SOFTWARE ENGINEERING

- Ctrl + F (to find word in chrome)

- **What is software or software product ?**

Software products are nothing but software systems delievered to customer with the documentation that describes how to install and use the system. In certain cases, software products may be part of system products where hardware as well as software is delievered to a customer.

Nature of software

1) Software is intangible
2) Easy to reproduce.
3) industry is labor-intensive
4) Untrained people can hack something together
5) Software is easy to modify.
6) Software does not 'wear out'

Types of Software

1) Custom
   - for a specific customer
   - web sites, air-traffic control systems

2) Generic
   - Sold on open market.
   - Often called
     - COTS (commercial off the shelf), • shrink-wrapped
   - eg. wordpress, browsers, compilers, games, OS etc.

3) Embedded.
- built into hardware
- hard to change.

ON Basis of processing

1) Real time software
- Eg. control & monitoring systems
- Must react immediatly
- Safety often a concern

2) Data processing software
- Used to run bussinesses
- performs functions such as recording sales, managing accounts, printing bills etc.
- Accuracy & security of data are key.
  - Some software has both aspects.

Software engineering :- Process of solving customer's problems by the systematic development & evolution of large, high-quality software systems within cost, time and other constraints

- **Difficulties & Risk in Software engineering** :-
1) complexity & large no. of details
2) Uncertainity about technology
3) Constant change
4) Uncertainity about requirements
5) peteriosration of software design
6) uncertainity about software engineering skills.

Difference between Methodology & a Process :-

A process includes all the activities beginning from product inception through delivery and retirement. In addition, it resolves issues such as ordering of the activities, reuse, documentation, testing, parallel work performed by team members and coordination with the customer etc.

Whereas, a methodology includes only a single or at best few individual activities involved in development. eg testing methodology, design methodology etc.

• Stakeholders in Software Engineering :-

1) Users — Those who use the software
2) Customers (clients) - Those who pay for software
3) Software developers - develop & maintain the software
4) development managers - people who run organization that is developing the software.

• Software Quality -

1) Usability - Users can learn it fast & get their job done easily
2) Efficiency - Doesn't waste resources such as CPU time & memory.
3) Reliability - It does what it is required to do without failing
4) Maintainability - It can be easily changed
5) Reusability - It's parts can be used in other projects, so reprogramming is not needed.

• Internal Quality Criteria :-

These:
- Characterize aspects of design of the software
- Have an effect on external quality attributes

eg:
- amount of commenting of code
- complexity of code

• Software Engineering Projects

Three major categories:
- modifying an existing system
- Starting to develop system from scratch
- building most of new system from existing components, while developing new software only for missing details

→ Most projects are evolutionary or maintenance projects, involving work on legacy systems
- Corrective projects :- fixing defects
- Adaptive projects :- changing the system in response to changes in
  - Operating System
  - Database
  - Rules & regulations
- Enhancement projects :- adding new features for users
- Reengineering or Perfective projects : changing the system internally so it is more maintainable.

- 'Green field' projects

- New development
- minority of projects
- not constrained by design decisions & errors made by predecessors.

- Projects that involve building on a framework or a set of existing components.

- A framework is an application that is missing some important details
  eg - Specific rules of this organization

- Such projects :
  - Involve plugging together components that are
    - Already developed
    - Provide significant functionality
  - Benefit from reusing reliable software
  -

⇒ Activities Common to software Projects

- Requirements & specification

Includes → Domain analysis
→ Defining the problem
→ Requirements gathering
→ Requirements analysis
→ Requirements specification

- Design
  - Deciding how requirements should be implemented using available technology

Includes -

- Software engineering : Deciding what should be in hardware & what in software
- Software architecture : Deviding system into subsystems & deciding how subsystems will interact
- User interface design
- Design of database
- Detailed design of internals of a subsystem

- Modeling
  - Creating representations of domain or software
    - Use case modeling
    - Structural modeling
    - Dynamic & behavioral modelling
- Programming
- Quality assurance
  - Reviews, inspection, testing
- Deployment
  - Distributing & installing software & any other components of system like databases, special hardware etc.
- Managing the process
  - Planning
  - Estimating cost of system.

Reasons for software failures

- No planning of development work
- Deliverables to user not identified
- Poor understanding of user requirements
- No control or review
- Technical incompetence of developers
- Poor understanding of cost & effort by both developer & user.

SDLC (Software Development Life cycle)

Software Process

- Process defines a set of steps, these steps need to be carried out in particular order.
- Different types of processes in software domain
  - process of software development
  - Process of managing the project
  - Process for change & configuration managment
  - Process for managing the above process

Software Development Process

- SDLC (software development life cycle) is process which helps to develop good quality software products.
- SDLC is composed of number of clearly defined & distinct work steps or phases.
- A number of SDLC models or process models have been created such as waterfall, spiral etc.

Steps :-

- Problem Defination
- feasibility study
- Requirements analysis
- Design
- Implementation
- Testing
- Maintenance
- Archival

1) Problem Defination

- Ensure there exist a problem to be solved
- Define goal
- Avoid misunderstanding
- Check cost-effectiveness
- Categorizing problem
- Identify cause of problem

→ **Problem defination document**

- Problem statement
- Project objective
- Preliminary idea
- Time & cost for feasibility study.

2) feasibility study

- Understanding of problem & reasons.

types of feasibility study

- Economical
- Technical
- Operational

→ **feasibility Report**

- Brief statement of problem, system environment
- Important finding & recommendations
- Alternatives
- System description
- Cost-benefit analysis
- Evaluation of technical risk
- Legal consequences.

3) # Requirements Analysis

- Knowing user's requirement in detail
- Objective is to determine what the system must do to solve the problem (without describing how)
- Produce SRS document
- Incorrect, incomplete; inconsistent, ambiguous SRS often results in project failure.

→ **Challenges**

- Users may not know exactly what is needed
- Users may change their mind over time
- Users may have conflicting demands
- Users may not be capable of differentiating betn what is possible & what is impractical
- Analyst has no or limited domain knowledge
- Client may be different from user.

SRS

It identifies all functional & performance requirements.

Requirements Analysis Process

1) Interviewing clients
2) Studying existing things
3) Long process - should be organized systematically
4) Identifies users & business entities
5) Get functional or domain knowledge
6) often goes outside-in

4) Design

- Deals with 'How'
- Consider several technical alternatives
- input is SRS
- Report to technical managment review
- finally delivers design document.

Design Goals

- Processing components
- Data component
- Different Design paradigms
- System structure
  - Decomposes complex system
  - Defines subsystems or modules.

5) Implementation

- Coding are done
- Translating design specification into source code
- Source code along with internal documentation
- to reduce cost of later phases
- General coding standards.

6) Testing

- Testing is important because software bugs could be expensive or even dangerous.
- Process of evaluating whether current software product meets requirements or not
- Checks for missing requirements, bugs or errors, security, reliability & performance.

7) Maintenance

- Goal is to modify & update software after delivery.
  - Correcting errors
  - Improving performance or capabilities
  - Deletion of obsolete features
  - Optimization

→ Types of software maintenance

- Corrective
- Adaptive
- Preventive
- Perfective

(costliest Phase)

8) Software retirement process

- Application Decomission or Application sunsetting.
- final stage of life cycle.
- Shutting down
- Reasons
  - Replaced
  - Release no longer supported
  - Redundant
  - Obsolete.

#DOMAIN ANALYSIS

- The process by which a software engineer learns about the domain to better understand the problem.
  - The domain is general field of business or technology in which the clients will use the software.
  - The domain expert is a person who has a deep knowledge of domain.

⊙ Benefits of performing domain analysis is ;
- faster development - Better system - Anticipation of extensions

→ It is important to write summary of information found during domain analysis. This is called Domain Analysis Document.

#Types of REQUIREMENTS

1) What is requirement ?
It is statement describing either
1) an aspect of what proposed system must do
2) a constraint of system's development

→ The set of requirements as a whole represents a negotiated agreement among stakeholders.

→ A collection of requirements is requirements document.

→ The requirements themselves are descriptions of system services and constraints that are generated during requirements engineering process.

Types of Requirements

Functional requirements
- Describe what the system should do.
- Statements of services the system should provide, how system should react to particular inputs & how system should react in particular situation.

Non-functional requirements
- Constraints that must be adhered to during development.
- Constraints on the services or functions offered by system such as timing constraints, platform constraints, constraints on development etc.

• Functional requirements
- What inputs system should accept
- what output system should produce
- What data system should store that other systems might use
- what computations system should perform
- timing & synchronization of above.

• Non-functional requirements
All must be verifiable

Quality requirements
reflects: usability, reliability, efficiency, reusability, maintainability
- Response time
- Throughput
- Resource usage
- Reliability
- Availability
- Recovery from failure
- Allowances for reusability
- Allowances for maintainability & enhancement.

2) Platform requirements

Categories constraining environment & technology of system.

- Platform
- technology to be used.

) Process requirements

Categories constraining project plan & development methods

- Development process to be used
- Cost & delivery date.
  - Often put in contract or project plan instead.

Use Case ANALYSIS

- A use case is a typical sequence of actions that a user performs in order to complete a given task.
- The objective of use case analysis is to model system from point of view of:
  - how users interact with this system
  - when trying to achieve their objectives

It is one of the key activities in requirements analysis.

→ First step in use case analysis is to determine types of users or other systems that will use the faculties of this system. These are called actors.

→ Second step in use case analysis is to determine the task that each actor will need to do with system.
Each task is use case.

- How to describe a single use case

A. Name : Give a short, descriptive name to use case.
B. Actors : List the actors who can perform this use case.
C. Goals : Explain what actors are trying to achieve.
D. Preconditions : State of system before the use case.
E. Summary : Give a short informal description
F. Related use cases.
G. Steps : describe each step using 2-column format, The left columns showing actions taken by actor; right column showing systems response.
H. Post conditions : State of system in following completion.

A & G are most important.

example

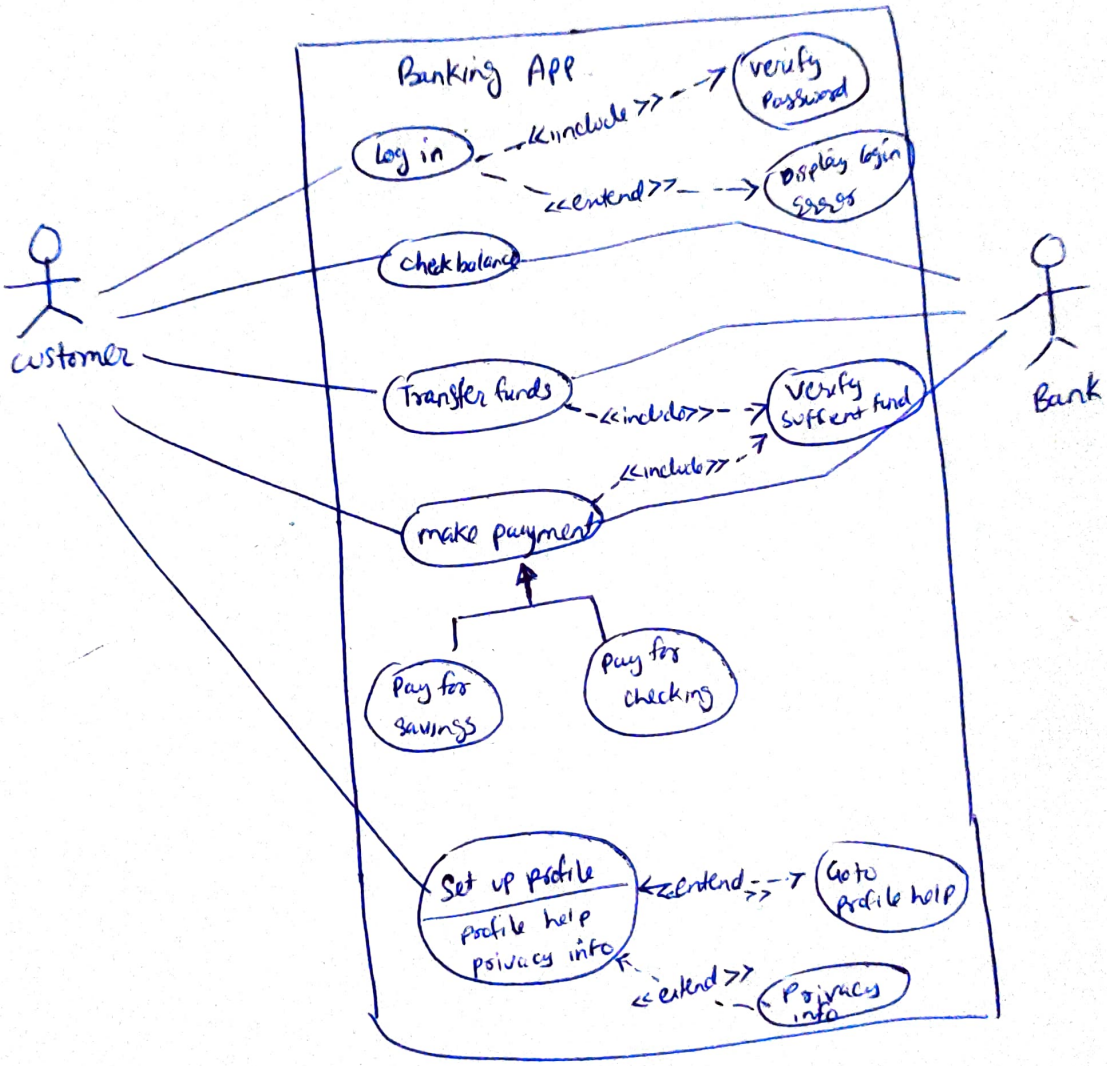
- Use case : Check out an item for a borrower
- Actors : Checkout clerk (regularly), chief librarian (occasionally)
- Goals : To help the borrower to borrow the item if they are allowed and to ensure a proper record in entered of loan.
- Preconditions : borrower must have a valid card & not owe any fines, the item must have a valid barcode & not be from the reference section.
- Steps :

Actor actions	system responses
1. Scan item's barcode & barcode of borrower's card	• Display confirmation that loan is allowed
2. Confirm that loan is to be initiated	• Display confirmation that loan has been recorded

- Post conditions : The system has a record of fact that the item is borrowed & date it is due.

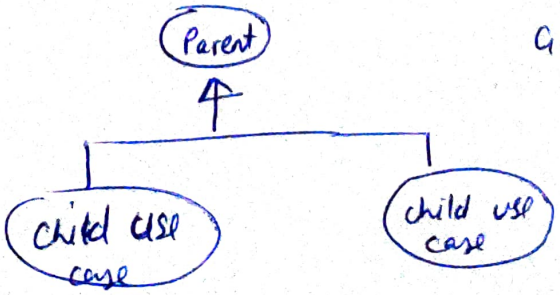
Use Case diagrams

- Use case diagrams are UML's notation for showing relationships among set of use cases & actors.
- There are 2 main symbols in use case diagrams:
  - An actor is shown as a stick person
  - a use case is shown as an ellipse.
- Lines indicate which actors perform which use case.



Generalization

also called inheritance



benefits of use case

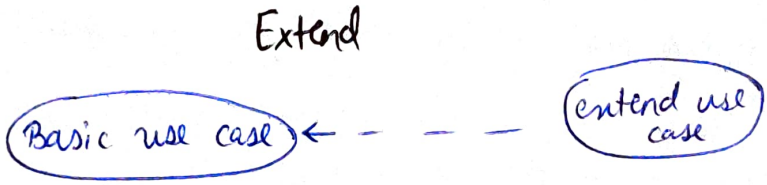
They can :

1) Help to define scope of system.
2) Be used to plan the development process
3) Be used to both develop & validate the requirements.
4) Form basis for definition of test case.
5) Be used to structure user manuals.

Extension

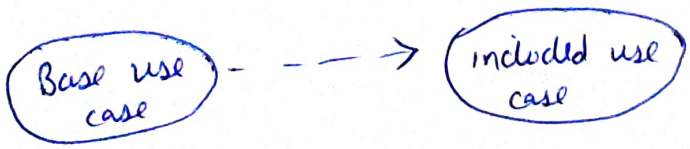
Used to make optional interactions explicit or to handle exceptional cases.

Keep description of basic use case simple.



Inclusion

- Show the relationship betn base use case & included use case.
- every time base use case is executed, the included use case is executed well.
- Allow one to express commonality betn several different use cases.
- Are included in other use cases.
  - → Even very different use cases can share sequence of actions
  - → Enable you to avoid repeating details in multiple use cases.



Some techniques for gathering & analysing Requirements

1) observation 2) interviewing 3) Brainstorming 4) Prototyping

- **Brainstorming**

Appoint an experienced moderator
Decide a 'trigger' question
- Arrange attendes around table.
- Ask each participant to write an answer & pass paper to its neighbour.

advantages

- Spontaneous new ideas
- Anonymity ensured
- Ideas created in parallel
- no need to wait for turn.

- **Prototyping**

- Simplest kind : paper prototype

set of pictures of system that are shown to users in sequence to explain what could happen.

- Most common : mock-up of the system's UI
  - Written in rapid prototyping language
  - Does not normally perform any computations, access any database or interact with any other systems.
  - Only a requirement gathering tool.

Some techniques for gathering & analysing Requirements

1) Observation 2) interviewing 3) Brainstorming 4) Prototyping

- **Brainstorming**
  - Appoint an experienced moderator
  - Decide a 'bigger question'
  - Arrange attendes around table.
  - Ask each participant to write an answer & pass paper to its neighbour.

advantages

- Spontaneous new ideas
- Anonymity ensured
- Ideas created in parallel
- no need to wait for turn.

- **Prototyping**
  - Simplest kind : paper prototype

set of pictures of system that are shown to users in sequence to explain what could happen.

  - Most common : mock-up of the system's UI

- Written in rapid prototyping language
- Does not normally perform any computations, access any databases or interact with any other systems.
- Only a requirement gathering tool.

Module 2

Principal of software design

Definition

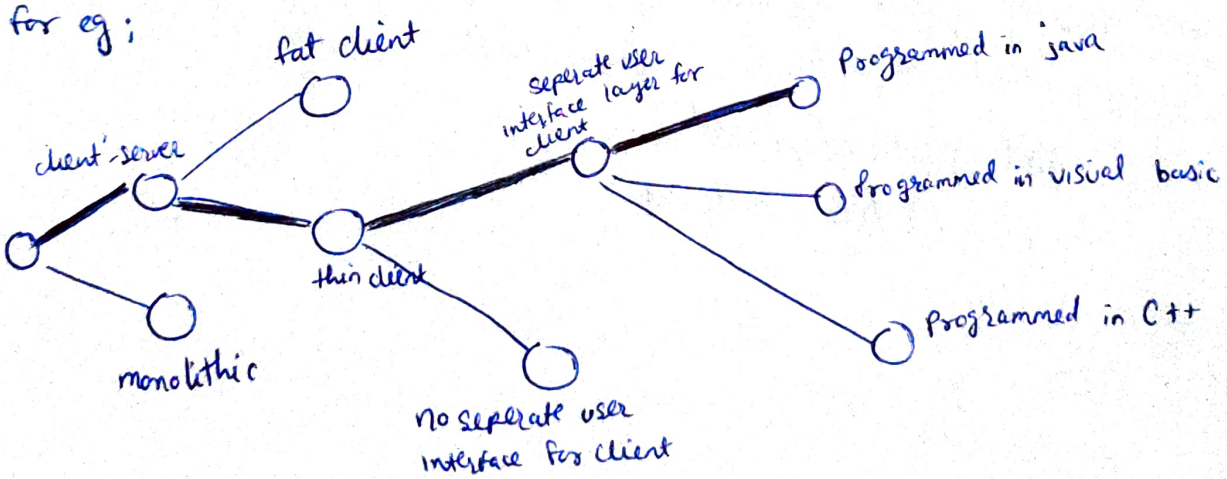
Design is a problem solving process whose objective is to find and describe a way;

- To implement the system's functional requirements
- While respecting constraints imposed by quality, platform & process requirements including budget & deadlines.
- And while adhering to general principles of good quality.

Design space

The space of possible designs that could be achieved by choosing different sets of alternatives is often called design spaces.

for eg;



Parts of a system

- subsystems
- components
- modules

Component

- Any piece of software or hardware that has a clear role.
  - → a component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
  - → Many components are designed to be reusable.
  - → Conversely, others perform special-purpose functions.

Module

- A component that is defined at programming language level
  - eg methods, classes & packages are modules in Java.

System

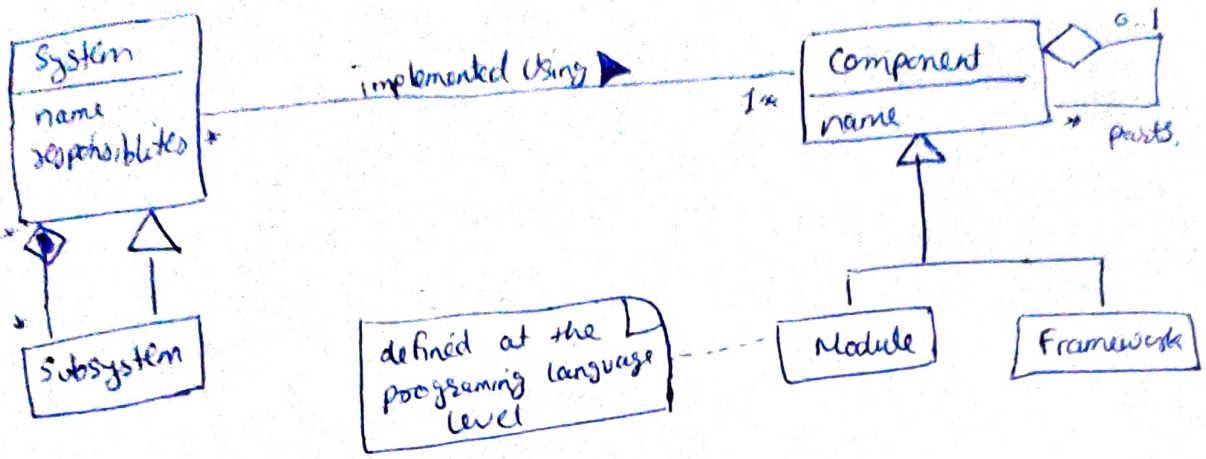
A logical entity, having a set of definable responsibilities or objectives and consisting of hardware, software or both.

- A system can have a specification which is then implemented by a collection of components.
- A system continues to exist, even if its components are changed or replaced
- Goal of requirements analysis is to determine responsibilities of system.

Subsystem

- A system that is part of a larger system, and which has definite interface.

UML diagram of system parts



Top-down & Bottom-up design

1) Top-down design
- First design very high level structure of system.
- Then gradually work down to detailed decisions about low-level constructs.
- Finally arrive at detailed decisions such as:
  - → format of particular data items.
  - → individual algorithms that will be used.

2) Bottom-up design
- Make decisions about reusable low-level utilities
- Then decide how these will be put together to create high level constructs.

→ A mix of top-down & bottom-up approach are normally used:
- Top-down design always needed to give system a good structure
- Bottom-up design is normally useful so that reusable components can be used.

Different aspects of design

1) Architecture design:
The division into subsystems and components.
   - How this will be connected
   - How they will interact
   - Their interface

2) Class design:
   The various features of classes

3) User interface design:

4) Algorithm design:
   The design of computational mechanisms

5) Protocol design:
   The design of communications protocol.

Principles of good designs

- Overall goals of good design:
  → Increasing profit by reducing cost & increasing revenue
  → Ensuring that we actually conform with requirements
  → Accelerating development
  → Increasing qualities such as
     - Usability
     - Efficiency
     - Reliability
     - Maintainability
     - Reusability

• Design Principle 1 : Devide & conquer

Seperate people work on each part, parts can be replaced or changed without affecting project.

- distributed system is devided up into clients & server
- system devided into subsystems
- subsystem can be devided up into one or more packages
- package is devided up into classes
- class is devided up into methods.

• Design Principle 2 : Increase cohesion where possible

A subsystem or module has high cohesion if it keeps together things that are related to each other, & keeps out other things

- Cohesion is called Intra-Module Binding
  → This makes the system whole easier to understand & change

  Type of cohesion:
  - functional, Layer • communicational • sequential • procedural
  - temporal • Utility.

- functional cohesion → achieved when all code that computes particular result is kept together - and everything else is kept out.
- Layer cohesion → All facilities for providing or accessing a set of related services are kept together & everything else is kept out.
- the layer should form hierarchy
  higher layer can access services of lower layer but lower layer can't access higher level,
- Set of procedures through which a layer provides its services is application programming interface (API)

- Communicational Cohesion → All the modules that access or manipulate certain data are kept together (in same class) & everything else is kept out.
- Sequential Cohesion → Procedures, in which one procedure provides input to the next are kept together - and everything else is kept out.
- Procedural cohesion → Procedures that are used one after another are kept together. weaker than sequential cohesion
- Temporal cohesion → Operations that are performed during the same face of execution of program are kept together.
- Utility Cohesion → When related utilities which can not be logically placed in other cohesive units are kept together.

Design Principle 3 : Reduce coupling where possible

- Coupling occurs when there are inter dependencies between one module & another.
  When interdependencies exist, changes in one place will require changes somewhere else
- Types of coupling :
  - content
  - common
  - control
  - stamp
  - data
  - Routine call
  - Type use
  - Inclusion / import
  - external

Data coupling ↑ Best
Stamp coupling
Control coupling
External coupling
common coupling
content coupling ↓ worst

Design Principle 4 : Keep level of abstraction as high as possible

- An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about internal details of implementation.
- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity.
- classes are data abstraction that contain procedural abstractions.
- Abstraction is increased by defining all variables as private.
- Attributes and associations are also data abstraction
- Methods are procedural abstractions.

Design Principle 5: Increase reusability where possible

- Design various aspects of your system so that they can be used again in other contexts.
- Strategies for increasing reusability are as follows:
  - → generalize your design as much as possible
  - → follow preceding three design principles
  - → Design your system to contain hooks
  - → Simplify your design as much as possible.

Design Principle 6 : Reuse existing designs & code where possible

- Design with reuse is complementary to design for reusability
  Cloning should not be seen as form of reuse.

Design Principle 7: Design for flexibility

- Also known as adaptability
- Actively anticipate changes that a design may have to undergo in future & prepare for them.
  - → Reduce coupling & increase cohesion
  - → Create abstraction
  - → Do not hard-code anything
  - → Use reusable code and make code reusable
  - → Level all options open.

Design Principle 8: Anticipate obsolescence

Plan for changes in technology or environment so that software will continue to run or can be easily changed.

Design principle 9: Design for portability

- Have a software run on as many platforms as possible:
- Avoid environment specific facilities

Design principle 10: Design for testability

Take steps to make testing easier.

Design principle 11: Design defensively

Never trust how others will try to use a component you are designing

Check that all the inputs to your component are valid.

1) Procedural Paradigm

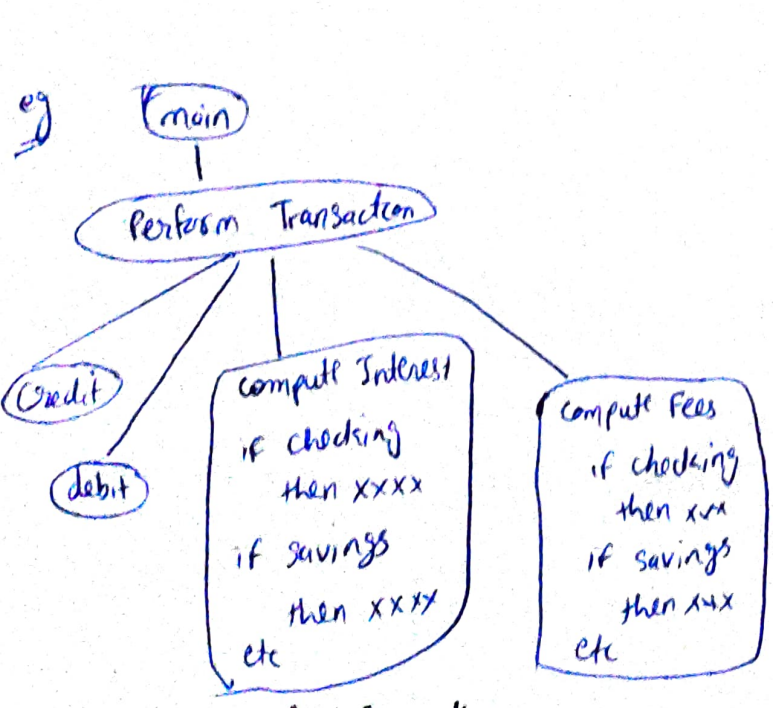
- Software is organized around notion of procedures.
- Procedural abstraction
  Work as long as data is simple

→ Adding data abstraction
- Groups together pieces of data that describe some entity
- Helps reduce system's complexity
  eg Records & structures.

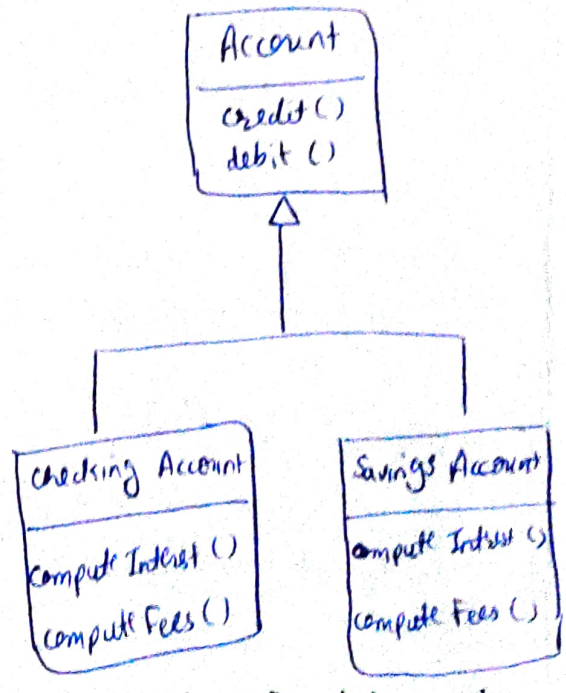
2) Object Oriented Paradigm

An approach to solution of problem in which all computations are performed in content of objects.

- The objects are instances of classes, which :
  - are data abstractions
  - contain procedural abstractions that operate on the objects.



Procedural Paradigm



Object Oriented paradigm

- **Object**

  A chunk of structured data in running software system

- **Class**

  A unit of abstraction in an object oriented program.
  Template or blueprint of objects

- an object is an element (or instance) of a class; objects have the behaviors of their class. The object is the actual component of programs, while the class specifies how instances are created and how they behave.

  Method :- is an action which an object is able to perform.

- **Instance variables**

  variables defined inside a class corresponding to data present in each instance. Also called fields or member variables.

  - Attributes → simple data eg name, dateOfBirth
  - Associations → relationship to other important classes eg supervisor

variable

- Refers to an object
- May refer to different objects at different points in time

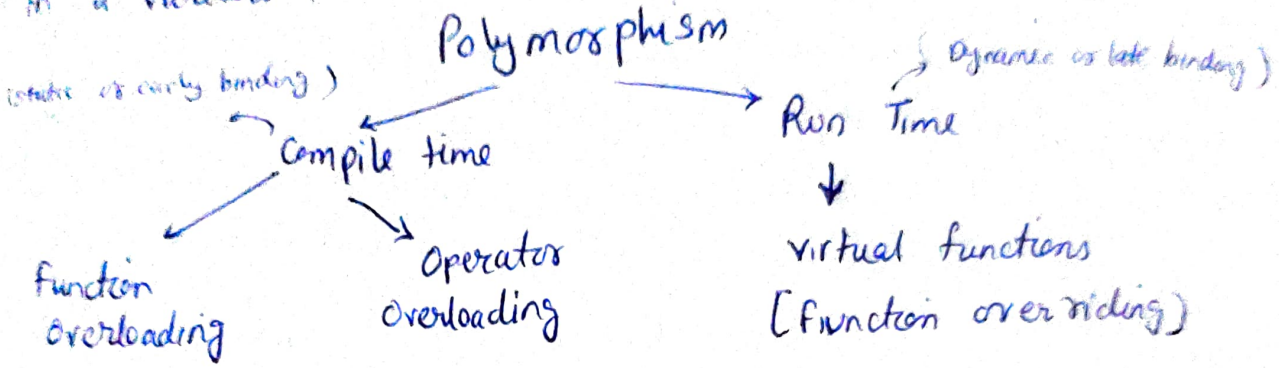
```
eg  Student s1 = new Student();
    s1.name = "ABC"
    s1.computeGrade()
```

- An object can be refered to by several different variables at same time.

```
Student s2 = new Student();
s2 = s1;
```

Polymorphism

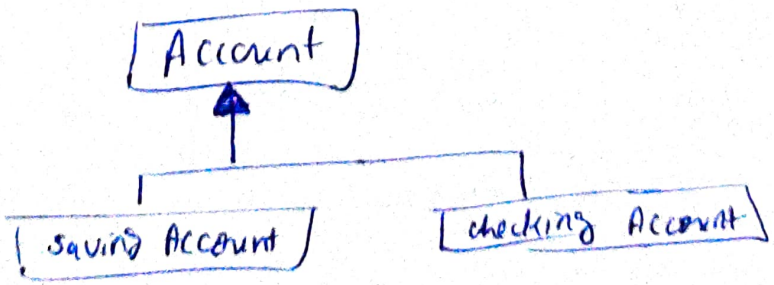
- A property of object oriented software by which an abstract operation may be performed in different ways in different classes.
  - → Requires that there are multiple methods of same name
  - → the choice of which one to execute depends on object that is in a variable.



Inheritance

- can be defined as process where one class acquires the properties (methods and fields) of another.
- inheritance hierarchies
  - Show relationship among superclasses & subclasses
  - A triangle (△) shows a generalization

eg



Is A rule

used to check generalization

- A student is a person
- A checking account is an account

but state is not a country. ∴ no inheritance

Abstract class

It is a restricted class that can not be used to create objects but act as foundation for another object.

use :- They are used to provide some common functionality across a set of related classes while also allowing default method implementations.

- Abstract class can not be instantiated
- An abstract classes contains abstract method, concrete methods or both
- Any class which extends abstract classes must override all methods of abstract class.

- overriding :- means same method name & same parameters, occur in different class that has inheritance relationship.

- Dynamic binding :-
  Occurs when decision about which method to run can only be made at run time.