Unit 1
Introduction

1. Object Oriented Analysis Design (OOAD):

Object-Oriented Analysis and Design (OOAD) is a popular technical approach for analyzing and designing an application, system, or business by applying the object-oriented programming

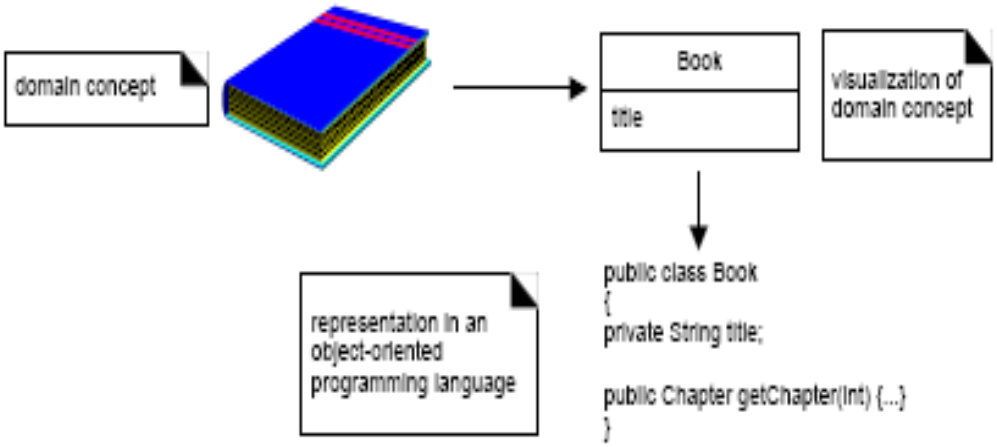
- uses visual modelling throughout the development life cycles for better stakeholder communication and product quality
- OOAD is best conducted in an iterative and incremental way

Object-Oriented Analysis:

Analysis: emphasizes investigation of problem and requirements rather than solution. Analysis is a broad term, best qualified as *requirements analysis* or *Object-oriented analysis*
During Object-Oriented Analysis, these is emphasis on finding and describing objects or concepts in the problem domain. Ex: Book, Library, Patron in Library information system

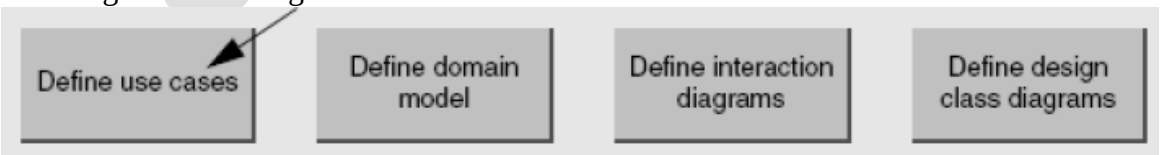
Object-Oriented Design (OOD):

Design: emphasizes a conceptual solution in software or hardware that fulfils requirements, rather than its implementation, best qualified as *Object-oriented design* or *data base design*
During Object-Oriented Design (OOD), there is emphasis on defining software objects and how they collaborate or interact to fulfil requirements. Ex: Book software object may have *title* attribute and *getChapter* method



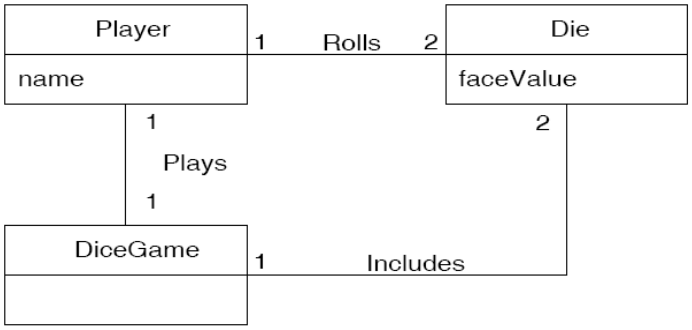
Simple Play a Dice Game:

A player picks up and rolls two dies, if the total is sever they win, otherwise they lose. Key steps and diagrams for this game are:



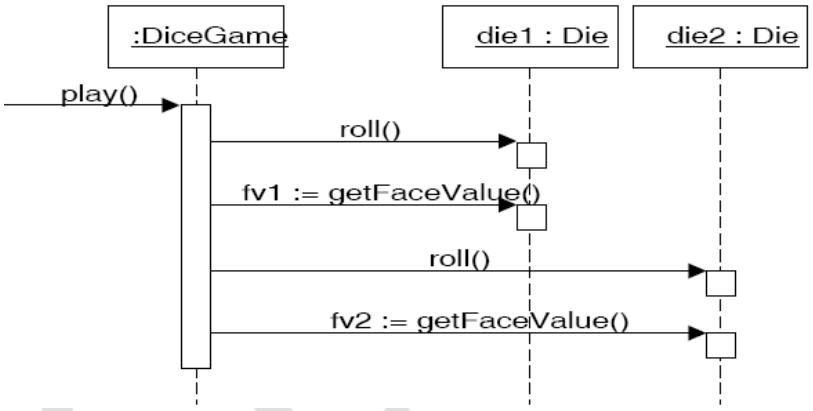
1. **Define Use cases:** use case is popular tool in requirements analysis, which may include a description of related domain processes known as Use cases Ex: *Play a Dice Game* use case
2. **Define Domain model:** Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects
   - A decomposition of the domain involves an identification of concepts, attributes, and associations that are considered

- The result can be expressed in a domain model, which is illustrated in a set of diagrams that show domain concepts or objects.

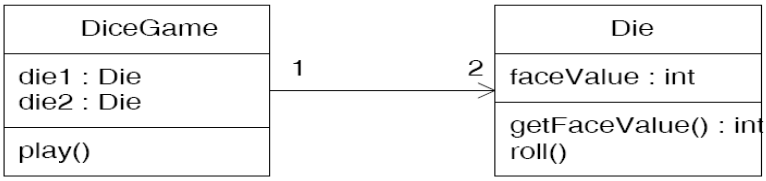


- This model illustrates concepts *Player*, *Die*, and *DiceGame*, with their associations and attributes

3. **Define Interaction diagrams:** Object-oriented design is concerned with defining software objects and their collaborations.
   - A common notation to illustrate these collaborations is the interaction diagram
   - It shows the flow of messages between software objects, and thus the invocation of methods



   - It illustrates the essential step of playing, by sending messages to instances of the *DiceGame* and *Die* classes
4. **Define Design class diagram:** In addition to a dynamic view of collaborating objects shown in interaction diagrams, it is useful to create a static view of the class definitions with a design class diagram. This illustrates the attributes and methods of the classes.



   - Since a play message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* methods

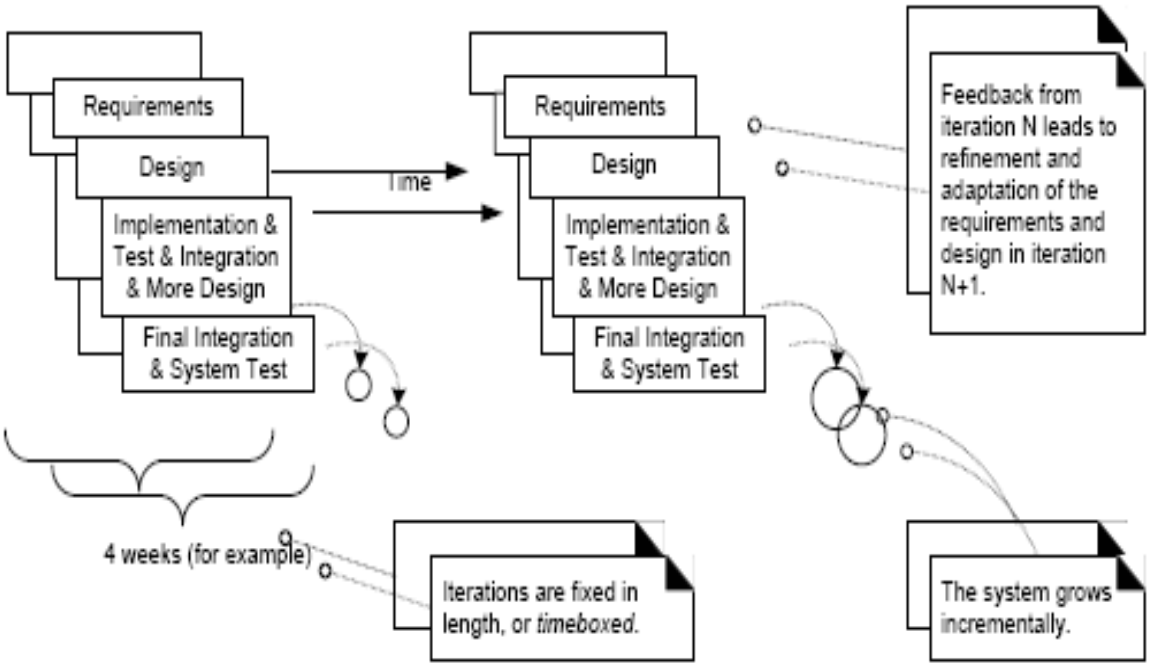
Interactive development and the Unified Process:

Iterative development is a skilful approach to software development and heart of OOAD. The Unified Process is an example iterative process. Informally, a software development process describes an approach to building, deploying, and possibly maintaining software.

Unified Process (UP) has emerged as a popular software development process for building object-oriented systems. In particular, Rational Unified Process or RUP a detailed refinement of the Unified Process has been widely adopted.

The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a

suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as Iterative and Incremental development.



The result of each iteration is an executable but incomplete system; it is not ready to deliver into production. The system may not be eligible for production deployment until after much iteration; Ex: 10 or 15 iterations. In general, each iteration tackles new requirements and incrementally extends the system. Iteration may occasionally revisit existing software and improve it; Ex: one iteration may focus on improving the performance of a subsystem, rather than extending it with new features.

Benefits of iterative development include:

- early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
- early visible progress
- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
- the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

Iteration length and time boxing:

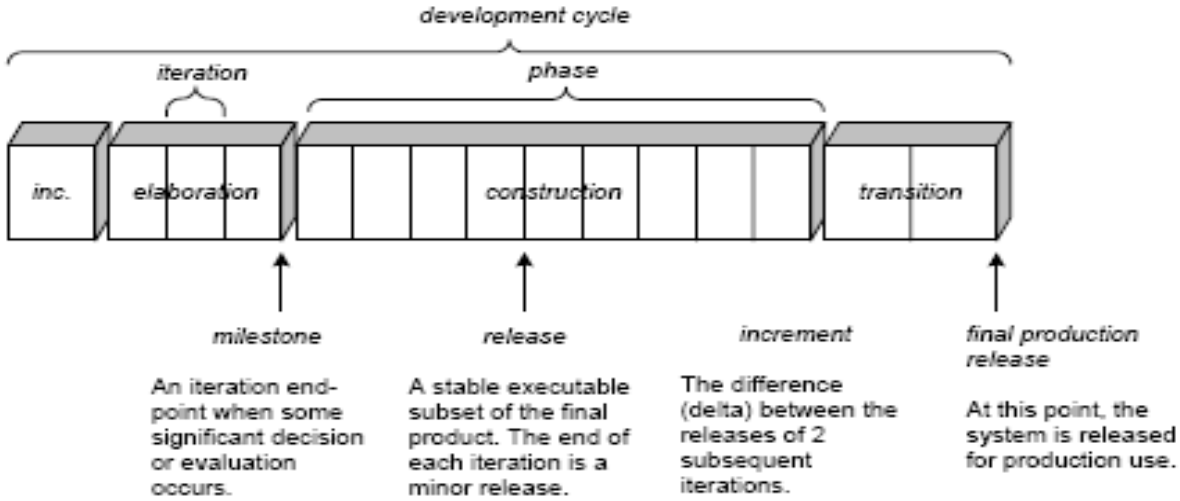
- The UP recommends an iteration length between two and six weeks
- Small steps, rapid feedback, and adaptation are central ideas in iterative development
- Much less than two weeks, and it is difficult to complete sufficient work to get meaningful throughput and feedback
- Much more than six or eight weeks, and the complexity becomes rather overwhelming, and feedback is delayed
- A very long iteration misses the point of iterative development. Short is good.
- A key idea is that iterations are time boxed, or fixed in length
- Massive teams (for example, several hundred developers) may require longer than six-week iterations to compensate for the overhead of coordination and communication; but no more than three to six months is recommended Ex: the successful replacement in the 1990s of the Canadian air traffic control system was developed with an iterative lifecycle and other UP practices.

- Six-month iteration is the exception for massive teams, not the rule

Unified Process Phases:

A Unified Process project organizes the work and iterations across four major phases:

1. **Inception:** approximate vision, business case, scope, vague estimates
2. **Elaboration:** refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates
3. **Construction:** iterative implementation of the remaining lower risk and easier elements, and preparation for deployment
4. **Transition:** beta tests, deployment



This is not the old "waterfall" or sequential lifecycle of first defining all the requirements, and then doing all or most of the design.

Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being—at least internally—sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and base-lined. A system's requirements may range from general vision statements to precise evaluation criteria, each specify particular functional or non-functional behaviour and each providing a basis for testing.

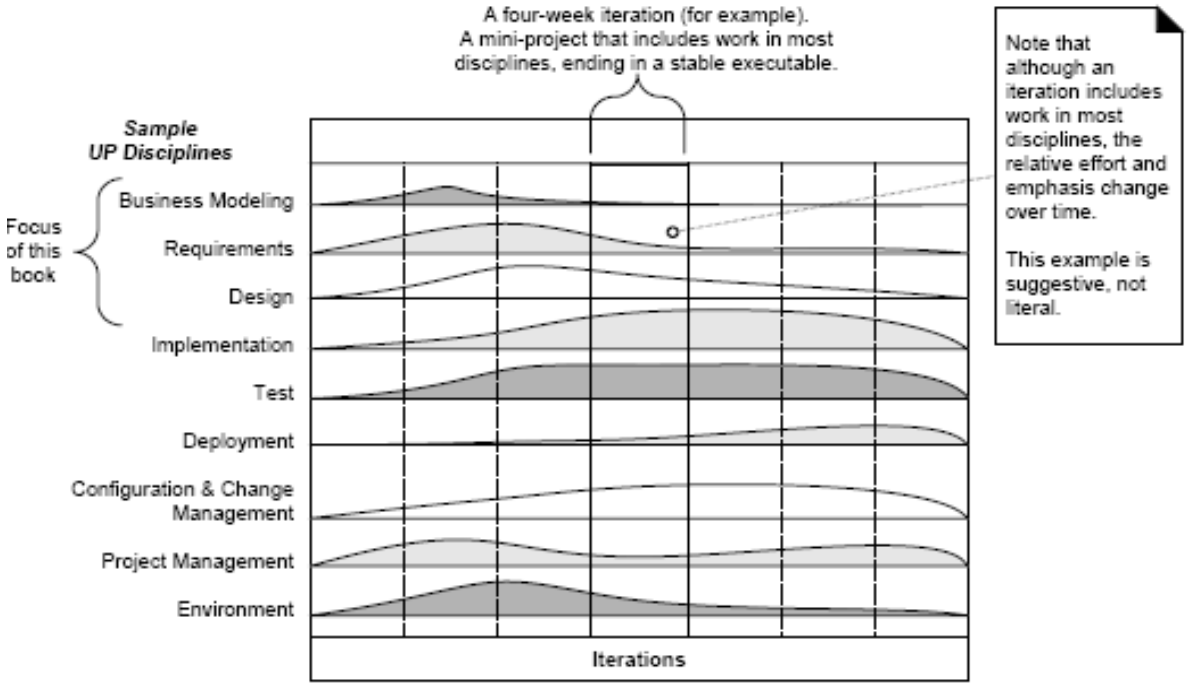
Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly re-examined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

Typical activities / workflows / disciplines in OOAD:

Unified Process Disciplines:

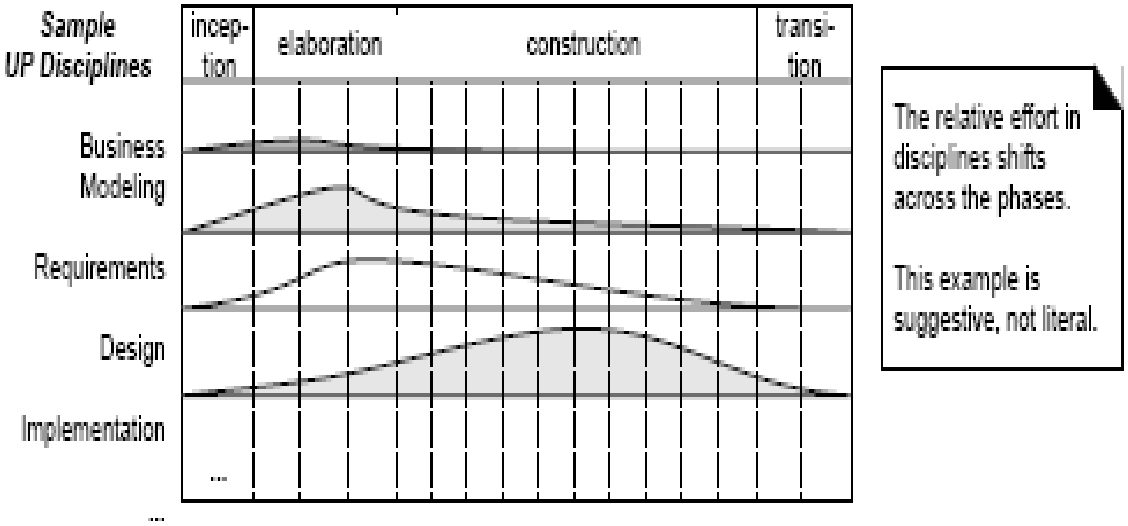
The UP describes work activities, such as writing a use case, within disciplines known as workflows. Informally, a discipline is a set of activities and related artifacts in one subject area, such as the activities within requirements analysis. **Artifact** is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on.



There are several disciplines in the UP:

- **Business Modelling**—when developing a single application, this includes domain object modelling. When engaged in large-scale business analysis or business process reengineering, this includes dynamic modelling of the business processes across the entire enterprise.
- **Requirements**— requirements analyses for an application, such as writing use cases and identifying non-functional requirements.
- **Design**—All aspects of design, including the overall architecture, objects, databases, networking, and the like.

Disciplines and Phases:

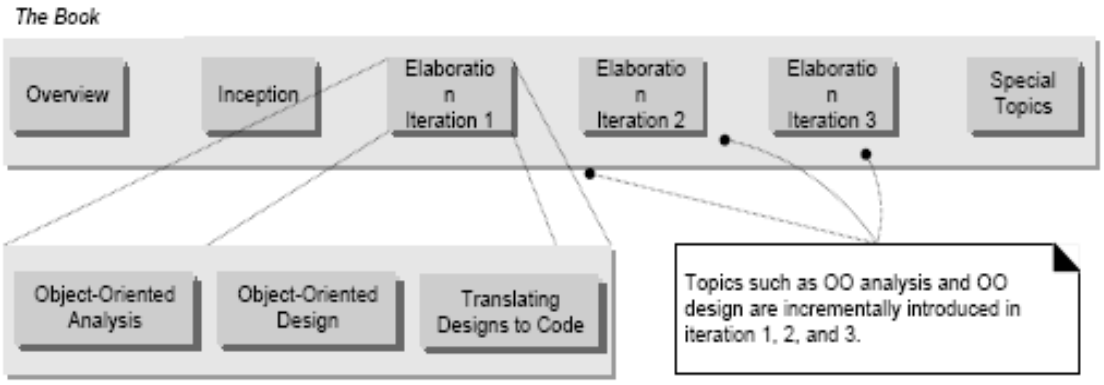


During, one iteration work goes on in most or all disciplines; however, the relative effort across these disciplines changes over time. Early iterations naturally tend to apply greater relative emphasis to requirements and design, and later ones less so, as the requirements and core design stabilize through a process of feedback and adaptation. Relating this to the UP phases (inception, elaboration, ...) as shown in figure the changing relative effort with respect to the phases. In elaboration, for example, the iterations tend to have a relatively high level of requirements and design

work, although definitely some implementation as well. During construction, the emphasis is heavier on implementation and lighter on requirements analysis.

Book structure and UP phases and disciplines:
Above figure describe the organization with respect to the UP phases:

1. The inception phase chapters introduce the basics of requirements analysis
2. Iteration 1 introduces fundamental OOAD and how to assign responsibilities to objects
3. Iteration 2 focuses on object design, especially on introducing some high-use design patterns
4. Iteration 3 introduces a variety of subjects, such as architectural analysis and framework design



Unified Modelling Language (UML)

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

- UML stands for **Unified Modelling 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.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- 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.
- UML has a direct relation with object oriented analysis and design.

A Conceptual Model of UML

A conceptual model can be defined as a model which is made of concepts and their relationships. A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other. As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

1. UML building blocks
2. Rules to connect the building blocks
3. Common mechanisms of UML

1. UML - Building Blocks

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. UML The building blocks of UML can be defined as −

- Things
- Relationships
- Diagrams

1.1 Things

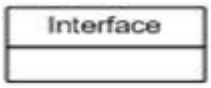
Things are the most important building blocks of UML. Things can be −

- Structural
- Behavioural
- Grouping
- Annotational

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

1. **Class** − Class represents a set of objects having similar responsibilities.

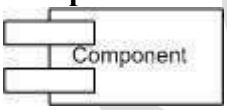
2. **Interface** − Interface defines a set of operations, which specify the responsibility of a class.



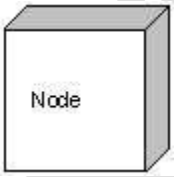
3. **Collaboration** −Collaboration defines an interaction between elements.

4. **Use case** −Use case represents a set of actions performed by a system for a specific goal.

5. **Component** −Component describes the physical part of a system.



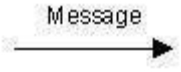
6. **Node** − A node can be defined as a physical element that exists at run time.



Behavioural Things

A behavioural thing consists of the dynamic parts of UML models. Following are the behavioural things −

1. **Interaction** − Interaction is defined as a behaviour that consists of a group of messages exchanged among elements to accomplish a specific task.

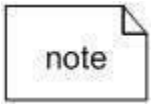


2. **State machine** − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

1. **Package** − Package is the only one grouping thing available for gathering structural and behavioural things.

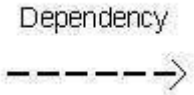
Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note -** It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



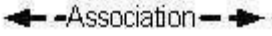
1.2 Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application. There are four kinds of relationships available.

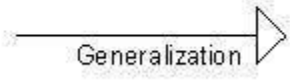
1. **Dependency** is a relationship between two things in which change in one element also affects the other.



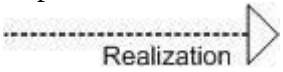
2. **Association** is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



3. **Generalization** can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



4. **Realization** can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



1.3 UML Diagrams

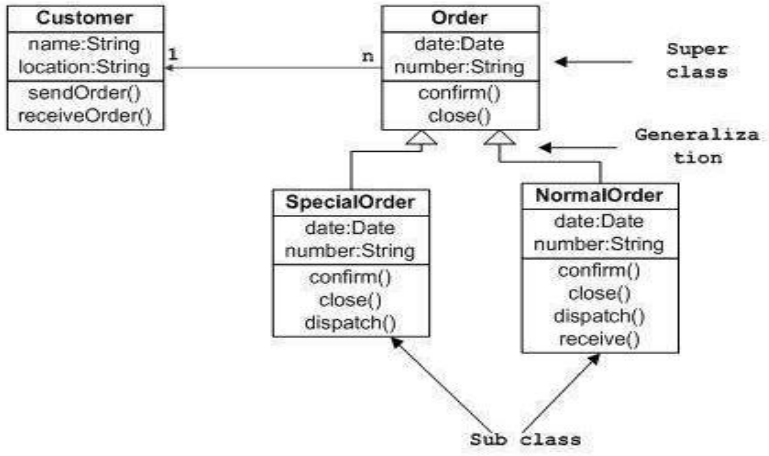
UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system. The visual effect of the UML diagram is the most important part of the entire process. There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams
- Behavioural Diagrams

Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable. These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are: Class diagram, Object diagram, Component diagram, Deployment diagram

1. Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature. Active class is used in a class diagram to represent the concurrency of the system. Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.



Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

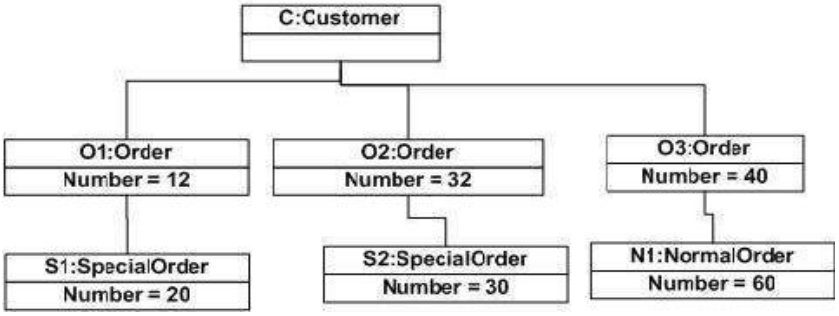
Where to Use Class Diagrams?

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system. Class diagrams are used for −

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

2. Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system. Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.

The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.



Purpose of Object Diagrams

The purposes of object diagrams are similar to class diagrams. The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature. It means the object diagram is closer to the actual system behaviour. The purpose is to capture the static view of a system at a particular moment.

The purpose of the object diagram can be summarized as −

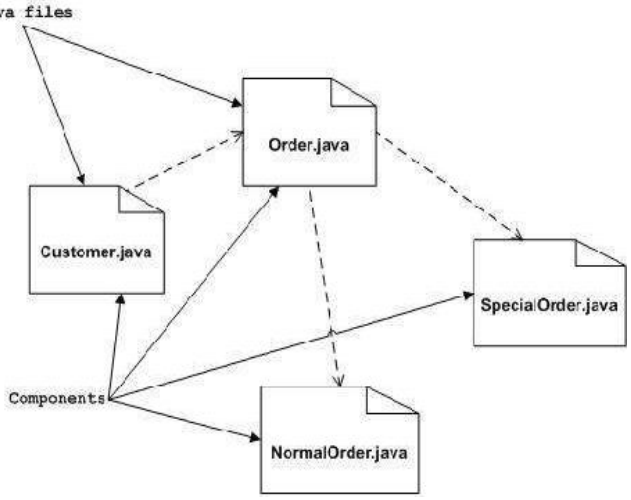
- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective

Where to Use Object Diagrams?

Object diagrams can be imagined as the snapshot of a running system at a particular moment. It can be said that object diagrams are used for −

- Making the prototype of a system.
- Reverse engineering.
- Modelling complex data structures.
- Understanding the system from practical perspective.

3. Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system. During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components. Finally, it can be said component diagrams are used to visualize the implementation.



Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc. A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.
The purpose of the component diagram can be summarized as −

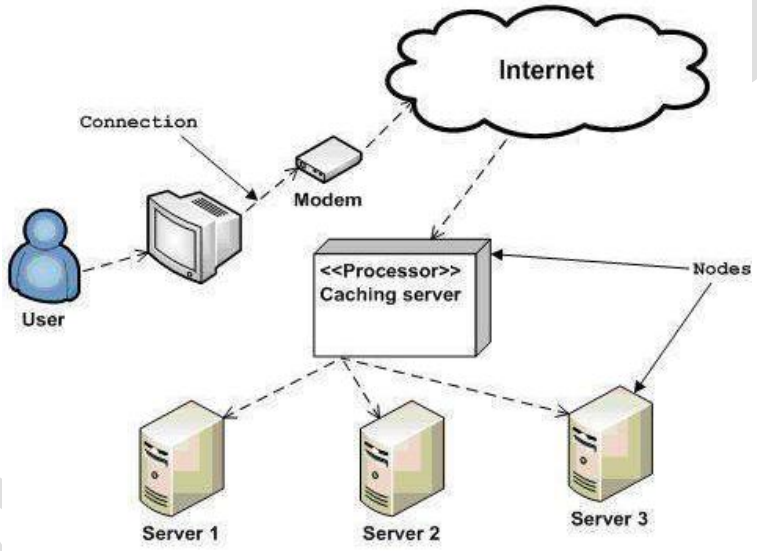
- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

Where to Use Component Diagrams?

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details
Component diagrams can be used to −

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.

4. Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed. Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.



Purpose of Deployment Diagrams

Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related. Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware. UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components. Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.
The purpose of deployment diagrams can be described as −

- Visualize the hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe the runtime processing nodes.

Where to Use Deployment Diagrams?

Now-a-days software applications are very complex in nature. Software applications can be standalone, web-based, distributed, main frame-based and many more. Hence, it is very important to design the hardware components efficiently.

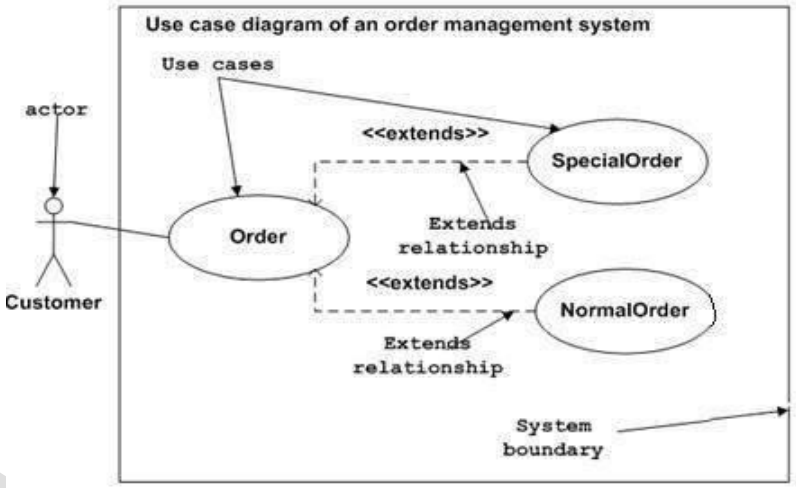
Deployment diagrams can be used −

- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

Behavioural Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered. Behavioural diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system. UML has the following five types of behavioural diagrams: Use case diagram, Sequence diagram, Collaboration diagram, State-chart diagram, Activity diagram.

5. Use case Diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system. A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors**.



Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified. When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Shows the interaction among the requirements is actors.

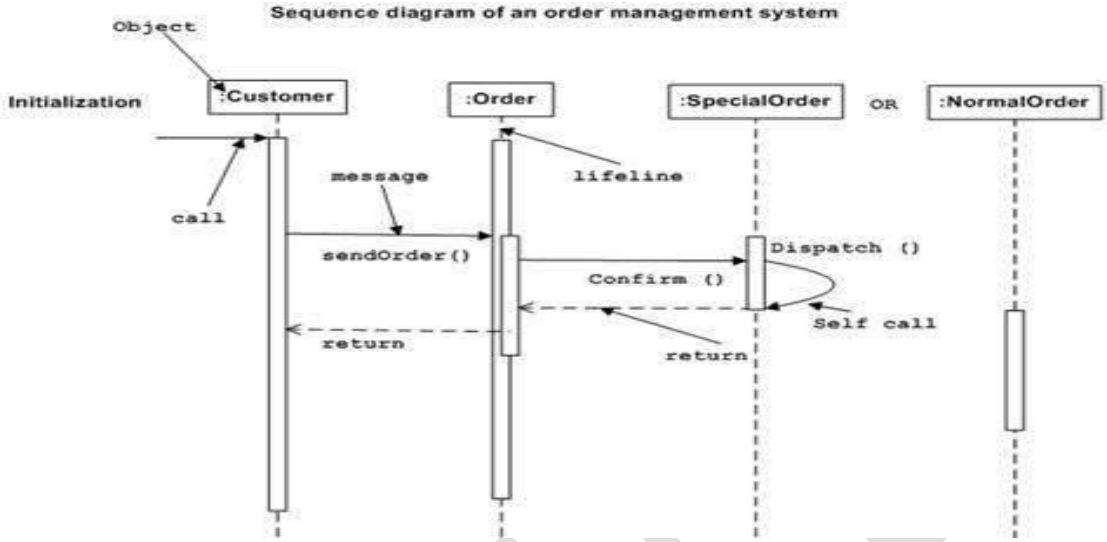
Where to Use a Use Case Diagram?

In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

Use case diagrams can be used for −

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

6. Sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another. Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.



Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behaviour of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction. Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

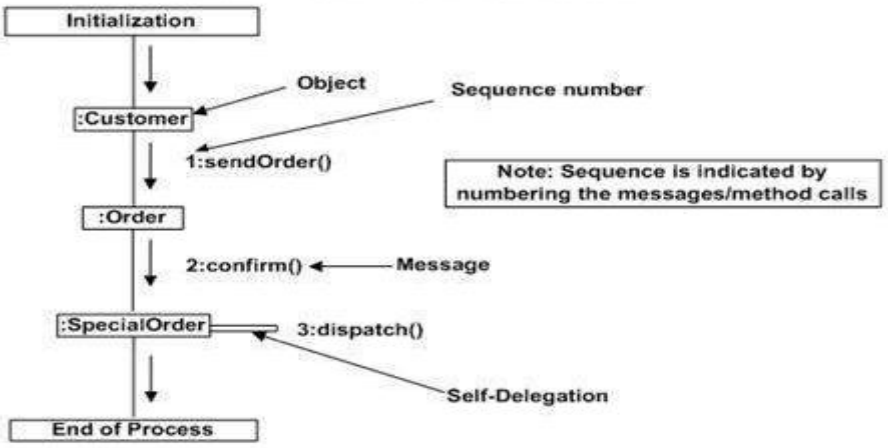
Where to Use Interaction Diagrams?

Interaction diagrams are used when we want to understand the message flow and the structural organization. Message flow means the sequence of control flow from one object to another. Structural organization means the visual organization of the elements in a system.

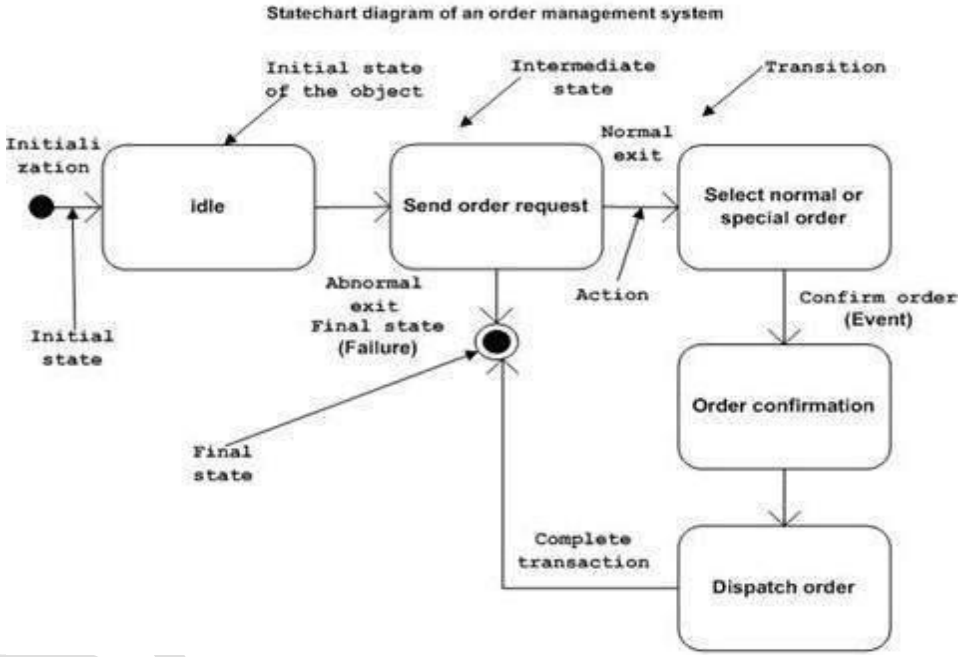
Interaction diagrams can be used −

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

7. Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links. The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization. To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



8. State-chart Diagram Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system. State-chart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc. State chart diagram is used to visualize the reaction of a system by internal/external factors.



Purpose of State-chart Diagrams

They define different states of an object during its lifetime and these states are changed by events. State-chart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events. State-chart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of State-chart diagram is to model lifetime of an object from creation to termination. State-chart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using State-chart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

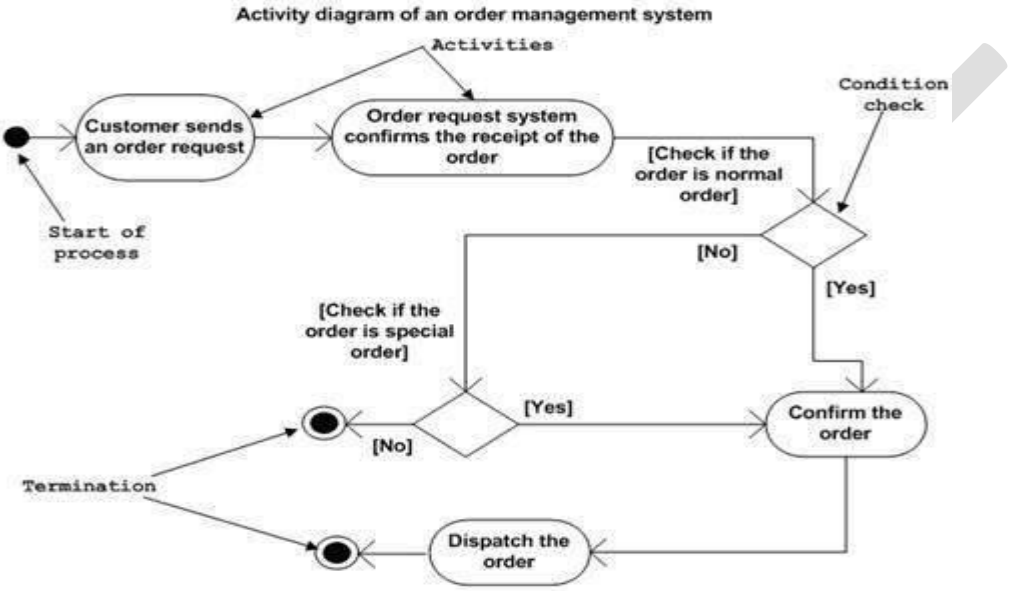
Where to Use State-chart Diagrams?

If we look into the practical implementation of State-chart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behaviour during its execution.

The main usage can be described as −

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

9. Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched. Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system. Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.



Purpose of Activity Diagrams

It captures the dynamic behaviour of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another. Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part. It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

Where to Use Activity Diagrams?

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

- Modelling work flow by using activities.
- Modelling business requirements.

- High level understanding of the system's functionalities.
- Investigating business requirements at a later stage.

Mapping disciplines to UML artifacts:

An **artifact** is a classifier that represents some **physical entity**, a piece of information that is used or is produced by a software development process, or by deployment and operation of a system. Artifact is a source of a deployment to a node. A particular instance (or "copy") of an artifact is deployed to a node instance. Some real life examples of UML artifacts are: text document, source file, script, binary executable file, archive file, database table

The **UML Standard Profile** defines several **standard stereotypes** that apply to artifacts:

«file»	A physical file in the context of the system developed.

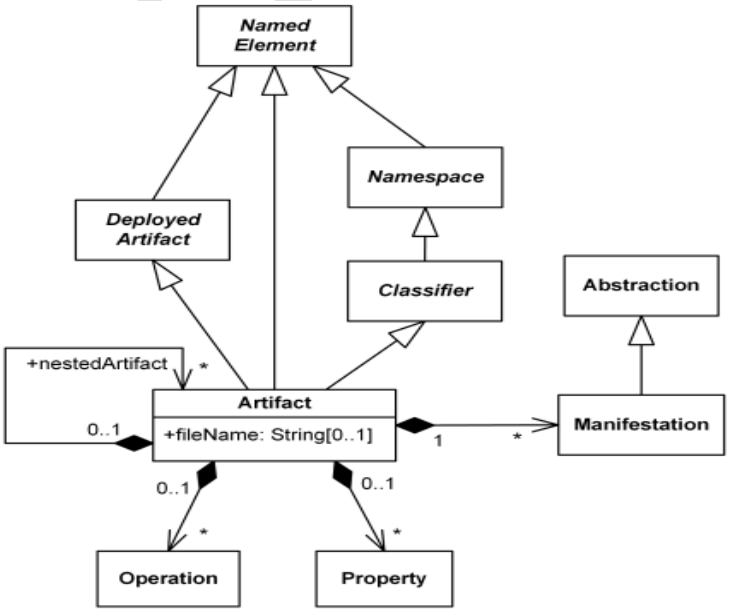
Standard stereotypes - subclasses of **«file»**:

«document»	A generic file that is not a «source» file or «executable».
«source»	A source file that can be compiled into an executable file.
«library»	A static or dynamic library file.
«executable»	A program file that can be executed on a computer system.
«script»	A script file that can be interpreted by a computer system.

Standard UML 1.x stereotype that is now obsolete:

«table»	Table in database.

Standard stereotypes can be further specialized into implementation and platform specific stereotypes in profiles. For example, Java profile might define **«jar»** as a subclass of «executable» for executable Java archives. Specific profiles are expected to provide some stereotype for artifact representing sets of files.



UML specifications define **artifact** as a subclass of abstract **deployed artifact**. Deployed artifact is described as an **artifact** or artifact instance that has been deployed to a deployment target. Common sense says that the relationship has to be reversed - deployed artifact should be a subclass of artifact, while it has to be allowed for some artifacts not to be deployed at all.

UML abstract syntax of Artifact:

Artifacts may have **properties** that represent features of the artifact, and operations that can be performed on its instances. Artifacts have **fileName** attribute - a concrete name that is used to refer to the artifact in a physical context - e.g. **file name** or **URI**. Artifact could have nested artifacts. Artifacts are deployed to a deployment target. **Instance specification** was extended in UML to allow instances of **artifacts** to be **deployed artifacts** in a deployment relationship.

An artifact is presented using an ordinary class rectangle with the keyword **«artifact»**. Examples in **UML** specification also show **document icon** in upper right corner.

Artifact web-app.war

C# source file artifact UserServices.cs

Library commons.dll

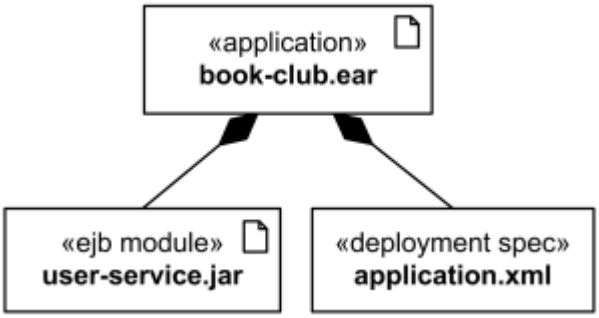
web-tools-lib.jar

Artifact web-tools-lib.jar

Alternatively, artifact may be depicted by an icon. Optionally, the underlining of the name of an artifact instance may be omitted, as the context is assumed to be known to users.

Associations between Artifacts

Artifacts can be involved in associations to other artifacts, e.g. composition associations. For instance, a deployment descriptor artifact for a component may be contained within the artifact that manifests that component. In that way, the component and its descriptor are deployed to a node instance as one artifact instance.

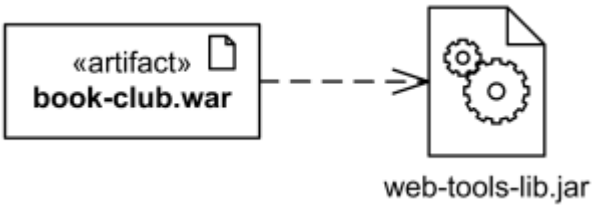


Application book-club.ear artifact contains EJB user-service.jar artifact and deployment descriptor.

Dependency between Artifacts

Artifacts can be involved in dependency relationship with other artifacts.

Dependency between artifacts is notated in the same way as general dependency, i.e. as a general dashed line with an open arrow head directed from client artifact to supplier artifact.



The book-club.war artifact depends on web-tools-lib.jar artifact.

Design Patterns:

A pattern describes problem which occurs over and over again in our environment, and then describes the core of solution to that problem in such way that we can use this solution a million times over, without ever doing it the same way twice is known as Design Pattern. Design patterns are based on the principles of the object-oriented programming paradigm and thus re-enforce the concepts of abstraction, inheritance, polymorphism and association.

In software design there are certain problems that recur in more than one application domain. Design patterns are solutions to these problems that can be used in any application domain.

For example, the general solution to the problem of variability of interfaces is to separate the core functionality of the application from the interface. This is referred to as the modeler view controller pattern (MVC). Furthermore, there does not exist a standardization for indexing patterns and general practices/processes for using design patterns during the design process have not as yet been established. In general, patterns have four essential elements:

1. **Pattern Name**: it is a handle used to describe the design problem, Increases vocabulary, Eases design discussions, Evaluation without implementation details
2. **Problem**: It describes when to apply a pattern, May include conditions for the pattern to be applicable, Symptoms of an inflexible design or limitation
3. **Solution**: It describes elements for the design, Includes relationships, responsibilities, and collaborations, Does not describe concrete designs or implementations, A pattern is more of a template
4. **Consequences**: Results and Trade Offs, Critical for design pattern evaluation, Often space and time trade offs, Language strengths and limitations

Describing Design Pattern

A pattern is defined in terms of classes and objects and relationships between them. Patterns can be viewed as modules or building blocks for more complex designs. Using existing well tested patterns saves time instead of deriving them from scratch each time. Patterns usually consist of smaller patterns/sub-patterns. Class diagrams are used to express design patterns. A more detailed pattern description template is sometimes used to define patterns. Such a template specifies the following:

- **Name and Classification:** Must have a meaningful name which conveys the essence of the pattern succinctly. Classification categorizes it into either creational, structural, or behavioral design patterns.
- **Intent:** A statement of the problem which answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue problem does it address?
- **Also Known As:** Other well-known names for the pattern, if any.
- **Motivation:** A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.
- **Applicability:** What are the situations in which the design pattern can be applied? What are examples of poor design that the pattern can address? How can you recognize these situations?
- **Structure:** A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). (UML, based on OMT, is now the standard notation used.)
- **Participants:** The classes and/or objects participating in the design pattern and their responsibilities.
- **Collaborations:** How the participants collaborate to carry out their responsibilities.
- **Consequences:** How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?
- **Implementation:** What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- **Sample Code and Usage:** Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk (or Java).
- **Known Uses:** Examples of the pattern found in real systems. We include at least two examples from different domains.
- **Related Patterns:** What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Types of Patterns

Patterns are classified by purpose and scope. The purpose is defined as:

- **Creational patterns**: deal with the creation of objects and help to make a system independent of how objects are created, composed and represented. They also enable flexibility in what gets created, who creates it, how it gets created and when it gets created. Focus on the best way to create instances of objects to promote flexibility. Programs should not depend on how objects are created or arranged. Patterns in this category include the factory method, the abstract factory method, the builder pattern, the prototype pattern and the singleton pattern.
- **Structural patterns**: deal with how objects are arranged to form larger structures. Focus on the composition of classes and objects into larger structures. The adapter, composite, flyweight, facade, bridge, proxy and decorator patterns fall into this category.
- **Behavioral patterns**: deal with how objects interact, the ownership of responsibility and factoring code in variant and non-variant components. Focus on the interaction between classes or objects. These patterns include the observer, mediator, chain of responsibility, template, interpreter, strategy, visitor, state, command and iterator patterns.

The scope of a pattern specifies whether the pattern applies to classes or objects. Class patterns describe relationships between classes and their subclasses. These relationships are static. Object patterns describe the relationships between objects. Theses relationships can be changed at runtime. The scope is defined as:

- **Class scope**: static relationships through class inheritance (white-box reuse)
- **Object scope**: dynamic relationships through object composition (black-box reuse) or collaboration

Pattern summary

There are 5 creational patterns, 7 structural patterns and 11 behavioral patterns:

		Purpose		
		Creational	***Structural***	***Behavioural***
Scope	***Class***	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Goals of good design:

Design faces many challenges to produce good product. Good design leads to software that has following goals:

- Codify good design
- Capture & preserve design information to improve documentation
- Give design structures explicit name
- Facilitate re-structuring/ refactoring: it cuts production cost of cod
- Correctness of design: does what it should
- Robustness of design: tolerant of misuse like faulty rate
- Flexibility of design: adaptable to shifting requirements
- Efficiency of design: good use of processor and memory

Case Study & MVC architecture:

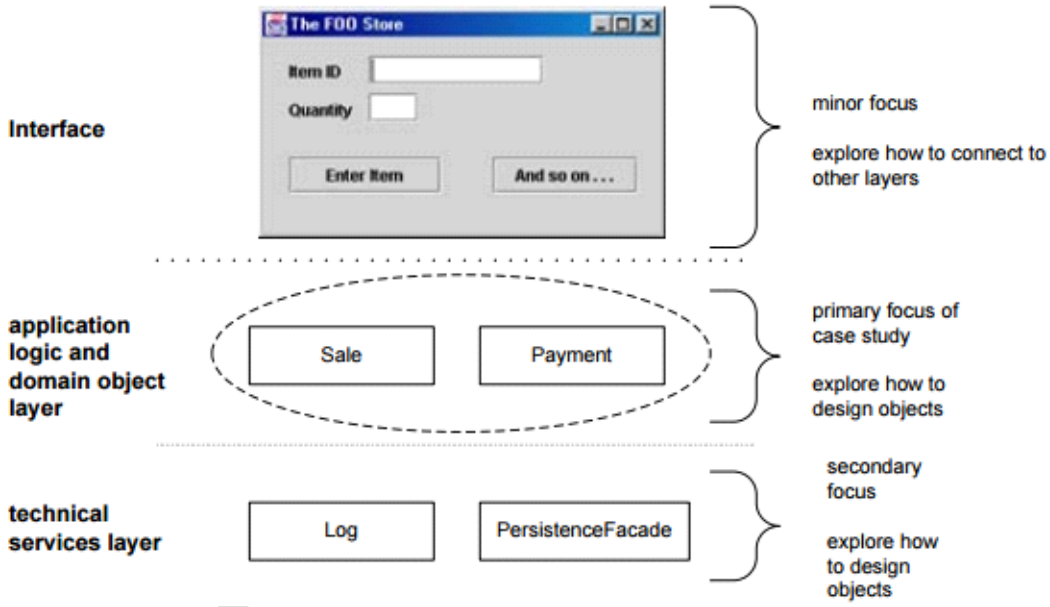
The case study is the NextGen point-of-sale (POS) system. In this type of apparently straightforward problem domain, we can see that interesting requirement and design problems are there to solve. Additionally, it is a realistic problem; organizations really do write POS systems using the object technologies.

POS system is one which is a computerized application that is used to record sales and handle the payments; it is particularly used in retail stores. This includes hardware components like a computer and bar code scanner, and software that is to run the system. It interfaces to several service applications, like a third-party tax calculator and an inventory control. These systems should be relatively fault-tolerant; This means that, even if remote services are temporarily unavailable , they should still be capable of capturing the sales and be capable of handling at least cash payments.

A POS system should increasingly support the multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, touch screen input, a regular personal computer with something like a Java Swing graphical user interface, wireless PDAs, and so on. Moreover, we are creating a commercial POS system that we may sell to different clients with desperate needs in terms of business rule processing. Each client will required or desire a unique set of logic in order to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism in order to provide this flexibility and customization.

By using an iterative development strategy, we are now proceeding through requirements, object-oriented analysis, design, and finally implementation.

Architectural Layers



Sample layers and objects in an object-oriented system, and case study focus

A typical object-oriented information system is designed in terms of different architectural layers or as shown in the above shown figure. The following is not a complete list, yet it shows an example:

- **User Interface:** graphical interface; windows.
- **Technical Services:** It is a general purpose objects and subsystems which the provide supporting technical services, like interfacing with database or an error logging. These services are generally application-independent and can be reused across several systems.
- **Application Logic and Domain Objects:** The software objects that represents the domain concepts

OOAD is most relevant for modeling application logic and technical service layers. NextGen case study primarily highlights the problem of domain objects, and also in allocating

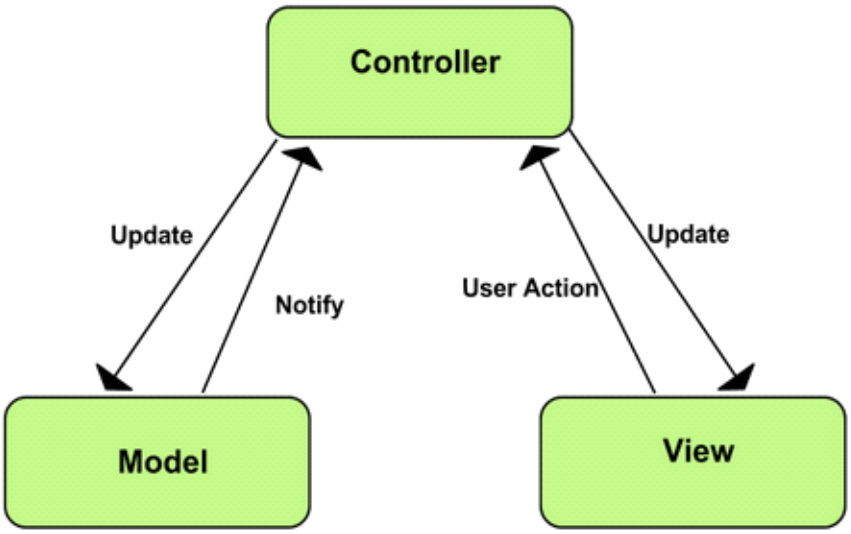
responsibilities to them to fulfill the requirements of application. Object-oriented design is applied in order to create a technical service subsystem for interfacing this with a database.
In this design approach, the User Interface layer has very little responsibility i.e. it is considered to be thin. Windows do not contain code which performs the application logic or processing. Rather, task requests are forwarded on to some other layers.

MVC architecture

MVC or Model View Controller as it is popularly known, is a software design pattern for web applications development. A Model View Controller pattern is made up of the following three parts. They are,

- **Model:** The lowest level of the pattern which is responsible for maintaining the data.
- **View:** This level is responsible for displaying all or a portion of the data to the user.
- **Controller:** Software Code which controls the interactions between the Model and the View.

MVC is popular as it isolates the application logic from the user interface layer and then supports the separation of concerns. The Controller receives all the requests for the application and then works with the Model in order to prepare any data that is needed by View. The View then uses this data which is prepared by the Controller in order to generate a final presentable response. The MVC abstraction is graphically represented as follows.



1. The model:
The model in the MVC layer is responsible for managing data of the application. It responds to the request from view and it also responds to instructions from controller to update itself.

2. The view
A presentation of data in a particular format, triggered by decision of the controller to present the data. They are the script based tinplating systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

3. The controller
The controller is responsible for responding to the user input and performs interactions on the data model objects. The controller receives the input, and it validates the input and then performs the business operation that modifies the state of the data model.

***