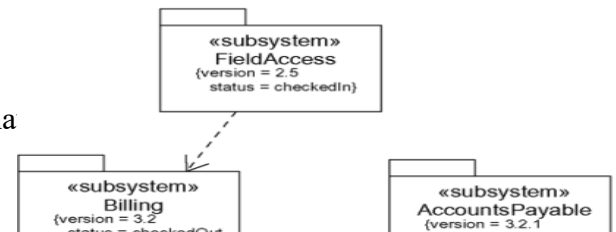
Modeling New Properties

The basic properties of the UML's building blocks—attributes and operations for classes, the contents of packages, and so on—are generic enough to address most of the things you'll want to model.

However, if you want to extend the properties of these basic building blocks, you need to use tagged values.

To model new properties,

1. First, make sure there's no way to express what you want by using basic UML
2. 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 Properties

Modeling New Semantics

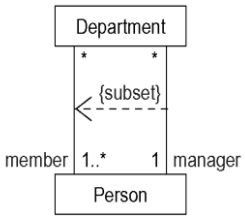
To express new semantics or that you need to modify the UML's rules, then you need to write a constraint.

To model new semantics,

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

If you're convinced there's no other way to express these semantics, write your new semantics 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.



Modeling New Semantics

Diagrams

- When you view a software system from any perspective using the UML, you use diagrams to organize the elements of interest.
- The UML defines nine kinds of diagrams, which you can mix and match to assemble each view.
- Not only the nine kinds of diagrams you could create your own diagrams.
- You'll use the UML's diagrams in two basic ways:
  - to specify models from which you'll construct an executable system (forward engineering)
  - and to reconstruct models from parts of an executable system (reverse engineering).

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
- Each diagram provides a view into the elements that make up the system
- Typically, you'll view the static parts of a system using one of the four following diagrams.
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram
- You'll often use five additional diagrams to view the dynamic parts of a system.
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

The UML defines these nine kinds of diagrams.

- Every diagram you create will most likely be one of these nine or occasionally of another kind
- Every diagram must have a name that's unique in its context so that you can refer to a specific diagram and distinguish one from another
- You can show any combination of elements in the UML in the same diagram. For example, you might show both classes and objects in the same diagram

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

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

Object Diagram

- An object diagram shows a set of objects and their relationships.
- Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.

- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

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

Deployment Diagram

- A deployment diagram shows a set of nodes and their relationships.
- We use deployment diagrams to illustrate the static deployment view of architecture.
- 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

UseCase 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

- 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.
- We use sequence diagrams to illustrate the dynamic view of a system.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Collaboration Diagram

- 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.
- We use collaboration diagrams to illustrate the dynamic view of a system.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

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

Statechart Diagram

- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
- Statechart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.
- 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.

Activity Diagram

- 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.
- We use activity diagrams to illustrate the dynamic view of a system.
- Activity diagrams are especially important in modeling the function of a system.

Common Modeling Techniques

Modeling Different Views of a System

- When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions
- To model a system from different views,
  - Decide which views you need to best express the architecture of your system
  - For each of these views, decide which part you need to create to capture the essential details of that view.
  - As part of your process planning, decide which of these diagrams you want to put under some sort of formal or semi-formal control.
- For example, if you are modeling a simple monolithic(Single) application that runs on a single machine, you might need only the following handful of diagrams

- If yours is a reactive system or if it focuses on process flow, you'll probably want to include statechart diagrams and activity diagrams, respectively, to model your system's behavior.
- Similarly, if yours is a client/server system, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.
- Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

• Use case view	:	Use case diagrams Activity diagrams (for behavioral modeling)
• Design view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling) * Statechart diagrams (for behavioral modeling)
• Process view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling)
• Implementation view	:	Component diagram
• Deployment view	:	Deployment diagrams

Modeling Different Levels of Abstraction

- Not only do you need to view a system from several angles, you'll also find people involved in development who need the same view of the system but at different levels of abstraction
- Basically, there are two ways to model a system at different levels of abstraction:
  - By presenting diagrams with different levels of detail against the same model

- By creating models at different levels of abstraction with diagrams that trace from one model to another.

- To model a system at different levels of abstraction by presenting diagrams with different levels of detail,
  - Consider the needs of your readers, and start with a given model
  - If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction which means that they'll need to reveal a lot of detail
  - If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction which means that they'll hide a lot of detail
  - Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

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.

- The main advantage of this approach is that you are always modeling from a common semantic repository.

- The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

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:

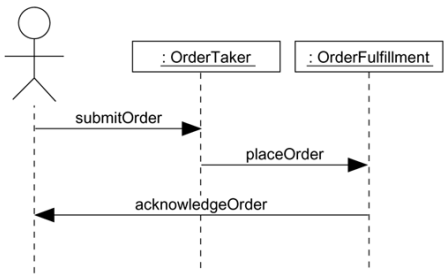
Components in an implementation model will trace to the elements in a design model.

Nodes and their components:

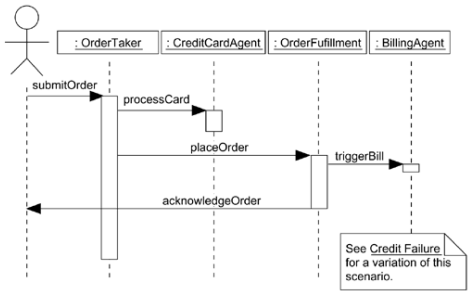
Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models.

The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized



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

- To model 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.

Advanced Structural Modeling

- A relationship is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associattions, and realizations.
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different 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
- A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships.
- There are 17 such stereotypes, all of which can be organized into six groups.

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

1	**bind**	Specifies that the source instantiates the target template using the given actual parameters
2	**derive**	Specifies that the source may be computed from the target
3	**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 a classifier 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

bind:

bind includes a list of actual arguments that map to the formal arguments of the template.

derive

When you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual.

friend

When you want to model relationships such as found with C++ friend classes.

instanceOf

When you want to model the relationship between a class and an object in the same diagram, or between a class and its metaclass.

instantiate

when you want to specify which element creates objects of another.

powertype

when you want to model classes that cover other classes, such as you'll find when modeling databases

refine

when you want to model classes that are essentially the same but at different levels of abstraction.

use

when you want to explicitly mark a dependency as a using relationship

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

9	**access**	Specifies that the source package is granted the right to reference the elements of the target package
10	**import**	A kind of access that specifies that the public contents of the target package 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 later point 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

* We'll use become and copy when you want to show the role, state, or attribute value of one object at
different points in time or space

* You'll use call when you want to model the calling dependencies among operations.

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

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

* We'll use send when you want to model an operation dispatching a given event to a target object.
* The send dependency in effect lets you tie independent state machines together.

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

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

* We'll use trace when you want to model the relationships among elements in different models

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).

In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,

The UML defines 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

?implementation

- We'll use implementation when you want to model private inheritance, such as found in C++.

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

1	**complete**	Specifies that all children in the generalization have been specified in the 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 the children as a type
4	**overlapping**	Specifies that objects of the parent may have more than one of the children as a type

complete

- We'll use the complete constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy);

incomplete

- We'll use incomplete to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show everything in the model).

Disjoint & overlapping

- These two constraints apply only in the context of multiple inheritance.
- We'll use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

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

We use associations when you want to show structural relationships.

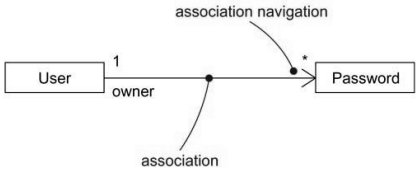
There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

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

Navigation
Vision
Qualification
various flavors of aggregation.

Navigation

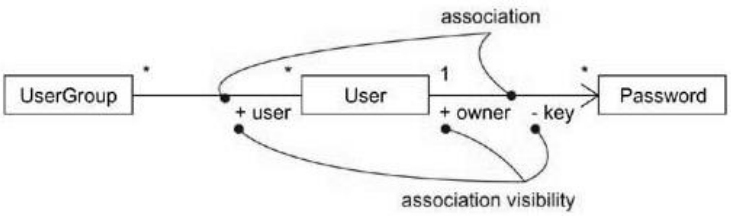
- unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.
- However, there are some circumstances in which you'll want to limit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



Navigation

Visibility

- Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation.
- However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.
- In the UML, you can specify three levels of visibility for an association end, just as you 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 objects outside the association.
- Protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.



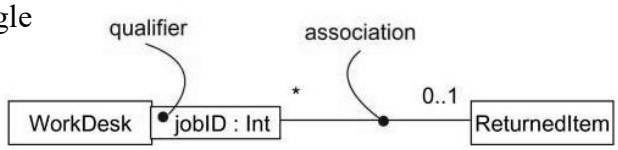
Visibility

Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end?

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



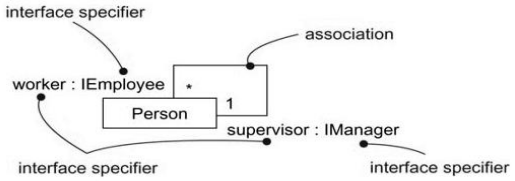
Qualification

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 of the behavior of that class.

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



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-many association, 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

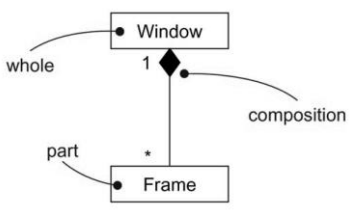
* Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."

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

* Parts with non-fixed multiplicity may be created after the composite itself, but once created they 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 composite at a time

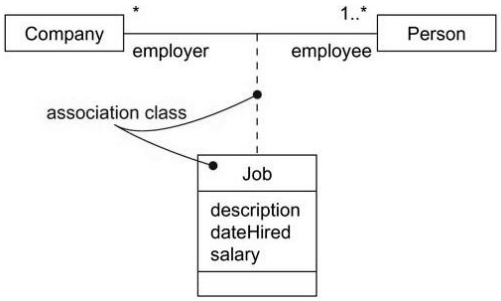
* In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts



Composition

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 a class that also has association properties.
* We render an association class as a class symbol attached by a dashed line to an association



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

implicit

* if you have an association between two base classes, you can specify that same association between two children of those base classes

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

ordered

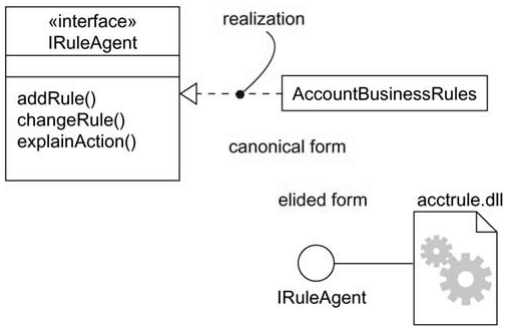
* For example, in a User/Password association, the Passwords associated with the User might be kept in a least-recently used order, and would be marked as ordered.

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

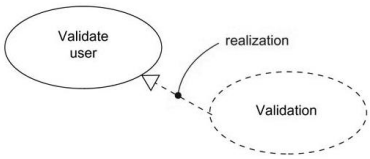
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

1. Realization is sufficiently different from dependency, generalization, and association relationships 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 context of collaborations
5. Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it
6. You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case



Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard

3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled
4. When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.

- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.

- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.

- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.

- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.

- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, type and roles

Interface

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

type

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

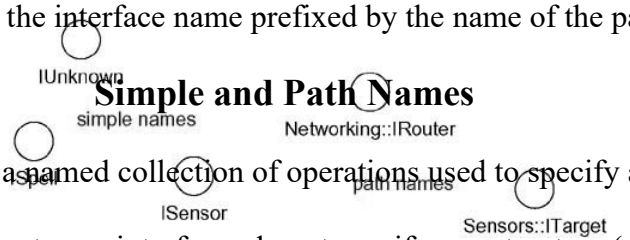
role

- A role is the behavior of an entity participating in a particular context.

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

Names

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

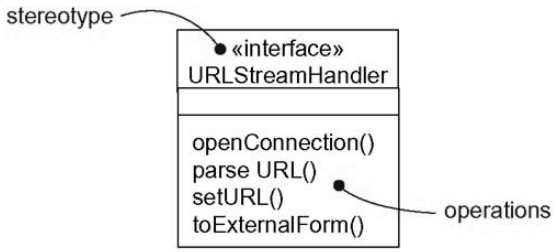


Simple and Path Names

Operations

- An interface is a named collection of operations used to specify a service of a class or of a component.
- Unlike classes or types, interfaces do not specify any 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 appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties



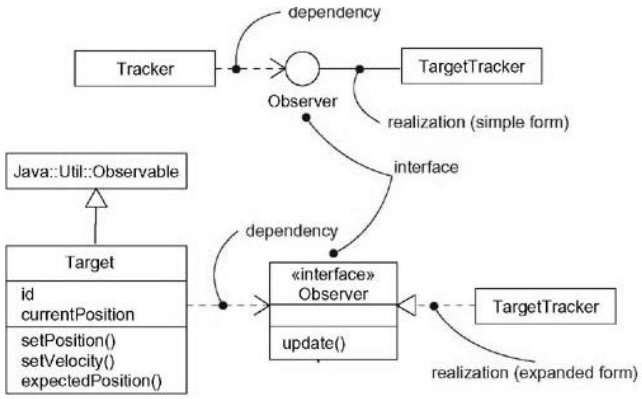
Operations

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 are rendered 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.

Realizations



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 postconditions 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.

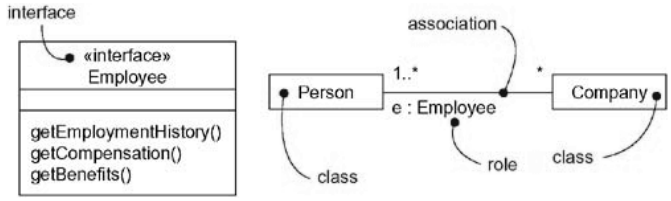
Types and Roles

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

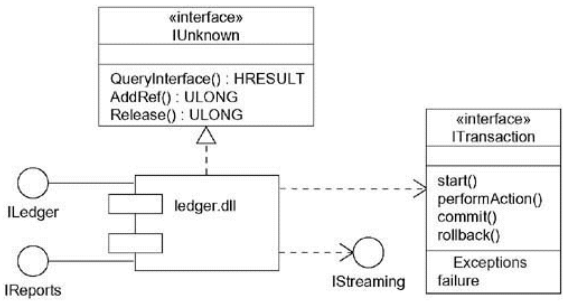
- A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.
- If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype type
- Type is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not.

Common Modeling Techniques

➢ **Modeling the Seams in a Systemeling the Seams in a System**

- The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.

- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,



Modeling the Seams in a System

- The above Figure shows the seams surrounding a component (the library ledger.dll) drawn from a financial system. This component realizes three interfaces: IUnknown, ILedger, and IReports. In this diagram, IUnknown is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by ledger.dll and are exported to other components for them to build on.
- As this diagram also shows, ledger.dll imports two interfaces, IStreaming and ITransaction, the latter of which is shown in its expanded form. These two interfaces are required by the ledger.dll component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces.
- By identifying interfaces such as ITransaction, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

➢ Modeling Static and Dynamic Typeseling Static and Dynamic Types

- Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created.
- Even so, that object will likely play different roles over time.
- Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow,
- To model a dynamic type

o 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 requires only behavior).

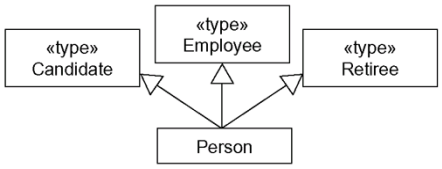
o Model all the roles the of the object may take on at any point in time. You can do so in two ways:

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.

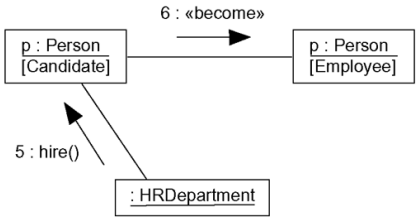
2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

- 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 the interaction, and connect these objects with a message stereotyped as become.



Modeling Static Types



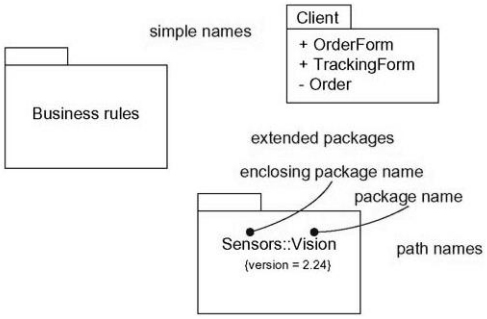
Modeling Dynamic Types

Package

"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 a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



Simple and Extended Package

Owned Elements

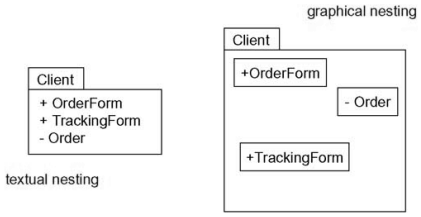
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.

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 uniquely owned 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 decompose your models hierarchically.

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



Owned Elements

Visibility

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

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

Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.

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.

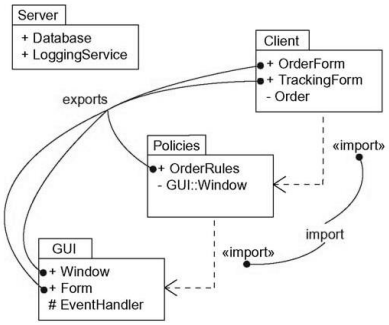
Import adds the contents of the target to the source's namespace

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



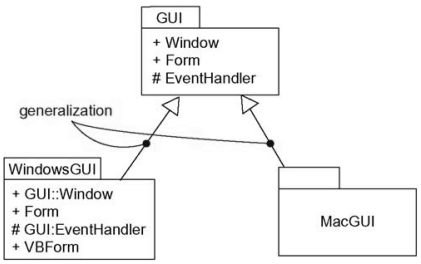
Importing and Exporting

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

Generalization among packages is very much like generalization among classes

Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as WindowsGUI) can be used anywhere a more general package (such as GUI) is used



Generalization Among Packages

Standard Elements

- All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).
- 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

The UML does not specify icons for any of these stereotypes

Common Modeling Techniques

Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.

There is one important distinction between classes and packages:

Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);

classes do have identity (classes have instances, which are elements of a running system).

To model groups of elements,

Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.

Surround each of these clumps in a package.

For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.

Explicitly connect packages that build on others via import dependencies

In the case of families of packages, connect specialized packages to their more general part via generalizations

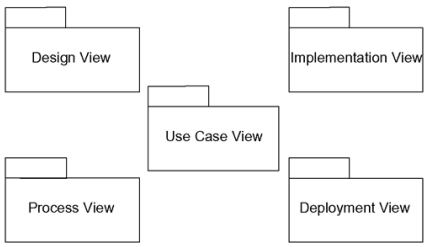
Modeling Architectural Views

- We can use packages to model the views of an architecture.
- Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
- This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
- Second, these packages own all the abstractions germane to that view.(Implementation view)
- To model architectural views,
- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, 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.



Questions Section

a) Short Questions:

1. Differentiate between a class and an object.
2. Define association.
3. What is aggregation?
4. Define composition.
5. What is dependency?
6. Define generalization.
7. What is realization?
8. What are common mechanisms in UML?
9. What are stereotypes?

10. What are tagged values?
11. What are constraints?
12. What is an interface?
13. Define advanced classes.
14. What are advanced relationships?
15. What are packages?
16. What is multiplicity?
17. Define visibility.
18. What are roles in UML?
19. What is a Class Diagram?
20. What is an Object Diagram?
21. Mention any four modeling techniques for Class Diagrams.
22. State two differences between Class and Object Diagrams.
23. What is package dependency?

b) Essay Questions (10 Marks)

1. Explain classes and objects with suitable examples.
2. Explain various UML relationships with neat diagrams.
3. Discuss common mechanisms of UML.
4. Explain advanced structural modeling in UML.
5. Describe advanced classes and advanced relationships.
6. Explain interfaces with suitable examples.
7. Explain types and roles in UML.
8. Discuss packages and package diagrams.
9. Explain the terms and concepts of Class Diagrams.
10. Explain the modeling techniques for Class Diagrams.
11. Explain Object Diagrams and their modeling techniques.
12. Compare Class Diagrams and Object Diagrams with examples.

C) Case Study Questions

Case Study 1 – Library Management System

1. Identify classes and objects.
2. Draw the Class Diagram.
3. Draw the Object Diagram.
4. Explain the relationships used.

Case Study 2 – Online Shopping System

1. Design the Class Diagram.
2. Explain aggregation, composition, and inheritance.
3. Identify interfaces.
4. Organize the classes into packages.

Case Study 3 – Hospital Management System

1. Draw the Class Diagram.
2. Draw the Object Diagram.
3. Explain interfaces and packages used.
4. Discuss multiplicity and visibility.

d) Design-Oriented / HOTS Questions

1. Design a complete Class Diagram for a College Management System showing all relationships.
2. Develop a Class Diagram for a Smart Hospital Management System using interfaces and inheritance.
3. Compare aggregation and composition using a real-world example and justify the appropriate choice.
4. Design a package structure for a University Management System that minimizes coupling and maximizes cohesion.
5. Create both Class and Object Diagrams for an ATM System and explain the differences.
6. Analyze an e-commerce application and identify advanced classes, interfaces, packages, and relationships.
7. Evaluate how common mechanisms improve software maintainability.
8. Design a Hotel Reservation System using advanced UML concepts.
9. Improve an existing Class Diagram by applying interfaces, stereotypes, constraints, and packages.
10. Design a reusable UML model for a Food Delivery System and justify every relationship used.

References:

3. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modelling Language User Guide, Pearson Education 2nd Edition.

4. J Object-Oriented Analysis and Design with the Unified Process By John W Satzinger, Robert B Jackson and Stephen D Burd, Cengage Learning.

Expected Outcomes

a) Promote Conceptual Understanding

- Develop a strong understanding of UML structural modeling concepts, including classes, objects, relationships, interfaces, packages, and structural diagrams.
- Comprehend the role of Class and Object Diagrams in software design and development.

b) Encourage Analytical and Critical Thinking

- Analyze software requirements to identify appropriate classes, relationships, interfaces, and packages.
- Evaluate different structural modeling approaches and select the most suitable UML constructs for solving real-world software design problems.

c) Align with Bloom's Taxonomy

- **Remember:** Define UML structural elements, relationships, interfaces, and packages.
- **Understand:** Explain Class Diagrams, Object Diagrams, and common mechanisms of UML.
- **Apply:** Construct UML Class and Object Diagrams for software systems.
- **Analyze:** Differentiate among association, aggregation, composition, dependency, and generalization relationships.
- **Evaluate:** Assess the effectiveness of UML structural models in representing system

architecture.

- **Create:** Design complete UML structural models for real-world applications using advanced classes, interfaces, and packages.

d) Support Semester-End Examination Preparation

- Provide comprehensive explanations, solved examples, UML diagrams, summary tables, and practice questions aligned with university examination patterns.
- Include short-answer, descriptive, case study, and Higher Order Thinking Skills (HOTS) questions to strengthen examination readiness.

e) Integrate Theory with Practical Relevance

- Demonstrate the application of UML structural modeling concepts through case studies such as Library Management, Banking, Hospital Management, Online Shopping, and Student Registration Systems.
- Enable students to translate software requirements into well-structured UML Class and Object Diagrams that can be implemented in real-world software development projects.