The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system's blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system.

Brief History

It was developed in 1990s as an amalgamation of several techniques, prominently OOAD technique by Grady Booch, OMT (Object Modeling Technique) by James Rumbaugh, and OOSE (Object Oriented Software Engineering) by Ivar Jacobson. UML attempted to standardize semantic models, syntactic notations, and diagrams of OOAD.

Systems and Models in UML

System − A set of elements organized to achieve certain objectives form a system. Systems are often divided into subsystems and described by a set of models.

Model − Model is a simplified, complete, and consistent abstraction of a system, created for better understanding of the system.

View − A view is a projection of a system's model from a specific perspective.

Conceptual Model of UML

The Conceptual Model of UML encompasses three major elements −

- Basic building blocks
- Rules
- Common mechanisms

Basic Building Blocks

The three building blocks of UML are −

- Things
- Relationships
- Diagrams

Things

There are four kinds of things in UML, namely −

- **Structural Things** − These are the nouns of the UML models representing the static elements that may be either physical or conceptual. The structural things are class, interface, collaboration, use case, active class, components, and nodes.
- **Behavioral Things** − These are the verbs of the UML models representing the dynamic behavior over time and space. The two types of behavioral things are interaction and state machine.
- **Grouping Things** − They comprise the organizational parts of the UML models. There is only one kind of grouping thing, i.e., package.
- **Annotational Things** − These are the explanations in the UML models representing the comments applied to describe elements.

Relationships

Relationships are the connection between things. The four types of relationships that can be represented in UML are −

- **Dependency** − This is a semantic relationship between two things such that a change in one thing brings a change in the other. The former is the independent thing, while the latter is the dependent thing.

- **Association** − This is a structural relationship that represents a group of links having common structure and common behavior.
- **Generalization** − This represents a generalization/specialization relationship in which subclasses inherit structure and behavior from super-classes.
- **Realization** − This is a semantic relationship between two or more classifiers such that one classifier lays down a contract that the other classifiers ensure to abide by.

Diagrams

A diagram is a graphical representation of a system. It comprises of a group of elements generally in the form of a graph. UML includes nine diagrams in all, namely −

- Class Diagram
- Object Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- State Chart Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

Rules

UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic rules for the following −

- Names
- Scope
- Visibility
- Integrity
- Execution

Common Mechanisms

UML has four common mechanisms −

- Specifications
- Adornments
- Common Divisions
- Extensibility Mechanisms

Specifications

In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. These are the specifications. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

Adornments

Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

Common Divisions

Object-oriented systems can be divided in many ways. The two common ways of division are −

- **Division of classes and objects** − A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
- **Division of Interface and Implementation** − An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

Extensibility Mechanisms

UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system. The extensibility mechanisms are −

- **Stereotypes** − It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones.
- **Tagged Values** − It extends the properties of UML building blocks.
- **Constraints** − It extends the semantics of UML building blocks.

Class

A class is represented by a rectangle having three sections −

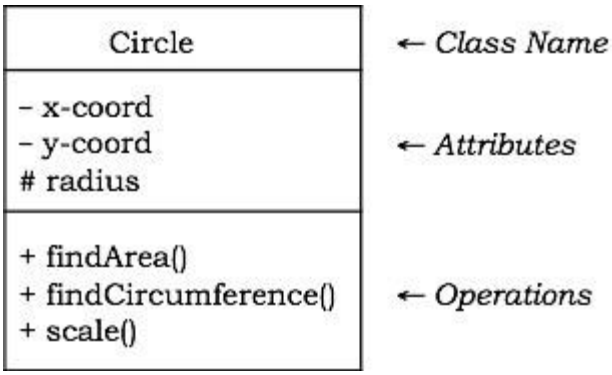
- the top section containing the name of the class
- the middle section containing class attributes
- the bottom section representing operations of the class

The visibility of the attributes and operations can be represented in the following ways −

- **Public** − A public member is visible from anywhere in the system. In class diagram, it is prefixed by the symbol '+'.
- **Private** − A private member is visible only from within the class. It cannot be accessed from outside the class. A private member is prefixed by the symbol '–'.
- **Protected** − A protected member is visible from within the class and from the subclasses inherited from this class, but not from outside. It is prefixed by the symbol '#'.

An abstract class has the class name written in italics.

Example − Let us consider the Circle class introduced earlier. The attributes of Circle are x-coord, y-coord, and radius. The operations are findArea(), findCircumference(), and scale(). Let us assume that x-coord and y-coord are private data members, radius is a protected data member, and the member functions are public. The following figure gives the diagrammatic representation of the class.



Object

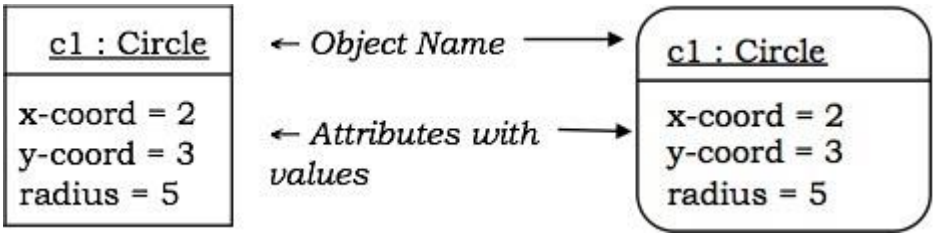
An object is represented as a rectangle with two sections −

- The top section contains the name of the object with the name of the class or package of which it is an instance of. The name takes the following forms −

- **object-name** − class-name
- **object-name** − class-name :: package-name
- **class-name** − in case of anonymous objects

- The bottom section represents the values of the attributes. It takes the form attribute-name = value.
- Sometimes objects are represented using rounded rectangles.

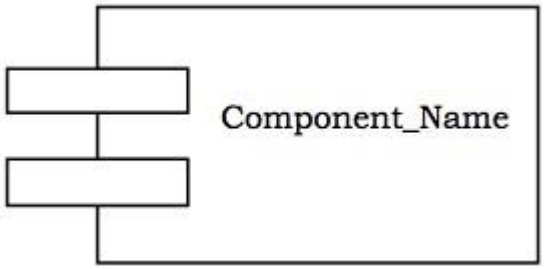
Example − Let us consider an object of the class Circle named c1. We assume that the center of c1 is at (2, 3) and the radius of c1 is 5. The following figure depicts the object.



Component

A component is a physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces. It represents the physical packaging of elements like classes and interfaces.

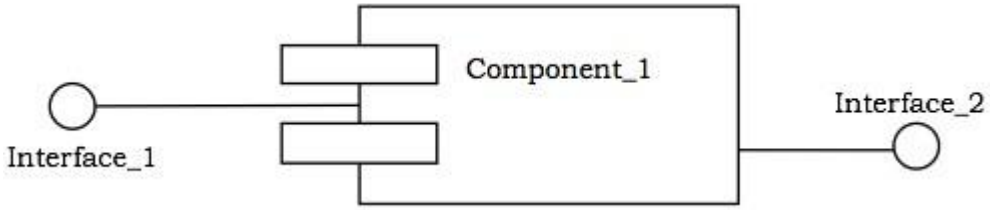
Notation − In UML diagrams, a component is represented by a rectangle with tabs as shown in the figure below.



Interface

Interface is a collection of methods of a class or component. It specifies the set of services that may be provided by the class or component.

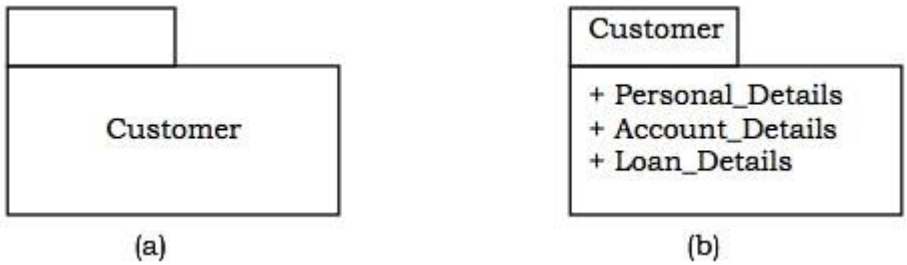
Notation − Generally, an interface is drawn as a circle together with its name. An interface is almost always attached to the class or component that realizes it. The following figure gives the notation of an interface.



Package

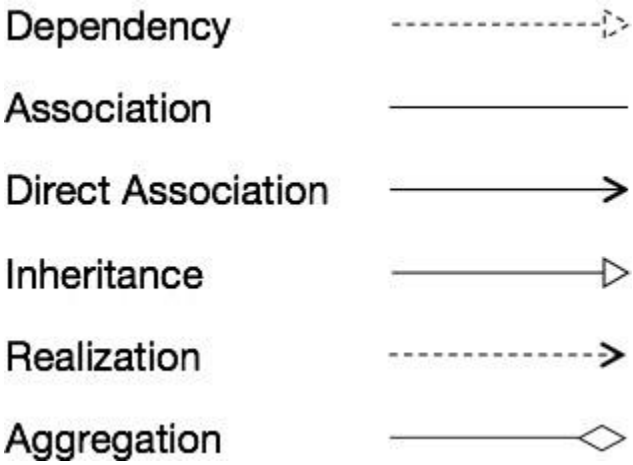
A package is an organized group of elements. A package may contain structural things like classes, components, and other packages in it.

Notation − Graphically, a package is represented by a tabbed folder. A package is generally drawn with only its name. However it may have additional details about the contents of the package. See the following figures.



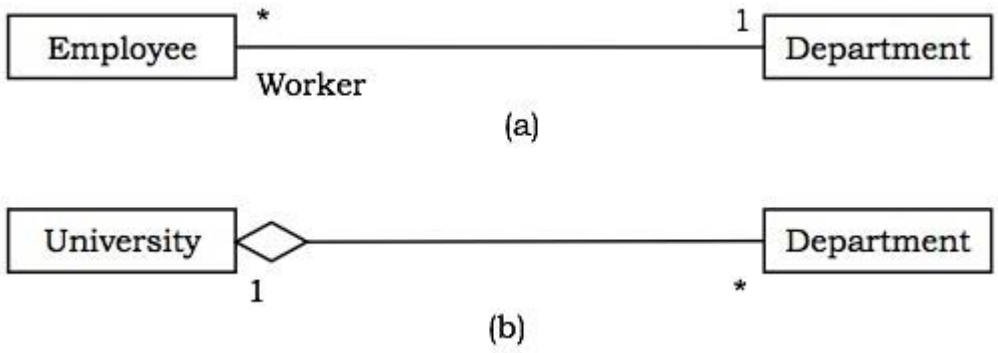
Relationship

The notations for the different types of relationships are as follows −



Usually, elements in a relationship play specific roles in the relationship. A role name signifies the behavior of an element participating in a certain context.

Example − The following figures show examples of different relationships between classes. The first figure shows an association between two classes, Department and Employee, wherein a department may have a number of employees working in it. Worker is the role name. The '1' alongside Department and '*' alongside Employee depict that the cardinality ratio is one–to–many. The second figure portrays the aggregation relationship, a University is the "whole–of" many Departments.



UML structural diagrams are categorized as follows: class diagram, object diagram, component diagram, and deployment diagram.

Class Diagram

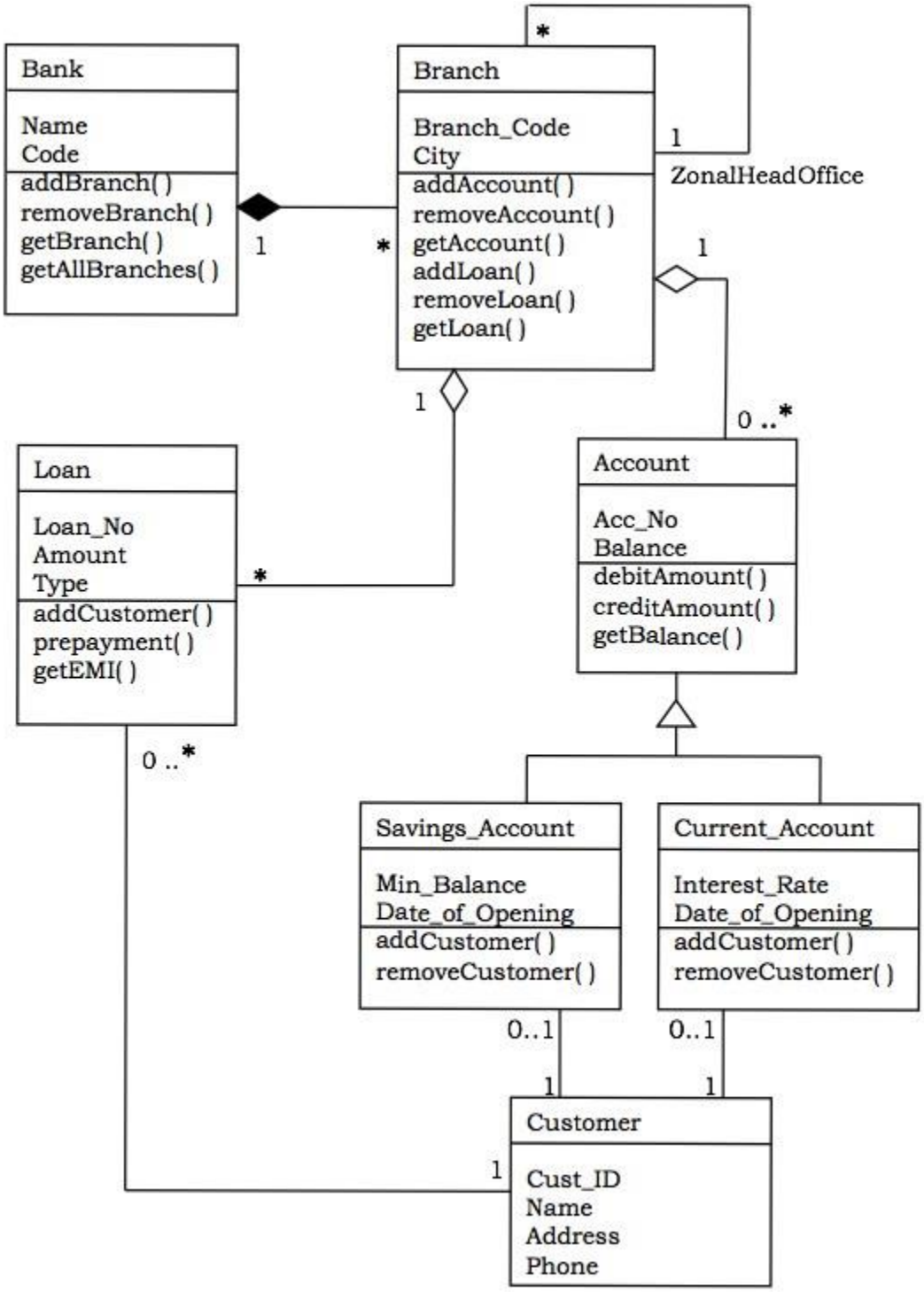
A class diagram models the static view of a system. It comprises of the classes, interfaces, and collaborations of a system; and the relationships between them.

Class Diagram of a System

Let us consider a simplified Banking System.

A bank has many branches. In each zone, one branch is designated as the zonal head office that supervises the other branches in that zone. Each branch can have multiple accounts and loans. An account may be either a savings account or a current account. A customer may open both a savings account and a current account. However, a customer must not have more than one savings account or current account. A customer may also procure loans from the bank.

The following figure shows the corresponding class diagram.



Classes in the system

Bank, Branch, Account, Savings Account, Current Account, Loan, and Customer.

Relationships

- **A Bank "has–a" number of Branches** − composition, one–to–many
- **A Branch with role Zonal Head Office supervises other Branches** − unary association, one–to-many
- **A Branch "has–a" number of accounts** − aggregation, one–to–many

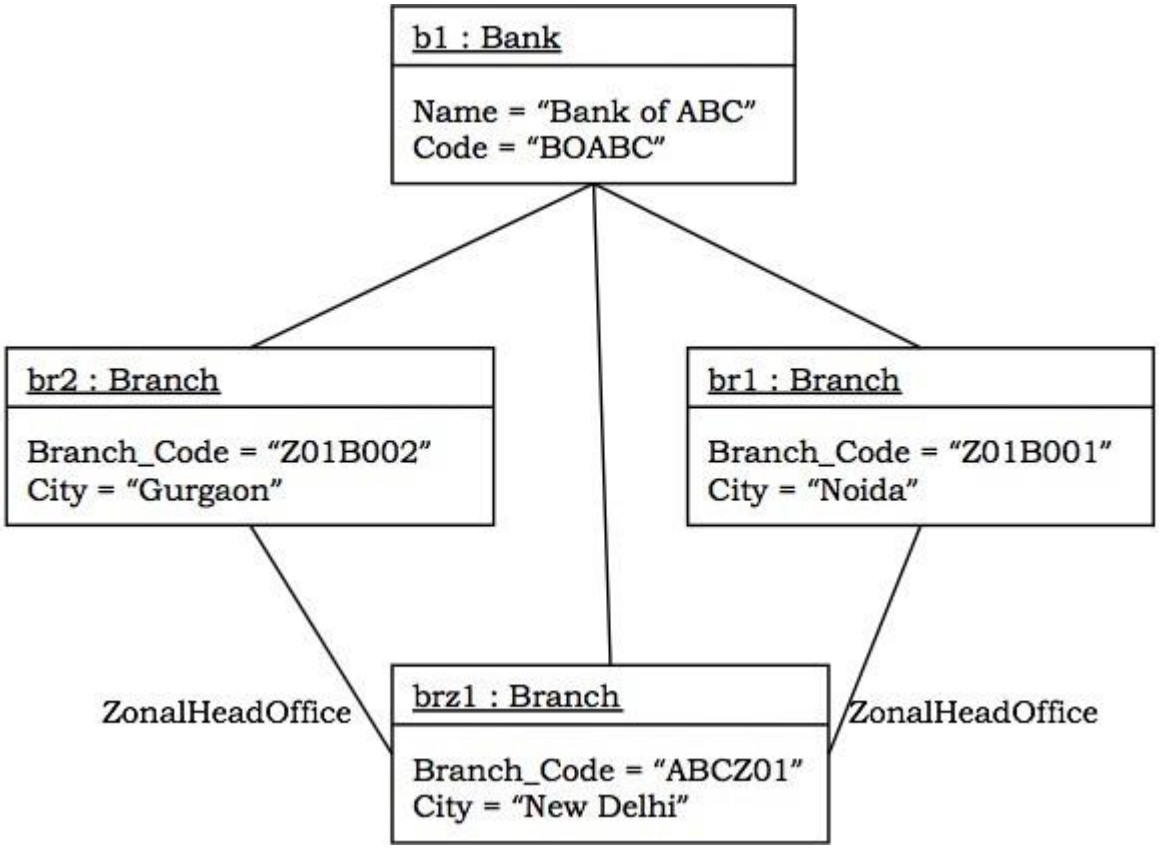
From the class Account, two classes have inherited, namely, Savings Account and Current Account.

- **A Customer can have one Current Account** − association, one–to–one
- **A Customer can have one Savings Account** − association, one–to–one
- **A Branch "has–a" number of Loans** − aggregation, one–to–many
- **A Customer can take many loans** − association, one–to–many

Object Diagram

An object diagram models a group of objects and their links at a point of time. It shows the instances of the things in a class diagram. Object diagram is the static part of an interaction diagram.

Example − The following figure shows an object diagram of a portion of the class diagram of the Banking System.



Component Diagram

Component diagrams show the organization and dependencies among a group of components.

Component diagrams comprise of −

- Components
- Interfaces
- Relationships

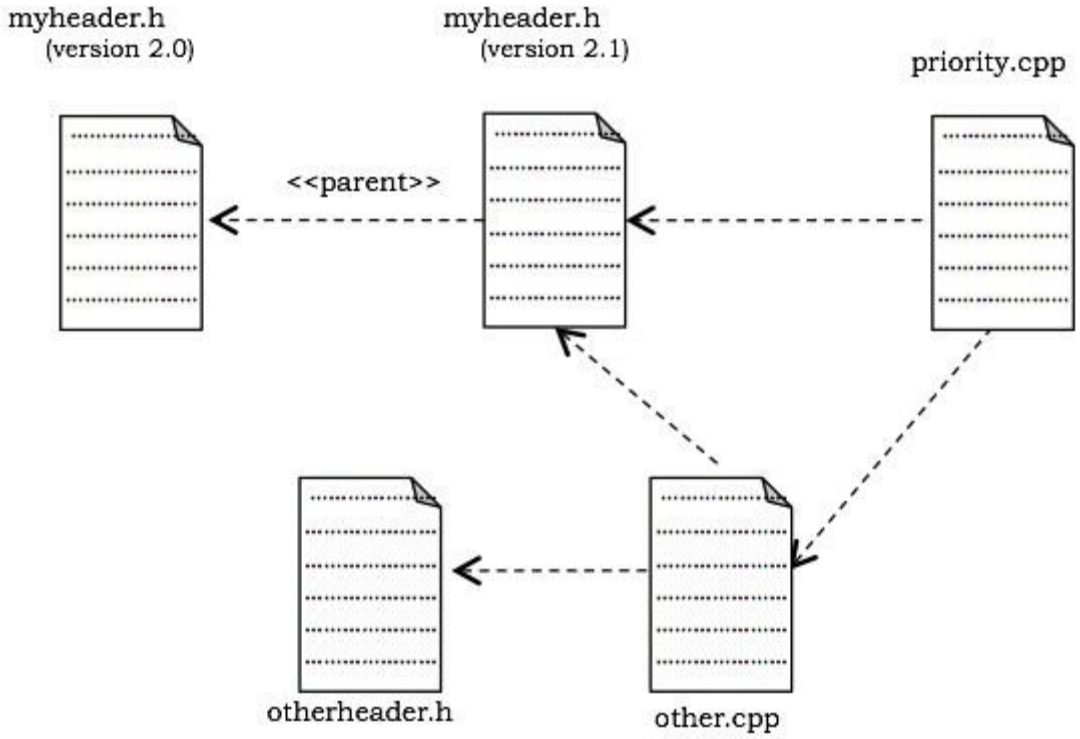
- Packages and Subsystems (optional)

Component diagrams are used for −

- constructing systems through forward and reverse engineering.
- modeling configuration management of source code files while developing a system using an object-oriented programming language.
- representing schemas in modeling databases.
- modeling behaviors of dynamic systems.

Example

The following figure shows a component diagram to model a system's source code that is developed using C++. It shows four source code files, namely, myheader.h, otherheader.h, priority.cpp, and other.cpp. Two versions of myheader.h are shown, tracing from the recent version to its ancestor. The file priority.cpp has compilation dependency on other.cpp. The file other.cpp has compilation dependency on otherheader.h.



Deployment Diagram

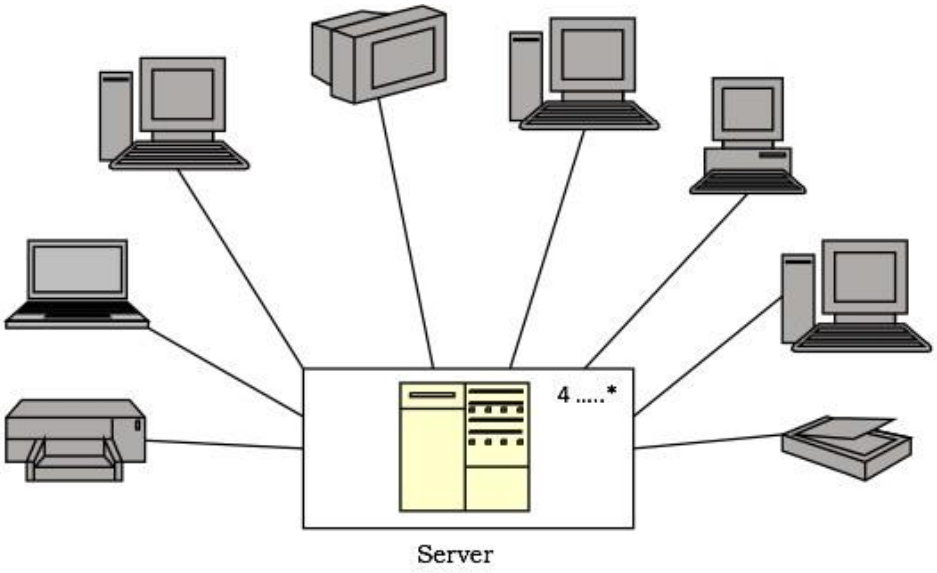
A deployment diagram puts emphasis on the configuration of runtime processing nodes and their components that live on them. They are commonly comprised of nodes and dependencies, or associations between the nodes.

Deployment diagrams are used to −

- model devices in embedded systems that typically comprise of software-intensive collection of hardware.
- represent the topologies of client/server systems.
- model fully distributed systems.

Example

The following figure shows the topology of a computer system that follows client/server architecture. The figure illustrates a node stereotyped as server that comprises of processors. The figure indicates that four or more servers are deployed at the system. Connected to the server are the client nodes, where each node represents a terminal device such as workstation, laptop, scanner, or printer. The nodes are represented using icons that clearly depict the real-world equivalent.



UML behavioral diagrams visualize, specify, construct, and document the dynamic aspects of a system. The behavioral diagrams are categorized as follows: use case diagrams, interaction diagrams, state–chart diagrams, and activity diagrams.

Use Case Model

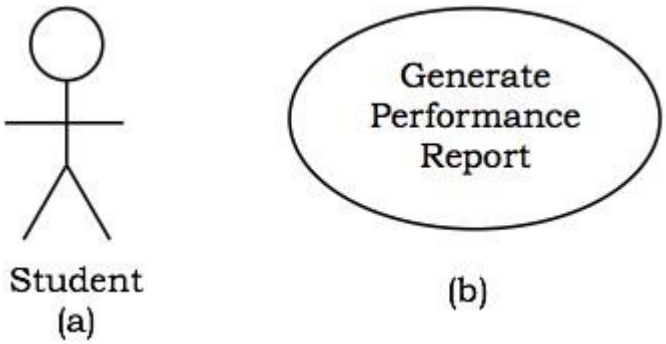
Use case

A use case describes the sequence of actions a system performs yielding visible results. It shows the interaction of things outside the system with the system itself. Use cases may be applied to the whole system as well as a part of the system.

Actor

An actor represents the roles that the users of the use cases play. An actor may be a person (e.g. student, customer), a device (e.g. workstation), or another system (e.g. bank, institution).

The following figure shows the notations of an actor named Student and a use case called Generate Performance Report.



Use case diagrams

Use case diagrams present an outside view of the manner the elements in a system behave and how they can be used in the context.

Use case diagrams comprise of −

- Use cases
- Actors

- Relationships like dependency, generalization, and association

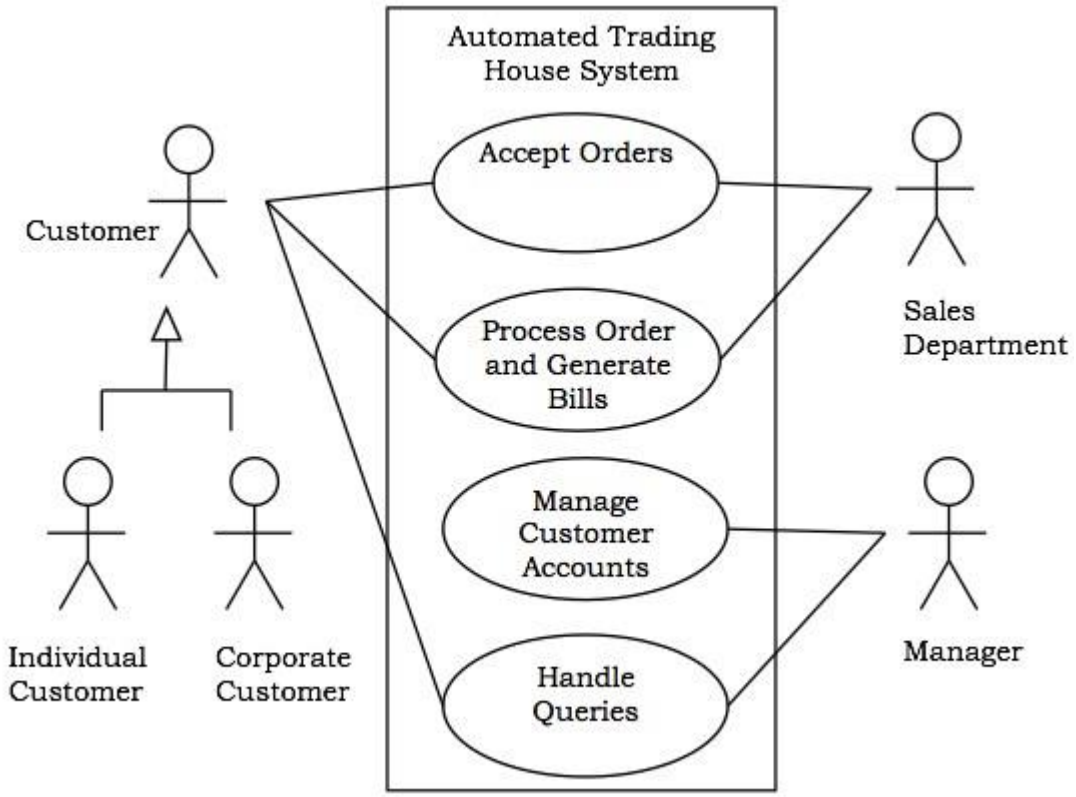
Use case diagrams are used −

- To model the context of a system by enclosing all the activities of a system within a rectangle and focusing on the actors outside the system by interacting with it.
- To model the requirements of a system from the outside point of view.

Example

Let us consider an Automated Trading House System. We assume the following features of the system −

- The trading house has transactions with two types of customers, individual customers and corporate customers.
- Once the customer places an order, it is processed by the sales department and the customer is given the bill.
- The system allows the manager to manage customer accounts and answer any queries posted by the customer.



Interaction Diagrams

Interaction diagrams depict interactions of objects and their relationships. They also include the messages passed between them. There are two types of interaction diagrams −

- Sequence Diagrams
- Collaboration Diagrams

Interaction diagrams are used for modeling −

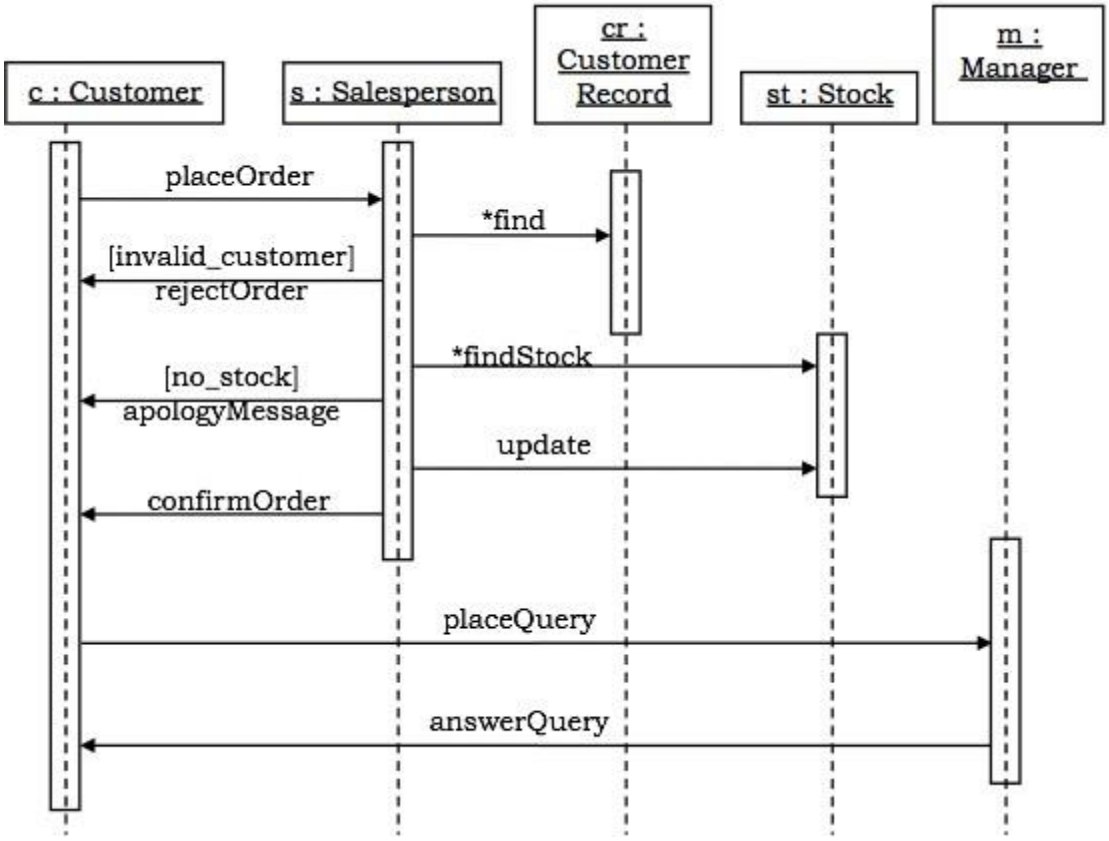
- the control flow by time ordering using sequence diagrams.
- the control flow of organization using collaboration diagrams.

Sequence Diagrams

Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.

Notations − These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x–axis. The messages that these objects send and receive are placed along the y–axis, in the order of increasing time from top to bottom.

Example − A sequence diagram for the Automated Trading House System is shown in the following figure.

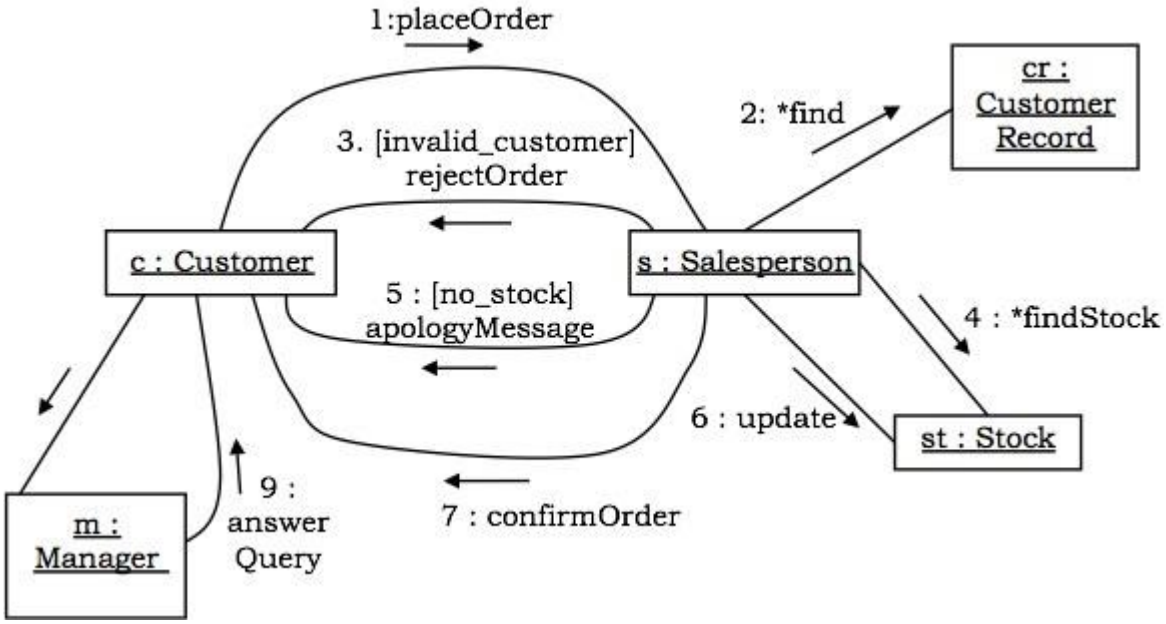


Collaboration Diagrams

Collaboration diagrams are interaction diagrams that illustrate the structure of the objects that send and receive messages.

Notations − In these diagrams, the objects that participate in the interaction are shown using vertices. The links that connect the objects are used to send and receive messages. The message is shown as a labeled arrow.

Example − Collaboration diagram for the Automated Trading House System is illustrated in the figure below.



State–Chart Diagrams

A state–chart diagram shows a state machine that depicts the control flow of an object from one state to another. A state machine portrays the sequences of states which an object undergoes due to events and their responses to events.

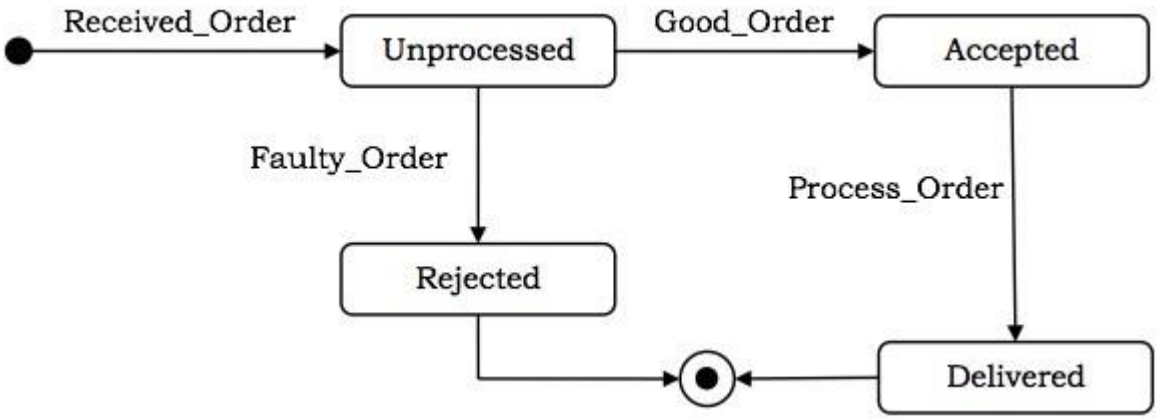
State–Chart Diagrams comprise of −

- States: Simple or Composite
- Transitions between states
- Events causing transitions
- Actions due to the events

State-chart diagrams are used for modeling objects which are reactive in nature.

Example

In the Automated Trading House System, let us model Order as an object and trace its sequence. The following figure shows the corresponding state–chart diagram.



Activity Diagrams

An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

Activity diagrams comprise of −

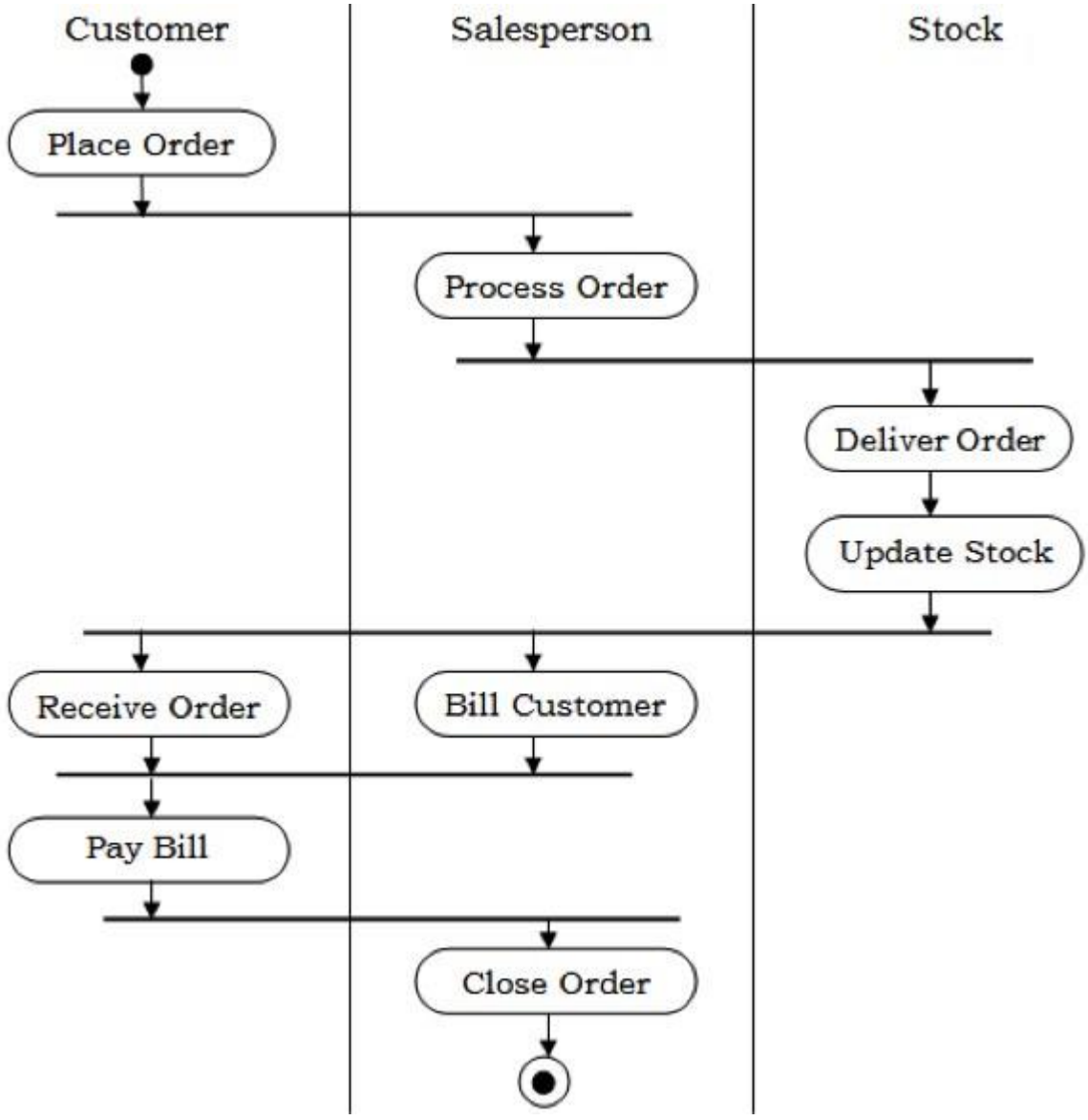
- Activity states and action states
- Transitions
- Objects

Activity diagrams are used for modeling −

- workflows as viewed by actors, interacting with the system.
- details of operations or computations using flowcharts.

Example

The following figure shows an activity diagram of a portion of the Automated Trading House System.



After the analysis phase, the conceptual model is developed further into an object-oriented model using object-oriented design (OOD). In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies

The stages for object–oriented design can be identified as −

- Definition of the context of the system
- Designing system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

System Design

Object-oriented system design involves defining the context of a system followed by designing the architecture of the system.

- **Context** − The context of a system has a static and a dynamic part. The static context of the system is designed using a simple block diagram of the whole system which is expanded into a hierarchy of subsystems. The subsystem model is represented by UML packages. The dynamic context describes how the system interacts with its environment. It is modelled using **use case diagrams**.
- **System Architecture** − The system architecture is designed on the basis of the context of the system in accordance with the principles of architectural design as well as domain knowledge. Typically, a system is partitioned into layers and each layer is decomposed to form the subsystems.

Object-Oriented Decomposition

Decomposition means dividing a large complex system into a hierarchy of smaller components with lesser complexities, on the principles of divide–and–conquer. Each major component of the system is called a subsystem. Object-oriented decomposition identifies individual autonomous objects in a system and the communication among these objects.

The advantages of decomposition are −

- The individual components are of lesser complexity, and so more understandable and manageable.
- It enables division of workforce having specialized skills.
- It allows subsystems to be replaced or modified without affecting other subsystems.

Identifying Concurrency

Concurrency allows more than one objects to receive events at the same time and more than one activity to be executed simultaneously. Concurrency is identified and represented in the dynamic model.

To enable concurrency, each concurrent element is assigned a separate thread of control. If the concurrency is at object level, then two concurrent objects are

assigned two different threads of control. If two operations of a single object are concurrent in nature, then that object is split among different threads.

Concurrency is associated with the problems of data integrity, deadlock, and starvation. So a clear strategy needs to be made whenever concurrency is required. Besides, concurrency requires to be identified at the design stage itself, and cannot be left for implementation stage.

Identifying Patterns

While designing applications, some commonly accepted solutions are adopted for some categories of problems. These are the patterns of design. A pattern can be defined as a documented set of building blocks that can be used in certain types of application development problems.

Some commonly used design patterns are −

- Façade pattern
- Model view separation pattern
- Observer pattern
- Model view controller pattern
- Publish subscribe pattern
- Proxy pattern

Controlling Events

During system design, the events that may occur in the objects of the system need to be identified and appropriately dealt with.

An event is a specification of a significant occurrence that has a location in time and space.

There are four types of events that can be modelled, namely −

- **Signal Event** − A named object thrown by one object and caught by another object.
- **Call Event** − A synchronous event representing dispatch of an operation.
- **Time Event** − An event representing passage of time.
- **Change Event** − An event representing change in state.

Handling Boundary Conditions

The system design phase needs to address the initialization and the termination of the system as a whole as well as each subsystem. The different aspects that are documented are as follows −

- The start–up of the system, i.e., the transition of the system from non-initialized state to steady state.

- The termination of the system, i.e., the closing of all running threads, cleaning up of resources, and the messages to be sent.
- The initial configuration of the system and the reconfiguration of the system when needed.
- Foreseeing failures or undesired termination of the system.

Boundary conditions are modelled using boundary use cases.

Object Design

After the hierarchy of subsystems has been developed, the objects in the system are identified and their details are designed. Here, the designer details out the strategy chosen during the system design. The emphasis shifts from application domain concepts toward computer concepts. The objects identified during analysis are etched out for implementation with an aim to minimize execution time, memory consumption, and overall cost.

Object design includes the following phases −

- Object identification
- Object representation, i.e., construction of design models
- Classification of operations
- Algorithm design
- Design of relationships
- Implementation of control for external interactions
- Package classes and associations into modules

Object Identification

The first step of object design is object identification. The objects identified in the object–oriented analysis phases are grouped into classes and refined so that they are suitable for actual implementation.

The functions of this stage are −

- Identifying and refining the classes in each subsystem or package
- Defining the links and associations between the classes
- Designing the hierarchical associations among the classes, i.e., the generalization/specialization and inheritances
- Designing aggregations

Object Representation

Once the classes are identified, they need to be represented using object modelling techniques. This stage essentially involves constructing UML diagrams.

There are two types of design models that need to be produced −

- **Static Models** − To describe the static structure of a system using class diagrams and object diagrams.
- **Dynamic Models** − To describe the dynamic structure of a system and show the interaction between classes using interaction diagrams and state–chart diagrams.

Classification of Operations

In this step, the operation to be performed on objects are defined by combining the three models developed in the OOA phase, namely, object model, dynamic model, and functional model. An operation specifies what is to be done and not how it should be done.

The following tasks are performed regarding operations −

- The state transition diagram of each object in the system is developed.
- Operations are defined for the events received by the objects.
- Cases in which one event triggers other events in same or different objects are identified.
- The sub–operations within the actions are identified.
- The main actions are expanded to data flow diagrams.

Algorithm Design

The operations in the objects are defined using algorithms. An algorithm is a stepwise procedure that solves the problem laid down in an operation. Algorithms focus on how it is to be done.

There may be more than one algorithm corresponding to a given operation. Once the alternative algorithms are identified, the optimal algorithm is selected for the given problem domain. The metrics for choosing the optimal algorithm are −

- **Computational Complexity** − Complexity determines the efficiency of an algorithm in terms of computation time and memory requirements.
- **Flexibility** − Flexibility determines whether the chosen algorithm can be implemented suitably, without loss of appropriateness in various environments.
- **Understandability** − This determines whether the chosen algorithm is easy to understand and implement.

Design of Relationships

The strategy to implement the relationships needs to be chalked out during the object design phase. The main relationships that are addressed comprise of associations, aggregations, and inheritances.

The designer should do the following regarding associations −

- Identify whether an association is unidirectional or bidirectional.

- Analyze the path of associations and update them if necessary.
- Implement the associations as a distinct object, in case of many–to-many relationships; or as a link to other object in case of one–to-one or one–to-many relationships.

Regarding inheritances, the designer should do the following −

- Adjust the classes and their associations.
- Identify abstract classes.
- Make provisions so that behaviors are shared when needed.

Implementation of Control

The object designer may incorporate refinements in the strategy of the state–chart model. In system design, a basic strategy for realizing the dynamic model is made. During object design, this strategy is aptly embellished for appropriate implementation.

The approaches for implementation of the dynamic model are −

- **Represent State as a Location within a Program** − This is the traditional procedure-driven approach whereby the location of control defines the program state. A finite state machine can be implemented as a program. A transition forms an input statement, the main control path forms the sequence of instructions, the branches form the conditions, and the backward paths form the loops or iterations.
- **State Machine Engine** − This approach directly represents a state machine through a state machine engine class. This class executes the state machine through a set of transitions and actions provided by the application.
- **Control as Concurrent Tasks** − In this approach, an object is implemented as a task in the programming language or the operating system. Here, an event is implemented as an inter-task call. It preserves inherent concurrency of real objects.

Packaging Classes

In any large project, meticulous partitioning of an implementation into modules or packages is important. During object design, classes and objects are grouped into packages to enable multiple groups to work cooperatively on a project.

The different aspects of packaging are −

- **Hiding Internal Information from Outside View** − It allows a class to be viewed as a “black box” and permits class implementation to be changed without requiring any clients of the class to modify code.
- **Coherence of Elements** − An element, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts are intrinsically related so that they serve a common goal.

- **Construction of Physical Modules** − The following guidelines help while constructing physical modules −
    - Classes in a module should represent similar things or components in the same composite object.
    - Closely connected classes should be in the same module.
    - Unconnected or weakly connected classes should be placed in separate modules.
    - Modules should have good cohesion, i.e., high cooperation among its components.
    - A module should have low coupling with other modules, i.e., interaction or interdependence between modules should be minimum.

Design Optimization

The analysis model captures the logical information about the system, while the design model adds details to support efficient information access. Before a design is implemented, it should be optimized so as to make the implementation more efficient. The aim of optimization is to minimize the cost in terms of time, space, and other metrics.

However, design optimization should not be excess, as ease of implementation, maintainability, and extensibility are also important concerns. It is often seen that a perfectly optimized design is more efficient but less readable and reusable. So the designer must strike a balance between the two.

The various things that may be done for design optimization are −

- Add redundant associations
- Omit non-usable associations
- Optimization of algorithms
- Save derived attributes to avoid re-computation of complex expressions

Addition of Redundant Associations

During design optimization, it is checked if deriving new associations can reduce access costs. Though these redundant associations may not add any information, they may increase the efficiency of the overall model.

Omission of Non-Usable Associations

Presence of too many associations may render a system indecipherable and hence reduce the overall efficiency of the system. So, during optimization, all non-usable associations are removed.

Optimization of Algorithms

In object-oriented systems, optimization of data structure and algorithms are done in a collaborative manner. Once the class design is in place, the operations and the algorithms need to be optimized.

Optimization of algorithms is obtained by −

- Rearrangement of the order of computational tasks
- Reversal of execution order of loops from that laid down in the functional model
- Removal of dead paths within the algorithm

Saving and Storing of Derived Attributes

Derived attributes are those attributes whose values are computed as a function of other attributes (base attributes). Re-computation of the values of derived attributes every time they are needed is a time–consuming procedure. To avoid this, the values can be computed and stored in their computed forms.

However, this may pose update anomalies, i.e., a change in the values of base attributes with no corresponding change in the values of the derived attributes. To avoid this, the following steps are taken −

- With each update of the base attribute value, the derived attribute is also re-computed.
- All the derived attributes are re-computed and updated periodically in a group rather than after each update.

Design Documentation

Documentation is an essential part of any software development process that records the procedure of making the software. The design decisions need to be documented for any non–trivial software system for transmitting the design to others.

Usage Areas

Though a secondary product, a good documentation is indispensable, particularly in the following areas −

- In designing software that is being developed by a number of developers
- In iterative software development strategies
- In developing subsequent versions of a software project
- For evaluating a software
- For finding conditions and areas of testing
- For maintenance of the software.

Contents

A beneficial documentation should essentially include the following contents −

- **High–level system architecture** − Process diagrams and module diagrams
- **Key abstractions and mechanisms** − Class diagrams and object diagrams.
- **Scenarios that illustrate the behavior of the main aspects** − Behavioural diagrams

Features

The features of a good documentation are −

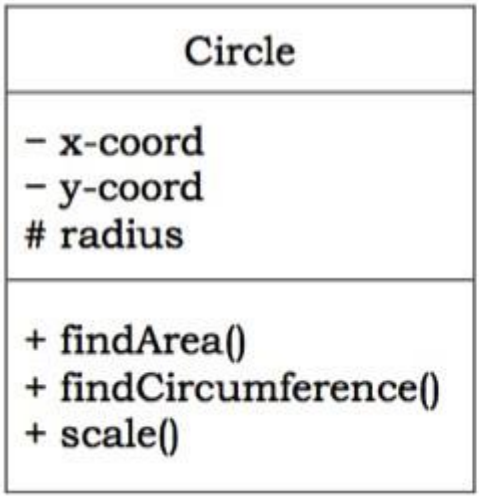
- Concise and at the same time, unambiguous, consistent, and complete
- Traceable to the system's requirement specifications
- Well-structured
- Diagrammatic instead of descriptive

Implementing an object-oriented design generally involves using a standard object oriented programming language (OOPL) or mapping object designs to databases. In most cases, it involves both.

Implementation using Programming Languages

Usually, the task of transforming an object design into code is a straightforward process. Any object-oriented programming language like C++, Java, Smalltalk, C# and Python, includes provision for representing classes. In this chapter, we exemplify the concept using C++.

The following figure shows the representation of the class Circle using C++.




```
class Circle
{
   private:
     double x_coord;
     double y_coord;
   protected:
     double radius;
   public:
     double findArea();
     double findCircumference();
     void scale();
};
```

Implementing Associations

Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

Associations may be either unidirectional or bidirectional. Besides, each association may be either one–to–one, one–to–many, or many–to–many.

Unidirectional Associations

For implementing unidirectional associations, care should be taken so that unidirectionality is maintained. The implementations for different multiplicity are as follows −

- **Optional Associations** − Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++ −

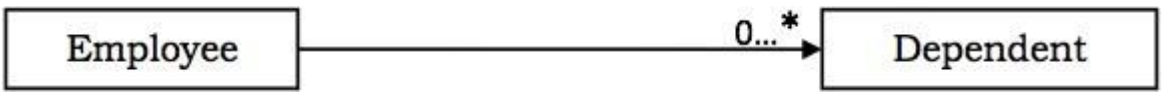
```
class Customer {
   private:
   // attributes
   Current_Account c; //an object of Current_Account as attribute

   public:

   Customer() {
      c = NULL;
   } // assign c as NULL

   Current_Account getCurrAc() {
      return c;
   }

   void setCurrAc( Current_Account myacc) {
      c = myacc;
   }

   void removeAcc() {
      c = NULL;
   }
};
```

- **One–to–one Associations** − Here, one instance of a class is related to exactly one instance of the associated class. For example, Department and Manager have one–to–one association as shown in the figure below.

This is implemented by including in Department, an object of Manager that should not be NULL. Implementation using C++ −

```
class Department {
   private:
   // attributes
   Manager mgr; //an object of Manager as attribute

   public:
   Department (/*parameters*/, Manager m) { //m is not NULL
      // assign parameters to variables
      mgr = m;
   }

   Manager getMgr() {
      return mgr;
   }
};
```

- **One–to–many Associations** − Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.



This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container −

```
class Employee {
   private:
```

```
   char * deptName;
   list <Dependent> dep; //a list of Dependents as attribute

   public:
   void addDependent ( Dependent d) {
      dep.push_back(d);
   } // adds an employee to the department

   void removeDeoendent( Dependent d) {
      int index = find ( d, dep );
      // find() function returns the index of d in list dep
      dep.erase(index);
   }
};
```

Bi-directional Associations

To implement bi-directional association, links in both directions require to be maintained.

- **Optional or one–to–one Associations** − Consider the relationship between Project and Project Manager having one–to–one bidirectional association as shown in the figure below.

Implementation using C++ −

```
Class Project {
   private:
   // attributes
   Project_Manager pmgr;
   public:
   void setManager ( Project_Manager pm);
   Project_Manager changeManager();
};

class Project_Manager {
   private:
   // attributes
   Project pj;

   public:
   void setProject(Project p);
   Project removeProject();
};
```

- **One–to–many Associations** − Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

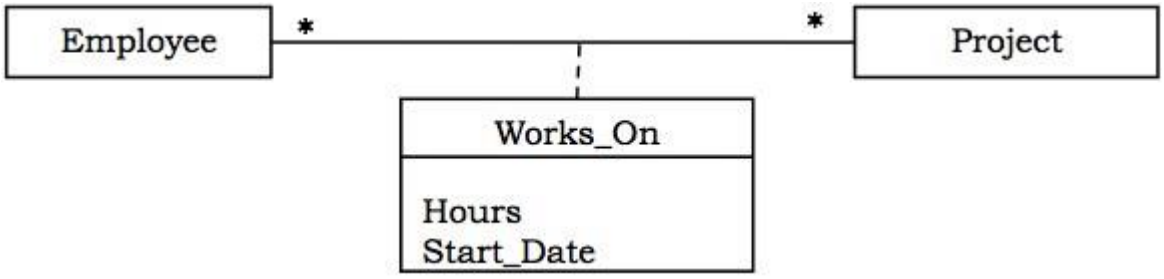
Implementation using C++ STL list container

```
class Department {
   private:
   char * deptName;
   list <Employee> emp; //a list of Employees as attribute

   public:
   void addEmployee ( Employee e) {
      emp.push_back(e);
   } // adds an employee to the department

   void removeEmployee( Employee e) {
      int index = find ( e, emp );
      // find function returns the index of e in list emp
      emp.erase(index);
   }
};
```

```
class Employee {
   private:
   //attributes
   Department d;

   public:
   void addDept();
   void removeDept();
};
```

Implementing Associations as Classes

If an association has some attributes associated, it should be implemented using a separate class. For example, consider the one–to–one association between Employee and Project as shown in the figure below.



Implementation of WorksOn using C++

```
class WorksOn {
   private:
   Employee e;
   Project p;
   Hours h;
   char * date;

   public:
   // class methods
};
```

Implementing Constraints

Constraints in classes restrict the range and type of values that the attributes may take. In order to implement constraints, a valid default value is assigned to the attribute when an object is instantiated from the class. Whenever the value is changed at runtime, it is checked whether the value is valid or not. An invalid value may be handled by an exception handling routine or other methods.

Example

Consider an Employee class where age is an attribute that may have values in the range of 18 to 60. The following C++ code incorporates it −

```
class Employee {
   private: char * name;
   int age;
   // other attributes

   public:
   Employee() {                    // default constructor
      strcpy(name, "");
      age = 18;                 // default value
   }

   class AgeError {};           // Exception class
   void changeAge( int a) {    // method that changes age
      if ( a < 18 || a > 60 )   // check for invalid condition
      throw AgeError();         // throw exception
      age = a;
   }
};
```

Implementing State Charts

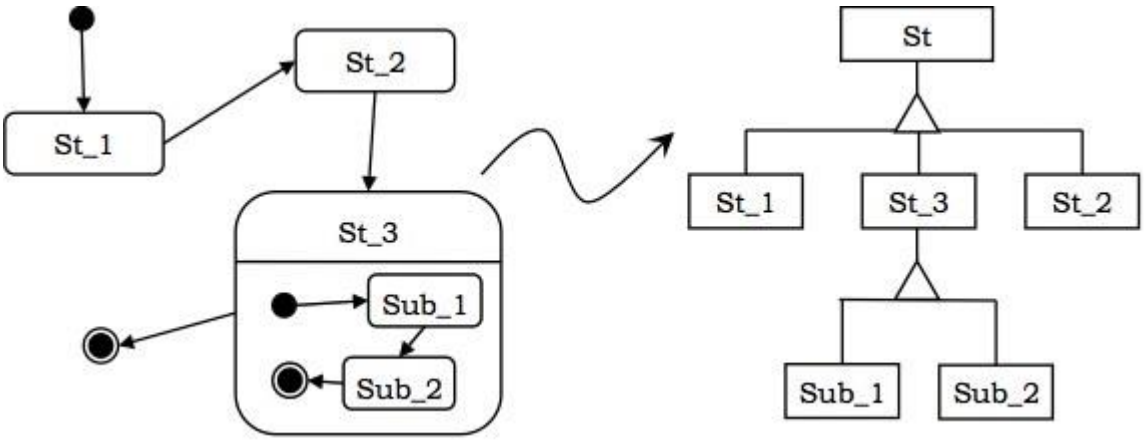
There are two alternative implementation strategies to implement states in state chart diagrams.

Enumerations within Class

In this approach, the states are represented by different values of a data member (or set of data members). The values are explicitly defined by an enumeration within the class. The transitions are represented by member functions that change the value of the concerned data member.

Arrangement of Classes in a Generalization Hierarchy

In this approach, the states are arranged in a generalization hierarchy in a manner that they can be referred by a common pointer variable. The following figure shows a transformation from state chart diagram to a generalization hierarchy.



Object Mapping to Database System

Persistency of Objects

An important aspect of developing object-oriented systems is persistency of data. Through persistency, objects have longer lifespan than the program that created it. Persistent data is saved on secondary storage medium from where it can be reloaded when required.

Overview of RDBMS

A database is an ordered collection of related data.

A database management system (DBMS) is a collection of software that facilitates the processes of defining, creating, storing, manipulating, retrieving, sharing, and removing data in databases.

In relational database management systems (RDBMS), data is stored as relations or tables, where each column or field represents an attribute and each row or tuple represents a record of an instance.

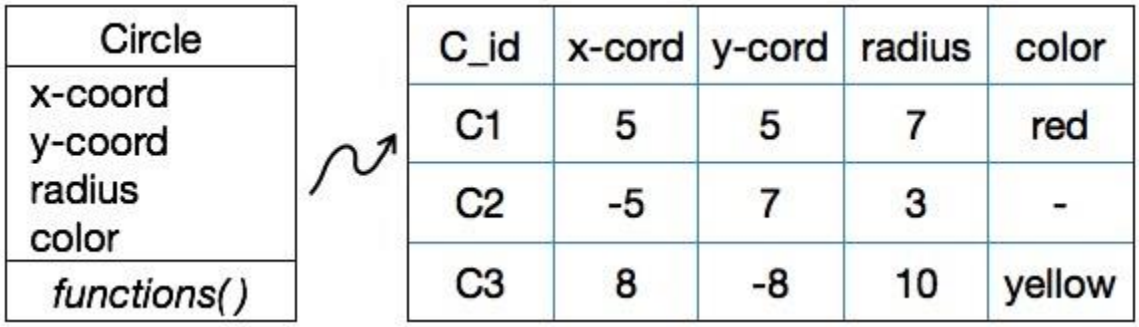
Each row is uniquely identified by a chosen set of minimal attributes called **primary key**.

A **foreign key** is an attribute that is the primary key of a related table.

Representing Classes as Tables in RDBMS

To map a class to a database table, each attribute is represented as a field in the table. Either an existing attribute(s) is assigned as a primary key or a separate ID field is added as a primary key. The class may be partitioned horizontally or vertically as per requirement.

For example, the Circle class can be converted to table as shown in the figure below.




```
Schema for Circle Table: CIRCLE(CID, X_COORD, Y_COORD, RADIUS, COLOR)
Creating a Table Circle using SQL command:
CREATE TABLE CIRCLE (
   CID    VARCHAR2(4) PRIMARY KEY,
   X_COORD INTEGER NOT NULL,
   Y_COORD INTEGER NOT NULL,
   Z_COORD INTEGER NOT NULL,
   COLOR
);
```

Mapping Associations to Database Tables

One–to–One Associations

To implement 1:1 associations, the primary key of any one table is assigned as the foreign key of the other table. For example, consider the association between Department and Manager −

SQL commands to create the tables

```
CREATE TABLE DEPARTMENT (
   DEPT_ID INTEGER PRIMARY KEY,
   DNAME VARCHAR2(30) NOT NULL,
   LOCATION VARCHAR2(20),
   EMPID INTEGER REFERENCES MANAGER
);

CREATE TABLE MANAGER (
   EMPID INTEGER PRIMARY KEY,
   ENAME VARCHAR2(50) NOT NULL,
   ADDRESS VARCHAR2(70),
);
```

One–to–Many Associations

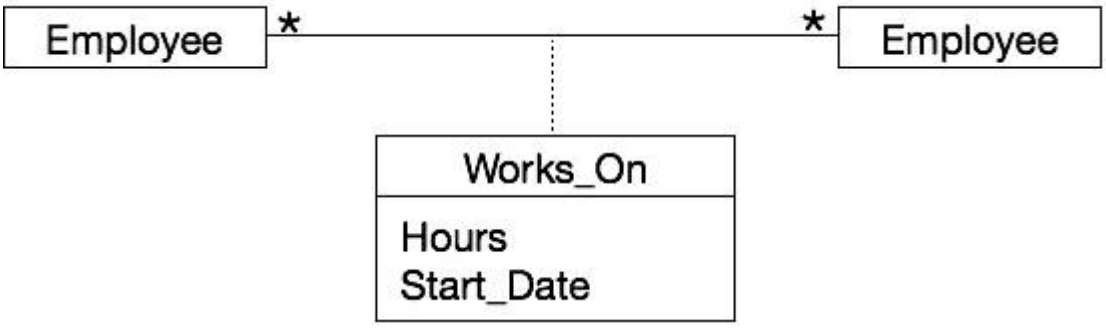
To implement 1:N associations, the primary key of the table in the 1-side of the association is assigned as the foreign key of the table at the N-side of the association. For example, consider the association between Department and Employee −

SQL commands to create the tables

```
CREATE TABLE DEPARTMENT (
   DEPT_ID INTEGER PRIMARY KEY,
   DNAME VARCHAR2(30) NOT NULL,
   LOCATION VARCHAR2(20),
);

CREATE TABLE EMPLOYEE (
   EMPID INTEGER PRIMARY KEY,
   ENAME VARCHAR2(50) NOT NULL,
   ADDRESS VARCHAR2(70),
   D_ID INTEGER REFERENCES DEPARTMENT
);
```

Many–to–Many Associations

To implement M:N associations, a new relation is created that represents the association. For example, consider the following association between Employee and Project −



Schema for Works_On Table − WORKS_ON (EMPID, PID, HOURS, START_DATE)

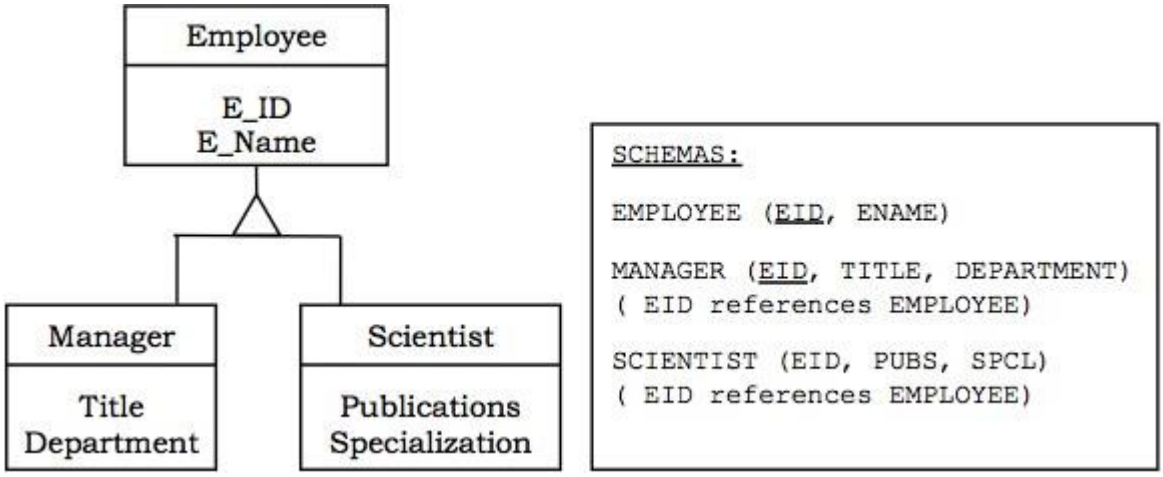
SQL command to create Works_On association − CREATE TABLE WORKS_ON

```
(
   EMPID INTEGER,
   PID INTEGER,
   HOURS INTEGER,
   START_DATE DATE,
   PRIMARY KEY (EMPID, PID),
   FOREIGN KEY (EMPID) REFERENCES EMPLOYEE,
   FOREIGN KEY (PID) REFERENCES PROJECT
);
```

Mapping Inheritance to Tables

To map inheritance, the primary key of the base table(s) is assigned as the primary key as well as the foreign key in the derived table(s).

Example



Once a program code is written, it must be tested to detect and subsequently handle all errors in it. A number of schemes are used for testing purposes.

Another important aspect is the fitness of purpose of a program that ascertains whether the program serves the purpose which it aims for. The fitness defines the software quality.

Testing Object-Oriented Systems

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

Unit Testing

In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure.

Subsystem Testing

This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem.

System Testing

System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.

Object-Oriented Testing Techniques

Grey Box Testing

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are −

- **State model based testing** − This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing** − Each scenario in each use case is tested.
- **Class diagram based testing** − Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing** − The methods in the messages in the sequence diagrams are tested.

Techniques for Subsystem Testing

The two main approaches of subsystem testing are −

- **Thread based testing** − All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing** − The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

Categories of System Testing

- **Alpha testing** − This is carried out by the testing team within the organization that develops software.
- **Beta testing** − This is carried out by select group of co-operating customers.
- **Acceptance testing** − This is carried out by the customer before accepting the deliverables.

Software Quality Assurance

Software Quality

Schulmeyer and McManus have defined software quality as “the fitness for use of the total software product”. A good quality software does exactly what it is supposed to do and is interpreted in terms of satisfaction of the requirement specification laid down by the user.

Quality Assurance

Software quality assurance is a methodology that determines the extent to which a software product is fit for use. The activities that are included for determining software quality are −

- Auditing
- Development of standards and guidelines
- Production of reports
- Review of quality system

Quality Factors

- **Correctness** − Correctness determines whether the software requirements are appropriately met.
- **Usability** − Usability determines whether the software can be used by different categories of users (beginners, non-technical, and experts).
- **Portability** − Portability determines whether the software can operate in different platforms with different hardware devices.
- **Maintainability** − Maintainability determines the ease at which errors can be corrected and modules can be updated.
- **Reusability** − Reusability determines whether the modules and classes can be reused for developing other software products.

Object-Oriented Metrics

Metrics can be broadly classified into three categories: project metrics, product metrics, and process metrics.

Project Metrics

Project Metrics enable a software project manager to assess the status and performance of an ongoing project. The following metrics are appropriate for object-oriented software projects −

- Number of scenario scripts

- Number of key classes
- Number of support classes
- Number of subsystems

Product Metrics

Product metrics measure the characteristics of the software product that has been developed. The product metrics suitable for object-oriented systems are −

- **Methods per Class** − It determines the complexity of a class. If all the methods of a class are assumed to be equally complex, then a class with more methods is more complex and thus more susceptible to errors.
- **Inheritance Structure** − Systems with several small inheritance lattices are more well–structured than systems with a single large inheritance lattice. As a thumb rule, an inheritance tree should not have more than 7 (± 2) number of levels and the tree should be balanced.
- **Coupling and Cohesion** − Modules having low coupling and high cohesion are considered to be better designed, as they permit greater reusability and maintainability.
- **Response for a Class** − It measures the efficiency of the methods that are called by the instances of the class.

Process Metrics

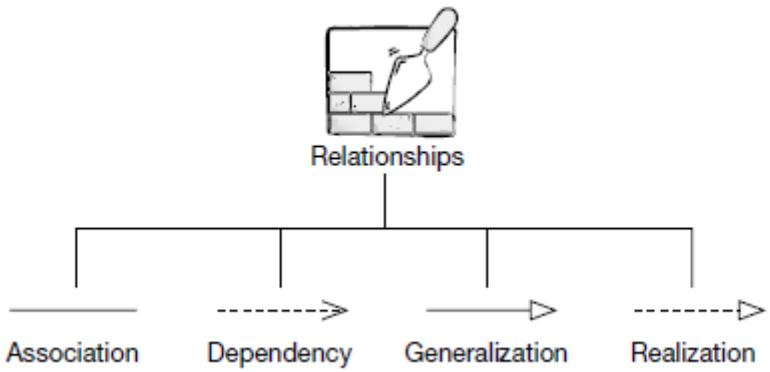
Process metrics help in measuring how a process is performing. They are collected over all projects over long periods of time. They are used as indicators for long-term software process improvements. Some process metrics are −

- Number of KLOC (Kilo Lines of Code)
- Defect removal efficiency
- Average number of failures detected during testing
- Number of latent defects per KLOC

What is UML Relationship?

Relationships in UML are used to represent a connection between structural, behavioral, or grouping things. It is also called a link that describes how two or more things can relate to each other during the execution of a system. Type of UML Relationship are Association, Dependency , Generalization , and Realization.

Lets study them in detail



- **Association**

It is a set of links that connects elements of the UML model. It also defines how many objects are taking part in that relation.

- **Dependency**

In a dependency relationship, as the name suggests, two or more elements are dependent on each other. In this kind of a relationship, if we make a change to a particular element, then it is likely possible that all the other elements will also get affected by the change.

- **Generalization**

It is also called a parent-child relationship. In generalization, one element is a specialization of another general component. It may be substituted for it. It is mostly used to represent inheritance.

- **Realization**

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of **interfaces.**

In this UML tutorial, you will learn:

Association

It is a structural relationship that represents objects can be connected or associated with another object inside the system. Following constraints can be applied to the association relationship.

- **{implicit}** – Implicit constraints specify that the relationship is not manifest; it is based upon a concept.
- **{ordered}** – Ordered constraints specify that the set of objects at one end of an association are in a specific way.
- **{changeable}** – Changeable constraint specifies that the connection between various objects in the system can be added, removed, and modified as per the requirement.
- **{addOnly}** – It specifies that the new connections can be added from an object which is situated at the other end an association.
- **{frozen}** – It specifies that when a link is added between two objects, then it cannot be modified while the frozen constraint is active on the given link or a connection.

We can also create a class that has association properties; it is called as an association class.

Reflexive association

The reflexive association is a subtype of association relationship in UML. In a reflexive association, the instances of the same class can be related to each other. An instance of a class is also said to be an object.

Reflexive association states that a link or a connection can be present within the objects of the same class.

Let us consider an example of a class fruit. The fruit class has two instances, such as mango and apple. Reflexive association states that a link between mango and apple can be present as they are instances of the same class, such as fruit.

Directed association

As the name suggests, the directed association is related to the direction of flow within association classes.

In a directed association, the flow is directed. The association from one class to another class flows in a single direction only.

It is denoted using a solid line with an arrowhead.

Example:

You can say that there is a directed association relationship between a server and a client. A server can process the requests of a client. This flow is unidirectional, that flows from server to client only. Hence a directed association relationship can be present within servers and clients of a system.

Dependency

Using a dependency relationship in UML, one can relate how various things inside a particular system are dependent on each other. Dependency is used to describe the relationship between various elements in UML that are dependent upon each other.

Stereotypes

- **«bind»** – Bind is a constraint which specifies that the source can initialize the template at a target location, using provided parameters or values.
- **«derive»** – It represents that the location of a source object can be calculated from the target object.
- **«friend»** – It specifies that the source has unique visibility in the target object.
- **«instanceOf»** – It specifies that the instance of a target classifier is the source object.
- **«instantiate»** – It specifies that the source object is capable of creating instances of a target object.
- **«refine»** – It specifies that the source object has exceptional abstraction than that of the target object.
- **«use»** – It is used when packages are created in UML. The use stereotype describes that the elements of a source package can be present inside the target package as well. It describes that the source package makes use of some elements of a target package.
- **«substitute»** - specifies that the client may be substituted for the supplier at runtime.
- **«access»** – It specifies that the source package access the elements of the target package **which is also called as a private merging.**
- **«import»** – It specifies that the target can import the element of a source package like they are defined inside the **target which is also called as a public merging.**
- **«permit»** - specifies that source element has access to the supplier element whatever the declared visibility of the supplier.
- **«extend»** – Helps you to specifies that the target can extend the behavior of the source element.
- **«include»** – Allows you to specifies the source element which can be included the behavior of another element at a specified location. (same as a function call in c/c++)
- **«become»** – It specifies that the target is similar to the source with different values and roles.
- **«call»** – It specifies that the source can invoke a target object method.
- **«copy»** – It specifies that the target object is independent, copy of a source object.
- **«parameter» - the** supplier is a parameter of the client operations**.**
- **«send» -** the client is an operation that sends the supplier some unspecified target.

Stereotypes among state machine

- **«send»** – Specifies that the source operation sends the target event.

Generalization

It is a relationship between a general entity and a unique entity which is present inside the system.

In a generalization relationship, the object-oriented concept called **inheritance** can be implemented. A generalization relationship exists between two objects, also called as entities or things. In a generalization relationship, one entity is a parent, and another is said to be as a child. These entities can be represented using inheritance.

In inheritance, a child of any parent can access, update, or inherit the functionality as specified inside the parent object. A child object can add its functionality to itself as well as inherit the structure and behavior of a parent object.

This type of relationship collectively known as a generalization relationship.

Stereotypes and their constraints

- **«implementation»** – This stereotype is used to represent that the child entity is being implemented by the parent entity by inheriting the structure and behavior of a parent object without violating the rules.**Note** This stereotype if widely used in a singl**e inheritance**.

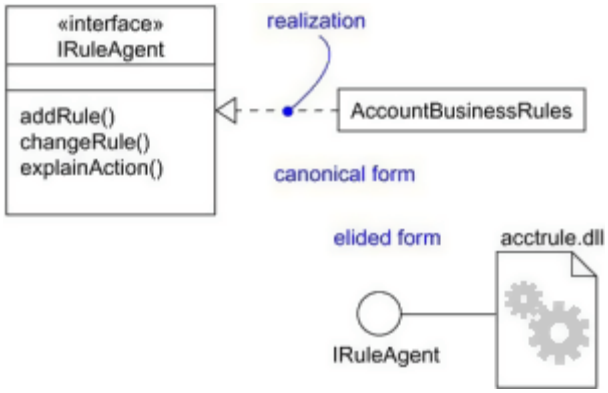
Generalization relationship contains constraints such as complete, incomplete to check whether all the child entities are being included in the relationship or not.

Realization

In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of **interfaces.**

Realization can be represented in two ways:

- Using a canonical form
- Using an elided form



In the above diagram, account business rules realize the interface IRuleAgent.

Types of realization:

1. Canonical form

In a realization relationship of UML, the canonical form is used to realize interfaces across the system. It uses an interface stereotype to create an interface and realization relationship is used to realize the particular interface.

In a canonical form, the realization relationship is denoted using the dashed directed line with a sizeable open arrowhead.

In the above diagram, interface Iruleagent is realized using an object called as Account Business Rules.

2. Elided form

   Realization in the UML class diagram can also be shown using an elided form. In an elided form, the interface is denoted using a circle which is also called as a lollipop notation.

   This interface, when realized using anything present inside the system, creates an elided structure.

   In the above diagram, the interface Iruleagent is denoted using an elided form which is being realized by acctrule.dll.

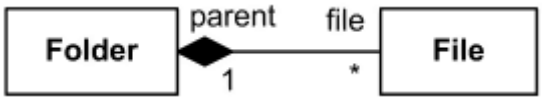
Composition

It is not a standard UML relationship, but it is still used in various applications.

Composite aggregation is a subtype of aggregation relation with characteristics as:

- it is a two-way association between the objects.
- It is a whole/part relationship.
- If a composite is deleted, all other parts associated with it are deleted.

Composite aggregation is described as a binary association decorated with a filled black diamond at the aggregate (whole) end.



A folder is a structure which holds n number of files in it. A folder is used to store the files inside it. Each folder can be associated with any number of files. In a computer system, every single file is a part of at least one folder inside the file organization system. The same file can also be a part of another folder, but it is not mandatory. Whenever a file is removed from the folder, the folder stays un-affected whereas the data related to that particular file is destroyed. If a delete operation is executed on the folder, then it also affects all the files which are present inside the folder. All the files associated with the folder are automatically destroyed once the folder is removed from the system.

This type of relationship in UML is known by composite aggregation relationship.

Aggregation

An aggregation is a subtype of an association relationship in UML. Aggregation and composition are both the types of association relationship in UML. An aggregation relationship can be described in simple words as " an object of one class can own or access the objects of another class."

In an aggregation relationship, the dependent object remains in the scope of a relationship even when the source object is destroyed.

Let us consider an example of a car and a wheel. A car needs a wheel to function correctly, but a wheel doesn't always need a car. It can also be used with the bike, bicycle, or any other vehicles but not a particular car. Here, the wheel object is meaningful even without the car object. Such type of relationship is called an aggregation relation.

Summary

- Relationship in UML allows one thing to relate with other things inside the system.
- An association, dependency, generalization, and realization relationships are defined by UML.
- Composition relationship can also be used to represent that object can be a part of only one composite at a time.
- Association is used to describe that one object can be associated with another object.
- Dependency denotes that objects can be dependent on each other.
- A realization is a meaningful relationship between classifiers.
- Generalization is also called as a parent-child relationship.