Here's a formatted version of your text to ensure consistency:

### Agility

Latest Model Used Today

Agility is the ability to move quickly and gracefully. It involves:

- **Effective response to change.**
- **Effective communication** among all stakeholders.
- **Characteristics of being dynamic, content-specific,** and **growth-oriented.**

Agility focuses on:

- Drawing the customer into the team and organizing it so that it is in control of the work performed.
- Utilizing lightweight, people-based methods rather than plan-based methods.
- Emphasizing the software itself rather than extensive design and documentation.
- Following an iterative approach.
- Delivering a working model of software quickly to the customer.
- Continuous collaboration with the customer, who is part of the development team.
- Recognizing that the project plan must be flexible.
- Dividing large projects into smaller chunks, developing and testing each chunk, then releasing, getting feedback, enhancing if required, and re-releasing.
- Adapting to changes in requirements and working incrementally on unpredictability.
- Example: Adapting to latest technology updates quickly.

### Q1. What is Agile Model?

Agile Model is a project management approach that involves breaking the project into phases and emphasizes continuous collaboration and improvement.

Agile Software Development is an iterative and incremental approach focusing on delivering a working product quickly and frequently.

The **Agile Manifesto** (2001) outlines four main values and twelve 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 development prioritizes:

- Creating working software quickly
- Frequent collaboration with customers
- Adapting to changes easily

This methodology is especially beneficial for complex projects with uncertain requirements.

Agile Process Breakdown:

1. **Preparation**
   - The product owner creates a backlog of features (product backlog).
   - The development team estimates how long each feature will take.

2. **Sprint Planning**
   - The team decides which features from the product backlog to work on during the sprint.
   - A sprint is a set period (usually two weeks) with specific goals.
   - Tasks are categorized and added to the sprint backlog.

3. **Sprint**
   - The team works on tasks from the sprint backlog.
   - New issues are added to the product backlog if they arise.
   - At the end of the sprint, the team demonstrates completed features and discusses improvements.

### Steps in the Agile Model

The Agile model combines iterative and incremental processes. The steps are:

1. **Requirement Gathering**

   - Interact with the customer to gather requirements.

   - Plan the time and effort needed to build the project.

   - Evaluate technical and economical feasibility.

2. **Design the Requirements**

   - Use user-flow diagrams or high-level UML diagrams to illustrate new features.

   - Design user interfaces and wireframes.

3. **Construction / Iteration**

   - Start working on the project to deploy a working product.

4. **Testing / Quality Assurance**

   - **Unit Testing:** Checks small pieces of code to ensure proper functionality.

   - **Integration Testing:** Identifies and resolves issues when combining different units of software.

   - **System Testing:** Ensures the software meets user requirements and works in all scenarios.

5. **Deployment**

   - Deploy the working project to end users.

6. **Feedback**

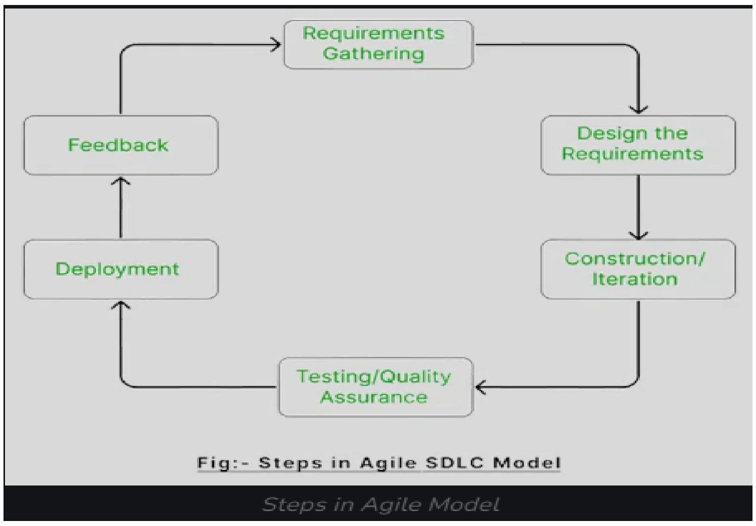
   - Receive feedback about the product.

   - Correct bugs and make improvements based on customer feedback.

Steps in the Agile Model

The agile model is a combination of iterative and incremental process models. The steps involve in agile SDLC models are:

- Requirement gathering
- Design the Requirements
- Construction / Iteration
- Testing / Quality Assurance
- Deployment
- Feedback



Steps in Agile Model

1. **Requirement Gathering:-** In this step, the development team must gather the requirements, by interaction with the customer. development team should plan the time and effort needed to build the project. Based on this information you can evaluate technical and economical feasibility.
2. **Design the Requirements:-** In this step, the development team will use user-flow-diagram or high-level UML diagrams to show the working of the new features and show how they will apply to the existing software. Wireframing and designing user interfaces are done in this phase.
3. **Construction / Iteration:-** In this step, development team members start working on their project, which aims to deploy a working product.
4. **Testing / Quality Assurance:-** Testing involves Unit Testing, Integration Testing, and System Testing. A brief introduction of these three tests is as follows:
    - **Unit Testing:-** Unit testing is the process of checking small pieces of code to ensure that the individual parts of a program work properly on their own. Unit testing is used to test individual blocks (units) of code.
    - **Integration Testing:-** Integration testing is used to identify and resolve any issues that may arise when different units of the software are combined.
    - **System Testing:-** Goal is to ensure that the software meets the requirements of the users and that it works correctly in all possible scenarios.

5. **Deployment:-** In this step, the development team will deploy the working project to end users.
6. **Feedback:-** This is the last step of the **Agile Model.** In this, the team receives feedback about the product and works on correcting bugs based on feedback provided by the customer.

Sprint
During the sprint, the team works on completing the tasks in the sprint backlog. They may also come across new issues to address. If this happens, they will add these issues to the product backlog and prioritize them accordingly. At the end of the sprint, the development team should have completed all features in the sprint backlog.

If not, the team will carry them over to the next sprint. The team then holds a sprint review meeting where they demo completed features to the product owner and stakeholders. They also discuss what went well during the sprint and how they could improve their next one.

Finally, the team holds a retrospective meeting, where they reflect on what went well and what didn't go so well during the sprint. They then create a plan of action for addressing these issues in future sprints. This feedback loop helps to ensure that each sprint is more successful than the last.

Scrum
Here’s a simplified and detailed explanation of Scrum in software development:

### What is Scrum?

Scrum is a framework used in **agile project management**. It helps teams manage and complete their work efficiently by following a set of values, principles, and practices.

Origin: Scrum was introduced in a 1986 Harvard Business Review paper titled *"The New New Product Development Game"* by Hirotaka Takeuchi and Ikujiro Nonaka.

### Key Concepts of Scrum

1. **Framework for Agile Work:**

   - Scrum is a way to apply agile principles to manage and complete projects, especially in software development.

   - Although it's most commonly used in software, its principles can be applied to other types of teamwork.

2. **Core Roles:**

   - **Product Owner:**

     - Acts as the liaison with stakeholders (those interested in the project's outcome).

     - Communicates tasks and expectations to the development team.

   - **Scrum Master:**

     - Facilitates the Scrum process and helps the team adhere to Scrum principles.

     - Ensures the team has what it needs to succeed and removes any obstacles.

   - **Development Team Members:**

     - Work on tasks and organize their own work with guidance from the Scrum Master.

3. **Sprints:**

   - Work is broken down into goals to be completed in **time-boxed iterations** called **sprints**.

   - Each sprint is a set period (typically 2-4 weeks) during which specific work must be completed.

4. **Meetings in Scrum:**

   - **Daily Scrum (Stand-up Meeting):**

     - A short (up to 15 minutes) daily meeting where team members discuss progress, plans, and any issues.

   - **Sprint Review:**

     - Held at the end of the sprint to demonstrate the completed work to stakeholders and get feedback.

  - **Sprint Retrospective:**

    - An internal meeting where the team reflects on what went well and what could be improved.

5. **Phases of Scrum:**

  - **Initiation:** Define the project and goals.

  - **Planning and Estimates:** Break down the work and estimate the effort required.

  - **Implementation:** Work on tasks during the sprint.

  - **Review and Retrospective:** Evaluate the completed work and discuss improvements.

  - **Release:** Deliver the finished product or feature.

### Features of Scrum

- **Lightweight Framework:** Easy to understand and implement.

- **Self-Organization:** Teams are encouraged to manage their own work and collaborate effectively.

- **Continuous Feedback:** Regular reviews and retrospectives help teams adapt and improve.

- **Flexibility:** Allows teams to adapt to changes and incorporate feedback quickly.

### The Role of a Scrum Master

- **Leadership:** Leads the Scrum process and meetings.

- **Problem-Solving:** Helps identify and resolve issues early.

- **Facilitation:** Promotes team collaboration and self-organization.

### Why Use Scrum?

- **Iterative Process:** Scrum helps teams work incrementally, allowing for regular updates and adjustments.

- **Team Collaboration:** Encourages close collaboration and frequent communication among team members.

- **Self-Organizing Teams:** Teams manage their own work and make decisions collectively.

By following Scrum, teams can effectively manage their projects, adapt to changes, and deliver high-quality results efficiently.

Extreme Programming (XP)

Extreme Programming (XP) is a software development method focused on flexibility, teamwork, and high-quality code. It helps teams build software quickly, in small pieces, and adapt to customer changes easily.

### Main Activities in XP:

1. *Planning:*

   - The customer writes simple *user stories* (features they want).

   - The team decides which stories to work on in the next short cycle, called an *iteration* (1-2 weeks).

2. *Design:*

   - Keep designs *simple* and improve them over time (called *refactoring*).

3. *Coding:*

   - *Pair Programming:* Two developers work together on the same code.

   - *Test-Driven Development (TDD):* Write tests first, then write the code to pass the tests.

   - *Continuous Integration:* Add new code often to keep the project working smoothly.

4. *Testing:*

   - Developers write *unit tests* to check each part of the code.

   - Customers help write *acceptance tests* to ensure features work as expected.

5. *Customer Involvement:*

   - The customer works closely with the team, giving *feedback* after each iteration.

### Key Ideas:

- *Frequent releases* of working software.

- *Simple design* and constant improvements.

- Continuous *feedback* and testing to keep the software on track.

XP helps teams respond to changes quickly, build better software, and keep the customer happy.

Pair Programming is a software development practice where two developers work together at one workstation to complete a task. One developer, known as the **"Driver,"** writes the code, while the other, the **"Observer"** or **"Navigator,"** reviews each line of code as it is written and offers suggestions. Here's a detailed explanation and some key points about pair programming:

refractoring pair programing

1. **Roles in Pair Programming:**
  - **Driver:**
    - Writes the code and is focused on the immediate task at hand.
    - Handles the keyboard and mouse, translating ideas into code.
  - **Navigator (Observer):**
    - Reviews the code as it is written, thinking about the overall design and potential issues.
    - Provides suggestions, identifies problems, and ensures the code aligns with the requirements.

2. **Benefits of Pair Programming:**
  - **Improved Code Quality:**
    - Immediate feedback helps catch errors and improve the code quality.
  - **Knowledge Sharing:**
    - Both programmers share their knowledge and skills, which can lead to better code and enhanced team skills.
  - **Faster Problem Solving:**
    - Two minds working together can solve problems more quickly than one.
  - **Reduced Debugging Time:**
    - Continuous review during coding reduces the number of bugs that need to be fixed later.
  - **Increased Team Collaboration:**
    - Encourages communication and collaboration between team members.

3. **Best Practices for Pair Programming:**

  - **Switch Roles Regularly:**

    - Rotate between Driver and Navigator roles to keep both developers engaged and ensure balanced knowledge sharing.

  - **Communicate Clearly:**

    - Discuss ideas, concerns, and decisions openly to ensure both developers are on the same page.

  - **Focus on the Task:**

    - Stay focused on the task at hand and avoid distractions to maximize productivity.

  - **Keep It Collaborative:**

    - Avoid dominating the session; instead, work together to solve problems and make decisions.

  - **Respect Each Other’s Contributions:**

    - Value each other’s input and work collaboratively to achieve the best results.

4. **Challenges of Pair Programming:**

  - **Compatibility Issues:**

    - Differences in working styles or personalities can affect the effectiveness of pair programming.

  - **Potential for Conflicts:**

    - Disagreements or conflicts between partners can disrupt the workflow.

  - **Increased Fatigue:**

    - Working closely together for long periods can be tiring, so it’s important to take breaks.

5. **Pair Programming vs. Solo Programming:**

  - **Pair Programming:**

    - Provides real-time feedback and collaboration.

    - Can lead to higher quality code and more effective problem-solving.

  - **Solo Programming:**

    - Offers more individual focus and less immediate feedback.

    - May lead to longer debugging sessions and less knowledge sharing.

### How to Implement Pair Programming

1. **Choose the Right Partner:**

   - Select a pair with complementary skills and compatible working styles.

2. **Set Clear Goals:**

   - Define the task or project goals to focus the pair programming session.

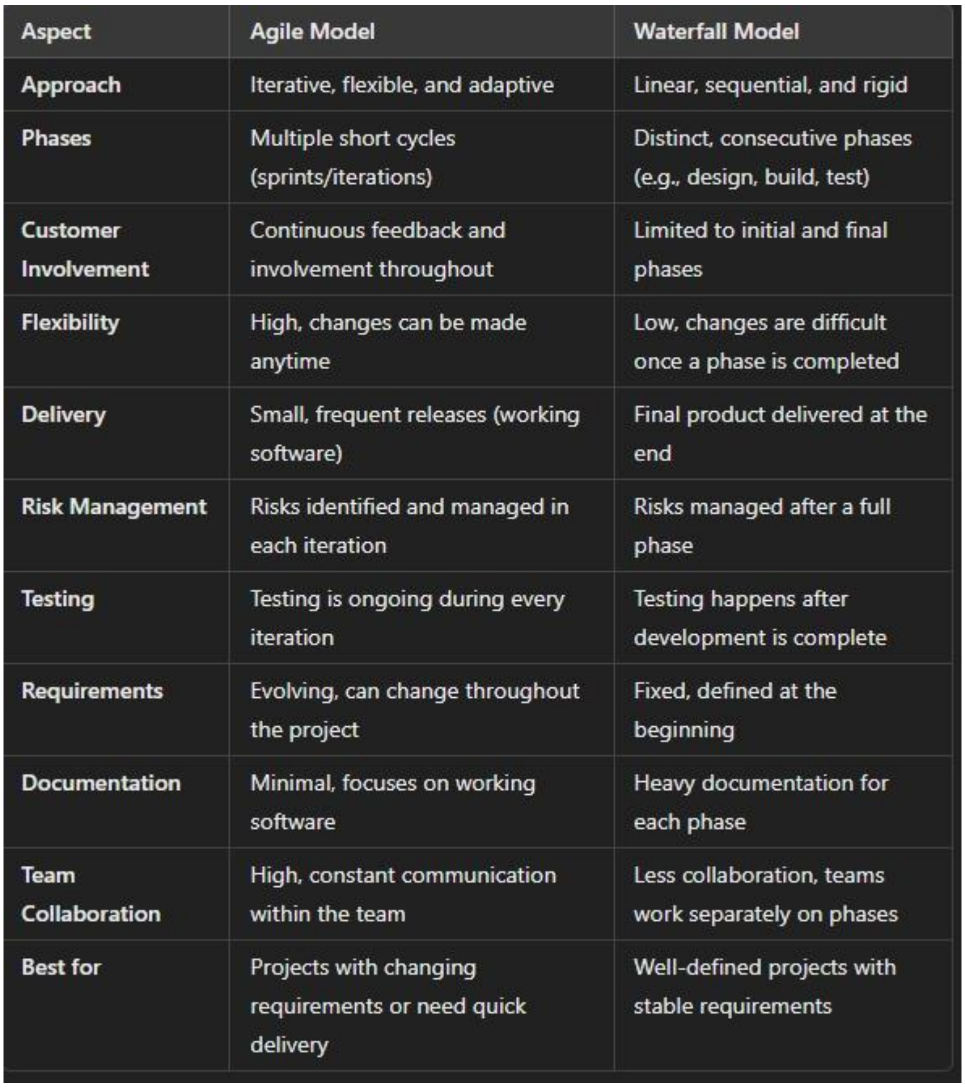
3. **Establish Ground Rules:**

   - Agree on how to handle communication, feedback, and decision-making.

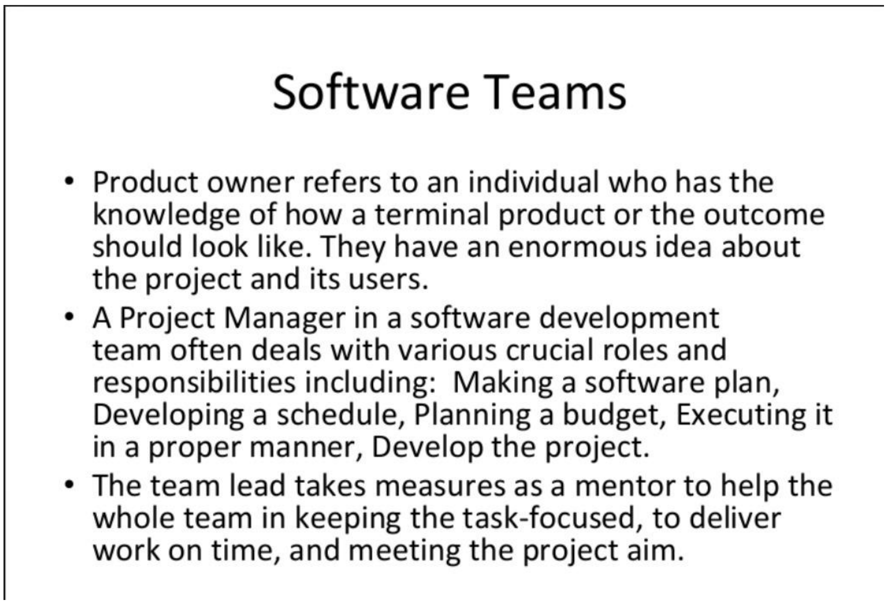
4. **Monitor and Adjust:**

   - Observe the effectiveness of the pair programming sessions and make adjustments as needed.

Pair programming is a valuable practice that, when implemented effectively, can lead to improved code quality, enhanced collaboration, and accelerated problem-solving.



Aspect	Agile Model	Waterfall Model
Approach	Iterative, flexible, and adaptive	Linear, sequential, and rigid
Phases	Multiple short cycles (sprints/iterations)	Distinct, consecutive phases (e.g., design, build, test)
Customer Involvement	Continuous feedback and involvement throughout	Limited to initial and final phases
Flexibility	High, changes can be made anytime	Low, changes are difficult once a phase is completed
Delivery	Small, frequent releases (working software)	Final product delivered at the end
Risk Management	Risks identified and managed in each iteration	Risks managed after a full phase
Testing	Testing is ongoing during every iteration	Testing happens after development is complete
Requirements	Evolving, can change throughout the project	Fixed, defined at the beginning
Documentation	Minimal, focuses on working software	Heavy documentation for each phase
Team Collaboration	High, constant communication within the team	Less collaboration, teams work separately on phases
Best for	Projects with changing requirements or need quick delivery	Well-defined projects with stable requirements



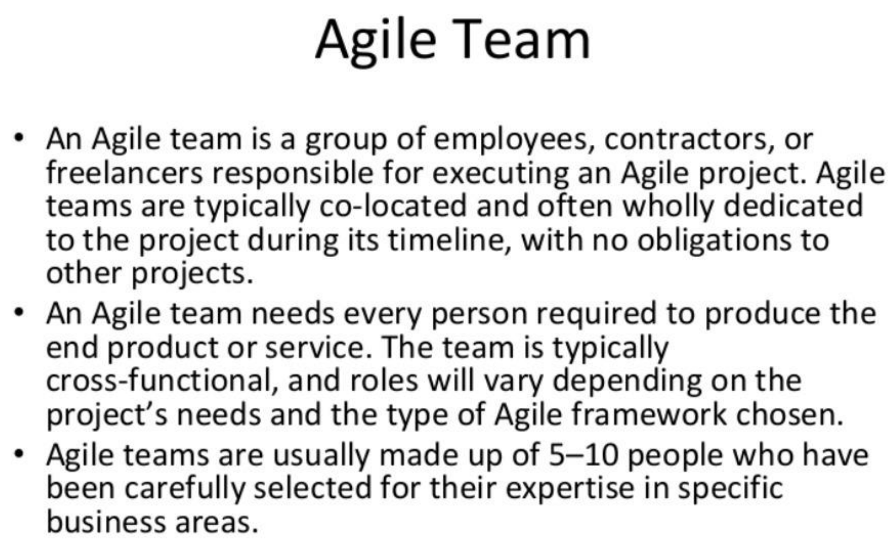
Software Teams

- Product owner refers to an individual who has the knowledge of how a terminal product or the outcome should look like. They have an enormous idea about the project and its users.
- A Project Manager in a software development team often deals with various crucial roles and responsibilities including: Making a software plan, Developing a schedule, Planning a budget, Executing it in a proper manner, Develop the project.
- The team lead takes measures as a mentor to help the whole team in keeping the task-focused, to deliver work on time, and meeting the project aim.

- Software Developers are responsible for utilizing the technical requirements from the technical leads to form cost and deadline estimates. They write code and evolve the software products. Developers are the actual members who write code to make the software function.
- Testers are in-charge of ensuring the software solution meets the demands and complies with the qualities level. They need to understand feature requirements. Also, they form and execute test cases to detect bugs or deficiencies.
- Software Architect designs overall framework and structure of the software. Comes up with system specifications and technical standards. Responsible for ensuring software's security, scalability, and performance.
- Quality Assurance engineer creates testing plans and strategies. Responsible for performing both manual and automated testing. Tracks and reports glitches, bugs, and inconsistencies. Makes sure that the software meets the performance and quality standards.

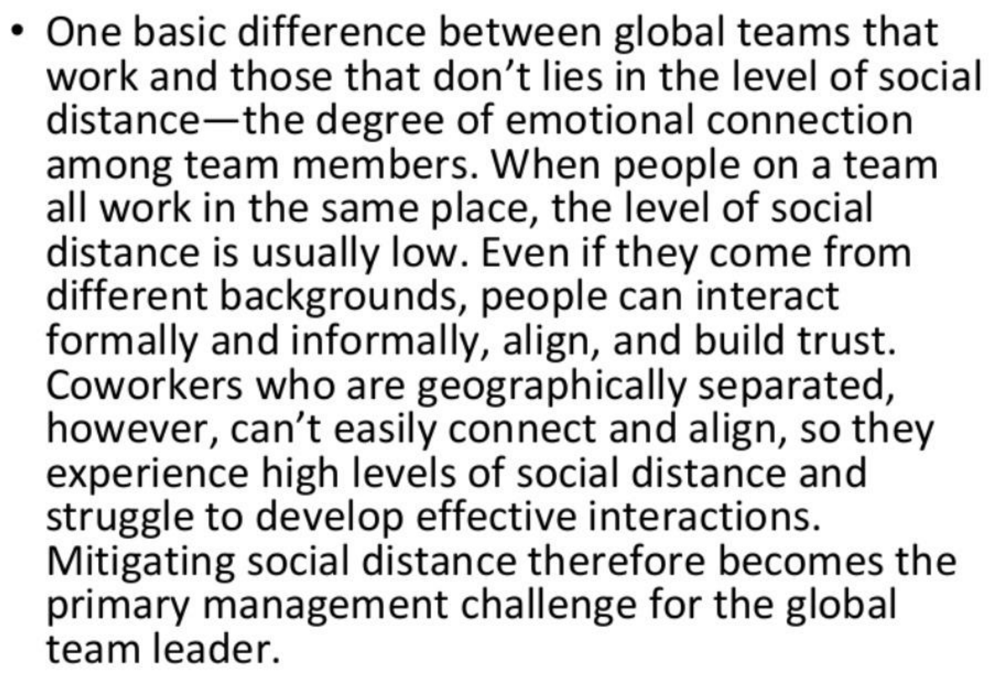
Global Team

- To succeed in the global economy today, more and more companies are relying on a geographically dispersed workforce. They build teams that offer the best functional expertise from around the world, combined with deep, local knowledge of the most promising markets.
- In a classic business model, employees typically work in the same office.
- A global team, on the other hand, consists of employees who work remotely across different countries and time zones.
- When you are part of this type of team, the virtual space is your common meeting ground. Teammates ditch the physical office setup and more traditional approach to take advantage of different types of software and technology to connect and collaborate with one another.



Agile Team

- An Agile team is a group of employees, contractors, or freelancers responsible for executing an Agile project. Agile teams are typically co-located and often wholly dedicated to the project during its timeline, with no obligations to other projects.
- An Agile team needs every person required to produce the end product or service. The team is typically cross-functional, and roles will vary depending on the project's needs and the type of Agile framework chosen.
- Agile teams are usually made up of 5–10 people who have been carefully selected for their expertise in specific business areas.



- One basic difference between global teams that work and those that don't lies in the level of social distance—the degree of emotional connection among team members. When people on a team all work in the same place, the level of social distance is usually low. Even if they come from different backgrounds, people can interact formally and informally, align, and build trust. Coworkers who are geographically separated, however, can't easily connect and align, so they experience high levels of social distance and struggle to develop effective interactions. Mitigating social distance therefore becomes the primary management challenge for the global team leader.

What is Requirement Elicitation?

The process of investigating and learning about a system's requirements from users, clients, and other stakeholders is known as requirements elicitation. Requirements elicitation in software engineering is perhaps the most difficult, most error-prone, and most communication-intensive software development.

Requirement Elicitation can be successful only through an effective customer-developer partnership. It is needed to know what the users require.

Requirements elicitation involves the identification, collection, analysis, and refinement of the requirements for a software system.

Requirement Elicitation is a critical part of the software development life cycle and is typically performed at the beginning of the project.

Requirements elicitation involves stakeholders from different areas of the organization, including business owners, end-users, and technical experts.

The output of the requirements elicitation process is a set of clear, concise, and well-defined requirements that serve as the basis for the design and development of the software system.

Requirements elicitation is difficult because just questioning users and customers about system needs may not collect all relevant requirements, particularly for safety and dependability.

Interviews, surveys, user observation, workshops, brainstorming, use cases, role-playing, and prototyping are all methods for eliciting requirements

Requirements Elicitation Activities

Requirements elicitation includes the subsequent activities. A few of them are listed below:

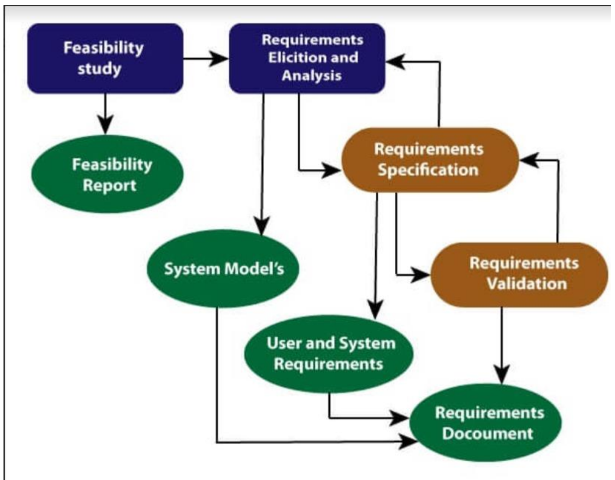
Knowledge of the overall area where the systems are applied.

The details of the precise customer problem where the system is going to be applied must be understood.

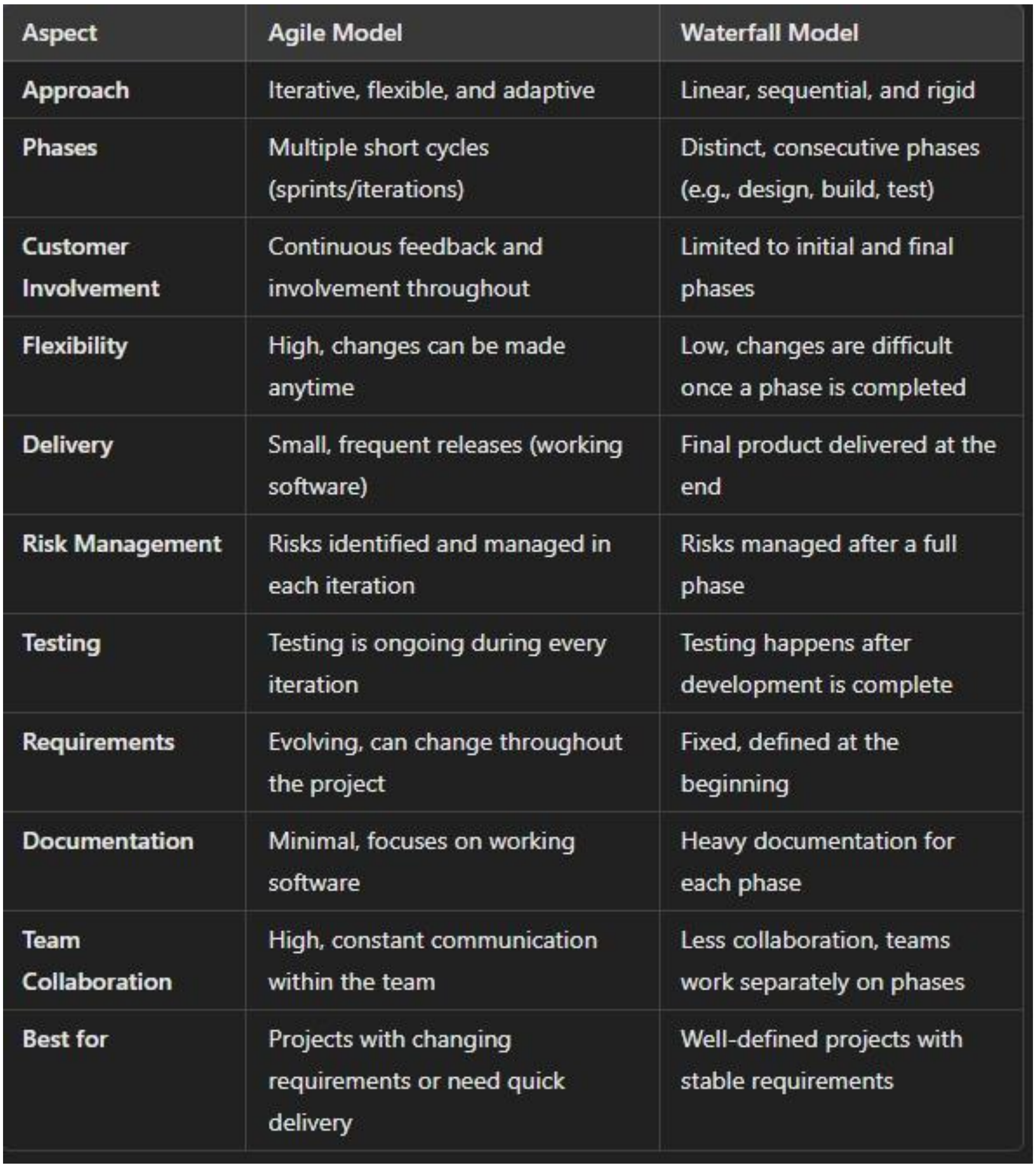
Interaction of system with external requirements.

Detailed investigation of user needs.

Define the constraints for system development.



Requirement Engineering Process



Aspect	Agile Model	Waterfall Model
Approach	Iterative, flexible, and adaptive	Linear, sequential, and rigid
Phases	Multiple short cycles (sprints/iterations)	Distinct, consecutive phases (e.g., design, build, test)
Customer Involvement	Continuous feedback and involvement throughout	Limited to initial and final phases
Flexibility	High, changes can be made anytime	Low, changes are difficult once a phase is completed
Delivery	Small, frequent releases (working software)	Final product delivered at the end
Risk Management	Risks identified and managed in each iteration	Risks managed after a full phase
Testing	Testing is ongoing during every iteration	Testing happens after development is complete
Requirements	Evolving, can change throughout the project	Fixed, defined at the beginning
Documentation	Minimal, focuses on working software	Heavy documentation for each phase
Team Collaboration	High, constant communication within the team	Less collaboration, teams work separately on phases
Best for	Projects with changing requirements or need quick delivery	Well-defined projects with stable requirements

Golden Rules of Software Design

Strive for consistency: Designing "consistent interfaces" means using the same design patterns and the same sequences of actions for similar situations.

Keep users in control: experienced users strongly desire the sense that they are in charge of the interface and that the interface responds to their actions. They don’t want surprises or changes in familiar behavior, and they are annoyed by tedious data-entry sequences, difficulty in obtaining necessary information, and inability to produce their desired result.

Reduce short-term memory load: Interfaces should be as simple as possible with proper information hierarchy, and choosing recognition over recall. It means that cellphones should not require reentry of phone numbers, website locations should remain visible, and lengthy forms should be compacted to fit a single display.

Golden Rules of Software Design

4. Offer Simple Error Handling: A good interface should be designed to avoid errors as much as possible. But when errors do happen, your system needs to make it easy for the user to understand the issue and know how to solve it. Simple ways to handle errors include displaying clear error notifications along with descriptive hints to solve the problem.

5. Design dialogs to yield closure: Sequences of actions should be organized into groups with a beginning, middle, and end. Informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans from their minds, and an indicator to prepare for the next group of actions. For example, e-commerce websites move users from selecting products to the checkout, ending with a clear

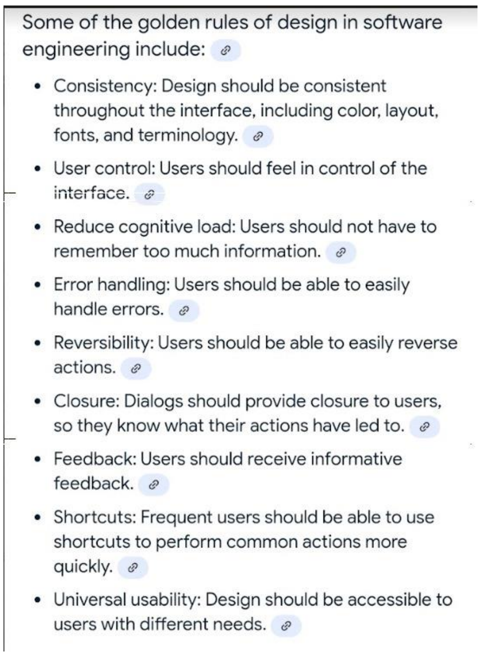
confirmation page that completes the transaction.

Golden Rules of Software Design

6. Offer informative feedback: The user should know where they are at and what is going on at all times. For every action there should be appropriate, human-readable feedback within a reasonable amount of time.

7. Enable frequent users to use shortcuts: With increased use comes the demand for quicker methods of completing tasks. For example, both Windows and Mac provide users with keyboard shortcuts for copying and pasting, so as the user becomes more experienced, they can navigate and operate the user interface more quickly and effortlessly.

8. Permit easy reversal of actions: Designers should aim to offer users obvious ways to reverse their actions.



Some of the golden rules of design in software engineering include:

- Consistency: Design should be consistent throughout the interface, including color, layout, fonts, and terminology.
- User control: Users should feel in control of the interface.
- Reduce cognitive load: Users should not have to remember too much information.
- Error handling: Users should be able to easily handle errors.
- Reversibility: Users should be able to easily reverse actions.
- Closure: Dialogs should provide closure to users, so they know what their actions have led to.
- Feedback: Users should receive informative feedback.
- Shortcuts: Frequent users should be able to use shortcuts to perform common actions more quickly.
- Universal usability: Design should be accessible to users with different needs.

Software Design Process – Software Engineering

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels or phases of design:

Interface Design

Architectural Design

Detailed Design

Interface Design

Interface design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored, and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

Precise description of events in the environment, or messages from agents to which the system must respond.

Precise description of the events or messages that the system must produce.

Specification of the data, and the formats of the data coming into and going out of the system.

Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

Architectural Design

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored. Issues in architectural design includes:

Gross decomposition of the systems into major components.

Allocation of functional responsibilities to components.

Component Interfaces.

Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.

Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Detailed Design

Detailed design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures. The detailed design may include:

Decomposition of major system components into program units.

Allocation of functional responsibilities to units.

User interfaces.

Unit states and state changes.

Data and control interaction between units.

Data packaging and implementation, including issues of scope and visibility of program elements.

Algorithms and data structures.

What is Requirement Elicitation?

The process of investigating and learning about a system's requirements from users, clients, and other stakeholders is known as requirements elicitation. Requirements elicitation in software engineering is perhaps the most difficult, most error-prone, and most communication-intensive software development.

Requirement Elicitation can be successful only through an effective customer-developer partnership. It is needed to know what the users require.

Requirements elicitation involves the identification, collection, analysis, and refinement of the requirements for a software system.

Requirement Elicitation is a critical part of the software development life cycle and is typically performed at the beginning of the project.

Requirements elicitation involves stakeholders from different areas of the organization, including business owners, end-users, and technical experts.

The output of the requirements elicitation process is a set of clear, concise, and well-defined requirements that serve as the basis for the design and development of the software system.

Requirements elicitation is difficult because just questioning users and customers about system needs may not collect all relevant requirements, particularly for safety and dependability.

Interviews, surveys, user observation, workshops, brainstorming, use cases, role-playing, and prototyping are all methods for eliciting requirements

Requirements Elicitation Activities

Requirements elicitation includes the subsequent activities. A few of them are listed below:

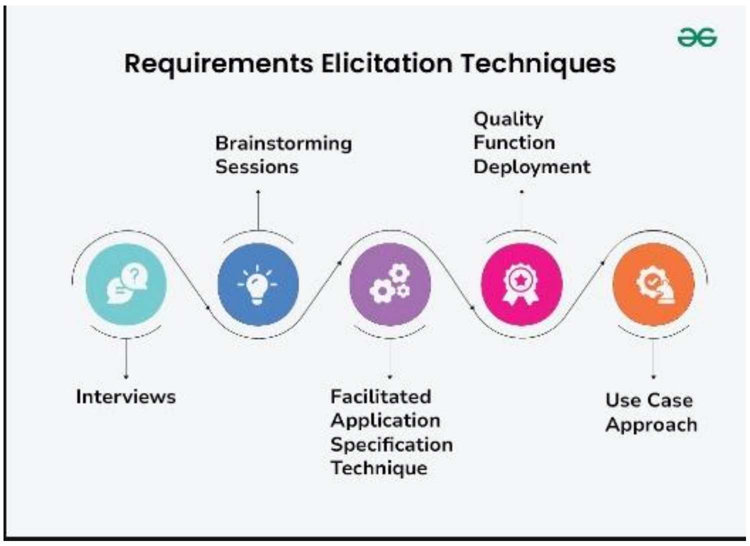
Knowledge of the overall area where the systems are applied.

The details of the precise customer problem where the system is going to be applied must be understood.

Interaction of system with external requirements.

Detailed investigation of user needs.

Define the constraints for system development.


Requirements Elicitation Techniques
Brainstorming Sessions
Quality Function Deployment
Interviews
Facilitated Application Specification Technique
Use Case Approach

1. Interviews

The objective of conducting an interview is to understand the customer’s expectations of the software.

It is impossible to interview every stakeholder hence representatives from groups are selected based on their expertise and credibility. Interviews may be open-ended or structured.

In open-ended interviews, there is no pre-set agenda. Context-free questions may be asked to understand the problem.

In a structured interview, an agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview.

2. Brainstorming Sessions

Brainstorming Sessions is a group technique

It is intended to generate lots of new ideas hence providing a platform to share views

A highly trained facilitator is required to handle group bias and conflicts.

Every idea is documented so that everyone can see it.

Finally, a document is prepared which consists of the list of requirements and their priority if possible.

3. Facilitated Application Specification Technique

Its objective is to bridge the expectation gap – the difference between what the developers think they are supposed to build and what customers think they are going to get. A team-oriented approach is developed for requirements gathering. Each attendee is asked to make a list of objects that are:

Part of the environment that surrounds the system.

Produced by the system.

Used by the system.

Each participant prepares his/her list, different lists are then combined, redundant entries are eliminated, the team is divided into smaller sub-teams to develop mini-specifications and finally, a draft of specifications is written down using all the inputs from the meeting.

4. Quality Function Deployment

In this technique customer satisfaction is of prime concern, hence it emphasizes the requirements that are valuable to the customer.

3 types of requirements are identified:

Normal requirements: In this the objective and goals of the proposed software are discussed with the customer. For example – normal requirements for a result management system may be entry of marks, calculation of results, etc.

Expected requirements: These requirements are so obvious that the customer need not explicitly state them. Example – protection from unauthorized access.

Exciting requirements: It includes features that are beyond customer's expectations and prove to be very satisfying when present. For example – when unauthorized access is detected, it should back up and shut down all processes.

5. Use Case Approach

Use Case technique combines text and pictures to provide a better understanding of the requirements.

The use cases describe the 'what', of a system and not 'how'. Hence, they only give a functional view of the system.

The components of the use case design include three major things – Actor, use cases, and use case diagram.

Actor: It is the external agent that lies outside the system but interacts with it in some way. An actor may be a person, machine, etc. It is represented as a stick figure. Actors can be primary actors or secondary actors.

Primary actors: It requires assistance from the system to achieve a goal.

Secondary actor: It is an actor from which the system needs assistance.

Use cases: They describe the sequence of interactions between actors and the system. They capture who(actors) do what(interaction) with the system. A complete set of use cases specifies all possible ways to use the system.

Use case diagram: A use case diagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system.

A stick figure is used to represent an actor.

An oval is used to represent a use case.

A line is used to represent a relationship between an actor and a use case.

The success of an elicitation technique used depends on the maturity of the analyst, developers, users, and the customer involved.

# Software Requirement Specification:

### Software Requirements Specification

Software Requirements Specification (SRS) is a document created by a software analyst after gathering requirements from various sources. The requirements, initially received from the customer in ordinary language, are translated into technical language to be understood and utilized by the development team.

#### Models Used at This Stage

1. **Data Flow Diagrams (DFDs):**

   - **Purpose:** Widely used for modeling the requirements.

   - **Description:** Shows the flow of data through a system, which can be a company, an organization, a set of procedures, computer hardware, software, or a combination of these.

   - **Also Known As:** Data flow graph or bubble chart.

2. **Data Dictionaries:**

   - **Purpose:** Repositories for storing information about data items defined in DFDs.

   - **Description:** Ensures that the customer and developers use the same definitions and terminologies for data items.

3. **Entity-Relationship Diagrams (E-R Diagrams):**

   - **Purpose:** Detailed logical representation of data for the organization.

   - **Description:** Uses three main constructs:

     - **Data Entities**

     - **Relationships**

     - **Associated Attributes**

### Software Requirement Validation

Verification:

   - **Definition:** Ensures that the software correctly implements a specific function.

Validation:

   - **Definition:** Ensures that the software built is traceable to customer requirements. If not validated, errors in the requirement definitions may propagate to subsequent stages, leading to modifications and rework.

   - **Steps Include:**

- Consistency: Requirements should not conflict with each other.

- Completeness: Requirements should be complete in every sense.

- Achievability: Requirements should be practically achievable.

- Feasibility Checks: Requirements should be checked against the following conditions:

  - Practical implementation

  - Correctness and alignment with functionality

  - Absence of ambiguities

  - Completeness

  - Clear description

### Requirements Validation Techniques

1. **Requirements Reviews/Inspections:**

   - **Description:** Systematic manual analysis of the requirements.

2. **Prototyping:**

   - **Description:** Using an executable model of the system to verify requirements.

3. **Test-case Generation:**

   - **Description:** Developing tests for requirements to ensure testability.

4. **Automated Consistency Analysis:**

   - **Description:** Checking the consistency of structured requirements descriptions.