OOAD & Software Engineering (UE18CS353)
Unit 1

Aronya Baksy Vishnu Dixit

January 2021

1 Introduction

- Software engineering is the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of software. (*defn. acc to NATO conference of 1968*)
- Software engineering is the establishment and use of sound engineering principles in order to obtain economically, software that is reliable and works on real machines. (*defn. acc to IEEE Standard Glossary*).

1.1 Characteristics of Software Engineering

- *Concerns the development of large programs*: No clear distinction between large and small projects, but generally refers to multi-person and long-term development work of any software that consists of many interconnected components.
- *Mastering complexity is the central theme*: Complexity of software projects are generally due to vast number of details to be handled. Thus projects are split into smaller units (modules, components etc.) and each component is handled separately.
- *Software Evolves*: Evolution of software along with changing reality is a cost that has to be factored in during development.
- *Efficiency of SE is crucial*: Minimization of costs (labour, infra, time etc.) associated with software development is important, thus giving rise to efficient methods of development and maintenance (useful: tools/methods that allow reuse of components).
- *Co-operation between people*: Clear arrangements for work distribution, communication and responsibilities etc. Standards and procedures are used to enforce these arrangements and ensure discipline.
- *Effective user support*: Identify functional requirements, address usability, reliability, responsiveness, user-friendliness. Also documentation and manuals for creating the right envt for software to function.
- *Balancing act*: between initial requirements (which may not be static) and the resources available to the developer (in terms of expertise)

2 Phases of Software Engineering (SDLC)

- The Software Development Life Cycle (SDLC) consists roughly of the following phases: **requirements engineering**, **design**, **implementation**, **testing** and **maintenance**.

2.1 Requirements Engineering

- Goal: get a complete description of the problem statement and the requirements posed by and on the functional environment of the software.

- Important sub-phase here is feasibility study, which assesses the technical and economic feasibility of the solution to this problem.
- Problem statement description may include the following:
  - Functions of the software
  - Possible future extensions
  - Amount and kind of documentation needed
  - Performance requirements and constraints (eg: max response time)
- The output of this phase is a document called the **requirement specification**.

2.2 Design

- Creation of a model that aims to solve the problem.
- The first part of the design phase is *interface design* wherein the interaction between the software and its environment is described.
- The second part of design phase involves *architectural design*, wherein a global high-level description of the system is created. This involves the component breakdown, roles/responsibilities of each component, properties of each component and interfaces between components.
- The *detailed design* is the further decomposition of components into program units which are allocated some functional responsibility. Data structures and algorithms, data communication between components, state definitions and change of state properties etc. are all dealt with here.
- Output of the design phase is the **technical specification**

2.3 Implementation

- Direct translation between specification to code, or might include an additional phase of translation to high-level *pseudocode* first.
- Goal is to produce well-documented, reliable, easy-to-read, flexible and correct program. Emphasis over these as against efficiency using fancy optimizations.
- The overall structure that was created in the design phase might get lost in translation to programming. More modern languages allow one to retain this structure to a large extent.
- The output of this phase is an executable program.

2.4 Testing

- Testing most often does not follow implementation phase, rather it happens in parallel with implementation.
- **Verification**: Testing whether the transition between phase boundaries in the SD process is valid or not.
- **Validation**: Checking whether the current status of the software is in keeping with the user requirements specified in the requirement engineering phase.
- The output of this phase is the tested executable program.

2.5 Maintenance

- Activities that keep system operational after delivery to the user.
- This involves changes/enhancements to the user requirements post delivery, as well as correction of any errors that are not detected in the testing phase.

As per the *40-20-40 rule* 40% of effort is used up in the requirement engineering and design phases, 20% for implementation and 40% for testing.

3 Development Life Cycles

3.1 Software Development Life Cycle

- SDLC is a structured set of activities in a fixed order, generating intermediate and final results.
- Each activity (step) has a guiding principle that explains its end goal
- A product is the outcome of applying a process (i.e. this sequence of steps) on a project.
- Each phase has the following:
  1. Entry criteria
  2. Exit criteria
  3. Task to be done
  4. Person responsible
  5. Dependencies
  6. Constraints (scheduling, etc.)

3.2 Product Development Life Cycle

- A process that is responsible for bringing to market a new product and generally includes the business units.
- SDLC is more pointed towards solving software-specific problems. Hence SDLC is a subset of the PDLC, and the SDLC points to specific steps within the PDLC.
- A generic PDLC consists of the following phases:
  1. **Planning**: Involves requirement, scope gathering
  2. **Analysis**: Involves research and usability engineering
  3. **Design**
  4. **Implementation**
  5. **Testing and Quality Assurance (QA)**
  6. **Deployment**
  7. **Training and sales support**
  8. **Maintenance and technical support**

3.3 Project Management Life Cycle

- A high level description of processes involved in delivering a successful project.
- The PMLC consists of the following phases:
  1. **Initiation**: Involves developing a business case, identifying scope and stakeholders
  2. **Planning**: Workflow plan, budget plan, gather resources
  3. **Execution, Direction and Control**: Involves requirement engg, design, implementation and testing
  4. **Closure**: Project review (goals met or not), team review (team's performance), documentation, process improvement for future projects

3.4 Software Maintenance Life Cycle

- Structured process of implementing any changes to a delivered software.
- The SMLC is triggered by a change request from the client.
- After a change request is received, the steps in the SMLC are:
  - **Problem Identification**: Assign a unique number to the issue, add it to repository and accept/reject the request after validation.
  - **Analysis**: Technical review, feasibility study, identify potential side effects (security, performance etc.)
  - **Design**: Revise design, develop test cases, verify the revised design
  - **Implementation**: Implement revised design, review software, unit test
  - **System Testing**: System, acceptance and regression testing
  - **Delivery**

3.5 Product Life Cycle

- The phases of a product from the time it is introduced to consumers into the market until it is removed from the shelves.
- The phases of PLC are:
  - **Introduction**: Invest in marketing and advertisement
  - **Growth**: Growing demand, ramp up production, expand availability of the product
  - **Maturity**: Most profitable stage, marketing and production costs decline
  - **Decline**: Increased competition leads to reduced market share and decline in profits.

4 Legacy SDLC Models

4.1 Waterfall Model

- In figure 1, V&V stands for *Verification* and *Validation.*
- Before moving on to the next stage of the waterfall model, the previous stage undergoes both Verification (whether the software meets its own requirements) and Validation (whether the software meets the user requirements).

4.1.1 Use case of Waterfall Model

The waterfall model is suitable when:

- Requirements are clear, well-documented, fixed and unambiguous
- Definition of product is stable
- Technology is well understood, ample resources are available to support
- Project is short

4.1.2 Advantages

- Simple to understand and use, with well defined phases and milestones for each stage.
- Process and results are well documented.
- Easy to manage due to the inherent rigidity of the model.
- Each phase has specific deliverables and review process.

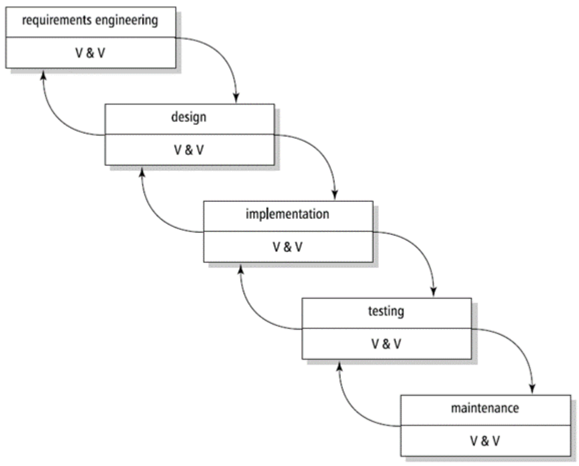

Figure 1: Waterfall Model

4.1.3 Disadvantages

- No working software is produced until late in the development cycle.
- High risk and uncertainty due to the rigidity of the model. This makes it hard to accommodate changing requirements and scope.
- Poor choice for long, complex, object-oriented projects
- Difficult to measure progress within stages
- Integration is done only at the end (in a big-bang manner), hence it is hard to identify technical or business challenges/bottlenecks early.

4.2 V Model

- For each phase in the development pipeline, there is a corresponding test phase. The corresponding development and test phases are *planned* (ie. designed) in parallel.
- The left side of the V indicates the Verification phases and the right side indicates the Validation phases.
- Verification phases:
  - **Requirement Analysis**: Get requirements from customer
  - **System Design**: Component design, communication and hardware setup for system
  - **Architectural Design**: Break down into modules, functionalities of each module as well as inter- and intra-module communication is detailed.
  - **Module Design**: Low level Design (LLD) of modules is specified here
- Validation Phases
  - **Unit Testing**: Elimination of bugs at the unit level or code level
  - **Integration Testing**: Verify communication of modules with each other
  - **System Testing**: Functionalities of the complete software along with all inter-dependencies and communication is tested here. Tests functional and non-functional requirements
  - **Acceptance Testing**: This is done in a real-life environment that resembles the actual production environment. This verifies that the system meets the requirements and is suitable for real-world use.

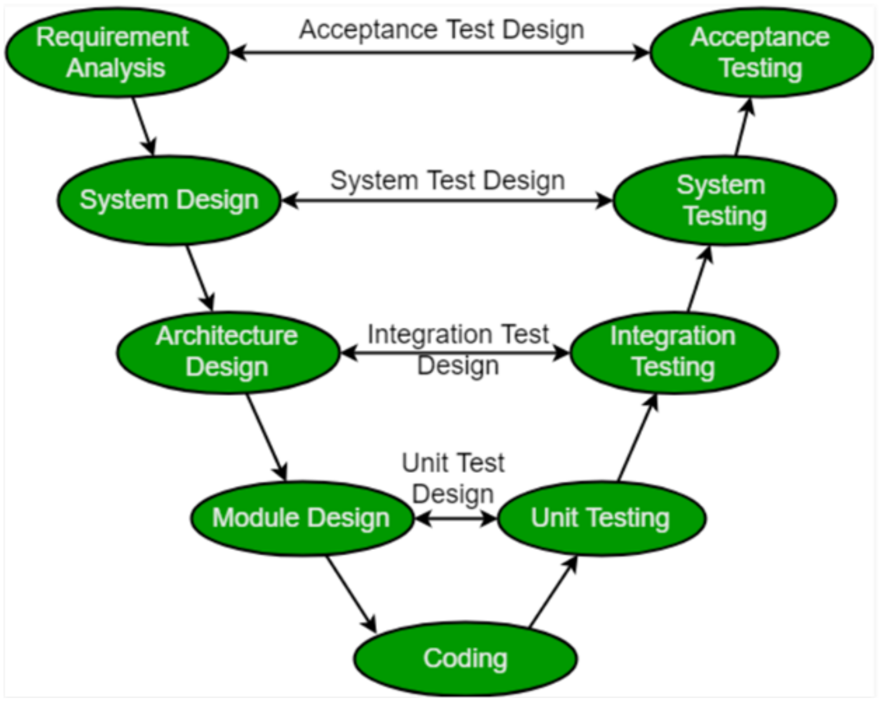

Figure 2: V Model

4.2.1 Advantages

- Focus on verification and validation activities early on in life cycle leads to error-free and better quality final product.
- More accurate progress tracking by project management
- Highly disciplined model with sequential phases

4.2.2 Disadvantages

- High risk and uncertainty, not suitable for dynamic user requirements
- Does not support iteration of phases, does not handle concurrent events
- Not good for object-oriented and complex projects
- No code is generated until the implementation phase hence no early prototypes are produced.
- Changes in verification stage need to be mirrored in the validation (testing) phase which leads to duplication of effort.

4.3 Prototyping Model

- This model is used when customer requirements are not well defined initially.
- A prototype (ie. an initial basic model of the system) is built and refined iteratively as per customer feedback until an acceptable prototype is built. This serves as the basis for the final software product.
- Types of prototyping activities:
  - **Rapid Throwaway Prototyping**: In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults.

- **Evolutionary Prototyping**: The initial prototype is incrementally modified on the basis of customer feedback until an accepted prototype is created. This saves time and effort in comparison to throwaway prototyping.
- **Incremental Prototyping**: The project is initially broken up into components, each component is prototyped individually and all components are integrated at the end. Leads to substantial cost saving but proper communication has to be ensured so that modules integrate properly at the end.
- **Extreme Prototyping**: Used in website design typically. Consists of 3 stages
  1. Basic prototype with existing static pages (wireframe)
  2. Functional screens that simulate a data process using a prototype service layer
  3. All services implemented and associated with final prototype

4.3.1 Advantages

- Code is generated early in the life cycle, meaning more opportunities to get customer feedback and refine the system.
- Easy to accommodate new requirements and add new features/missing features.
- Early error detection that reduces costs and effort, while improving software quality
- Prototypes once built can be reused for future projects

4.3.2 Disadvantages

- Costly wrt time and money, especially if user requirements vary a lot.
- Documentation suffers due to constantly changing requirements.
- Leads to increasing complexity if changes suggested exceed the original scope of the project.
- Performance of the resulting system may not be optimal

4.4 Incremental Model

- The functionality of the system is produced and delivered to the customer in small increments.
- Each subsequent release of the project adds a new functionality to the previous release.
- The 'Big Bang' effect is avoided in this approach as software is developed in a phased manner module by module, hence leading to more flexibility in terms of changing requirements.
- The process of development is split into partitions, where each partition corresponds to one release of the software. Each partition is planned using a model like the Waterfall model.

4.4.1 Advantages

- Early delivery to customer means greater value to customer.
- More flexible, less costly to change requirements and scope of project.
- Easy to manage risk, by handling the most risky components in conjunction with customer feedback.
- Reduces over-functionality, incremental model leads to development of essential modules first and additional modules later.

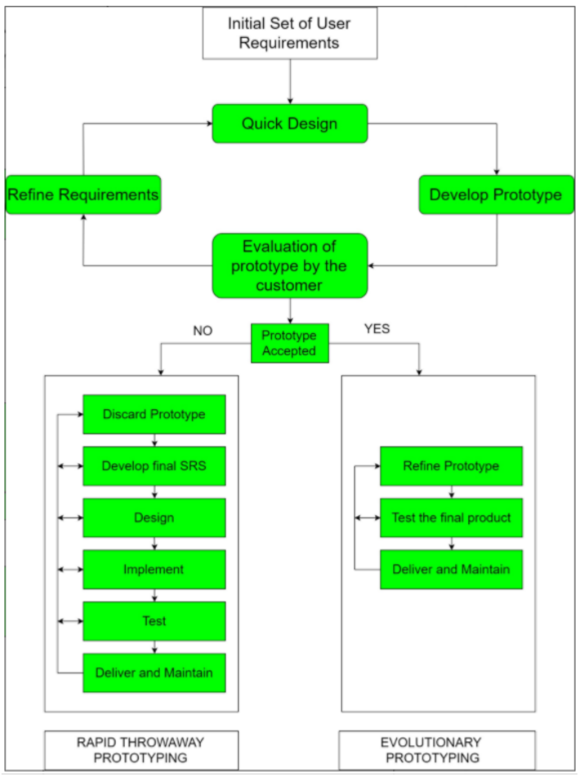

Figure 3: Prototyping Model

4.4.2 Disadvantages

- Good planning and design, needs complete and comprehensive definition of project before splitting into increments.
- Higher cost than equivalent waterfall model.
- Hard to identify facilities common to each increment.
- Management visibility is reduced which could lead to surprises

4.5 Iterative Model

- Initially a skeleton implementation, is refined with user feedback and iterative evolution to give a complete system.
- Dummy modules substitute for incomplete components,
- Rapid prototyping and successive refinement are the driving principles

4.5.1 Advantages

- Early visualization of solution
- Risk mitigation supported
- Effects of a defect flowing down the life cycle is reduced
- Incremental investment

4.5.2 Disadvantages

- Rigid with overlaps
- Costly system architecture or design issues may arise because not all requirements are gathered up front for the entire life cycle

4.6 Limitations of legacy models

- Require predictive development methods, upfront planning
- Suitable only for projects where requirements don't change often, and are well understood at the start
- Do not facilitate regular interaction with client

5 Agile and SCRUM

5.1 Agile

- Agile practices involve discovering requirements and developing solutions through the **collaborative effort** of **self-organizing** and **cross-functional** teams and their clients.
- Aims:
  - Deliver value to client as quickly as possible
  - Allow continuous realignment of development goals with customer needs
  - Reduce planning overheads

5.1.1 Advantages of Agile methodologies

- Rapid development and demonstration of functionalities, delivers partially working solutions faster (concurrent delivery and development in a planned context)
- Works when client's requirements are dynamic
- Minimal resource and planning requirements, easy to manage
- Promote teamwork, cross-training, flexibility to developers

5.1.2 Drawbacks of Agile methodologies

- Not suitable for complex dependencies
- Driven by client requirements, if client is not clear then team works in wrong direction
- Sustainability, maintainability and extensibility are more risky to implement
- Minimum documentation leads to high individual dependency and high cost of knowledge transfer.

5.2 Scrum

- Scrum is an Agile framework for developing, delivering and maintaining software products. It is an *iterative* approach
- Key ideas of Scrum:
  1. Iterative and incremental approach, encourages physical or close online collaboration of team members
  2. Requirements are not static
  3. Maximize the team's ability to deliver quickly, and adapt to technological change, competition and changing requirements.

5.2.1 Scrum Roles

- **Product Owner**: Represents the customer and the stakeholders,and is responsible for delivering business results.
- The product owner defines the product in terms of user stories, adds them to the Product Backlog, and prioritizes them based on importance and dependencies.
- **Scrum Master**: Not a traditional team lead or manager, instead focus on removing obstacles towards successful goal delivery.
- Scrum Master ensures that Scrum principles are followed, and facilitates team meetings.
- **Developer**: build increments of valuable work after every sprint. Can be anyone involved in R&D, statisticians, programmers and testers.
- Developers are also referred to as scrum team members. A team of developers is self-organized and cross-functional.

5.2.2 Scrum Artifacts

- The **product backlog** is a description of the product in the form of features told from the user's perspective (user stories)
- The scrum team estimates the work associated with each story and ranks them in the order of importance
- The weighted-ranked features/stories in the backlog results in **roadmap**
- The features/stories in the backlog planned for a sprint is the **sprint backlog**

5.2.3 Scrum Events

- The **Split** is the basic unit of development time. Commonly kept at 2 weeks, it is a *sandboxed* effort (the time is pre-planned and fixed).
- **Sprint Planning** involves deciding the scope of a sprint, prepare a sprint goal and a sprint backlog.
- A **Daily Scrum** meeting involves discussing work done yesterday, work to be done today and obstacles to achieving the sprint goal.
- A **Sprint Review** is held at the end of each sprint. It involves reviewing work done and not done as planned in the sprint, a *demo* of the completed work and collaborating with stakeholders on what to work on next.
- A **Sprint Retrospective** is also held at the end of each sprint. The team reflects on the past sprint, identifies and agrees on continuous process improvement actions
- The retrospective is attended by the Scrum master.

6 Component-Based Software Engineering

6.1 CBSE

- A reuse-based approach to define, implement or select off-the-shelf components and integrate these loosely-coupled components into functioning systems.
- Essential properties of CBSE are:
  - Independent components that are completely specified by the public interfaces
  - Component standards that facilitate integration
  - Middleware that provides support for integration
  - Development process geared towards CBSE

- A component is an independent executable entity that implements a functionality while hiding its implementation details.
- All interactions with the component are made via the published interface.
- Required dependencies and published interfaces are indicated as part of a component.
- Component selection involves:
  - Select existing components after searching
  - Compose existing components using direct connection or adapters for different interfaces.
  - Validation
- Specification describes properties to be realized: realization contract, Interface describes how components interact: usage contract
- CBSE model consists of the following:
  - Interfaces: defines interaction, and names, parameters, exceptions to be included in the interface definition
  -
- Advantages of CBSE:
  - Reduces complexity of systems as components are black boxes
  - Reduced development time due to reuse, increased productivity.
  - Increase in quality, maintainability (explicit dependencies, reuse)
- Drawbacks of CBSE:
  - Component trustworthiness, certification
  - Requirement trade-off
  - Emergent property prediction

6.2 Service-Oriented Architecture

- Structured collections of software modules, known as services, that collectively provide the complete functionality of a large application.
- Consists of:
  - **Infrastructure Service Layer**: Supporting activities such as data storage, databases etc.
  - **Business Service Layer**: Business logic modules deployed in the form of individual containers
  - **Orchestration Layer**: Manages and co-ordinates the containers deployed.
  - **System Bus** for communication between layers

6.3 Product Lines

- Software product lines refers to techniques for creating a collection of similar software systems from a shared set of software assets
- A top-down, planned, proactive approach to achieve reuse of software within a family of products.
- Exploits commonality and variability between requirements of different product lines, for increased customization.

6.3.1 Key Drivers

- Predictive rather than opportunistic (ad-hoc) reuse
- Artifacts are created when reuse is predicted in one or more products in a well defined product line
- These artifacts are either reusable components or design patterns that compose multiple components for a particular solution.

7 Requirement Analysis

- The first step of any software project. The most error-prone and costly stage of the project.
- Errors in this phase propagate and are hard to fix in later stages.
- Requirement is the property which must be exhibited by software to solve a particular problem (more focus on what and not how)

7.1 Properties of a requirement

- **Clear**: Precise, simple language, with active present tense, consistent terminology and keep requirements separate
- **Concise**: Explain only one property in fewest possible words
- **Consistent**: Requirements should not contradict one another
- **Unambiguous**: Only one interpretation
- **Feasible**: Can be realized within a specific timeframe
- **Traceable**: Back to client request, and forward to the software interface
- **Verifiable**: Has a testable criterion and a cost-effective to verify that it has been achieved
- **Quantizable**: Can be quantized, easier to verify
- **Prioritized**: Each requirement has a priority

7.2 Feasibility Study

- In discussion with the clients
- Identify current solution, target customer, future market place
- Potential benefits
- A high level scope and understanding of the solution
- Technological, marketing and financial considerations
- Issues, Assumptions, Risks and Constraints
- High level planning and budget requirements, project organization
- Alternatives and their consideration
- The end result is a go/no go decision

7.3 Steps in Requirement Engineering

7.3.1 Elicitation

- Gathering needs of stakeholders, communicating their problems.
- Establish clear scope and boundary for project.
- Understand the domain, the problem, the user needs, constraints. Identify the business objectives
- **Active Elicitation**: Ongoing and frequent interaction between client and stakeholders. Involves the use of interviews, prototyping, role-playing, scenarios and ethnography
- **Passive Elicitation**: Infrequent interaction between users and stakeholders. Involves the use of use cases, workflows, documentation, checklists, questionnaires, business process analysis

7.3.2 Analysis

- Understand requirements from product and process perspective
- Classify requirements into coherent clusters, as functional/non-functional/domain requirements and system/user requirements
- Model the requirements (informal: prose, formal: flowchart, pseudocode, ER Diagram, dynamic models, UML based use-case diagram)
- Analyze requirements using fish-bone diagram (cause-effect)
- Recognize and resolve conflict (functionality vs timeline vs cost)
- Negotiate requirements, prioritize them using MoSCoW rule (**M**ust Have, **S**hould have, **Co**uld have, **W**on't have)
- Identify risks
- Decide to build or buy (COTS) and refine requirements

7.3.3 Specification

- Document the requirements and give direction to the rest of the phases in the development life cycle.
- The SRS (Software Requirement Specification) is the basis for customers and suppliers agreeing on what the product will and won't do. It describes both the functional and nonfunctional requirements
- Documentation leads to more clarity, visibility, maintenance and evolution, team communication
- Document must be accurate, up-to-date, maintained online, simple and professional in appearance
- The SRS defines:
  - **Functional Requirements**: Exact functionalities of the system
  - **Non-functional Requirements**: All quality criteria that drive the functionality, such as performance (response/recovery time), availability, portability etc.
  - **External Interfaces**: Interaction of the software with external hardware, software and people
  - **Design Constraints**: Implementation language, standards and guidelines, resource limits, database integrity constraints, security and operating envt. etc.
- The **IEEE 830** template (1998) is a widely-accepted SRS template.

7.3.4 Validation

- Essential because repairing requirements in later phases is very expensive.
- Verification: Determines whether the requirement is specified correctly.
- Review: for correctness, completeness, verifiability, traceability, and consistency.
- Prototyping: allows engineers and users to be involved in requirement engineering process, useful mostly for systems with high level of user interaction
- Validation: determines whether the user needs are satisfied by the requirement and whether the right problem is solved
- Usage of the Fish Bone Analysis technique to validate if the requirements identified is addressing the reasons needing the solution to the problem which had led to the requirements
- Acceptance Testing

7.3.5 Requirement Management

- Ensure that all requirements are handled at each phase of SDLC, and requirement changes are handled appropriately.
- The Requirement Traceability Matrix (RTM) is filled out at each stage of the SDLC, and allows for forward and backward tracing of requirements.

7.3.6 Change Management Process

- **Log** the change request (who, why, what to change), assign a change request identifier.
- **Impact analysis** on schedule, quality, effort
- Solicit **Formal approval** from customer
- **Log** what changes were made, who made changes, when they were made, who reviewed and who tested changes, and the release stream that contains this change.

8 Project Management

- Planning and control of large projects that involve lots of people working over long time periods
- Characteristics of software project:
  1. Consist of a sequence of unique activities that do not repeat and are inter-related to each other
  2. Goal-oriented
  3. Time-bound
- Project Management aims to maintain an equilibrium between cost, time and scope of the project.
- Refer to section 3.3

8.1 Project Planning

- Focus on what to be implemented, how, make provisions for resources that may be required in the future.
- Project plan depends on the type of project (development, research etc.). It is an evolutionary document that changes with goals achieved, risks encountered, change in context or unforseen circumstances.
- Project Plan answers the questions:

- **Sponsor's POV**:
  - Where does the product fit in with the roadmap of our organization
  - Investment and revenue
  - Time needed, resources needed, risks involved
  - Deliverables and exit criteria
  - Who is responsible, how is progress tracked
- **Customer Perspective**:
  - Does the team understand the problem
  - Time and cost needed
  - Exit criteria
- **Executor's Perspective**:
  - Life cycle to follow
  - How to prioritize the requirements
  - Project organization (upstream/downstream relationships with other parts of the company, roles, user involvement)
  - Standards, guidelines, comm mechanisms
  - Schedule with detailed work breakdown and ownership

8.1.1 Steps in project planning

1. Understand the deliverables: customer expectations, market forces behind the project, high level decisions of make-buy-reuse (based on the feasiblity study)
2. Process plan: choose life cycle (based on goals, boundaries, time constraints, resource constraints, inter-related activities etc.). Also choose standards, guidelines to be followed (
3. Identify organizational structure in terms of people, team, responsibilities, as well as their partners (for build, install, documentation, product management, sales, support)
4. Determine deliverables (buy/build/reuse)
5. Work Breakdown Structure: Split project into level 1 sub-project, then split level 1 sub-projects into level 2 activities.
   - Aggregate tasks into phases, identify entry and exit criteria for each phase along with milestones and checkpoints
   - Identify effort and time using either an experience based model like Delphi, Modified Delphi, comparative studies etc.
   - Formal methods for estimating time and effort include the CoCoMo (Constructive Cost Model) where projects are organized into teams that are one of the following:
     - **Organic**: Small, experienced teams that understand the problem well
     - **Embedded**: Large team size, complex, need people with sufficient experience
     - **Semi-Detached**: in between the above
   - The basic CoCoMo model consists of the following

     $$E = a(KLOC)^b$$
     $$t = c(E)^d$$

     where E denotes effort in person-months, t denotes development time in months
6. Schedule and allocate resources: Schedule built by all stakeholders, resources include hardware/software/human resources, validate upstream & downstream dependencies of each phase
   - Schedlues are visualized using Gantt Charts
   - Schedule building involves risk mitigation, reducing inter-task dependencies to avoid waits, taking care of resource availability, and costing in terms of budget

7. Identify, assess and analyze, and plan for fallback for risks. Plan triggers for the fallback actions
8. Develop quality management process: Communication plan, tracking plan, QA Plan, test completion and verification criteria
9. Project plan management, release management

8.2 Monitoring execution and control

- Encompasses all the tasks and uses all the measures and metrics necessary to ensure that the project is on track e.g. within scope, on-time, within budget and proceeds with minimal risk
- Monitoring involves using Quantitative data which is continuously collected all along the project
- Control involves making decisions or adjustments in time, cost, organizational structure, scope etc. for the project under execution
- Dimensions of monitoring and control:
  - Time in terms of effort and schedule
  - Propagation and availability of information (documentation)
  - Organization and its structure, roles & responsibilities
  - Built-in quality (not add later)
  - Post-mortem for better future performance

8.3 Project Closure

- Hand over deliverables to customer & obtain project or UAT (User Acceptance Testing) sign-off from client to confirm that the team has met the project objectives & the agreed requirements
- Complete documentation, pass it to the business dept
- Release resources, inform stakeholders of the project closure
- Post Mortem - To determine the projects success and identify the lessons learned.