Software Processes

Notes by Mannan Ul Haq

Definition:

The software process refers to a structured set of activities needed to develop a software system. It guides how software is created from conception to completion.

Key Components:

- **Specification**: Defining what the software system should do, outlining its features, functionalities, and requirements based on customer needs and expectations.
- **Design and Implementation**: Planning and organizing the internal structure of the software system and then building it according to the design specifications.
- **Validation**: Ensuring that the software behaves as intended and meets the customer's requirements and expectations through testing and verification.
- **Evolution**: Adapting and modifying the software over time to address changing customer needs, technological advancements, and market demands.

Software Process Model:

A software process model is like a blueprint or a roadmap that describes the steps and activities involved in the software development process. It offers a structured approach to software development from a specific perspective.

Incremental Development:

Definition: Building and releasing one feature at a time based on priorities set by the customer.

Process:

1. Gather all requirements.
2. Design the complete product, leaving out details that can be decided later.
3. Slice the product into manageable chunks and build each separately.
4. Integrate each chunk with previously completed parts.

Example: Ecommerce Website:

- Initial release includes basic functionality like search, product information, adding products to cart, and checkout.
- Subsequent releases add features incrementally, such as favorites and customer reviews.

Iterative Development:

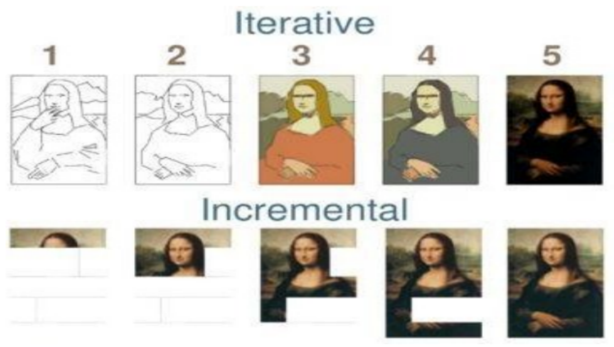
Definition: Discovering and refining requirements as you go, building the overall solution and refining areas as needed.

Process:

1. Start with a basic idea of the product's goals.
2. Design, build, and test a small version of the product.
3. Collect feedback from stakeholders.
4. Keep or discard features based on feedback and iterate.

Example: Ecommerce Website:

- Initial release includes all required functionality in a basic form.
- Subsequent releases improve existing features based on feedback and may add new ideas or requirements.



Plan-Driven and Agile Processes:

Plan-Driven Processes:

- **Definition:** In plan-driven processes, all process activities are planned in advance, and progress is measured against this plan.
- **Characteristics:** Emphasizes detailed planning, strict adherence to the plan, and less flexibility to accommodate changes.
- **Example:** Traditional waterfall model.

Agile Processes:

- **Definition:** In agile processes, planning is incremental, and it's easier to change the process to reflect changing customer requirements.
- **Characteristics:** Emphasizes flexibility, collaboration, and responsiveness to customer feedback throughout the development process.
- **Example:** Agile methodologies like Scrum, and Extreme Programming (XP).

Combination of Approaches:

In practice, most software development processes incorporate elements of both plan-driven and agile approaches to adapt to different project requirements and constraints.

Plan-Driven Processes:

The Waterfall Model:

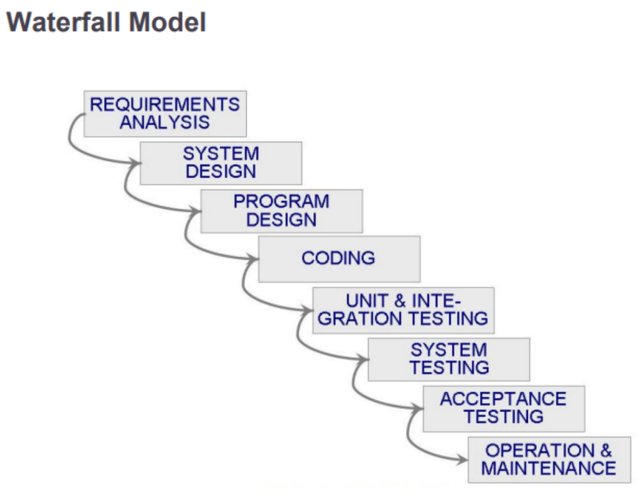
- Sequential steps where each phase (like planning, designing, coding) finishes before the next starts.
- Moves like a waterfall from one phase to another, producing specific outputs at each stage.

Incremental Development:

- Steps like planning, making, and testing happen together in cycles.
- Breaks the work into smaller parts, adding more features with each cycle.

Integration and Configuration:

- Uses existing pieces like libraries or modules to build the whole system.
- Combines these pieces, adjusting them to fit the specific needs of the project.



Waterfall Model Phases:

Process Activities:

1. **Requirements Analysis and Definition**:

   Gathering and documenting user needs and system requirements to define the scope and objectives of the software project.

2. **System and Software Design**:

   Planning and organizing the architecture, structure, and components of the software system based on the specified requirements.

3. **Implementation and Unit Testing**:

Writing code to implement the design and performing unit testing to verify the functionality of individual components or modules.

4. **Integration and System Testing**:

   Combining and testing the integrated system to ensure that all components work together as intended and meet the overall system requirements.

5. **Operation and Maintenance**:

   Deploying the software into production environments, providing support to users, and performing ongoing maintenance and updates to address issues and improve performance.

Waterfall Model Drawbacks:

- **Difficulty in Accommodating Change**:

  The waterfall model makes it hard to adjust to changes once the process starts. Each phase must be finished before moving to the next, making it inflexible to changes in requirements.

- **Inflexible Partitioning**:
  - The project is divided into distinct stages, making it tough to respond to changing customer needs. It's best for projects with well-understood requirements and limited changes during design.
  - This model is mainly suitable for large projects where the system is developed across multiple sites. The plan-driven nature helps coordinate work across different locations.

Software Prototyping:

Definition: Software prototyping involves creating an initial version of a system to demonstrate concepts, explore design options, and gather feedback.

Purpose:

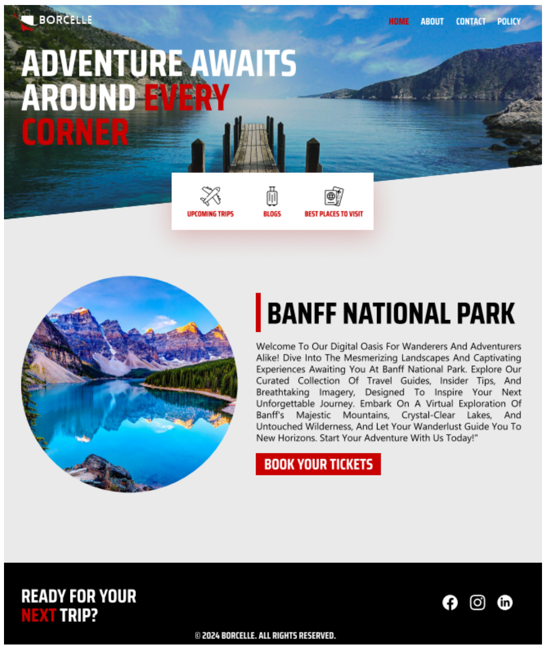
- **Requirements Elicitation and Validation**: Helps in understanding and refining customer requirements.
- **Design Exploration**: Explores different design options and user interface (UI) designs.
- **Testing**: Allows for running tests to evaluate functionality and gather feedback.

Benefits:

- **Improved Usability**: Helps in creating a system that closely matches users' real needs.
- **Enhanced Design Quality**: Allows for refining the design based on early feedback.
- **Reduced Development Effort**: Enables early detection and correction of issues, reducing rework later in the development process.

Prototype Development Process:

- **Method**: May use rapid prototyping languages or tools.
- **Functionality**: Focuses on areas of the product that are not well-understood; may leave out error checking, recovery, and non-functional requirements like reliability and security.

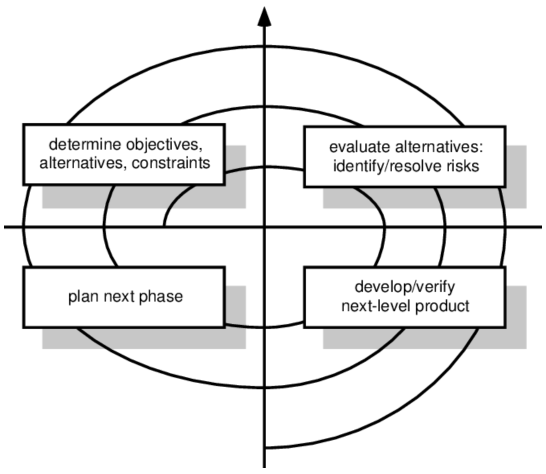


Welcome To Our Digital Oasis For Wanderers And Adventurers Alike! Dive Into The Mesmerizing Landscapes And Captivating Experiences Awaiting You At Banff National Park. Explore Our Curated Collection Of Travel Guides, Insider Tips, And Breathtaking Imagery, Designed To Inspire Your Next Unforgettable Journey. Embark On A Virtual Exploration Of Banff's Majestic Mountains, Crystal-Clear Lakes, And Untouched Wilderness, And Let Your Wanderlust Guide You To New Horizons. Start Your Adventure With Us Today!"

Tool: Figma

Boehm's Spiral Model:

Definition: The Spiral Model is a risk-driven software development process model proposed by Barry Boehm in 1986. It combines elements of iterative development with the systematic approach of the waterfall model.


determine objectives, alternatives, constraints
evaluate alternatives: identify/resolve risks
plan next phase
develop/verify next-level product

Phases:

1. **Planning**: Determining objectives, constraints, and alternatives.
2. **Risk Analysis**: Identifying, analyzing, and mitigating risks.
3. **Engineering**: Developing, testing, and verifying the product incrementally.

4. **Evaluation**: Reviewing the results and determining whether to proceed to the next iteration or phase.

Key Characteristics:

- **Iterative**: Progresses through multiple iterations or cycles, each addressing a specific aspect of the project.
- **Risk-Driven**: Emphasizes risk management throughout the development process, with risk analysis conducted at each iteration.
- **Flexibility**: Allows for adjustments based on feedback and changes in requirements or project conditions.
- **Feedback Loop**: Incorporates feedback from stakeholders and end-users to refine the product incrementally.

When to Use the Spiral Model:

- **Budget Constraint and Risk Evaluation**:

  When there's a tight budget, but it's crucial to evaluate and manage risks effectively.
- **Long-term Project Commitment**:

  When there's a long-term commitment to the project, and requirements may change over time.
- **Uncertain Customer Requirements**:

  When customers are unsure about their requirements, which is common in many projects.
- **Complex Requirements**:

  For projects with complex requirements that need careful evaluation and clarification.
- **New Product Line**:

  When introducing a new product line and releasing it in phases to gather customer feedback.
- **Expected Significant Changes**:

  When significant changes are expected during the development cycle.

Benefits and Drawbacks:

- **Allows Prototyping**:

  Benefits: Prototyping helps in understanding and refining requirements.
- **Management Complexity**:

  Drawback: Managing risks and iterations adds complexity to project management.
- **Early Client Feedback**:

  Benefit: Clients can see the system early on, leading to better collaboration and satisfaction.
- **Not Suitable for Small Projects**:

  Drawback: It's not recommended for small projects due to the overhead of risk management and iteration.

- **Ease of Requirement Changes**:

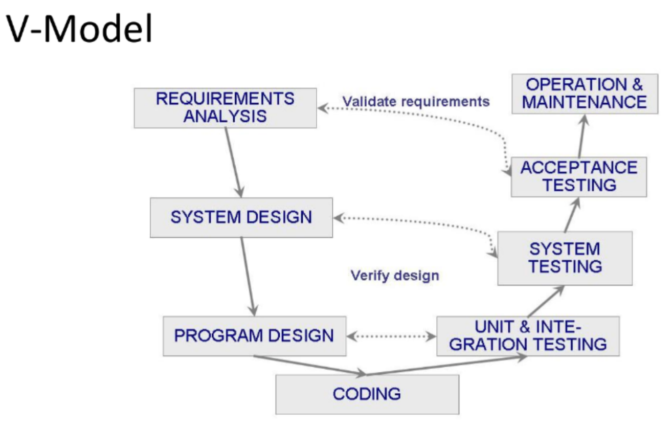
  Benefit: Changes in requirements can be accommodated easily during development.

- **Documentation Requirement**:

  Drawback: Requires extensive documentation to track risks, iterations, and changes.

V-Model:

Definition: The V-Model is a software development model that emphasizes the relationship between each phase of the development life cycle and its associated testing phase.



Key Features:

- **Step-by-Step Phases**:

  Each part of making the software and testing it happens one after the other, in a sequence. When one part is done, its testing part is also done.

- **Checking and Confirming**:

  It's all about making sure the software is made right (verification) and making sure it's the right software (validation).

- **Connecting the Dots**:

  It's like making sure everything in the process connects correctly. From what the software should do (requirements) to how it's made (design) and tested. Everything should be linked and make sense.

Phases:

1. **Requirements Analysis**: Gathering and documenting user needs and system requirements.
2. **System Design**: Planning the architecture and structure of the software system based on requirements.
3. **Module Design**: Designing individual software modules or components.
4. **Implementation**: Writing code to implement the design.

5. **Unit Testing**: Testing individual modules or components.
6. **Integration Testing**: Testing the integration of modules to ensure they work together.
7. **System Testing**: Testing the entire system to ensure it meets specified requirements.
8. **Acceptance Testing**: Testing conducted by the customer to validate the system against their requirements.

Advantages:

- Ensures thorough testing at each stage of development.
- Promotes early detection and correction of defects.
- Provides clear traceability between requirements and testing activities.

Differences Between above Development Models and When to Use Them

Iterative Model

Difference:

- Focuses on repeated cycles of development and refinement.
- Allows for revisiting and refining previous stages based on feedback and learning.

When to Use:

- When project requirements are not fully known from the start.
- When there is a need for flexibility and room for changes during development.
- Suitable for complex projects that benefit from gradual improvement.

Incremental Model

Difference:

- Delivers the system in small, functional increments or parts.
- Each increment is a complete piece of the final product, adding functionality progressively.

When to Use:

- When you need to deliver usable parts of the system quickly.
- When the project can be broken into smaller, independent modules.
- Ideal for projects requiring early deployment of some features.

Plan-Driven Model

Difference:

- Emphasizes detailed planning and a structured approach.
- Follows a predefined sequence of steps, with extensive documentation.

When to Use:

- When requirements are well-understood and unlikely to change.
- Suitable for large projects with clear specifications and a need for predictability.
- Best for projects in highly regulated industries or those requiring thorough documentation.

Spiral Model

Difference:

- Combines iterative development with systematic risk management.
- Each cycle includes planning, risk analysis, engineering, and evaluation.

When to Use:

- When the project involves significant risks that need careful management.
- Suitable for large, complex, and high-risk projects.
- When continuous refinement and risk assessment are crucial for project success.

V-Shape Model

Difference:

- Similar to the waterfall model but emphasizes validation and testing at each stage.
- Each development phase has a corresponding testing phase, ensuring thorough verification.

When to Use:

- When requirements are well-defined and stable.
- Suitable for small to medium-sized projects with straightforward development processes.
- When thorough and systematic testing is a priority at each development stage.

Agile Processes:

What Led to Agile?:

People weren't happy with how much time and effort traditional software methods took. So, they came up with agile methods to focus more on writing code and less on planning and paperwork.

Key Ideas:

- **Code Over Design**: Instead of spending a lot of time planning, agile methods jump straight into writing code.
- **Iterative Approach**: Agile breaks the work into small chunks and keeps improving them bit by bit.
- **Quick Delivery**: The goal is to get working software out fast and keep improving it based on what users need.

Agile Manifesto:

Main Values:

- **People and Interactions**: Agile puts more importance on working together than following strict rules.
- **Working Software**: It values having software that does something over having lots of documentation.
- **Customer Collaboration**: Agile prefers talking and working with customers over making strict contracts.
- **Responding to Change**: It's okay to change plans when needed instead of sticking to a rigid plan.

Principles of Agile Methods:

- **Customer Involvement**: Customers are a big part of the development process. They give input and check the progress regularly.
- **Incremental Delivery**: Software is built piece by piece, with customers deciding what gets added in each part.
- **People, Not Process**: Teams are trusted to find their own best way of working instead of following strict rules.
- **Embrace Change**: Agile expects requirements to change, so it plans for it and designs systems that can adapt easily.
- **Maintain Simplicity**: Keep things simple both in how the software is made and in the process of making it. Complexity makes things harder.

Where Agile Fits:

- **Product Development**: It's great for making small or medium-sized products, like apps, quickly and adapting them as needed.
- **Custom System Development**: When a company needs a software system made just for them, and they're ready to work closely with the developers to get it right.

Plan-driven vs. Agile:

1. **Flexibility**: Plan-driven is rigid, following a fixed plan, while agile is flexible, allowing changes throughout the project.
2. **Feedback**: Agile encourages feedback and adjustments along the way, while plan-driven waits until the end to see if everything went according to plan.
3. **Team Collaboration**: Agile emphasizes teamwork and communication, while plan-driven often relies on individuals following their assigned tasks.
4. **Adaptability**: Agile adapts to changes quickly, while plan-driven might struggle to change course once the plan is set.
5. **Risk Management**: Agile addresses risks early and often, while plan-driven may wait until later stages to identify and address risks.

6. **Customer Involvement**: Agile involves customers throughout the process, gathering feedback and adjusting accordingly, while plan-driven may not involve customers until the end product is ready.
7. **Iterations**: Agile breaks projects into smaller chunks called iterations or sprints, allowing for incremental progress, while plan-driven aims for one big delivery at the end.
8. **Predictability**: Plan-driven aims for predictability, following a detailed plan, while agile accepts that some aspects of the project may change and focuses on delivering value.
9. **Documentation**: Plan-driven relies heavily on upfront documentation, while agile values working software over comprehensive documentation.
10. **Complexity Handling**: Agile handles complexity by breaking it down into smaller, manageable pieces, while plan-driven may struggle with complex projects due to its fixed plan approach.

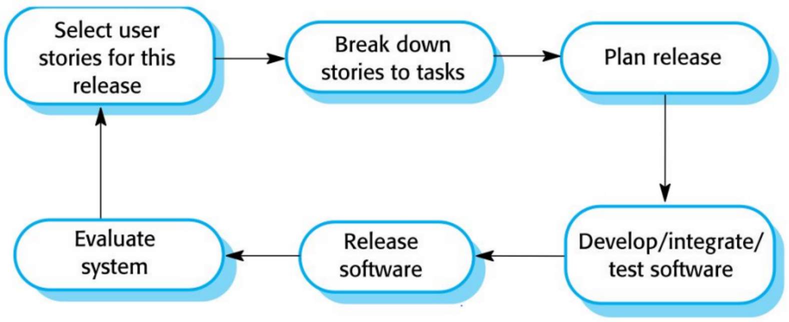
Types:

Extreme Programming (XP):

Introduction: Extreme Programming (XP) emerged as a response to the dissatisfaction with traditional software development methods, aiming to streamline the process and deliver high-quality software quickly.

Extreme Approach:

- XP takes a radical approach to iterative development, where new versions of the software are built frequently, sometimes multiple times a day.
- These iterations are delivered to customers every **two** weeks, allowing for rapid feedback and adaptation.


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

Extreme Programming Practices:

1. **Incremental Planning**:
    - Requirements are recorded on story cards, and the team determines which stories to include in each release based on priority and available time.
    - Stories are broken down into smaller development tasks.
2. **Small Releases**:

- XP emphasizes delivering the minimal useful set of functionality first and then incrementally adding more features with each subsequent release.

3. **Simple Design**:
    - Only enough design work is done to meet current requirements, avoiding over-engineering.
    - XP encourages simplicity in both the software being developed and the development process itself.
4. **Test-First Development**:
    - In Test-Driven Development, we write tests as programs before writing the actual code. This helps us define what the code should do.
    - Tests are written in a way that they can be run automatically. We use frameworks like JUnit for this purpose.
    - Whenever we add new functionality, both new and existing tests are automatically executed. This ensures that any new changes don't break existing functionality.
    - Since testing is automated, we can quickly identify any problems introduced by new code. This helps in fixing issues early in the development process.
5. **Refactoring**:
    - Developers continuously improve the codebase by restructuring it without changing its external behavior.
    - This helps maintain code quality and adaptability.
    - Before, people spent a lot of time guessing what changes might come up in the future. But XP says it's hard to predict changes accurately.
    - With XP, we're always on the lookout for ways to make our code better, even if there's no immediate need. This makes the code easier to understand and work with.
    - It helps us understand the code better, makes it easier to make changes later, and reduces technical debt.
    - **Examples**:
        - Removing duplicate code by restructuring class relationships to improve code reuse and maintainability.
        - Clarifying the names of attributes and methods to make their purpose and functionality clearer.
        - Substituting inline code with calls to pre-existing methods in a program library to promote consistency and efficiency.
6. **Pair Programming**:
    - Both programmers have equal responsibility for the code they write, fostering a sense of shared ownership.

   - Pair programming spreads knowledge and expertise across the team as programmers collaborate and learn from each other.
   - Each line of code is reviewed by both programmers, helping to catch errors and improve code quality in real-time.
   - With two minds at work, pair programming encourages continuous improvement of the codebase through refactoring.
7. **Collective Ownership**:
   - All developers are responsible for the entire codebase, allowing anyone to make changes anywhere.
   - This prevents silos of expertise and promotes collaboration.
8. **Continuous Integration**:
   - Completed work is integrated into the system as soon as possible.
   - After integration, all unit tests in the system must pass to ensure the integrity of the codebase.
9. **Sustainable Pace**:
   - XP discourages working long hours to maintain productivity and code quality.
   - Teams strive for a sustainable pace to prevent burnout and maintain effectiveness.
10. **On-site customer**:

In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Scrum:

Management Approach: Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.

Three Phases:

- **Outline Planning**: Establish general objectives and design software architecture.
- **Sprint Cycles**: Develop increments of the system in short cycles (usually 2-4 weeks).
- **Project Closure**: Wrap up the project, complete documentation, and assess lessons learned.

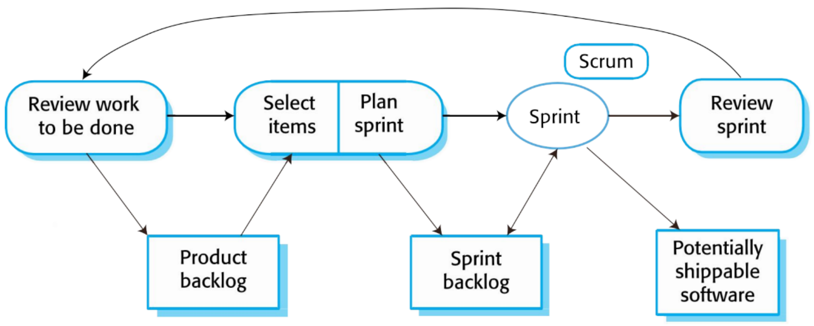
Scrum Terminology:

- **Development Team**: A group of up to **7** self-organizing software developers responsible for creating the software and related project documents.
- **Potentially Shippable Product Increment**: The deliverable from a sprint, intended to be in a finished state and ready for use without further work.
- **Product Backlog**: A list of tasks or requirements to be addressed by the Scrum team, including feature definitions, requirements, user stories, or supplementary tasks like architecture or documentation.

- **Product Owner**: An individual or small group responsible for identifying product features, prioritizing them for development, and continuously reviewing the product backlog to ensure alignment with critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
- **Scrum**: A daily meeting where the Scrum team reviews progress and prioritizes tasks for the day.
- **ScrumMaster**: Responsible for ensuring adherence to the Scrum process, guiding the team, and preventing distractions. Not a project manager but facilitates effective use of Scrum.
- **Sprint**: A development iteration typically lasting **2-4** weeks.
- **Velocity**: An estimate of how much effort from the product backlog a team can cover in a single sprint, aiding in sprint planning and performance measurement.

Sprint Cycle:

- **Planning**: Select features from the product backlog for development.
- **Development**: Team works on developing software in isolation, with communication through the Scrum Master.
- **Review**: Work done is presented to stakeholders for feedback, and the next sprint begins.



Teamwork:

- **Scrum Master**: Facilitates daily meetings, tracks progress, communicates with stakeholders.
- **Daily Meetings**: Short meetings where team members share progress, discuss issues, and plan for the day.

Benefits:

- Breaks down product into manageable chunks.
- Progress is not hindered by changing requirements.
- Improved team communication and visibility.
- Customers see on-time delivery and provide feedback.
- Establishes trust between customers and developers.

Sprint Velocity:

In Agile development, sprint velocity is an estimate of how much work a team can complete in future sprints based on their past performance. Here's how you can estimate sprint velocity:

Step 1: Count Completed Story Points:

- At the end of each sprint, tally up the story points completed by the team.
    - Example:
        - Sprint 1: Committed to 5 user stories, each with 8 story points. Completed 3 stories.
        - Sprint 2: Committed to 7 user stories (including unfinished from Sprint 1), each with 8 story points. Completed 4 stories.
        - Sprint 3: Committed to 9 user stories, each with 8 story points. Completed 5 stories.

Step 2: Calculate Average Completed Story Points:

- Add up the total story points completed across all sprints and divide by the number of sprints.
    - Example:
        - Total completed story points: 3*8 (Sprint 1) + 4*8 (Sprint 2) + 5*8 (Sprint 3) = 96
        - Average sprint velocity: 96 ÷ 3 = 32 story points per sprint.

Using Sprint Velocity:

- With an average sprint velocity of 32 story points, you can plan future sprints accordingly.
- For example, if your project has 160 story points remaining, you can estimate needing around 5 sprints to complete it.

Note:

- For teams new to Agile, track story points completed in initial sprints to establish a baseline for sprint velocity estimation.

Extreme Programming (XP)

Difference:

- Emphasizes frequent releases in short development cycles, improving productivity and introducing checkpoints for new customer requirements.
- Strong focus on technical excellence and good design through practices like pair programming, test-driven development, and continuous integration.

When to Use:

- When requirements are expected to change frequently.
- Suitable for small to medium-sized teams working on projects where customer involvement is high.
- Ideal for projects that require high-quality code and rapid response to changes.

Scrum

Difference:

- Agile framework that breaks the project into fixed-length iterations called sprints (usually 2-4 weeks).
- Focuses on delivering a potentially shippable product increment at the end of each sprint.
- Roles are clearly defined: Product Owner, Scrum Master, and Development Team.

When to Use:

- When the project can be divided into iterative cycles with regular deliverables.
- Suitable for projects with a need for regular customer feedback and flexibility.
- Ideal for teams that can benefit from structured roles and regular sprint reviews.

Rapid Application Development (RAD)

Definition

Rapid Application Development (RAD) is an agile software development methodology that focuses on quickly developing applications through iterative prototyping, without requiring extensive pre-planning. It emphasizes user feedback, reusability of components, and the use of powerful development tools.

Key Characteristics

- **Prototyping**: Developing prototypes that are quickly refined based on user feedback.
- **User Involvement**: Active user participation throughout the development process to ensure the final product meets user needs.
- **Iterative Development**: Rapid cycles of development, feedback, and refinement.
- **Component Reusability**: Using pre-built or existing components to speed up development.
- **Time-boxed Phases**: Each phase or cycle is given a fixed duration to ensure timely delivery.
- RAD projects are often completed in a range of 60 to 90 days.

When to Use

- When the project requires quick delivery and time-to-market is critical.
- Suitable for projects where user requirements are expected to evolve during development.
- Ideal for applications that can benefit from user feedback and incremental improvements.
- When there is access to reusable components and development tools that can accelerate the process.

Difference from Other Models

- **Compared to Waterfall**: RAD is more flexible and adaptive, with less emphasis on extensive upfront planning.

- **Compared to Agile (Scrum, XP)**: RAD specifically emphasizes rapid prototyping and user feedback, whereas Agile methodologies like Scrum and XP have structured roles and processes for iterative development.