An Introduction of Software Engineering Concepts
By Dr. Maryam Khanian

Software Engineering:
A branch of computer science which uses well-defined engineering concepts to produce good software products

1

Software: executable codes + associated libraries + documentations

Engineering: developing products using well-defined (systematic, disciplined, quantifiable) scientific principles and methods.

Software Evolution: process of developing a software product using software engineering:

1. Requirement gathering
2. Create a prototype and get the user feedback
3. Production of the desire software after many iteration over the previous step
4. Update the existing software to match with latest requirements and advancing technology.

Software Categories (Lehman):

- **Static (S):** strict to primary specification and solutions (e.g. calculator program)
- **Practical (P):** procedures are determined but the solution is not obviously instant. (e.g. gaming software)
- **Embedded (E):** requirement of the real-world environment (high degree of evolution). (e.g. online trading software)

E-type Software evolution:(Lehman Eight Laws):

- **Continuing change,** to adapt with the real-world changes
- **Increasing complexity,** due to its evolution, some actions should be taken to reduce complexity
- **Conservation of familiarity,** with the knowledge about why and how the software is developed in this specific manner
- **Continuing growth,** (size of implementation)
- **Reducing quality,** unless rigorously adapted to the changing environment
- **Feedback systems,** multi-loop multi-level feedback system to be successfully modified
- **Self regulation:** able to perform automatic regulations
- **Organizational stability,** its global activity is invariant over the lifetime
-

Software Paradigms: methods and steps that are taken while designing the software:

- **Software Development**
  - **Requirement gathering**
  - **Software Design**

    - design
    - Maintenance
    - **Software Programming**
        - Coding
        - Testing
        - integration

Why software engineering:

- **Large software:** as the size of the software becomes large, engineering has to step to give it a scientific process
- **Scalability,**scientific engineering concepts make it easy to scale an existing software
- **Cost,** a proper engineering process can decrease the cost of the software production
- **Dynamic nature,** adaptation of the software with the changes is easier by having an engineering process
- **Quality concerns,** better engineering process provides better quality

Good software: how to measure a software quality
Quality should be considered in 3 different aspects:

- **Operational,** how well the software works on predetermined operations
- **Transitional,** how well it can be moved from one platform to another
- **Maintenance,** how adaptable it is to all type of the changes

2

Software Development Life Cycle (SDLC):
Steps to **design** and **develop** a software

- **Communication, User ←→ service provider** (initiates the **request for a software**)
- **Requirement gathering, developing team ←→stakeholders** to recognize **user**, **system** and **functional** requirements. (**previous software, interviews, database, questionnaires**)
- **Feasibility study, rough plan** of software process to check the feasibility of the software project, **financially, practically, and technologically** (for both sides).
- **System analysis,** decide the **roadmap** and the **software model**
- **Software design, logical/physical design of the product** based on the whole knowledge about requirements and previous analyses
- **Coding, programming phase (implementation)**
- **Testing,** early discovery of **errors** and their **remedy** (module testing, program testing, product testing, in-house testing, end-user testing)
- **Integration,** with necessary **libraries**, **databases** and **other programs** (outer world entities).

- **Implementation,** **installation** on **user machines** and doing post-installation **configurations**
- **Operations and maintenance,** **monitor the software operation** for more efficiency and less errors (user training, software updating, solving hidden bugs and real-world unidentified problems)
- **Disposition,** **retirement** of the software according to organizational needs, laws and regulations, and the Disposition plan.

Software Developments Paradigm (Methods/Strategies):

Tools, **methods**, and **procedures**, that define SDLC

- **Waterfall model,** simplest model, **linear manner,** one phase ends the other starts, **without undo/redo possibility** (good for **similar operations to previously done developments**)

- **Iterative model,** **cyclic manner** of development (each cycle includes its own **design**, **coding** and **test steps** as a small portion of the whole development process) (**high resource consumption**)

- **Spiral model,** **SDLC model + cyclic process (iterative model)**, **risk consideration**,

- **V - model (verification and validation model),** **testing** of software at **each stage** in **reverse manner**.

- **Big-Bang Model,** putting together lots of **programming** and **funds** to achieve the best software product with **less emphasis on planning**. (for **learning** and **experimenting**)

3

Software Project Manager (SPM)

- **undertakes the responsibility of executing the software project**
- **is thoroughly aware of all the phases of SDLC that the software would go through**
- **may never directly involved in producing the end product**
- **closely monitors the development process**
- **prepares and executes various plans**

- **arranges necessary and adequate resources**
- **Maintains communication among all team members**

Software Management Activities:

- **planning of project,** a set of multiple processes, which facilitates software production

- **deciding scope of software product,** the task of determining limited and quantifiable activities & processes need to be done in order to make a software product.

- **estimation of cost in various terms,** including software size, effort, time and cost estimation
    - Estimation techniques:
        - **Decomposition Technique**
            - **Lines of Code (LoC)**: Project estimation based on line of codes in the software product.
            - **Function Points (FPs):** Project estimation based on the amount of various functionalities that the final product provides to the user.
        - **Empirical Estimation Technique,** using empirically derived formulae to make estimation

- **scheduling of tasks and events**, determination of the roadmap of all activities to be done with specified order and within time slot allotted to each activity by:
    - Detection of smaller tasks that form each phase
    - Finding tasks correlations
    - Time estimation for each task
    - Determination of work units
    - Determination of work units for each task
    - Project total time calculation

- **resource management,** (**adequate resources**, **less**→ delay, **more** → cost problem)
  ⇒ need of **allocation/deallocation** process of:
    - human resource
    - productive tools
    - Software libraries
- **Risk Management**
    - **Identification**, to determine **possible risks**
    - **Categorization**, to determine **risk levels (low, medium, high)**
    - **Management**, **risk probability** estimation
    - **Monitoring**

- **Project execution & monitoring** (based on project plan) including:

    - **Activity monitoring**, (day-to-day basis)
    - **Status reports**, weekly completion of the activities
    - **Milestones checklist**, major tasks (milestones) completion once every few week

- **Project Communication Management**, **oral/written** bridges gaps between client and the organization and among the team members and other stakeholders such as hardware suppliers and includes:
    - **Planning,** identifications of all the stakeholders
    - **Sharing,** correct information with the correct person at the correct time to keep everyone up-to-date in his scope
    - **Feedback,** Using various measures and feedback mechanism to gather ideas
    - **Closure,** competence announcement of each major step among the stakeholders

- **Configuration Management, tracking** and **controlling** the **changes** in software (requirements, design, functions) (should be aware of cost and time overrun) containing:
    - **Baseline,** a measurement that defines completeness of a phase
    - **Change Control,** ensuring that all changes made to software system are consistent with organizational rules and regulations including:
        - **Identification,** of the change request
        - **Validation,** validation check of the change request
        - **Analysis,** impact analysis of the change request
        - **Control,** take approval of high authorities for effective changes
        - **Execution** of the change request
        - **Close request,** verification of the correct implementation of the change

Project Management Tools

- **Gantt Chart,** represents **project schedule** with respect to time periods
- **Program Evaluation & Review Technique (PERT) Chart,** depicts project as **network diagram**, graphically representing events in both parallel and consecutive ways, shows dependency of the later event over the previous one.
- **Resource Histogram,** representing number of resources (usually skilled staff) required for a project event
- **Critical Path Analysis,** helps to find out the **shortest** path or **critical path** to complete the project successfully

Project properties:

- **Unique and distinct goal**
- **Not routine/day-to-day operation**
- **Having start/end time**
- **Temporary phase (ends when its goal is achieved)**
- **Needs adequate resources**

Software Project:
The complete procedure of software development from requirement gathering to testing and maintenance.

Why Software Project manager (Me):
The **underlying technology** changes so **frequently and rapidly** that the experience of one product may not be applied to the other one.

Cost, **Time** and **Quality** management.

4

Software Requirements

Requirements: description of **features** and **functionalities** of the target system, **expectations** of users from the product with following properties:

- Obvious/hidden
- Known/unknown
- Expected/unexpected (from the client's point of view)

Requirement Engineering
gather the software requirements from **client**, **analyze**, and **document** them
Aim: develop and maintain sophisticated and descriptive 'System Requirements Specification' document
Steps:

1. **Feasibility Study**
    - do a detailed study about whether the desired system and its functionality are feasible to develop towards goal of the organization (**output:** *feasibility study report* - adequate comments and recommendations for management)
2. **Requirement Gathering**
    - communicate with the client and end-users to know their ideas
3. **Software Requirement Specification (SRS)**
    - a **document** created by **system analyst**, defines how the intended software will interact with the outside world such as hardware and the

external interfaces as well as system properties, security, quality, limitations etc.
    - **Conversion** of the requirements from **natural** language to **technical** language for the development team.
4. **Software Requirement Validation**
    - check if the user requirements are **practical**, **Valid** (in the domain of the software), **clear** (without ambiguities), **complete** (fully defined and understood) and **demonstrable**.

Requirement Elicitation (Extraction) Process

- Requirements **gathering** (from the user)
- **Organizing** Requirements (determining priorities)
- **Negotiation** & discussion (to eliminate ambiguities and conflicts)
- **Documentation**

Requirement Elicitation (Extraction) Techniques

- **Interviews** (closed, open, oral, written, one-to-one, group)
- **Surveys** (querying stakeholders about their expectation and requirements)
- **Questionnaires** (pre-defined set of objective questions)
- **Task analysis** (analyze the operation for which the new system is required)
- **Domain Analysis** (using the expert people in the domain)
- **Brainstorming** (An informal debate among various stakeholders)
- **Prototyping** (building user interface without providing detail functionality and noting the user feedback for a new cycle of requirement analysis)
- **Observation** (visiting the client's organization or workplace)

Requirements Characteristics

- **Clear** شفاف
- **Correct** صحیح
- **Consistent** سازگار
- **Coherent** هماهنگ
- **Comprehensible** قابل فهم
- **Modifiable** قابل اصلاح

- **Verifiable قابل اثبات**
- **Prioritized اولویت بندی شده**
- **Unambiguous بدون ابهام**
- **Traceable قابل ردیابی**
- **Credible source دارای منبع موثق**

Types of Requirements

- **Functional** (search options, type of reports, user groups, compatibility)
- **Non-functional** (security, configuration, performance, cost, recovery)

Logical classification of requirements (MoSCoW method)

- **Must have** (Critical)
- **Should have** (important but not necessary)
- **Could have** (desirable but not necessary)
- **Wish List** (optional)

User Interface requirements

- **Easy to use (attractive and clear)**
- **Quick in response**
- **Efficient error handler**
- **Simple and consistent**

Software System Analyst

A person, who analyzes the requirement of proposed system and ensures that requirements are conceived and documented properly and accurately.

Software Metrics and Measures

The process of **quantifying** and **symbolizing** various **attributes** and **aspects** of software (You cannot control what you cannot measure)

- **Size Metrics - Lines of Code**
- **Complexity Metrics - upper bound of the number of independent paths in a program represented in terms of graph theory concepts by using control flow graph**
- **Quality Metrics - Defects, their types and causes, consequence, intensity of severity**

- **Process Metrics - used methods and tools**
- **Resource Metrics - Effort, time, and resources used**

5

Software Design Output

- design documentation
- pseudo codes
- detailed logic diagrams
- process diagrams
- detailed description of all functional or non-functional requirements

Software design

The process of **transforming** the **user requirements (in natural language)** into some **suitable form for programmers (in technical language)**. The output of this process can directly be used into implementation in programming languages.

Software Design Levels

- **Architectural Design**, the **highest abstract version** of the system in which the software is presented as **a system with many components** interacting with each other.
- **High-level Design**, breaks the 'single entity multiple component' concept of architectural design into less-abstracted view of **subsystems** and **modules** and depicts their interaction with each other.
- **Detailed Design**, the **implementation part (modules and their implementations)** of what is seen as a system and its sub-systems in the previous two designs.

Module:

set of instructions put together in order to achieve some tasks

Modularization

The technique of dividing a software system into multiple discrete and independent modules that carry out tasks (be executed and/or compiled) independently with the following advantages:

- Easier maintain (of smaller modules)
- Problem division (divide and conquer)
- Bring desired level of abstraction
- Reusability of the components
- Parallel execution of different modules
- Better
- Security enhancement

Concurrency

Splitting the software into **multiple independent units** of execution, like modules and **executing them in parallel**.

Coupling and Cohesion

measures by which the **quality** of a **design** of modules and their **interaction** among them can be measured.

Cohesion پیوستگی و انسجام

A measure that defines the **degree of intra-dependability within elements of a module**.

((The greater the cohesion, the better is the program design))

Types of cohesion

- **Co-incidental cohesion** (**unplanned** and **random** cohesion)
- **Logical cohesion** (When **logically categorized elements** are put together into a module)
- **Temporal Cohesion** (when elements of a module are organized such that they are **processed at a similar point of time**)
- **Procedural cohesion** (When elements of module are grouped together, which are **executed sequentially** in order to perform a task)

- **Communicational cohesion** (When elements of module are grouped together, which are **executed sequentially and work on same data**)
- **Sequential cohesion** (When elements of module are grouped because the **output of one element serves as input to another** and so on)
- **Functional cohesion** (**the highest degree of cohesion**) (when all elements of a module **contribute to a single well-defined function**)

Coupling

A measure that defines the **level of inter-dependability among modules of a program** with the following levels:

- **Content coupling,** (When a module can **directly access or modify or refer to the content of another** module)
- **Common coupling,** (When multiple modules have **read and write access to some global data**)
- **Control coupling,** (if between two modules, one of them decides the function of the other module or changes its flow of execution)
- **Stamp coupling,** (When multiple modules ***share common data structure*** and work on different part of it)
- **Data coupling,** (when two modules interact with each other by means of **passing data**)

Design Verification

The early any mistake is detected, the better it is or it might not be detected until testing of the product.

By **structured verification approach**, reviewers can detect defects that might be caused by overlooking some conditions.

good design review ⇒ **good software design, accuracy, and quality.**

6

Software Analysis & Design tools

Software analysis and design

includes all activities, which help the transformation of requirement specification into implementation.

Requirement specifications
specify all functional and non-functional expectations from the software that come in the shape of human readable and understandable documents.

Data Flow Diagrams (DFD)
A **graphical representation** of **flow of data** in an **information system** in the form of

- incoming data flow
- outgoing data flow
- stored data

While the **flowchart** depicts **flow of control** in program modules.

DFD Types

- **Logical** (concentrates on the **system process**, and **flow of data in the system**)
- **Physical** (how the data flow is actually **implemented** in the system)

DFD Components

- **Entity** (**sources** and **destinations** of information **data**)
- **Process** (**Activities** and **action** taken on the data)
- **Data Storage**
- **Data Flow** (Movement of the data)

DFD Levels

- **Level 0** (depiction of the **entire information system as one diagram** concealing all the underlying details)
- **Level 1** (depiction of the **basic modules** in the system and flow of data among various modules and mentioning basic processes and sources of information)
- **Level 2** (data flows inside the modules)

Structure Charts

A **chart derived from Data Flow Diagram**. It represents the system in **more detail** than DFD that describes **functions** and **sub-functions** of **each module** of the system to a greater detail.

Structure Charts symbols

- **Module(s)**
- **Condition(s)**
- **Jump(s)**
- **Loop(s)**
- **Data Flow(s)**
- **Control Flow(s)**

Hierarchical Input Process Output (HIPO) Diagram

represents the **hierarchy of modules in the software system (decomposes functions into sub-functions in a hierarchical manner)** to **analyze the system** and provide the **means of documentation**. (**No data flow or control flow**)

Input Process Output (IPO) diagram

depicts the **flow of control and data in a module**

Structured English

Description of what is required to code and how to code it that helps the programmer to write error-free code

Pseudo Codes

An explanation that is written more close to programming language.

Pseudo code **avoids variable declaration** but they are written using some actual programming language constructs, like C, C++ and Java.

Decision Tables

represents **conditions** and the **respective actions** to be taken to address them, in a structured **tabular format**. It is a powerful tool to **debug** and **prevent errors** that can be created by following steps:

1. **Identification of all the possible conditions**
2. **Determination respective actions**
3. **Verification by the end-users**
4. **Simplification by eliminating duplicate rules and actions**

Entity-Relationship Model

A type of **database model** based on the **notion of real world entities** and **relationship among them** by defining a set of **entities** with their **attributes**, a set of **constraints** and **relation** among them.

Data Dictionary (metadata repository)

- is the **centralized collection** of **information about data:**
  - **meaning** and **origin** of data
  - **relationship** with other data
  - data **format**
- Should be **updated whenever DFD is changed**

Data Dictionary Contents

- **Data Flow**
- **Data Structure**
- **Data Elements**
  - Primary Name
  - Secondary Name (Alias)
  - Use-case (How and where to use)
  - Content Description (Notation etc.)
  - Supplementary Information (preset values, constraints etc.)
- **Data Stores** (Files and Tables**)**
- **Data Processing** (**Logical** user-related / **Physical** Software-related)

Software Design Strategies

Structured
conceptualization of problem into several **well-organized elements** of solution based on **divide and conquer** strategy

Function oriented
In function-oriented design, the system comprises of many **smaller subsystems** known as **functions**.

Object oriented
works around the **entities** and their **characteristics** instead of functions involved in the software system. (**objects, classes, encapsulation, inheritance, polymorphism**)

Top-Down Approach
takes the **whole software system** as **one entity** and then **decomposes it** to achieve more than one **sub-system or component** based on some characteristics. **Each sub-system** or component is then treated as a system and **decomposed further.** This process keeps on running until the **lowest level** of system (**non-decomposable elements**) in the top-down hierarchy is achieved.

Bottom-Up Approach
Starts with most specific and **basic components**. It proceeds with composing higher level of components by using basic or lower level components. It keeps **creating higher level components** until the **desired system** is not evolved as one single component.

Software User Interface (UI) Design

User Interface: the **front-end** application view that provides fundamental platform for **human-computer interaction** with the following types:

- Graphical
- Text-based
- audio-video based

Which should be:

- Attractive
- Simple
- Responsive
- Clear
- Consistent

And can be:

- A combination of hardware and software

Command Line Interface (CLI)

The **minimum interface** a software can provide to its users. It provides a **command prompt**, the place where the user types the command and feeds to the system. It is the first choice of many technical users and programmers with the following elements:

- Command prompt
- Cursor
- Command

Graphical User Interface (GUI)

Provides the user **graphical means** to **interact with the system**. GUI can be **combination of both hardware and software**. Using GUI, user interprets the software. Typically, **GUI** is **more resource consuming** than that of **CLI** with the following fundamental elements:

- Window(s)
- Tab(s)
- Menu(s)
- Icon(s)

- Cursor
- Dialogue Box(es)
- Text Box(es)
- Radio Button(s)
- Button(s)
- Check box(es)
- List Box(es)

Some GUI Implementation Tools

- **FLUID**
- **AppInventor (Android)**
- **LucidChart**
- **Wavemaker**
- **Visual Studio**

GUI Golden Rules

- Strive for consistency
- Enabling frequent users to use short-cuts
- informative feedback
- simple error handling
- easy reversal of actions
- Allow experienced users to be initiators of actions rather than the responders
- Reduce short-term memory load

9

Software Design Complexity

software complexity measures

- **Halstead's Complexity Measures**
  - thinks a program as **sequence of operators and their associated operands** and is computed directly from the **operators and operands from source code**, in static manner
- **Cyclomatic Complexity Measures**
  - It is **graph driven model** that is based on **decision-making constructs** of program such as if-else, do-while, repeat-until, switch-case and goto statements.
- **Function Point**
  - This method concentrates on functionality provided by the system. Features and functionality of the system are used to measure the software complexity through five parameters:
    - **External Input** (Every unique input to the system, from outside)
    - **External Output** (All output types provided by the system)
    - **Logical Internal Files** (files that hold logical (functional/control) data of the system)
    - **External Interface Files** (files containing shared data with some external software or used as parameter to some function)
    - **External Inquiry** (external user inputs and the response of the system to that inputs)

10

Software Implementation

Structured Programming
encourages the developer to use **subroutines and loops** instead of using simple jumps in the code, thereby **bringing clarity** in the code wich uses the following concepts:

Top-down analysis
Under top-down analysis, the **problem is broken down into small pieces** where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

Modular Programming
Jumps are prohibited and **modular format is encouraged**

Structured Coding
sub-divides the modules into further smaller units of code in the order of their execution.

Functional Programming
provides **means of computation** as **mathematical functions**, which **produces results irrespective of program state (the function always produce the same result on receiving the same argument)**. This makes it possible to predict the behavior of the program and uses the following concepts:

First class and High-order functions
functions that have this capability to **accept another function as argument** or they **return other functions as results**

Pure functions
Functions that do not affect any I/O or memory and can easily be removed

Recursion
a function calls itself and repeats the program code in it unless some predefined condition matches

Strict evaluation
evaluating the expression passed to a function as an argument:

- **Strict evaluation** always evaluates the expression before invoking the function
- **Non-strict evaluation** does not evaluate the expression unless it is needed

λ-calculus
Most functional programming languages such as Lisp, Scala, Haskell, Erlang, and F# use λ-calculus as their type systems. λ-expressions are executed by evaluating them as they occur.

Programming style
set of **coding rules** followed by all the programmers to write the code
An appropriate programming style includes using **function and variable names relevant to the intended task**, **using well-placed indentation**, **commenting code for the convenience of reader** and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease the documentation and updation.

Software Documentation
A well written document provides a great tool and means of information repository necessary to know about **software process**. Software documentation also provides information about **how to use the product**. A well-maintained documentation should involve the following documents:

- **Requirement** documentation
- **Software Design** documentation
- **Technical** documentation
- **User** documentation

Software Implementation Challenges

- **Code reuse:** including **compatibility checks** and deciding how much code to reuse

- **Version Management:** maintain **version and configuration related documentation**
- **Target-Host:** when the software program needs to be designed for host machines at the **customer side**.

11

Software Testing (Overview)

Validation
process of examining whether or not the software **satisfies** the **user requirements**.

Verification
process of confirming if the software is **meeting** the **business requirements**

Black-Box (Behavioral) Testing
In this testing method, the **design and structure of the code are not known** to the tester. The tester in this case, has a set of **input values** and respective **desired results**. On providing input, if the **output matches with the desired results**, the program is tested **OK**, and problematic otherwise.

White-Box (Structural) Testing
In this testing method, the **design and structure of the code are known** to the Tester and includes:

- **Control-flow testing:** to set up **test cases** which **covers all statements and branch conditions**
- **Data-flow testing**: It covers all the **data variables** and tests where the **variables** were **declared** and **defined** and where they were used or changed

12

Software Maintenance (Overview)

Software re-engineering

13

Software Case Tool (Overview)