Week1

- Software Engineering is an art. COnsists of
  - Processes (waterfall, agile),
  - Tools (Different phases of SDLC)
  - Code organization.
- Not monolithic, but one feature at a time, incrementally.
- Componentization of a system allows to focus thoughts about each parts of the system, and allows dividiing the work for sake of planning/scheduling.
- Two components of Amazon - Inventory mgmt (customized home pages), payment gateway.
- The first step in building a software solution is gathering of requirements.
- Requirements must cater to the needs of the clients , including internal modules of the system and its end-users.
- Main objective of a requirement specification is to capture the goals of the system to be implemented.
- Design phase provides a big picture view and a structure to the software system being developed.
- Development phase implements component interfaces, without exposing the implementation details to its clients. Interfaces typically remain independent of their implementation.
- Failure to address bugs can cause severe catastrophes.
- Testing can include Unit testing, Integration testing and Acceptance testing. Acceptance testing can include alpha testing (internal) and beta testing (external users).
- Maintenance phase requires to monitor how the system is being used, and also includes regular upgrades/updates to the system.
- Exploratory approach of software development - Build & Fix.
- Software Development Lifecycle Models:
  - Waterfall - Requirements, Design, Development, Testing, Maintenance, each phase one after the other, in a sequential manner. The primary disadvantage is the turn-around time, since it's difficult to forecast the requirements/design that suits the client.
  - Prototype model. Build a working prototype before development. Get early feedback from the customers.
  - Spiral model. Combination of the above two. Here's the general approach.
    * Determine objectives, alternatives, constraints
    * Evaluate alternatives, Identify, Resolve risks
    * Develop, Verify.
    * Prepare/plan for and repeat the above steps.

        * Get feedback from the customers at the end of each iteration.
    - All of the above models offer a Plan and Document perspective. All of them suffer from similar problems - inflated budget, undue delays, or even complete abandonment.
    - Agile model. Agile Manifesto are based on 4 key principles.
        * Individuals and interactions over processes and tools.
        * Working software over comprehensive documentation.
        * Customer collaboration over contract negotiation.
        * Responding to change over following a plan.
- Agile model differs from the prototype model in that prototype created during each iterations/sprint of an agile model is not thrown away, but refined in future iterations/sprints. This is unlike prototype model, where the prototype is discarded after getting customer feedback.
- Agile processes can be thought of as being in maintenance mode across iterations.
- Agile approaches
    - Extreme programming (XP). Key features: Behaviour driven development, Pair programming, Test driven development
    - Scrum. Key features: Sprints.
    - Kanban. Key features: Work items represented in Kanban board.
- When to use Agile Vs other models.

	Question: A no answer suggests Agile; a yes suggests Plan and Document
1	Is specification required?
2	Are customers unavailable?
3	Is the system to be built large?
4	Is the system to be built complex (e.g., real time)?
5	Will it have a long product lifetime?
6	Are you using poor software tools?
7	Is the project team geographically distributed?
8	Is team part of a documentation-oriented culture?
9	Does the team have poor programming skills?
10	Is the system to be built subject to regulation?

  In short, it could depend on 3 primary factors - its customers, software characteristics and development team.
- Agile calls for collaborative cross-functional teams. Open communication, collaboration, adaptation, and trust amongst team members are at the heart of agile

Week2 - Requirements

- It's important to handle the mismatch between what the customer/end-user *requires* and what the team is developing. If not done properly, cost, time and quality will suffer.
- System users can be
  - primary: end-users of the system
  - secondary: those who are indirect users of the system, and use through an intermediary (say, managers)
  - tertiary: those who could be impacted by the system deployment, or could influence the purchase of the system. May not be users of the system.
- After gathering requirements, analysis is essential to weed out any issues with what was gathered, like ambiguous, incomplete and/or inconsistent/contradictory requirements.
- Techniques used to collect requirements
  - Questionnaires - Good for getting answers to specific questions from a large group of people. Use in conjunction with other techniques
  - Interviews - online/phone, structured/unstructured.
  - Focus groups - Gain consensus. Highlight areas of conflicts or disagreements.
  - Naturalistic observations - while spending time with stakeholders during their regular work.
  - Documentation - Refer to SOP, regulatory requirements.
- Requirements can be functional or non-functional (say, performance-related, reliability, robustness, security) in nature.
- In the Waterfall model, System Analyst typically performs requirement gathering and analysis, and consolidates the findings into a System Requirements Specification (SRS) document. SRS need not have a rigid structure.
- Advantages of SRS
  - forms an agreement between the customers and developers.
  - Reduces future reworks
  - Provides a basis for estimating costs and schedules
  - Facilitates future enhancements.
- When customers are unsure of requirements, Agile perspective helps.
- Behaviour driven design (BDD) asks questions about behaviour of the system during development, and documented in the form of user stories. It follows role-feature-benefit pattern/template.
- An example user story *View inventory* is written as
  - As an independent seller,
  - I want to view my inventory,
  - So that I can take stock of products which are low in number.

- User stories have the advantages of being light-weight, facilitate prioritization of work, and help ease communication between users and the developers.
- SMART user stories are specific, measurable, achievable, relevant, timebound.
- Agile might not work for very large, safety critical systems where documentation is critical.
- Storyboards are prototypes that are in the form of a hand-drawn comic featuring the setting, sequence and satisfaction. They emphasise on how the interface accomplishes a task, while avoiding commitment to a particular user interface.
- Wireframes are presented on multiple papers as hand-drawn UIs of different screens.
- VeriSIM is a self-learning environment designed to assist software developers in understanding and comprehending complex software designs.

Week3 - User Interface.

- Interface often makes or breaks a user experience, and is more important than a functionality from a UX point of view.
- User interfaces are designed with the help of Interaction designers.
- Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.
- effectiveness, efficiency, safety, learnability and memorability are the commonly known and documented usability goals.
- One way to quantify efficiency is to count of number of clicks to achieve a task.
- One of the ways to achieve safety is to avoid unwanted actions accidentally, or availability of a recovery mechanism.
- Learnability and memorability are important, so as to reduce the learning curve associated with the product.

- Process of interaction design.

  - Identifying needs and requirements.
  - Developing alternative design that meet the requirements.
  - Build interactive versions
  - Evaluate
- Prototyping helps choose between alternatives. It helps at
  - Test out technical feasibility of an idea
  - Clarify vague requirements
  - User testing and evaluation
- Types of prototypes, in the increasing order of fidelity, are Storyboards, Paper prototypes, Digital mockups(like Photoshop, Powerpoint), Interactive prototypes (like web app with fake data) and the Final product.
- Advantages of Storyboarding are
  - It emphasizes how interface accomplishes a task
  - Avoid commitment to a particular user interface.
  - Promotes shared understanding among stakeholders.
- Advantages of paper prototyping are
  - Easier than writing code.
  - Starts conversation about user interactions.
  - Elements can be changed immediately, based on feedback.
- User interfaces can be evaluated through feedback from the client users, and/or expert designers. Heuristics help during evaluation.
- Heuristics can be based on
  - Understanding: Consistency, Language/metaphor usage, Clean/functional design.
  - Action: Freedom(to explore, to recover), Flexibility (personalization, customization), Recognition over recall.
  - Feedback: Show status (time remaining, next steps), Prevent errors (constraints, auto-suggestions), Error recovery (tool-tips, provide alternatives), Provide help (in-context help).

Week4 - Project Management

- Role of a Project Manager as defined in a traditional model gets redefined as product owner in the Agile model. He/she serves as the interface between product development team and the customer.
- Activities of a Project Manager in the traditional model are as follows:
  - helps breaks down activities to tasks, assigns them to the team
  - help estimate the time for the tasks
  - ensures timely completion of tasks by the team.
  - helps assess and mitigate risks during the project execution
  - helps at making decisions by connecting the relevant parties
  - helps with configuration management.

- Advantages of estimation
  - Establish cost in terms of effort (person months) or money.
  - Establish schedule
  - Helps adhere to contractual obligations
- Estimates are often carried out by consulting experts (in well co-ordinated rounds), or by using heuristics. This is called delphi technique of estimation.
- COCOMO, a popular heuristics-based estimation technique, uses $code\ size$ drawn from similar projects to estimate the effort demanded by the current project. COCOMO helps compute initial estimates on the effort as $a * (code\ size)^b$, where a and b are chosen empirically, based on pre-defined project types - organic, semi-detached, embedded etc. The above formula gives the esitmate in person-months.
- The initial estimates by COCOMO are multiplied by *effort adjustment factor* to account for other factors that contribute to the final effort estimates, including
  - Expertise available in the team.
  - Technical attributes (say, reliability) of the project.
  - Tools/practices used.
- Schedule helps to monitor timely completion of task and take corrective action if it's falling behind schedule.
- Before creating schedule, the project manager must identify the tasks, break them down, represent using WBS and identify their dependencies among each other. Now, work out the estimates for the individual activities, before creating an activity network or a gantt chart for the project with all of the above information.
- Risk is an anticipated, unfavorable event or circumstance that can occur while a project is underway.
- Because software development is intangible, risks are more than that in projects in other industries. Risks can be due to insufficient expertise in technology, domain expertise, external components, business/market risks, gold plating or project risks in general (schedule, cost, staff shortage etc).
- Here is a list of various risks in a software project
  - Business Risks: Risks stemming from unmet client expectations, market demands, or financial implications.
  - Technical Risks: Challenges related to technology, such as compatibility issues, new tech adoption, insufficient knowledge of the product or system failures
  - Project Risks: Risks involving schedule delays, resource allocation, staff turnover, or lack of domain knowledge.
  - Process Risks: Issues arising from inefficiencies or gaps in development processes and workflows.
  - Operational Risks: Problems with deployment, maintenance, or day-to-day operations of the software.
  - Security Risks: Threats like cyberattacks, data breaches, or weak security

practices impacting system integrity.

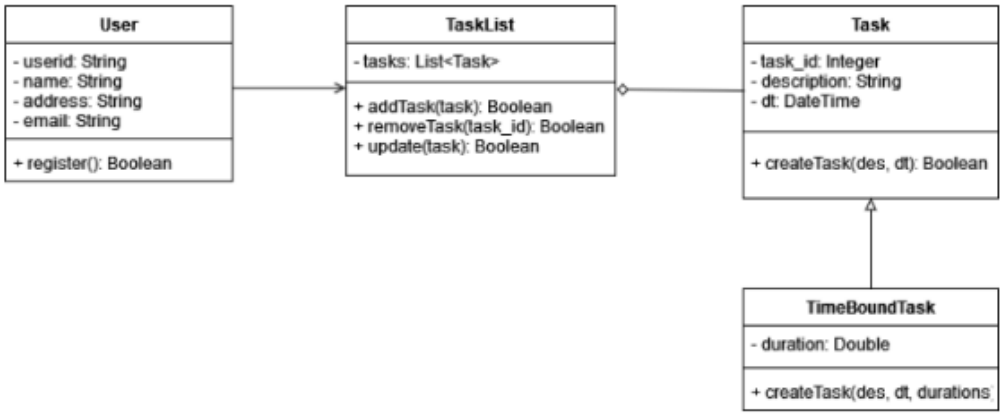
- A *risk table* lists all possible risk items with a *probability and impact (severity)* attached to each. *Risk factor* is calculated as a product of the *probability* and *impact.* *Impact* can be one of negligible, marginal, crifical and catastophic (1 - 4).

$$R = P * I.$$

  Rework on items with the highest risk factor.
- Under Agile process, cost and schedule is NOT predicted at the start of the project, since user stories could be reprioritized from one iteration/sprint to the next.
- Activities in scrum are
  - Sprint
  - Sprint planning
  - Daily scrum meeting
  - Sprint review
  - Sprint retrospective
- A s*print* is defined as a short, time-boxed period (say, 2 weeks) when a scrum team works to complete a set amount of work.
- Scrum team consists of 3 roles.
  - Development team - including developers, designers, testers.
  - Product owner - interfaces between the client and development team.
  - Scrum master - ensure all activities are being done well.
- During the estimation phase, each user story is rated using a scale into one of straight-forward(1 point), medium(2 points) or very complex(3 points). Now, these stories are added into the backlog - a prioritized list of user stories and requirements.
- *Planning* is separately done for each sprint. During sprint planning, user stories are pulled into the sprint from the product backlog. *Sprint planning* is conducted with the entire scrum team in attendance, where the team seeks answers to two basic questions
  - What work can get done in this sprint?
  - How will the chosen work get done?
- Points associated with each story helps in assessing the velocity of the team (number of points per sprint). This would help during the planning for the next sprint.
- During the daily scrum meeting, each members answers 3 questions.
  - What did I work on yesterday?
  - What am I working on today?
  - What issues are blocking me?
- Activities of a project manager in the Agile model are as follows:
  - defines/owns the user stories
  - helps assign points to each user story.
- Number of points per iteration/sprint is termed velocity.

Week5 - Design

- Outcome of design
  - Components/modules that accomplish well-defined tasks
  - Interfaces that define how component communicate with each other
  - Data structures for individual components
  - Algorithms to implement individual components
- Characteristics of a good software design - correctness, efficiency, maintainability, understandability
- Coupling is a measure of the degree of interaction between the modules, and when the coupling is low, the design is said to be modular. Types of coupling are
  - Data coupling - datatype based
  - Control coupling - flag/switch based
  - Common coupling - shares global data
  - Content coupling - references across internals of the modules
- Cohesion is a measure of how functions in a module cooperate together to perform a single objective. Types of cohesion are
  - Functional cohesion - all functions used by a module are present in the same module.
  - Sequential cohesion - output from one function is input to next
  - Communicational cohesion - updates shared data between components.
  - Procedural cohension - functions executed in a sequence, but unrelated by functionality.
  - Coincidental cohesion - functions with meaningless relations.
- Good design will have high cohension, low coupling.
- Class relationships
  - Association - A delegates to B
  - Composition/Aggregation - A is a collection of B's, B is part of A
  - Inheritance - A is a specialization of B.
  - Dependency - B depends on A (eg. A is an interface, and B implements A)
- UML diagrams is a classical vehicle used in communication and idea generation in a team environment. It helps guide software development.
- Structural view of UML describes the
  - Structure/components of the software system and their relationships
  - logical parts of the system - classes, data and functions
- Dynamic behavioural views of UML
  - State Machine view - models different states of a class object
  - Activity view - models flow of control among various activites
  - Interaction view - sequence of message exchanges, like with sequence diagrams.
- Consider the below class diagram



As per the above class diagram,

- There is an association relation between class User and TaskList
- There is a composition relation between class Task and TaskList
- There is an inheritance relation between class Task and TimeBoundedTask

Week7 - Development

- Writing code - IDEs, Frameworks
- Sharing and maintaining code versions - Version control systems (like Git)
- Version control helps in tracking and managing changes to source code or other documents and maintaining multiple versions of the code.
  - Client-server based centralized VCS. Eg. Perforce, SVN.
  - Distributed VCS. Eg. Git keeps a local copy also.
- 3 stages for project files - Modified, Staged, Committed.
- After branching off to *temp* from *main* branch, if you want to incorporate the latest changes from *main* branch into *temp* branch without merging them, you must use *git rebase*
- Debugging terminology
  - Error - discrepancy between actual and intended behaviour.
  - Failure - observable error - incorrect output value, exception etc.
  - Fault - where the failure has occured (lines in code)
- Debugging techniques
  - Insert print statements.
  - Use logging data, compare between executions.
  - Use debugger tool.
  - Use Profiling tool to find how often and how long are various parts of a program are run.
- Debugging strategies
  - Input manipulation: Edit inputs and observe differences in output.

  - Backwards: Find statement that generated incorrect output, follow data and control dependencies backwards to find incorrect line of code.
  - Forwards: Find event that triggered incorrect behaviour, follow control flow forward until incorrect state reached.
  - Blackbox debugging: Find documentation, code examles to understand correct use of API
- Software Metrics
  - McCabe complexity = Number of decisions in the code + 1. Constructs like if, $elif$, $while$ and for each increase the number of decisions by 1. Typicaly, when the complexity $\geqslant 41$, the code is considered unstable.
  - Raw metrics: Source Lines of code, Logical lines of code, Comments
  - Halstead metrics (#operators, #operands etc)
- Code smells are problematic characteristics in code

Week8 - Software Architecture

- Efficiency of space, code.
- Types/styles of architectures:
  - Client-server systems - data is transacted in response to requests.
  - Pipe and filter systems - data passed from one component to another, while transforming and filtering it along the way.
  - MVC: Views are separated from manipulations of data.
  - Peer to Peer (P2P) = diffferent systems form nodes and share resources between them.
- *Components* communicate with each other through *Connectors* (code), based on specific *Protocols*. Examples of *connectors* include REST API, or functions in general.
- Proposed by Rober Martin, SOLID principles are Object-oriented principles for software design
  - Single Responsibility: Every class should've a single responsibility/purpose.
  - Open-Closed Principle: Software entities should be open for extension (inheritance), but closed for modification.
  - Liskov Substitution Principle: Subclass should implement the whole of parent class (and not override or weaken its methods), and hence subsitutable for parent class.
  - Interface Segregation Principle: Do not force any client to implement an interface which is irrelevant to them. So, keep the interfaces devoid of such irrelevant methods. The best solution is to create additional interfaces.
  - Dependency Inversion Principle: Prefer abstraction/interfaces over implementations.
- Design patterns are descriptions of communicating objects and classes that are

customized to solve a general design problem in a particular context.

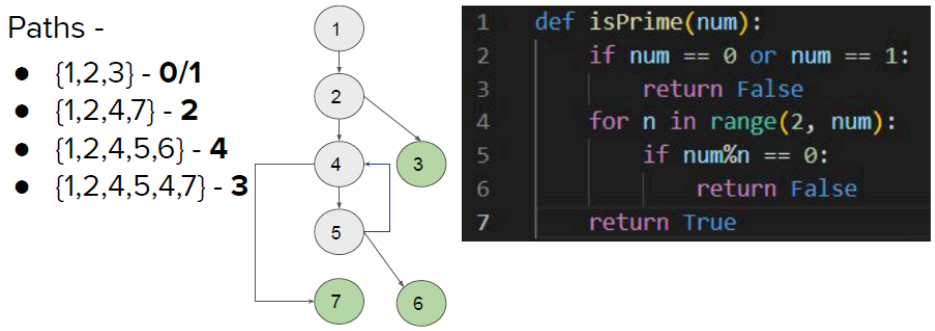
- Types of design patterns:
  - Creational: Used during process of object creation
  - Structural: Composition of classes or objects
  - Behavioral - Characterize the ways in which classes or objects interact.
- Creational patterns:
  - Factory: a parent abstract class with a method that constructs an object using new(). Adheres to single-responsibility and open-closed principles, but could make the code more complicated with a lot of new sub-classes.
  - Builder: a builder class with the same set of attribtues as the original class, and separate methods for each attribute. Allows construction of objects step by step, and adheres to single-responsibility principle, but could make the code more complicated. Allows the use of *telescopic* constructors.
- Structural patterns:
  - Facade: Simple interface to a library, or a complex set of classes.
  - Adapter: Intermediary (adapter class) between client and the adaptee class. Adheres to single responsibility and open-closed principle, but could make the code more complicated.
- Behavioural patterns:
  - Iterator: Iterating through collections (list, queues, stacks, trees); must be independent of the storage mechanism used. Iterator defines the hasNext() method, and hence the client doesn't need to know its implementation. Adheres Single responsibility and Open/Closed principles, but could be an overkill for simple collections.
  - Observer: Notifies multiple observers via separate observer interface implementations.
  - Strategy: Context(original class) delegates to Strategy inteface. Helps at isolating implementation details of the algorithm. Adheres to open/closed principle. May be an overkill if there are limited number of strategies.

Week9 - Software Testing

- Test case is a triplet of data input(I), program state (S), expected result (R)
- Test suite is a set of test cases.
- Designing a minmal test suite is necessary, since domain of inputs is large.
- Black-box tests cover all possible outcomes, whereas white-box tests cover all possible code paths.
- Unit testing - Integration testing - System testing (Alpha/beta) - Acceptance testing
- Unit testing is done while coding the module/unit. Need drivers/stubs.
- A driver module contains the non-local data structures accessed by the module under

test.

- A stub provides dummy procedures that have the same input and output parameters as the function called by the unit under test, but it has a highly simplified behaviour.
- In black box test design, identify equivalence classes, every input data of which behave similarly. Identify boundaries.
- White-box testing uses heuristics - branch coverage, multiple condition coverage, path coverage.
- To list tests based on path coverage, draw the control flow graph for the program, where the nodes represent line numbers and edges represent program flow. Following picture shows the code for *is_prime* on the right, its control flow graph in the middle, and possible paths (including the test inputs) on the left.



- Number of tests representing path coverage is given by cyclomatic complexity:

$$Number\ of\ decision\ and\ loop\ statements\ +\ 1$$

NOTE: $if\ True$ is not considered as a decision point, so it doesn't add to the complexity. Further, *else* clause in an *if* condition doesn't add to complexity.

- Integration testing could detect errors at the module interfaces.
- Integration testing can be done using
  - Big bang approach - difficult to pinpoint issue sources.
  - Bottom-up approach - modules of each subsystem are integrated. Don't need stubs.
  - Top down approach - starting with the root, integrate 1-2 sub-modules at a time. Don't need drivers.
  - Mixed approach - integrate as and when modules are available.
- System testing can be
  - Alpha testing - within the organization.
  - Beta testing - by select customers.
  - Acceptance testing - by the customers at the last leg of software delivery.
- Smoke testing tests for basic functionalities and conducted before system testing
- Load testing checks if the system can support normal load and stress testing checks if the system can take peak loads, while measuring input data volume, rate, processing time, utilization of memory etc.

- In TDD, write test first, make it fail(Red), then write minimum code to make it pass (Green), and then refactor.
- Here are some terminologies related to software testing.
  - Error: discrepancy between actual behaviour and intended behaviour
  - Failure: observable error like incorrect output value, exception etc.
  - Fault: where the failure has occured.
  - Debugging: determining the cause of failure.

Week10 - Software Deployment

- Deployment - Make software available to its users
- Monitoring - Capture and analyse various metrics including performance issues
- Types of deployment environment
  - Development: Local to developer (working copy) + shared code base.
  - Testing: - rolled back to previous version, in case of issues.
  - Staging - pre-production, and replicates the production.
  - Production
- Why staging?
  - To avoid system crashes due to production issues.
  - Preview new features
  - Performance testing should be done in an environment as close to production as possible.
  - Live updates/upgrades.
- To help with the transition from staging to production, following strategies may be used.
  - Blue/Green: Two different environments (blue and green) are used. Users are routed/re-routed to one of the environments as required. Obvious advantage is that the rollback is easy, since it's just a matter of re-routing the users to another environment. The environments will be cycling among each other.
  - Canary deployment. Uses phased/Incremental rollout. Roll-out to various subsets of users, decided by their profiles, demographics or region. Disadvantage is that multiple versions need to be managed at anytime.
  - Versioned deployment. Allow users to choose if they want to update their versions. Disadvantage is that multiple versions need to be managed at anytime.
- The hosting environment must be up and running at all times, and user data must be securely stored. Options are
  - Bare metal servers: Setup from ground up, with racks and blades. Gives the highest performance. Very expensive and time/effort intensive. Maintenance is expensive.
  - Infrastructure as a service (IaaS). Uses a virtual system, hosted by the IaaS provider. Advantage is reduced cost, time and effort. Disadvantages include

compromises in terms of system configuration, reduced performance, Example: Digital Ocean.
  - Platform as a service (PaaS). In addition to the infrastructure, the provider also deploys our choices of software (OS, database etc). Easiest of all, but lacks control. Example: Heroku, Google App Engine.
- Continuous integration helps automate integration and deployment of software
- Continuous Integration (CI) server pulls code from the VCS, runs tests on it, then signals the staging server to take over. Next, the staging server pulls code from the VCS, runs tests on it and then deploys it. Build tools helps in automating these steps.
- Frequent commits are recommended, so that we can conflicts can be reduced during merging.
- Benefits of Continous Integration
  - Reduces deployment time
  - Avoids last minute rush near release dates.
  - Beneficial to the end-users because of the availability of the staging server for quick demos
  - It pushes the developers to create modular code
- Techniques to improve performance
  - Caching - Fetch from memory, instead of database. Can occur at the level of web browser, web server or database.
  - Task queues for resource and time-consuming tasks. Async execution. Having a pool of workers who could divide among themselves the the queued up tasks, could further improve the performance.
  - Minify code, so loading and rendering pages can be done quickly.
-