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:

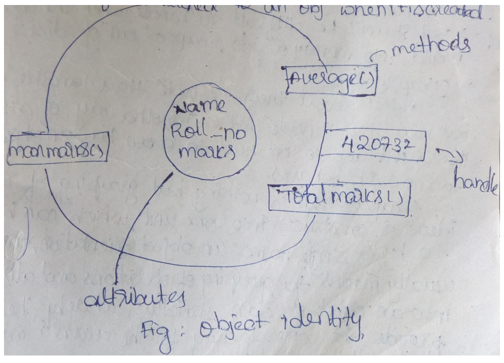
  Value: A data value is used for identity (ex, primary key of a table in a relational db)

  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



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

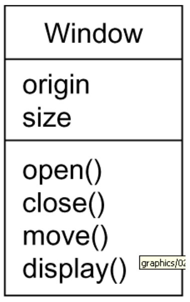
a. Things in the UML

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

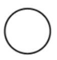
Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

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.

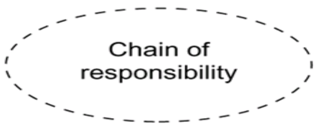


Interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a 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



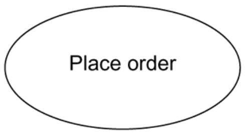
Ispelling

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

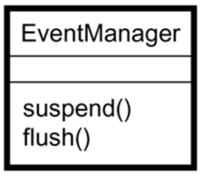


Usecase

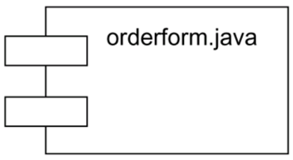
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse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



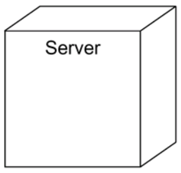
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



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

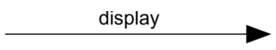


<u>Behavioral Things</u> are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things

0 i. Interaction
1 ii. state machine

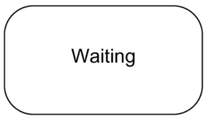
i. Interaction

0 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

0 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

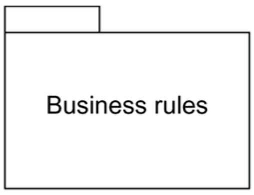


Grouping Things:-

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:-

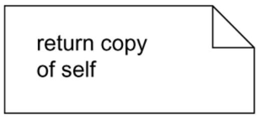
A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Package is purely conceptual. Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes,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

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

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



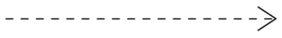
Relationships in the UML: There are four kinds of relationships in the UML:

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

Dependency:-

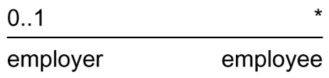
0 Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing

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



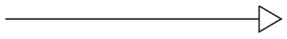
Association is a structural relationship that describes a set of links, a link being a connection among objects.

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

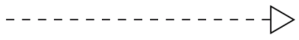


Aggregation is a special kind of association, representing a structural relationship between a whole and its parts.

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

0 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.

0

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

2

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

4 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

0 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.

1

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.

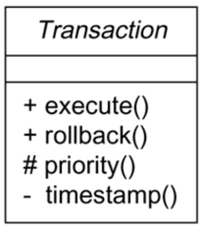
<u>Common Mechanisms</u>

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

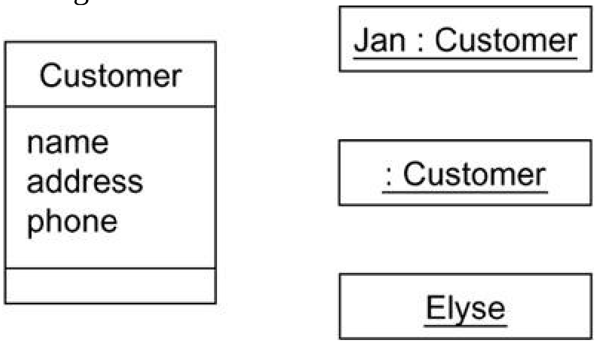
1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

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.

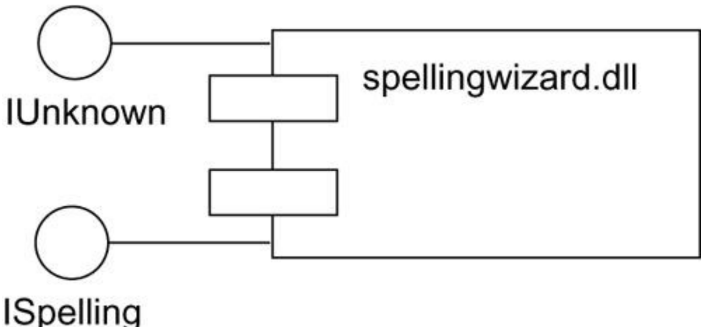
Adornments Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many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



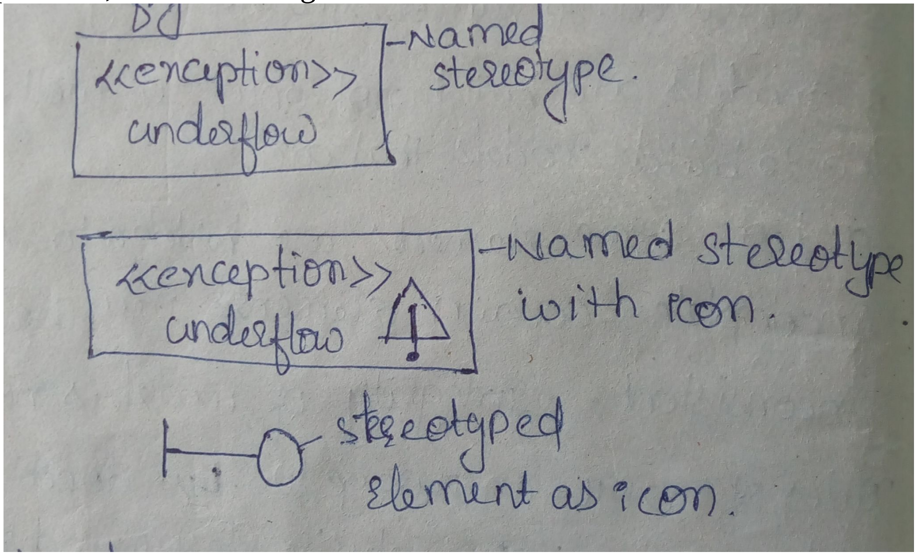
Extensibility Mechanisms

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)

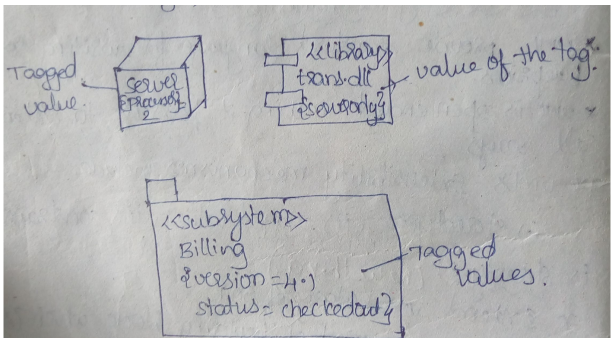
Stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. Stereotype is represented as a name enclosed by gulliments (<<name>>) and place over another element. It can be defined as icon, 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)

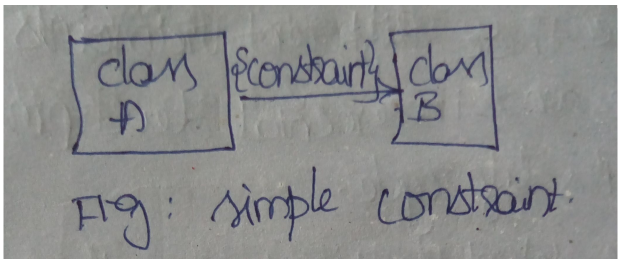
With stereotype new things can be added but with tagged values new properties can be added. For ex, when a project is released it is important to keep track version no. Here tagged values can be used to add this information to the models. The tag value applies to the element itself and not its instances. 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

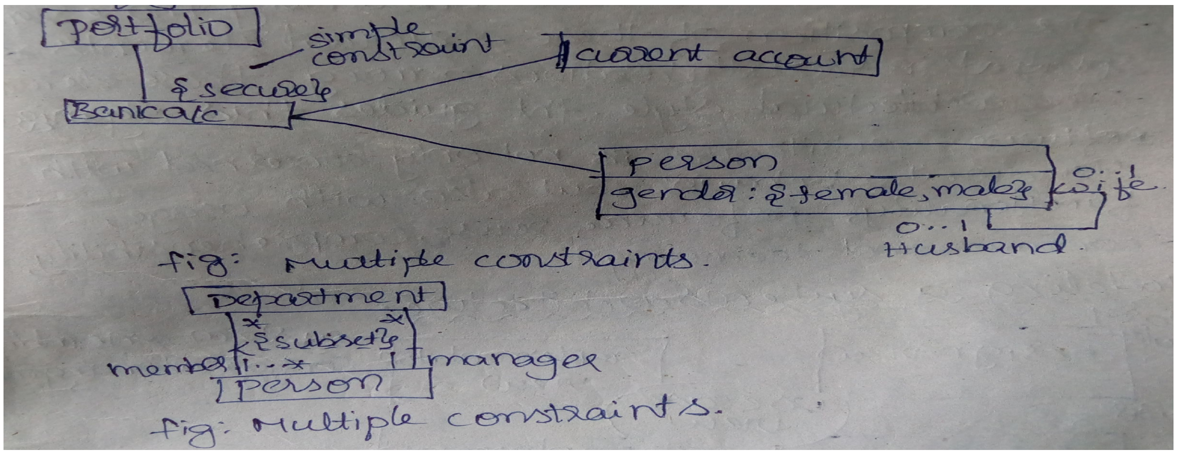

Tagged value
server {processor = 2}
<<library>> trans.dll {secondary}
value of the tag.
<<subsystem>> Billing {version = 4.1 status = checkedout}
Tagged values.

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.



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



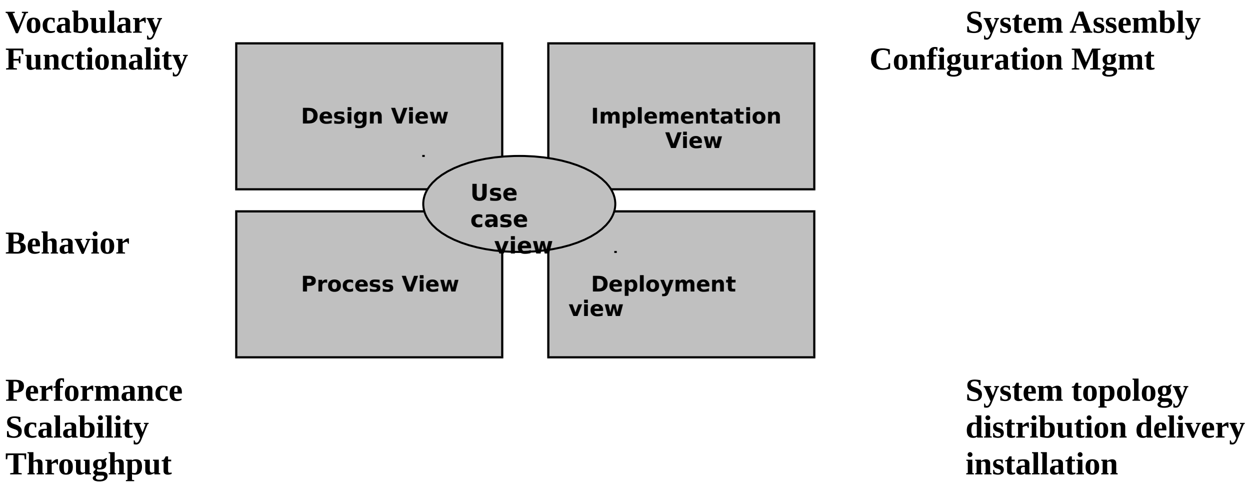
1.5. Architecture

0 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

0 - The organization of a software system
1 - The selection of the structural elements and their interfaces by which the system is
2 composed
3 - Their behavior, as specified in the collaborations among those elements
4 - The composition of these structural and behavioral elements into progressively larger
5 subsystems
6 - The architectural style that guides this organization: the static and dynamic elements and
7 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



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

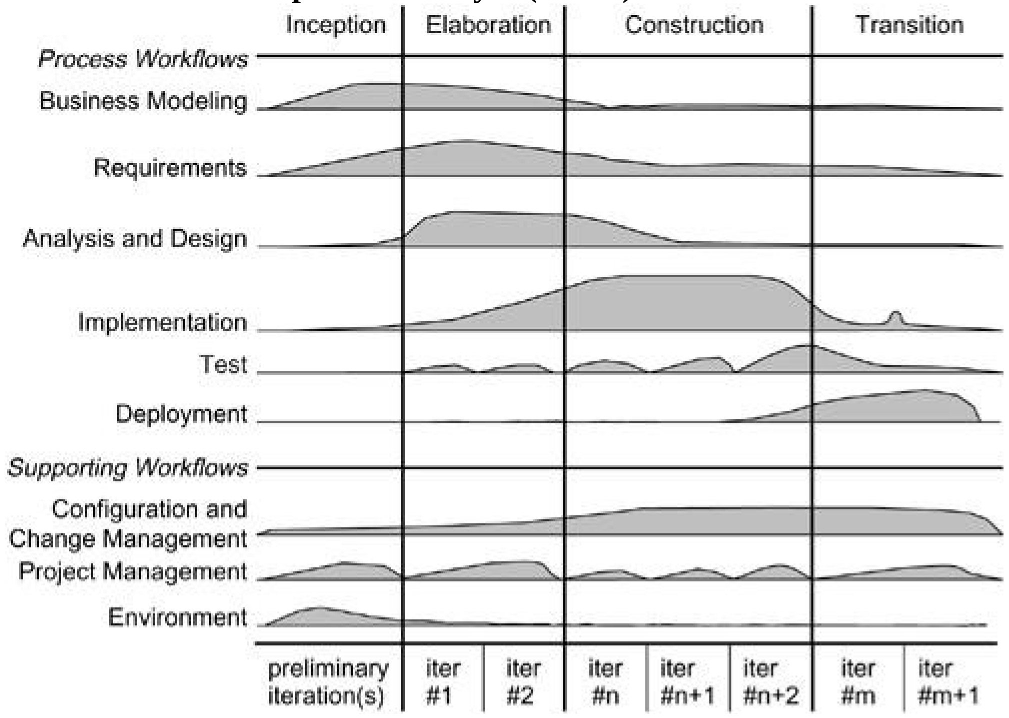
0 The implementation view of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various 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.