Lecture 1

Software

A program written for the computer is known as software. But software is not just a program; many things other than the program are also included in software.

- Program
- Data
- Documentation

Uses of Software:

- Business decision making
- Modern scientific investigation and engineering problem solving
- Games
- Embedded Systems

Engineering

"The process of productive use of scientific knowledge is called engineering."

Software Crisis

In early 60s software had suffered from the similar kind of problem to which we call Software Crisis. Techniques that were used to develop small software were not applicable for large software systems. This thing resulted in the following consequences:

- In most of the cases that software which was tried to be build using those old tools and techniques were not complete.
- Most of the times it was delivered too late.
- Most of the projects were over-budgeted.
- And in most of the case systems build using these techniques were not reliable – meaning that they were not be able to do what they were expected to do.

Software Engineering

Software Engineering is the set of processes and tools to develop software. Software Engineering is the combination of all the tools, techniques, and processes that used in software production.

The Balancing Act

Software Engineering is actually the balancing act. You have to balance many things like cost, user friendliness, Efficiency, Reliability etc. You have to analyze which one is the more important feature for your software is it reliability, efficiency, user friendliness or something else.

What Is meant by "software does not wear out"?

Software does not run out of life like hardware. Hardwares can have failures, they have a life-period and then they need to be changed. Software, on the other hand, requires just some modifications and does not completely go out of order.

Lecture 2

Software Development

The process of creating and managing a software is called software development. It hasx two phases:

- Construction
- Management

Software Engineering Framework:

Quality Focus: As we have said earlier, the given framework is based on the organizational commitment to quality.

Processes: The processes are set of key process areas (KPAs) for effectively manage and deliver quality software in a cost-effective manner.

Methods: Methods provide the technical "how-to's" to carry out these tasks.

Tools: Tools provide automated or semi-automated support for software processes, methods, and quality control.

Steps of Software Development Loop:

Problem Definition: In this stage we determine what is the problem against which we are going to develop software

Technical Development: In this stage we try to find the solution of the problem on technical grounds and base our actual implementation on it'

Technical Development: In this stage we try to find the solution of the problem on technical grounds and base our actual implementation on it

Status Quo: After going through the previous three stages successfully, when we actually deployed the new system at the user site then that situation is called status quo.

Software Engineering Phases:

Vision: Here we determine why are we doing this thing and what are our business objectives that we want to achieve.

Definition: Here we actually realize or automate the vision developed in first phase. Here we determine what are the activities and things involved.

Development: Here we determine, what should be the design of the system, how will it be implemented and how to test it.

Maintenance: This is very important phase of software development. Here we control the change in system, whether that change is in the form of enhancements or defect removel.

- Boehm (1975) quotes a pathological case where the development cost of an avionics system was $30 per line of code but the maintenance cost was $4000 per instruction

Lecture 3

Requirement Engineering:

Requirements engineering (RE) is, as its name suggests, the engineering discipline of establishing user requirements and specifying software systems.

Definition of Software Requirements as per IEEE:

- A condition or capability needed by user to solve a problem or achieve an objective.
- A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
- A documented representation of a condition or capability as in 1 or 2.

-Boehm (1981) has reported that correcting an error after development costs 68 times more. Other studies suggest that it can be as high as 200 times.

Lecture 4

Risks from Inadequate Requirement Process:

Insufficient user involvement leads to unacceptable products.

If input from different types of users is not taken, the output is bound to lack in key functional areas, resulting in an unacceptable product. Overlooking the needs of certain user classes (stake holders) leads to dissatisfaction of customers.

Creeping user requirements contribute to overruns and degrade product quality.

Requirement creep is one of the most significant factors in budget and time overruns. It basically means identifying and adding new requirements to the list at some advanced stages of the software development process. The following figure shows the relative cost of adding requirements at different stages.

Ambiguous requirements lead to ill-spent time and rework.

Ambiguity means that two different readers of the same document interpret the requirement differently. Ambiguity arises from the use of natural language. Because of the imprecise nature of the language, different readers interpret the statements differently.

As an example, consider the following Urdu Phrase: “Rooko mut jane doo”.

Gold-plating by developers and users adds unnecessary features.

Gold-plating refers to features are not present in the original requirement document and in fact are not important for the end-user but the developer adds them anyway thinking that they would add value to the product. Since these features are outside the initial scope of the product, adding them will result in schedule and budget overruns.

Minimal specifications lead to missing key requirements and hence result in an unacceptable product.

As an example, let us look at the following requirement. The requirement was stated as: “We need a flow control and source control engineering tool.” Based upon this requirement, system was built. It worked perfectly and had all the functionality needed for source control engineering tool and one could draw all kinds of maps and drawings. The system however could not be used because there was there was no print functionality.

Incompletely defined requirements make accurate project planning and tracking impossible.

Levels of Software Requirements:

Business Requirements:

These are used to state the high-level business objective of the organization or customer requesting the system or product. They are used to document main system features and functionalities without going into their nitty-gritty details.

User Requirements:

User requirements add further detail to the business requirements. They are called user requirements because they are written from a user's perspective and the focus of user requirement describe tasks the user must be able to accomplish in order to fulfill the above stated business requirements.

Functional Requirements:

The next level of detail comes in the form of what is called functional requirements. They bring-in the system's view and define from the system's perspective the software functionality the developers must build into the product to enable users to accomplish their tasks stated in the user requirements - thereby satisfying the business requirements

Non-Functional Requirements:

Any other requirement except for the functional requirement like Availability, Reliability, Recoverability, Data Integrity, etc are included in non-functional requirements.

Stakeholder:

Stakeholders are different people who would be interested in the software.

- **Requirement Statement Characteristics – For MCQs**

- Requirement Specification Characteristics – For MCQs

<u>Lecture 5</u>

The Vision Statement:

The vision statement should reflect a balanced view that will satisfy the need of diverse customers. It can be somewhat idealistic but should be grounded in the realities of existing or anticipated customer markets, enterprise architectures, organizational strategic directions, and resource limitations.

Use Case Model:

- Use case model describes the interaction of the users and the system.
- Use case model describes what functionality does a system provides to its users.
- Use case model has two important elements actors and use cases.
- Actors are the one who directly interacts with the system.
- The use cases in the use case model are the interactions between the actors and the system. It can also be summarized as functionality that a system offers to its actor is a use case.

What is UML?

Unified Modeling Language (UML) is a standard software modeling language.

Lecture 6

Relationship among use cases

The UML allows us to extend and reuse already defined use cases by defining the relationship among them. Use cases can be reused and extended in two different fashions: extends and uses.

Elaborated Use Cases

Use Case Name: Delete Information

Priority: 3

Actors: User

Summary: Deleting information allows the user to permanently remove information from the system.

Preconditions: Information was previously saved to the system and a user needs to permanently delete the information.

Post-Conditions: The information is no longer available anywhere in the system.

Uses: Record Transactions, Cancel Action

Extends: None

Lecture 7

Source

A stakeholder describes requirements (needs, constraints) to be included as system functionality. These can be processes that generate certain information that the system may have to process or maintain. Sources of requirements are the origins from where the corresponding business process is initiated.

Sink

Sink is the consumer of certain information. It is that entity which provides a logical end to a business process. Thus, 'sinks of requirements' is a concept that helps in identifying persons, organizations or external systems that gets certain functionality from the system.

Sink and Source Analysis

In source and sink analysis the analyst determines all the sources of requirements and where do these requirements consume (sinks). Now evaluate a report which displays certain information, the source of this report is the data (and who enters it) that is input to be retrieved later in the form of the report. Similarly, whoever needs this report become the sink of the report.

Process Models

Domain Models

During requirements analysis phase, different models are developed to express requirements of the system. Though it is difficult to draw a line between these models as they complement each other, they differ in the manner information is expressed in these models.

Logical System Models

System models are techniques used to understand user needs and software engineer use these techniques in order to understand business domain. Software engineers develop diagrams to model different business processes.

Business Process Models

The first model that we will look at is called the process model. This model provides a high-level pictorial view of the business process. This model can be used as a starting point in giving the basic orientation to the reader of the document.

Lecture 8

State Transition Diagrams

State transition diagrams (STDs) are another technique to document domain knowledge. In many cases, information flows from one place to the other and at each place certain action is taken on that piece of information before it moves to the next place. A file in an office is example of such system. In this case, different people make comments and add information to that file and it moves from one table to the other.

A trouble report

From time to time all systems, including communications networks, develop problems or malfunctions referred to as "troubles". A "trouble" in a communications network is a problem that has an adverse effect on the quality of service perceived by network users.

Trouble report states and status

A trouble report may go through any of six states during its life cycle.

Queued

A trouble report is in a queued state when process on it has started but the trouble resolution process has not yet been initiated. A trouble report which is in the queued state may be cancelled by the manager.

Open/active

The trouble report becomes "open/active" when appropriate actions to resolve the trouble are initiated.

Deferred

This state indicates that corrective action to resolve the trouble has been postponed. This can occur when the faulty resource is inaccessible for a period and repair activity cannot proceed.

Cleared

A trouble report is moved by the agent to the "cleared" state when it determines that the trouble has been resolved.

Closed

This state indicates that the trouble resolution process is complete. Upon closure, the trouble report attributes are captured in a historical event generated at trouble report closure which may then be stored in a log of trouble history records, for future reference.

Disabled

A "disabled" value is exhibited when a trouble report's information cannot be updated due to local conditions. In the "disabled" condition only read operations can be performed.

Lecture 9 - Typical Processes

Data Flow Model

Captures the flow of data in a system.

It helps in developing an understanding of system's functionality.

What are the different sources of data, what different transformations take place on data and what are final outputs generated by these transformations.

The Notation

Process

What are different processes or work to be done in the system.

External Agent

External systems which are outside the boundary of this system. These are represented using the squares

Data Store

Where data is being stored for later retrieval. Provides input to the process. Outputs of the processes may be going into these data stores.

Data Flow

Where the data is flowing. Represents the movement of the data in a data flow diagram.

Lets take an example of a Bank

Processes

1. Open account
2. Deposit funds into an account
3. Pay a bill
4. Withdraw funds from an account

External agents

1. Bank
2. Creditor
3. Employee
4. Other income sources

Data stores

1. Monthly Account statement
2. Bank accounts
3. Account transactions

CRUD Operations

These are the processes done and shown in DFDs

Create: creates data and stores it.

Read: retrieves the stored data for viewing.

Update: makes changes in an stored data.

Delete: deletes an already stored data permanently.

Lecture 10 - Prototyping & GUI Design

GUI - Graphical User Interface

A graphics-based operating system interface that uses icons, menus and a mouse (to click on the icon or pull down the menus) to manage interaction with the system.

Why use GUI?

- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors.
- Poor user interface design is the reason why so many software systems are never used.

Disadvantages of GUI

- UIs distract from business process understanding (what) to interfacing details (how).
- Unstable requirements cause frequent modifications in UIs.
- An extra work to be done at the requirement level each time a GUI change has to be incorporated.

Prototyping

A prototype is not the real product. It is rather just a real looking mock-up of what would be eventually delivered and might not do anything useful.

Prototyping is yet another technique that can be used to reduce customer dissatisfaction at the requirement stage. The idea is to capture user's vision of the product and get early feedback from user to ensure that the development team understands requirements

From a user's perspective, it is easier to play with a prototype and try it out than to read SRS document.

Lecture 11 - Software Design

What is Software Design phase?

Recalling our discussion of software construction process, once the requirements of a software system have been established, we proceed to design that system. During the design phase, the focus shifts from what to how.

Software Design Process

Software design is not a sequential process. Design of a software system evolves through a number of iterations. The design process usually involves developing a number of different models, looking at the system from different angles and describing the system at various levels of abstraction.

Software Design Strategies

Software design process revolves around decomposing of the system into smaller and simpler units and then systematically integrates these units to achieve the desired results.

Two fundamental strategies have been used to that end. These are functional or structured design and object oriented design.

In the functional design, the structure of the system revolves around functions. The entire system is abstracted as a function that provides the desired functionality (for example, the main function of a C program). This main function is decomposed into smaller functions and it delegates its responsibilities to these smaller functions and makes calls to these functions to attain the desired goal.

The object-oriented design takes a different approach. In this case the system is decomposed into a set of objects that cooperate and coordinate with each other to implement the desired functionality.

Software Design Qualities

A software design can be looked at from different angles and different parameters can be used to measure and analyze its quality. These parameters include efficiency, compactness, reusability, and maintainability. A good design from one angle may not seem to be suitable when looked from a different perspective. For example, a design that yields efficient and compact code may not be very easy to maintain.

Maintainable Design

A maintainable design is the one in which cost of system change is minimal and is flexible enough so that it can be easily adapted to modify existing functionality and add new functionality.

In order to make a design that is maintainable, it should be understandable and the changes should be local in effect. That is, it should be such that a change in some part of the system should not affect other parts of the system. This is achieved by applying the principles of modularity, abstraction, and separation of concern.

Lecture 12 - 13

Concepts of Coupling & Cohesion

Coupling

Coupling is a measure of independence of a module or component. Loose coupling means that different system components have loose or less reliance upon each other.

Cohesion

Strong cohesion implies that all parts of a component should have a close logical relationship with each other. That means, in the case some kind of change is required in the software, all the related pieces are found at one place.

Coupling and cohesion are contrasting concepts but are indirectly related to each other. Cohesion is an internal property of a module whereas coupling is its relationship with other modules.

Abstraction and Encapsulation

Abstraction is a technique in which we construct a model of an entity based upon its essential characteristics and ignore the inessential details. The principle of abstraction also helps us in handling the inherent complexity of a system by allowing us to look at its important external characteristic, at the same time, hiding its inner complexity. Hiding the internal details is called encapsulation.

Function Oriented versus Object Oriented Design

Action-oriented paradigm focuses only on the functionality of a system and typically ignores the data until it is required. Objectoriented paradigm focuses both on the functionality and the data at the same time. The basic difference between these two is decentralized control mechanism versus centralized control mechanism respectively. Decentralization gives OO the ability to handle essential complexity better than action-oriented approach.

Object

The basic unit of object oriented design is an object. An object can be defined as a tangible entity that exhibits some well defined behavior.

Behavior is how an object acts and reacts in terms of its state changes and message passing. The behavior of an object is completely defined by its actions.

A message is some action that one object performs upon another in order to elicit a reaction.

The operations that clients may perform upon an object are called methods.

The Object Model

The elements of object oriented design collectively are called the Object Model. The object model encompasses the principles of abstraction, encapsulation, and hierarchy or inheritance.

Abstraction and encapsulation are complementary concepts. Abstraction provides the

outside view to the client and encapsulation prevents clients from seeing its inside view.

Association and Aggregation - Some basic differences

Objects do not exist in isolation. They rather collaborate with one another in many different ways to achieve an overall goal. The different types of relationships in which these objects are involved include association, aggregation, and inheritance. Briefly, inheritance denotes a “kind of” relationship, aggregation denotes a “part of relationship, and association denotes some semantic connection among unrelated classes.

Object Creation and Life Time

From the object creation and life time point of view, when an object is instantiated, all of its parts must also be instantiated at the same time before any useful work can be done and all of its part die with it. While in the case of association, the life time of two associated object is independent of one another.

Coupling and Linkages

As mentioned earlier, aggregation implies a much tighter coupling than association. In case of aggregation, the links between the whole and its part are permanent while in case of association the links may be maintained only just for the period an object requires the services of its associated object and may be disconnected afterwards.

Lecture 14 - 15

Object Oriented Analysis & The Notation

Object Oriented Design

OOD transforms the analysis model into design model that serves as a blueprint for software construction. OOD results in a design that achieves a number of different levels of modularity. The four layers of the OO design pyramid are

- **The subsystem layer**

Contains a representation of each of the subsystems that enable the software to achieve its customers defined requirements and to implement the technical infrastructure that supports customer requirements.

- **The class and object layer**

Contains the class hierarchies that enable the system to be created using generalization. The layer also contains design representations for each object.

- **The message layer**

Contains the details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.

- **The responsibility layer**.

Contains the data structures and algorithmic design for all attributes and operations for each object.

Object-Oriented Analysis using Abbot's Textual Analysis

The first object-orientation technique that we will study is one of the oldest techniques to identify objects and their relationships. This technique is called Textual Analysis. It was initially developed by Abbot and then extended by Graham and others. In this technique different parts of speech are identified within the text of the specification and these parts are modeled using different components.

The Notation

Many different notations are used for documenting the object oriented design. Most popular of these include, Rumbaugh, Booch, and Coad, and UML(Unified Modeling Language). We will be using UML to document our design.

Lecture 16-17

Derivation of the Object Model – The Coad Methodology

An object model of a system captures the static structure of a system by showing the objects in the systems, their relationships, their attributes, and their services. To stream line the derivation of the object model, Peter Coad has divided the process into 5 activities, each being further subdivided into a number of steps.

- **Select Objects – who am I?**

We have used an approach that divides the objects into different categories to make it easier to find them and establish their attributes, services, and collaborations.

Select actors

Actors are people and organizations that take part in the system under consideration. Examples of actors are: person, organization (agency, company, corporation, foundation). Note that we are talking about actors and not their "roles". e.g. a customer is a role that a person plays.

Select Participants

A participant is a role that each actor plays in the system under consideration. Examples of participants are: agent, applicant, buyer, cashier, clerk, customer, dealer, distributor, donor, employee, investor, member, officer, owner, policy holder, recipient, student, supervisor, supplier, teacher, worker.

Select Places

Places are where things come to rest or places that contain other objects. Examples of places are: airport, assembly-line, bank, city, clinic, country, depot, garage, hanger, hospital, plant, region, sales outlet, service center, shelf, station, store, warehouse, zone.

Select Transactions

Transactions are the "events" that must be remembered through time. These are entries that must be maintained in a historical record or log which may be used to answer questions or perform assessments

Select Container Objects

Containers are objects that hold other objects. Note the similarity of definition between container and places. The difference is that a place is a place in the literal sense while a container is a any object that can hold other objects, e.g. bin, box, cabinet, folder, locker, safe, shelf, etc.

Select Tangible things

Take a "walk" through the system and select "tangible" things around you used in the problem domain. These may be characterized as all the remaining (not yet selected) "nouns" that make up the problem domain.

- **Identify Structures**

A structure is a manner of organization which expresses a semantically strong organization within the problem domain.

There are two type of structures:

Generalization-Specialization (Gen-Spec) and whole-part

Identify Whole-Part structures (Aggregations) - What are my components?

For each object that you have identified, consider it as a whole and then try to find out its parts - objects that make up this object.

- **Define Attributes - What I Know?**

The first two activities would identify most of the objects (classes) in the problem domain. Now is the time to think about the role and responsibilities of these objects. The first thing to consider is their attributes, i.e., what it knows.

- **Show Collaborations (associations and aggregations) - Who I know?**

The second step in establishing each object's responsibility is to identify and show how this object collaborates with other objects, i.e., who it knows.

- **Define Services - What I do?**

The third and last step in establishing each object's responsibility is to define what services does each object in the problem domain provide, i.e., what it does. Putting the right service with the right object is also very important since any mistake in judgment will increase coupling and reduce cohesion.

Give a thorough read to lec 18-19 and understand it. No need to memorize anything from it.

Lecture 20-21

Interaction Diagrams

A series of diagrams can be used to describe the dynamic behavior of an object-oriented system. This is done in terms of a set of messages exchanged among a set of objects within a context to accomplish a purpose.

The purpose of Interaction diagrams is to:

- Model interactions between objects
- Assist in understanding how a system (a use case) actually works
- Verify that a use case description can be supported by the existing classes
- Identify responsibilities/operations and assign them to classes

The Sequence Diagrams

These diagrams illustrate how objects interacts with each other and emphasize time ordering of messages by showing object interactions arranged in time sequence. These can be used to model simple sequential flow, branching, iteration, recursion and concurrency.

The syntax used for naming objects in a sequence diagram is as follows:

- syntax: [instanceName][:className]
- Name classes consistently with your class diagram (same classes).
- Include instance names when objects are referred to in messages or when severalobjects of the same type exist in the diagram.

An interaction between two objects is performed as a message sent from one object to another.

Message Types

Sequence diagrams can depict many different types of messages. These are: synchronous or simple, asynchronous, create, and destroy.

1. **Synchronous Messages** are “call events” and are denoted by the full arrow. They represent nested flow of control which is typically implemented as an operation call. In case of a synchronous message, the caller waits for the called routine to complete its operation before moving forward.

2. **Asynchronous messages** are “signals,” denoted by a half arrow. They do not block the caller. That is, the caller does not wait for the called routine to finish its operation for continuing its own sequence of activities.

3. **Object Creation and Destruction** An object may create another object via a <<create>> message. Similarly an object may destroy another object via a <<destroy>> message. An object may also destroy itself. One should avoid modeling object destruction unless memory management is critical.

Collaboration diagrams

Collaboration diagrams can also be used to depict the dynamic behaviour of a system. They show how objects interact with respect to organizational units.

Collaboration diagrams have basically two types of components: objects and messages. Objects exchange messages among each-other.

Lecture 22

Software and System Architecture

Software and System Architecture

The design process for identifying the sub-systems making up a system and the

framework for sub-system control and communication is architectural design. The output

of this design process is a description of the software architecture.

The study of software architecture is in large part a study of software structure that began in 1968 when Edsger Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result.

David Parnas pressed this line of observation with his contributions concerning information-hiding modules, software structures, and program families.

Software Architecture Definitions

UML 1.3:

Architecture is the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

Garlan and Perry, guest editorial to the IEEE Transactions on Software Engineering, April 1995:

Software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

IEEE Glossary

Architectural design: The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

Importance of Architecture

Why is architecture important and why is it worthwhile to invest in the development of a architecture? Fundamentally, there are three reasons:

- **Mutual communication**

Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.

- **Early design decisions**

Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life.

- **Reusable abstraction of a system**

Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

Architectural Attributes

Software architecture must address the non-functional as well as the functional requirements of the software system. This includes performance, security, safety, availability, and maintainability.

Performance

– Performance can be enhanced by localising operations to minimise subsystem communication. That is, try to have self-contained modules as much as possible so that inter-module communication is minimized.

Security

– Security can be improved by using a layered architecture with critical assets put in inner layers.

Safety

– Safety-critical components should be isolated

Availability

– Availability can be ensured by building redundancy in the system and having redundant components in the architecture.

Maintainability

– Maintainability is directly related with simplicity. Therefore, maintainability can be increased by using fine-grain, self-contained components.

Architectural design process

Just like any other design activity, design of software architecture is a creative and iterative process. This involves performing a number of activities, not necessarily in any particular order or sequence. These include system structuring, control modeling, and modular decomposition.

System structuring

It is concerned with decomposing the system into interacting sub-systems. The system is decomposed into several principal sub-systems and communications between these sub-systems are identified.

Control modelling

Control modelling establishes a model of the control relationships between the different parts of the system.

Modular decomposition

During this activity, the identified sub-systems are decomposed into modules.