SMT K.B.PAREKH COLLEGE OF COMPUTER SCIENCE,MAHUVA

Unit: 2 Planning and Design of Software

Prepared by :Amit Bhaliya

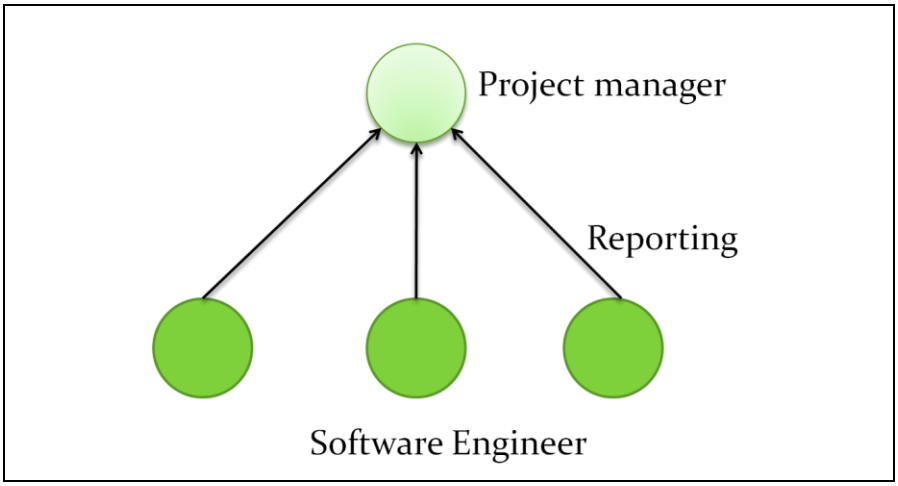
2020

Team structure

Team structure addresses organization of the individual project team. There are formally three different types of team structure are there such as,

1. Chief programmer team
2. Egoless Team
3. Mixed control tem organization.

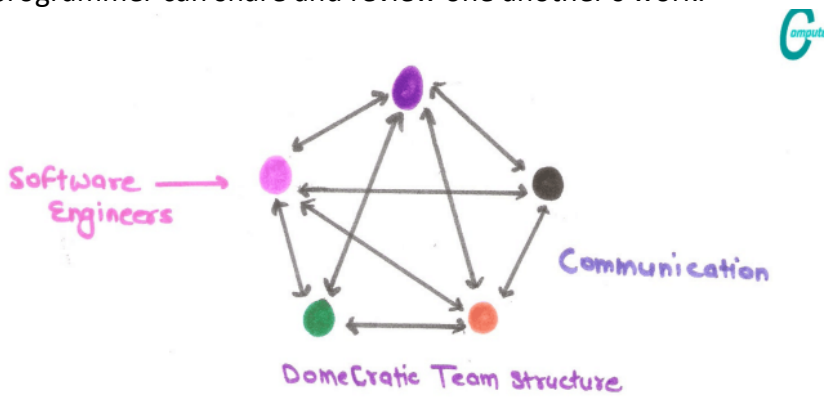
1.Chief Programmer Team:



- A **chief programmer team** is a programming team which is organized around a chief programmer who is both a programmer and an expert on the system's intentions.
- The other team members have other, specialized roles, such as librarian, which support the chief programmer in his primary task of designing and coding the software.
- In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer.
- The chief programmer partitions the task into small activities and assigns them to the team members.
- He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown as per above.
- However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking.
- The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

2.Egoless/Democratic Team:

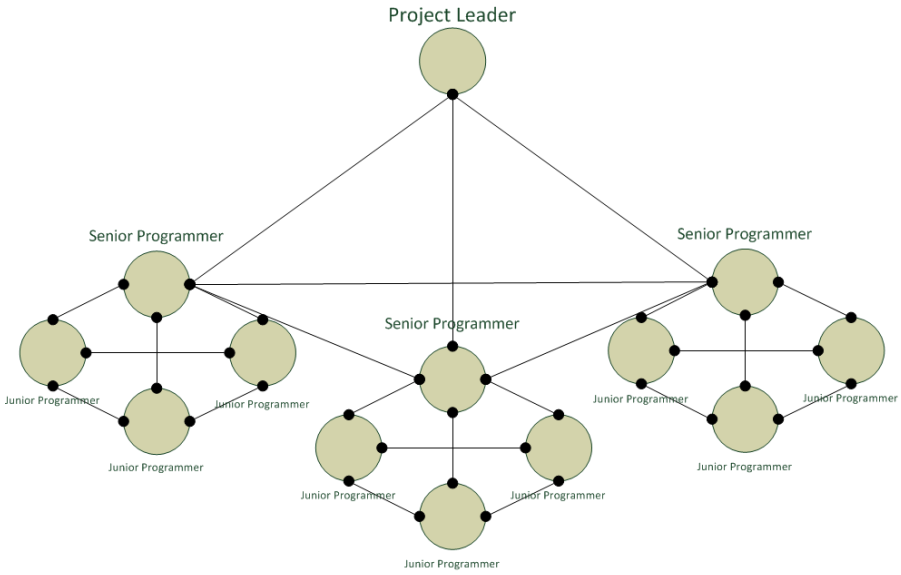
- According to Marilyn Mantei, individuals that are a part of a decentralized programming team report higher job satisfaction.[2]
- But an egoless programming team contains groups of ten or fewer programmers. Code is exchanged and goals are set amongst the group members.
- Leadership is rotated within the group according to the needs and abilities required during a specific time. T
- he lack of structure in the egoless team can result in a weakness of efficiency, effectiveness, and error detection for large-scale projects.
- Egoless programming teams work best for tasks that are very complex.
- In this a manager provides the administrative leadership, at different times, different members of the group provide technical leadership.
- It suffers from less manpower turnover. Democratic team structure suits to less understood problems, since a group of Engineers can invent better solution than a single individual as in a chief programmer team. It encourages egoless programming as programmer can share and review one another's work.



3.Control decentralized Team Organization:

- In a decentralized-control team organization, decisions are made by consensus, and all work is considered group work.
- Team members review each other's work and are responsible as a group for what every member produces.

Controlled decentralized team



- Figure above shows the patterns of control and communication among team members in a decentralized-control organization. The ring like management structure is intended to show the lack of a hierarchy and that all team members are at the same level.
- Such a "democratic" organization leads to higher morale and job satisfaction and, therefore, to less turnover. The engineers feel more ownership of the project and responsibility for the problem, resulting in higher quality in their work.
- A decentralized-control organization is more suited for long-term projects, because the amount of intragroup communication that it encourages leads to a longer development time, presumably accompanied by lower life cycle costs.
- The proponents of this kind of team organization claim that it is more appropriate for less understood and more complicated problems, since a group can invent better solutions than a single individual can. Such an organization is based on a technique referred to as "egoless programming," because it encourages programmers to share and review one another's work.
- On the negative side, decentralized-control team organization is not appropriate for large teams, where the communication overhead can overwhelm all the engineers, reducing their individual productivity.

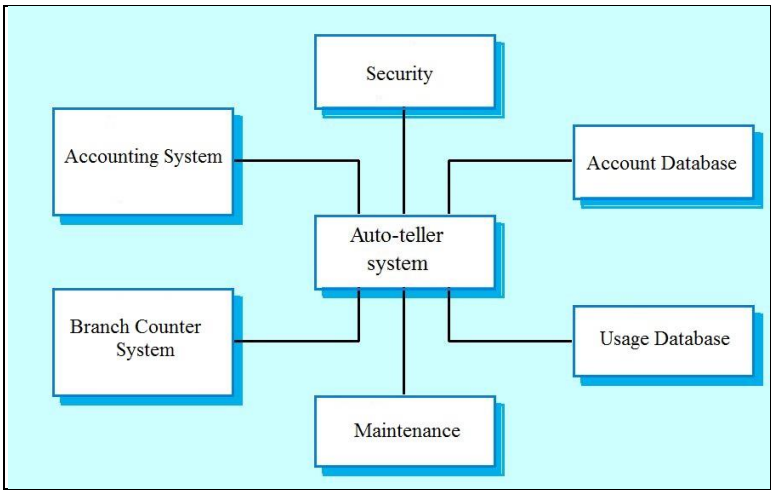
Cohesion and coupling

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

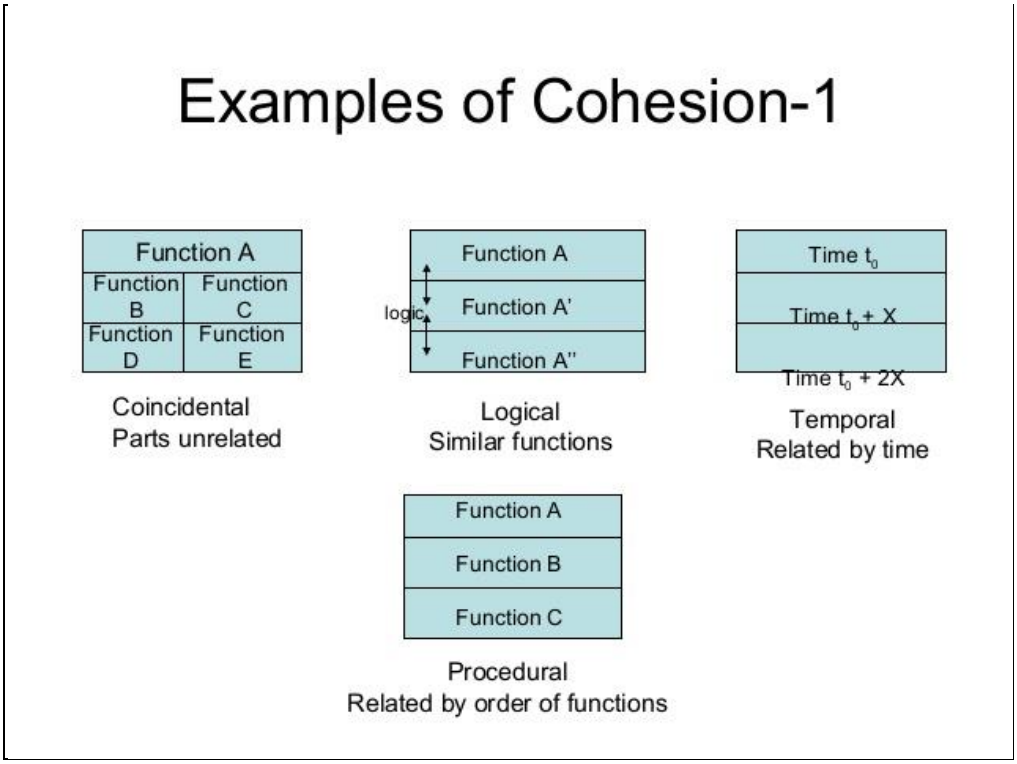
Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

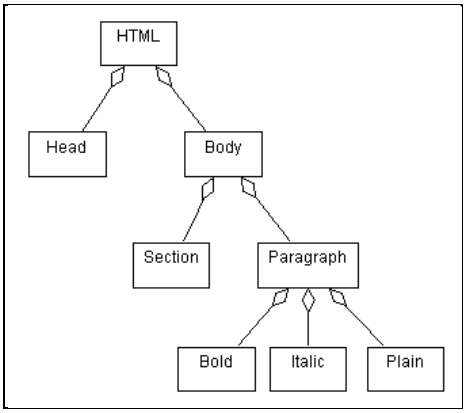


There are seven types of cohesion, namely –

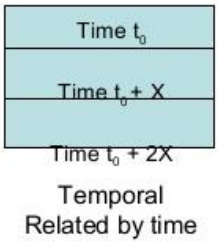
- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.



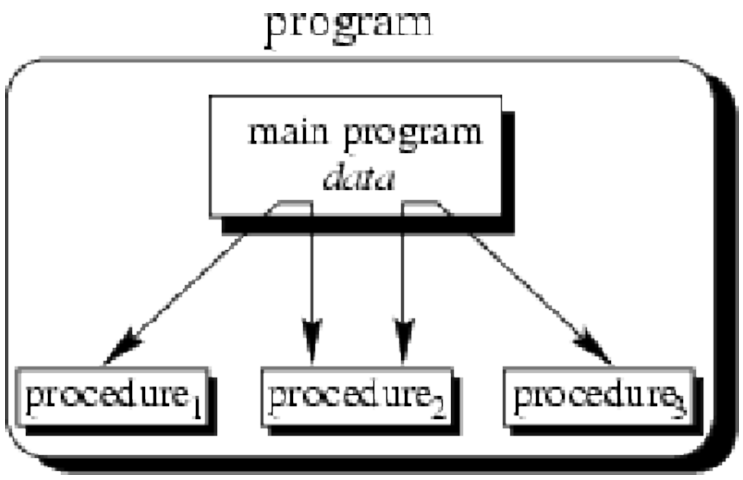
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.



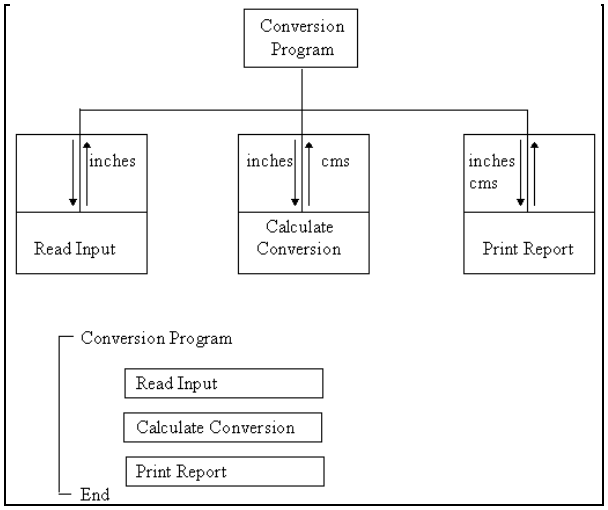
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.



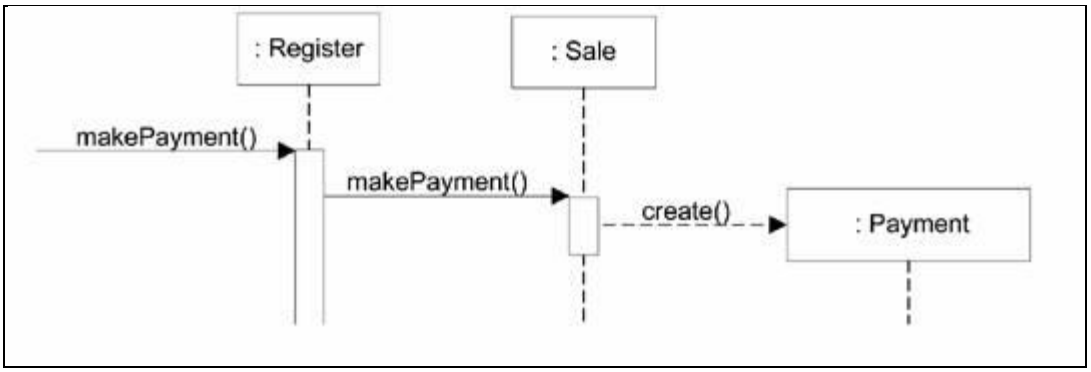
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.



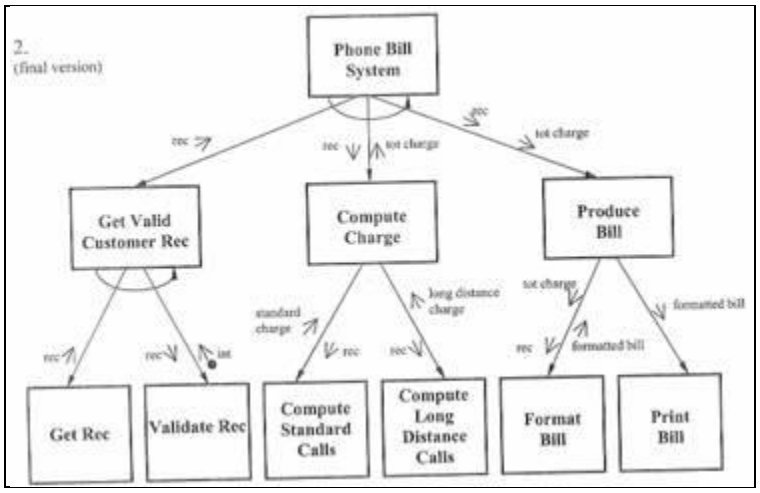
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.



- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

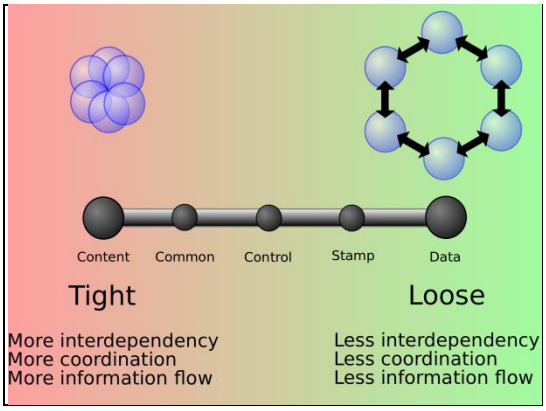


- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It
- can also be reused.



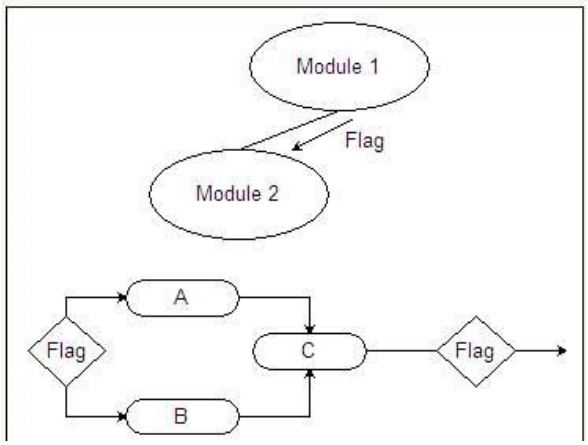
Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

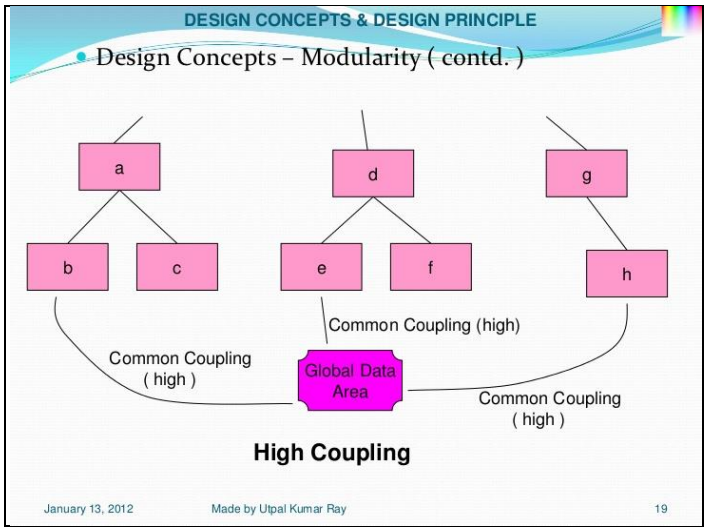


There are five levels of coupling, namely -

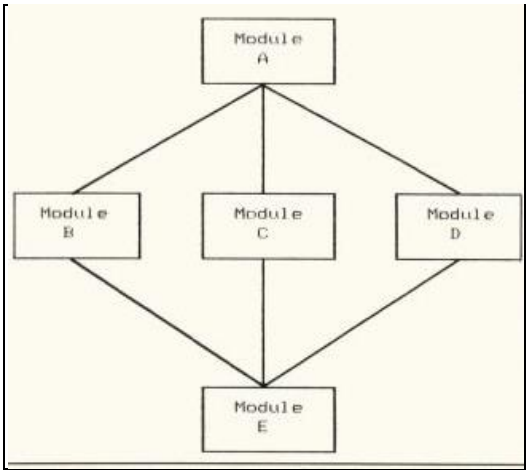
- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.



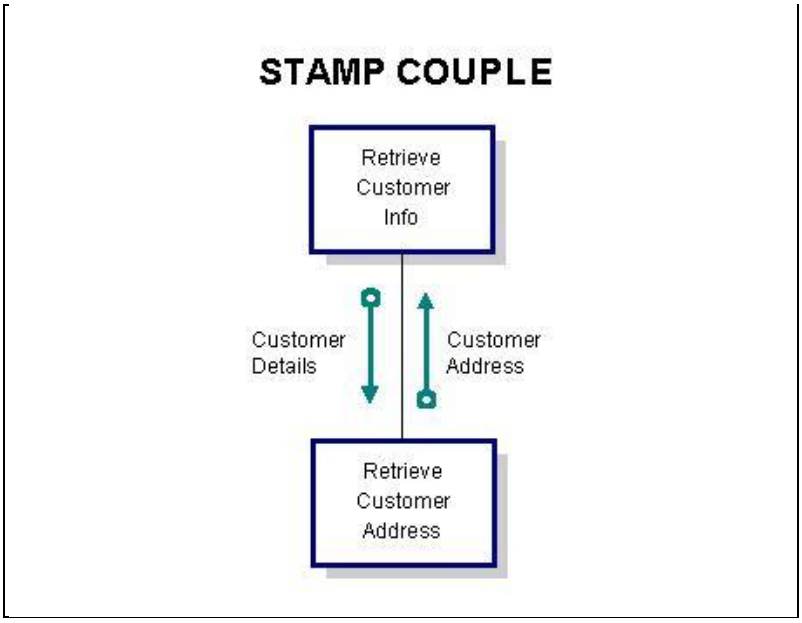
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.



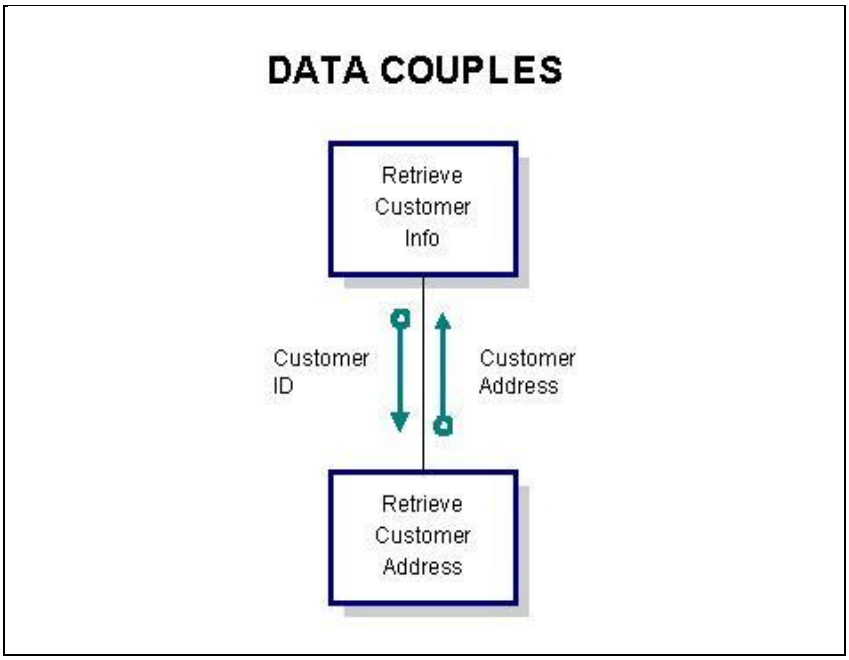
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.



- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.



- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.



Ideally, no coupling is considered to be the best.

Difference between cohesion and coupling

Cohesion is the indication of the relationship within **module**.	**Coupling** is the indication of the relationships between modules.
Cohesion shows the module's relative **functional** strength.	Coupling shows the relative **independence** among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the **single** thing.	Coupling is a degree to which a component / module is connected to the **other** modules.
While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system.	While designing you should strive for **low coupling** i.e. **dependency** between modules should be less.
Cohesion is the kind of natural extension of data hiding	Making private fields, private methods and

for example, **class** having all members visible with a package having default visibility.	non public classes provides loose coupling.
Cohesion is **Intra – Module** Concept.	Coupling is **Inter -Module** Concept.

Software Quality Assurance Plan

Software Quality

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component or process meets customer or user needs or expectations.

Software Quality Assurance

- A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
- A set of activities designed to evaluate the process by which products are developed or manufactured.

Software Quality Assurance Plan

The Software Quality Assurance Plan shall include the sections listed below to be in compliance with this standard. The sections should be ordered in the described sequence. If the sections are not ordered in the described sequence, then a table shall be provided at the end of the SQAP that provides a cross-reference from the lowest numbered subsection of this standard to that portion of the SQAP where that material is provided. If there is no information pertinent to a section, the following shall appear below the section heading, "This section is not applicable to this plan," together with the appropriate reasons for the exclusion.

1. Purpose;
2. Reference documents;
3. Management;
4. Documentation;
5. Standards, practices, conventions, and metrics;
6. Reviews and audits;
7. Test;
8. Problem reporting and corrective action;
9. Tools, techniques, and methodologies;
10. Code control;
11. Media control;
12. Supplier control;
13. Records collection, maintenance, and retention;
14. Training;
15. Risk management.

1. Purpose

This section shall delineate the specific purpose and scope of the particular SQAP. It shall list the name(s) of the software items covered by the SQAP and the intended use of the software. It shall state the portion of the software life cycle covered by the SQAP for each software item specified.

2. Reference documents

This section shall provide a complete list of documents referenced elsewhere in the text of the SQAP.

3. Management

This section shall describe organization, tasks, and responsibilities.

a. Organization
b. Tasks
c. Responsibilities

a. ***Organization***

This paragraph shall depict the organizational structure that influences and controls the quality of the software. This shall include a description of each major element of the organization together with the delegated responsibilities. Organizational dependence or independence of the elements responsible for SQA from those responsible for software development and use shall be clearly described or depicted.

b. ***Tasks***

This paragraph shall describe

a) That portion of the software life cycle covered by the SQAP;

b) The tasks to be performed with special emphasis on software quality assurance activities; and

c) The relationships between these tasks and the planned major checkpoints.

The sequence of the tasks shall be indicated.

c. ***Responsibilities***

This paragraph shall identify the specific organizational elements responsible for each task.

4. Documentation

- **a.** Purpose
- **b.** Minimum documentation requirements
  - **i.** Software Requirements Specification (SRS)
  - **ii.** Software Design Description (SDD).
  - **iii.** Software Verification and Validation Plan (SVVP).
  - **iv.** Software Verification and Validation Report (SVVR).
  - **v.** User documentation
  - **vi.** Software Configuration Management Plan (SCMP)
- **c.** Other.

a. Purpose:

This section shall perform the following functions:

a) Identify the documentation governing the development, verification and validation, use, and maintenance of the software.

b) State how the documents are to be checked for adequacy. This shall include the criteria and the identification of the review or audit by which the adequacy of each document shall be confirmed, with reference to Section 6 of the SQAP.

b. Minimum documentation requirements

To ensure that the implementation of the software satisfies requirements, the documentation in 4.4.2.1 through 4.4.2.6 is required as a minimum.

1) ***Software Requirements Specification (SRS)***

i. The SRS shall clearly and precisely describe each of the essential requirements (functions, performances, design constraints, and attributes) of the software and the external interfaces. Each requirement shall be defined such that its achievement is capable of being objectively verified and validated by a prescribed method (e.g., inspection, analysis, demonstration, or test).

2) Software Design Description (SDD)

i. The SDD shall depict how the software will be structured to satisfy the requirements in the SRS. The SDD shall describe the components and subcomponents of the software design, including databases and internal interfaces. The SDD shall be prepared first as the Preliminary SDD (also referred to as the top-level SDD) and shall be subsequently expanded to produce the Detailed SDD.

3) Software Verification and Validation Plan (SVVP)

The SVVP shall identify and describe the methods (e.g., inspection, analysis, demonstration, or test) to be used to19

a. Verify that
   - The requirements in the SRS have been approved by an appropriate authority;
   - The requirements in the SRS are implemented in the design expressed in the SDD; and
   - The design expressed in the SDD is implemented in the code.
b. Validate that the code, when executed, complies with the requirements expressed in the SRS.

4) Software Verification and Validation Report (SVVR)

The SVVR shall describe the results of the execution of the SVVP.

5) ***User documentation User documentation***

(e.g., manual, guide) shall specify and describe the required data and control inputs, input sequences, options, program limitations, and other activities or items necessary for successful execution of the software. All error messages shall be identified and corrective actions shall be described. A method of describing user identified errors or problems to the developer or the owner of the software shall be described. (Embedded software that has no direct user interaction has no need for user documentation and is therefore exempted from this requirement.)

6) ***Software Configuration Management Plan (SCMP)***

The SCMP shall document methods to be used for identifying software items, controlling and implementing changes, and recording and reporting change implementation status

c. Other

Other documentation may include the following:
a) Software Development Plan;
b) Standards and Procedures Manual;
c) Software Project Management Plan;
d) Software Maintenance Manual.

5. Standards, practices, conventions, and metrics

1 Purpose
This section shall
a) Identify the standards, practices, conventions, and metrics to be applied;
b) State how compliance with these items is to be monitored and assured. 4.5.2 Content The subjects covered shall include the basic technical, design, and programming activities involved, such as documentation, variable and module naming, programming, inspection, and testing. As a minimum, the following information shall be provided:22

a) Documentation standards;

b) Logic structure standards;
c) Coding standards;
d) Commentary standards;
e) Testing standards and practices;
f) Selected software quality assurance product and process metrics such as
    a. Branch metric;
    b. Decision point metric;
    c. Domain metric;
    d. Error message metric;
    e. Requirements demonstration metric.

6. Reviews and audits

a. Purpose

This section shall

a) Define the technical and managerial reviews and audits to be conducted;
b) State how the reviews and audits are to be accomplished;
c) State what further actions are required and how they are to be implemented and verified.

b. Minimum requirements

As a minimum, the reviews and audits in 4.6.2.1 through 4.6.2.10 shall be conducted

1. Software Requirements Review (SRR)

The SRR is held to ensure the adequacy of the requirements stated in the SRS.

2. Preliminary Design Review (PDR)

The PDR (also known as the top-level design review) is held to evaluate the technical adequacy of the preliminary design (also known as the top-level design) of the software as depicted in the preliminary software design description.

3. Critical Design Review (CDR)

The CDR (also known as detailed design review) is held to determine the acceptability of the detailed software designs as depicted in the detailed software design description in satisfying the requirements of the SRS.

4. Software Verification and Validation Plan Review (SVVPR)

The SVVPR is held to evaluate the adequacy and completeness of the verification and validation methods defined in the SVVP.

5. Functional audit

This audit is held prior to the software delivery to verify that all requirements specified in the SRS have been met.

6 Physical audit

This audit is held to verify that the software and its documentation are internally consistent and are ready for delivery

7. In-process audits
In-process audits of a sample of the design are held to verify consistency of the design, including the following: a) Code versus design documentation; b) Interface specifications (hardware and software); c) Design implementations versus functional requirements; d) Functional requirements versus test descriptions.

8 Managerial reviews
Managerial reviews are held periodically to assess the execution of all of the actions and the items identified in the SQAP. These reviews shall be held by an organizational element independent of the unit being reviewed, or by a qualified third party. This review may require additional changes in the SQAP itself.

9 Software Configuration Management Plan Review (SCMPR)
The SCMPR is held to evaluate the adequacy and completeness of the configuration management methods defined in the SCMP.

10 Post-mortem reviews
This review is held at the conclusion of the project to assess the development activities implemented on that project and to provide recommendations for appropriate actions.

c. Other
Other reviews and audits may include the user documentation review (UDR). This review is held to evaluate the adequacy (e.g., completeness, clarity, correctness, and usability) of user documentation.

7. Test
This section shall identify all the tests not included in the SVVP for the software covered by the SQAP and shall state the methods to be used.

8. Problem reporting and corrective action
This section shall a) Describe the practices and procedures to be followed for reporting, tracking, and resolving problems identified in both software items and the software development and maintenance process; b) State the specific organizational responsibilities concerned with their implementation.

9. Tools, techniques, and methodologies
This section shall identify the special software tools, techniques, and methodologies that support SQA, state their purposes, and describe their use.

10. Code control
This section shall define the methods and facilities used to maintain, store, secure, and document controlled versions of the identified software during all phases of the software life cycle. This may be implemented in conjunction with a computer program library. This may be provided as a part of the SCMP. If so, an appropriate reference shall be made thereto

11. Media control

This section shall state the methods and facilities to be used to a) Identify the media for each computer product and the documentation required to store the media, including the copy and restore process; and b) Protect computer program physical media from unauthorized access or inadvertent damage or degradation during all phases of the software life cycle. This may be provided as a part of the SCMP. If so, an appropriate reference shall be made thereto.

12. Supplier control
This section shall state the provisions for assuring that software provided by suppliers meets established requirements. In addition, this section shall state the methods that will be used to assure that the software supplier receives adequate and complete requirements. For previously developed software, this section shall state the methods to be used to assure the suitability of the product for use with the software items covered by the SQAP. For software that is to be developed, the supplier shall be required to prepare and implement an SQAP in accordance with this standard. This section shall also state the methods to be employed to assure that the developers comply with the requirements of this standard.

13 Records collection, maintenance, and retention This section shall identify the SQA documentation to be retained; shall state the methods and facilities to be used to assemble, safeguard, and maintain this documentation; and shall designate the retention period.

14. Training This section shall identify the training activities necessary to meet the needs of the SQAP.

15. Risk management
This section shall specify the methods and procedures employed to identify, assess, monitor, and control areas of risk arising during the portion of the software life cycle covered by the SQAP.

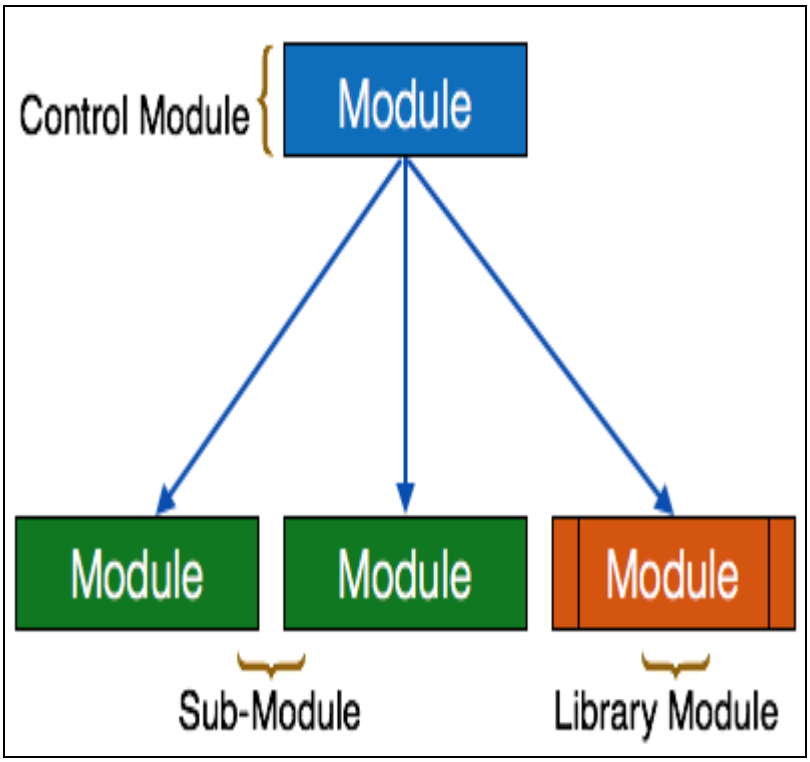
Structure Charts

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

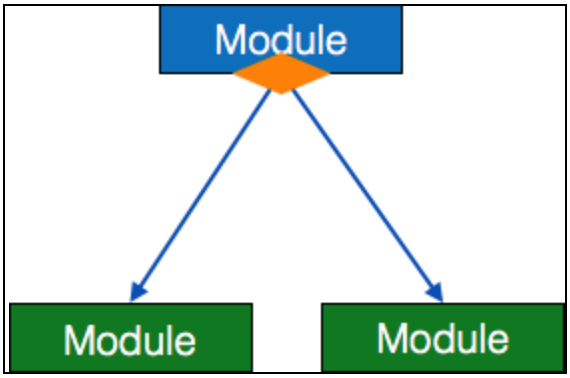
Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

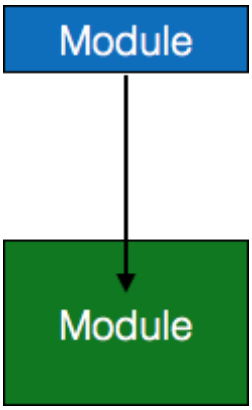
- **Module** - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and in vokable from any Module.



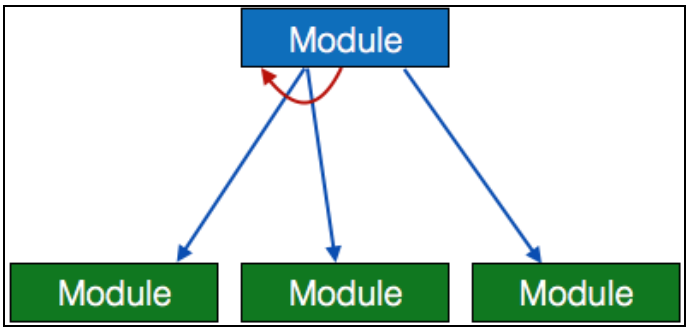
- **Condition** - It is represented by small diamond at the base of module. It depicts that control module can select any of sub-routine based on some condition.



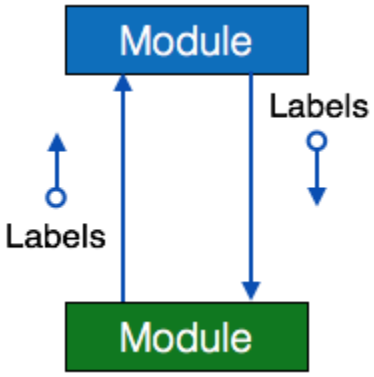
- **Jump** - An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.



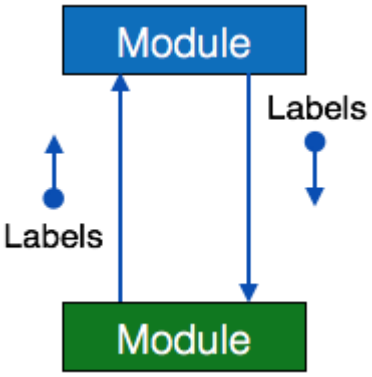
- **Loop** - A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



- **Data flow** - A directed arrow with empty circle at the end represents data flow.



- **Control flow** - A directed arrow with filled circle at the end represents control flow.



System Design Principles

Software design can be viewed as both a process and a model.

"The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. However, it is not merely a cookbook; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes "good" software, and have a commitment to quality.

The design model is equivalent to the architect's plans for a house. It begins by representing the totality of the entity to be built (e.g. a 3D rendering of the house), and slowly refines the entity to provide guidance for constructing each detail (e.g. the plumbing layout). Similarly the design model that is created for software provides a variety of views of the computer software." – adapted from book by R Pressman.

The set of principles which has been established to aid the software engineer in navigating the design process are:

1. The design process should not suffer from **tunnel vision** – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints.

2. The design should be **traceable** to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model

3. The design should not **reinvent** the wheel – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

4. The design should minimise **intellectual** distance between the software and the problem as it exists in the real world – That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.

5. The design should exhibit **uniformity and integration** – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

6. The design should be structured to **degrade** gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

7. The design should be **reviewed** to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax if the design model.

8. Design is not coding, coding is not design – Even when **detailed designs** are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.

9. The design should be **structured** to accommodate change

10. The design should be assessed for **quality** as it is being created

When these design principles are properly applied, the design exhibits both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors relate to the technical quality (which is important to the software engineer) more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

DFD

Data Flow Diagram

Data flow diagram is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

Types of DFD

Data Flow Diagrams are either Logical or Physical.

- **Logical DFD** - This type of DFD concentrates on the system process, and flow of data in the system. For example in a Banking software system, how data is moved between different entities.
- **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components

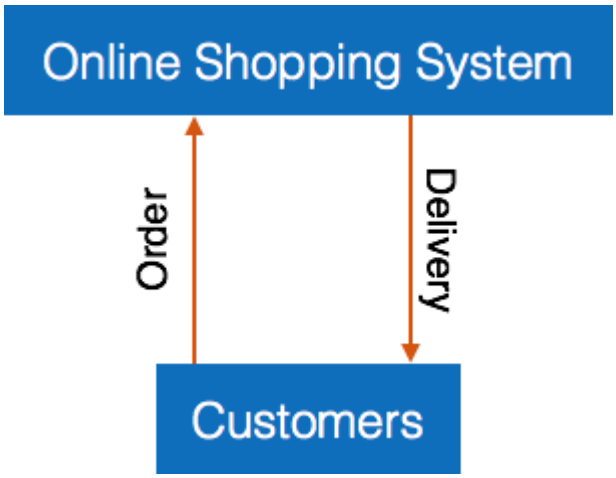
DFD can represent Source, destination, storage and flow of data using the following set of components -

- **Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.

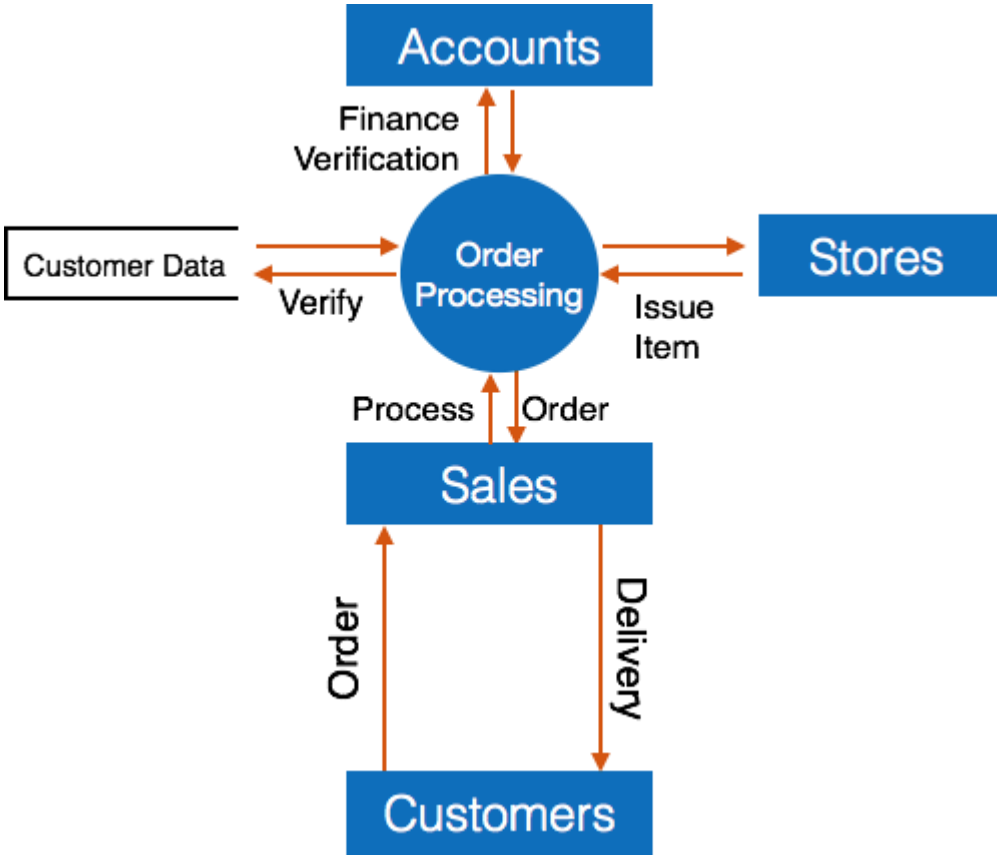
- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



- **Level 1** - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



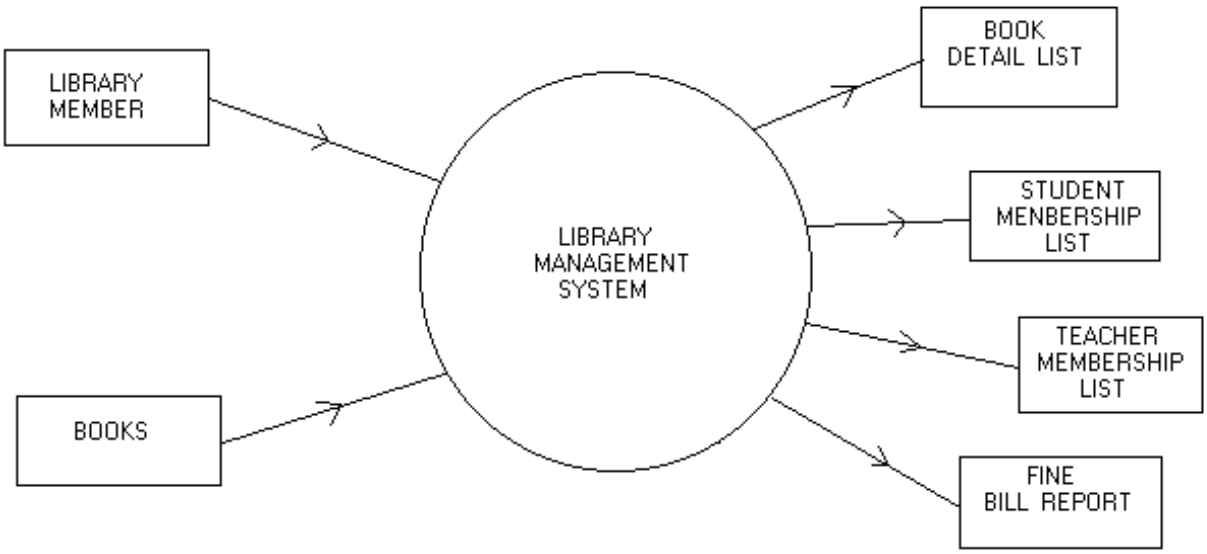
- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

  Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

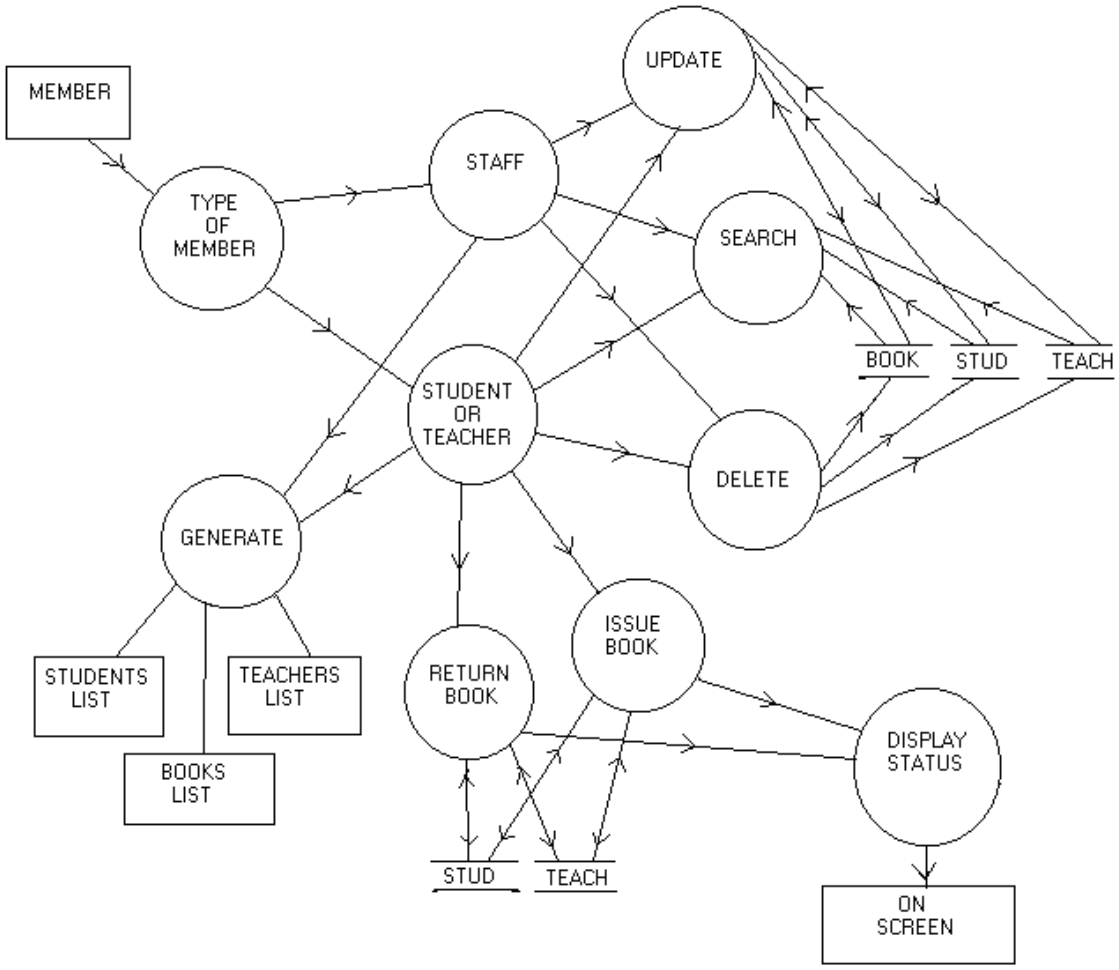
Case study

Data Flow Diagram

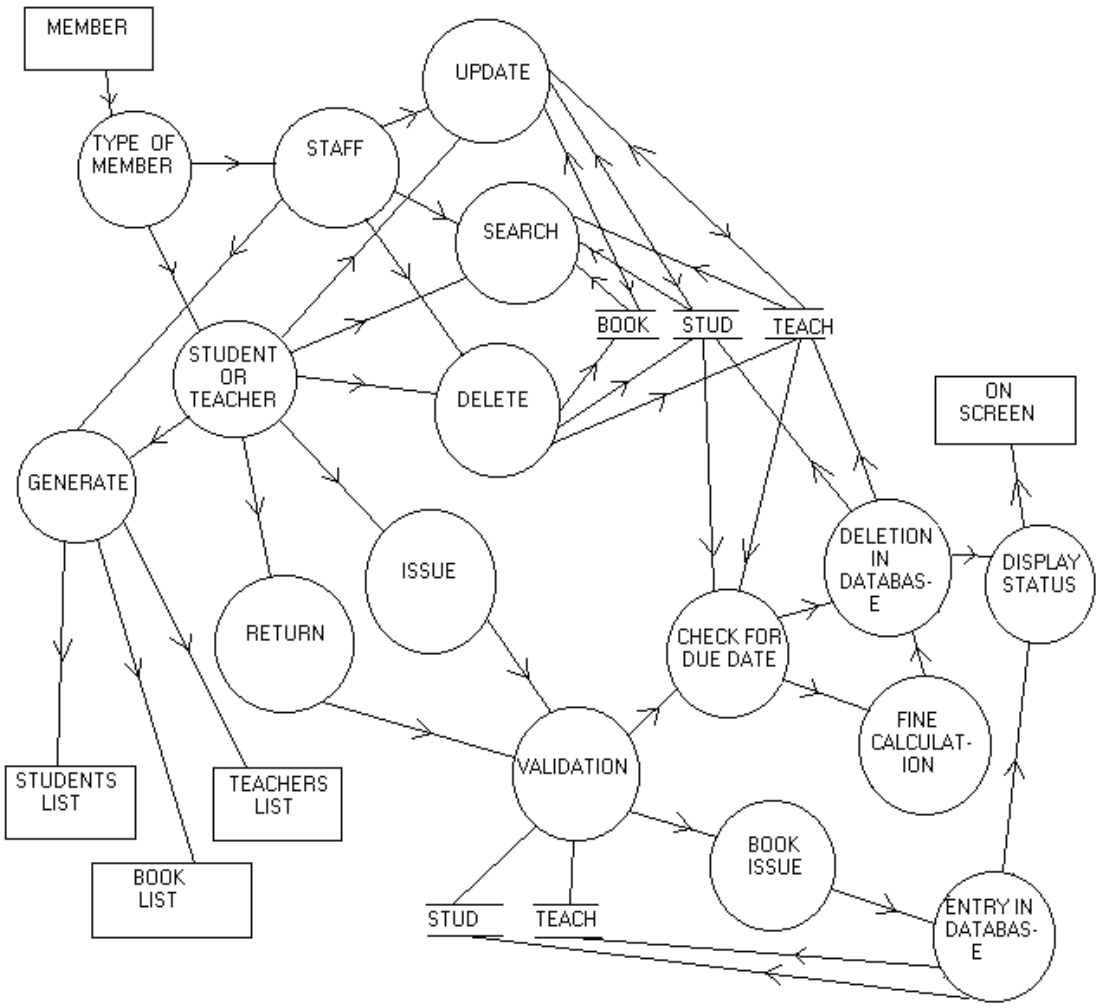
Level 0



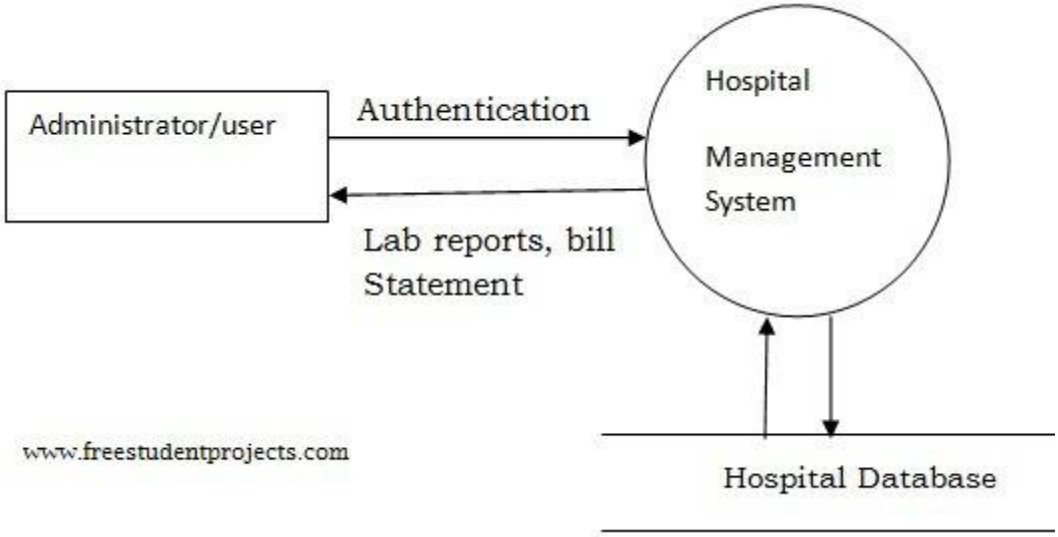
Level 1


MEMBER
TYPE OF MEMBER
STAFF
UPDATE
SEARCH
DELETE
STUDENT OR TEACHER
GENERATE
STUDENTS LIST
TEACHERS LIST
BOOKS LIST
RETURN BOOK
ISSUE BOOK
DISPLAY STATUS
ON SCREEN
BOOK
STUD
TEACH
STUD
TEACH

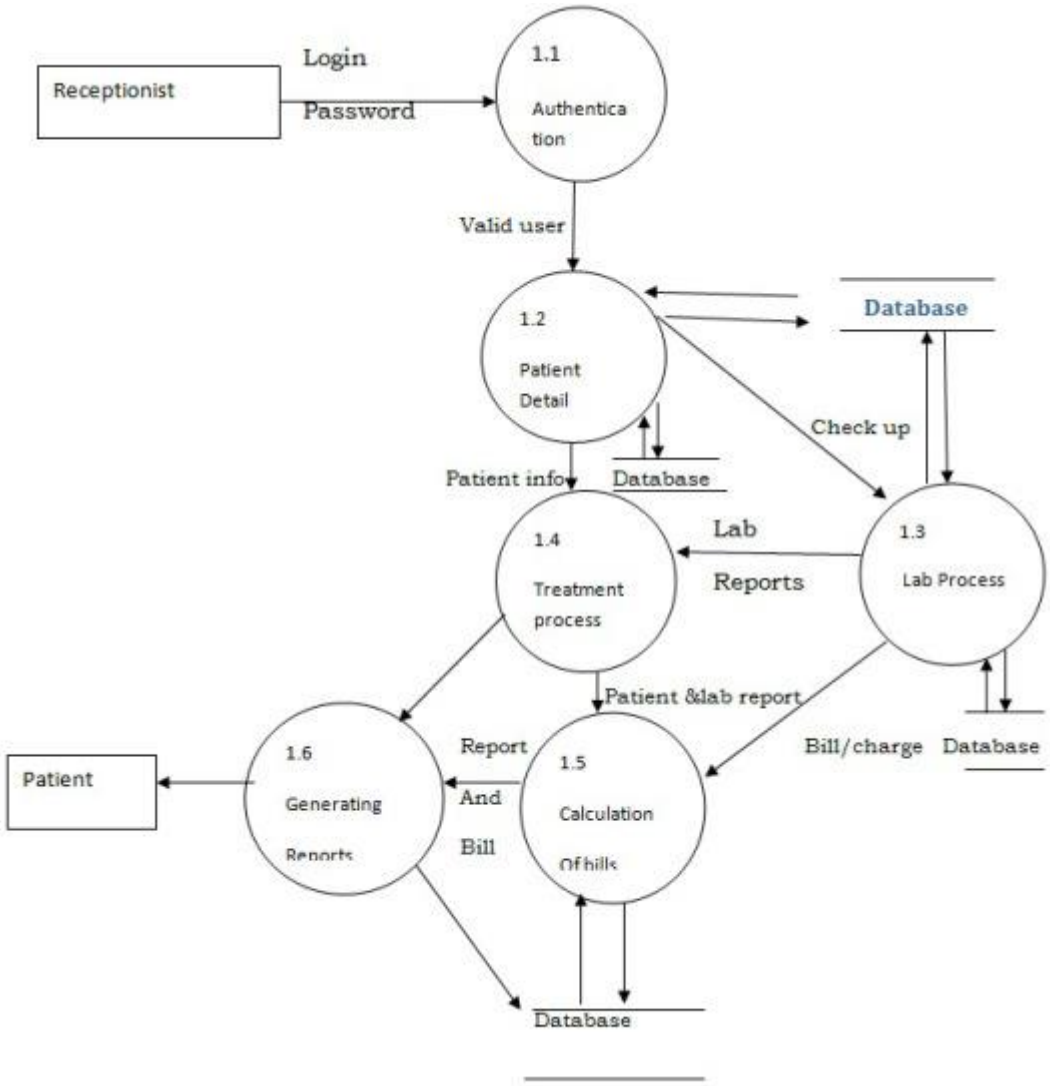
Level 2


MEMBER
TYPE OF MEMBER
STAFF
UPDATE
SEARCH
DELETE
BOOK
STUD
TEACH
STUDENT OR TEACHER
GENERATE
ISSUE
RETURN
ON SCREEN
DELETION IN DATABAS-E
DISPLAY STATUS
CHECK FOR DUE DATE
FINE CALCULAT-ION
VALIDATION
BOOK ISSUE
ENTRY IN DATABAS-E
STUDENTS LIST
TEACHERS LIST
BOOK LIST
STUD
TEACH

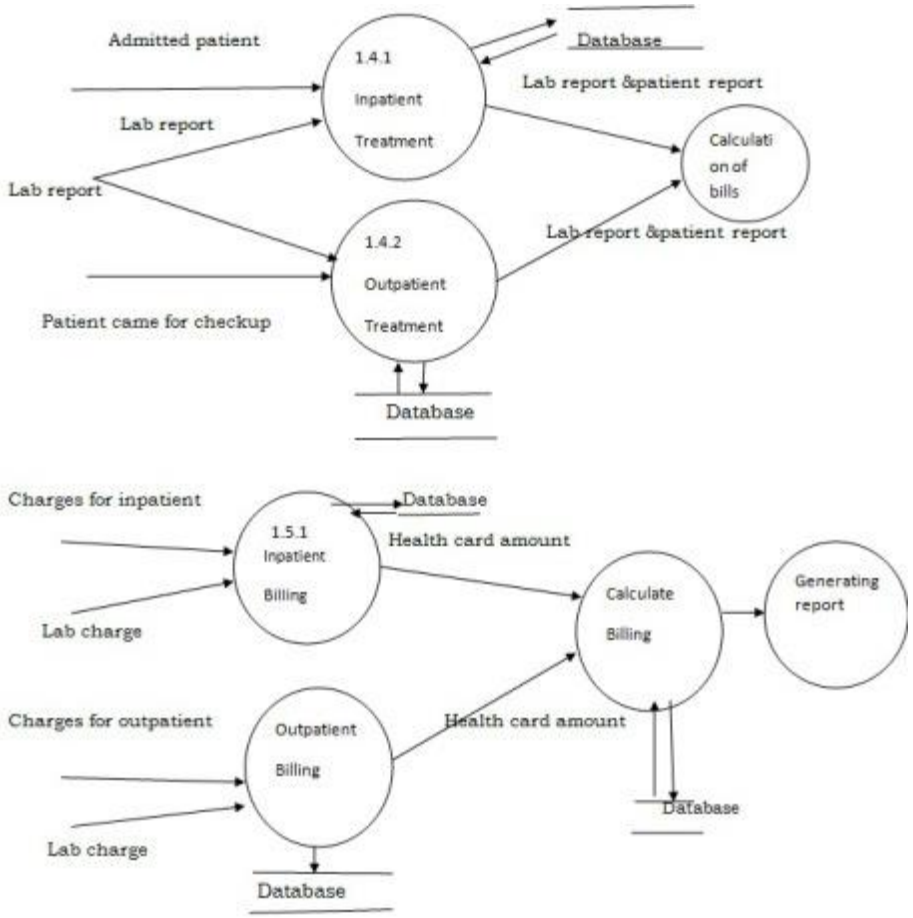
Hospital management system



Level-1


Receptionist
Login
Password
1.1
Authentica
tion
Valid user
1.2
Patient
Detail
Database
Check up
Patient info
Database
1.4
Treatment
process
Lab
Reports
1.3
Lab Process
Patient &lab report
Bill/charge
Database
Patient
1.6
Generating
Reports
Report
And
Bill
1.5
Calculation
Of bills
Database

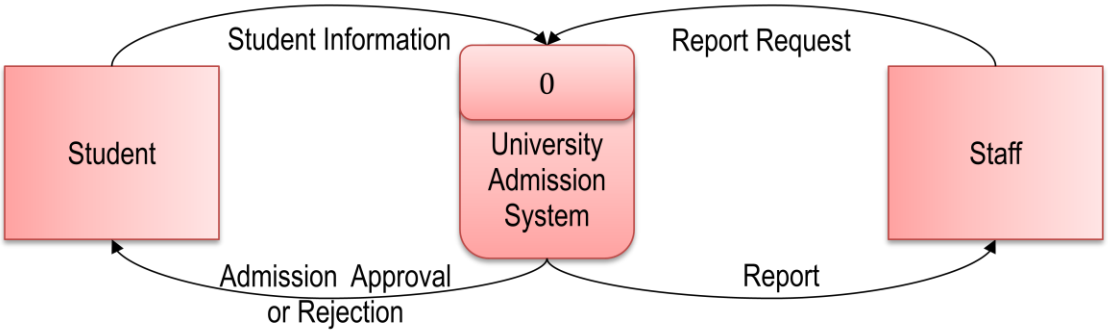
Level-2


Admitted patient
1.4.1
Inpatient
Treatment
Database
Lab report &patient report
Lab report
Calculati
on of
bills
Lab report
1.4.2
Outpatient
Treatment
Lab report &patient report
Patient came for checkup
Database
Charges for inpatient
Database
1.5.1
Inpatient
Billing
Health card amount
Calculate
Billing
Generating
report
Lab charge
Charges for outpatient
Outpatient
Billing
Health card amount
Database
Lab charge
Database

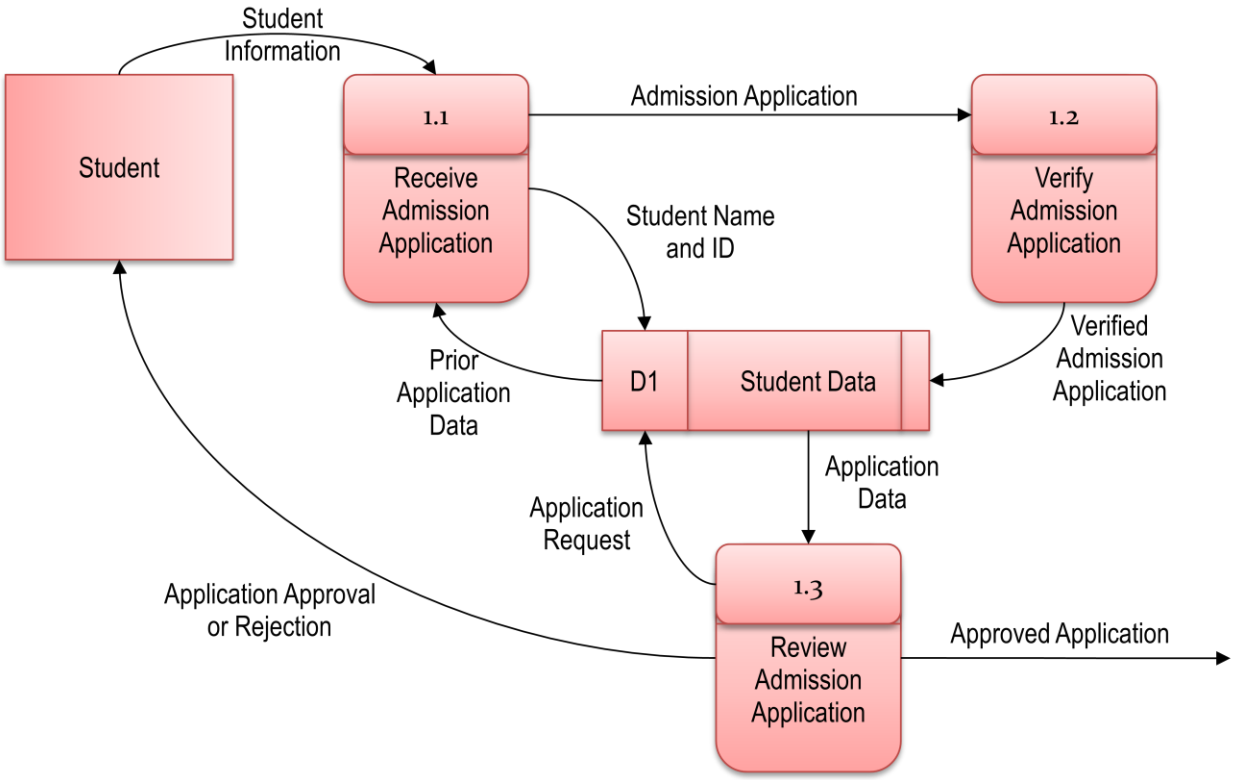
University admission system

DFD for University Admission System

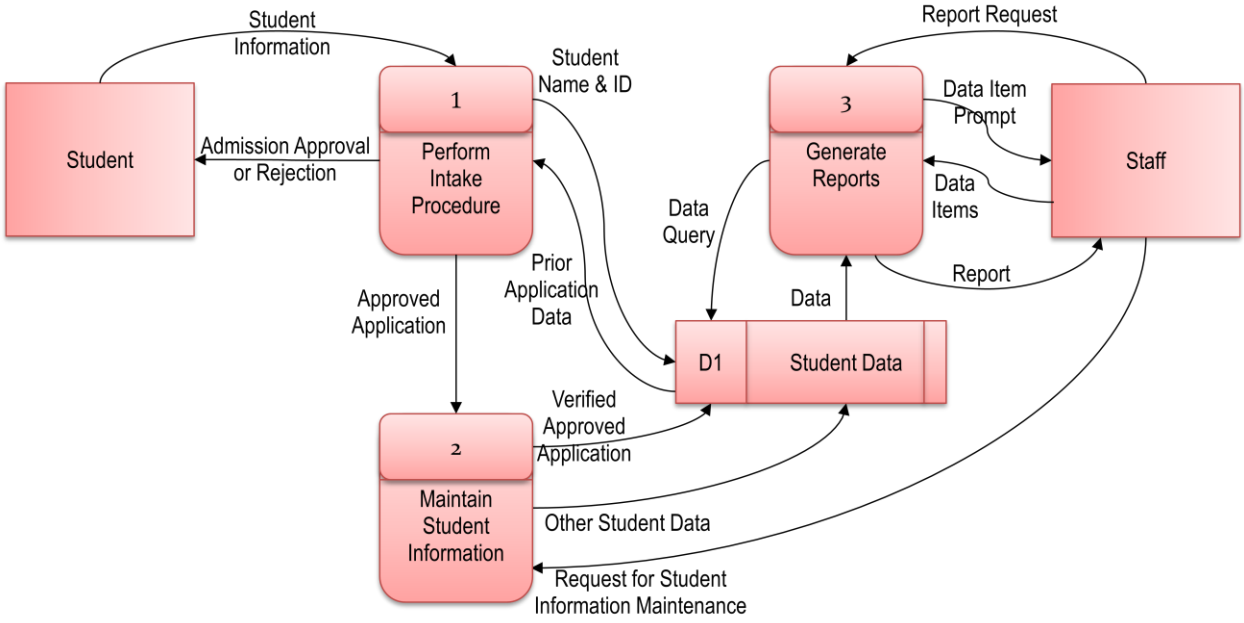
Context Diagram



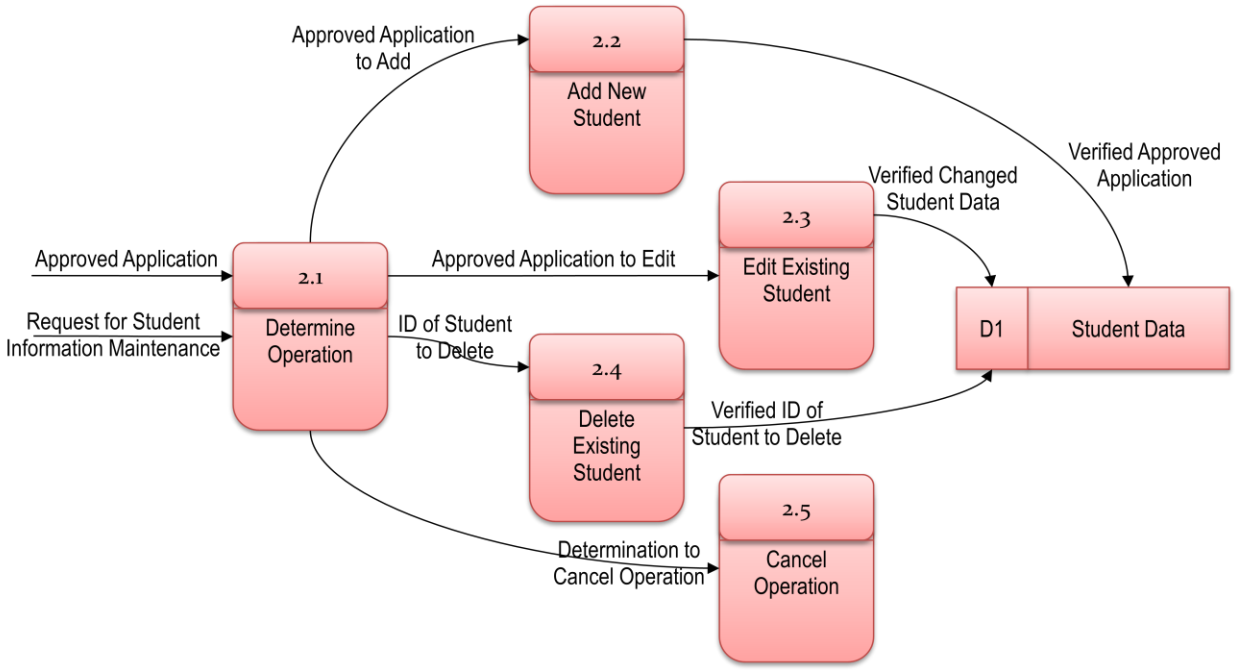
Level 1 Process 1, Perform Intake Procedure



Level 0

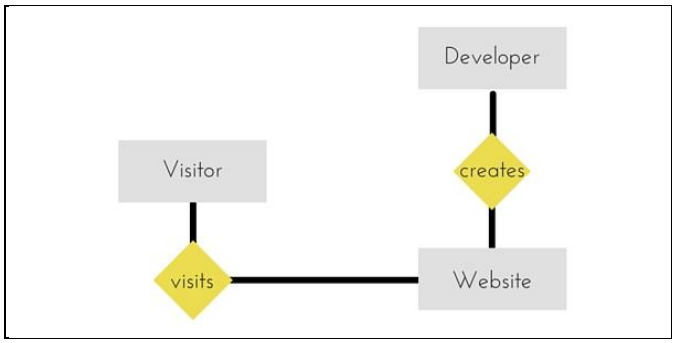

Student
Information
Student
Name & ID
Report Request
1
Perform
Intake
Procedure
3
Generate
Reports
Data Item
Prompt
Staff
Student
Admission Approval
or Rejection
Data
Query
Data
Items
Report
Approved
Application
Prior
Application
Data
Data
D1
Student Data
Verified
Approved
Application
2
Maintain
Student
Information
Other Student Data
Request for Student
Information Maintenance

Level 1 Process 2, Maintain Student Information

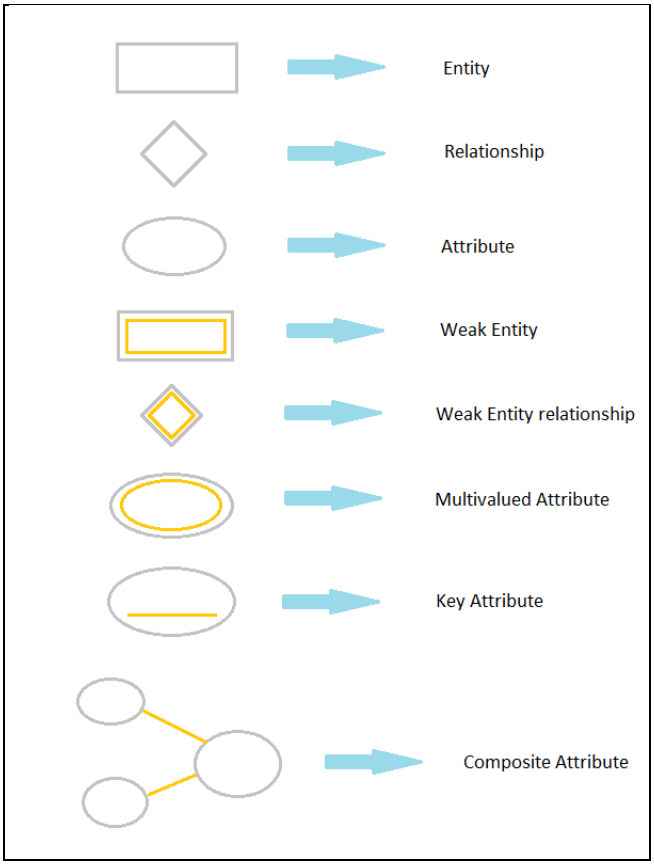


E-R Diagram

ER-Diagram is a visual representation of data that describes how data is related to each other.



Symbols and Notations



Components of E-R Diagram

The E-R diagram has three main components.

1) Entity

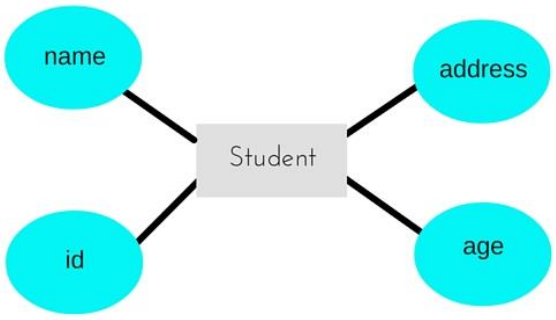
An **Entity** can be any object, place, person or class. In E-R Diagram, an **entity** is represented using rectangles. Consider an example of an Organization. Employee, Manager, Department, Product and many more can be taken as entities from an Organization.

Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doen't have key attribute of their own. Double rectangle represents weak entity.

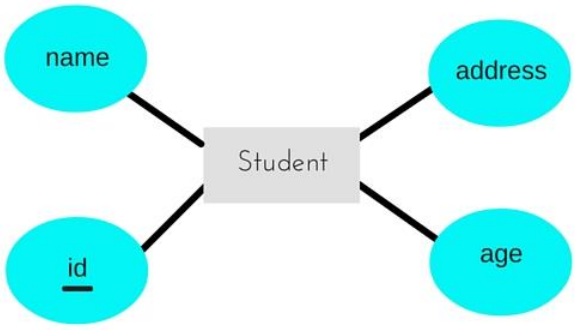
2) Attribute

An **Attribute** describes a property or characteristic of an entity. For example, Name, Age, Address etc can be attributes of a Student. An attribute is represented using eclipse.



Key Attribute

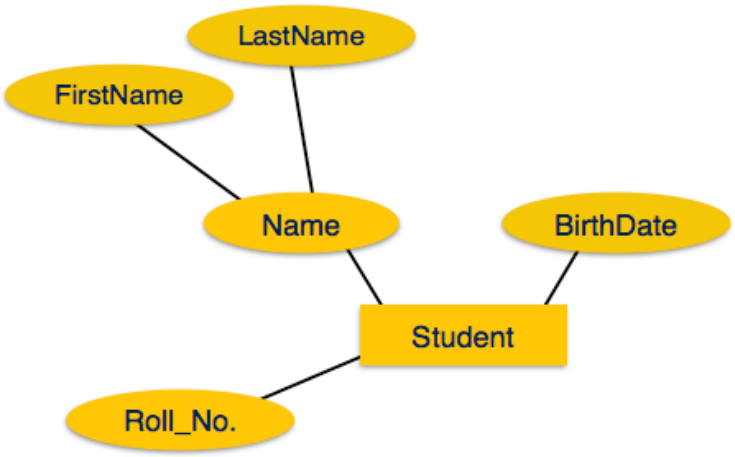
Key attribute represents the main characteristic of an Entity. It is used to represent Primary key. Ellipse with underlying lines represents Key Attribute.



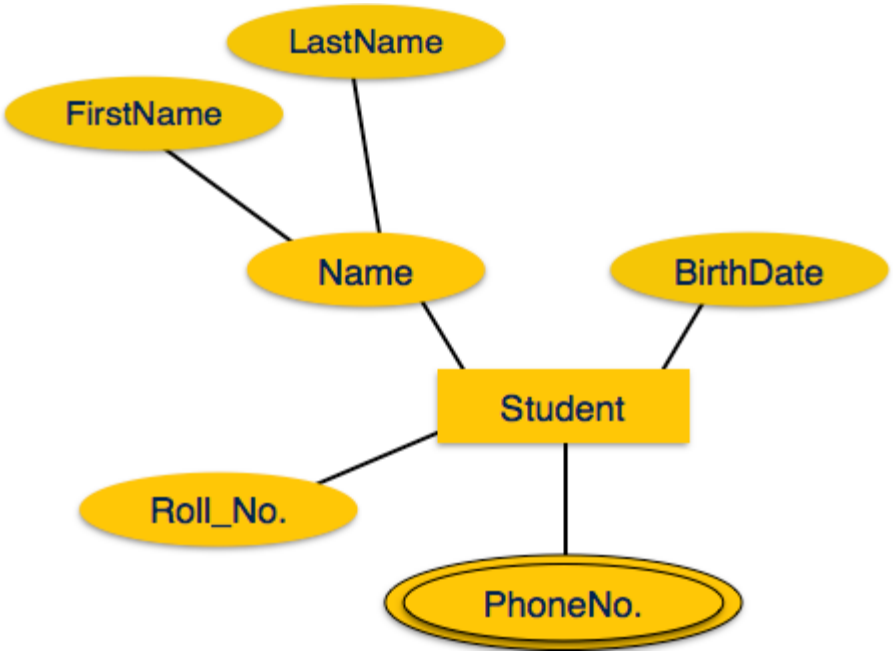
Composite Attribute

An attribute can also have their own attributes. These attributes are known as **Composite** attribute.

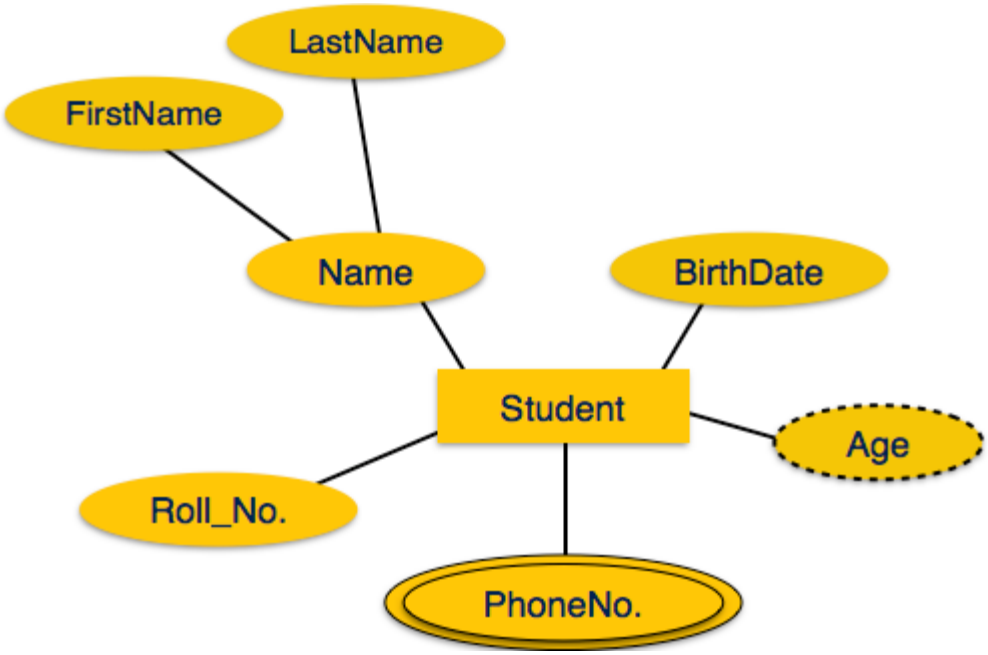
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



Multivalued attributes are depicted by double ellipse.



Derived attributes are depicted by dashed ellipse.



3) Relationship

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

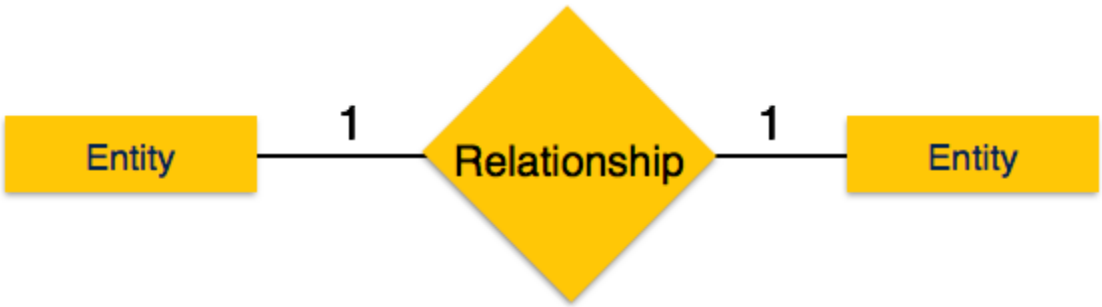
There are three types of relationship that exist between Entities.

Binary Relationship
Recursive Relationship
Ternary Relationship

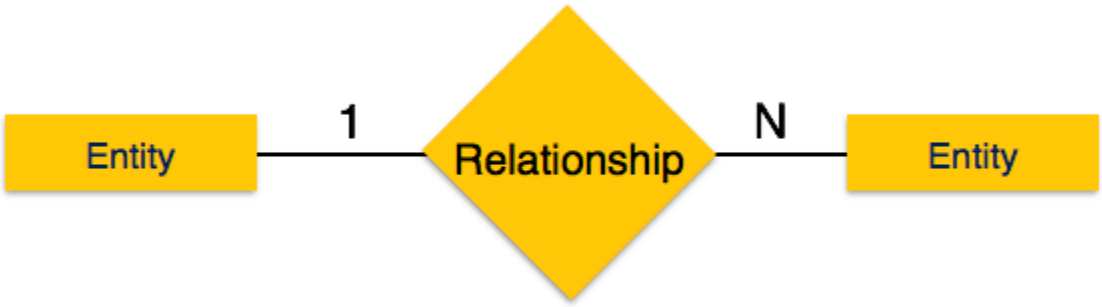
Binary Relationship and Cardinality

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

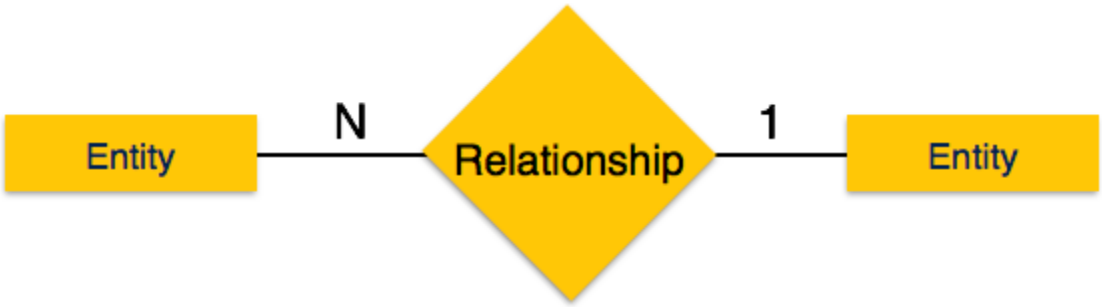
- **One-to-one** − When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



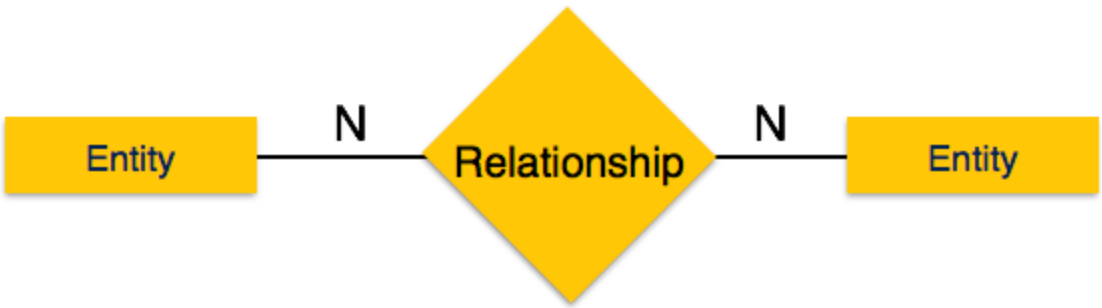
- **One-to-many** − When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.



- **Many-to-one** – When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.

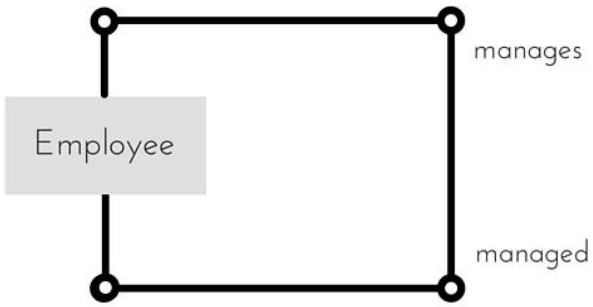


- **Many-to-many** – The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



Recursive Relationship

When an Entity is related with itself it is known as **Recursive** Relationship.

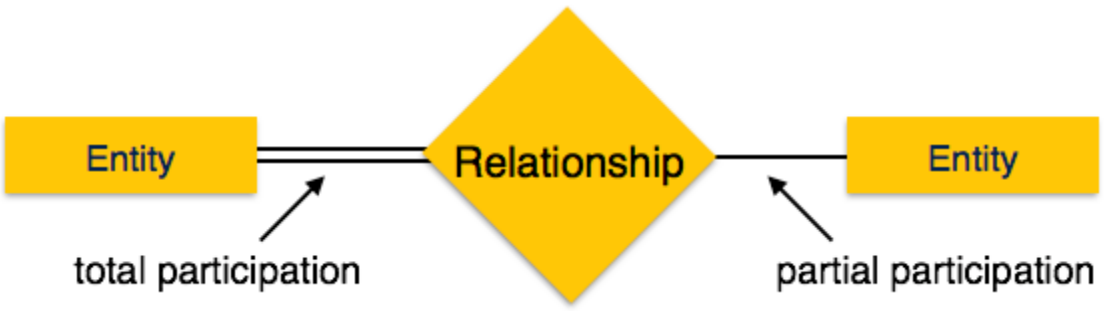


Ternary Relationship

Relationship of degree three is called Ternary relationship.

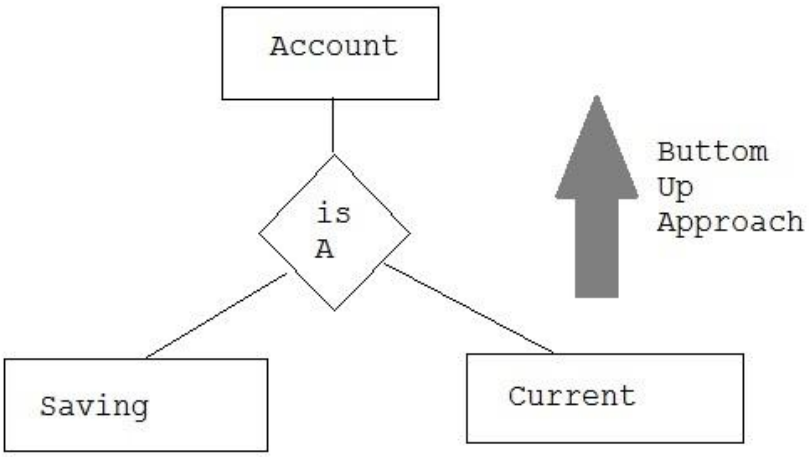
Participation Constraints

- **Total Participation** – Each entity is involved in the relationship. Total participation is represented by double lines.
- **Partial participation** – Not all entities are involved in the relationship. Partial participation is represented by single lines.



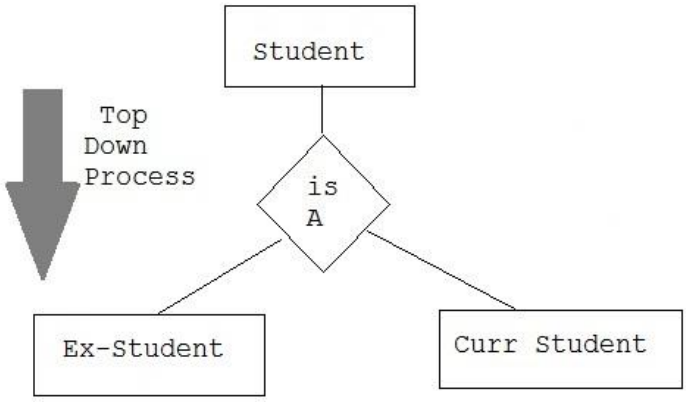
Generalization

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity.



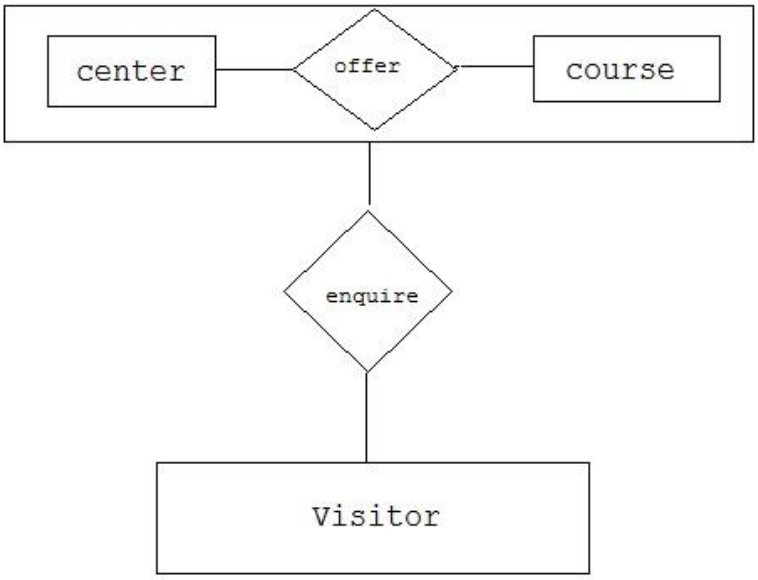
Specialization

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.

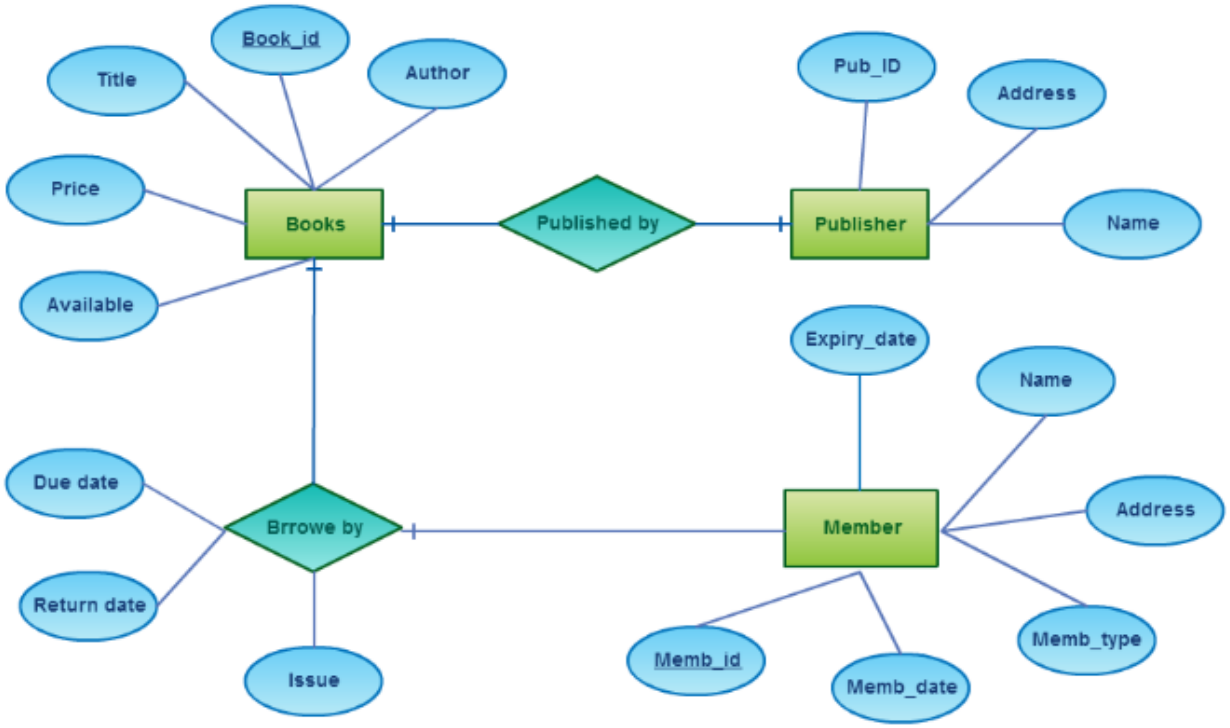


Aggregation

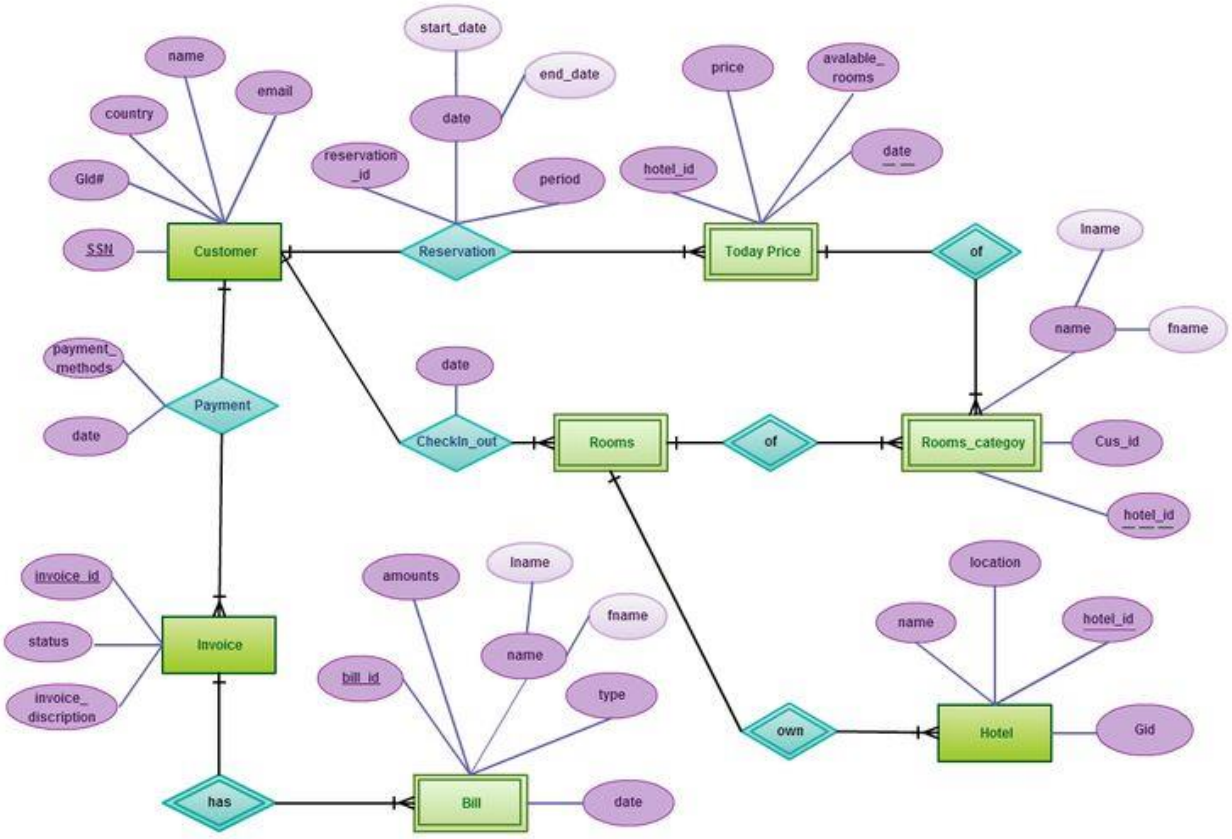
Aggregration is a process when relation between two entity is treated as a single entity. Here the relation between Center and Course, is acting as an Entity in relation with Visitor.



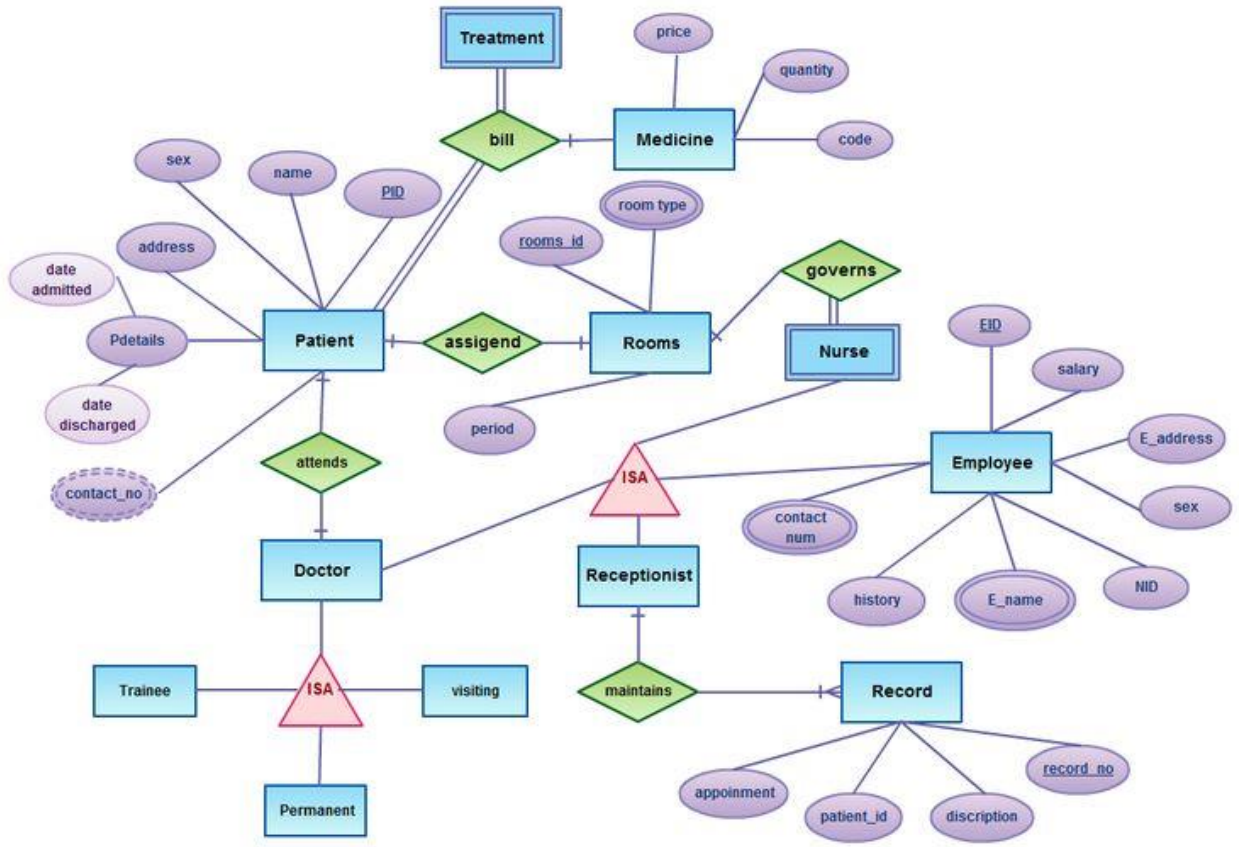
E-R Diagram of Library Management System



E-R Diagram for Hotel Management System



E-R Diagram for Hospital Management System



UML

UML stands for Unified Modeling Language which is used in object oriented software engineering. Although typically used in software engineering it is a rich language that can be used to model an application structures, behavior and even business processes. There are **14 UML diagram types** to help you model this behavior.

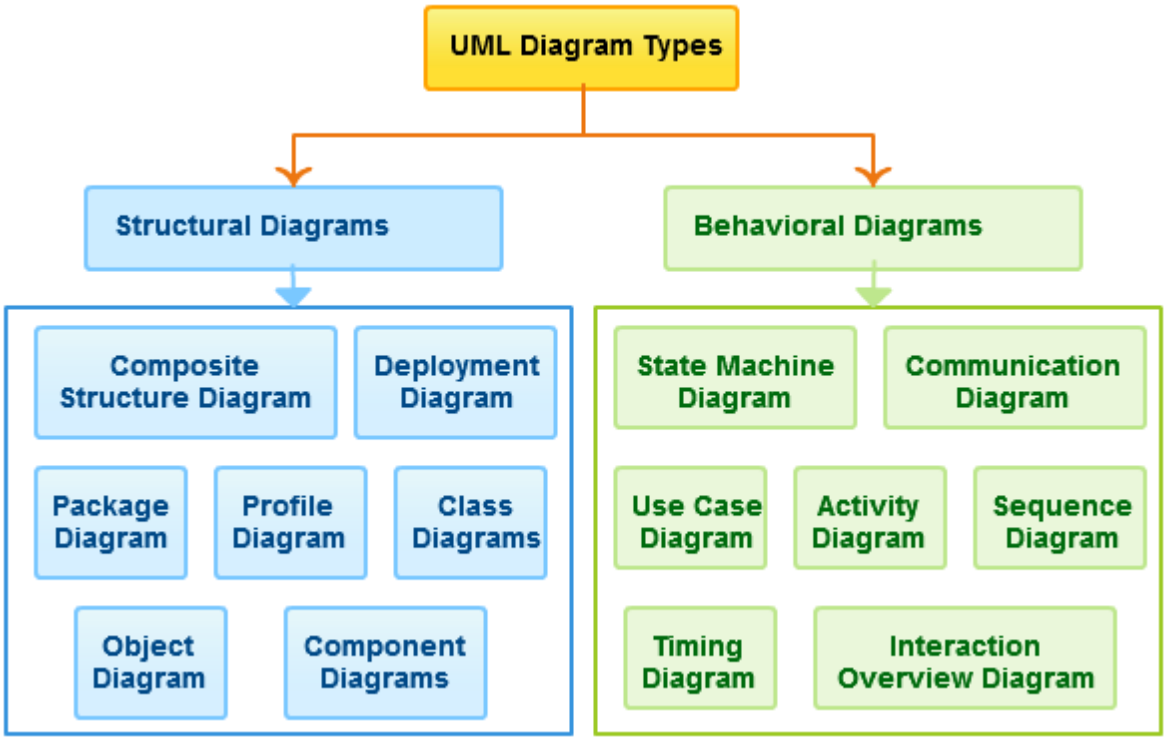
They can be divided into two main categories structure diagrams and behavioral diagrams. All 14 UML diagram types are listed below with examples, brief introduction to them and also how they are used when modeling applications.

List of UML Diagram Types

Types of UML diagrams with structure diagrams coming first and behavioral diagrams starting from position 8. Click on any diagram type to visit that specific diagram types description.

- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Profile Diagram
- Composite Structure Diagram
- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram

- Timing Diagram



Structure diagrams show the things in a system being modeled. In a more technical term they show different objects in a system. **Behavioral diagrams** shows what should happen in a system. They describe how the objects interact with each other to create a functioning system.

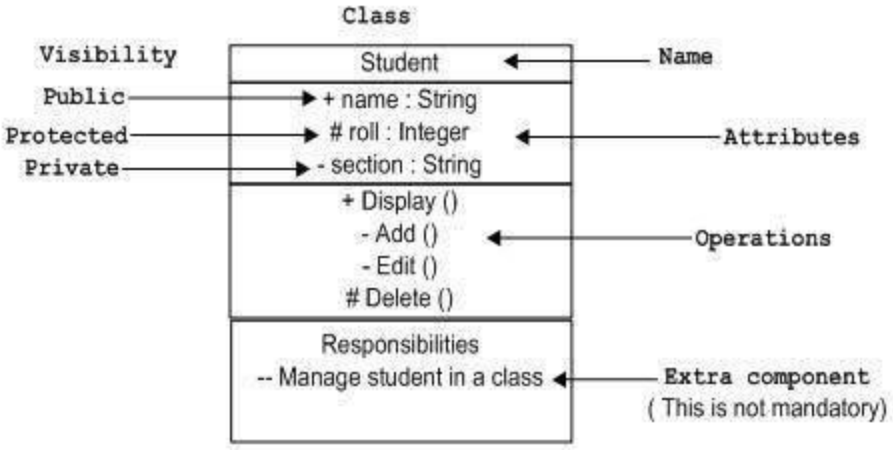
Class Diagram

Class diagrams are arguably the most used UML diagram type. It is the main building block of any object oriented solution. It shows the classes in a system, attributes and operations of each class and the relationship between each class.

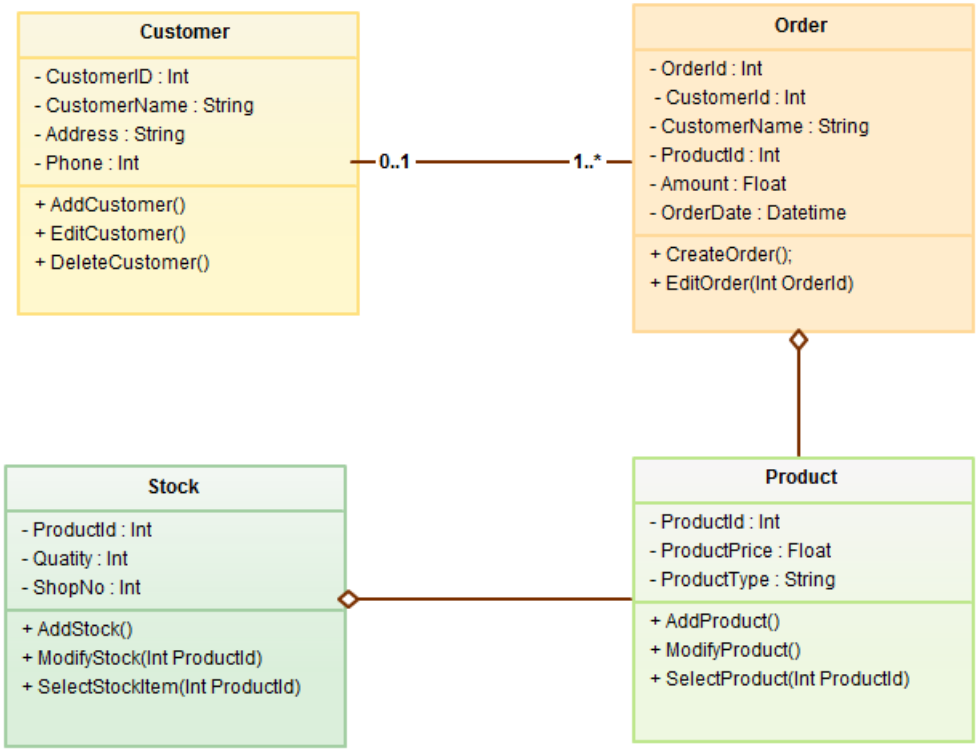
In most modeling tools a class has three parts, name at the top, attributes in the middle and operations or methods at the bottom. In large systems with many related classes, classes are

grouped together to create class diagrams. Different relationships between classes are shown by different types of arrows.

Notation:



Class Diagram for Order Processing System

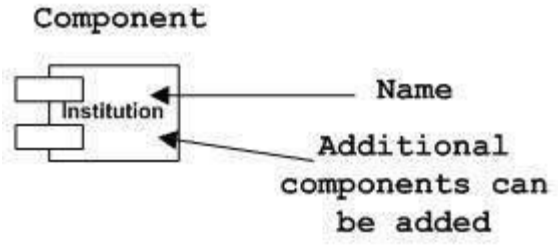


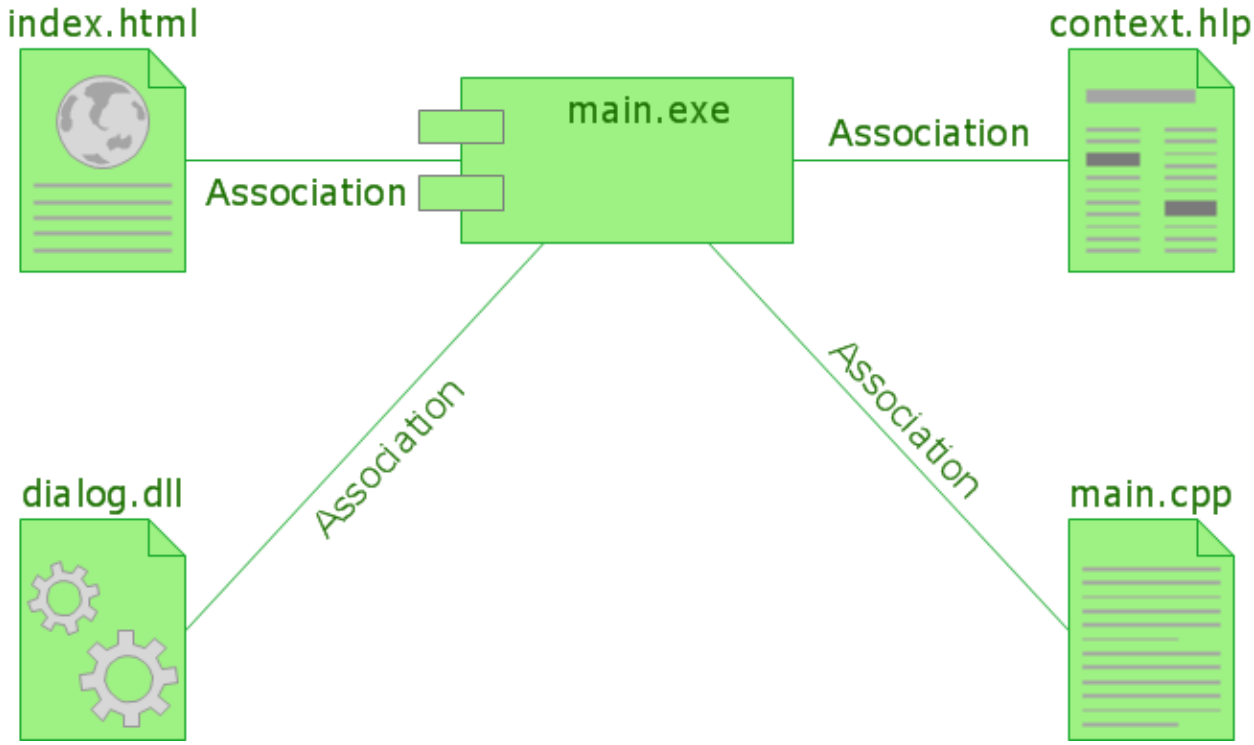
Component Diagram

A component in UML is shown as below with a name inside. Additional elements can be added wherever required.

Component is used to represent any part of a system for which UML diagrams are made.

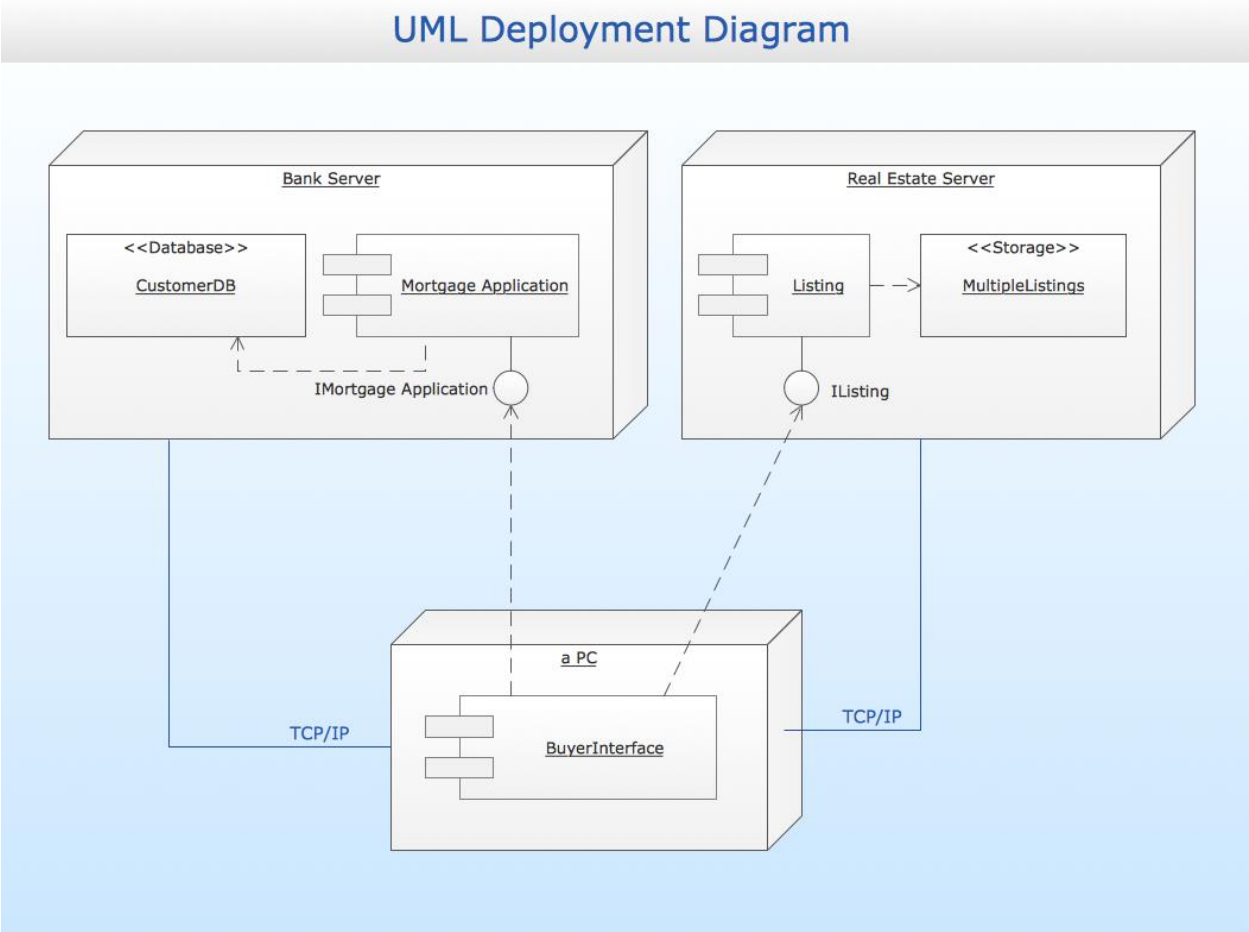
Notation:





Deployment Diagram

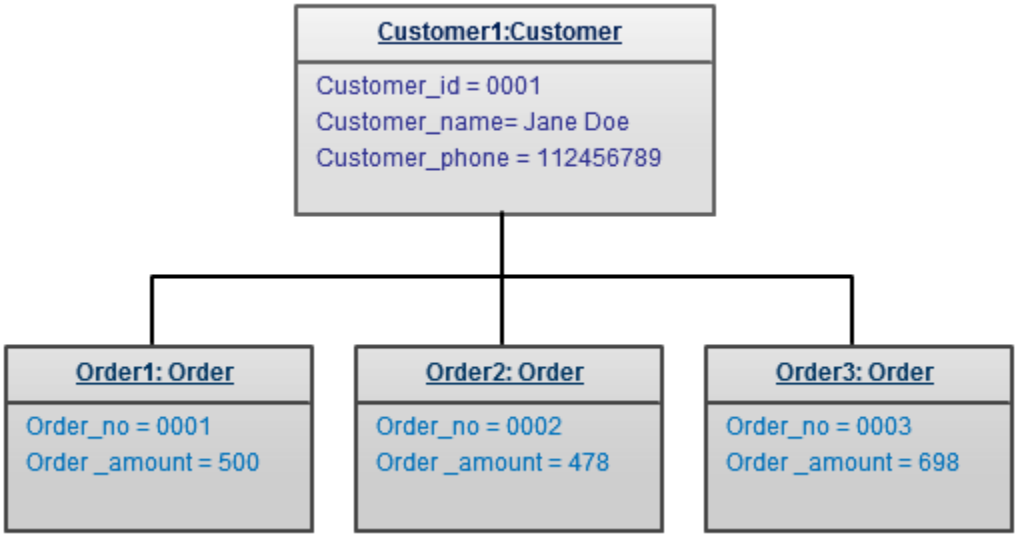
A deployment diagrams shows the hardware of your system and the software in those hardware. Deployment diagrams are useful when your software solution is deployed across multiple machines with each having a unique configuration. Below is an example deployment diagram.



Object Diagram

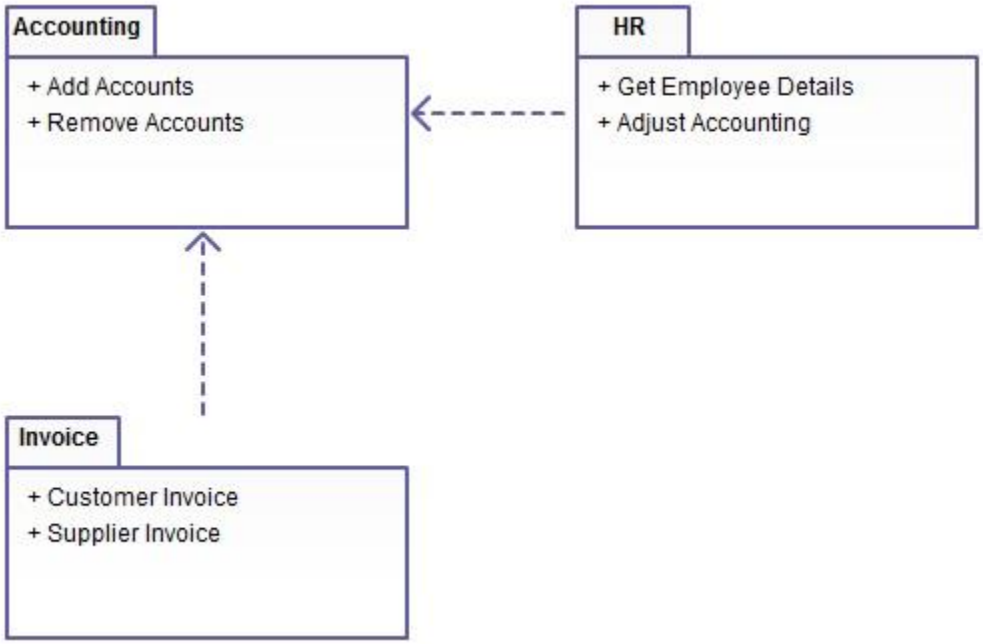
Object Diagrams, sometimes referred as Instance diagrams are very similar to class diagrams. As class diagrams they also show the relationship between objects but they use real world examples. They are used to show how a system will look like at a given time.

Because there is data available in the objects they are often used to explain complex relationships between objects.



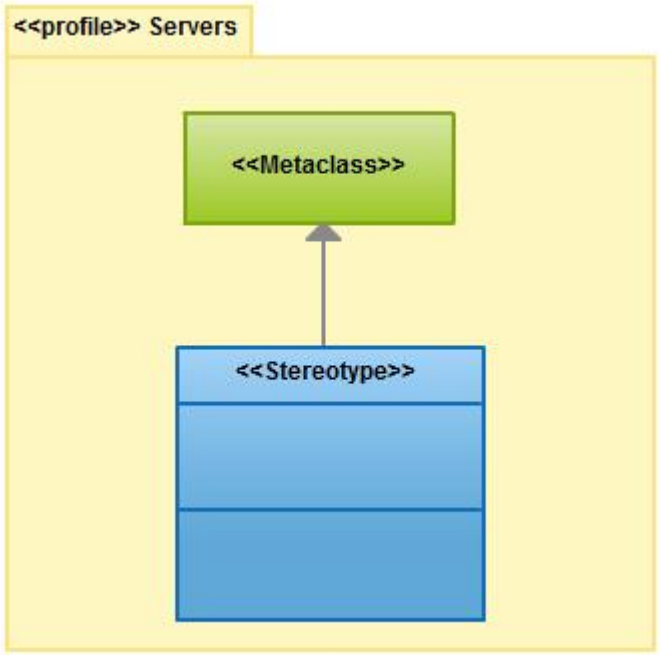
Package Diagram

As the name suggests a package diagrams shows the dependencies between different packages in a system.



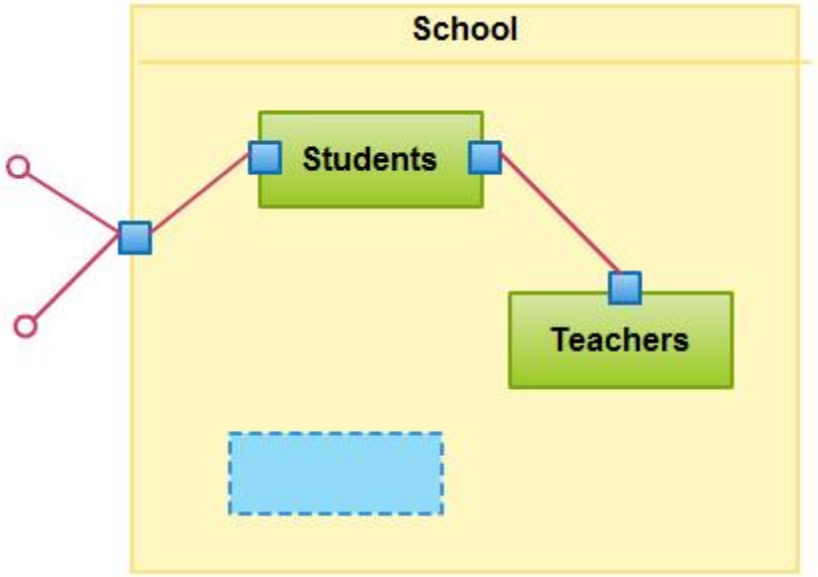
Profile Diagram

Profile diagram is a new diagram type introduced in UML 2. This is a diagram type that is very rarely used in any specification. For more detailed technical information about this diagram type check this link.



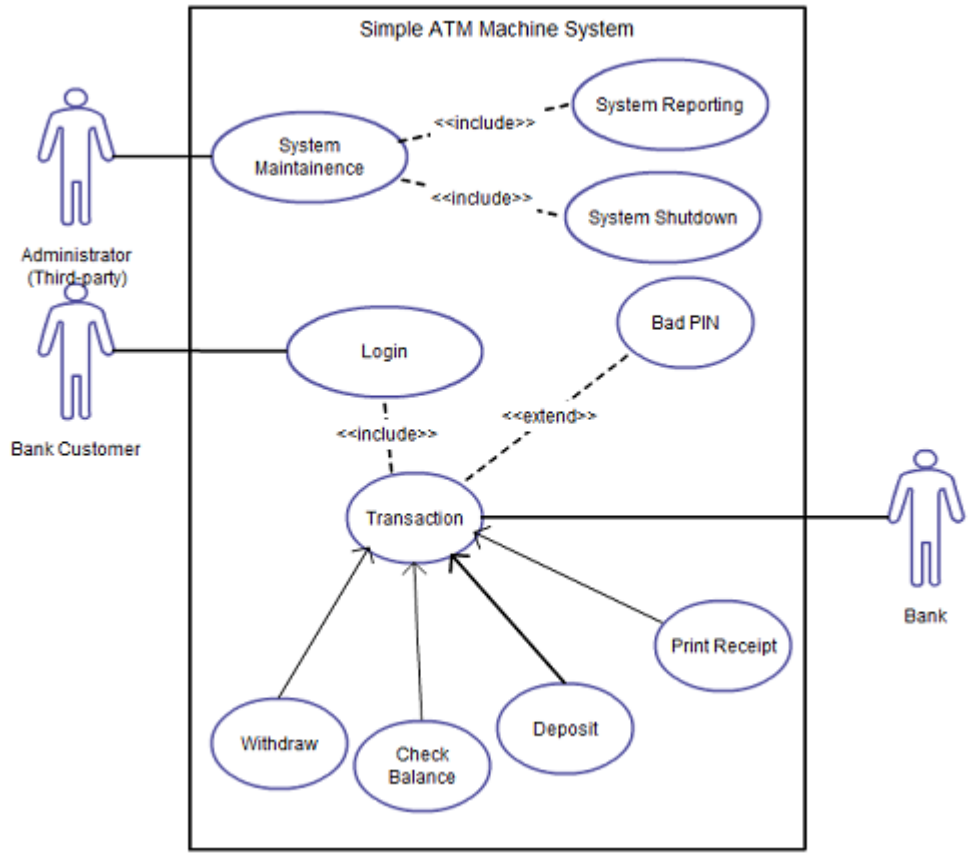
Composite Structure Diagram

Composite structure diagrams are used to show the internal structure of a class.



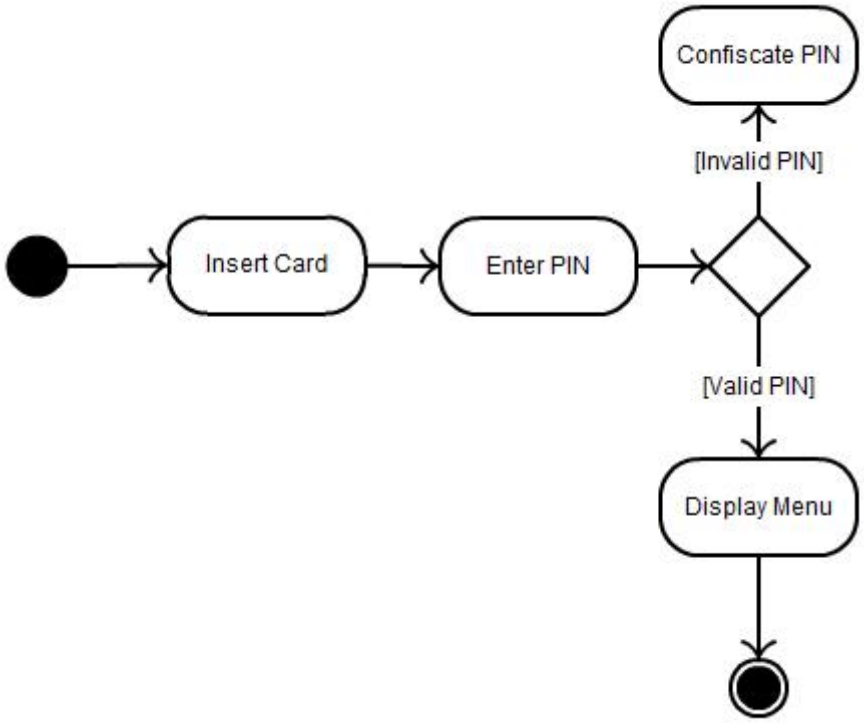
Use Case Diagram

Most known diagram type of the behavioral UML diagrams, Use case diagrams gives a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions are interacted.



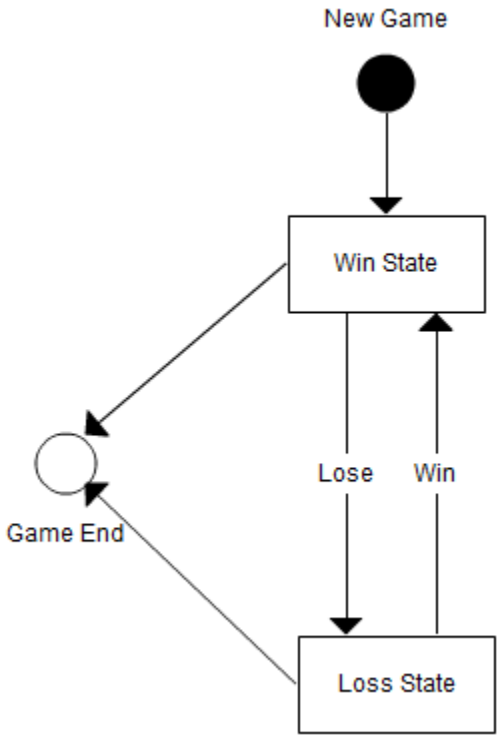
Activity Diagram

Activity diagrams represent workflows in an graphical way. They can be used to describe business workflow or the operational workflow of any component in a system. Sometimes activity diagrams are used as an alternative to State machine diagrams.



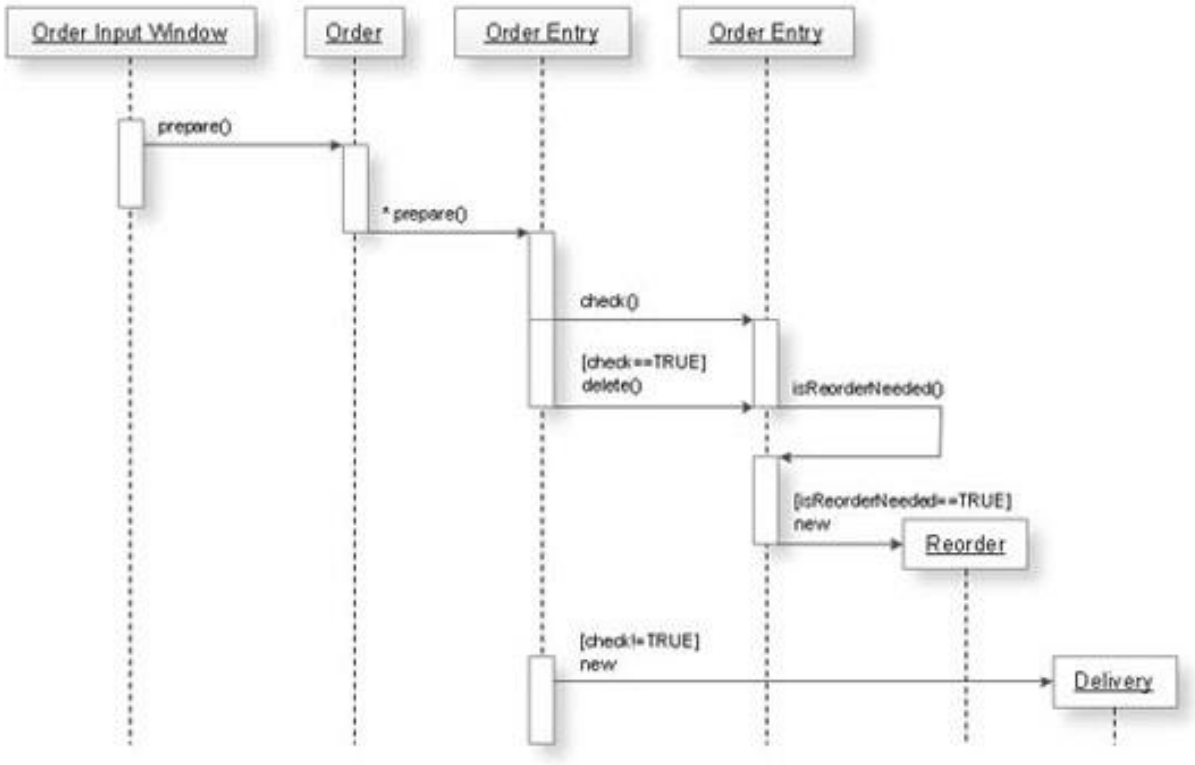
State Machine Diagram

State machine diagrams are similar to activity diagrams although notations and usage changes a bit. They are sometime known as state diagrams or start chart diagrams as well. These are very useful to describe the behavior of objects that act different according to the state they are at the moment. Below State machine diagram show the basic states and actions.



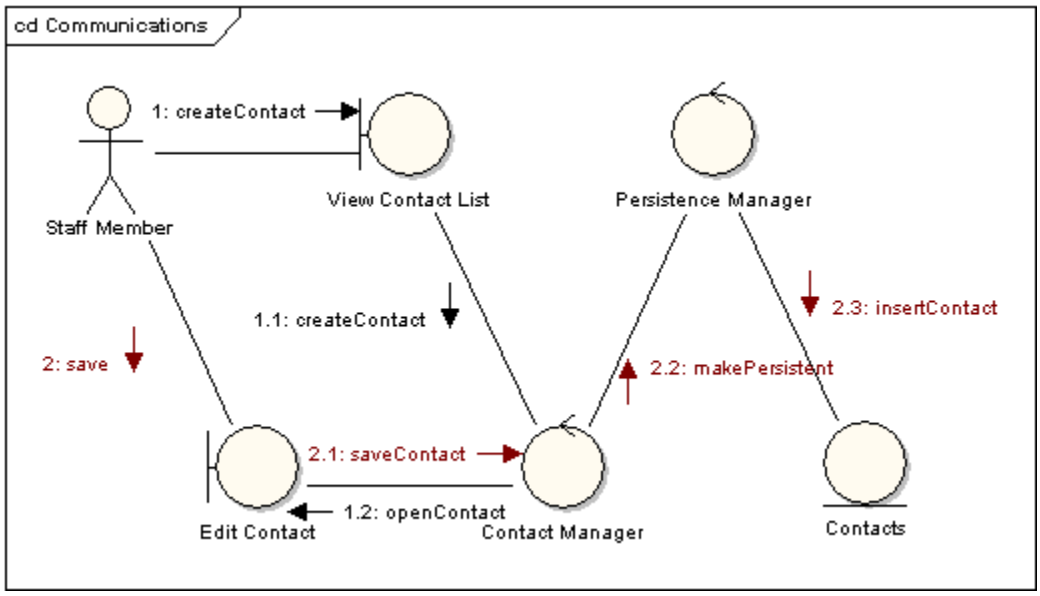
Sequence Diagram

Sequence diagrams in UML shows how object interact with each other and the order those interactions occur. It's important to note that they show the interactions for a particular scenario. The processes are represented vertically and interactions are show as arrows.



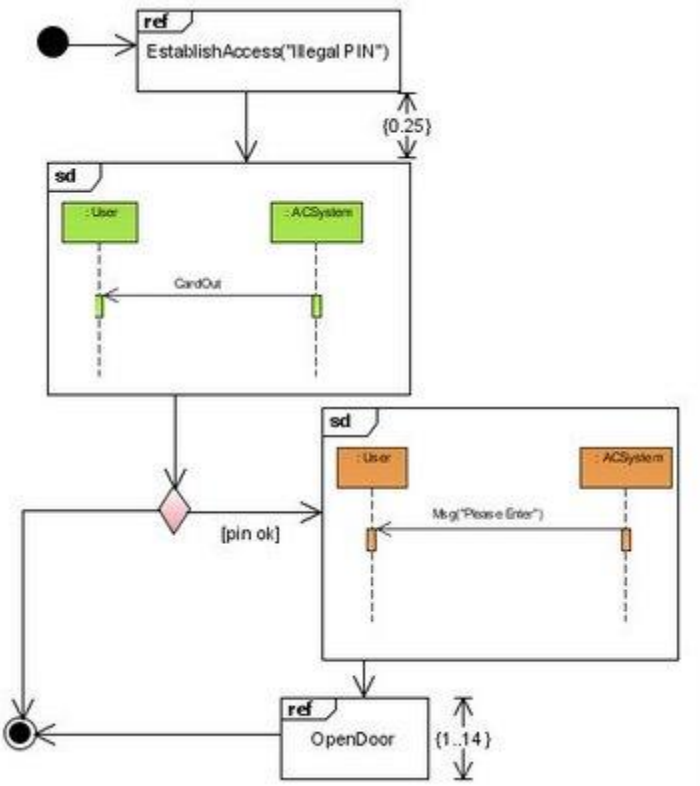
Communication Diagram

Communication diagram was called collaboration diagram in UML 1. It is similar to sequence diagrams but the focus is on messages passed between objects. The same information can be represented using a sequence diagram and different objects



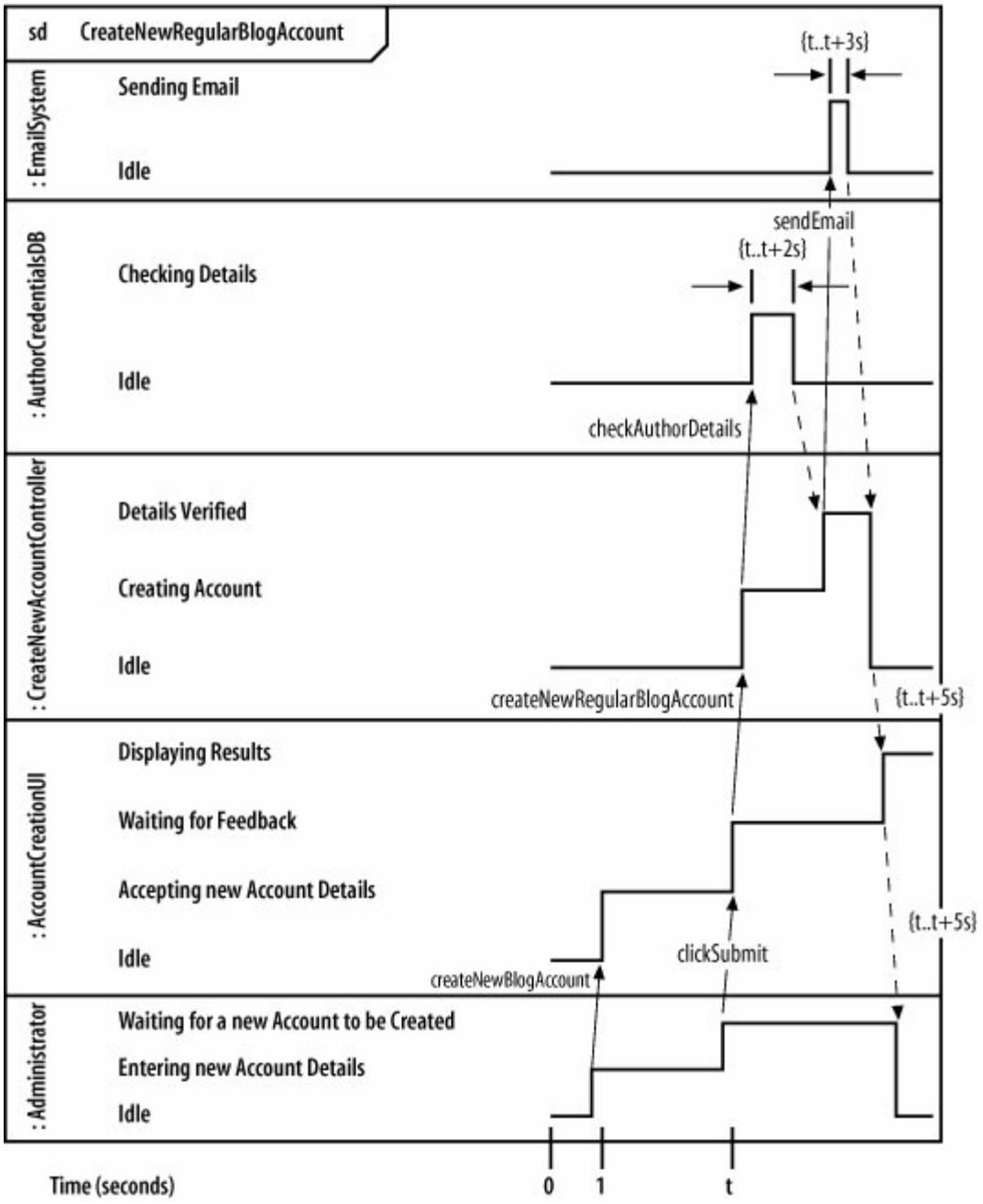
Interaction Overview Diagram

Interaction overview diagrams are very similar to activity diagrams. While activity diagrams shows a sequence of processes Interaction overview diagrams shows a sequence of interaction diagrams. In simple term they can be called a collection of interaction diagrams and the order they happen. As mentioned before there are seven types of interaction diagrams, so any one of them can be a node in an interaction overview diagram.



Timing Diagram

Timing diagrams are very similar to sequence diagrams. They represent the behavior of objects in a given time frame. If its only one object the diagram is straight forward but if more then one objects are involved they can be used to show interactions of objects during that time frame as well


sd CreateNewRegularBlogAccount
: EmailSystem
Sending Email
Idle
{t..t+3s}
: AuthorCredentialsDB
Checking Details
Idle
sendEmail
{t..t+2s}
checkAuthorDetails
: CreateNewAccountController
Details Verified
Creating Account
Idle
createNewRegularBlogAccount
{t..t+5s}
: AccountCreationUI
Displaying Results
Waiting for Feedback
Accepting new Account Details
Idle
clickSubmit
{t..t+5s}
createNewBlogAccount
: Administrator
Waiting for a new Account to be Created
Entering new Account Details
Idle
Time (seconds)
0
1
t

Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how changes data and state of entire system functions.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.

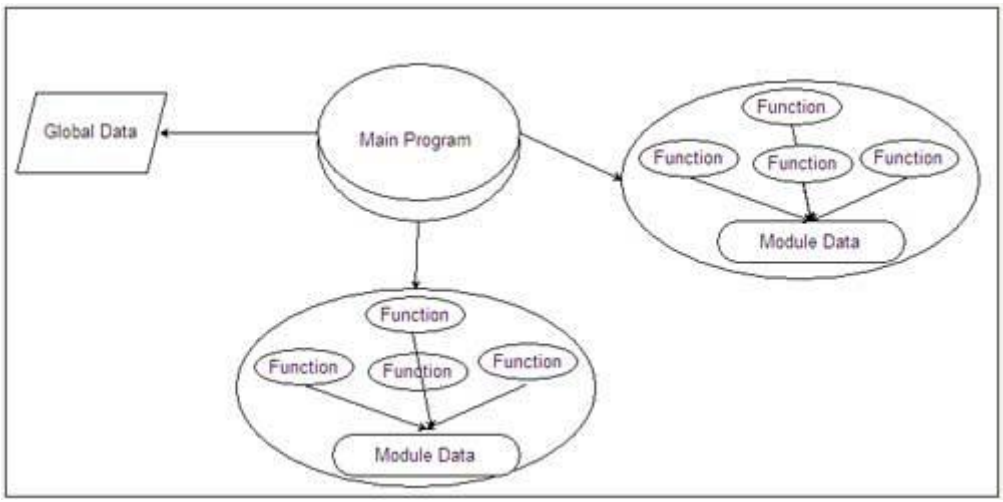
- Class hierarchy and relation among them is defined.
- Application framework is defined.

Function oriented approach vs. Object oriented Approach

Topic	Function oriented approach	Object oriented Approach
decompose	we decompose in function/procedure level	we decompose in class level
approach	Top down Approach	Bottom up approach
diagram	Begins by considering the use case diagrams and Scenarios.	Begins by identifying objects and classes.
abstraction	The basic abstraction is available to users of the system	The Basic Abstraction is not the services available to the users of the system
Data storage	The state information is available in a centralized shared data store.	The state information exists in the form of data distributed among several objects of the system
Design	In the functional oriented design approach, the basic abstraction, which are given to the user, are real world function, such as sort, merge, track, display etc.	In the object oriented design approach the basic abstraction are not the real world functions, but are the data abstraction where the real world entities are represents such as picture ,machine, radar system, customer,student,employee etc.

Module Level Concepts

Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.



Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conducts maintenance work without adversely affecting the functioning of the software.