Software design

Good software design should exhibit:

- Firmness: A program should not have any bugs that inhibit its function
- Commodity: A program should be suitable for the purposes for which it was intended
- Delight: The experience of using the program should be pleasurable one

Functional independence

- Cohesion is an indication of the relative functional strength of a module.
- Coupling is an indication of the relative interdependence among modules.
- Aspect is a representation of a cross-cutting concern.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure (sort algorithm)

User interface design

- Easy to learn?
- Easy to use?
- Easy to understand?

Typical Design Errors

- lack of consistency
- too much memorization
- no guidance / help
- no context sensitivity
- unfriendly

Golden rule

- Place the user in control
- reduce user's memory load
- make the interface consistent
  - Provide for flexible interaction (color, font, language, etc.)
  - Allow user interaction to be interruptible and undoable
  - Reduce demand on short-term memory (navigation)
  - Establish meaningful defaults
  - Hide technical internals from the casual user

INTERFACE DESIGN PRINCIPLES-I

- Anticipation
- Communication
- Consistency
- Controlled autonomy
- Efficiency

INTERFACE DESIGN PRINCIPLES-II

- Focus
- Fitt's Law
- Human interface objects
- Latency reduction
- Learnability

INTERFACE DESIGN PRINCIPLES-III

- Maintain work product integrity
- Readability
- Track state
- Visible navigation

Software testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user..

Testing Shows

- Error
- Requirements Conformance
- Performance
- An indication of quality

Who tests the software?

Developer

- Understands the system but, will test "gently“ and, is driven by "delivery“
- Experiencing the software operation (known to the developer)

Independent tester

- Must learn about the system, but, will attempt to break it and, is driven by “quality”
- Exploring the software operation (unknown to the tester)

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification refers to the set of tasks that ensure that software correctly implements a specific function/process.

Boehm states this another way:

- Validation: "Are we building the right product?“
- Verification: "Are we building the product right?"

Testing Strategy

- System engineering
- System testing
- Requirements
- Validation testing
- Degine
- Integration testing
- Code
- Unit testing

Black-box testing

- Focuses on the functional requirements of the software
- Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions

(2) interface errors

(3) errors in external database access (accessibility)

(4) behavior or performance errors

(5) initialization and termination errors

White-Box Testing: In structural testing, one primarily examines source code with a focus on control flow and data flow.

Control flow refers to flow of control from one instruction to another.

Data flow refers to the propagation of values from one variable or constant to another variable.

Black-Box Testing : In functional testing, one does not have access to the internal details of a program and the program is treated as a black box. A test engineer is concerned only with the part that is accessible outside the program, that is, just the input and the externally visible outcome.

Testing Strategic issues

- Product requirements in a quantifiable
- Users of the software
- Testing plan that emphasizes "rapid cycle testing
- Technical reviews as a filter prior to testing;
- Test strategy and test cases
- Continuous improvement

Software quality attributes

Software quality attributes

External Quality	Brief Description
Availability	The extent to which the system's services are available when and where they are needed
Installability	How easy it is to correctly install, uninstall, reinstall, and update the application
Integrity	The extent to which the system protects against data loss and corruption
Interoperability	How easily the system can interconnect and exchange data with other systems
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system runs before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application and its data
Usability	How easy it is for people to learn, remember, and use the system
Internal Quality	**Brief Description**
Efficiency	How efficiently the system uses computer resources
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions
Verifiability	How readily developers and testers can confirm that the software was implemented correctly

- **Maintainability :** Maintainability indicates how easy it is to correct a defect or modify the software
- **Testability** : Testability refers to the ease with which software components or the integrated product can be tested to look for defects

Function-Oriented Metrics

- This model can be used effectively as a means for measuring the functionality delivered by a system
- Function points are derived using an empirical relationship based on countable measures of software's information domain and assessments of software complexity

Information domain values are defined in the following manner:

- Number of external inputs (EIs): input transactions that update internal computer files
- Number of external outputs (EOs): transactions where data is output to the user, e.g. printed reports
- Number of internal logical files (ILFs): group of data that is usually accessed together, e.g. purchase order file
- Number of external interface files (EIFs): file sharing among different applications to achieve a common goal
- Number of external inquiries (EQs): transactions that provide information but not update internal file

software project estimation

software project estimation

The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

WHY?

- ❑ So the end result gets done on time, on budget, and with quality

project planning task set-1

- ❑ Establish project scope
- ❑ Determine feasibility
- ❑ Analyze risks
- ❑ Define required resources
    - ❑ Determine require human resources
    - ❑ Define reusable software resources
    - ❑ Identify environmental resources
- ❑ Estimate cost and effort
    - ❑ Decompose the problem
    - ❑ Develop two or more estimates using size, function points, process tasks (complexity),
    - ❑ Reconcile the estimates

- Develop a project schedule
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

Project estimation Principles

- Project scope must be understood
- Elaboration (decomposition) is necessary
- Historical metrics (pervious data) are very helpful
- At least two different techniques should be used
- Uncertainty is inherent in the process

Definition of COCOMO Model

The COCOMO (Constructive Cost Model) is one of the most popularly used software cost estimation models .It estimates the effort required for the project, total project cost and scheduled time for the project. This model depends on the number of lines of code for software product development.

1.Organic Project

It belongs to small & simple software projects which are handled by a small team with good domain knowledge and few rigid requirements.

Example: Small data processing or Inventory management system.

2. Semidetached Project

It is an intermediate (in terms of size and complexity) project, where the team having mixed experience (both experience & inexperience resources) to deals with rigid/nonrigid requirements.

Example: Database design or OS development.

3. Embedded Project

This project having a high level of complexity with a large team size by considering all sets of parameters (software, hardware and operational).

Example: Banking software or Traffic light control software.

Project scheduling

Why are projects late?

- ❑ Unrealistic deadline established
- ❑ Changing customer requirements
- ❑ An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job
- ❑ Predictable and/or unpredictable risks that were not considered when the project commenced
- ❑ Technical difficulties that could not have been foreseen in advance
- ❑ Human difficulties that could not have been foreseen in advance
- ❑ Miscommunication among project staff that results in delays

Scheduling principles

- ❑ compartmentalization—define distinct tasks
- ❑ interdependency—indicate task interrelationship
- ❑ effort validation—be sure resources are available
- ❑ defined responsibilities—people must be assigned
- ❑ defined outcomes—each task must have an output
- ❑ defined milestones—review for quality

Risk management

RISK : Project plans have to be based on *assumptions. Risk* is the possibility that an assumption is wrong. When the risk happens it becomes a *problem* or an *issue*

Reactive (hote pare)

- ❑ project team reacts to risks when they occur
- ❑ mitigation—plan for additional resources to reduce the severity of damages
- ❑ fix on failure—resources are found and applied when the risk strikes

Proactive (Na Hoar casta kora)

- ❑ formal risk analysis is performed
- ❑ organization corrects the root causes of risk
  - examining risk sources that lie beyond the bounds of the software (C=A/B)
  - developing the skill to manage change

Risk projection, also called risk estimation, attempts to rate each risk in two ways

- ❑ Probability: the likelihood or probability that the risk is real
- ❑ Consequences: the consequences of the problems associated with the risk, should it occur

The project planner, along with other managers and technical staff, performs four risk projection activities:

- Probability: establish a scale that reflects the perceived likelihood of a risk,
- Consequences: define the consequences of the risk,
- Impact: estimate the impact of the risk on the project and the product,
- Accuracy: note the overall accuracy of the risk projection so that there will be no misunderstandings.

Risk component & drivers (CATAGOREY)

- **Performance risk:** the degree of uncertainty that the product will meet its requirements and
  be fit for its intended use
- **Cost risk:** the degree of uncertainty that the project budget will be maintained
- **Support risk:** the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
- **Schedule risk:** the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time

Risk check list

- ❑ Product size (PS)
- ❑ Business impact (BU)
- ❑ Customer characteristics (CU)
- ❑ Process definition (PR)
- ❑ Development environment (DE)
- ❑ Technology to be built (TE)
- ❑ Staff size and experience (ST)

Why system fails?

- ❑ The system fails to meet the business requirements
- ❑ There are performance shortcomings in the system
- ❑ Errors appear in the developed system causing unexpected problems..
- ❑ Users reject the implemented system.
- ❑ Systems are initially accepted but over time become un-maintainable

Software development life cycle (SDLC)

- ❑ Requirements Analysis
- ❑ Designing/Modeling
- ❑ Coding /Development
- ❑ Testing
- ❑ Implementation/Integration phase
- ❑ Operation/Maintenance
- ❑ Documentation
- ❑ 60 to 70 percent of faults are specification and design faults