# Software Engineering

Dr. Binu P Chacko
Principal
Prajyoti Niketan College,
Pudukad, THRISSUR

# Introduction

- Student s/w Vs industrial strength s/w
- Industrial strength s/w: S/W should be produced at reasonable cost, in a reasonable time (schedule), and should be of good quality
- Cost: person-months of effort (labor intensive)
- Productivity: LOC (KLOC) per person-month
- Pursuit of higher productivity is a basic driving force behind s/w engg and a major reason for using the different tools and techniques
- Unreliability of the s/w: dos and donts are different. The reason is the presence of defects in the s/w

# Attributes of s/w quality

- Functionality: The capability to provide functions which meet stated and implied needs when the s/w is used
- Reliability: The capability to provide failure-free service
- Usability: The capability to be understood, learned and used
- Efficiency: The capability to provide appropriate performance relative to the amount of resources used
- Maintainability: The capability to be modified for purposes of making corrections, improvements or adaptation
- Portability: the capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product

# s/w maintenance

- Corrective maintenance: the defects need to be corrected
- Adaptive maintenance: to satisfy the enhanced needs of the users and the environment
- Scale: Different set of methods must be used for developing large s/w
- Change: additional requirements need to be incorporated during development stage

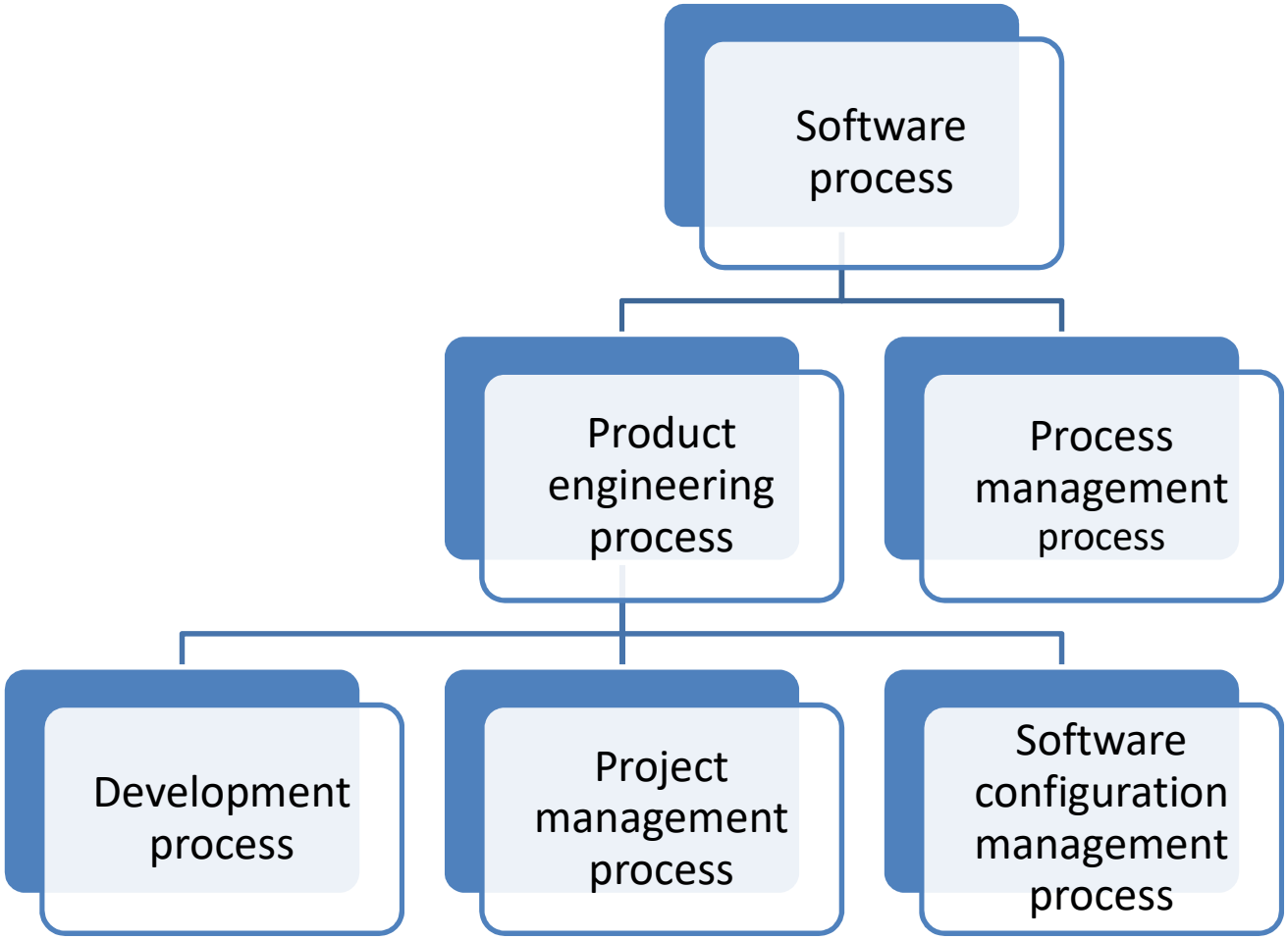
# Software Process

- Software Engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software
- Final quality delivered and productivity achieved depends on the skills of the people involved in the software project, the processes people use to perform the different tasks in the project, and the tools (technology) they use
- SE focuses on the process for producing the products
- Process is a sequence of steps performed for a given purpose. A process may be divided into *subprocesses* (sequence of steps for a stage)
- *Process specification is a description of the process* which presumably can be followed in some project to achieve the goal for which the process is designed
- Process model specifies a general process, which is optimum for a class of projects

# Cont…

- Software process: processes that deal with technical and management issues of software development
- components – development process, project management process
- Development process specifies all the engineering activities that need to be performed by programmers, designers, testers, etc.
- Management process (by project management) specifies how to plan and control these activities so that cost, schedule, quality, and other objectives are met
- s/w configuration control process deals with managing change by configuration controller
- The above three processes comprise product engineering processes (objective is to produce the desired product)
- Process management process (by SEPG) : The whole process of understanding the current process, analyzing its properties, determining how to improve, and then affecting the improvement
- Nonsoftware process: business, social and training processes

# Cont…


Software process
Product engineering process
Process management process
Development process
Project management process
Software configuration management process

# s/w Development Process Models

- Defines the tasks the project should perform and the order in which they should be done
- Provides generic guidelines for developing a suitable process for a project
- *Activities*: design, coding, testing
- Waterfall model (by Royce): phases are organized in a linear order
- Feasibility analysis → Requirements analysis and project planning → System design → Coding → Testing and integration → Installation → Operations and maintenance
- Each phase deals with a distinct and separate set of concerns, and thus helps the engineers and managers to handle the complexity of the problem
- To clearly identify the end of a phase and the beginning of the next, some certification mechanism (verification and validation) has to be employed at the end of each phase

# Cont…

- A good plan is based on the requirements of the system
- o/p of each phase is called work product which is in the form of *documents* (requirements document, Project plan, design document, test plan and test reports, final code, software manuals)
- Adv: simplicity – divides the large task into separate phases – each phase deals with separate logical concern
- Limitations: for new systems, user does not know the requirements
- A large project might take few years to complete – h/w technology changes
- The entire s/w is delivered in one shot at the end
- All requirements must be stated at the beginning
- Requires formal documents at the end of each phase

# Prototyping

- A throwaway prototype is built to help understand the requirements. This prototype is developed based on the currently known requirements
- Good idea for complicated and large systems
- Activities: design, coding, testing
- Develop requirements specification document → allow the clients to use and explore the prototype → give feedback to the developer → modify the prototype based on the feedback → repeat this cycle
- Reduce the cost for prototyping – include only valuable features, focus on quick development rather than quality, reduce testing
- Benefits: reduce the cost of actual s/w development, fewer changes in the requirements, quality of the s/w will be improved, developing a prototype mitigates many risks

# Iterative Development

- s/w should be developed in increments
- A simple initial implementation is done for a subset of the overall problem
- A project control list is created that contains all tasks that must be performed to obtain the final implementation. Each task should be performed in one step of the iterative enhancement process
- Designing the implementation for the selected task → coding and testing the implementation → performing an analysis of the partial system and updating the list as a result of the analysis. The process is iterated until project control list is empty
- Another approach: do requirements and architecture design in a standard waterfall or prototyping approach, but deliver the s/w iteratively. Adv: feedback from an iteration can be incorporated in the next iteration
- Popularity due to – after each iteration, some working s/w is released. Changing requirements can be incorporated in the s/w. Helps in developing stable requirements for the next iteration

# Rational Unified Process

- Iterative model by Rational (part of IBM)
- Designed for object oriented development using UML
- Development of a s/w is divided into cycles, each cycle delivers a fully working system (adds some capability to the existing system)

Each cycle is broken into:-

- Inception phase: establish goals and scope of the project. Completion: lifecycle objectives
- Specify vision and high level capability of the system, expected benefits, plan (cost & schedule), risks
- Based on the o/p of this phase, a go/no-go decision is taken
- Elaboration phase: design the architecture of the system. Completion: lifecycle architecture
- Take decision regarding technology and architecture
- *o/p*: technical evaluation of the proposed solution, cost/benefit analysis

# Cont…

- Construction phase: build and test the s/w
- Deliver the s/w along with manuals. Completion: initial operational capability
- Transition phase: move the s/w from development environment to client's environment
- Additional testing, training, conversion of old data to this s/w.
- Completion: product release
- Subprocesses: requirements, analysis and design, implementation, test, deployment, project mgt, configuration mgt
- Difference: RUP separated the phases from tasks and allows multiple subprocesses to function within a phase. Provides flexible process model

# Timeboxing Model

- Parallelism b/w different iterations is employed. Development of a new release happens in parallel with the development of the current release
- Reduces average delivery time for iterations by utilizing additional manpower
- Basic unit of development is a time box of fixed duration. Requirements or features should be fit into the time box
- Each time box is divided into a sequence of stages – each stage performs some clearly defined task for the iteration – duration of each stage is same – separate team for each stage
- A time box consisting of 3 stages: requirement specification, build, deployment
- If time box size is T days, first s/w delivery after T days, subsequent deliveries after every T/3 days
- Suitable for project with large number of features
- Increased complexity of the project mgt

# Agile Processes

Agile approaches are based on

- Working s/w is the key measure of progress in a project
- s/w should be developed and delivered rapidly in small increments
- Late changes in the requirements should be entertained
- Face-to-face communication is preferred over documentation
- Continuous feedback and involvement of customer is necessary
- Simple design which evolves and improves with time is a better approach
- Decide delivery dates
- Deliver high quality s/w at low cost and cycle time

# Agile Methodology – eXtreme Programming

- Changes are inevitable rather than treating changes as undesirable – for this, development process has to be lightweight and quick to respond – develop s/w iteratively
- Suitable where requirements change is high and where requirement risks are considerable
- XP starts with user stories (no detailed requirements)
- Using time estimate and stories release planning is done
- Bugs found during acceptance testing for an iteration can form work items for the next iteration
- An iteration starts with iteration planning. High-value and high-risk stories are considered as higher priority and implemented in early iterations
- Development is done by pairs of programmers
- Test-driven development: code should be written to pass the unit test before the actual code is written
- Refactoring to improve the design, and then use refactored code for further development. During refactoring, no new functionality is added
- Encourages frequent integration of different units
- www.extremeprogramming.org

# Project Management Process

- How long each phase should last, how many resources should be assigned to a phase, how a phase should be monitored
- Basic task is to plan the detailed implementation of the process for the particular project and then ensure that the plan is properly executed

Activities

- Planning: develop a plan for s/w development
- Cost estimation, schedule and milestone determination, project staffing, quality control plans, controlling and monitoring plans
- Project monitoring and control includes all activities the project management has to perform while the development is going on
- Monitor potential risks for the project, interpretation of the information
- Termination analysis is performed when the development process is over
- Provide information about the development process and learn from the project to improve the process

# Software Requirements Specification

- Major parties interested in a new system: need of a client, created by developer, used by end user
- Purpose: to bridge communication gap between client and developer
- Advantages: SRS establishes the basis for agreement between client and supplier
- SRS provides a reference for validation of the final product. An error in SRS will reflect in the final product also
- High quality SRS is a prerequisite to high quality software
- High quality SRS reduces the development cost

# Requirement Process

- Sequence of activities that need to be performed in the requirements phase

Tasks

- Problem analysis: starts with problem statement
- Problem domain and environments are modeled, constraints on the system, i/p, o/p
- Meeting of analyst, client and end users
- Documents describing the work or organization and o/p from existing methods may be given to SA
- Role of SA: listener, seek clarification, verification, document information, build some models, brainstorming, explain to client what he understands
- Requirements specification
- Requirements validation: to verify and ensure quality
- Requirements process terminate with the production of validated SRS

# Requirements Specification

- Modeling is a tool to get complete knowledge about the proposed system
- Modeling focuses on problem structure, not its external behavior, performance constraints, design constraints, standards compliance or recovery
- SRS is written based on knowledge acquired during analysis

## Characteristics

- Correct
- *Complete*: more details or too few details may not be desirable
- Unambiguous: if every requirement specified has one and only one interpretation
- Verifiable
- Consistent
- Ranked for importance and/or stability of requirements

# Components

- Functional requirements specify the expected behavior of the system – which o/p should be produced from the given i/p
- Relationship b/w i/p and o/p, system behavior in abnormal situations
- System behaviour for all foreseen inputs/system states should be specified
- Description of i/p and their source, units of measure, range of valid i/p, operation to be performed on i/p
- Performance requirements specifies performance constraints on the s/w system
- Types: static, dynamic
- Static requirements (capacity of the system) don't impose constraint on the execution characteristics of the system. E.g. number of terminals to be supported, number of simultaneous users, number and size of files to process
- Dynamic requirements specify constraints on execution behavior (response time and throughput) of the system.
- Acceptable range of different performance parameters, acceptable performance for both normal and peak workload conditions

# Cont…

Design constraints

- Standards compliance: specify the requirements for the standards the system must follow, report format and accounting procedures, audit requirements
- h/w limitations: type of machines used, OS, languages supported, limits on primary and secondary storage
- Reliability and fault tolerance: What the system should do if some failure occurs?
- Security: restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, use of passwords and cryptography techniques, maintain log of activities in the system, assessment of security threats, programming techniques, use tools to detect buffer overflow

External interface: interactions of s/w with people, h/w and other s/w

- Create preliminary user manual

# General Structure of SRS

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definition, Acronyms, Abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 General constraints
   - 2.5 Assumptions and dependencies
3. Specific requirements

- No one method is suitable for all projects

# Cont…

- Section 1: clarify motivation, business objectives and scope of the project
- Section 2: how the system fits into larger system, an overview of all the requirements
- Product perspective: relationship of the product to other products
- Schematic diagrams showing different functions and their relationships with each other can be given

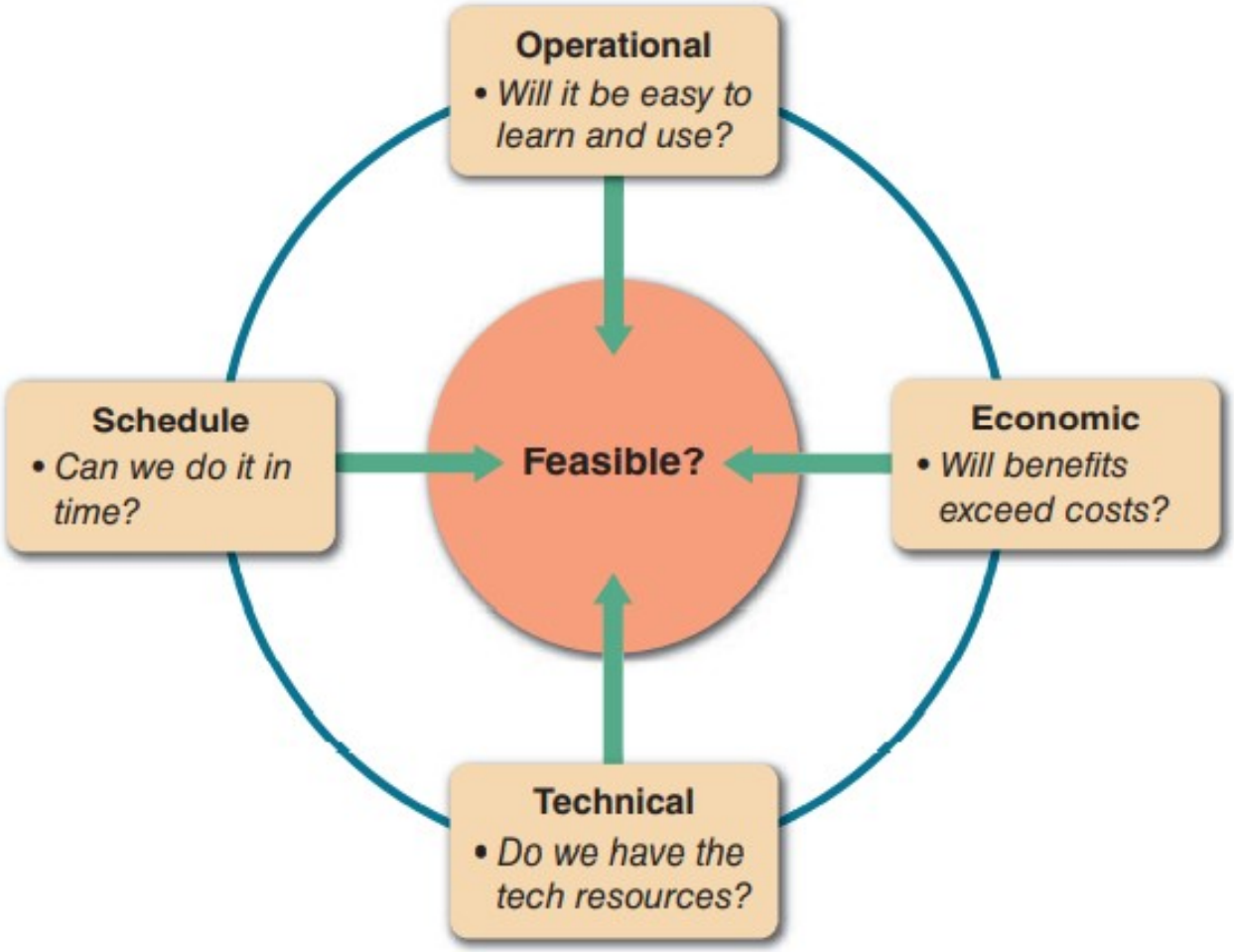
# Organization of Specific Requirements

3. Detailed requirements
   - 3.1 External interface requirements
     - 3.1.1 User interface
     - 3.1.2 H/W interface
     - 3.1.3 s/w interface: s/w required to run this system
     - 3.1.4 Communication interface
   - 3.2 Functional requirements
     - 3.2.1 Mode 1
       - 3.2.1.1 Functional requirement 1.1
       - :
       - 3.2.1.n Functional requirement 1.n
     - :
     - 3.2.m Mode m
       - :
   - 3.3 Performance requirements
   - 3.4 Design constraints
   - 3.5 Attributes
   - 3.6 Other requirements

# Cont…

- Communication interface specify the communication with other systems
- Section 3.2: functional capabilities of all modes of operation
- Required i/p (source of i/p, units of measure, valid ranges, accuracies), desired o/p, and processing requirements have to be specified
- Section 3.3: specify both static and dynamic performance requirements

# Feasibility Study



# Cont…

- Operational feasibility: whether the system will be used effectively
- Economic feasibility: whether the projected benefits outweighs estimated cost of the system
- Tangible cost, intangible cost (low employee morale)
- Tangible benefit: decrease in expense, increase in revenue
- Intangible benefit: user friendly system
- Technical feasibility: technical resources needed to develop, purchase, install, or operate the system
- Schedule feasibility: whether the project can be implemented in an acceptable time frame. Consider the interaction between time and cost

# Project Planning

Project manager performs…

- Estimation of cost, duration and effort
- Scheduling manpower and other resources
- Staff organisation and staffing plans are made
- Risk management: risk identification, analysis, abatement planning
- Quality assurance plan, configuration management plan
- Sliding window planning: Project planning over a number of stages
- Project managers document the plans in a software project management plan (SPMP) document

# Cost Estimation

- Estimation techniques: Emperical, Heuristic, Analytical

## Delphi cost estimation

- By a group of experts and a co-ordinator
- Co-ordinator gives SRS document and a form for recording cost estimate to each estimator
- Estimators submit cost estimate
- Co-ordinator prepares the summary of responses and distribute to estimators
- Based on the summary, estimators re-estimate. This process continues
- Finally, co-ordinator compiles the results and prepare the estimate

# COnstructive COst estimation MOdel

## Basic COCOMO

- Project cost = estimated effort in man_month * manpower cost per month

# Functional Specification with Use Cases

- Use cases (UC) specify *functionality of a system* by specifying the external behavior of the system, captured as interactions of the users with the system
- Used to describe the business processes of larger business or organization
- Actor: person or system which uses the software system for achieving some goal
- Primary actor: main actor who initiates a UC for achieving a goal
- Some agent may actually executes UC for primary actor
- E.g. time-driven trigger
- Scenario: describes a set of actions performed to achieve a goal under some specified conditions
- A step in a scenario is a logically complete action by the actor (enter information), some logical step that the system performs (validate information), or an internal state change by the system (update the record)

# Cont…

- Main success scenario: describes the interaction in which all steps in the scenario succeed
- Extension (exception) scenarios: describe the system behavior if some of the steps in the main scenario don't complete successfully
- Use case is a collection of all success and extension scenarios related to the goal
- To achieve goal, a system can divide it into sub goals. These sub goals may be treated as separate use cases
- Use case also specifies a scope
- Scope of system use case is the system being built; scope of a component use case is a subsystem
- Adv: use cases focus on external behavior, thereby avoiding doing internal design during requirements

# Developing Use Cases

Four levels

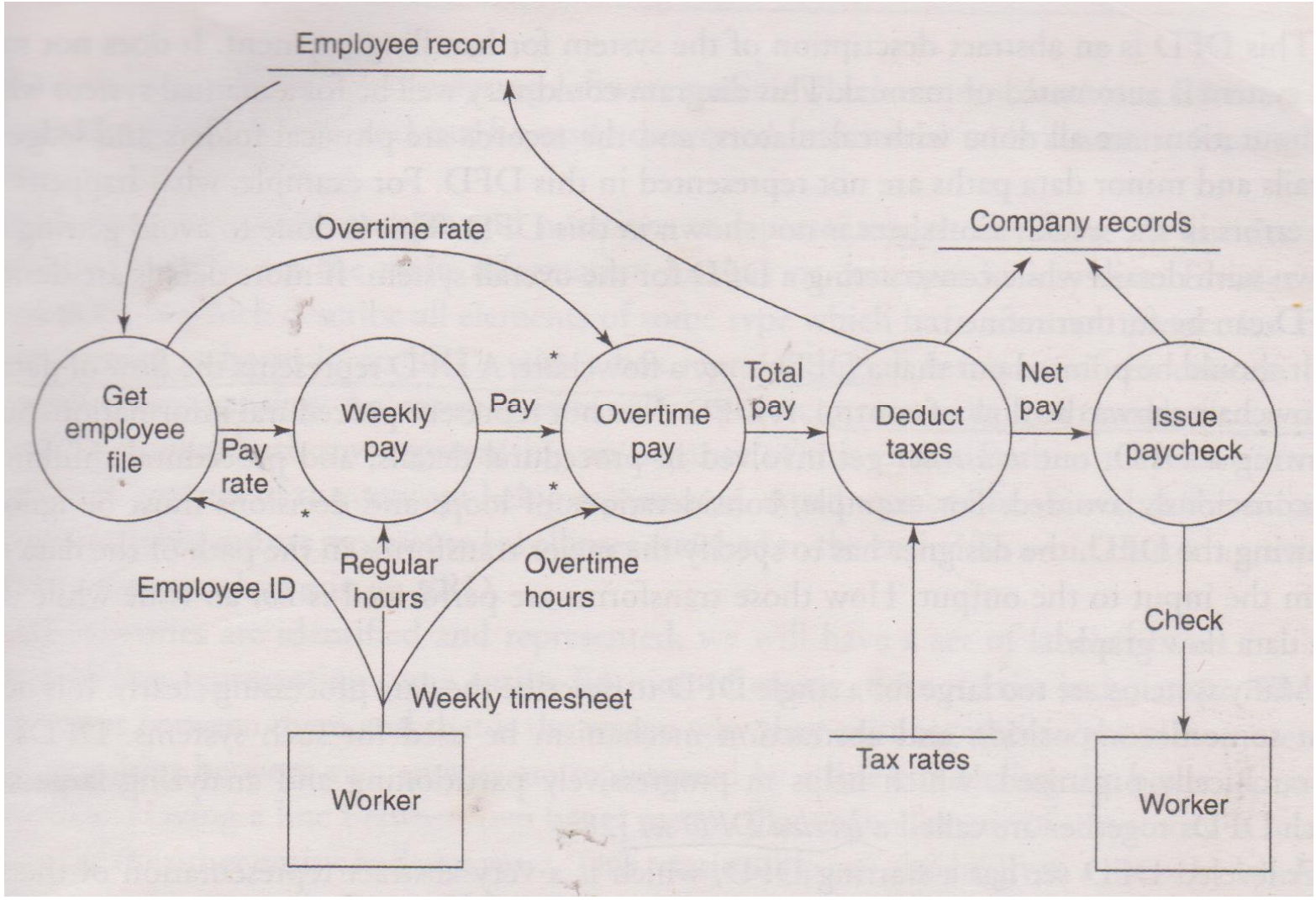
- Actors and goals: specifies actors for each goal. UCs together specify the scope of the system and give an overall view of what it does
- Main success scenarios: system behavior for each use case is specified
- Failure conditions: identify all possible failure conditions
- Failure handling: what should be the behavior under different failure conditions

# Problem Analysis

- Aim: obtain a clear understanding of the needs of the clients and users
- Consultant: helps the clients and users to identify their needs
- Principle – Divide and conquer – partition the problem into sub problems
- Concepts of state and projection (view point) of system can be used in partitioning process

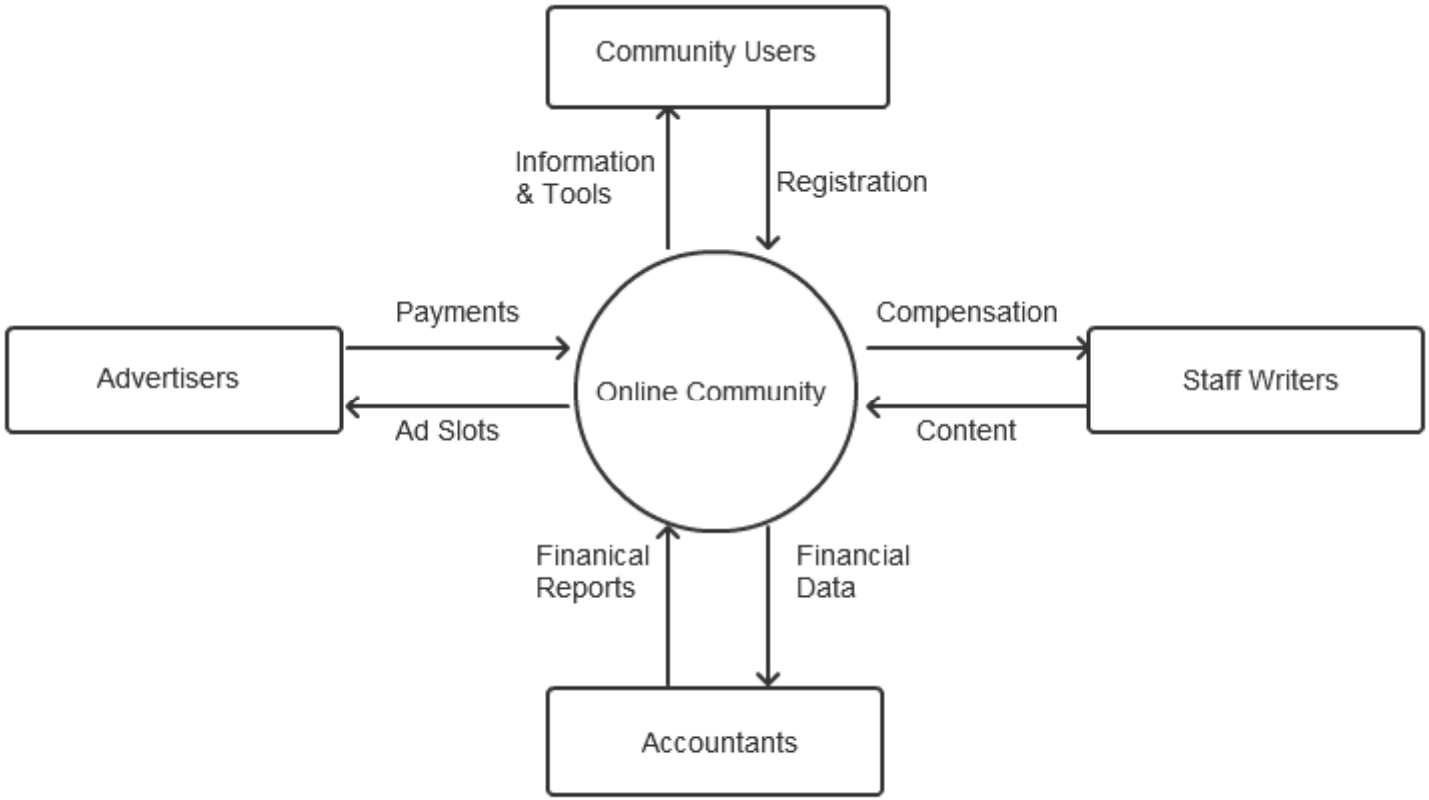
# DFD

- Shows flow of data through a system (transforms input to output)
- Process: the agent that performs transformation of data from one state to another is called a process
- DFD shows movement of data through different processes in the system
- It is used with manual or automated system
- It can refined with more details (leveled DFD). Process is expanded into DFD. Data decomposition also occurs with process decomposition
- DFD – data flow, Flowchart – control flow


Employee record
Overtime rate
Company records
Get employee file
Pay rate
Weekly pay
Pay
Overtime pay
Total pay
Deduct taxes
Net pay
Issue paycheck
Employee ID
Regular hours
Overtime hours
Check
Weekly timesheet
Tax rates
Worker
Worker

* AND + OR

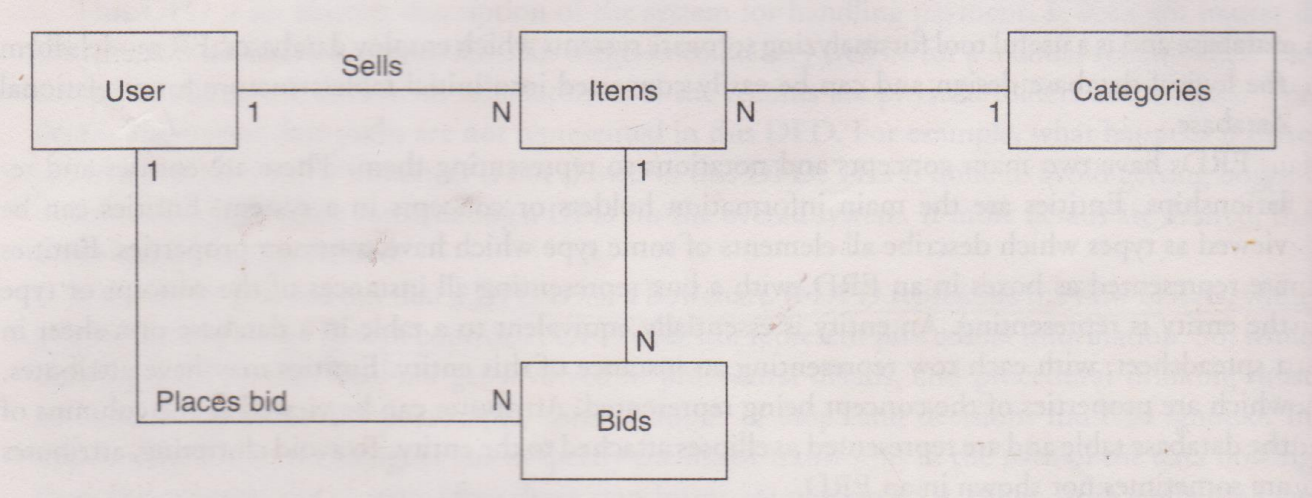
# Context Diagram




- Entire system is drawn as a single process with all its inputs, outputs, sources and sinks (before DFD)

# Entity Relationship Diagram

- To model data in the system, how data items relate to each other
- To represent the structure of database
- Entities (tables) are represented as boxes
- Entity may have attributes (columns)
- Each element of one entity is related to elements of other entity or same entity
- Item table should have a User-ID field (one-to-many), Bid table must have a User-ID and Item-ID



# Coupling

- Coupling and cohesion are two modularization criteria
- A system is considered modular if it consists of discrete modules so that each module can be implemented separately, and a change to one module has minimal impact on other modules
- A modular system can be built easily by putting its modules together
- Helps in system debugging - Isolates system problem to a module
- Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules
- Loosely coupled module is preferred to solve and modify it separately
- Coupling will increase if a module is used via internals or shared variables
- Two kinds of information that can flow along an interface - data (lowest degree of coupling) or control

# Cont…

- Interaction coupling occurs due to methods of a class invoking methods of other classes
- Worst form – if methods directly access internal parts of other methods
- Lowest – if methods communicate directly through parameters
- Component coupling refers to the interaction between two classes where a class has variables of the other class
- Three situations - A class has an instance variable of the type of other class, A class has a method whose parameter is of the type of other class, A class has a method which has a local variable of the type of other class
- Inheritance coupling is due to inheritance relationship between classes

# Cohesion

- How closely the elements of a module are related to each other
- Greater the cohesion of each module in the system, lower the coupling between modules

Levels of cohesion

- Coincidental cohesion (lowest level) occurs when there is no meaningful relationship among the elements of a module
- A module has Logical cohesion if some logical relationship between the elements of a module. E.g. a module that performs all inputs or all outputs
- Temporal cohesion is same as logical cohesion, except that the elements are related in time and are executed together. e.g. modules that perform initialization, cleanup, termination
- A procedurally cohesive module contains elements that belong to a common procedural unit. E.g. a loop or a sequence of decision making statements in a module may be combined to form a separate module

# Cont…

- A module with communicational cohesion has elements that are related by a reference to the same input or output data. i.e., the elements are together
- Output of one forms the input of another, we get sequential cohesion
- In functional cohesion (strongest cohesion) all the elements of the module are related to performing a single function. E.g. sort the array

Cohesion in object oriented systems

- Method cohesion: same as functional cohesion
- Class cohesion focuses on why different attributes and methods are together in this class
- Inheritance cohesion focuses on why classes are together in a hierarchy. Reasons – to model generalization-specialization relationship, for code reuse

# Validation

- To ensure that the SRS reflects the actual requirements accurately and clearly
- To check that the SRS document is itself of good quality

Types of errors

- Omission: some user requirement isn't included in SRS
- Inconsistency: due to contradictions within the requirements themselves or due to incompatibility of the stated requirements with the actual requirements of the client or with the environment in which the system will operate
- Incorrect fact: some fact recorded in SRS isn't correct
- Ambiguity: some requirements have multiple meanings
- Validation should focus on uncovering the above types of errors
- Inspection of SRS (or requirements review) is the most common method for validation
- Review group: author of requirements document, someone who understands the needs of the client, a person from design team, person responsible for maintaining the requirements document, s/w quality engineer

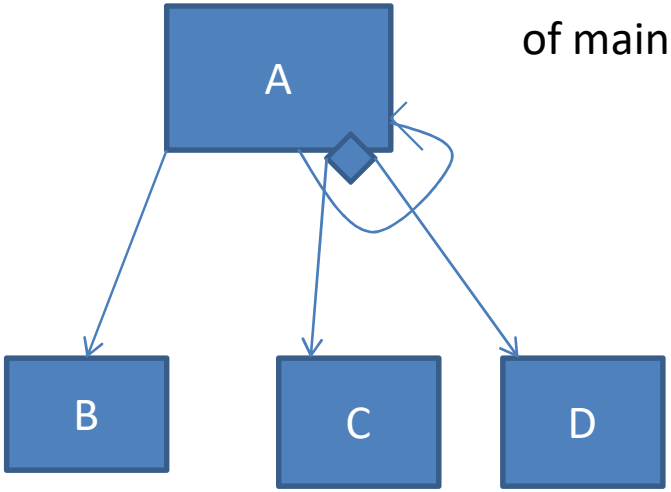
# Function Oriented Design

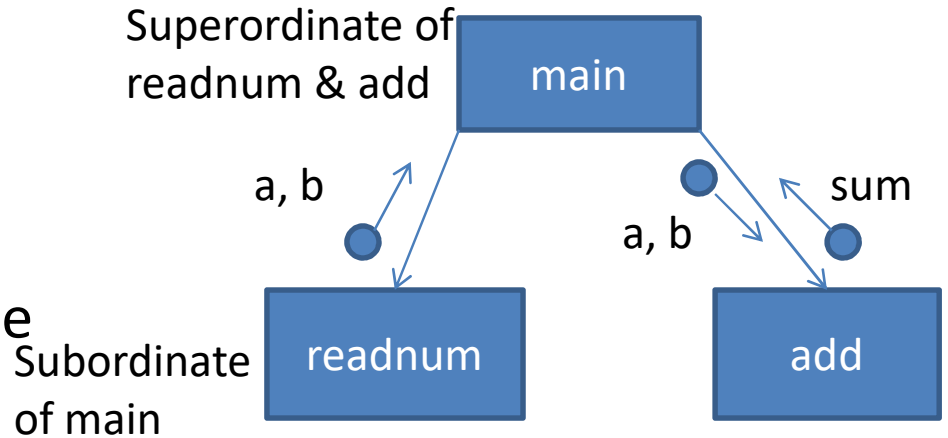
- Structure chart: used to represent the design graphically
- Major loops and decisions

can also be represented

in a structure chart

- i/p module, o/p module, transform module, coordinate module



# Structured Design Methodology

- Principle: problem partitioning at top level into subsystems – one for i/p, one for o/p, one for transformation
- i/p modules have to deal with issues of screens, reading data, formats, errors, exceptions, completeness of info, structure of info
- o/p modules have to prepare o/p in presentation formats, make charts, produce reports

Restate the problem as a DFD

- Construct DFD – deal with solution domain and develop a model of the system – shows major transforms or functions in the s/w

Identify the most abstract i/p and o/p data elements

- In most cases, i/p is first converted into a form on which transformation can be applied. Transformation module often produce o/p that have to be converted into desired o/p
- Goal: separate the transforms in DFD that convert i/p or o/p to the desired format from the ones that perform the actual transformations

# Cont…

- Identify highest abstract level of i/p and o/p
- Most abstract i/p data elements: data elements in the DFD that are farthest removed from physical i/p but are still i/p to the system. Data elements obtained after error checking, data validation, proper formatting, and conversion
- Most abstract o/p data elements: data elements that are most removed from the actual o/p but still considered as outgoing. These data elements are considered logical o/p data items. Transforms convert logical o/p into the required o/p form
- Central transforms: take the most abstract i/p and transforms it into most abstract o/p

First level factoring

- Specify main (coordinate) module whose purpose is to invoke the subordinate modules
- For each of the most abstract input/output data items, a subordinate module to the main module is specified
- Transform module

# Cont…

## Factoring of i/p, o/p and transform branches

- To simply these branches and distribute their work
- i/p module is considered as main module, and a subordinate i/p module is created for each i/p stream
- o/p module is created for each data stream
- Determine subtransforms that will together compose the overall transform and then repeat the process for newly found transforms until we reach the atomic modules

# Object Oriented Design

- Permit changes more easily
- New applications can use existing modules more effectively, thereby reducing development cost and cycle time
- Classes and Objects are the basic building blocks of OO design. Object is the basic runtime entity.
- Abstraction, encapsulation, interface
- In an object, the state is preserved and it persists through the life of the object
- The state and services of an object together define its behavior

Relationship among objects

- A link exists b/w objects if an object uses some services (by sending messages) of the other object
- Links b/w objects capture client-server type of relationship
- Aggregation generally implies containment. If an object A is an aggregation of object B, then object B will generally be within object A. A contained object can't survive without its containing object

# Program

```
class sample
{
  int a, b;
public:
  void read()
  {  cin>>a>>b;    }
  void display()
  {  cout<<a<<b; }
};
void main()
{  sample s;
   s.read();
   s.display();      }
```

# Inheritance

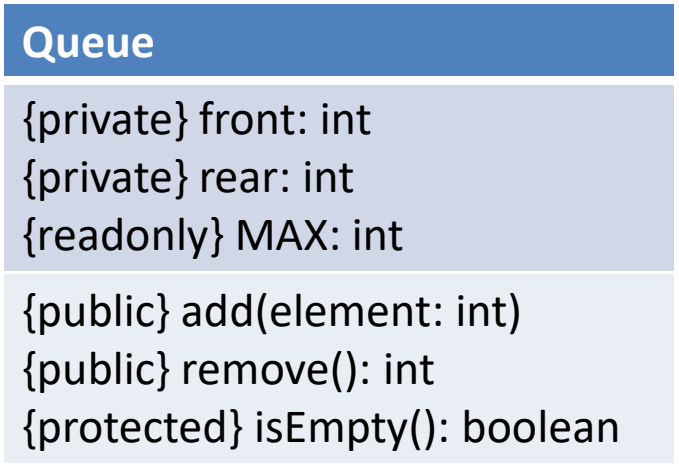
- Strict inheritance: a subclass takes all features of parent class + additional features
- Nonstrict inheritance: subclass doesn't have all the features of parent class or some features are redefined
- Polymorphism: a reference can refer to objects of different types at different times
- An entity has a static type and a dynamic type. Static type of an object is the type of which the object is declared in the program. Dynamic type can change from time to time and is known only at reference time.
- Polymorphism requires dynamic binding (code associated with a given procedure call isn't known until the moment of call)
- Dynamic binding reduces the size of code

# UML

- A graphical notation for expressing OOD

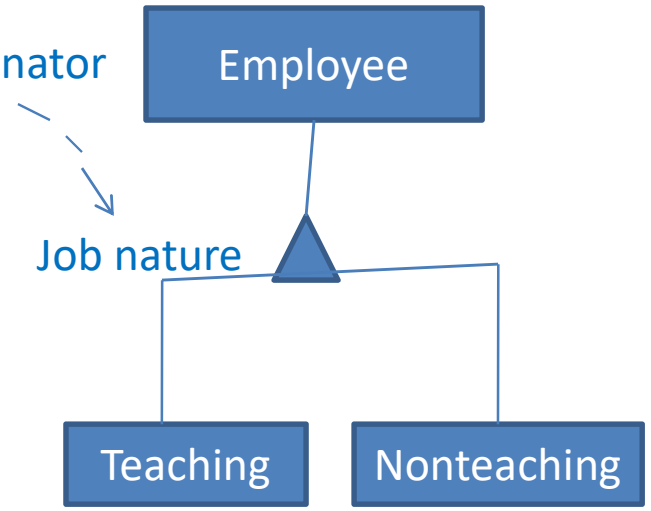
## Class diagram

- Defines classes, association b/w classes, and subtype-supertype relationship



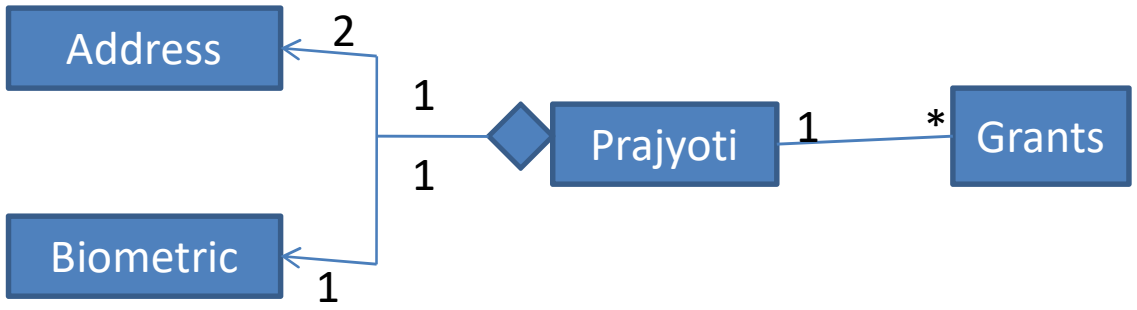

• Generalization-specialization relationship:

Properties of general significance are assigned to superclass, while properties which specialize an object are put in subclass. A subclass contains its own properties as well as those of superclass

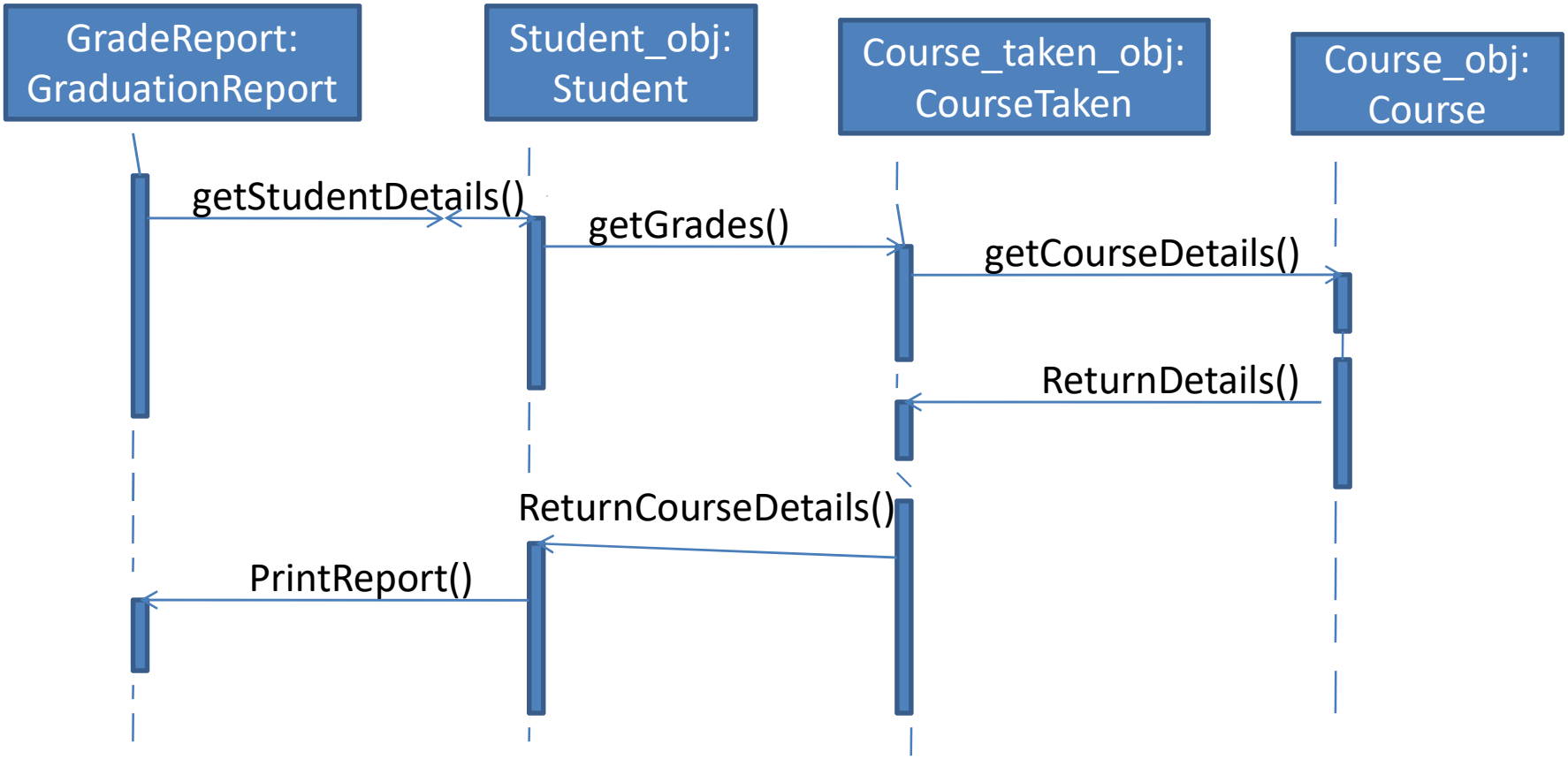


# Cont…

- Association (one-to-one, one-to-many, etc.): object of one class needs services from objects of other class
- Part-whole relationship: an object is composed of many parts, each part itself is an object – containment or aggregation

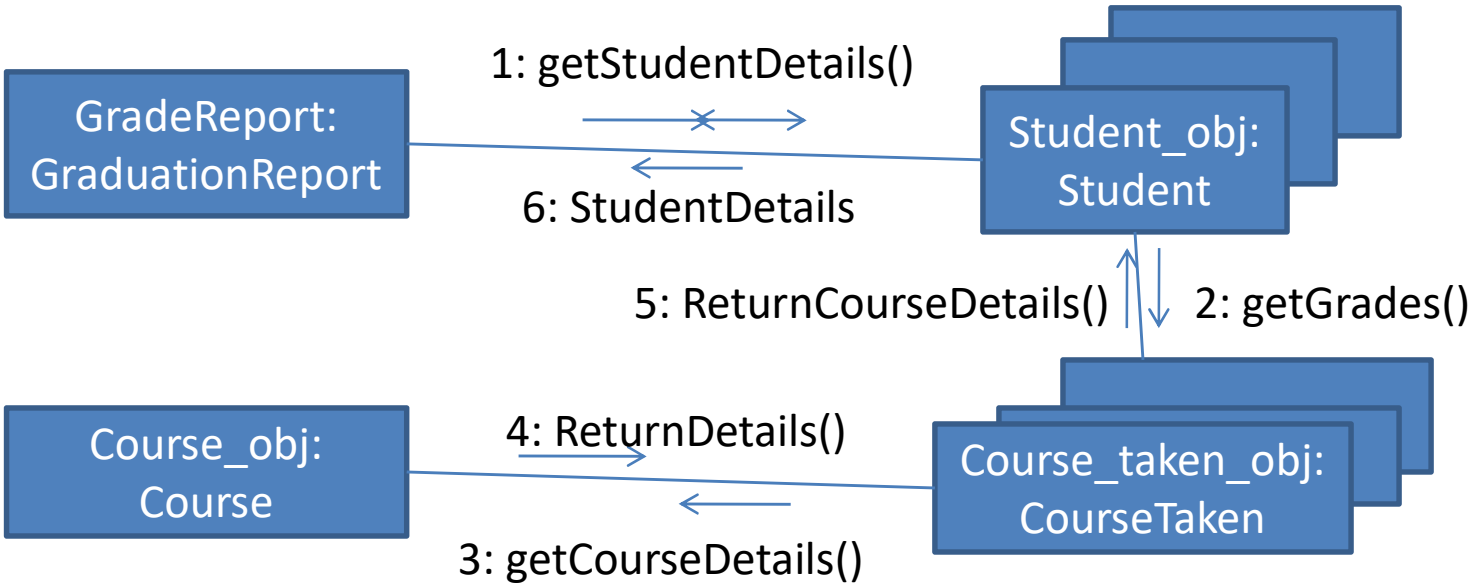


# Sequence Diagram



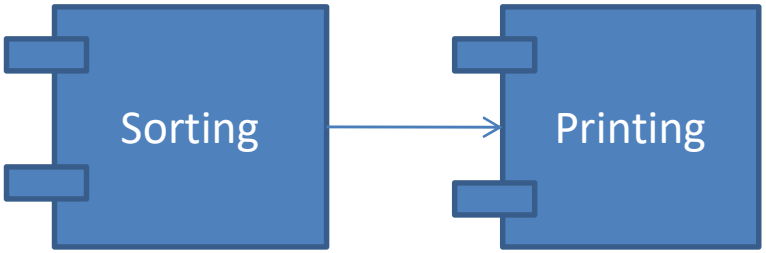

- Shows the series of messages exchanged b/w objects and their temporal ordering (shows dynamic behaviour)
- To model (objects' lives) the interaction b/w objects for a particular use case
- Vertical bar: lifeline of an object, arrow: message from one object to another

# Collaboration Diagram

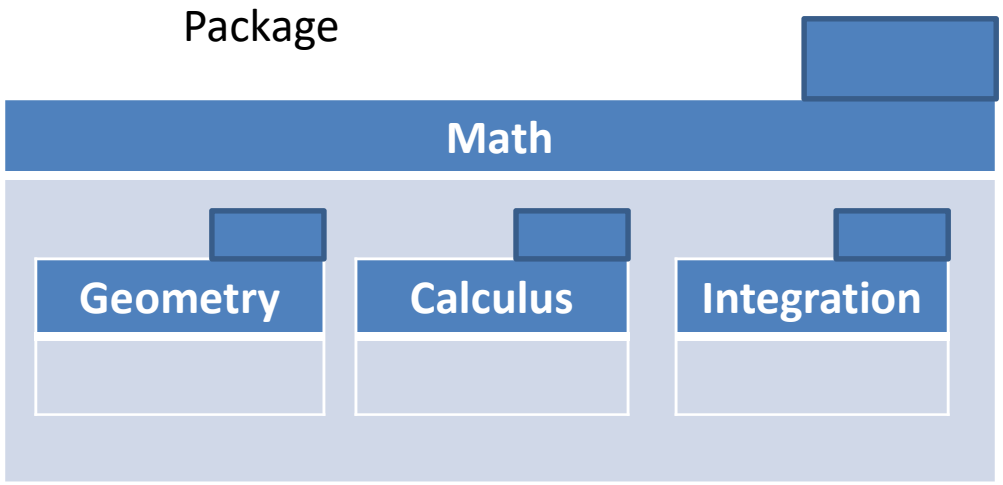


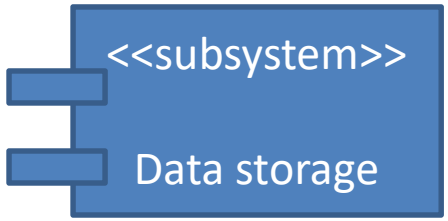

- Shows how objects communicate
- Chronological ordering of messages is given by message numbering
- Class diagram captures the structure of system's code, whereas many diagrams are needed to model the dynamic behavior of the system
- Interaction diagram (sequence diagram, collaboration diagram): how a system internally implements an operation

# Other Diagrams




Component - Connector




Subsystem

- Components often encapsulates subsystems and provide clearly defined interfaces through which components can be used by other components in the system
- Packages are formed by combining many classes
- State diagram: used to model the behavior of objects. How an object evolves as operations are performed on it
- State represents different states of the object; transition captures the performing of different operations on that object
- Activity diagram (like flowchart): for modeling dynamic behavior. Model a system by modeling the activities

# Design Methodology

- During architecture design, system is divided into high level subsystems or components

OOD consists of:-

- Identifying classes and relationships: requires identification of object types, structures b/w classes (both inheritance and aggregation), attributes of classes, association b/w classes, and the services provided. Trying to define class diagram (module-level design)
- Include an entity as an object if the system needs to remember it, needs some services from it, or it has multiple attributes
- Attributes add detail about the class and are the repositories of data for an object. Position each attribute properly using structures; common attribute in super class, and specific attribute in subclass
- While identifying attributes, new classes may be defined or old classes may disappear
- Structure must reflect the hierarchy in the problem domain
- For associations, we need to identify relationship b/w objects.

# Cont…

- Dynamic modeling: aims to specify how the state of various objects changes when events occur
- Events are interactions with the outside world and object-to-object interactions
- Each event has an initiator and a responder. Internal events – both initiator and responder are within the system. External event – initiator is outside the system. E.g. user or sensor
- A scenario is a sequence of events that occur in a particular execution of the system
- From scenarios different events performed on objects can be identified, which are then used to identify services on objects. Different scenarios together characterize the behavior of the system
- Dynamic modeling involves preparing interaction diagrams
- First model the main success scenarios, then the exceptional ones

# Cont…

- Functional modeling: how to compute o/p from i/p
- It is useful in cases where i/p – o/p mapping involves many steps
- It is represented by DFD. Most of the processing is done by operations on classes
- Defining internal classes and operations: each class is critically evaluated to see if it is needed in its present form in the final implementation
- Implementation of operations on classes: rough algorithms for implementation might be considered. A complex operation may get defined in terms of lower level operations on simpler classes
- Optimize and package: issue of efficiency
- Final structure shouldn't deviate too much from logical structure produced

# Detailed Design

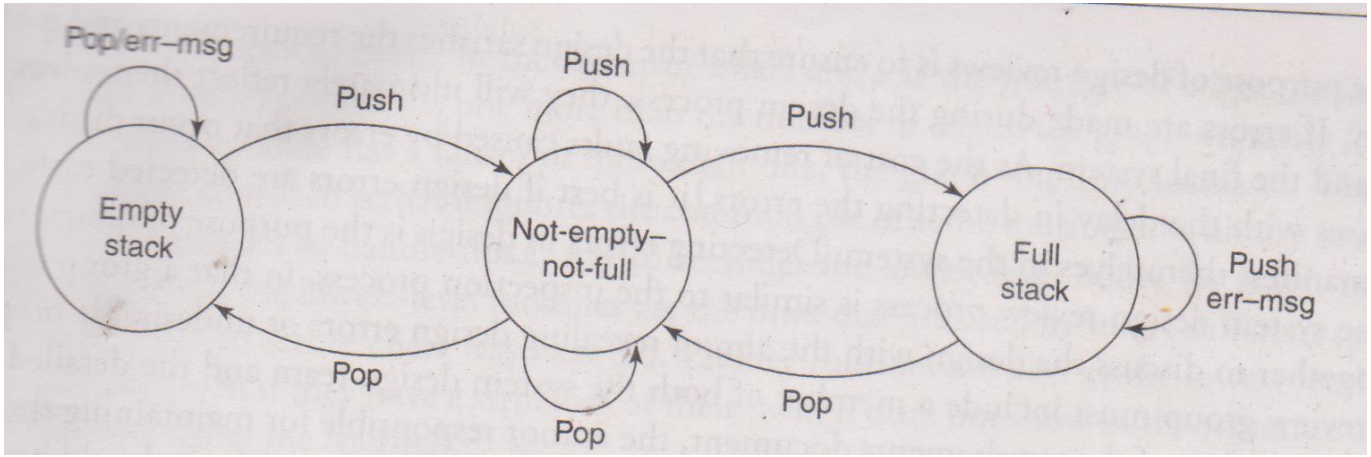
- Goal: specify logic for different modules. For this, develop an algorithm

Logic/Algorithm design

- Statement of the problem
- Develop a mathematical model for the problem
- Design the algorithm: decide data structure and program structure. Verify the correctness of the algorithm
- *Stepwise refinement technique*: a method for designing algorithms or logic for a module
- The process starts by converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements. In each step, one or several statements are decomposed into more detailed instructions. The process terminates when all instructions are converted into programming language statements

# State Modeling of Classes

- An object has some state and many operations on it
- State is the value of object's attributes, and event is the performing of an operation on the object
- State diagram shows how the state changes when an event is performed



# Verification

- o/p of the design activity should be verified before proceeding to the next phase
- Check for internal consistency. E.g interface of a module is consistent, data usage is consistent with declaration
- Most common approach for verification is design review. Purpose is to detect errors and to ensure that design satisfies the requirements and is of good quality
- Review group: member of system design team and detailed design team, author of requirements document and author responsible for maintaining design document, s/w quality engineer
- This team should reveal design errors or undesirable properties
- Design error: design doesn't fully support the requirements
- Design quality: modularity, efficiency

# User Interface Design

## Characteristics

- Speed of learning: use of metaphors and intuitive command names, consistency, component based interface (interfaces are developed using some standard UI components)
- Speed of use: time and user effort necessary to initiate and execute different commands should be minimal
- Speed of recall of command should be maximised
- Error prevention
- Aesthetic and attractive
- Commands supported by a UI should be consistent
- Provide feedback to various user actions
- Support for multiple skill level of users
- Error recovery (Undo facility)
- User guidance and online help

# Basic Concepts

- Online help system, Guidance messages, Error messages
- Mode (state)-based interface: only a subset of user interaction tasks can be performed. It is represented using state transition diagram
- Modeless interface: same set of commands can be invoked at any time
- GUI: multiple windows are displayed, Iconic information representation and symbolic information manipulation, command selection using menu, pointing device can be used, requires special terminals with graphics capabilities

# Types of UI

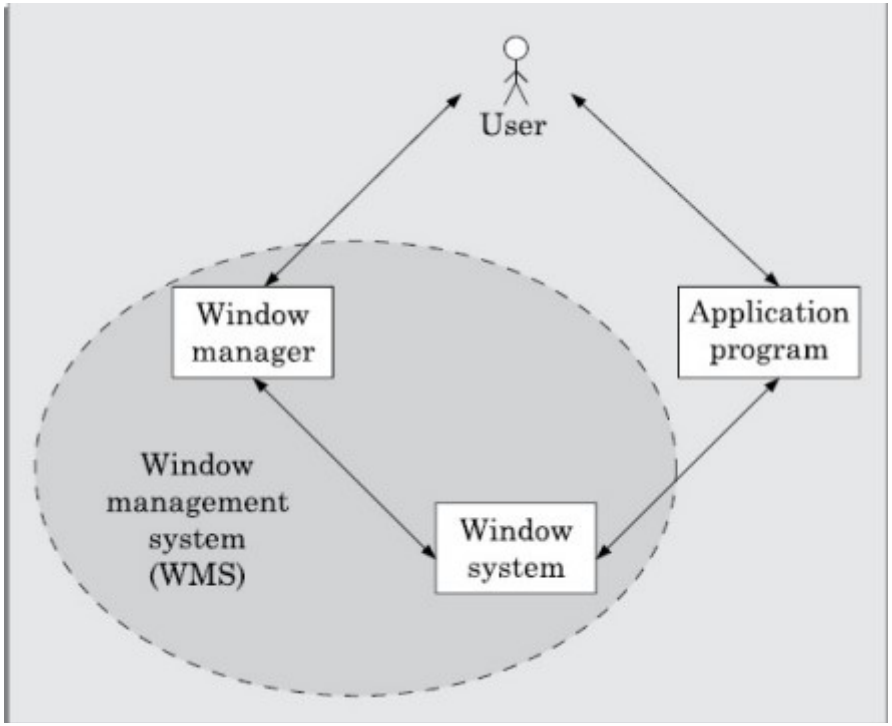
- Command language based interface: allows fast interaction with the computer, Easier to develop, LEX and YACC are used to develop it
- Difficult to learn, Most users make errors
- Menu based interface: major challenge is to structure large number of menu choices into manageable forms
- Techniques to structure menu items: Scrolling menu, Walking menu (submenu is displayed adjacent to a selected menu item), Hierarchical menu (current menu will be replaced by submenu, suitable for mobile phones)
- Direct manipulation (iconic) interface: user issues commands by performing actions on the visual representation of objects

# Component Based GUI Development

- Development style based on *widgets*. Window Objects are standard UI components
- UI is built from predefined components such as menu, dialog box, form

## Window system

- To carry out independent user activities
- Client part (for client application), non-client part (window manager)
- WMS: window manager + window system
- Window manager: determines the look and behavior of window
- It is the component of WMS through which end-user interacts to do operations such as window repositioning, resizing, iconification




## Advantages of using widgets

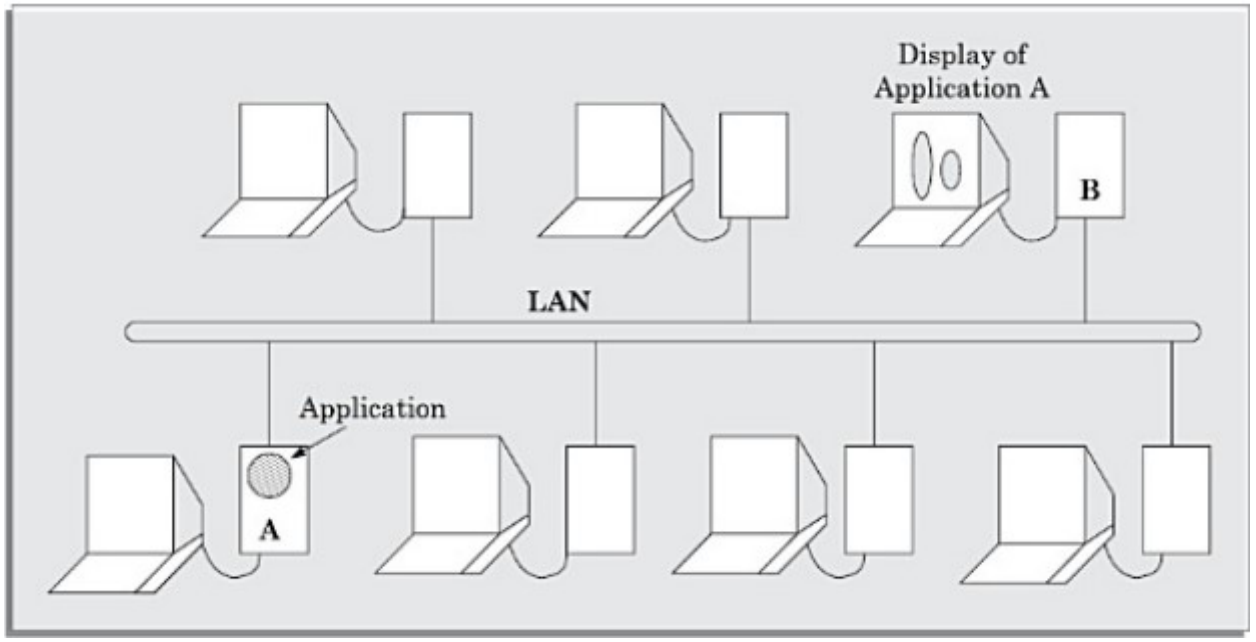
- User learns the interface quickly
- Programmer can use the same components in different applications
- Reduces the work of application programmer

## Visual Programming

- Program development through the manipulation of several visual objects (Drag & drop style)

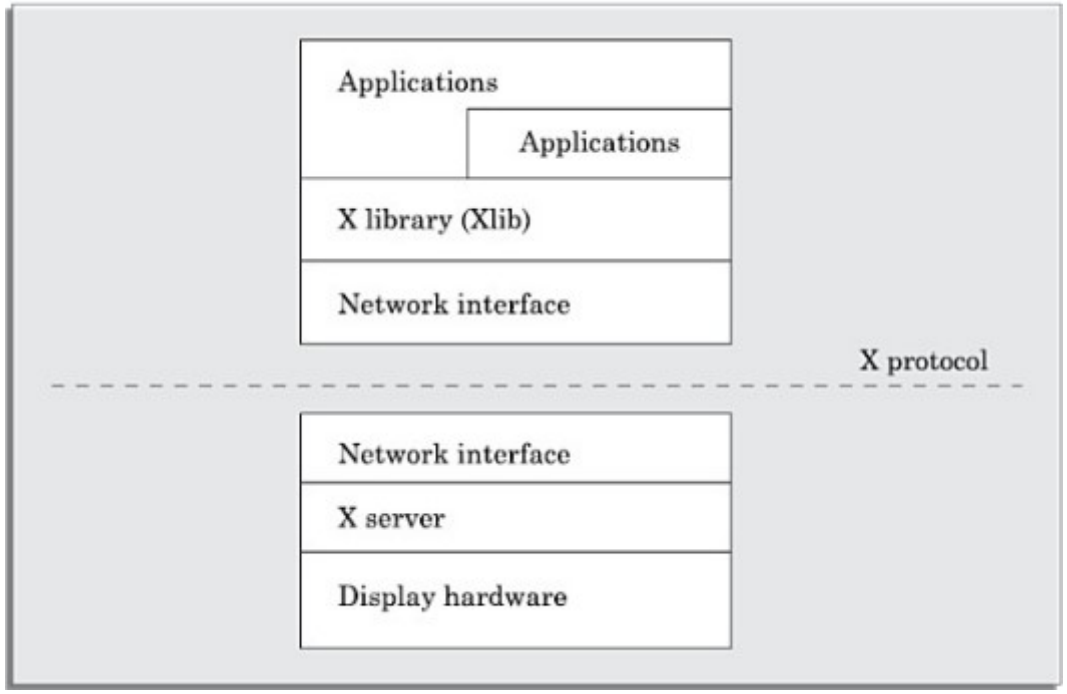
# Types of Widgets

- Label, Push button, Radio button, Container (accommodates other widgets), Dialog box, Pull-down menu (permanent menu)
- Pop-up menu: appears upon pressing the mouse button, irrespective of pointer position
- Combo box: to choose one item from a menu of many items
- X-Window: allows development of portable GUI
- Applications developed using X-window are device independent and network independent
- E.g. AWT and Swing components of Java




- Size of GUI is measured in widget points (wp) or number of screens
- wp: number of widgets used in the interface

# X Architecture




- Xserver: runs on the hardware in which display and keyboard are attached
- X protocol: defines the format of request between client applications and display server over the network
- X library: a set of 300 utility routines for applications to call. These routines convert procedure call into request -> transmitted to server
- Xtoolkit: intrinsics + widgets
- Intrinsics: set of library routines that allow a programmer to combine a set of widgets into a user interface

# UI Design Methodology

Things to remember while designing UI

- Limited memory: User need not remember too much information
- Frequent task closure: divide the task into subtasks and clear out information regarding the completed task
- Recognition of information from the alternatives is more acceptable than recall the information
- Procedural design focuses on tasks, prompting the user in each step of the task. An object oriented design focuses on objects. It allows the users a wide range of options

# GUI Design Methodology

- Examining the use case model: UI should be developed using one or more metaphors (white board, shopping cart, desktop, etc.)
- Task and object modeling: A *task model* should show the structure of subtasks that the user needs to perform to achieve the goal
- A user *object model* is a model of (ActiveX) business objects which the end-user needs to interact with
- Metaphor selection: It should be simple and fit with user' common sense
- Interaction design involves mapping the subtasks into appropriate controls
- At one extreme, all the views could be placed in its own window. At the other extreme, all the views could be placed in one window side-by-side
- No other window is accessible when a modal dialog window is active
- UI inspection: Inform the user about the status of the system, System should communicate in user's language, User should be able to undo and redo operations, Consistency in the use of words, concepts and operations in different operations, All data and instructions should be available on the screen for selection, Support for multiple skill levels, Dialog box and screen should not contain irrelevant information, Help and error messages, Prevent the user from entering wrong values

# Coding

- Goal: implement the design. Try to reduce cost of the later phases
- Coding (less cost) affects both testing and maintenance (costly)
- Write code quickly that are easy to read, modify and understand
- Programming is a skill acquired by practice. It is independent of programming language, although well structured programming languages make the programmer's job simpler
- Structured programming: "goto-less" programming
- A program has a static structure (text of the program) and a dynamic structure (execution sequence of statements)
- Sequence in static structure of a program is fixed, while it will differ from execution to execution in dynamic structure
- Closer the corr. b/w execution and text structure, program is easy to understand. Goal is to ensure that the static structure and dynamic structure are same
- State the i/p conditions in which program is to be invoked – precondition of the program
- Expected final state of the program is called post condition of the program
- Program verification: determine precondition for which post condition will be satisfied

# Cont…

- Linearizing the control flow by using control structures. Selection, iteration, sequencing
- *Key property*: single-entry, single exit
- Information hiding: information captures in the data structures should be hidden from the rest of the system, and is visible only to the access functions in the data structure. Rest of the modules in the s/w use these functions to access and manipulate data structures
- It is an effective tool to manage the complexity of developing s/w
- Make the system more maintainable.

# Programming Practices

- Control constructs
- Gotos
- Information hiding
- User defined types
- Nesting
- Module size
- Module interface
- Side effects
- Robustness: a program is robust if it does something planned even for exceptional conditions

# Cont…

- Switch case with default
- Empty catch block: an exception is caught, but if there is no action, it may represent a scenario where some of the operations to be done aren’t performed
- Empty if, while statement
- Read return to be checked
- Return from finally block: should be avoided
- Correlated parameters
- Trusted data sources: check i/p data if it is from a user or network
- Give importance to exceptions

# Coding Standards

- Provides rules and guidelines for programming in order to make the code easier to read

## Naming conventions

- Package names should be in lowercase
- Type names should be nouns starting with uppercase
- Variable names should be nouns starting with lowercase
- Constant names should be in uppercase
- Method names should be verbs starting with lowercase
- Private class variables should have _ suffix
- Variables with large scope should have long names; small scope – short names; loop variables - i, j, k, etc.
- Prefix *is* with boolean methods and variables. Avoid negative boolean variable names (isNot)
- The term *compute/find* can be prefixed with methods related to that operation
- Exception classes should be suffixed with *Exception*

# Cont…

- Files: File extension .java
- Each file should contain one outer class and the class name should be same as the file name
- Line length should be limited to 80 columns, avoid special characters
- Comments: should explain what the code is doing or why the code is there
- Prologue (comments for a module) describes functionality and purpose of the module, its public interface and how the module is to be used, parameters of the interface, assumptions it makes about the parameters, and any side effects it has
- Single line comments for a block of code should be aligned with that block
- There should be comments for all major variables
- Block of comments within /* and */
- Trailing comments after statements should be short

# Cont…

- Statements: declare variables with smallest possible scope, initialize with declaration
- Declare related variables in a common statement
- Class variables shouldn’t be declared as public
- Avoid complex conditional expressions
- Layout: how a program should be indented, how it should use blank lines, white space, etc. to make it more readable

# Coding Process

- Incremental coding process: write code for implementing part of the functionality of the module. Compile and test the code. If it is success, add further functionality to it
- Adv: detect error as and when a new functionality is added
- Test driven development: a new approach used in extreme programming (XP)
- A programmer first writes the test scripts and then writes the code to pass the tests. Tests being written based on specifications. The whole process is done incrementally
- First few test cases are likely to focus on main functionality. Code for higher priority features will be developed earlier
- It is the task of test cases to check that the code contained all the required functionality
- Reduce interface error
- It is tedious to write test cases for all scenarios or special conditions
- Pair programming: written by a pair of programmers – one person will type the program while other will review it
- DisAdv: loss of productivity (2 people assigned to a task), issue of accountability and code ownership when pairs aren't fixed

# Unit Testing

- For code (of the module) verification
- A unit may be a function / a small collection of functions for procedural languages or a class / a small collection of classes for OO languages
- Programs are executed with some test cases. If the behavior isn't as expected, programmer finds the defect in the program and fixes it (debugging)

Approaches

- Testing procedural units: behavior of the module depends on the value of its parameters as well as the overall state of the system
- If a module call other modules (i.e. a module has other modules below it in structure chart):-
  - Bottom-up approach: first test the modules at the bottom of the structure chart and then move up
  - Write stubs: stubs are throwaway code written for the called functions to facilitate testing of the caller function

# Cont…

- Testing of a module involves:- set system state as needed by the test case
- Set value of parameters
- Call procedure with parameters
- Compare results of the procedure with expected results
- Declare whether the test case has succeeded or failed
- The test suit succeeds if all the test cases succeed. If a test case fails, then the test framework will decide whether to stop or continue execution
- *Testing frameworks*: CuTest, Cunit, Cutest, Check
- Unit testing of classes: create object of the class, take object to a particular state, invoke a method on it, and check whether the state of the object is as expected. Do it for all methods
- *Framework*: Junit

# Code Inspection

- Proposed by Fagan
- Applied at unit level
- Goal: To improve quality (efficiency, compliance to coding standards, etc.) and productivity
- Static testing – detect defects not by executing the code but through a manual process

Characteristics

- Conducted by programmers for programmers
- A structured process with defined roles for the participants
- Focus is on identifying defects (review of code), not fixing them
- Inspection data is recorded and used for monitoring the effectiveness of the inspection process
- Moderator of the review team has to ensure that the review is done in a proper manner and all steps in the review process are followed

# Stages

- Planning: prepare for inspection – form the inspection team which should include a moderator and author of the code
- Author ensures that code is ready for inspection and entry criteria are satisfied
- *Entry criteria* – code compiles correctly and the available static analysis tools have been applied
- A package consisting of the code, specifications for which code was developed, and a checklist is prepared and distributed to the inspection team
- Author may give a brief overview of the product and the special areas to be noticed during the opening meeting
- Self review: by each reviewer in a continuous time span of < 2 hours
- Self preparation log: go through the code and logs all defects and mark them on the work product itself. Also prepare a summary of self review and the time spent

# Cont…

- Group review meeting: *purpose* – come up with final defect list based on the initial list of defects reported by the reviewers and the new ones found during discussion in the meeting
- *Entry criterion* – moderator is satisfied that all the reviewers are ready for the meeting
- *o/p* – defect log and defect summary report
- Team (*reader*) goes through the code line by line. If any reviewer raises an issue, there will a discussion on it. The author accepts the issue as a defect or give clarification to it.
- Team (*scribe*) records the defects in the log. At the end of the meeting, scribe reads out the defects for final review
- Review team identify the defects; author should have a solution for them
- Summary report: describes the code, total effort spent and its breakup in different review process activities, type, size and number of defects found for each category
- If defects are few, group review status is *accepted*, otherwise, re-review might be necessary
- Recommendations of the moderator is also included

# Testing

- Final code is likely to have requirement errors and design errors
- Two approaches for identifying defects in the software - static and dynamic
- Static analysis: code inspection or code is evaluated using tools
- Dynamic analysis: code is executed to determine the defects (testing)
- Software under test (SUT) is executed with a finite set of test cases. Effectiveness and efficiency of testing depends critically on the test cases selected
- *Error* refers to the difference between the actual o/p of a software and the correct o/p
- Error also refers to the human action that result in defects in the software
- *Fault* is a condition that causes a system to fail in performing its required function
- *Failure* is the inability of a system or component to perform a required function according to its specifications

# Cont…

- Presence of an error implies that a failure must have occurred, and the observance of a failure implies that a fault must be present in the system
- If we observe some failure, we can say that there are some faults in the system
- Testing is a quality control activity
- Test suite: A group of related test cases that are generally executed together to test some specific behavior or aspect of the SUT
- Test i/p and execution condition: add a record which is already existing to a database
- Automated testing: a test case is a function call which sets the test data and test conditions, invokes the SUT as per test case, compares the results returned with expected results, and declares to the tester whether the SUT failed or passed the test case
- Test framework (test harness) makes the life of a tester simpler by providing easy means of defining a test suite, executing it, and reporting the results

# Cont…

- Testing is a destructive process; conduct testing in order to show that a program doesn't work
- Independent testing: testing and development are done by different groups

Levels of testing

- Unit testing: done by the programmer for verification of the *code*
- Integration testing: testing the *design*
- Many unit tested modules are combined into subsystems and then tested
- Emphasis is on testing interfaces between modules
- System testing: the entire software system is tested
- The goal is to see if the software meets its *requirements* as per SRS – a validation activity
- Acceptance testing is performed at the *client* site with live data
- Regression testing is performed when some changes are made to an existing system
- This is to ensure that both new and old services are performing as desired
- If a small change is made to a large system, regression testing will be done with only a subset of test cases

# Testing Process

- Test plan: A general document for the entire project that defines the scope, approach to be taken, schedule of the testing, test items and the testing personnel
- I/P: project plan, requirements document, architecture or design document
- Test planning can be done in parallel with coding and design
- Test plan should be consistent with the overall quality plan; testing schedule should match with that of project plan
- Testing is performed incrementally

Test plan contains

- A test unit should be easy to test; it should be possible to form meaningful test cases and execute the unit without much effort
- Features to be tested: functionality, performance, design constraints, attributes
- Approach for testing: testing criterion for evaluating the test cases
- Testing deliverables: list of test cases, detailed results of testing, test summary report, and data about code coverage
- Schedule, task allocation and tools to be used

# Cont…

## Test case design

- Test case specification gives the list of test cases, I/P to be used, conditions being tested, and expected O/P
- It helps the tester to see the testing of unit in totality and the effect of total set of test cases
- It also allows optimizing the number of test cases. Some test cases may be redundant
- Select the test cases that satisfy the criterion and approach specified
- Evaluation of test cases is done through test case review. Test case specification document is reviewed to make sure that the test cases are consistent with the policy specified in the plan, satisfy the chosen criterion, conditions are tested, and cover various aspects of the unit to be tested

## Test case execution

- A tester may find the defect while the developer fix it
- Defects found must be properly logged in a system and their closure tracked
- General life cycle of a defect – submitted, fixed, closed

# Black-box Testing

- Two approaches for designing test cases – black-box testing, white-box testing
- In black-box testing test cases are decided based on requirements or specifications of the program or module
- Tester knows about inputs and outputs only. This form of testing is also called functional or behavioral testing
- Exhaustive testing, randomly generate test cases – not practical/suitable

Techniques to select test cases …

- Equivalence class partitioning: Divide the input domain into a set of equivalence classes, so that if the program works correctly for a value then it will work correctly for all other values in that class
- An equivalence class is formed of the inputs for which the behavior of the system is specified or expected to be similar
- We should define equivalence classes for invalid inputs also
- If there is a reason that the entire range of input will not be treated in the same manner, then the range should be split into two or more equivalence classes

# Cont…

- Consider any special value for which the behavior could be different as an equivalence class
- Consider equivalence classes in the output. The goal is to have inputs such that the output for that test case lies in the output equivalence class
- Select test cases: Select each test case covering as many valid equivalence classes as it can, and one test case for each invalid equivalence class. OR
- Select a test case that cover at most one valid equivalence class for each input, and have one separate test case for each invalid equivalence class
- Boundary value analysis: A boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data
- Six boundary values: min-1, min, min+1, max-1, max, max+1
- Two strategies for combining boundary values for different variables
  Select the boundary values for one variable, and keep the other variables at nominal value (6n + 1 test cases)
- Try all combinations of values for different variables ($7^n$ test cases)

# Cont…

- Pairwise testing: To find double-mode faults, conduct test for all combinations of values for each pair of parameters
- Single-mode fault: Many of the defects in software generally involve one condition – some special of value of one of the parameters
- Combinatorial testing: Multi-mode faults can be revealed during testing by trying different combinations of parameter values ($n^m$ combinations – n parameters each having m values)
- Use algorithms to generate smallest number of test cases for pairwise testing
- Conduct software testing for products that are expected to run on different platforms/systems/configurations
- Special cases: Programs produce incorrect behavior for some combination of special inputs. E.g division by 0, empty file

# Cont…

- State-based testing: For identical inputs system behave differently at different times and may produce different outputs. The state of the system depends on the past inputs
- State represents the cumulative impact of all the past inputs on the system
- State space of a program is a combination of states each representing a logical combination of values of different state variables
- Any software that saves state can be modeled as a state machine

Components of a state model

- *States*: Represent the impact of past inputs to the system
- *Events*: Inputs to the system
- *Transitions*: Represent how the state of the system changes from one state to another in response to some events
- *Actions*: Outputs for the events
- State model shows what state transitions occur and what actions are performed in a system in response to events

# Cont…

- Once the state model is built, we can use it to select test cases. When the design is implemented, these test cases can be used for testing the code
- Making state model require detailed information about system design
- State-based testing comes under gray-box testing, i.e., in between black-box and white-box testing

Criteria to generate test cases (T)

- *All transition coverage (AT)*: T must ensure that every transition in the state graph is exercised
- *All transitions pair coverage (ATP):* T must execute all pairs of adjacent transitions (incoming and outgoing transitions from a state)
- *Transition tree coverage (TT):* T must execute all simple paths, where a simple path is one which starts from the start state and reaches a state that it has already visited in this path or a final state

# White-Box (Structural) Testing

- Testing the implementation of the program. Exercise different programming structures and data structures used in the program

Control flow-based testing

- Using control flow graph (CFG)
- A node in the graph G of program P represents a block of statements that is always executed together
- Start node, exit node (last statement in the block is an exit statement)
- An edge(i, j) from node *i* to node *j* is the transfer of control from the last statement in block (i) to block (j)
- Path, complete path (first node is the start node and the last node is the exit node)
- Statement coverage: Each statement of the program be executed at least once during testing. It required that the paths executed during testing include all the nodes in the graph (all-nodes criterion)

# Cont…

- Branch coverage (testing): Each edge in the CFG be traversed at least once during testing (all-edges criterion). i.e., each decision should be evaluated to true and false values
- Path coverage (testing): All possible paths in CFG be executed during testing (all-paths criterion)

Phases in generating coverage data

- Instrument the program with probes: Insert some statements called probes in the program by a preprocessor. Probes are used to generate data about program execution during testing that can be used to compute the coverage
- Execute the program (by the tester) with test cases
- Analyze the results of the probe data