UNIT-III Introduction to UML: Why we model, conceptual model of UML, Architecture, Classes, Relationships, Common Mechanisms, Class diagrams, Object diagrams.
Basic Behavioral Modeling: Interactions, Interaction Diagram, Use Cases, Use Cases Diagram, Activity Diagram.

Introduction:

A successful software organization is one that consistently deploys quality software that meets the needs of its users.

To develop software of lasting quality, you have to craft a solid architectural foundation that's resilient to change. To develop software rapidly, efficiently, and effectively, with a minimum of software scrap and rework, you have to have the right people, the right tools, and the right focus.

Modeling is a proven and well-accepted engineering technique.

What is a model?

A model is a simplified representation of a thing or A model is a simplification of reality.

The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.

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.

- Model is a catalog
- Blueprint of the system.
- Contains all the details 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 entirety.

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

- Easily understand
- Better communication

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.
2. Every model may be expressed at different levels of precision.

3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

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

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.

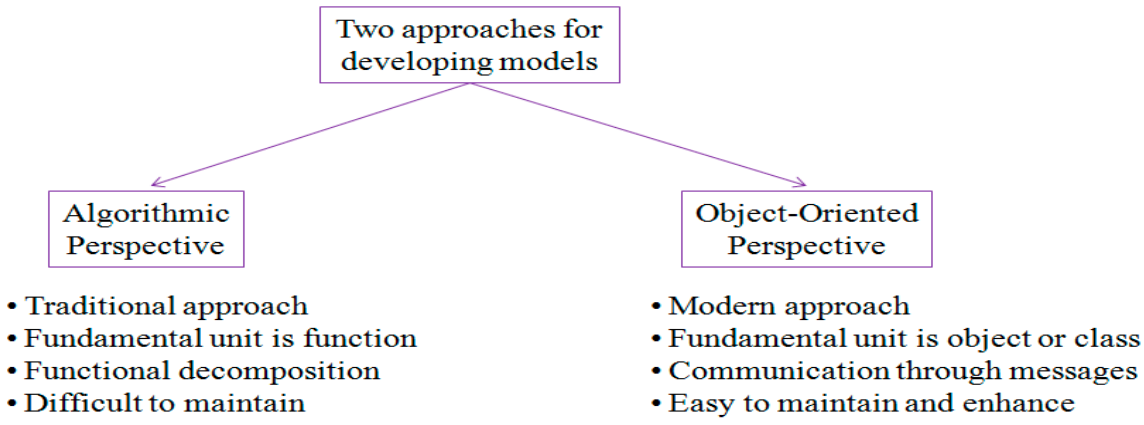
Object-oriented perspective

The contemporary view of software development takes an object-oriented perspective.

In this approach, the main building block of all software systems is the object or class.

A class is a description of a set of common objects. Every object has identity, state, and behavior.

Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.



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.

i) History of UML:

- UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.
- In 2000, ISO recognized UML as a standard language. The version of UML was 1.0.
- In 2004, another major upgrade was made to UML's specification which is known as UML 2.0.
- The latest version of UML till date is UML 2.5.1 published in Dec 2017

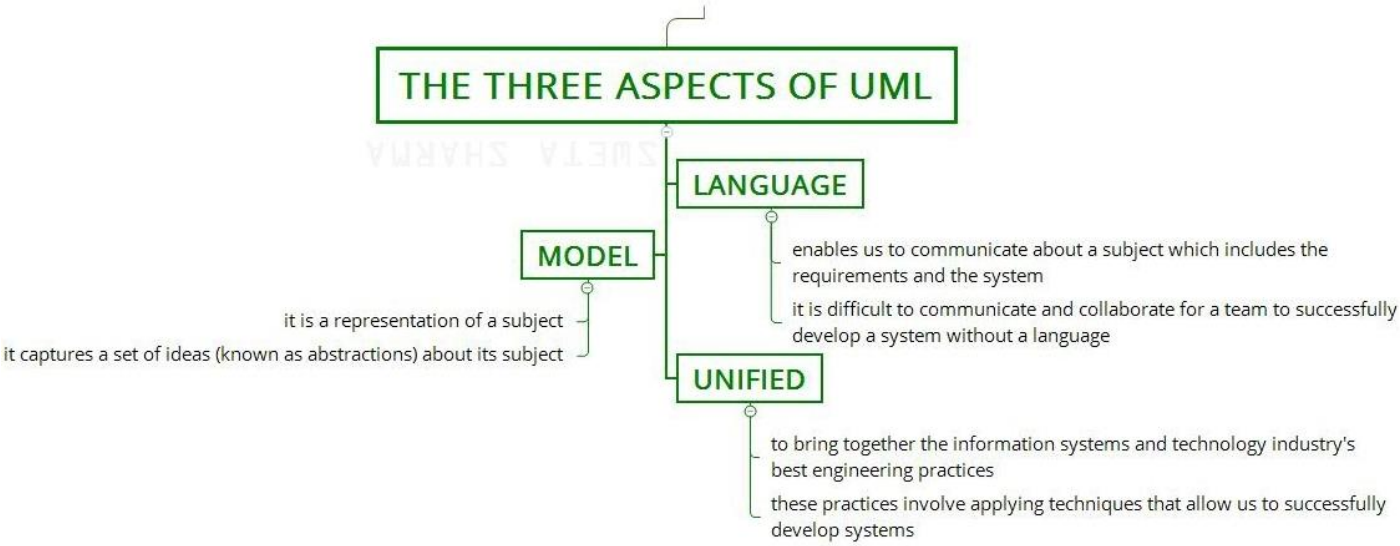
ii) About UML:

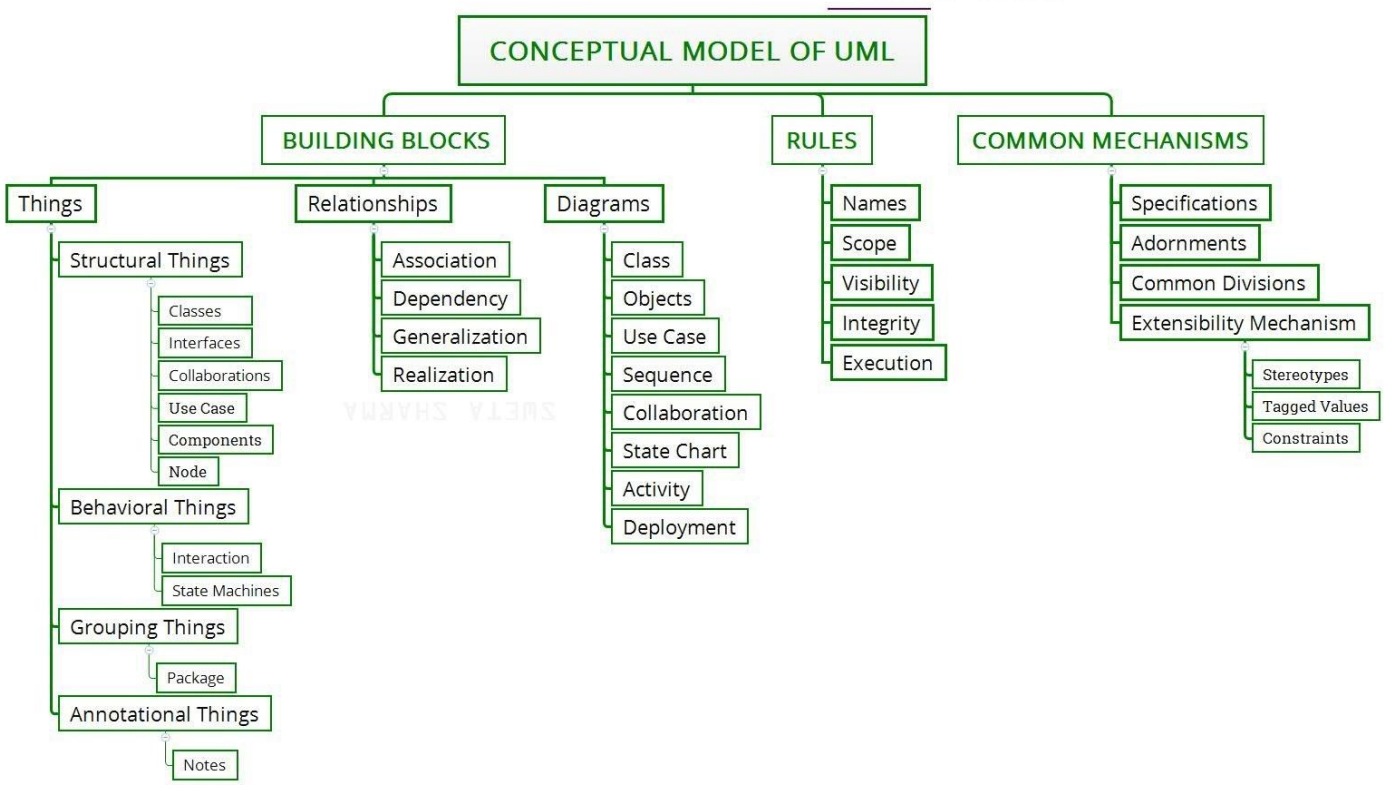
- UML stands for Unified Modeling Language.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.
- UML is not a programming language but tools can be used to generate code in various languages using UML diagrams.
- Forward and Reverse Engineering
- forward engineering: The generation of code from a UML model into a programming language. The reverse is also possible: You can reconstruct a model from an implementation back into the UML.
- Combining these two paths of forward code generation and reverse engineering yields round-trip engineering, meaning the ability to work in either a graphical or a textual view, while tools keep the two views consistent

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

- **Visualizing** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously
- **Specifying** means building models that are precise, unambiguous, and complete.
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages
- **Documenting** a healthy software organization produces all sorts of artifacts in addition to raw executable code.
- These artifacts include
    - Requirements
    - Architecture
    - Design
    - Source code
    - Project plans
    - Tests
    - Prototypes
    - Releases





To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements:

1. Things
2. Relationships
3. Diagrams

Things in the UML

There are four kinds of things in the UML: Structural things Behavioral things Grouping

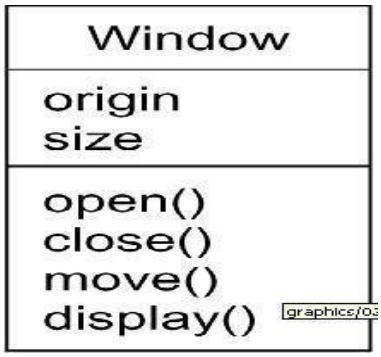
things
Annotational
things

Structural things

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

1.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.



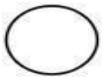
2. Interface

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 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



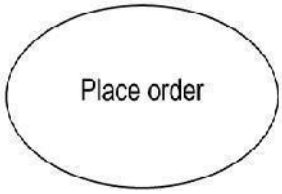
3. Collaboration

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

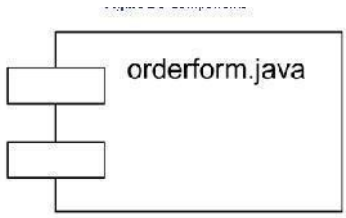
4. Usecase

- 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

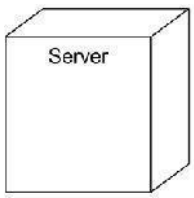


5.Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations

6. 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



7.Node is a physical element that exists at run time and represents a co including only its name



Behavioral Things

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

Interaction
state machine

1.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

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

2.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

State machine involves a number of other elements, including states, transitions, events and activities

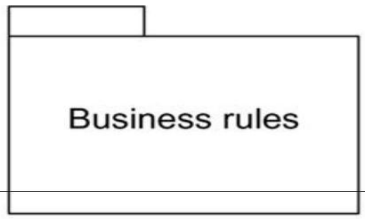
Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

Grouping Things:-

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

Package:-

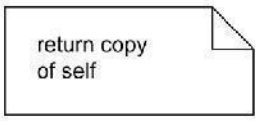
- A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package
- Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



Annotational things

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

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

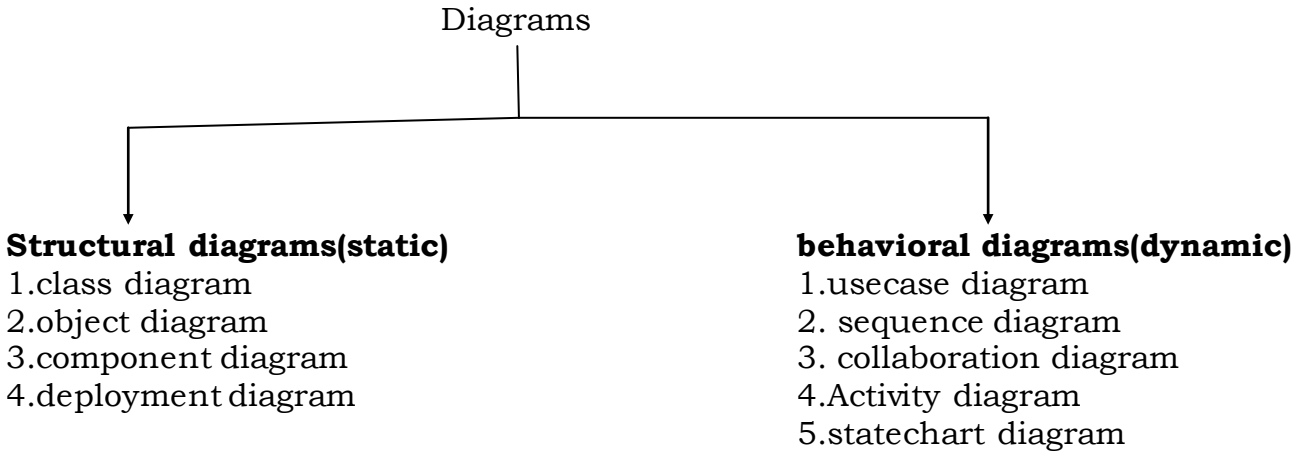
Association is a structural relationship that describes a set of links, a link being a connection among objects.
Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names

Generalization is a special kind of association, representing a structural relationship between a whole and its parts. 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. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship

Diagrams in the UML

- **Diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- In theory, a diagram may contain any combination of things and relationships.
- For this reason, the UML includes nine such diagrams:
  - Class diagram
  - Object diagram
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram
  - Component diagram
  - Deployment diagram



Class diagram

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

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.
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object.

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

Component diagram

- A component diagram shows the organizations and dependencies among a 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
- Deployment diagrams address the static deployment view of an architecture

Rules of the UML

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

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

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

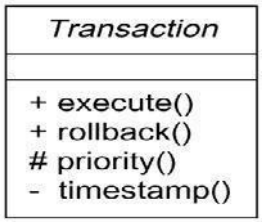
Common Mechanisms in the UML

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

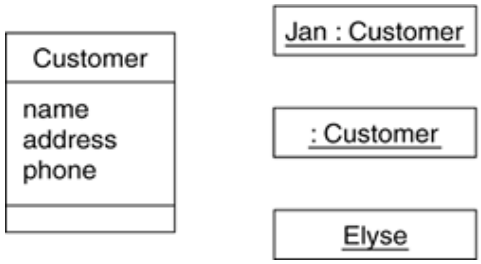
1.Specification that provides a textual statement of the syntax and semantics of that building block. The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion

2.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.



3. Common Divisions: There is the division of class and object.

A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown below.



4.Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

- 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
- A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification
- A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones

Architecture

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

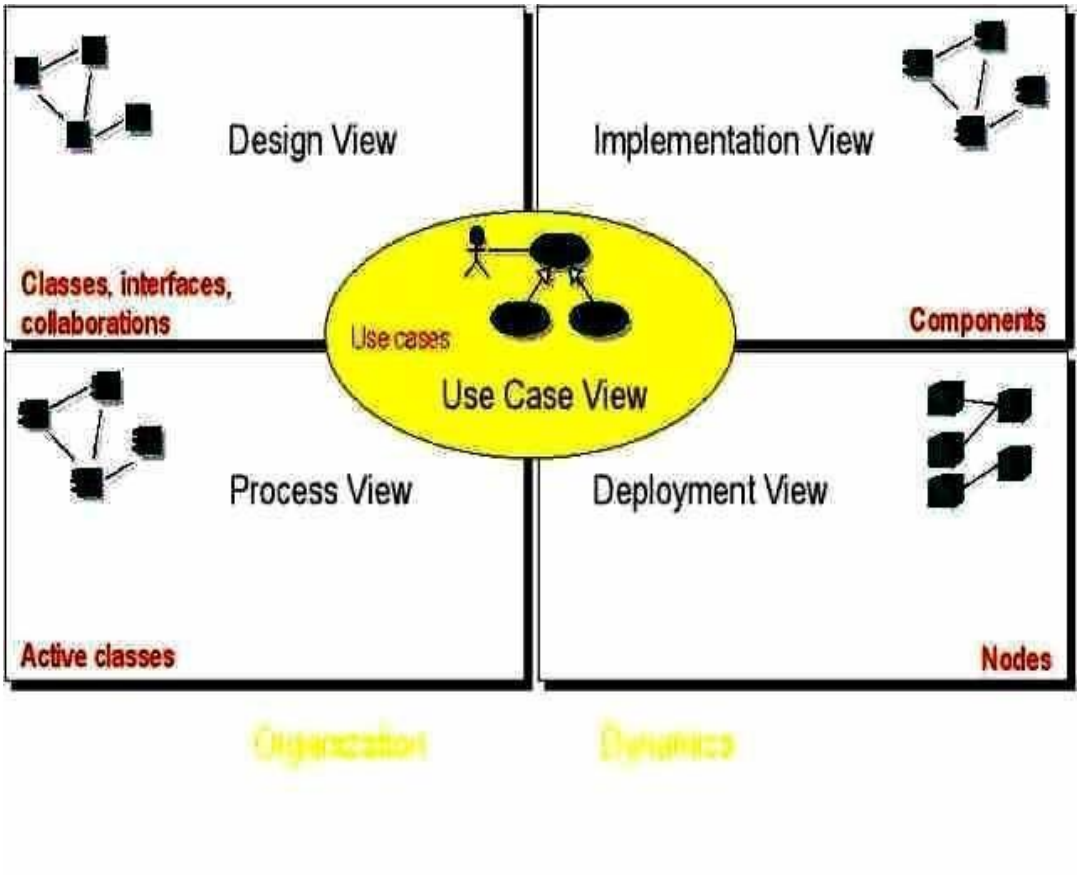
The selection of the structural elements and their interfaces by which the system is composed

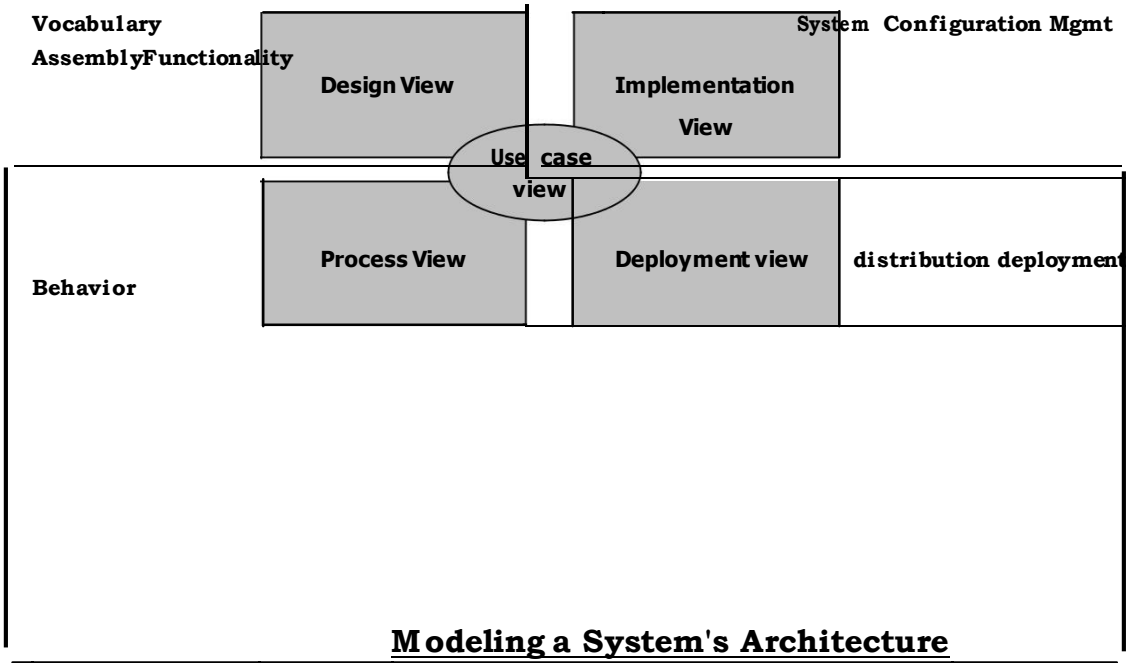
Their behavior, as specified in the collaborations among those elements

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.

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





Modeling a System's Architecture

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.

With the UML, the static aspects of this view are captured in use case diagrams

The dynamic aspects of this view are captured in interaction diagrams, state 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.

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

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.

Deployment view

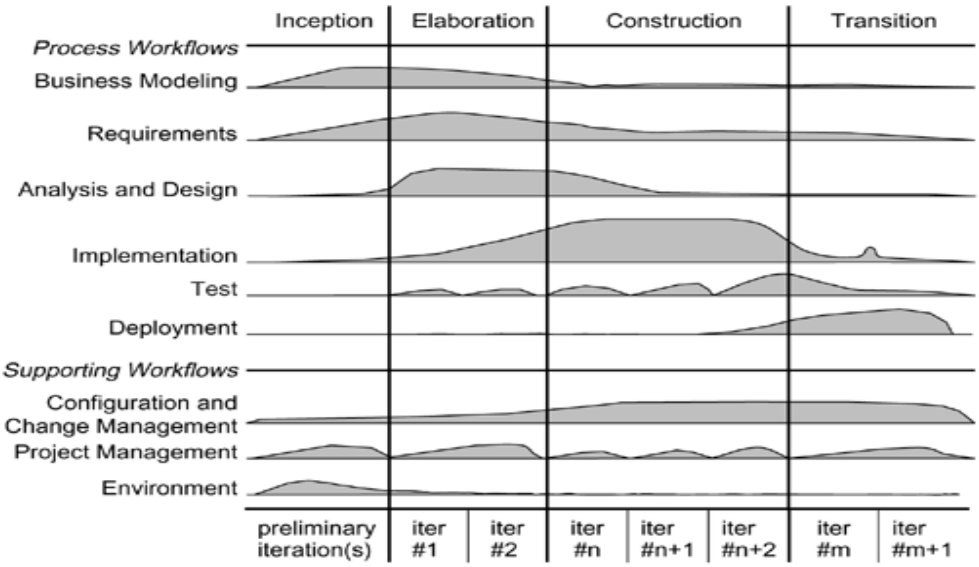
The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes.

This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

Software Development Life Cycle:

- The UML is largely process-independent, meaning that **it is not tied to any particular software development life cycle**.
- However, to get the most benefit from the UML, you should consider a process that is

1. **Use case driven.**
2. **Architecture-centric.**
3. **Iterative and incremental.**



1. **Use case** driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.

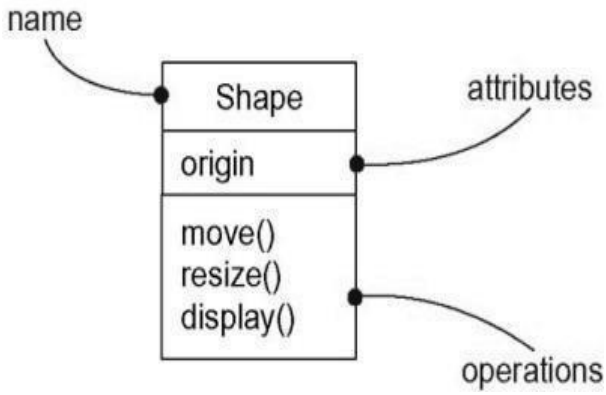
2. Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.

3. An **iterative process** is one that involves managing a stream of executable releases.

An iterative and incremental process is risk-driven, meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.

Class

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- A class implements one or more interfaces.
- The UML provides a graphical representation of class



Graphical Representation of Class in UML Terms and Concepts

Names

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

A name is a textual string that name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives.

Simple Name **Path Name**

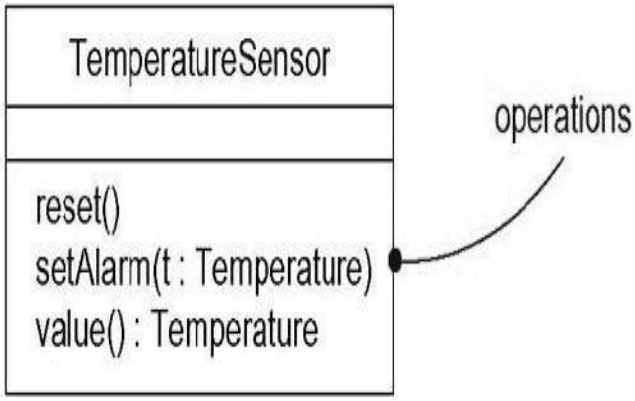
Attributes

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

Attributes and Their Class

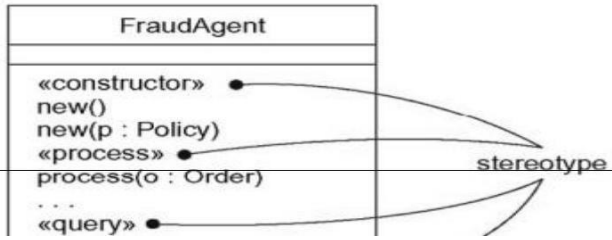
Operations

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



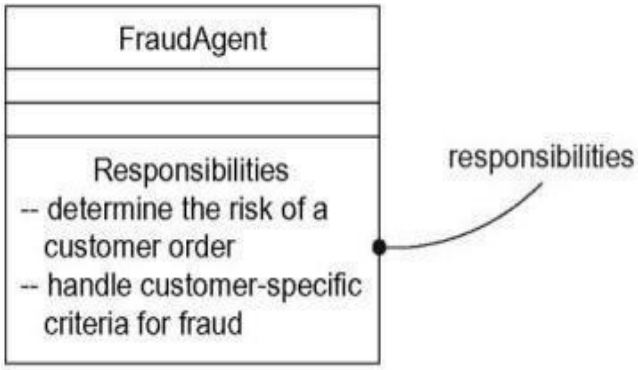
Organizing Attributes and Operations

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes



Responsibilities

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



Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- They represent the things that are important to users and to implementers
- To model the vocabulary of a system
    - Identify those things that users or implementers use to describe the problem or solution.
    - Use CRC cards and use case-based analysis to help find these abstractions.
  - For each abstraction, identify a set of responsibilities.
    - 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 more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.
- To model the distribution of responsibilities in a system
    - Identify a set of classes that work together closely to carry out some behavior.
  - Identify a set of responsibilities for each of these classes.
    - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
    - Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Modeling Non software Things

- Sometimes, the things you model may never have an analog in software
- Your application might not have any software that represents them
- To model nonsoftware things
  - Model the thing you are abstracting as a class.
    - If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
    - If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

Modeling Primitive Types

At the other extreme, the things you model may be drawn directly from the programming language you are using t o 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, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

Relationships

In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships

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

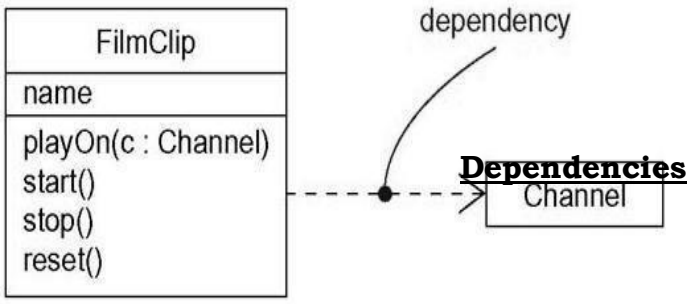
Dependencies
Generalizations
Associations

Dependency

A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse.

Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.

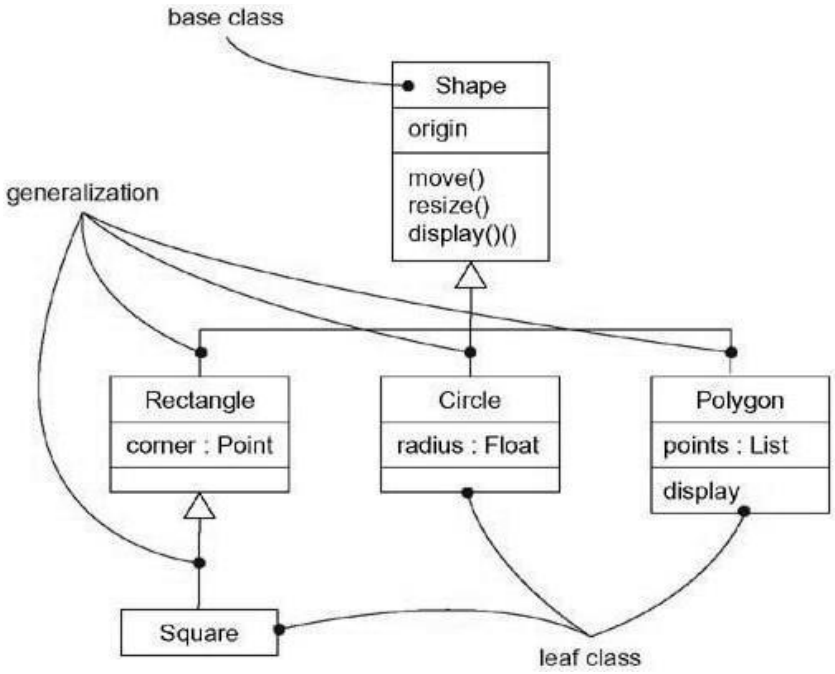
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

Generalization

- A generalization is a relationship between a general thing (called the super class or parent) and a more specific kind of that thing (called the subclass or child).
- Generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations
  - 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



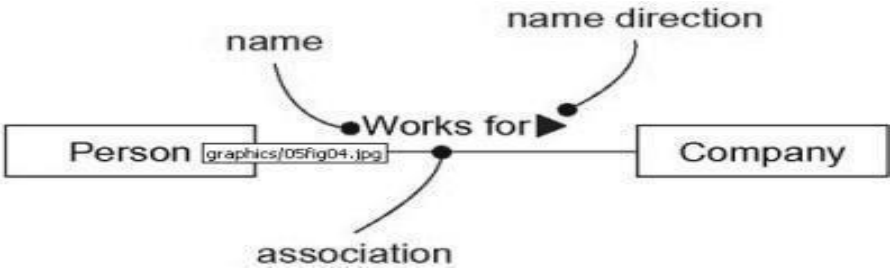
Generalization

Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
- An association that connects exactly two classes is called a binary association
- 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 association

Name

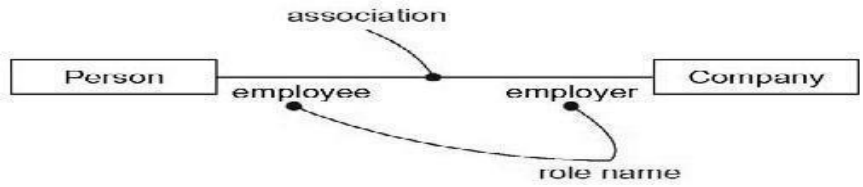
- An association can have a name, and you use that name to describe the nature of the relationship



Association Names

Role

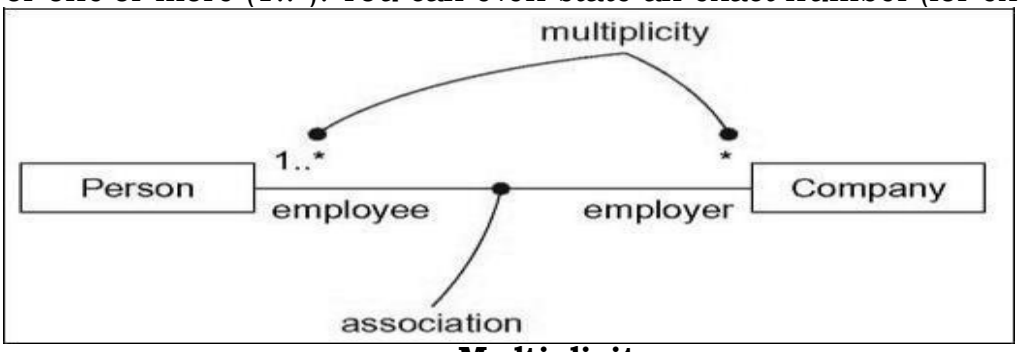
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

Multiplicity

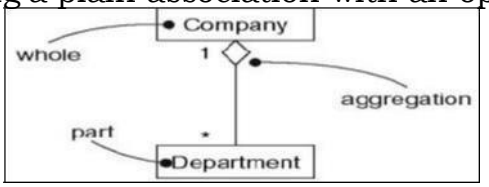
- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- This "how many" is called the multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more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



Multiplicity

Aggregation

- Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").
- This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

Common Modeling Techniques

1. 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 using relationship

- 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 the 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++.

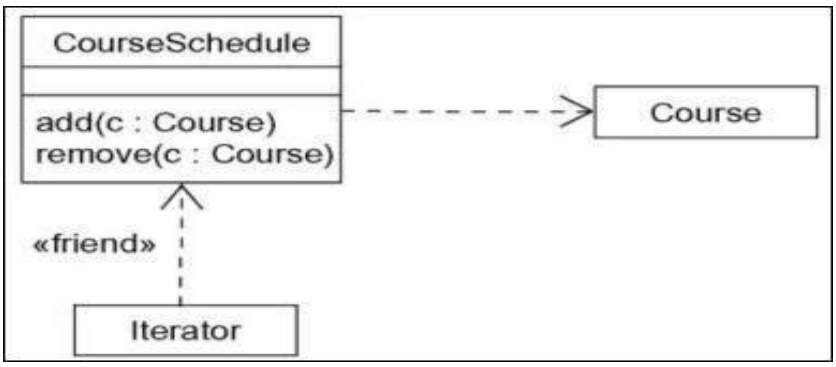


Figure: Dependency Relationships

2. Modeling Single Inheritance

To model inheritance relationships

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these
- 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.

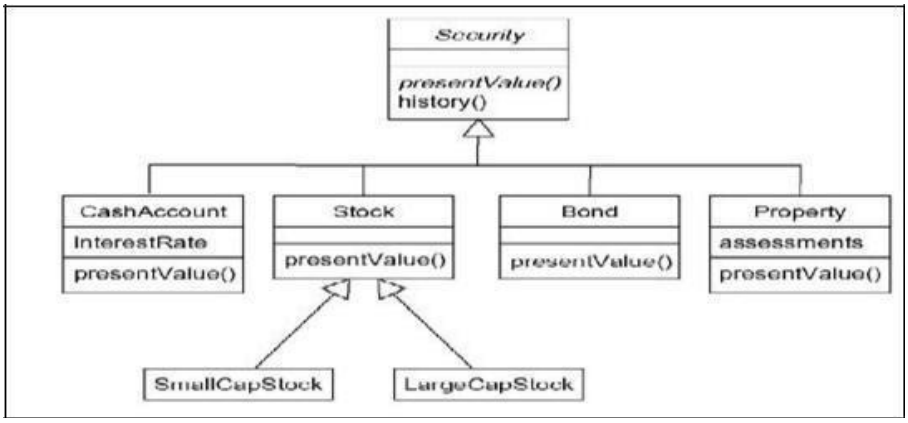


Figure: Inheritance Relationships

3. Modeling Structural Relationships

- When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction
- Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children.
- Dependency and generalization relationships are one-sided.
- Associations are, by default, bidirectional; you can limit their direction
- Given an association between two classes, both rely on the other in some way, and you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

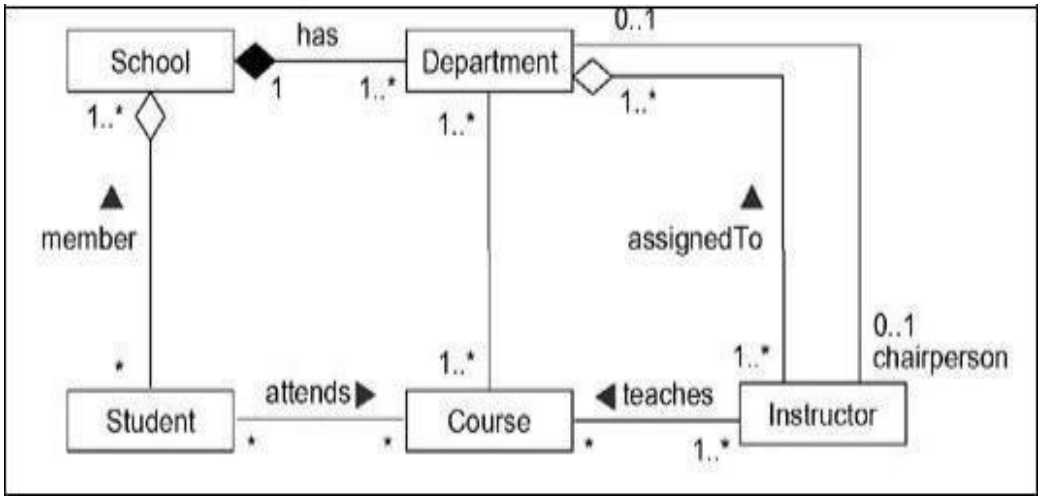
To model structural relationships

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

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

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

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



Structural Relationships

Common Mechanisms

Note

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

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

A note may contain any combination of text or graphics

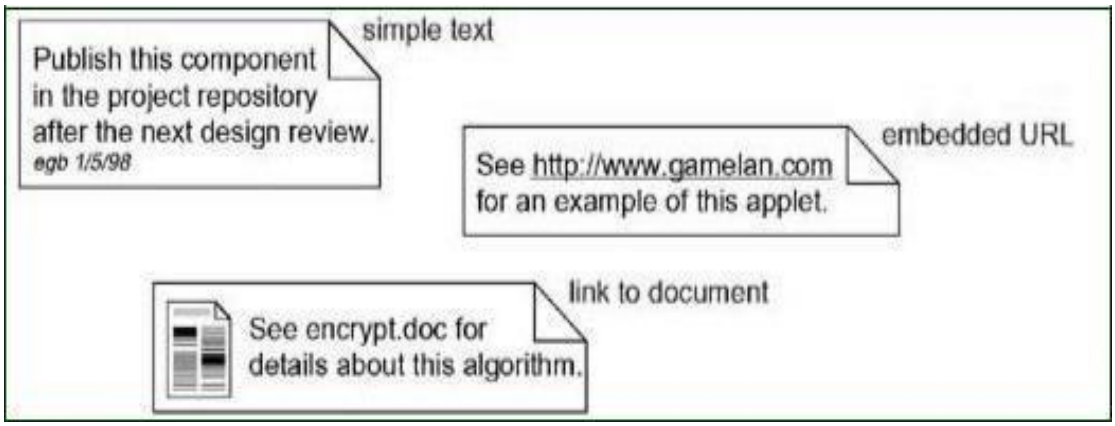


Figure: Notes

Stereotypes

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

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

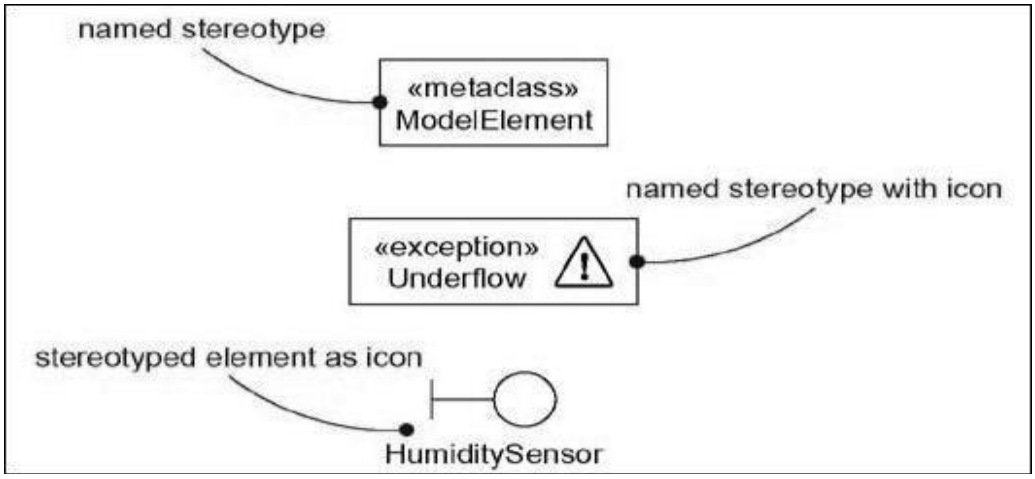
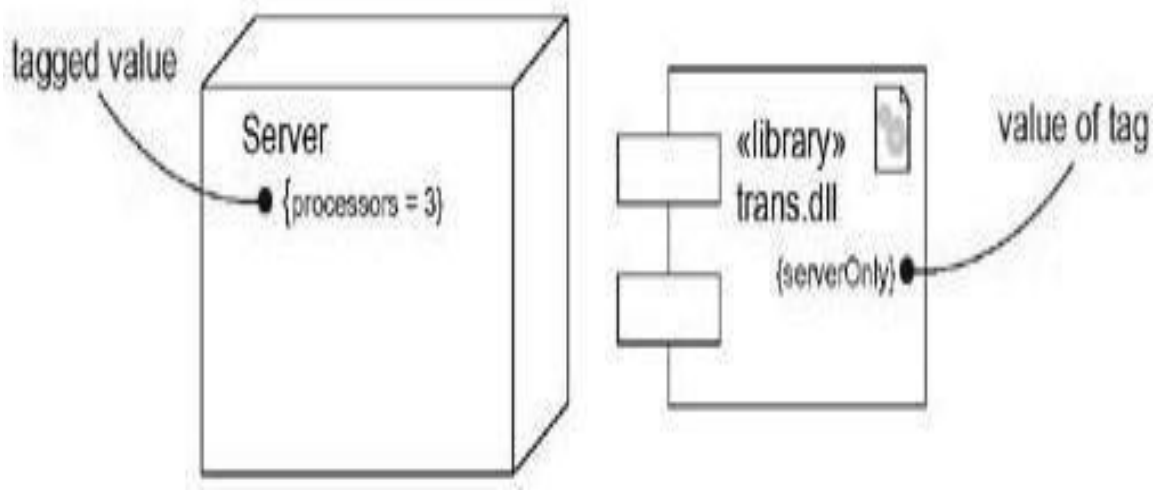


Figure: Stereotypes

Tagged Values

- Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.
- A tagged value is not the same as a class attribute. Rather, you can think of 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.
- Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- 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).

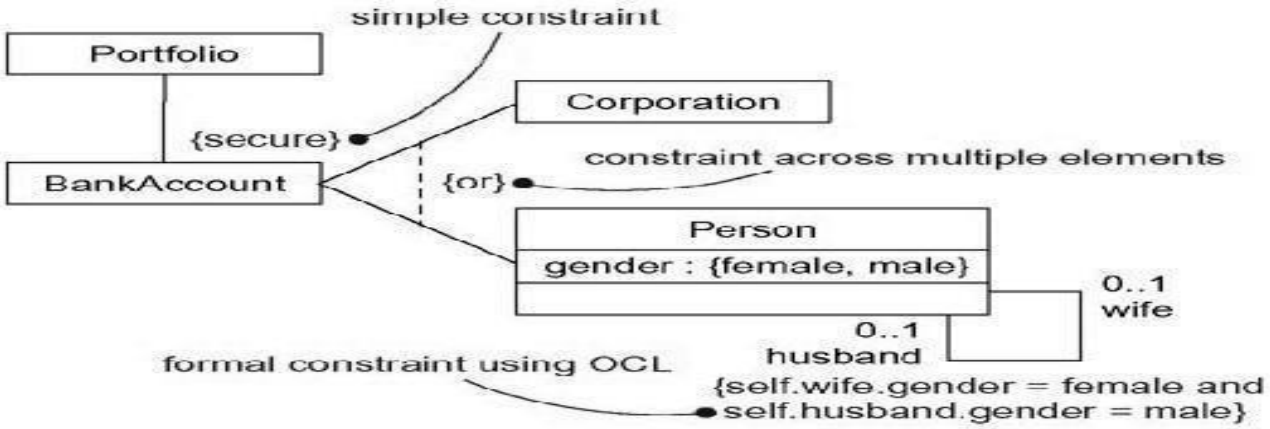


Constraints

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

1.Modeling Comments

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

By putting these comments directly in your models, your models can become a common repository for all the disparate artifacts you'll create during development.

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

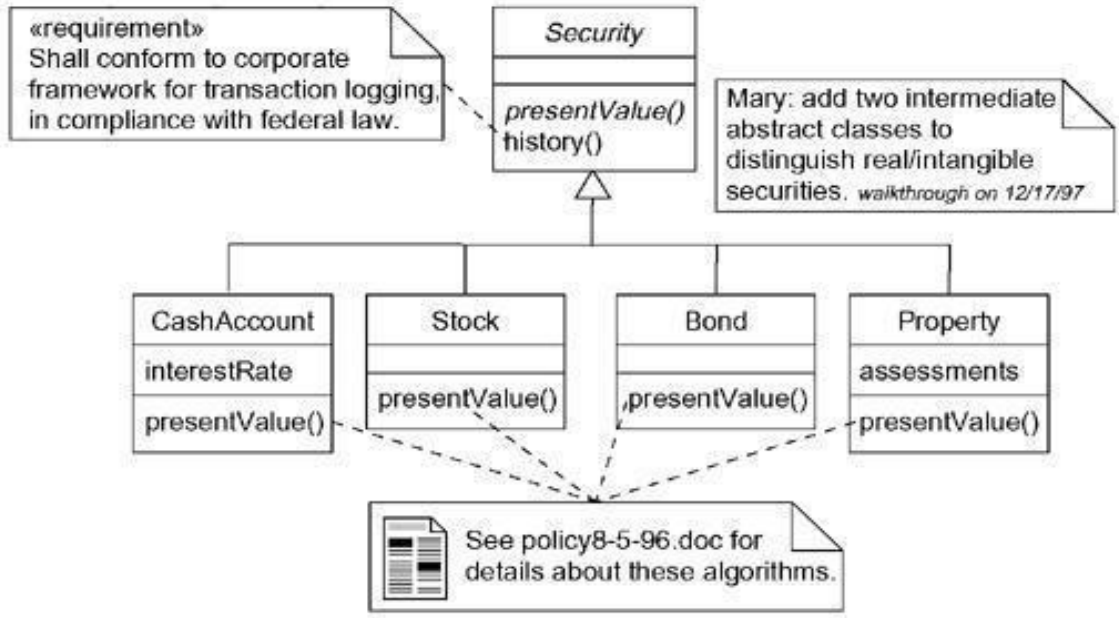


Figure:Modeling Comments

2.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 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, 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

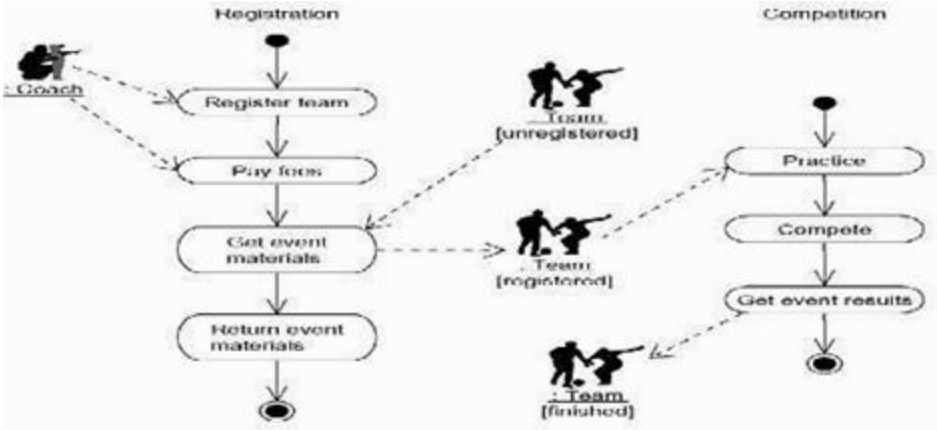


Figure:Modeling New Building Blocks

3.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,

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, add this new property to an individual element or a stereotype.

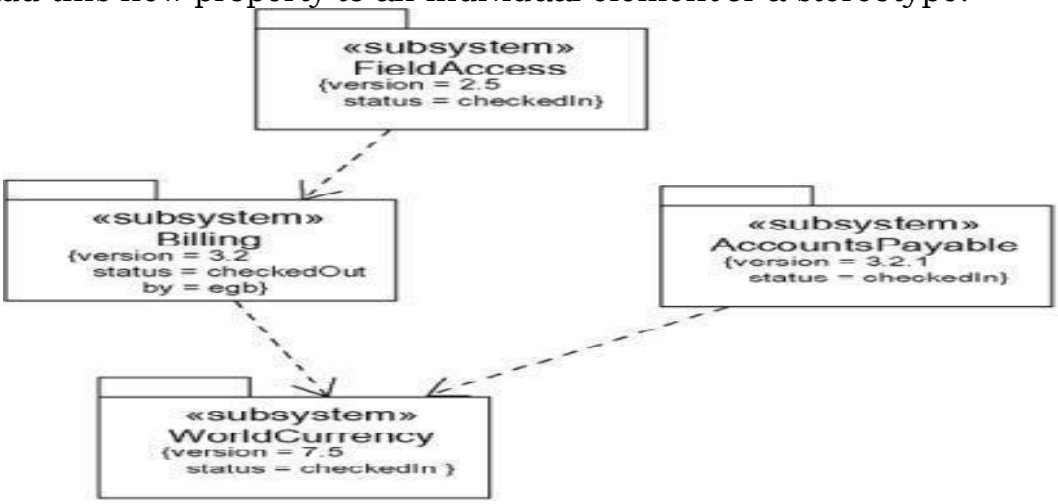


Figure:Modeling New Properties

4.Modeling New Semantics

When you create a model using the UML, you work within the rules the UML lays down

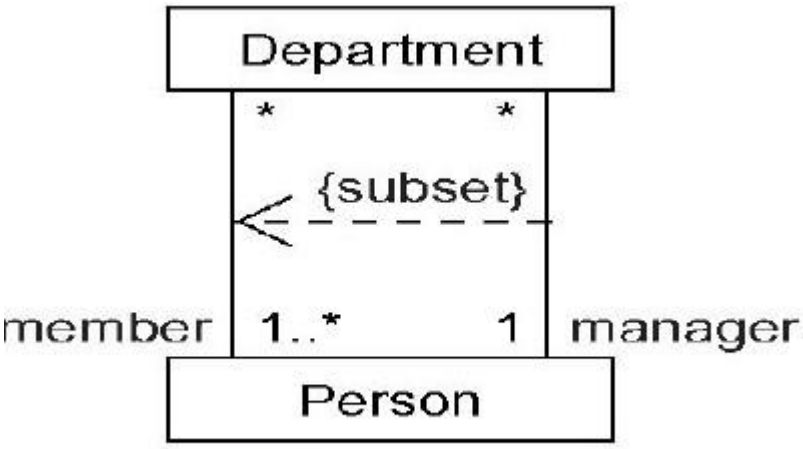
However, if you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.

To model new semantics,

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

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

Classes and Object Diagrams

Class Diagrams

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

Common Properties

Contents

- Class diagrams commonly contain the following things:
  - Classes
  - Interfaces
  - Collaborations
  - Dependency, generalization, and association relationships

- Like all other diagrams, class diagrams may contain notes and constraints
- Class diagrams may also contain packages or subsystems

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

Common Uses

- You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system
- We'll typically use class diagrams in one of three ways:

1. To model the vocabulary of a system
2. To model simple collaborations
3. To model a logical database schema

Modeling the vocabulary of a system

- Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries

Modeling simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements..

Modeling logical database schema

- We can model schemas for these databases using class diagrams.

Common Modeling Techniques in class diagram

1. Modeling Simple Collaborations

- When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.

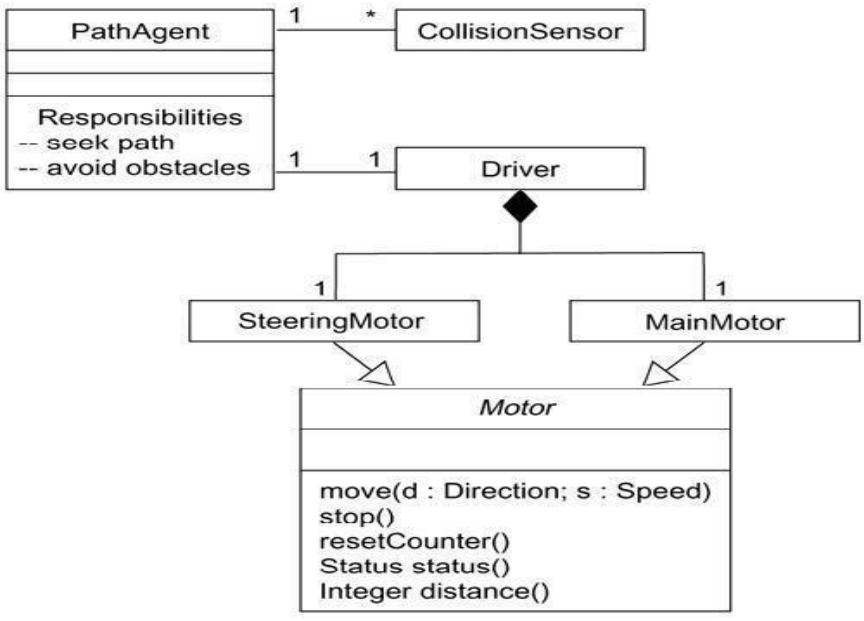
  To model a collaboration

  - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

  - For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.

  - Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.

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



2. Modeling a Logical Database Schema

- The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.
- The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database these logical operations are generally turned into triggers or stored procedures.

  To model a schema,

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

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

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

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

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

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

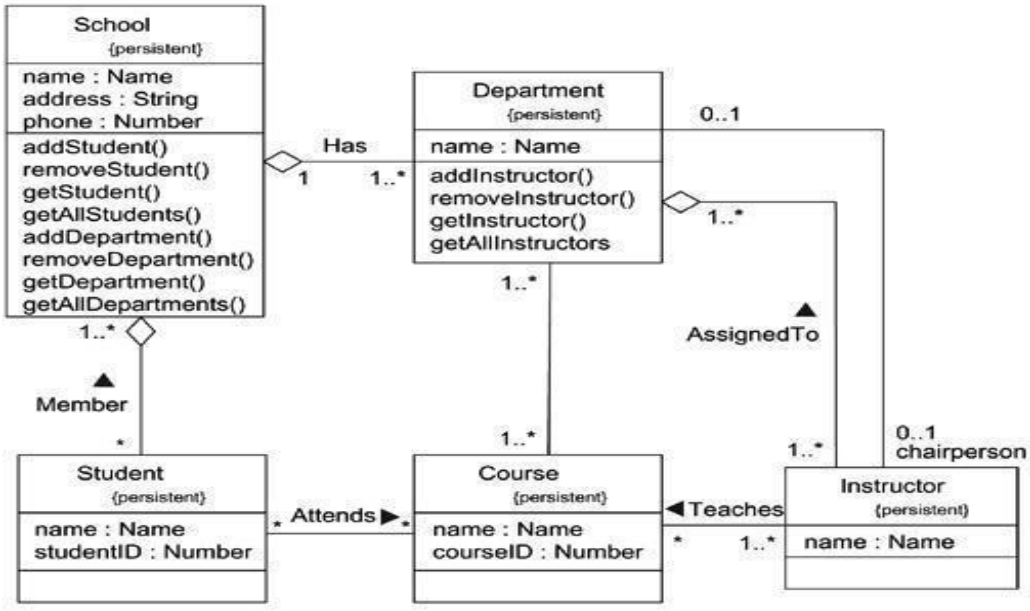


Figure:Modeling a Schema

3.Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language
- Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language.

  To forward engineer a class diagram,

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

  - Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
  - Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
  - Use tools to forward engineer your models.

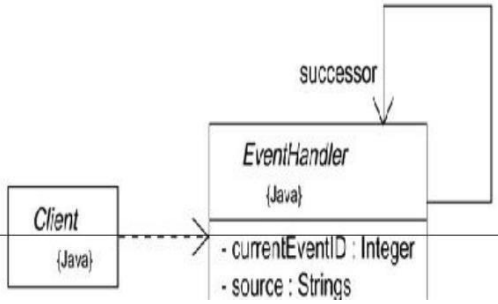


Figure:Forward Engineering

Reverse engineering :

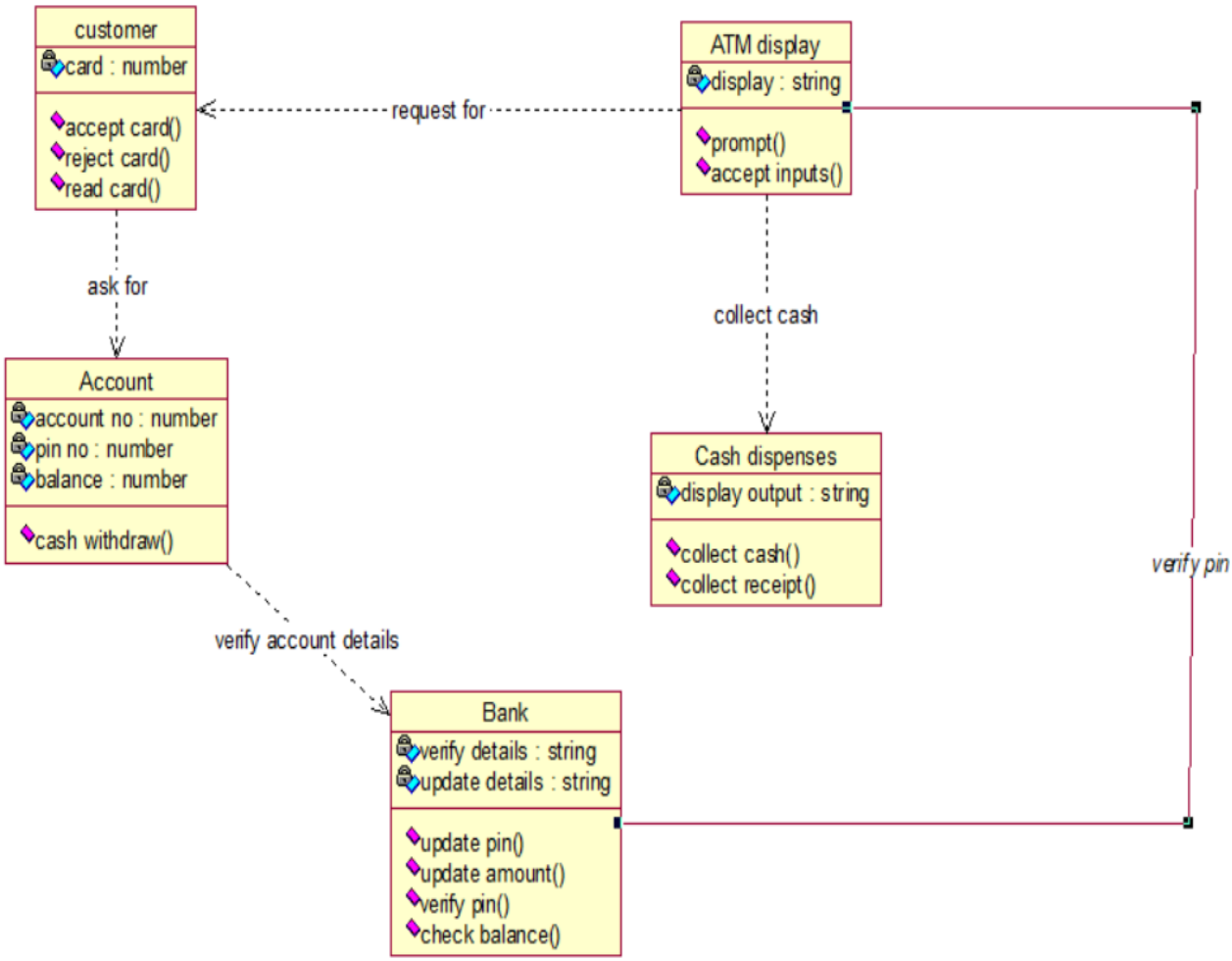
Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language.

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

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

CLASS DIAGRAM FOR ATM APPLICATION



Object Diagram

- An object diagram is a diagram that shows a set of objects and their relationships at a point in time.
- Graphically, an object diagram is a collection of vertices and arcs
- An object diagram is a special kind of diagram and shares the same common properties as all other diagrams—that is, a name and graphical contents that are a projection into a model

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

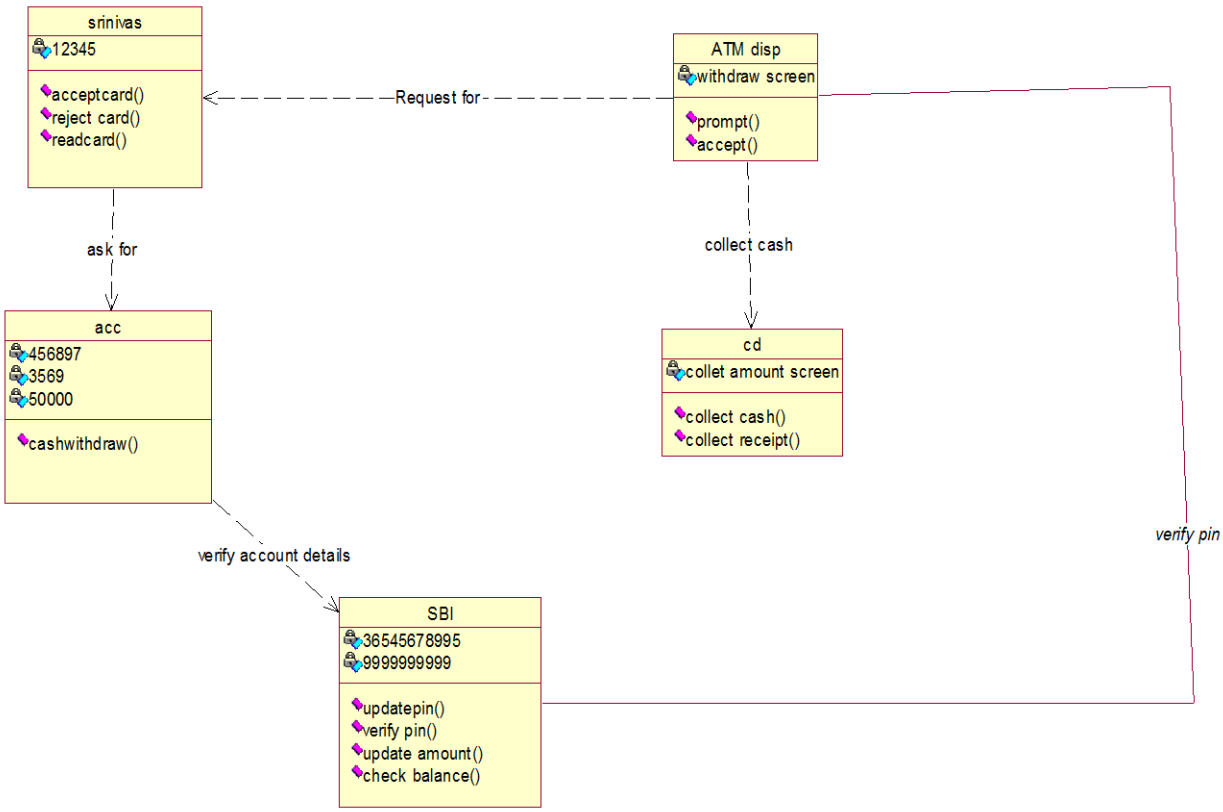


Figure: object diagram

Contents

- Object diagrams commonly contain
  - Objects
  - Links
- Like all other diagrams, object diagrams may contain notes and constraints.
- Object diagrams may also contain packages or subsystems

Common Uses

- You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams
- When you model the static design view or static process view of a system, you typically use object diagrams in one way:
  - To model object structures
  -

Modeling Object Structures

- Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time.
- An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram

Common Modeling Techniques in object diagram

1.Modeling Object Structures

- An object diagram shows one set of objects in relation to one another at one moment in time.
- To model an object structure,
    - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
    - For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.
    - Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
    - Expose the state and attribute values of each such object, as necessary, to understand the scenario.
    - Similarly, expose the links among these objects, representing instances of associations among them.

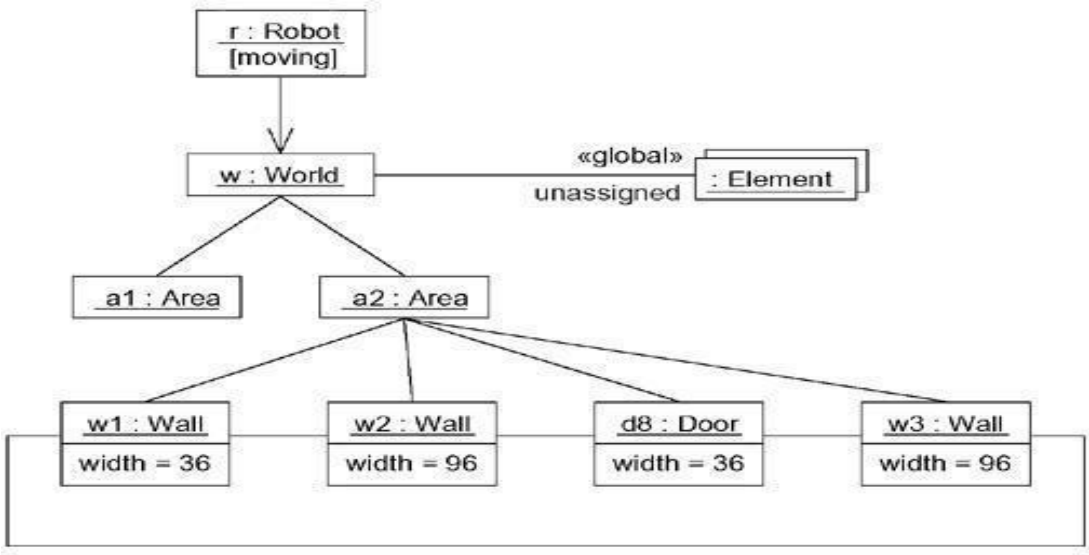


Figure: Modeling Object Structures

2. Forward and Reverse Engineering

- Forward engineering an object diagram is theoretically possible but pragmatically of limited value
- In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.
- Component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.
- Reverse engineering an object diagram is a very different thing

To reverse engineer an object diagram,

- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.