System Development – Object Basics – Development Life Cycle – Methodologies – Patterns – Frameworks – Unified Approach – UML

System Development (SD) : -

- SD refers to all activities that go into producing an information systems solution. These activities consist of systems analysis, modeling, design, implementation, testing and maintenance.
- A software development methodology is a series of processes that if followed can lead to the development of an application.
- According to *Niklaus Wirth* – A software system is a set of mechanisms for performing certain action on certain data.
- There are *two orthogonal views* of the software differs in their primary focus.
  1. The traditional approach focuses on the functions of the system and says software as a collection of programs (or functions) and isolated data.
  2. Object oriented systems development centers on the object, which combines data and functionality i.e., Programs = Algorithms + Data Structures.

Object Oriented System Development : -

- Object oriented systems development is a way to develop software by building self – contained modules or objects that can be easily replaced, modified and reused.
- In an object–oriented environment, software is a collection of discrete objects that encapsulate their data as well as the functionality of model real–world events "*objects*" and emphasizes its cooperative philosophy by allocating tasks among the objects of the applications.
- A *class* is an object oriented system carefully delineates between its interface (specifications of what the class can do) and the implementation of that interface (how the class does what it does).

Need of Object Orientation

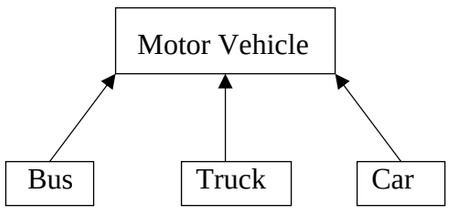
An object orientation produces systems that are easier to evolve, more flexible, more robust, and more reusable than a top – down approach. An object orientation

- *Allows higher level of abstraction*: Object oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) & functions (methods), they work at a higher level of abstraction. This makes designing, coding, testing & maintaining the system much simpler.
- *Provides Seamless transition among different phases of software development*: Object oriented approach, essentially uses the same language to talk about analysis, design, programming and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.
- *Encourage good development techniques*: In a properly designed system, the routines and attributes within a class are held together tightly, the classes will be grouped into subsystems but remain independently and therefore, changing one class has no impact on other classes and so, the impact is minimized.
- *Promotes of reusability*: Objects are reusable because they are modeled directly out of a real – world problem domain. Here classes are designed, with reuse as a constant background goal. All the previous functionality remains and can be reused without changed.

Object Basics: -

- *Objects are Grouped in Classes* : Class is a set of objects that share a common structure and a common behavior, a single object is simply an instance of a class. A class is a specification of structure (instance variables), behavior (methods) and inheritance for objects. A method or behavior of an object is defined by its class.
- *Attributes*: Object state and properties. Properties represent the state of an object. Often, we refer to the description of these properties rather than how they are represented in a particular programming language.
- *Object Behavior and Methods*: Behavior denotes the collection of methods that abstractly describes where an object is capable of doing. Methods encapsulate the behavior of the object, provide interfaces to an object and hide any of the internal structures and states maintained by the object.
- *Objects respond to Messages*: Message is the instruction and method is the implementation. An object understands a message when it can match message to a method that has same name as of it.
- *Encapsulation and Information Binding*: Information hiding is principle of concealing the internal data & procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings. Encapsulation protection deals with protection of an object capsule from other object by means of per–class protection and per–object protection.
- *Class Hierarchy*: At the top of the class hierarchy are the most general classes & at the bottom are the most specific. A subclass inherits all of the properties and methods defined in its super class.

Formal or abstract classes have no instances but define common behaviors that can be inherited by more specific classes.



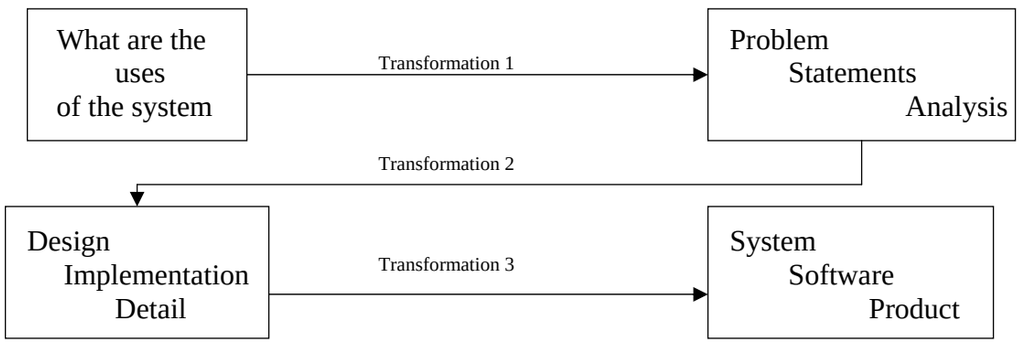
- *Inheritance*: It is the property of object–oriented systems that allows objects to be built from other objects & allows explicitly taking of commonality of objects when constructing new classes. Dynamic inheritance allows objects to change and evolve over time and refers to the ability to add, delete, or change parents from objects (or classes) at run time.
- *Multiple Inheritance*: Some object–oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class and is referred to as multiple inheritances.
- *Polymorphism*: It means that the same operation may behave differently on different classes. Booach defines it as the relationship of objects of many different classes by some common super class; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way.
- *Object Relationships*: Association represents the relationships between objects and classes. Associations are bidirectional with different annotations. Cardinality specifies how many instances of one class may relate to a single instance of an associated class. A special form of association is a consumer–producer relationship also known as client–server association or use-relationship and is can be viewed as one way interaction.

Association – bidirectional Consumer / Producer – Unidirectional

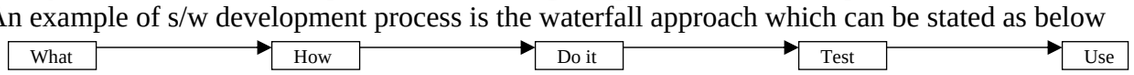
- *Aggregation and Object Containment*: As each object has an identity, one object can refer to other objects and is known as aggregation, where an attribute can be an object itself.

Object – Oriented System Development Life Cycle : -

- The basic software development life cycle consists of *analysis, design, implementation, testing and refinement*. Its main aim is to transform users' needs into a software solution.
- The development is a process of change, refinement, transformation or addition to the existing product. The software development process can be viewed as a series of transformations, where the output of one transformation becomes input of the subsequent transformation.



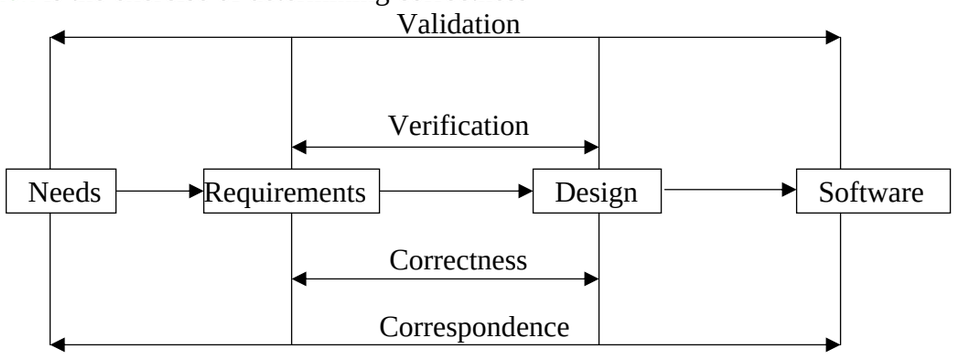
- Transformation 1 (*analysis*) translates the users' needs into system requirements & responsibilities.
- Transformation 2 (*design*) begins with a problem statement and ends with a detailed design that can be transformed into an operational system. It includes the bulk of s/w development activity.
- Transformation 3 (*implementation*) refines the detailed design into the system deployment that will satisfy the users' needs. It represents embedding s/w product within its operational environment.
- An example of s/w development process is the waterfall approach which can be stated as below



Building High–quality Software (BHS) : -

- *Ultimate goal* of BHS is user satisfaction. Blum describes a means of system evaluation in terms of four quality measures: correspondence, correctness, verification & validation.
- *Correspondence* measures how well the delivered system matches the needs of the operational environment as described in the original requirements statement.
- *Correctness* measures consistency of product requirements with respect to the design specification.

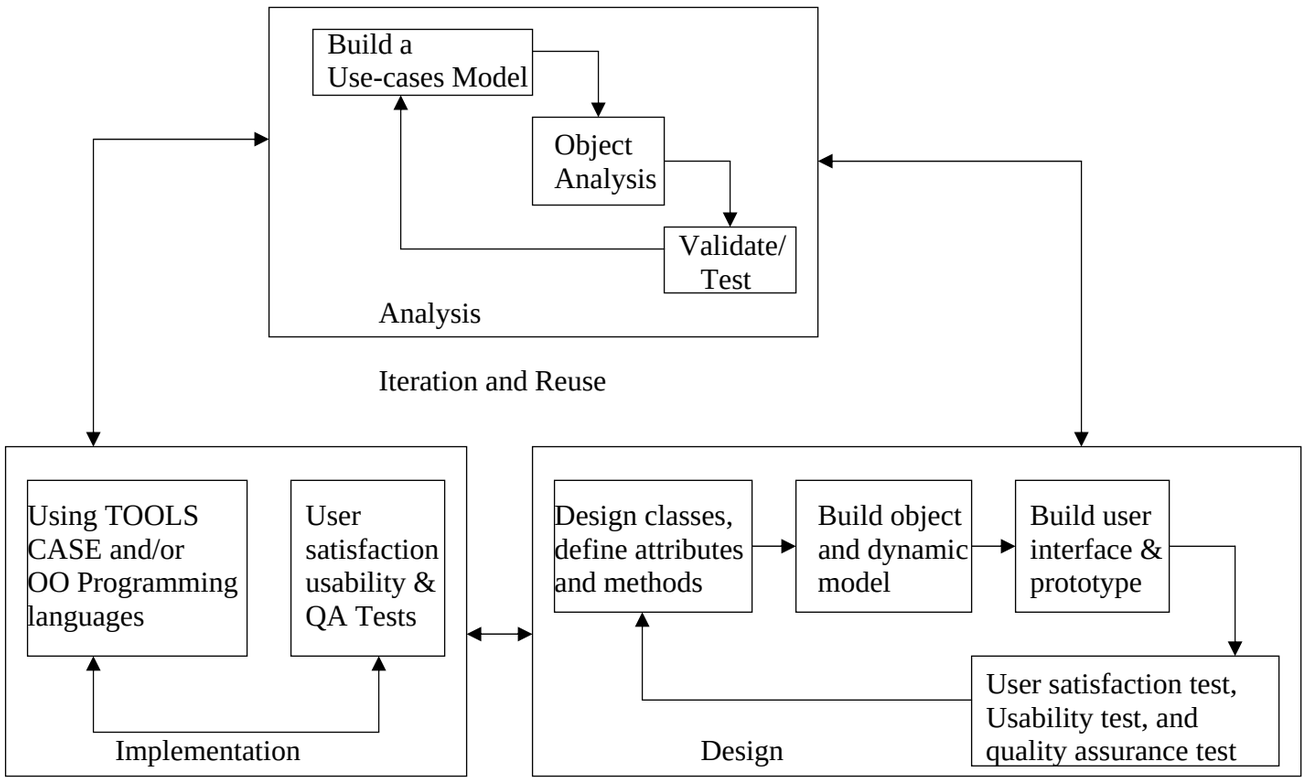
- *Verification* is the exercise of determining correctness



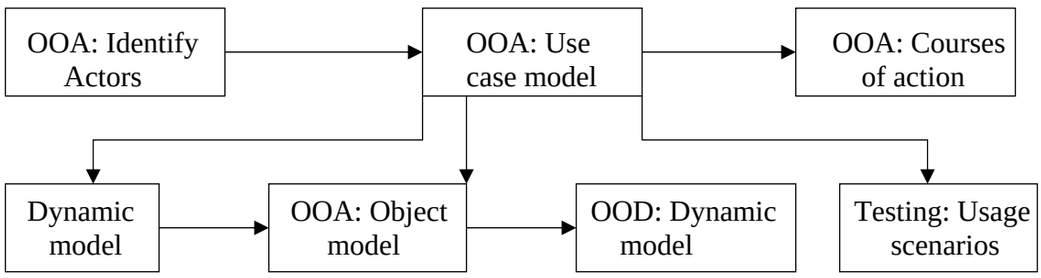
- *Validation* is the task of predicting correspondence. True correspondence cannot be determined until the system is in place.

Object Oriented System Development: A use Case Driven Approach : -

- The object oriented software development life cycle (*SDLC*) consists of *three macro processes*: object–oriented analysis, object–oriented design and object–oriented implementation.



- By following the life cycle model of *Jacobson, Ericsson and Jacobson*, once can produce designs that are traceable across requirements, analysis, design, implementation and testing. The main *advantage* is that all design decisions can be traced back directly to user requirements.



- Object–oriented system *development* includes object–oriented analysis, object–oriented design, Prototyping, Component–based development, Incremental testing.

Object–Oriented Analysis – Use case driven (OOA) : -

- This phase of s/w development is concerned with determining the system requirements and identifying classes and their relationship to other classes in the problem domain.
- Scenarios are a great way of examining who does what in the interactions among objects and what role they play; that is, their interrelationships. This intersection among objects' roles to achieve a given goal is called *collaboration*.

- In essence, a *use case* is a typical interaction between a user and system that captures users' goals and needs. Expressing high level processes & interactions with customers in scenario & analyzing it is referred to as use–case modeling. It represents users' view of the system or users' needs.
- Looking at the physical objects in the system also provides us important information on objects in the systems. The objects could be individuals, organizations, machines, units of information, pictures, or whatever else makes sense in the context of the real–world systems.
- In regarding documentation, *80 – 20 rule* is generally applies where 80 percent of the work can be done with 20 percent of the documentation. Good modeling implies good documentation.

Object–Oriented Design (OOD) : -

- The *goal of OOD* is to design the classes identified during the analysis phase and the user interface. Here, we identify & define additional objects, classes that support implementation of requirements.
- OOD is highly *incremental*. OOA can done, model it, create OOD them do some more on it.
- Here, we first, build the object model based on objects and their relationships, then iterate and refine the model such as
  - * Design and refine classes
  - * Design and refine attributes
  - * Design and refine methods
  - * Design and refine structures
  - * Design and refine associations.
- *Guidelines to use in OOD*:
  - * Reuse, rather than build, a new class. Know the existing classes
  - * Design a large number of simple classes, rather than a small number of complex classes
  - * Design methods
  - * Critique what you have proposed. If possible, go back and refine the classes.

Prototyping : -

- A *prototype* is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. It enables to fully understand how easy or difficult it will be to implement some of the features of the system.
- Prototyping can further define the use cases, and it actually makes use–case modeling much easier. The main idea is to build a prototype (use–case modeling) to design systems that users like & need.
- It provides the developer a means to test & refine user interface & increase usability of system.
- The following categories are some of accepted prototypes each having its own strength.
  - ⇒ *Horizontal Prototype*: It is a simulation of interface but contain no functionality. Advantages: Very quick to implement, good overall of system, user to evaluate interface on normal base
  - ⇒ *Vertical Prototype*: It is a subset of system features with complete functionality. Advantage: few implemented functions can be tested in great depth
  - ⇒ *Analysis Prototype*: It is an aid for exploring problem domain. It used to inform user & demonstrate proof of a concept
  - ⇒ *Domain Prototype*: It is an aid for incremental development of ultimate software development
- Prototyping should involve representation from all user groups that will be affected by the project, especially the end users and management members to ascertain that the general structure of the prototype meets the requirements established for the over all design.
- The *purpose of the review* is three fold are
  1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.
  2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.
  3. To give management and everyone connected with the project the first glimpse of what the technology can provide.
- Prototyping is a useful exercise at almost any stage of the development. It should be done in parallel with the preparation of the functional specification.

Implementation: Component Based Development (CBD): -

- Software components are built & tested in–house using a wide range of technologies. Computer–aided S/w Engineering (*CASE*) tools allow their users to rapidly develop information systems.
- The main *goal* of CASE technology is the automation of the entire information system's development life cycle process using a set of integrated software tools, such as modeling, methodology, and automatic code generation.
- CBD is an *industrialized approach* to the s/w development process. Application development moves from custom development to assembly of pre-built, pre-tested, reusable s/w components that operate with each other.
- *Two basic* ideas underlie CBD move application development from a craft activity to an industrial process fit to meet the needs of modern, highly dynamic, competitive, global businesses.

1. Application development can be improved significantly if application can be assembled quickly from prefabricated software components.
2. An increasingly large collection of interpretable s/w components could be made available to developers in both general and specialist catalogs.

- The *s/w components* are the functional units of a program, building blocks offering a collection of re-usable services. A s/w component can request a service from another component or deliver its own services on request.
- Rapid Application Development (*RAD*) is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods.
- The *main objective of RAD* is to build a version of an application rapidly to see whether we actually have understood the problem (analysis) & determines whether the system does what it is supposed to do (Design).

Incremental Testing : -

- The problem was that developers would turn over application to a quality assurance (QA) group for testing only after development was completed.
- In most cases testing an application for bugs and performance would waste of time and money. *Glenn Shimamoto*, vice president of technology and strategic planning at the New York Bank says "Our testing was very complete and good, but it was costing a lot of money and would add months onto a project". Thence, no clear picture of the system characteristics are been categorized.

Reusability: -

- A *major benefit of OOSD* is reusability and this is the most difficult promise to deliver on. For an really reusable object, more effort must be spent designing it.
- The potential benefits of *reuse are clear*: increased reliability, reduced time and cost for development, and improved consistency.
- For effective reusability of existing s/w components we are suppose ask two questions – Has my problem already been solved or partially solved? What has been done before to solve a problem similar to this one?
- The *reuse strategy* can be based on the following: -
  * Information hiding (encapsulation)
  * Conformance to naming standards
  * Creation and administration of an object repository.
  * Encouragement by strategic management of reuse as opposed to constant redevelopment.
  * Establishing targets for a percentage of the objects in the project to be reused. (50% of objects).

Object Oriented Methodologies : -

- Numbers of methodology are based on modeling the business problem and implementing the application in an object oriented fashion. The major differences between different methodologies lie primarily in the documentation of information, and modeling notations and language.
- The *three major methodologies* and their modeling notations developed by Rumbaugh et al., Booch and Jacobson which are the origins of the Unified Modeling Language.
- Each method has its strengths. *Rambaugh method* is well-suited for describing the object model or the static structure of the system. The *Jacobson et al. method* is good for producing user – driven analysis models. The *Booch method* produces detailed object–oriented design models.

Rambaugh et al.'s Object Modeling Technique(OMT) : -

- OMT presented by Jim Rambaugh and his co workers describes a method for the analysis, design and implementation of a system using an object-oriented technique.
- OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. Details such as class, attributes, method, inheritance and association also can be expressed easily.
- OMT consists of *four phases*, which can be performed iteratively.
  1. *Analysis*: The results are objects and dynamic and functional models
  2. *System Design*: The results are a structure of basic architecture of system along with high–level strategy decisions.
  3. *Object Design*: This phase produces a design document, consisting of detailed objects static, dynamic and functional models.
  4. *Implementation*: This activity produces reusable, extensible and robust code.
- OMT separates modeling into *three different parts*.
  1. An *object* model, presented by the object model and the data dictionary.
  2. A *dynamic* model presented by the state diagrams and even flow diagrams.
  3. A *functional* model presented by data flow and constraints.
- *Object Model*: It describes the structure of objects in a system: Their identity, relationships to other objects, attributes and operations. It is represented graphically with an object diagram which

contained classes interconnected by association lines. Each class contains a set of individual objects and association lines establish relationships among the class.

- *OMT Dynamic Model*: It provides a detailed and comprehensive dynamic model, in addition to letting us depict states, transitions, events and actions. The OMT state transition diagram is a network of states and events. Each state receives one or more events, at which time it makes the transition to the next state whereas the next state depends on current state as well as events.
- *OMT Functional Model*: OMT data flow diagram (DFD) shows the flow of data between difference processes in a business. It also provides a simple and intuitive method for describing business process without focusing on the details of the computer system. DFD uses 4 primary symbols
  1. The process is any function being performed.
  2. The data flow shows the direction of data element movement.
  3. The data store is a location where data are stored
  4. An external entity is a source or destination of a data element

Booch Methodology : -

- It is a widely used object oriented method that helps us design our system using the object paradigm. It covers the analysis and design phases of an object oriented system.
- We start with class & object diagrams in analysis phase and refine these diagrams in various steps. The Booch method consists of the following *diagrams* : Class diagrams, Object diagrams, State Transition diagrams, Module diagrams Process diagrams, Interaction programs.
- *The Macro Development process*: The macro process serves as a controlling framework for micro process & its main concern is technical management of system. It consists of following steps:
  1. *Conceptualization*: Core requirements, goal of system & Prototype to prove the concept.
  2. *Analysis & development of model*: Class, object & interaction diagrams for roles & responses
  3. *Design/create sys., architecture*: Schedules for multiple processes on each relevant processor
  4. *Evolution or implementation*: Refine through iterations and a stream of s/w implementations
  5. *Maintenance*: Make localized changes to system to add new requirements & eliminate bugs
- *The Micro Development process*: It is a description of the day–to–day activities by a single or small group of s/w developers which could look blurry to an outside viewer as analysis & design phases are not clearly defined.
  1. Identify classes and objects
  2. Identify class and object semantics.
  3. Identify class & object relationships,
  4. Identify class & object interfaces & implementation.

Jacobson Et Al. Methodology: -

- It consists of *OOBE , OOSE* cover the entire life cycle and stress traceability between different phases, both forward and backward. This traceability enables reuse of analysis & design work, possibly much bigger factors in the reduction of development time than reuse of code.
- *Use Cases*: At the heart of these methodologies is the use – case concept, which evolved with objectory (Object Factory for S/w Development). A use case is interaction between users and a system. It captures goal of user & responsibility of system to its user.
- The use cases are described as one of the following:
  * Non formal text with no clear flow of events
  * Text, easy to read but with a clear flow of events to flow (recommended style)
  * Formal style using pseudo code.
- The use case description *must contain*
  * How and when the use case begins and ends
  * Interaction between use case & its actors including when interaction occurs? What is exchanged?
  * How and when use case will need data stored in system or will store data in system.
  * Exceptions to the flow of events.
  * How and when concepts of the problem domain are handled.
- An abstract use case is not complete & has no actors that initiate it bust is used by another use case. This inheritance could be used in several levels. It has ones that have uses or extends relationships.
- *Object Oriented Software Engineering* (*OOSE*) : It is also called as objectory is a method of object – oriented development with the specific aim to fit the development of large, real – time systems.
- The development process called *use–case driven development* stresses that use cases are involved in several phases of the development including analysis, design, validation and testing. The use – case scenario begins with a user of the system initiating a sequence of interrelated events.
- *Objectory* has been developed and applied to numerous application areas and embodied in the CASE tool systems. Objectory is built around several different models:
  * *Use–case model*: It defines outside (actors) and inside (use case) of the system's behavior
  * *Domain object model*: The objects of the real world are mapped into the domain object model
  * *Analysis object model*: It presents how the source code (implementation) should be carried out
  * *Implementation model*: It represents the implementation of the system

* *Test model*: It constitutes the test plans, specifications and reports

- *Object–Oriented Business Engineering* (*OOBE*) : It is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the s/w engineering process. It has the following Phases in its course of development.
  - *Analysis phase*: It defines the system to be built in terms of the problem-domain object model, requirements model, analysis model. It reduces the complexity & promotes maintainability over life of the system. Jacobson suggest prototyping with a tool might be useful during this phase to help specify user interfaces.
  - *Design & implementation phases*: Implementation environment must be identified for design model. It includes such as DBMS, distribution process, constraints due to the programming language, available component libraries and incorporation of graphical user interface tools.
  - *Testing phase*: Finally, Jacobson describes several testing levels and techniques. The levels include unit testing, integration testing and system testing.

Patterns: -

- A *pattern* is instructive information that captures essential structure, insight of a successful family of proven solutions to a recurring problem that arises within a certain context & system of forces.
- A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints. A *good pattern* will do the following:
  - It solves a problem. Patterns capture solutions, not just abstract principles or strategies
  - It is a proven concept. Patterns capture solutions with a tract record, not theories or speculation
  - The solution is not obvious. Best patterns generate a solution to problem indirectly in design
  - It describes a relationship. Patterns describe deeper system structures and mechanisms
  - The pattern has a significant human component. All s/w serves human comfort or quality of life; best patterns explicitly appeal to aesthetics and utility.
- *Generative and Non-generative patterns*: Patterns that describes recurring problem along with generating way is generative patterns and others which are static and passive are non-generative patterns. Generative patterns shows characteristics of good systems & teaches to build them
- *Pattern Templates*: Pattern should be explained "in the form of rule [template] which establishes a relationship between a context, a system of forces which arises in that context and configuration which allows these forces to resolve themselves in that context". The following essential components be clearly mentioned:
  - ⇒ *Name*: A meaningful name must be given along with nickname if exists
  - ⇒ *Problem*: A statement describing goals, objectives to reach within given context & forces
  - ⇒ *Context*: Preconditions of problem recurring & desirable solution stating applicability
  - ⇒ *Forces*: Reveal Motivation factors for s/w necessity & encapsulate all forces impact on it
  - ⇒ *Solution*: It should describe static structure & dynamic behavior
  - ⇒ *Examples*: It may be supplemented by sample implementation to show one way to get solution
  - ⇒ *Resulting Context*: Has post-conditions, side effects of pattern said as resolution of forces
  - ⇒ *Rationale*: Provides insight into deep structures & key mechanisms going deep into system
  - ⇒ *Related Patterns*: Must have static & dynamic relationships between types of similar patterns
  - ⇒ *Known uses*: Occurrences of pattern , its application and proven solution of recurring problem
- A good pattern often begins with an abstract that provides a short summary or overview. This gives readers a clear picture of pattern and quickly informs them of its relevance to any problems they may solve sometimes called a thumbnail sketch of pattern or pattern thumbnail
- *Antipatterns*: It represents worst practice or a lesson learned come in two varieties those describing
  1. a bad solution to a problem that resulted in a bad situation
  2. how to get out of a bad situation & how to proceed from there to good ones
- *Capturing Patterns*: Process of looking for patterns to document is pattern mining or reverse architecting and guidelines and certain guidelines to get pattern of such type are
  - ⇒ *Focus on practicability*: Patterns should describe proven solutions to recurring problems
  - ⇒ *Aggressive disregard of originality*: Patterns writers need to be original inventor of document
  - ⇒ *Non-anonymous review*: Pattern submitted to shepherd & they discuss with authors
  - ⇒ *Writers' workshops instead of presentations*: Patterns are discussed in workshops to improve
  - ⇒ *Careful editing*: Authors have opportunity to improve in shepherd comments & insights

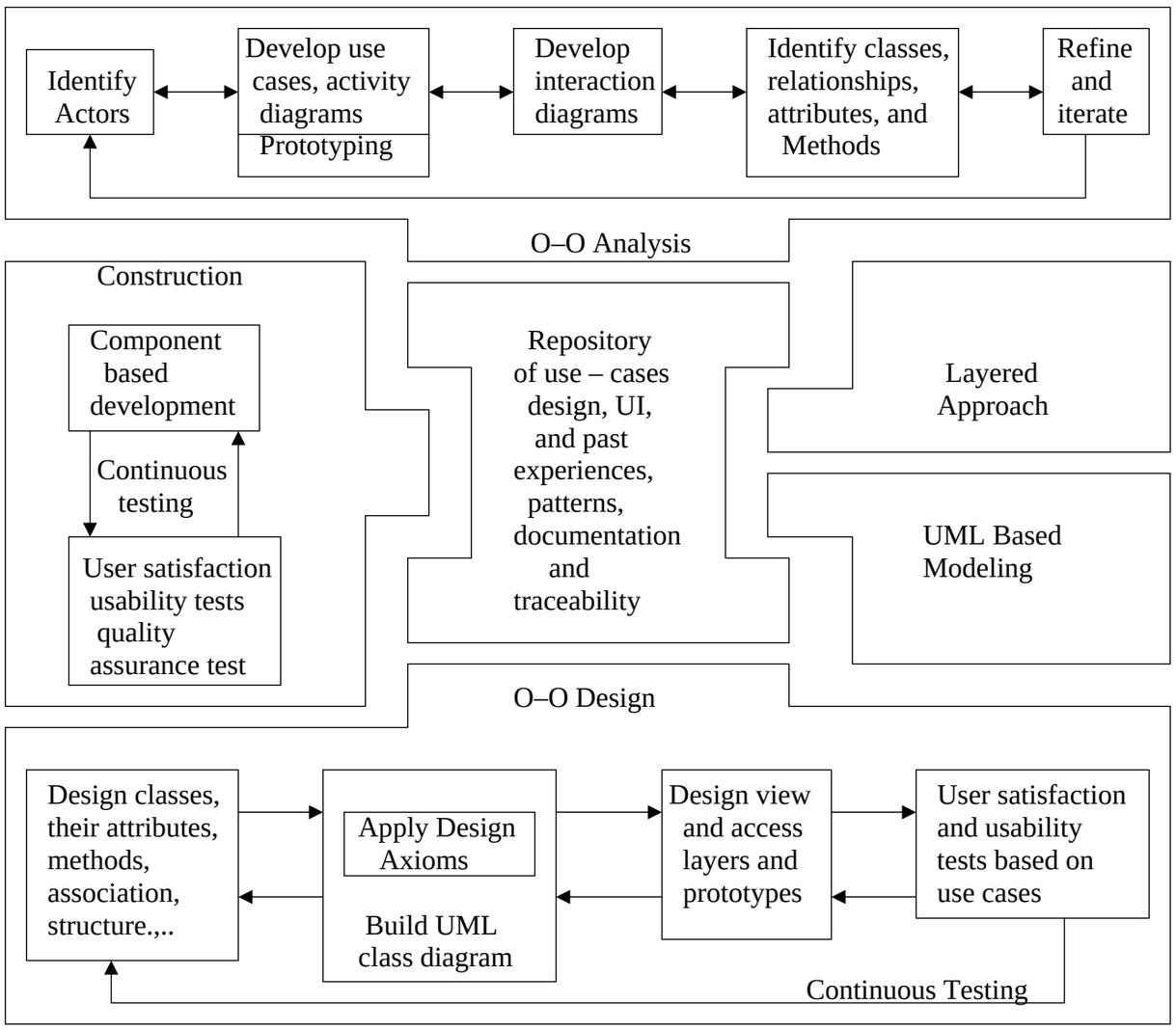
Frameworks: -

- A *framework* is a way of presenting a generic solution to a problem that can be applied to all levels in development. An object – oriented software framework is a set of cooperating classes that made up a reusable design for a specific class of software.

- It provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities & collaborations. A developer customizes a framework to a particular application by sub-classing & composing instances of framework classes
- The framework captures design decisions that are common to its application domain. It thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses we put to work immediately
- *Differences* between framework & patterns are as follows
  * Design patterns are more abstract than frameworks
  * Design patterns are smaller architectural elements than frameworks
  * Design patterns are less specialized than frameworks

Unified Approach (UA) : -

- UA is a methodology for software development is based on methodologies by Booch, Rumbaugh and Jacobson. It tries to combine best practices, processes, and guidelines along with UML notations & diagrams for better understanding object – oriented concepts and system development
- UA to s/w development revolves around the following process & concepts – Use–case driven development, Object–oriented analysis, Object– oriented design, Incremental development and prototyping and Continuous testing.
- The methods and methodology employed include UML based modeling, Layered approach, Repository for OOSD patterns and frameworks and CBD technology
- The *diagrammatic representation of processes & components of unified approach*



- *Object – Oriented Analysis* (*OOA*): The goal is first to find domain of the problem and system's responsibilities by knowing all user needs which is accomplished by models. These models concentrate on describing what the system does rather than how does it. OOA has the following steps: 1. Identify the Actors, 2. Develop a simple business process model using UML Activity diagram, 3. Develop the Use Case, 4. Develop interaction diagrams, 5. Identify classes
- *Object – Oriented Design* (*OOD*): UA combines Jacobson's analysis and interaction diagrams, Booch's object diagrams & Rumbaugh's domain models to get good design. OOD consists of
  * Designing classes, attributes, methods, associations, structures & protocols, apply design axioms
  * Design the Access Layer * Design and prototype User interface
  * User satisfaction and Usability Tests based on Usage / Use cases
  * Iterate and refine the design

- *Iterative Development & Continuous Testing*: Iteration is done by reprototyping and retesting. During iterative process, prototypes will be incrementally transformed into actual application. UA encourages the integration of testing plans from 1st day of project i.e., usability test
- *UML Modeling*: UML merges best of notations used by three most popular analysis and design methodologies: Booch's methodology, Jacobson and Rumbaugh's object modeling technique. UA uses UML to describe and model the analysis and design phases of system development
- *UA Proposed Repository*: The idea promoted is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks and user interfaces in an easily accessible manner with a completely available and easily utilized format
- *Layered approach to s/w development*: In general there exists 2 – layered architecture similar applied in client – server architecture constitutes of interface and data layers, 3- layered architecture constitute Access layer, Business layer and View layer.
  - ✧ *Business Layer*: It contains all objects that represent the business. Responsibilities is straightforward: Model objects of business & how they interact to accomplish business process
  - ✧ *View* (*User Interface Layer*): It consists of objects with which the user interacts as well as objects needed to manage or control the interface. This layer is responsible for two major aspects: * Responding to user interaction * Displaying business objects
  - ✧ *Access Layer*: It contains objects that know how to communicate with place where the data actually reside & it has two responsibilities: * Translate request *Translate results

Modeling: -

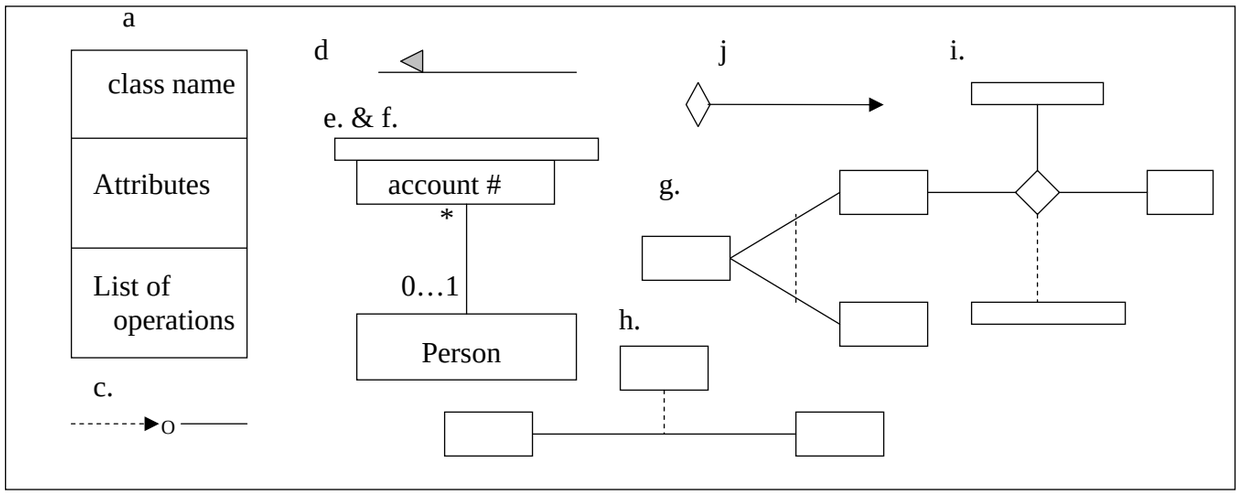
- A model is an *abstract representation* of a system constructed to understand the system prior to building or modifying it. The term system refers to any process of structure. As a general aspects, models can represent static or dynamic situations
- *Static Model*: It can be viewed as a snapshot of a system's parameters at rest or at a specific point in time. It is used to represent structural or static aspect of system. It assume stability and absence of change in data over time. E.g: UML – class diagram
- *Dynamic Model*: It can viewed as a collection as collection of procedures or behaviors taken together which reflect behavior of system over time. Dynamic relationships show the business objects interaction in performing task. Dynamic modeling is most useful during design and implementation phases of system development. E.g.,: UML – interaction diagram
- *Need of modeling*: A model for s/w is similar to blueprint for building. Good models are essential for communication among project teams. A modeling language must include
  - *Model elements* – fundamental modeling concepts and semantics
  - *Notation* – visual rendering of model elements
  - *Guidelines* – expression of usage within trade
- The use of visual notation to represent or model a problem can provide us several benefits relating to *clarity*, *familiarity*, *maintenance* and *simplification*. Turban states following advantages of modeling:
  1. Models make it easier to express complex ideas
  2. Reduction of complexity makes easier understanding of complex situations
  3. Models enhance and reinforce learning and training
  4. Cost of modeling analysis is much lower than experimentation on real system
  5. Manipulation of model (changing variables) is easier than in real system

Unified Modeling Language (UML) : -

- UML is a language for specifying, constructing, visualizing and documenting the s/w system and its components. UML is a graphical language with set of rules & semantics expressed in English in form known as *object constraint language* (OCL)
- The *goals of unification* of all modeling concepts are to cast away elements of existing Booch, OMT & OOSE methods that didn't work in practice, to add elements from other methods that are more effective and to invent new methods only when an existing solution was unavailable
- The *primary goals* of design of UML are as follows:
  1. Provides users a ready–to–use, expressive visual modeling language so they can develop and exchange meaningful models
  2. Provide extensibility and specialization mechanisms to extend the core concepts
  3. Be independent of particular programming languages and development processes
  4. Provided a formal basis for understanding the modeling language
  5. Encourage the growth of OO tools market
  6. Support higher – level development concepts
  7. Integrate best practices and methodologies
- *UML Diagrams*: Every complex system is best approached through small set of nearly independent views of a model with number of diagrams at each development level. UML defines *9 graphical diagrams*: 1. Class diagram (static diagram), 2. Use–case diagram,

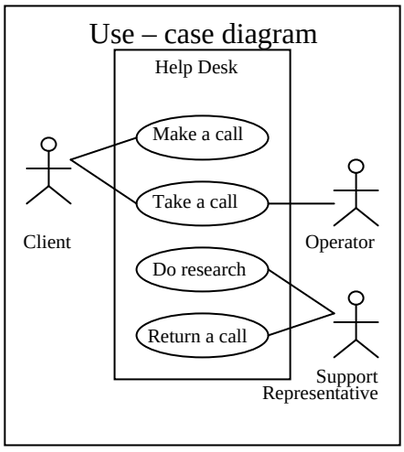
3. Behavior diagram (dynamic),
   a. Interaction diagram – Sequence diagram, Collaboration diagram
   b. State chart diagram, c. Activity diagram
4. Implementation diagram : Component diagram, Deployment diagram

- *Class diagram*: It is referred to object modeling, is main static structure analysis diagram for system. It represents the class structure of a system with relationships between classes and inheritance structure. It is developed through use–case, sequence and collaboration diagrams
  a. *Class notation*: Drawn as rectangle with three components – name, attributes, list of operations
  b. *Object diagram*: A static object diagram is an instance of class diagram. It shows as snapshot of detailed state of system at a point in time. Notation is same as class notation
  c. *Class Interface* notation is used to describe externally visible behavior of a class. It is a design activity and its notation is doted arrow attached to small circle
  d. *Association Role*: A simple association is binary association drawn as solid line connector
  e. *Qualifier*: It is an association attribute. Eg: A person object associated to Bank object and an attribute of this association is account# which is a qualifier of this association
  f. *Multiplicity*: It specifies range of allowable associated classes. It is given for roles with associations, parts within compositions, repetitions denoted by lower bound ……. Upper bound
  g. *OR Association*: Any instance of class participates in, at most, one of associations at some time. It is denoted by dashed line connecting two associations with {or} labeling
  h. *Association class*: It is an association that also has class properties. It is shown as class symbol attached by dashed line to an association line
  i. *N–Ary Association*: An association between more than one class & shown in large diamond
  j. *Generalization*: It is relationship between a more general class and more specific class. It is displayed as a directed line with a closed, hollow arrowhead at super class end

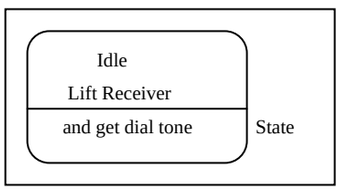


- *Use–case diagram*: It captures information on how the system or business works or how you wish it to work. It is a scenario – building approach in which you model the processes of the system. It is an excellent way to lead into OOA of the system.

  A use – case diagram is a graph of actors, as set of use cases enclosed by a system boundary, communication associations between the actors and use cases and generalization among use cases. 3 relationships shown are communication, uses and extends



- *Sequence diagram*: It is for dynamic modeling, where objects are presented by vertical lines, messages passed back & forth between objects are modeled by horizontal vectors between objects
- *Collaboration diagram*: An alternative view of sequence diagram, showing in a scenario how objects interrelate with one another



- *State chart diagram*: It is another form of dynamic modeling, focus on events occurring within a single object as it responds to messages.
- *Activity diagram*: It is used to model an entire business process. Thus, an activity model can represent several different classes
- *Implementation diagram*: It show the implementation phase of systems development, such as source code and run –time implementation structures. Implementation diagram shows structure of code itself and deployment diagram diagrams show the structure of run-time system