1 Thinking of Software in Terms of Components

1.1 Case Study: Amazon System

- Components are a way of breaking the complexity of a task into manageable parts, so that different teams can work on different components of the system and put everything in a timely manner.
- Everyone need not the working of a component, just need to know the input and output.
- Amazon components: Inventory Management, Payment Gateway, Order Management, Shipping System.
- Inventory Management is the act of measuring the amount, location, pricing, mix of products available on Amazon.
- Inventory gets updated based on current purchasing and seasonal trends.
- Payment Gateway is a service that authorizes electronic payments.
- **Key Takeaway**: Software can be divided into separately addressable components called **modules** that are integrated to satisfy requirements.
- Amazon Pay: A mobile wallet, can link credit or debit card information, can link bank accounts, or you can transfer money online to the mobile wallet. Instead of using your debit or credit card to make purchases you can pay with your smartphone that has this mobile wallet. There are many categories like Recharges, Bill payments, Travel and Insurance, etc.
- First Step in create a new software component could be **study existing components of the system**, **learn a programming language**, **look at similar systems to understand features**.
- **Study Existing components of the system**: To understand how the new component will interact with existing components.
- We need to first understand what is the problem we want to solve?, and based on an analysis of existing or similar systems we need to come up with an explicit set of goals for our own system or for what our implementation should provide.
- **Requirements**: Goals the implemented system should have, and they should cater to the need of clients.
- Client can be an external user or internal users. Could be for employees or customers. Client can also be another software as well.
- Think about **who** is going to use your software, for **what purpose**, and in **what way**.
- **Key Takeaway**: Requirement specification is the first step in the software development process, through this we need to ensure that the requirements capture clients needs.

1.2 Different phases of Development

- If we went directly to coding after specifying requirements, then we can face issues during integration where different developers may have different ideas about how the functionality should be implemented, difficulties while adding new features that is if I want to add a new feature then it would help to have a big picture view of the system.
- **Software Design**: Big picture view of the software system, provides a structure to the software system.
- When a feature is being implemented, multiple developers work together and write code for the feature, they use tools like GitHub to collaborate and write code and very often coding is done in a distributed manner with developers working in different location and even in different time zones. Hence, it is very important that everyone working on the codebase has a consistent understanding of what the code does. For this reason, all developers write **documentation** for their code and write precise interface definitions.
- **Interface**: An interface is the description of the actions your functions can do without describing the implementation in detail. The interface shows what requests are accepted and in what format is the corresponding response given.

- **Software Development**: Write code based on the requirements and the design. Usually distributed, and the developers write documentation and precise interface definitions.
- **Testing** is done to ensure that the software behaves according to the requirements, many bugs might still exist in the system. A failure to address bugs can even cause severe catastrophes.
- Testing is done at different granularities, examples include unit testing, integration testing, acceptance testing.
- Alpha testing done by internal employees and Beta testing done by actual users.
- **Maintenance**: After the feature is rolled out, monitor how users are using the feature. Purpose of doing this is to monitor what users are doing, and how they are using the software, change the code for upgrades/updates, or add features.
- **Overall Process**: Requirements → Design → Development → Testing → Maintenance

1.3 Software Development Life Cycle

- **Software Lifecycle**: Different stages over which a software evolves from the initial customer request to a fully developed software.
- **Waterfall model**: Plan and document perspective. Each phase occurs one after the other.
- If we follow all phases sequentially, then the time taken could be very long, and if the client doesn't like it or has some changes we will have to start the process all over again.
- **Drawbacks of Waterfall**: Increase in cost, time if changes are required later on. Clients may not know what they need. Designers may not know which design might be the most feasible/usable by the clients. Can take quite long.
- **Prototype Model**: Build a working prototype before the development of the actual software. Prototype usually not used later.
- **Advantages of prototype**: Exact form of solution and technical issues are unclear, and it is useful to get feedback from customers.
- **Disadvantages of prototype**: Increased development costs, bugs can appear later in the development cycle.
- **Spiral Model**: Incrementally build the software and get feedback, refine. Combines advantages of Waterfall and Prototype model. Each iteration can still take a long time.

1.4 Agile Perspective

- **Agile Manifesto**: Emphasizes individuals and interactions over process and tools, emphasizes over delivering working software rather than comprehensive documentation, emphasizes on customer collaboration over contract negotiation, emphasizes on responding to change over following a plan.
- **Incremental Development**: Teams work together to deliver the product in small increments.
- **Agile Approaches**: Extreme Programming(XP), Scrum(Product is built in a series of iterations known as sprints which are roughly 1–2 weeks long, this helps break done a project into several small byte sized pieces), Kanban(Software to be built is divided into small work items and these are represented on a kanban board allowing team members to see the state of any piece at any given time).
- **When to use Agile/Plan and Document?**: If the answer is no then Agile else Plan and Document
  - Is specification required?
  - Are customers unavailable?
  - Is the system to be built large?
  - Is the system to be built complex?
  - Will it have a long product lifetime?
  - Are you using poor software tools?
  - Is the project team geographically distributed?
  - Is team part of a documentation-oriented culture?
  - Does the team have poor programming skills?
  - Is the system to be built subject to regulation?
- What is Agile?

2 Requirements Gathering and Analysis

2.1 Case Study: Amazon Seller Portal

- Our vision of what the software should look like and behave is quite different from what the user has in mind.
- We want to make sure that developers understand what customers want, customers come to an agreement about their requirements. If this does not happen we could end up with increased cost and iterations.
- Amazon wants to develop a portal for sellers. Products which sellers list on the portal will be available for people to buy on the Seller portal.
- **Primary Users**: Frequent users of the system. Example include Independent sellers, Sales team of consumer companies, Independent authors and publishers.
- **Secondary Users**: Do not directly use the system, use the system through an intermediary. Examples include Sales team managers.
- **Tertiary Users**: Do not use the software at all, affected by the introduction of the software, and Influence the purchase of the software. Examples include logistics, shipping companies, banks, people buying on Amazon.
- Requirements can be vague or unclear. Requirements can be inconsistent or contradicting. Requirements can be incomplete.

2.2 Identifying Users and Requirements

- **Questionnaires**: Series of questions designed to elicit specific information from users. Good for getting answers to specific questions from a large group of people. This should be used in conjunction with other techniques.
- **Interviews**: Asking a set of questions, can be face-to-face, telephonic/online interviews. Can be structured, unstructured, or semi-structured. This helps to get people to explore issues, used early to elicit scenarios.
- **Focus Groups**: Get a group of stakeholders to discuss issues and requirements. Advantages include gaining consensus, highlighting areas of conflict, disagreement.
- **Naturalistic Observations**: Spending time with stakeholders as they go about their day-to-day tasks, observing their work in their natural setting. Shadowing a stakeholder, make notes, ask questions, observe.
- **Documentation**: Procedures and rules for a task, steps involved in an activity, regulations governing a task.

Technique	Good for
Questionnaires	Answering specific questions
Interviews	Exploring issues
Focus Groups	Collecting multiple viewpoints
Naturalistic Observations	Understanding context
Documentation	Procedures, regulations, standards

- **Basic Guidelines**: Focus on identifying stakeholders needs, involve all stakeholder groups, use combination of data gathering techniques. Run a pilot session if possible to ensure your data-gathering session is likely to go as planned.
- Data gathering is expensive, time-consuming - have to be pragmatic, make compromises.
- **Functional Requirements**: Captures a functionality required by the users from the system.
- **Non-Functional Requirements**: Essentially specifies how the system should behave. Examples include Reliability, Robustness, Performance, Portability, Security, etc.
- **Reliability**: Te extent to which a program behaves the same way over time in the same operating environment.
- **Robustness**: The extent to which a program can recover from errors or unexpected input.

2.3 Software Requirement Specification

- **Requirement gathering and analysis**: Done by system analyst, along with other members of the software team. Organize these requirements in **Software Requirements Specification**(SRS) document.

1. Introduction 1.1 Purpose 1.2 Scope 1.3 Definitions, acronyms, and abbreviations 1.4 References 1.5 Overview 2. Overall Description 2.1 Product Perspective 2.2 Product Functions 2.3 User Characteristics 2.4 Constrains 2.5 Assumptions and Dependencies	Broad outline and description of the software system
3. Specific Requirements 3.1 External Interface Requirements 3.1.1 User Interfaces 3.1.2 Hardware Interfaces 3.1.3 Software Interface 3.1.4 Communication Interfaces 3.2 System Features 3.2.1 System Feature I 3.2.1.1 Introduction/Purpose of Feature 3.2.1.2 Stimulus/Response Sequence 3.2.1.3 Associated Function Requirements 3.2.1.3.1 Functional Requirement I ... 3.2.1.3.n Functional Requirement n 3.2.2 System Feature 2 ... 3.2.m System Feature 2 3.3 Performance Requirements 3.4 Design Constraints 3.5 Software System Attributes 3.6 Other Requirements	Functional and Non-Functional Requirements

Table 1: Standard Structure of SRS document

- table ① is a guideline of how an SRS document should look like and is not very rigid.
- SRS helps form an agreement between customers and developers. It helps to reduce future reworks. Provides a basis for estimating costs and schedules.

2.4 Behavior Driven Design - User Stories

- Plan and Document perspective requires customers to be clear about their requirements before building the software, but if they are unsure of the requirements then we can follow agile perspective.
- **Behaviour Drive Design**: Asks questions about the behaviour of an application before and during development. Requirements are continuously refined to meet user expectations.
- **User Stories**: Short, Informal, plain language description of what a user wants to do within a software product which is of value of them. Smallest unit of work which can be done in 1 sprint, which is about 1–2 weeks.
- Role-feature-benefit pattern/template: As a [type of user], I want [an action], So that [a benefit/value].

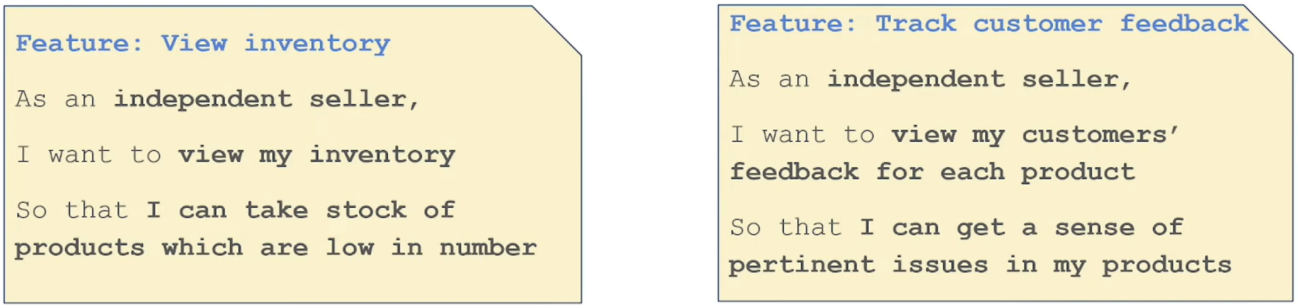


Figure 1: User Story Examples

- User Stories are lightweight and help plan and prioritize development. Concentrate on behaviour rather than implementation of the application. Conversation between users and the development team.
- **SMART**: Specific(know exactly what to implement), Measurable(known expected results for some inputs), Achievable(Implement the user story in 1-2 weeks), Relevant(Business value to one or more stakeholders), Timeboxed(Stop implementing a feature once time budget expected).
- May be difficult to have continuous contact with users. Not able to scale to very large projects, safety critical applications.
- Brief overview of the difference between requirements and user stories.

3 Software User Interfaces

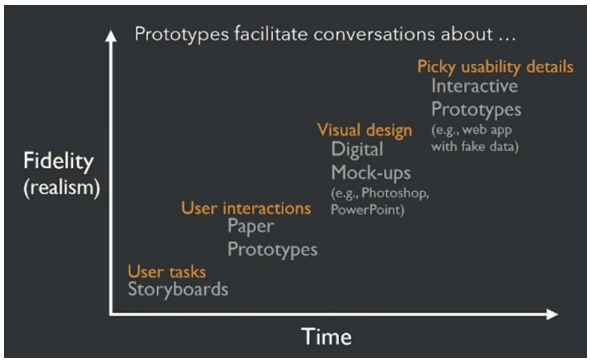
3.1 Introduction to Interfaces

- Most user stories, require us to create a user interface or a UI that acts as an interaction point between the user and the software.
- Activities involved in Interaction Design include identifying needs and requirements, developing alternative design that meet those requirements, Build interactive versions, and Evaluate each of these designs are useful for the user.
- **Usability**: The extent to which a product can be used by specified end users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
    1. **Effectiveness**: How good a system is at doing what it is supposed to do.Is the system capable of allowing people to learn well, carry out their work efficiently, access the information they need, buy the goods they want etc.
    2. **Efficiency**: How does a system support users in carrying out their tasks. Common tasks through minimal number of steps.
    3. **Safe to use**: Protecting the user from dangerous conditions and undesirable situations. Helping users in any situation to avoid carrying out unwanted actions accidentally.
    4. **Learnability**: How easy a system is to learn to use. Want to get started right away and carry out tasks without much effort.
    5. **Memorability**: How easy a system is to remember how to use, once learned.
- **User Experience Goals**: Want users to experience positive emotions while using the software, More subjective, How users experience a product from their perspective.
- **Prototypes** allow you to quickly test on users, get feedback, iterate, and pivot.
- Prototypes answer questions and support designers in choosing between alternatives.
- **Prototyping**: Test out technical feasibility of an idea, clarify some vague requirements, and User testing and evaluation.
- **Storyboard**: A hand drawn comic that features Setting, Sequence, and Satisfaction
    1. **Setting**: People involved, Environment, Task being accomplished
    2. **Sequence**: What steps are involved? What leads someone to use the app? What task is being illustrated?

3. **Satisfaction**: What motivates people to use the system? What does it enable people to accomplish? What need does the system fill?

- **Benefits of Storyboard**: Emphasizes how interface accomplishes a task. Avoids commitment to a particular user interface. Shared understanding among stakeholders. Some resources video-1 and video-2.
- **Paper Prototypes**: Hand-drawn interface on multiple pieces of paper.
- **Benefits of Paper Prototypes**: Easier than writing code for user interface. Starts conversation about user interactions. Elements can be changed immediately based on given feedback.
- **Digital Mock-ups**: Using stuff like photoshop, PowerPoint, transform a paper prototype into a digital mockup.

(a) User Goals Summary



(b) Prototypes Summary

Figure 2: Summary of Interfaces

3.2 Evaluation using Design Heuristics

- **Heuristic Evaluation**: Heuristics are the strategies derived from the previous experiences with similar problems. Rule of thumb/guidelines.
- **Heuristics for Understanding**
  1. **Consistency**: Consistent Layout. Consistent Name.
  2. **Use Familiar Languages and Metaphors**
  3. **Clean and Functional Design**
- **Heuristics for Action**
  1. **Freedom**: Freedom to Undo. Freedom to explore.
  2. **Flexibility**: Experts as well as new users should be able to carry out tasks efficiently.
     Personalization: Tailoring content/functionality for individual users.
     Customization: Allow users to make selections about how they want the product to work.
  3. **Recognition over Recall**: Users find it easier to recognize something they have seen earlier. Interface - buttons, navigation etc. should help the user reach his goal.
- **Heuristics for Feedback**
  1. **Show Status**: Keep users informed about what is happening, through appropriate feedback within a reasonable amount of time. Provide next steps. Provide warnings in advance.
  2. **Prevent Errors**: Include helpful constraints. Offer Suggestions.
  3. **Support Error Recovery**: Make the problem clear. Provide a solution. Provide an alternate.
  4. **Provide Help**: Ensure help is easy to search. Provide help in context

- Experts evaluate the prototype, i.e., do multiple passes and provide a list of issues that violate design heuristics.

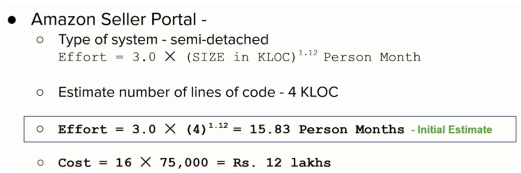
Figure 3: Design Heuristics Summary

- Importance of typography.

4 Project Estimation Techniques

4.1 Introduction

- **Importance of Estimation**: Establishing cost and Schedule. From a Client's perspective cost and schedule must be provided to them.
- **Key Estimation Parameters**: Size/Lines of code(KLOC, number of 1000 lines of code), Effort(How many people are required in the team, Person-month, effort an individual can typically put in a month).
- **Empirical Estimation**: Ask people who have completed similar projects.
- **Expert Judgment**: They can make an educated guess, but can encounter human errors, individual bias, optimistic estimates, overlook some factors, lack of adequate knowledge. To some extent, this can be averted by having a group of experts.
- **Delphi Technique**: Coordinator provides multiple Experts the SRS document and a form for recording cost estimates. Experts submit their estimates to the coordinator. Then the coordinator prepares a summary and distributes to all experts. Now, experts look at this summary and re-estimate the cost. This process can be iterated over several rounds.
- **Heuristic Technique**: Modelled using suitable mathematical expressions.
- **COCOMO Estimation Model**: Constructive Cost Estimation Model, $Effort = a \times SIZE^b$. a and b depend on the type of project.
- **Types of Projects**: Organic(Well understood application program, and team size is small and experiences), Semi-detached(Mix of experienced and inexperienced people), Embedded(Strongly couple with hardware).
- For Organic projects $a = 2.4$ and $b = 1.05$, for semi-detached project $a = 3.0$ and $b = 1.12$, and for embedded projects $a = 3.6$ and $b = 1.20$.
- **Effort Estimation Parameters**: People working on the project, Technical attributes of the project, Tools and practices used by the team.
- In COCOMO model, after getting initial estimates we add **cost driver attributes**. This would include Reliability, Database sizes, etc.



(a) COCOMO model

Cost Drivers	Rating				
	Very Low	Low	Nominal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

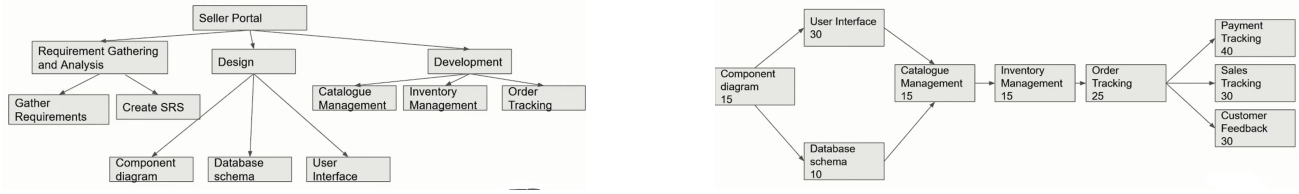
- Initial estimate = 15.83 PM
- Effort adjustment factor = 1.40 x 1.08 x 1.13 x 0.95 = 1.62
- Final estimate = 15.83 x 1.62 = 25.65 PM

(b) Cost Drivers

Figure 4: Amazon Seller Portal Example

4.2 Project Scheduling

- Helps monitor timely completion of a task, and take corrective action if it falls behind.
- The Schedule can be built in steps as follows
  1. Identify all major activities,
  2. Break down each activity into tasks,
  3. Determine the dependency among different tasks,
  4. Estimations for time durations required to complete the tasks,
  5. Represent this information in a chart/graph/network,
  6. Determine task starting and end dates from the representation,
  7. Determine the critical path(a chain of tasks that determine the duration of the project),
  8. Allocate resource to the tasks.
- Breakdown of activities can be done using **Work Breakdown Structure(WBS)**, it will create a tree like structure as follows
  1. **Root**: Project name
  2. Each node is broken down into smaller activities, **children**
  3. Each leaf represents a task which can be allocated to a developer and scheduled.
  4. **Task**: Each task should take roughly two weeks to develop.
- Once we have the breakdown, now we create the activity network.
- **Activity Network**: Different activities making up a project, estimated durations, interdependencies. Leaf nodes of the WBS become the nodes of the activity network.



(a) Work Breakdown Structure

(b) Activity Network

Figure 5: Seller Portal Example

- **Gantt Chart**: Another way to represent Activity network, it is a type of chart.

4.3 Risk Management

- A risk is an anticipated unfavorable event or circumstance that can occur while a project is underway. This can be due to intangible nature of software, conflicts in a team.
- **Technical Risks**: Due to development team's insufficient knowledge about the product.
- Developing the wrong functions and user interfaces, can be mitigated by communicating with clients and build prototypes.
- Shortcomings in external components, can be mitigated by benchmarking and regular inspections.
- **Project Risks**: Project risks occur due to problems in budget, schedule, personnel, resources, and customer related problems.
- Most common type is schedule slippage, the project falls behind schedule, can be mitigated by creating detailed milestones, constant iterations, communicate frequently with clients.
- Insufficient domain knowledge/technical knowledge, can be mitigated by hiring developers with relevant experience, or we could outsource to third party vendors.
- Personnel shortfalls, can be mitigated by cross-training, train multiple people with skills required to work on the project.

- **Business Risks**: Risks which can harm the business aspects of the software product.
- Product is no longer competitive in the market, can be mitigated by exploring the market for similar products.
- Gold plating, developing unnecessary features, can be mitigated by communicating with clients and do cost-benefit analysis.
- **Risk Assessment**: Project manager asks everyone in the team for worst case scenario and the PM then creates a "risk table".
- Then a probability(P) is assigned to each risk, each risk is also assigned Impact(I), which can negligible, marginal, critical, catastrophic(1-4).
- The risk is then calculated as $Risk = P \times I$, which then is sorted in descending order and decide which risk need to mitigated first.

4.4 Project Management in Agile

- Does not predict cost and schedule at the start of the project.
- **Team formation**: Usually team size is 4–9 people, and we organize development using **Scrum**. At the heart of scrum are sprints.
- **Sprint**: short, time-boxed period when a scrum team works to complete a set amount of work.
- **Development Team**: Whoever is required to complete work in that given sprint.
- **Product Owner**: Interfaces between the client and the development team.
- **Scrum Master**: Ensures all activities are being done well.
- **Sprint Planning**: This is a collaborative event between product owner, scrum master, and the development team. We ask two basic questions, What work can get done in this sprint? and How will the chosen work get done?. This meeting is roughly 2 hours per week of its iteration.
- **Product Backlog**: Prioritized list of work for the development team that is derived from user stories and requirements. Prioritizing will be done in sprint planning meetings.
- **Standup/Daily Scrum Meeting**: Daily meeting which involves everybody. Each member answers three questions, What did I work on yesterday?, What am I working on today?, and What issues are blocking me?
- **Sprint Review**: After the sprint the team demonstrates what they have completed. Move things from To-Do, In Progress to Done.
- **Sprint Retrospective**: Evaluate the last sprint, Discuss user stories/tasks that went well/didn't go well, and finally create and implement a plan.
- **Project Scheduling**: Key indicator of progress is how many user stories are implemented. Project estimation can be simply counting the number of user stories completed per iteration/sprint.
- Not all user stories require the same effort, and hence this can lead to mis-prediction. This is mitigated by assigning points to user stories, and calculate **velocity** which is the number of points per iteration/sprint.
- One good software to do this is **Pivotal Tracker**, Link for which can be found here.

5 Software Design

5.1 Outcomes of the Design Process

- We will design an high level view of software architecture
  1. **Components**: Collection of functions and data, should accomplish some well-defined tasks.
  2. **Interfaces**: How components communicate with each other.
  3. **Data Structures**: Suitable data structures for storing and managing data.
  4. **Algorithms** required to implement individual components.
- Characterizing a Good Software Design

1. **Correctness**: Correctly implement all the functionalities of the system.
2. **Efficiency**: Ensure that resources, time, space, cost, are managed well.
3. **Maintainability**: Easy to Change
4. **Understandable** by everyone in the development team.

5.2 Design Modularity

- When all functions in a module perform a single objective, the module is said to have good cohesion.
- **Coupling**: Measure of the degree of interaction between two modules.
- **Cohesion**: Measure of how functions in a module cooperate together to perform a single objective.
- A good design will have high cohesion and low coupling.
- **Modular**: Problem has been decomposed into a set of modules that have only limited interactions with each other.
- **High cohesion**: Functions of the module cooperate with each other for performing a single objective.
- Two modules are **data coupled** if their communication is using a primitive data type.
- **Control Coupling**: Data is passed that influence the internal logic of a module.
- **Common Coupling**: If two modules share global data items.
- **Content Coupling**: If one module refers to the internals of the other module.
- **Functional Cohesion**: Different functions of the module cooperate to complete a single task.
- **Sequential Cohesion**: Different functions of the module execute in a sequence. Output from one function is input to next in the sequence.
- **Communicational Cohesion**: If all functions of the module refer to or update the same data structure.
- **Procedural Cohesion**: Activities in the module are related by sequence. Set of functions in the module are expected one after the other, work towards entirely different purposes.
- **Coincidental Cohesion**: Module has functions with meaningless relationships with one another.

5.3 Object-Oriented Design

- **Object** are key building blocks. Working of a software in terms of interacting objects. Usually represent a tangible real-world entity.
- An object contains data, methods, and they follow encapsulation. This ensures data hiding/abstraction.
- **Class** are template for object creation. All objects possessing similar attributes and methods constitute a class.
- **Association**: Take each other's help to perform some functions.
- **Composition**: Represents whole/part relationships.
- **Inheritance**: Extend features of an existing class.
- **Dependency**: If class B depends on class A, if any changes are made to class A, changes have to be made to class B as well.

5.4 Unified Modelling Language Diagrams

- **Modelling**: Creating an external, explicit representation of the system to be built.
- **UML** help represent the software design via multiple views and greater level of detail.
- **Structural View**: Structure/components of the software system, and relationships. It describes logical parts of the system, i.e., classes, data, and functions.
- **Class Diagram**: Describes the structure of the system. Describes the system's classes, attributes, operations, relationships among objects.

- **Dynamic Behavioral View**: Describes behavior of the system over time. Further classified into
  1. **State Machine View**: Models different states of an object of a class.
  2. **Activity View**: Models flow of control among computational activities.
  3. **Interaction View**: Sequence of message exchanges among parts of a system.

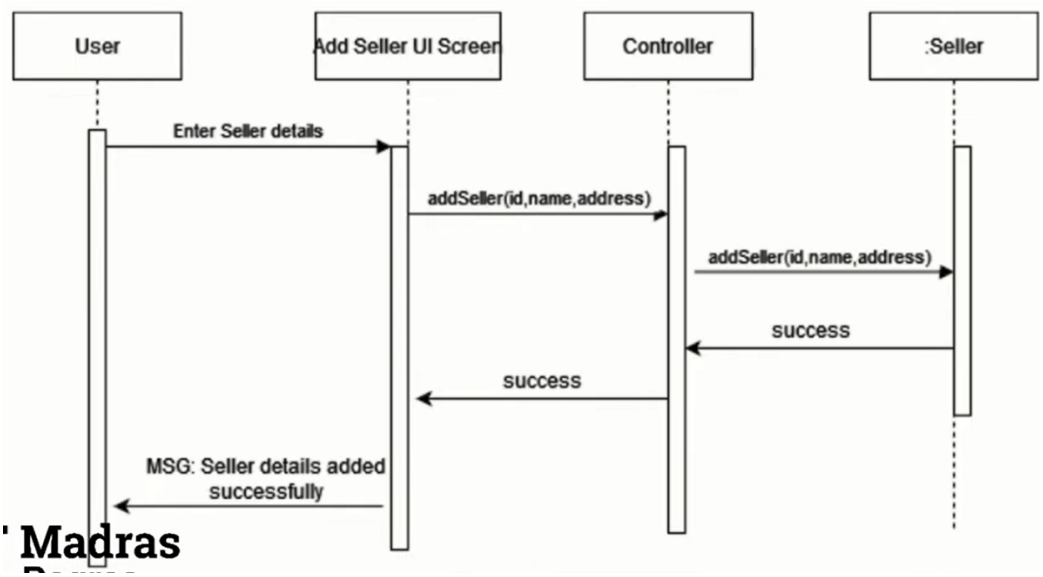


Figure 6: Sequence Diagram for adding seller using Interaction View

- **Purposes of Modelling**
  1. Serves as a vehicle for communication and idea generation.
  2. Guide Development of software
  3. Close correspondence with the implementation. Generate code from models.
- **VeriSIM**: **Veri**fying designs by **Sim**ulating Scenarios. Develop an integrated understanding of class and sequence diagrams. A VeriSIM learning platform can be found here.
- **Design Tracing Strategy**: Construct a state diagram which models the scenario.
- Check chapter 3 of the reference manual for an overview, it can be found here.

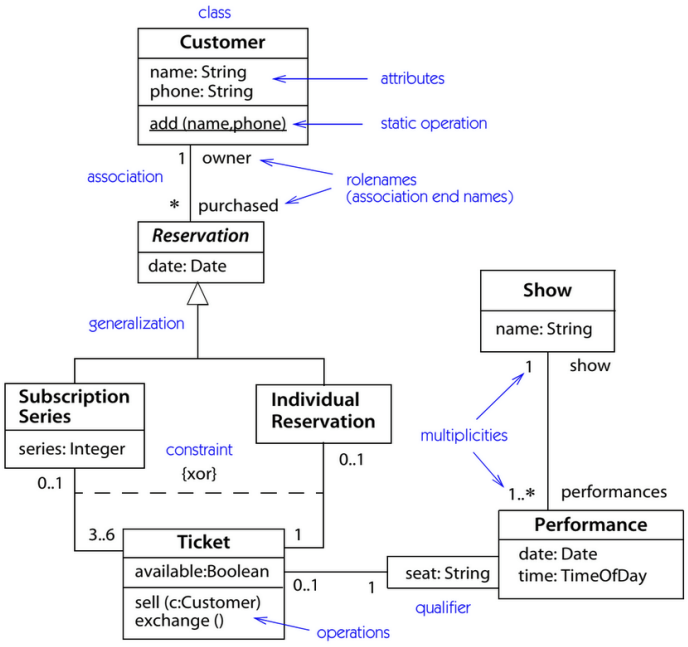


Figure 7: UML Class Diagram

6 Software Development

6.1 Rest APIs

- Use *editor.swagger.io*, can be found here.
- Use Insomnia software for testing downloaded YAML file and code. Download link can be found here.

6.2 Version Control System

- Several developers are working on a project. One developer changed some code that caused a failure and now the system cannot start. Either
  Identify and fix the issue. Could take a long time to debug as you don't know the change history, till then no other team member can test their new code changes as the bug will not let the system start.
  Or just roll back to the previous version of code that was running stable.
- Consider a company with hundreds of clients of a software system. Every client specific feature needs a new version to be released and maintained.
  Huge code vase, code duplicated.
  Several versions result in multiple copies of code. Difficult to maintain and administer these copies manually.
- **VCS**: A system that helps in tracking and managing changes to the **source code** or other documents, and maintaining versions of the code.
- Increases productivity, Better communication and collaboration, Saving space on multiple revisions through diffs/snapshots, Better efficiency even as teams scale.
- **Centralized VCS**: maintained on the server, that everybody shares. Entirely based on client server model. Checkout whole or part of the repository, make changes and push back the changes to the server. Highly dependent on the server.
  Anything goes wrong, the development halts till the server is up again.
  May even lose entire history or repository if no backups configure.
- **Distributed VCS**: Entire repository is mirrored locally that includes the full change history. Centralized repository hosting possible but complete dependency is not on it. Basically everyone owns their own local copies of the repository.
- **Git**: Free and open source VCS. Focus is on speed, data integrity and working on multiple tasks.
- **Modified**: After you change the file in your working directory.
- **Staged**: You marked the file to be committed.
- **Committed**: Changed stored in the local database. But only the staged files will be committed.
- **Create a new repository**: *git init* initializes a new git repository in your current working directory. *git add filename* stage the modified files. *git commit -m "change description"* commit the staged files to DB.
- **Clone a repository**: *git clone url local directory* mirror repo located at "url" to "local directory".
- *git push* to push changes to the remote repo.
- *git pull* fetch and download changes from remote rep and update local repo.
- *git status* view the status of your local files in the working directory and staging area.
- *git diff* show changes between commits, commit and working tree
- *git reset HEAD filename* unstage the file
- *git checkout filename* undo the changes that are committed but not yet pushed to remote.
- *git log* check commit logs/history.
- **Branching**: Branches are independent versions of repository. Mechanism to diverge work from the main project line.
  git branch feature1 create a new branch named feature1 from the base branch you are working on.
  git branch list all local branches
  git checkout command to switch branch, *git checkout feature1*

- **Merging**: A way to combine changes made through one or more branches to a single branch. *git merge feature1* merge the branch "feature1" to the current working branch.
- **Rebase**: Incorporate changes from one branch to another. Incorporate latest changes from master branch to feature1 branch. *git rebase*.
- For more details refer git-scm book, can be found here.
- An issue is reported by developer or tester or user when encountered, this needs to be saved and tracked till it is fixed. We need a system to do this.
- Can create a branch or pull request to map to an issue. Can map more than one issues to a branch.
- **Code Reviewing**: Quality assurance in which multiple people examine the changes done by a developer.
- Improves code Quality, Focus not just on correctness but also on aspects like efficiency, complexity and security.
- Minimize Technical debt, Well documented code with consistent design and implementation reduces maintenance costs and efforts.
- Risk reduction, Testing cannot guarantee software to be completely bug free.
- Supports Knowledge transfer, Learn from others expertise.
- Make QA testing easier, Many risks can be identified at earlier stage itself.
- **Pair programming**: An agile software development technique in which two programmers work together at one workstation. One writes code while the other reviews the code. The two programmers switch roles frequently. Getting popular slowly.

6.3 Debugging

- **Error**: discrepancy between actual behaviour and intended behaviour
- **Failure**: Observable error, Incorrect output value, exception etc.
- **Fault**: Where the failure has occurred (lines in code)
- **Debugging**: determining the cause of failure
- Reproduce the problem → Find cause of defect → Investigate fix → Implement fix → Test fix
- **Logging**: insert print statements
- **Dump & diff**: use diff tool to compare logging data between executions
- **Stepping in debugger**
- **Profiling tool**: how often, how long various parts of program are executed.
- **Input manipulation**: Edit inputs, 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 behavior, follow control flow forward until incorrect state reached
- **Black box debugging**: Find documentation, code examples to understand correct use of API
- Pdb: Python Debugger

6.4 Software Metrics

- Quantitative way to measure the quality of your code
- **Cyclomatic Complexity**: Number of decisions a block of code contains +1, use *Radon*(python package).

Construct	Effect on CC	Reasoning
if	+1	An `if` statement is a single decision.
elif	+1	The `elif` statement adds another decision.
else	+0	The `else` statement does not cause a new decision. The decision is at the `if`.
for	+1	There is a decision at the start of the loop.
while	+1	There is a decision at the `while` statement.
except	+1	Each `except` branch adds a new conditional path of execution.

Figure 8: Cyclomatic Complexity effects

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated program length: $\widehat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \log_2 \eta$
- Difficulty: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$
- Effort: $E = D \cdot V$
- Time required to program: $T = \frac{E}{18}$ seconds
- Number of delivered bugs: $B = \frac{V}{3000}$.

(a) Halstead's Metrics

CC score	Rank	Risk
1 - 5	A	low - simple block
6 - 10	B	low - well structured and stable block
11 - 20	C	moderate - slightly complex block
21 - 30	D	more than moderate - more complex block
31 - 40	E	high - complex block, alarming
41+	F	very high - error-prone, unstable block

(b) CC Score

- **LOC**: The total number of lines of code.
- **LLOC**: The number of logical lines of code. Every logical line of code contains exactly one statement
- **SLOC**: The number of source lines of code
- **Comments**: The number of comment lines
- **Code Smells**: Certain problematic characteristics in a code
- Redundant comments, Commented out code
- Functions doing more than one thing, Functions that are too long, Functions having too many arguments, Functions having flag arguments, Dead functions
- **Don't repeat yourself (DRY)**: Clumps of identical code throughout the program, Put this code in simple methods, Incorrect behaviour at the boundaries, Use explanatory variables
- **Refactoring**: Changing the code by improving its structure, without changing its behaviour.

7 Software Architecture

7.1 Introduction

- Way of organizing your code
- Define software elements/modules, relations among them, and properties of both elements and relations.
- **Client-server systems**: data is transacted in response to requests
- **Pipe and filter**: data is passed from component to component, and transformed and filtered along the way

- **Model-view-controller**: Architectural style where views of the data are separated from the manipulations of data
    1. **Model**: the component which models the data required for the service
    2. **View**: the GUI objects, presentation layer, visual representation of the Model
    3. **Controller**: coordinates multiple Views on the screen, and helps users manipulate the model
- **Peer to Peer Architecture**: Distributed application - different systems form nodes, and share resources with each other.
- No centralized system which monitors all transactions of the system
- Nodes in the network make processing power, disk storage etc. directly available to other nodes in the network
- **Component**: A well-defined functionality or behavior separate from other functionality and behavior
- **Connectors**: Code that transmit information between components. Responsible for regulating interactions between components
- **Protocols**: Set of pre-defined rules which describe how components should interact with each other
- **Design Patterns**: Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context
- **Design Smells**: warning signs that your code may be heading towards an antipattern
- **SOLID Guidelines**: Avoid design smells
- **Refactoring**: moving code between classes, creating new classes or modules, removing classes that aren't required

7.2 SOLID Principles

- Object-oriented principles for software design, Makes code easy to understand, modular
    1. **S**: Single Responsibility Principle, Every class has a single responsibility/purpose
    2. **O**: Open-Closed Principle, Software entities should be open for extension, but closed for modification
    3. **L**: Liskov Substitution Principle, Derived classes should be substitutable by their base class
    4. **I**: Interface Segregation Principle, Specify what the interface should do, not how.
       "do not force any client to implement an interface which is irrelevant to them"
    5. **D**: Dependency Inversion Principle, Prefer abstraction/interfaces over implementations

7.3 Design Patterns

- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
- **Problem**: When to apply the design patterns, in what context
- **Solution**: Describe the elements that make up the design, their relationships, responsibilities and collaborations. Essentially make a template that can be applied in many different situations.
- **Creational**: Used during the process of object creation
    1. **Factory Design Pattern**: Replace object construction calls with calls to a special factory method.
       Pros: Single Responsibility Principle, Open-Closed Principle
       Cons: Code may become more complicated, a lot of new subclasses needed to implement the pattern
    2. **Builder Design Pattern**: Optional parameters necessitates use of multiple constructors, this is known as telescoping constructors.
       Solution: move object construction code it to separate objects called builders
       Pros: Construct objects step by step, Single Responsibility Principle
       Cons: Code may become more complicated, new classes, methods need to be created

- **Structural**: Composition of classes or objects

    1. **Facade Design Pattern**: Reusing code, will have to know details about different objects, functions
       Solutions: Provides a simple interface to a library, or a complex set of classes
       Pros: Isolate code from other libraries/classes' complexity
       Cons: Tightly coupled to other objects, maintenance becomes more difficult
    2. **Adapter Design Pattern**: Some products have cost in dollars/euros. Conversion to rupees is needed
       Pros: separate data conversion code from primary business logic, New types of adapters
       Cons: Overall code complexity increases

- **Behavioral**: Characterize the ways in which classes or objects interact and distribute responsibility

    1. **Iterator Design Pattern**: Different types of collections, How to access and iterate through elements?
       Solution: Separate the behaviour of how elements are accessed into a separate object called an iterator
       Pros: Separate access of elements from other functionalities, new types of iterators, collections
       Cons: Can be an overkill for simple collections
    2. **Observer Design Pattern**: Notify a particular set of buyers when a new product is launched
       Solution: Subject object maintains a list of observers, Notifies them of automatically of any changes
    3. **Strategy Design Pattern**: Process orders based on different strategies
       Solution: Extract different strategies (algorithms) into separate classes, Original class (context) delegates the work to strategy object
       Pros: Isolate implementation details of algorithm, new strategies without changing existing classes
       Cons: Not required if there are only a few algorithms which will be used

8 Testing

8.1 Motivation

- Errors in software programs, even simple errors can cause entire systems to crash.
- **Test case**: triplet $[I, S, R]$
  I: data input to the programming
  S: State of the program at which the data is to be input
  R: result expected to be produced by the program
- **Test suite**: Set of all test cases which have been designed to test a given program.
- Testing for a large collection of randomly selected test cases do not guarantee that all errors will be uncovered.
- Domain of all input values in a software system is sufficiently large.
- Necessary to design a minimal test suite where each test case helps detect different types of errors.
- **Unit testing**: Individual functions/units of a program are tested.
- **Integration testing**: Units are incrementally integrated and tested after each step of integration.
- **System testing**: the fully integrated system is tested
- **Alpha testing**: test team within the organization
- **Beta testing**: select group of customers
- **Acceptance testing**: customer to determine whether to accept the delivery of the software.

8.2 Unit Testing

- While testing a module, other modules with which this module needs to interface may not be ready.
- Makes debugging easier.
- When: During the coding of the module. Not in the testing phase
- Who: Person writing the code for the module.
- Python unit testing: *unittest* and *pytest*

8.3 Black box and White box testing

- **Black box testing**: Examining input/output values only, No knowledge of design or code required
- **Equivalence Class Partitioning**: Domain of input values partitioned into a set of equivalence classes. Program behaves similarly for every input data in a particular equivalence class.
- **Boundary Value Analysis**: Examine the values at boundaries of the equivalence classes.
- **White box testing**: Analyze the structure of the program using some heuristics
- **Branch coverage**: Every branch in the program needs to be taken at least once
- **Multiple branch coverage**: Each component condition takes true and false values
- **Path coverage**: access all linearly independent path at least once.

8.4 Integration and System testing

- When at least a few or all modules have undergone unit testing
- **Integration testing**: detect errors at the module interfaces
- **Big Bang Approach**: All modules integrated in a single step
- **Bottom up Approach**: Modules of each sub system are integrated, Not necessary to create stubs, Drivers are required.
- **Top-down Approach**: Starts with the root module + 1-2sub-ordinate modules of the root module, Stubs are required.
- **Mixed Approach**: Use both top-down and bottom-up testing
- **Smoke testing**: Carried out before initial system testing, Checking whether basic functionalities are working, Few basic test cases designed
- **Performance testing**: Check whether the system meets non-functional requirements

8.5 Test Driven Development

- Used especially in Agile Processes
- Write tests first for the functionality that we want to implement
- Drive design and development of that functionality from tests
- Express the functionality/feature/requirement in the form of a test
- Create a test Run the test - See it FAIL
- Create the Minimum code to meet the needs of the test
- Run it and See it PASS
- REFACTOR code - quality, make it modular, more elegant

9 Software Deployment

9.1 Development Environment and Strategies

- **Development Environment**: Local environment of a software developer, Contains IDE and other tools
- After Development or during we have a testing environment.
- **Staging Environment**: Exactly resembles the production environment, Run on a remote machine
- Deploying a software system involves a lot of activities, configurations etc.
- Staging helps Preview new features, Performance testing
- After staging, we push it to production environment.

- **Blue/Green Deployment**: Staged Deployment, Create a new separate production environment for the new version, without affecting the current one, Regular cycling between real and previous versions
  Advantage: rollback is easy
- **Canary Deployment**: Phased Rollout/Incremental Rollout, Slowly roll out the change to a small subset of users
  Drawback: manage multiple instances at once
- **Versioned Deployment**: Allow users to choose version, Keep all versions alive, If user updates, route them to the new version
  Drawback: have to maintain multiple versions

9.2 Deployment Hosting

- Infrastructure required to host applications, Usually hosted on server systems
- no unplanned downtime
- **Bare Metal Servers**: Purchase actual server hardware, Server - CPU, motherboard, RAM, disk
  Advantage: highest performance
  Disadvantage: Most expensive upfront, Time, effort in setting up, maintenance
- **Infrastructure-as-a-service(IaaS)**: IaaS provider provides a part of the infrastructure
  IaaS examples: Digital Ocean, AWS, Linode
  Advantages: Cheaper, No maintenance overhead
  Disadvantages: Each IaaS has its own set of configurations, Shared by others so performance may suffer
- **Platform-as-a-Service(PaaS)**: Provides a software layer as well on which a web app can be deployed
  Examples: Heroku, Google App Engine
  Advantage: very easy to deploy
  Disadvantage: lack of control

9.3 Continuous Integration

- Automating the integration and deployment of software
  1. Developer commits code to version control system
  2. Continuous Integration Server pulls the new code
  3. CIS build and tests the new code
  4. Once code passes all tests, CIS signals Staging/Deployment server
  5. Staging server pulls the code from version control system
  6. Then it builds, tests, and deploys the new code.
  7. Once deployed Staging server signals back to CIS, and the process starts again.
- Maintain a single source repository
- Automate the Build, Build Tools: Ant, Gradle, Builder
- Make the Build Self-Testing, Create automated tests, When a test fails send notification to the developer
- Commit to the main branch everyday
- Infrequent commits can have increased chances of more conflict errors
- Frequent commits encourage developers to break down their works into very small chunks
- Every commit should build the main branch on an integration server
- **Continuous Integration Servers**: Jenkins, Cruise Control
- Fix broken builds immediately, take the system back to the last-known good build
- Write Scripts to automate deployment
- Benefits of Continuous Integration
  1. Reduces deployment time, Avoids last-minute confusion and rush at release dates
  2. Beneficial to users, Can see an initial prototype
  3. Beneficial for developers, pushes developers to create modular code
- However, Initial effort required to setup processes

9.4 Performance and Monitoring

- **Caching**: Store results of common operations in memory, Fetch from memory instead from database, examples include Memcached, Redis
- Different levels of caching are Web browser, Web server(page cache), Database query cache, results of recent queries which haven't changed
- **Asynchronous work queue**: executed outside the HTTP request-response cycle
- Consists of Queue of jobs to be performed, parameters and Pool of workers take multiple jobs from queue
- Quick loading of pages is important, minifying code helps, and compressing static pages.
- Issues can still arise in the live environment
- **Generate reports**: Analyze page resources, find optimization suggestions, check SEO and accessibility metrics and calculate your performance score. One example of this is Lighthouse.
- **Clickstreams**: which sequences of pages do your users visit the most
- **Think time/dwell time**: how long does a typical user stay on a given page
- **Abandoment**: Check various flows in your application
- Embed a small piece of JS in every page on your site
- In canary deployment, check which version is better.

10 Other Aspects

10.1 Software Organizations

- **Marketing Team**: Look for opportunities in the market, to provide value, Conduct market research, identify audiences

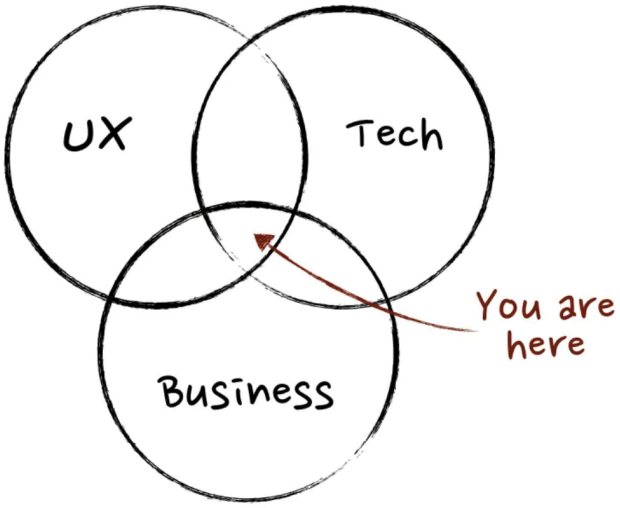


Figure 10: Product Manager

- **Business**: Works closely with marketing teams and sales, Understand business goals, how the product will maximize return on investment
- **Tech**: Should know the technology stack, Understand the level of effort involved, Take important technical decisions.
- **UX**: Passionate about the user, What does the user want
- A good product manager must be experienced in at least one, passionate about all three, and conversant with practitioners in all.

- **Designers**: User Experience (UX) roles, Transform requirements to solutions, Talk to users, create prototypes
- **Software Engineers**: Write code with others in the team to implement requirements, Software engineers don't get to decide what product is made, or what problems the product solves
- **Engineering Managers**: In large organizations, transmitting information between higher and lower parts, Also known as project manager.
- PMs are responsible for Organizing and prioritizing work, Coordinating between different teams, Resolving interpersonal conflict between engineers
- **Sales Team**: Sell the product it to users that marketing team has identified. Provide feedback to marketing, product, and design teams regarding the product, which engineers then address.
- **Support Team**: Resolve problems that clients have, Provide feedback to product, design, and engineering about the product and it's defects/shortcomings
- **Data Scientists**: Analyze data generated from different teams, users, Help organization make better decisions
- **Ethics and Policy Specialists**: People with background in law, social science, policy, Shape the terms of service of the software product, software licenses, privacy policy etc. Important for any company that works with data

10.2 Communication, Collaboration and Productivity

- **Conceptual Integration**: Everyone on a team has the same understanding of what is being built and why, Effective communication ensures conceptual integrity
- **Knowledge Sharing Tools**: used for sharing documents and archiving decisions
- **Issue Tracker**: JIRA, Pivotal Tracker, Track different issues, History of who all worked on these issues
- **GitHub Pages**: Many libraries, frameworks are hosted on GitHub
- **Stack Overflow**: Helps to resolve issues that you are facing, Provides links to additional learning resources
- Knowledge of one project might be needed in another project. Useful if it is documented and archived properly
- When people leave the organization, specialized knowledge goes along with them
- Mitigation: "cross-training", rotating developers between projects
- **Productivity**: Traditionally work done per unit time, this is difficult to define in software engineering, Not necessarily no of lines of code
- **Project Management Tools**: Enables all members of the team to get a big picture as well as detailed view of progress
- **Development Tools**: IDEs, features in IDEs help developers become more productive

10.3 What Makes a Great Software Engineer?

- **Macro Designs**: design and architecture level, e.g. which libraries, frameworks to use
- **Micro Designs**: Algorithms, data structures for a particular module
- Rational Decision-Making Process
    1. Identify the decision to be made
    2. Systematically identify alternatives
    3. Think through potential outcomes
    4. Evaluate which of the outcome is best for the given context
    5. Make a decision