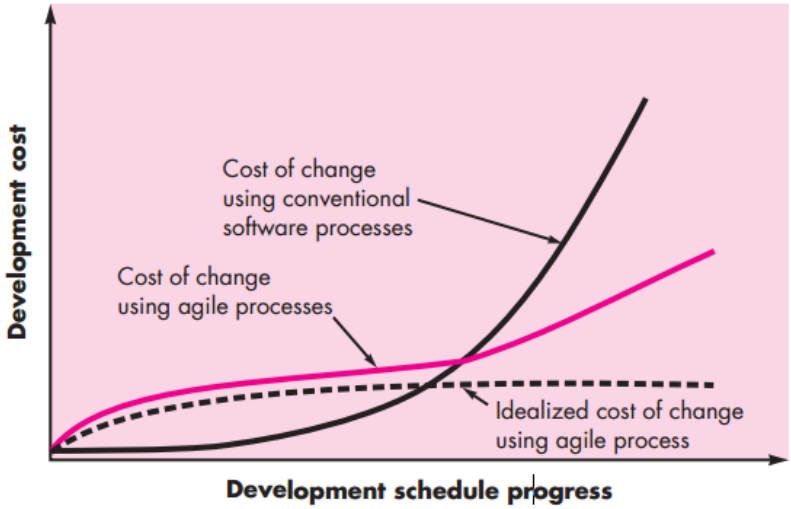
UNIT-2: AGILE DEVELOPMENT

Agility:

- Agility means effective (rapid and adaptive) response to change, effective communication among all stakeholder.
- In software development, the term ‘’agile’’ means “the ability to respond to changes from requirements, technology and people.
- It is an iterative and incremental process.
- Direct collaboration with the customers.
- Agility can be applied to any software process.
- An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.
- It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile (simple or easy).

Agility and the Cost of Change:

- The conventional wisdom in software development is that the cost of change increases nonlinearly as a project progresses (Figure, solid black curve).



- It is relatively easy to accommodate a change when a team is gathering requirements early in a project.
- If there are any changes, the costs of doing this work are minimal. But if the middle of validation testing, a stakeholder is requesting a major functional change. Then the change requires a modification to the architectural design, construction of new components, changes to other existing components, new testing and so on. Costs escalate (rise) quickly.
- A well-designed agile process may “flatten” the cost of change curve by coupling incremental delivery with agile practices such as continuous unit testing and pair

programming. Thus team can accommodate changes late in the software project without dramatic cost and time impact.

Agile Process:

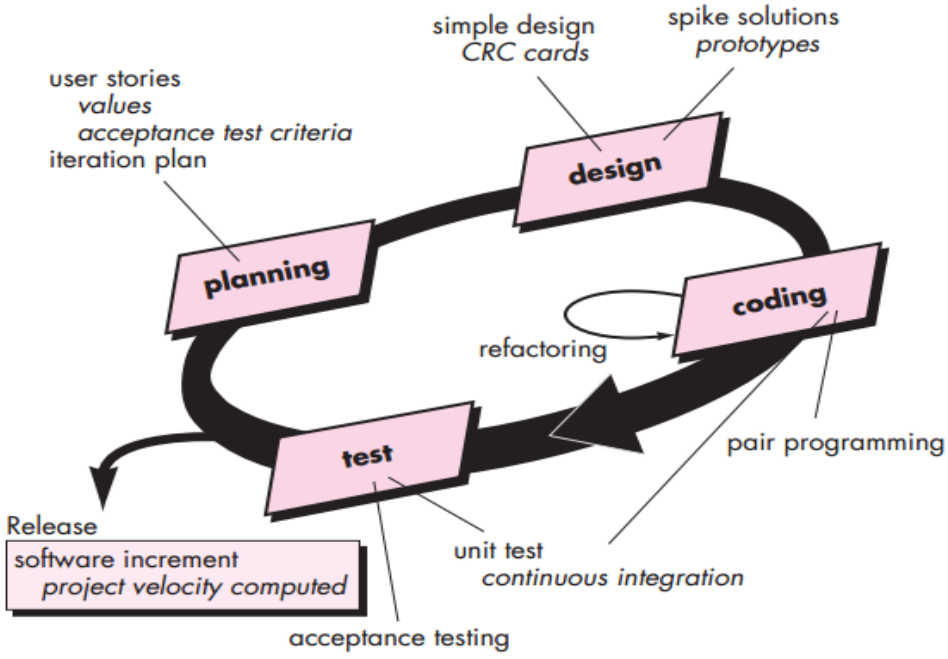
- Is driven by customer descriptions of what is required (scenarios). Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:
  1. Recognizes that plans are short-lived (some requirements will persist, some will change. Customer priorities will change).
  2. Develops software iteratively with a heavy emphasis on construction activities (design and construction are interleaved, hard to say how much design is necessary before construction. Design models are proven as they are created.)
  3. Analysis, design, construction and testing are not predictable.
- Thus has to "Adapt" as changes occur due to unpredictability.
- Delivers multiple 'software increments', deliver an operational prototype or portion of an OS to collect customer feedback for adaption.
- This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.
- **Agility Principles:**

  The Agile Alliance defines 12 agility principles for those who want to achieve agility:
  1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
  3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
  4. Business people and developers must work together daily throughout the project.
  5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
  6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
  7. Working software is the primary measure of progress.
  8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
  9. Continuous attention to technical excellence and good design enhances agility.
  10. Simplicity—the art of maximizing the amount of work not done—is essential.
  11. The best architectures, requirements, and designs emerge from self–organizing teams.
  12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Extreme Programming (XP):

- The most widely used agile process, originally proposed by Kent Beck in 2004. It uses an object-oriented approach.
- **XP Values:**
  1. Beck defines a set of five values that establish a foundation for all work performed as part of XP such that communication, simplicity, feedback, courage, and respect.
  2. In order to achieve effective communication between software engineers and other stakeholders, collaboration between customers and developers, the establishment of effective metaphors (customers, programmers, and managers can tell about how the system works), continuous feedback, and the avoidance of voluminous documentation as a communication medium.
  3. To achieve simplicity, XP restricts developers to design only for immediate needs, rather than consider future needs.
  4. Feedback is derived from three sources: the implemented software itself, the customer, and other software team members.
  5. Beck argues that strict commitment to certain XP practices demands courage. A better word might be discipline. An agile XP team must have the discipline (courage) to design for today, recognizing future requirements, and implemented code.
  6. By following each of these values, the agile team generates respect among it members, between other stakeholders and team members.
- **The XP Process:**

  Extreme Programming (XP) uses an object-oriented approach includes a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure illustrates the XP process.



i. **Planning:**
- Begins with the listening, leads to creation of "user stories" that describes required output, features, and functionality. Customer assigns a value (i.e., a priority) to each story.
- Agile team assesses each story and assigns a cost (development weeks. If more than 3 weeks, customer asked to split into smaller stories).
- A commitment (stories to be included, delivery date and other project matters) is made.

ii. Design:
- It follows the KIS principle (keep it simple).
- Encourage the use of CRC (class-responsibility-collaborator) cards in an object-oriented context. The CRC cards are the only design work product produced as part of the XP process.. They identify and organize the classes that are relevant to the current software increment.
- For difficult design problems, suggests the creation of "spike solutions"—a design prototype for that portion is implemented and evaluated.
- Encourages "refactoring"—an iterative improvement of the internal program design.

iii. Coding:
- Recommends the construction of a unit test for a story before coding commences. So implementer can focus on what must be implemented to pass the test.
- Encourages "pair programming". Two people work together at one workstation. Real time problem solving, real time review for quality assurance. Take slightly different roles.

iv. Testing:
- All unit tests are executed daily and ideally should be automated. Regression tests are conducted to test current and previous components.
- "Acceptance tests" are defined by the customer and executed to assess customer visible functionality.

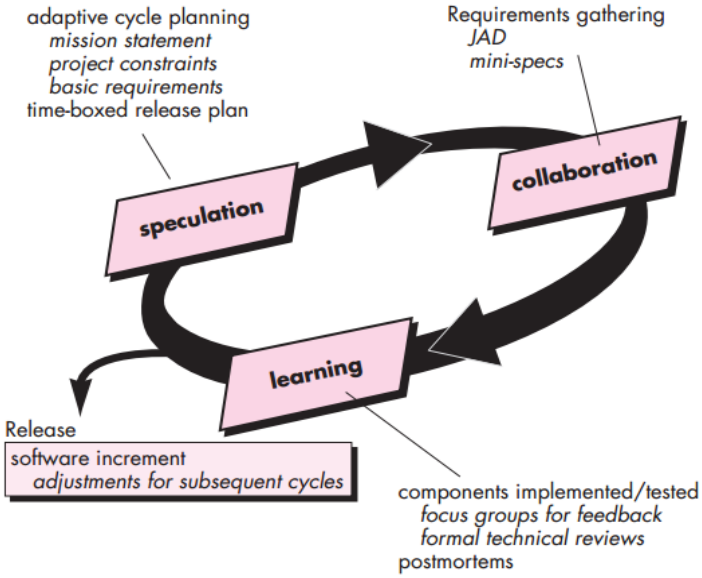
- **The XP Debate:**
  1) **Requirements volatility:** customer is an active member of XP team, changes to requirements are requested informally and frequently.
  2) **Conflicting customer needs:** different customers' needs need to be assimilated. Different vision or beyond their authority.
  3) **Requirements are expressed informally:** Use stories and acceptance tests are the only explicit manifestation of requirements.
  4) **Lack of formal design:** XP deemphasizes the need for architectural design. Complex systems need overall structure to exhibit quality and maintainability. Proponents said incremental nature limits complexity as simplicity is a core value.

OTHER AGILE PROCESS MODELS:

The most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. Among the most common are:

1) Adaptive Software Development (ASD)
2) Scrum
3) Dynamic Systems Development Method (DSDM)
4) Crystal
5) Feature Drive Development (FDD)
6) Lean Software Development (LSD)
7) Agile Modeling (AM)
8) Agile Unified Process (AUP)

1. Adaptive Software Development (ASD):



- Adaptive Software Development (ASD) has been proposed by Jim High smith as a technique for building complex software and systems.
- Phases of ASD:
    a. Speculation
    b. Collaboration
    c. Learning

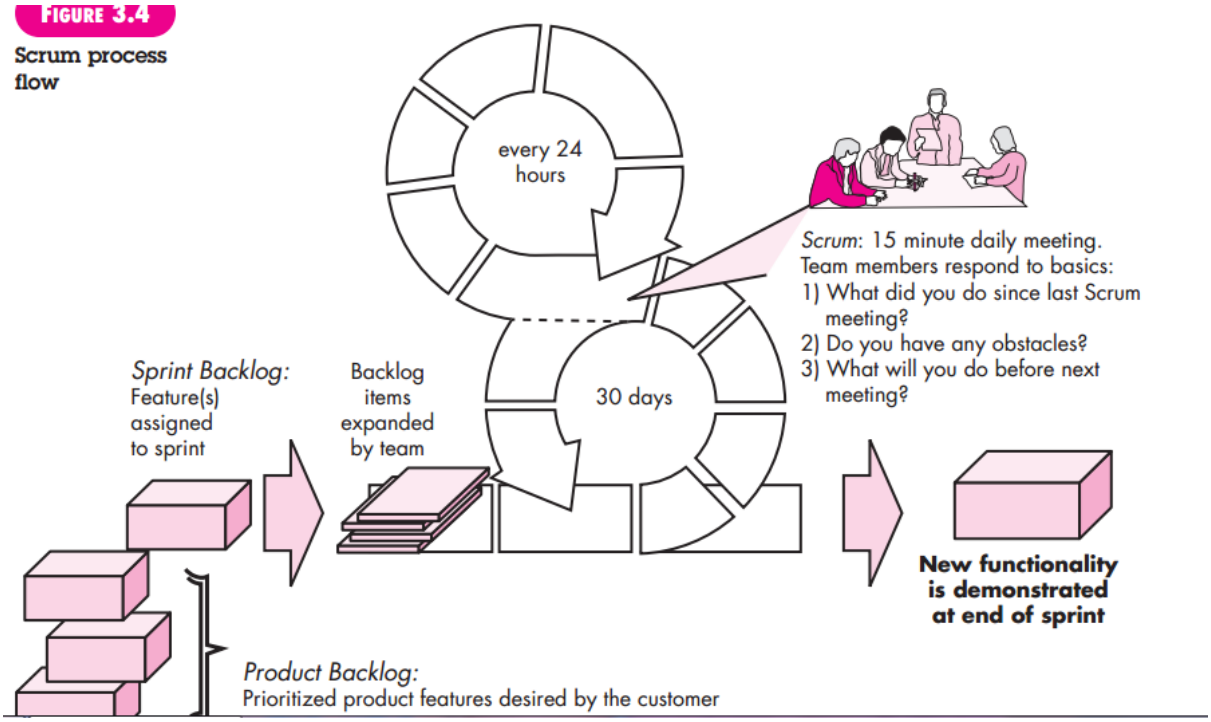
a. Speculation: project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning includes the customer's mission statement, project constraints (e.g. delivery date), and basic requirements to define the set of release cycles (increments) that will be required for the project.

b. Collaboration: It includes communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking.

c. Learning: As members of ASD team begin to develop the components, the emphasis is on "learning". Learning will help developers to improve their level of real understanding. Learning is expressed in three ways: focus groups, technical reviews and project postmortems.

2. Scrum:

FIGURE 3.4
Scrum process flow



- Scrum principles are consistent with the agile manifesto and are used to guided evelopment activities within a process that incorporates the following framework activities: requirements gathering, analysis, design, evolution, and delivery.
- Within each framework activity, work tasks occur within a process pattern called a sprint (time-box).
- The work conducted within a sprint (the number of sprints required for each framework activity) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements. Each of these process patterns defines a set of development actions:
    a. **Backlog**—a prioritized list of project requirements (story points) or features that provide business value for the customer. The product manager assesses the backlog and updates priorities as required.
    b. **Sprints**—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days).

c. **Scrum meetings**—are short (usually 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members:
   i) What did you do since the last team meeting?
   ii) What obstacles are you encountering?
   iii) What do you plan to accomplish by the next team meeting?

- A team leader, called a **Scrum master**,leads the meeting and assesses the responses from each person.
- **Demos**— are delivered to the customer with the time-box allocated. May not contain all functionalities. So customers can evaluate and give feedbacks.

3. Dynamic Systems Development Method (DSDM):

- It is an agile software development approach that provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment.
- DSDM is an iterative software process. Promoted by DSDM Consortium.
- The DSDM Consortium is a worldwide group of member companies. The consortium has defined an agile process model, called the DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:
  a. **Feasibility study—**establishes the basic business requirements and constraints.
  b. **Business study**—establishes the functional and information requirements.
  c. **Functional model iteration**—produces a set of incremental prototypes that demonstrate functionality for the customer.
  d. **Design and build iteration**
  e. **Implementation**

4. Crystal:

- Alistair Cockburn and Jim Highsmith created the Crystal family of agile methods.
- Crystal includes the following features:
  1) Actually a family of process models that allow "maneuverability (means changing position)"based on problem characteristics.
  2) Face-to-face communicationis emphasized.
  3) Suggests the use of "reflection workshops"to review the work habits of the team.

5. Feature Drive Development (FDD):

- Originally proposed by Peter Coad et al as object-oriented software engineering process model.
- Like other agile approaches, FDD adopts a philosophy that
  a. Emphasizes collaboration among people on an FDD team.

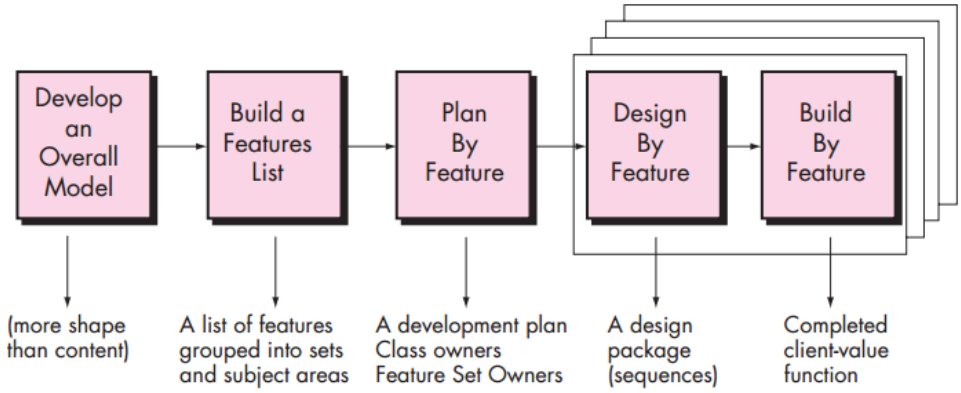
b. Manages problem and project complexity using feature-based decomposition followed by the integration of software increments.
c. In the context of FDD, a feature "is a client-valued function that can be implemented in two weeks or less".
d. Communication of technical detail using verbal, graphical, and text based means.

- The emphasis on the definition of features provides the following benefits:
  a. Because features are small blocks of deliverable functionality, users can describe them more easily, understand how they relate to one another more readily, and better review them for ambiguity, error.
  b. Features can be organized into a hierarchical business-related grouping.
  c. Because features are small, their design and code representations are easier to inspect effectively.
- Coad suggest the following template for defining a feature:

  <action> the**<result> <by for of to>** a(n)**<object>**

  Where an <object>is "a person, place, or thing (including roles, moments in time or intervals of time, or catalog -entry-like descriptions)."

  E.g. 1. Add the product to shopping cart.
  2. Display the technical-specifications of the product.
  3. Store the shipping-information for the customer.



6. Lean Software Development (LSD):

- Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering.
- The lean principles that inspire the LSD process can be summarized as
  a. Eliminate waste,
  b. Build quality in,
  c. Create knowledge,
  d. Defer (means delay) commitment,
  e. Deliver fast,
  f. Respect people, and
  g. Optimize (max efficiency) the whole.

- Each of these principles can be adapted to the software process. For example, eliminate waste within the context of an agile software project can be interpreted to mean:
  (1) Adding no extraneous (irrelevant or redundant) features or functions,
  (2) Assessing the cost and schedule impact of any newly requested requirement,
  (3) Removing any superfluous (not required or redundant) process steps,
  (4) Establishing mechanisms to improve the way team members find information,
  (5) Ensuring the testing finds as many errors as possible,
  (6) Reducing the time required to request and get a decision that affects the software or the process that is applied to create it,
  (7) Streamlining (reform or reshuffle) the manner in which information is transmitted to all stakeholders involved in the process.

7. Agile Modeling (AM):

- AM was originally proposed by Scott Ambler.
- Suggests a set of agile modeling principles listed as follows
  1. **Model with a purpose:** A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.
  2. **Use multiple models**: AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.
  3. **Travel light:** As software engineering work proceeds, keep only those models that will provide long-term value and discard the rest. Every work produce that is kept must be maintained as changes occur. Hence potentially enhancing communication within your team as well as with project stakeholders.
  4. **Content is more important than representation**: A syntactically (means acceptable) perfect model that gives little useful content and it provides valuable content for its audience.
  5. **Know the models and the tools you use to create them:** Understand the strengths and weaknesses of each model and the tools that are used to create it.
  6. **Adapt locally**: The modeling approach should be adapted to the needs of the agile team.

8. Agile Unified Process (AUP):

- The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building computer-based systems.

- By adopting the classic UP phased activities—inception, elaboration, construction, and transition.
- AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project.
- Each AUP iteration addresses the following activities:
    1. **Modeling:** UML representations of the business and problem domains are created. However, to stay agile, these models should be "just good enough" to allow the team to proceed.
    2. **Implementation:** Models are translated into source code.
    3. **Testing:** the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
    4. **Deployment:** Deployment focuses on the delivery of a software increment and the acquisition (means gaining) of feedback from end users.
    5. **Configuration and project management:** In the context of AUP, configuration management addresses change management, risk management. Project management tracks and controls the progress of the team and coordinates team activities.
    6. **Environment management:** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

CORE PRINCIPLES:

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods.

At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

1. Principles That Guide Process:

- **Principle 1: Be agile.**
  keep your technical approach as simple as possible, keep the work products you produce as concise (short) as possible, and make decisions locally whenever possible.
- **Principle 2: Focus on quality at every step.**
- **Principle 3: Be ready to adapt.**
  When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.
- **Principle 4: Build an effective team.**
- **Principle 5: Establish mechanisms for communication and coordination.**

- **Principle 6: Manage change.**
  Mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented
- **Principle 7: Assess risk.**
- **Principle 8: Create work products that provide value for others.**
  Create only those work products that provide value for other process activities, actions, or tasks. Be sure that the work product provides the necessary information without ambiguity.

2. Principles That Guide Practice:

- **Principle 1: Divide and conquer.**
- **Principle 2: Understand the use of abstraction.**
- **Principle 3. Strive (try) for consistency.**
- **Principle 4: Focus on the transfer of information.**
- **Principle 5: Build software that exhibits effective modularity.**
  Modularity must be effective. That is, each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents.
- **Principle 6: Look for patterns.**
  Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Pattern defines our understanding of good architectures that meet the needs of their users.
- **Principle 7: When possible, represent the problem and its solution from a number of different perspectives.**
- **Principle 8: Remember that someone will maintain the software**.

PRINCIPLES THAT GUIDE EACH FRAMEWORK ACTIVITY:

1. Communication Principles:

- **Principle 1: Listen.**
- **Principle 2: Prepare before you communicate.**
- **Principle 3: Someone should facilitate the activity.**
  Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction
- **Principle 4: Face-to-face communication is best.**
- **Principle 5: Take notes and document decisions.**
  Someone participating in the communication should serve as a “recorder” and write down all important points and decisions.
- **Principle 6: Strive(means try) for collaboration.**
- **Principle 7: Stay focused, modularize your discussion.**
  The facilitator should keep the conversation modular, leaving one topic only after it has been resolved.

- **Principle 8: If something is unclear, draw a picture.**
  A sketch or drawing can often provide clarity when words fail to do the job.

2. Planning Principles:

- **Principle 1: Understand the scope of the project.**
  Scope provides the software team with a destination.
- **Principle 2: Involve stakeholders in the planning activity.**
- **Principle 3: Recognize that planning is iterative.**
- **Principle 4: Estimate based on what you know.**
  The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.
- **Principle 5: Consider risk as you define the plan.**
  If you have identified risks that have high impact and high probability, planning is necessary.
- **Principle 6: Be realistic.**
- **Principle 7: Define how you intend to ensure quality.**
- **Principle 8: Describe how you intend to accommodate change.**
  We should identify how changes are to be accommodated as software engineering work proceeds.
- **Principle 9: Track the plan frequently and make adjustments as required.**

3. Modeling Principles:

- **Principle 1:** The primary goal of the software team is to build software, not create models.
- **Principle 2: Travel light—don't create more models than you need.**
  Create models that make it easier and faster to construct the software.
- **Principle 3:** Try to produce the simplest model that will describe the problem or the software.
- **Principle 4:** Build models in a way that makes them amenable (means willing) to change.
- **Principle 5:** Adapt the models you develop to the system at hand.
- **Principle 6:** Get feedback as soon as you can.

4. Construction Principles:

1. **Coding Principles:** The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:
   a. **Preparation principles: Before you write one line of code, be sure you.**
      - Understand of the problem you're trying to solve.
      - Understand basic design principles and concepts.
      - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.

- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed

b. Programming principles:

- Constrain (restrict) your algorithms by following structured programming practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Keep conditional logic as simple as possible.

c. Validation Principles:

- Perform unit tests and correct errors you've uncovered.
- Refactor the code.(code refactoring is the process of restructuring existing computer code).

2. Testing Principles:

- **Principle 1:** All tests should be traceable to customer requirements.
- **Principle 2:** Tests should be planned long before testing begins.
  Test planning can begin as soon as the requirements model is complete.
- **Principle 3:** Testing should begin "in the small" and progress toward testing "in the large."

5. Deployment Principles:

- **Principle 1:** Customer expectations for the software must be managed.
- **Principle 2:** A complete delivery package should be assembled and tested.
- **Principle 3:** A support regime (rule or command) must be established before the software is delivered.
- **Principle 4:** Appropriate instructional materials must be provided to end users.
- **Principle 5:** Buggy software should be fixed first, delivered later.

Requirements

REQUIREMENTS ENGINEERING:

- The broad spectrum (means range) of tasks and techniques that lead to an understanding of requirements is called requirements engineering.
- From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity.
- Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating (means discuss) a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

- It includes seven distinct tasks:
    1) Inception,
    2) Elicitation,
    3) Elaboration,
    4) Negotiation,
    5) Specification,
    6) Validation, and
    7) Management.

1) Inception:

At project inception (beginning), we establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

2) Elicitation:

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.

The following problems that are encountered as elicitation occurs:

- Problems of scope
- Problems of understanding
- Problems of volatility (The requirements change over time).

3) Elaboration:

- This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.

4) Negotiation:

- To resolve the conflicts through a process of negotiation (discussion).
- Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
- Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

5) Specification:

- The term specification means different things to different people.
- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

- For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.

6) Validation:

- Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously. That inconsistencies, and errors have been detected and corrected.
- The primary requirements validation mechanism is the technical review.
- The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content.

7) Management:

Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

ELICITING REQUIREMENTS:

- Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification.
- In order to encourage a communicative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, discuss different approaches and specify a preliminary set of solution requirements.
- There are a number of requirements elicitation methods. Few of them are listed below.
    i) Collaborative Requirements Gathering
    ii) Quality Function Deployment
    iii) Usage Scenarios
    iv) Elicitation Work Products

i) Collaborative Requirements Gathering:

- Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines.
    a) Meetings are conducted and attended by both software engineers and other stakeholders.
    b) Rules for preparation and participation are established.
    c) An agenda is suggested that is formal enough to cover all important points.
    d) A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
    e) A “definition mechanism” (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements

ii) Quality Function Deployment:

- Quality function deployment(QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD "concentrates on maximizing customer satisfaction from the software engineering process".
- QFD identifies three types of requirements listed as follows:
    a) **Normal requirements:** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.
    Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
    b) **Expected requirements:** These requirements are implicit (contained) to the product or system and may be so fundamental that the customer does not explicitly (clearly) state them.
    Examples of expected requirements are: ease of human/machine interaction, reliability, and ease of software installation
    c) **Exciting requirements:** These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

iii) Usage Scenarios:

- The scenarios, often called use cases, provide a description of how the system will be used.
- This 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, use case diagram.

iv) Elicitation Work Products:

- For most systems, the work products include
    a. A statement of need and feasibility.
    b. A bounded statement of scope for the system or product.

c. A list of customers, users, and other stakeholders who participated in requirements elicitation.
d. A description of the system's technical environment.
e. A list of requirements (function) and the domain constraints (non functional) that apply to each.
f. A set of usage scenarios that provide insight into the use of the system or product under different operating conditions

DEVELOPING USE CASES:

- **Use case diagram:** It describes the complete functionality of the system from actor's point of view.
- **Use case:** A use case describes a function provided by the system that provides a visible result for an actor.

Fig: Use Case

- Use cases are used during requirements elicitation and analysis to represent the functionality of the system.
- Use cases focus on the behavior of the system from an external (actor) point of view.
- **Actor:** An actor describes any entity that interacts with the system.
  Examples: User, another system, organization, database etc.

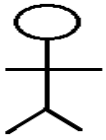
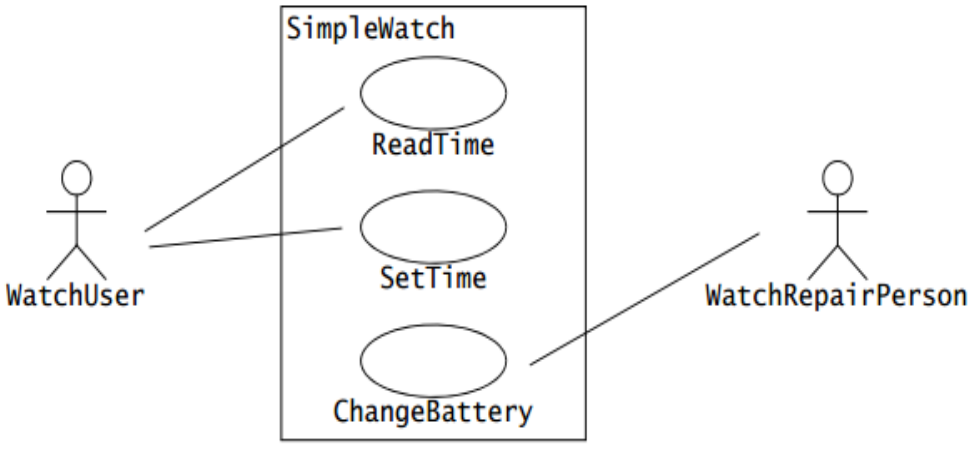


Fig: Actor

- The identification of actors and use cases results in the definition of the boundary of the system.
- The actors are outside the boundary of the system, where as the use cases are inside the boundary of the system.
- **Example:** Figure depicts a use case diagram for a **simple watch**.



- The basic use case presents a high-level story that describes the interaction between the actor and the system.

Template for Use Case:

The following template for detailed descriptions of use cases:

1. Name of the use case: The name of the use case is unique unambiguous.

2. **Participating actors:** Participating actors are actors interacting with the usecase.

3. **Entry Condition:** It describes the conditions that need to be satisfied before the use case is initiated.

4. **Flow of Events:** Flow of Events describe the sequence of interactions of the use case, which are to be numbered for reference.

5. Exit Condition: It describes the conditions satisfied after the completion of the use case.

Example: Use Case description for **Registration** in a website.

1. Name of the Use Case: Registration

2. Participating Actors: User, Database

3. Entry Condition: Enter URL

4. Flow of events: 1. Enter name
2. Select User id
3. Select Pass word
4. Enter email id
5. Enter mobile number

5. Exit Condition: Registration Successfully Completed.

Use Case Diagram-Relationships:

- Use case diagrams can include two types of relationships:
  1) Association
  2) Dependency

1) Association: It is also known as communication relation. Association is defined as a link between use cases and actors.

Communication relationship is represented by using a "solid line arrow" between the actor and use case symbol. i.e.,

" ——————————▶ "

2) Dependency: In this relation, one use case depends on another. Dependency relationship is represented by using a "dotted arrow" between use cases.

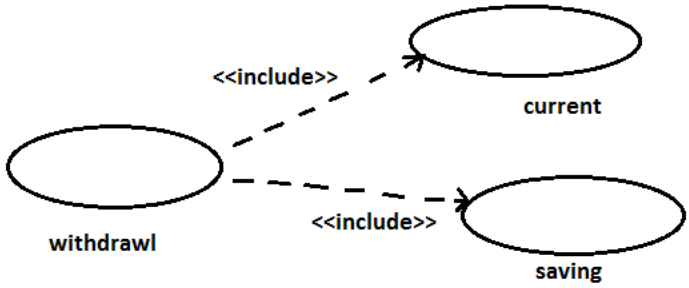
"----------> "

Dependency include

a. include
b. extend
c. uses

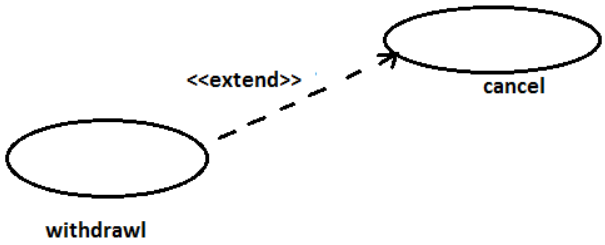
a. include: It explains continuous behaviour of the system.

Ex: Consider ATM Machine – withdrawl function,



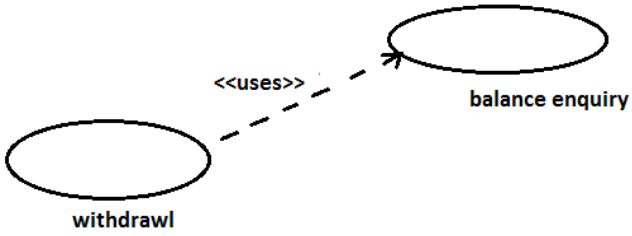
b. extend: It explains exceptional behaviour of the system.

Ex:



c. uses: It describes sub flow in common.

Ex:



BUILDING THE REQUIREMENTS MODEL:

- The intent of the analysis model is to provide a description of the required informational, functional, and behavioural domains for a computer-based system.
- The analysis model is a snapshot of requirements at any given time.

1) Elements of the Requirements Model:

- There are many different ways to look at the requirements for a computer-based system.
- Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering inconsistencies, and ambiguity.

2) Analysis Patterns:

- In software projects, certain problems re-occur across all projects within a specific application domain.
- These analysis patterns suggest solutions (e.g., a class, a function) within the application domain that can be reused when modeling many applications.
- Information about an analysis pattern is presented in a standard template.

NEGOTIATING REQUIREMENTS:

- The intent of this negotiation is to develop a project plan that meets stakeholder needs.
- The best negotiations strive for a "win-win" result.
- That is, stakeholders win by getting the system or product that satisfies the majority of their needs and developer win by working to realistic and achievable budgets and deadlines.
- Negotiation activities:
    i. Identification of the system or subsystem's key stakeholders.
    ii. Determination of the stakeholders' "win conditions."
    iii. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.

VALIDATING REQUIREMENTS:

- As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity.
- A review of the requirements model addresses the following questions:
    I. Is each requirement consistent with the overall objectives for the system/product?
    II. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
    III. Is each requirement bounded and unambiguous?
    IV. Do any requirements conflict with other requirements?
    V. Is each requirement testable, once implemented?
    VI. Does the requirements model properly reflect the information, function, and behaviour of the system to be built?
    VII. Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?
- These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.