## **Software Process Models**

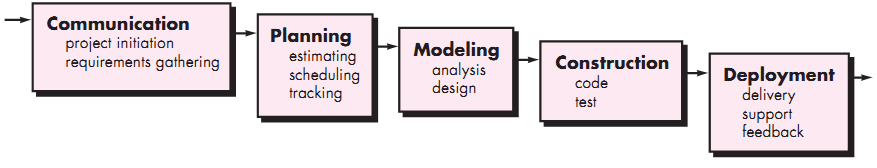
Every process model will have phases; however, there are major differences between models in terms of what activities constitute their phases and the sequencing of the phases. Three types of process models are[1]:

* **Linear process models** – phases that happen sequentially, one after another
* **Iterative process models** – phases that are repeated in cycles
* **Parallel process models** – activities that occur concurrently

Linear process models (The Waterfall model, the V model, and the Sawtooth model) follow a series of phases, one by one, with no return to prior phases. The product is essentially specified, designed, developed, and released in that order, with no opportunity to revisit earlier phases. Linear processes are best suited to projects where very little feedback or refinement is expected. Successful projects following a linear process model are well understood from the start, with little to no room for change [1].

## **Waterfall Model**

The Waterfall model is a linear process model, in which each phase produces an approved work product that is then used by the next phase. The work product flows into the next phase, and this continues until the end of the process, like a waterfall. Based on the waterfall process shown, at the end of the requirements phase, there is an output work product: a requirements document. This document is formally approved and then fed into the next phase: design. This passing of work continues from phase to phase until the software product is the final output [1-2].



**Benefits and Limitation of Waterfall Model**

It may be primitive, but the Waterfall model does have benefits - even in today’s development projects: This process is easy to understand; It clearly defines deliverables and milestones; It emphasizes the importance of analysis before design, and design before implementation; and it can be adopted within teams using more sophisticated processes, for well-specified parts of the product that can be outsourced[1-2].

The Waterfall model faces one major flaw: It is not very adaptable to change. That is, it is not amenable to Agile principles. Modern software development is often very dynamic, flexibility with respect to changing needs and emerging technologies is valued. The Waterfall model focuses on knowing all the requirements up front. It is simply not designed to address mid-stream changes easily or without great cost, because it would require revisiting earlier phases. In addition, testing occurs late in the process, and defects found late tend to be more difficult and costly to fix. Under the Waterfall model, the client does not see the product until close to the end of development, and so the developed product may not match what the client had envisioned[1-2].

## **V-Model**

The V-model is another linear process model. It was created in response to what was identified as a problem of the Waterfall model.It adds a more complete focus on testing the product. The distinguishing featureof the V-model are the two branches that turn a linear structure into a “V” shape. Like the Waterfall model, the V-model begins with analysis and the identification of requirements, which feeds product information into the design and implementation phases. The analysis and design phases comprise the left branch of the V. The right branch represents integration and testing activities. As testing work proceeds up the right branch, some level of the product (on the right side) is actively verified against the corresponding design or requirements (on the left side)[1-2].



**Benefits and Limitation of V-Model**

The V-model has the same advantages and disadvantages as the Waterfall model in that it is straightforward to understand, but it does not accommodate change well. However, allowing the development team to verify the product at multiple levels in explicit phases is an improvement over the Waterfall model. In both the Waterfall model and the V-model, the client does not see the finished product until everything is virtually complete. The next evolution in linear processes is devising a way to involve the client in the development process[1-2].

## **Sawtooth Model**

The Sawtoothmodel is also a linear process model, but unlike the Waterfall and V models, the client is involved to review intermediate prototypes of the product during the process. As the process proceeds linearly, there are phases that involve the client in the top section of the diagram and phases that involve only the development team in the bottom section, which creates a jagged “sawtooth” pattern [1].



Having the client involved earlier in the process increases the likelihood that the product will meet the client’s needs; this is a definite advantage of the Sawtooth model compared the Waterfall model and the V-model. However, like them, the Sawtooth model is still a linear process, so there is a limit to incorporating anything more than incremental changes [1].

**Iterative Models**

Iterative process models differ from linear process models in that they are designed for repeating stages of the process. That is, they are iterative or cyclical in structure. The advantage of iterative processes is the ability to loop and revisit previous phases (and their activities) as part of the process. Each “loop back” is an iteration, hence the name “iterative process.” Iterations allow for client feedback to be incorporated within the process as being the norm, not the exception. Iterative process models are readily amenable to Agile practices, yet they also embody sequential portions reminiscent of linear process models[1-2].

## **Spiral Model**

The Spiral model was introduced by Barry Boehm in 1986. The model outlined a basic process in which development teams could design and successfully implement a software system by revisiting phases of the process after they had been previously completed. A simplified version of the Spiral model has four phases, which have associated goals: determine objectives, identify and resolve risks, develop and test, and plan the next iteration. Taken in order, the four phases represent one full iteration. Each subsequent iteration has the sequence of the four phases revisited, and each iteration results in a product prototype. This allows the development team to review their product with the client to gather feedback and improve the product. Early iterations lead to product ideas and concepts, while later iterations lead to working software prototypes. The Spiral model is commonly charted with the four phases appearing as quadrants, and an outward growing spiral to indicate progression through the phases [1-2].



Estimating work can be more difficult, depending on the duration of the iteration cycle in the Spiral model. The longer the iteration cycle, the further into the future one needs to plan and estimate for; lengthy iterations can introduce more uncertainty in estimates. It is easier to estimate the effort on small things and plan for two weeks ahead than to estimate the effort needed on many big things for several weeks ahead. Also, the Spiral model requires much analytical expertise to assess risks. These extensive risk-assessment tasks can consume a great deal of resources to be done properly. Not all organizations will have the years of experience, data, and technical expertise available for estimation and risk assessment[1-2].

## **Unified Process Model**

The Unified Process model is an iterative and also parallel model of software development. Parallel processes use a similar style of iteration—products are built in iterations. However, parallel processes allow for more concurrency of activities and tasks. Its basic structure has sequential phases within a repeatable cycle. Within most of the Unified Process model’s phases, work happens in small iterations until the phase is deemed complete. Usually, phases are deemed complete when a milestone, a specific and identifiable point in a project, is reached [1].



While the general structure of the Unified Process is iterative, the model allows for tasks done in one phase to also occur in another. So, a requirements task or activity can happen throughout the phases instead of in just one. This also means that, for example, a requirements task, an architecture design task, and a test development task can happen in parallel with the same phase. To contrast this, in the Waterfall model, these tasks would be organized into specific, separate phases, with no parallelism for those tasks [1-2].

**Inception phase**

This first phase is meant to be shorttime to establish a strong enough business case and financial reason to continue on to the next phases and make the product. To do this, the inception phase typically calls for the creation of basic use cases, which outline the main user interactions with the product. Also definethe project’s scope and potential project risks. The inception phase is the only phase in Unified Process model that does not happen in iterations. If the inception phase is long, this might suggest wasted time over analyzing the requirements. The completion of the inception phase is marked by a lifecycle objective milestone. The work product for achieving this milestone is a reasonable description of product viability and the readiness to move on to the next phase of the process[1-2].

**Elaboration phase**

The Unified Process focuses on the importance of refining the product’s architecture over time. Architecture is a set of designs upon which the software product is built. So, the goal of the elaboration phase is to create design models and prototypes of the product as well as to address risks. This phase is also the first of the phases to incorporate small iterations within the phase. Besides defining the system architecture, developers refine the requirements conceived earlier in the inception phase. They also develop key requirements and architecture documentation, such as use case diagrams, and high-level class diagrams. This phase gives the foundation on which actual development will be built. Building a prototype will likely require several iterations before the requirements and architecture models are deemed complete enough to move on. At the end of the elaboration phase, developers deliver a plan for development in the next phase. This plan basically builds on what was developed during the inception phase; it integrates everything learned during the elaboration phase so that construction can happen effectively [1-2].

**Construction phase**

The construction phase has also iterations and focuses on building upon the previous work so far. This is where the software product begins to take shape. Since the Unified Process model is a parallel process, when the construction phase begins, elaboration phase work will still continue. The only difference is that the emphasis on the work may change. Testing and programming activities may have been important in the elaboration phase (for technical feasibility studies or to set up the development environment), but they become even more important in the construction phase. Similarly, assessing risks is important in the inception phase, but it’s less important in the construction phase. In the construction phase, full use cases are developed to drive product development. These improvements upon the basic versions developed in the inception phase consist of a set of possible sequential interactions between users and systems. They help to identify, clarify, and organize functionalities of a product. These use cases are more robust than the ones initiated in the inception phase and offer more specific insights into how the product should support end-user tasks. The product is built iteratively throughout the construction phase until it is ready to be released. At that point, the development team begins transitioning the product to the client and/or the end users[1-2].

**Transition phase**

During this phase, the development team receives feedback from users. The transition phase reveals how well the product design measures up against users’ needs. By gathering user feedback, the development team can make improvements to the product, like bug fixes and developing future updates to be released. Upon completing the transition phase, it is possible to cycle back through the phases of the Unified Process again. For example, there may be a need to create further releases of the product or to incorporate user feedback as a means of influencing the plans for later development[1-2].

## **Prototypes**

Developing software products through a series of intermediate prototypes was a theme in both the Spiral and Unified Process models. There are five types of prototypes.

**Illustrative prototype**

This is the most basic of all prototypes. They could be drawings on a napkin, a set of slideshow slides, or even a couple index cards with components drawn out. The essence of an illustrative prototype is to get a basic idea down. Illustrative prototypes can give the development team a common idea of which to base their work, or to help them get a system’s “look and feel” right without investing much time or money into developing a product. Use illustrative prototypes to weed out problematic ideas or as a guide for later development. Illustrative prototyping can involve storyboarding, or wireframes. They are all meant to show how a system will work or illustrate a concept using only diagrams and pictures[1-2].

**Exploratory Prototype**

Exploratory prototyping puts the focus on more than just the product’s look and feel. By building working code, the development team can explore what is feasible—fully expecting to throw the work out after learning from the prototype. With more time, a more comprehensive look at what the product will look like can emerge, as well as an understanding of the effort needed to build that product. Exploratory prototyping is expensive in terms of consuming resources, but it is used because it is better and less costly than finding out later in the process that a product solution just cannot work. The usual motivation behind exploratory prototyping is that the product developers want to study the feasibility of a product idea. Unlike illustrative prototyping, which is only concerned with what the product looks like, exploratory prototyping is about how realizable it is to develop the product or how useful the product may be, before committing further effort to the idea [1-2].

**Throwaway Prototype**

The first version of almost any product is bound to have various problems. So, why not just build a second, better, version from scratch and toss away the first? These first versions of products are known as throwaway prototypes – you build a functioning product that will get tossed out.Throwaway prototypes allow the opportunity to learn from past mistakes. There could be many useful lessons and problems revealed in the first version that can be avoided in a second version. This affords the chance to build the software product on a more robust second-generation architecture, rather than a first-generation system with patches and fixes[1-2].

**Incremental Prototype**

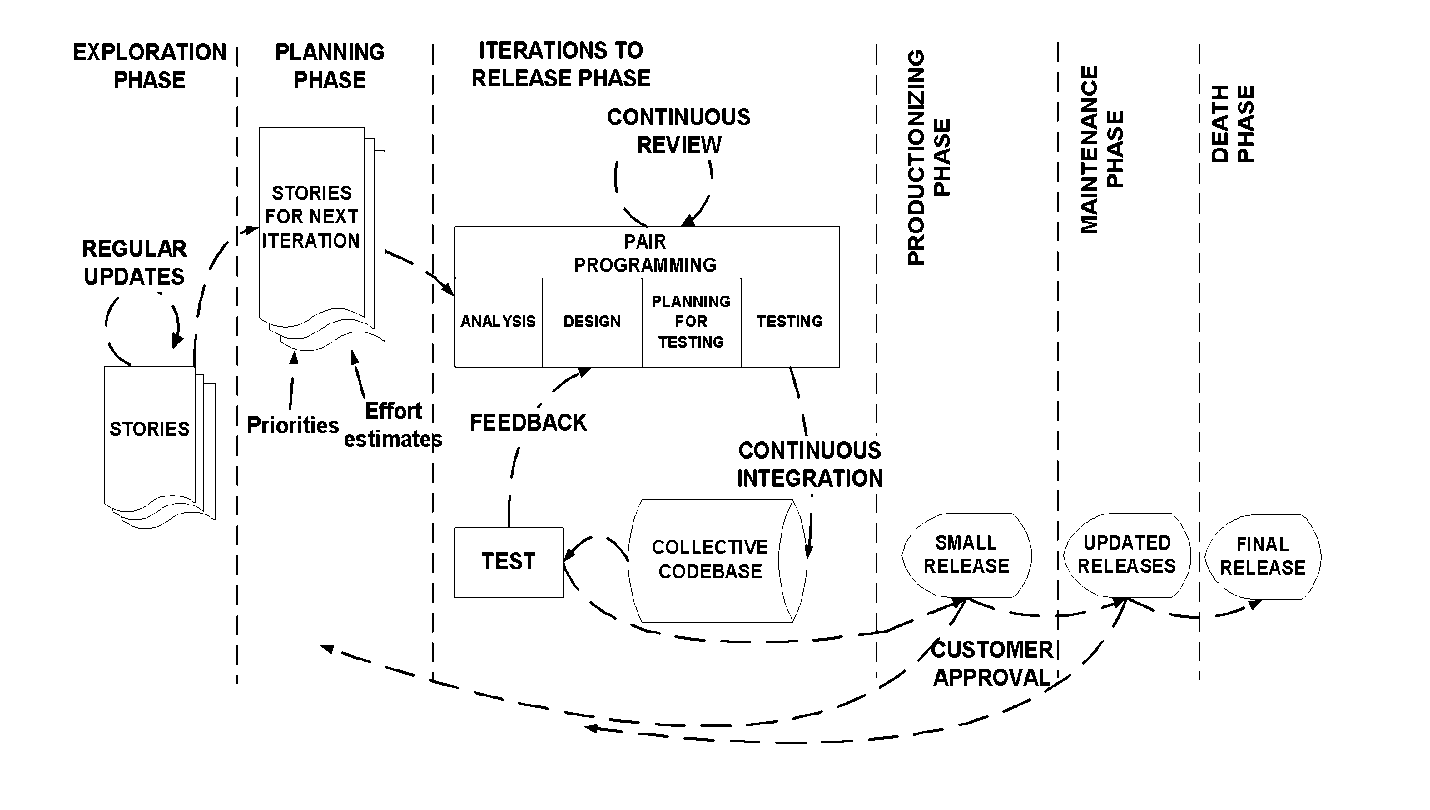
When a product is built and released in increments, it is an incremental prototype. Incremental prototyping works in stages, based on a triage system. “Triaging” means assessing each of the system’s components and assigning a priority. Based on that priority, a product’s components are built iteratively in increments from “most important” to “least important.” In this way, incremental prototypes make use of the process philosophies behind iterative processes models.Priorities for a software product’s features are based on three categories: “must do,” “should do,” and what “could do.” Core features are assigned the highest priority—must do. All the non-critical supporting features are “should do” items. Remaining extraneous features are the lowest priority—could do. Based on these priorities, the initial iterations of incremental development focus on delivering “must do” priorities. The resulting software product with the core features could be released as a complete first-generation incremental prototype. As resources permit, features under the “should do” priority can be developed for a similar second-generation incremental release, followed by a third release with features from the “could do” category. The end result of these iterations is a series of incremental prototypes from essential basic features to fully featured[1-2].

**Evolutionary Prototype**

The final type of prototype is the evolutionary prototype. This prototype approach is very similar to incremental prototypes, but the difference is found in the first-generation prototype. In evolutionary prototyping, the first-generation prototype has all the features of the product, even though some features may need to “evolve” or be refined. A comparable first-generation incremental prototype has only a basic set of core features.Both incremental and evolutionary prototyping are ways to make working software that can be shown at regular intervals to gain further feedback. In practice, both approaches can be blended. A major benefit is the morale boost for the development team as they see the working product early and are able to contribute improvements over time [1-2].

## **Extreme Programming (XP)**

Extreme Programming (XP) is an Agile methodology consisting of effective development practices to achieve client satisfaction. To apply XP, there are 12 practices to follow in extreme level.



**Practice 1: The Planning**

The client and development team work together in planning the product. This entails a larger planning session at project initiation and smaller sessions at each iteration. The larger session determines the product’s required features, their priorities, and when they should be released over time. A smaller session focuses on the features to be completed in an iteration and determines the development tasks needed[1-2].

**Practice 2: Small Releases**

Releases must occur as frequent as possible to gain plenty of feedback, which is best achieved if releases are small in terms of required new functionality. The required features should be prioritized to deliver value to the client early. The client and development team need to strike the right balance between what the client wants completed first and what can be developed early. Smaller releases also allow estimates to be more certain; it is easier to look a week or two ahead, rather than months. So, in the extreme, keep the iterations very short[1-2].

**Practice 3: System Metaphor**

A system metaphor makes it easier to explain the product to someone else; it can be an analogy to describe the product to someone who is not technical. For example, the desktop metaphor in graphical user interfaces helps to explain computer interactions in terms of something more familiar in the real world. Another example is the shopping cart metaphor in online shopping, which builds upon a concept from real-world shopping. A metaphor can also describe the product implementation to someone more technical. For example, the pipes and filters metaphor for software architecture helps to explain how a product implementation is structured around connecting and driving computing elements with information flows[1-2].

**Practice 4: Simple Design**

Focus on the simplest design that works to satisfy the client’s needs. Requirements change, so it would be wasteful in making elaborate designs for something that will change. Design what is needed to make the required features work. Do not over-engineer the design for a future that may not come. So, in the extreme, make the simplest thing that works[1-2].

**Practice 5: Continuous Testing**

In XP, tests are prepared for a required feature or functionality before its corresponding source code is written. This focuses efforts first on understanding what is required, making the user or programmatic interface to it simple, and preparing suitable tests to verify that the required behavior has been achieved. If the test was difficult to write, it is a sign to rework the interface or original requirement. The implementation of the required feature or functionality comes afterwards. This practice is referred to as Test-Driven Development or TDD. Automating the tests will allow them to be executed continuously. So, in the extreme, write tests first and run them all the time. There are two main types of tests involved: acceptance tests and unit tests. An *acceptance test* is for the client to verify whether a product feature or requirement works as specified and thus acceptable. An acceptance test typically involves something an end user does, touching a large part of the product. Such a test can be automated or can be a set of instructions and expected results that a human follow. A *unit test* is written and run by developers to verify whether low-level functionality works correctly. A unit test is typically specific, like testing an algorithm or data structure in some module of the software implementation.Consider a social media application with a required feature that an end user can post a message. An acceptance test for this feature would involve actions that end users do, like initiating a post, entering a message, and submitting the post. The test would check that the actions basically work for some trial data. Underlying this feature, and the product in general, is low-level functionality to store posts. Unit tests would more thoroughly check that this functionality works, such as dealing with international characters or very long texts [1-2].

**Practice 6: Refactoring**

Refactoring is a practice to restructure the internal design of the code without changing its behavior. The aim is to improve the design in small steps, as needed, to allow new product requirements to be added more easily, thus adapting to change. For example, refactoring could gradually reorganize the code to allow easier extension. A large, unwieldy module can be decomposed into smaller, more cohesive, more understandable pieces. Unnecessary code can be removed. As refactoring proceeds, the unit tests are run repeatedly to ensure that the behavior of the code stays the same. If refactoring is deferred or not done, then changes become more and more difficult. The undone work is like incurring an obligation, known as technical debt, to be “paid” in the future. A small amount may be fine to keep advancing the product, but a crisis could happen if a sufficiently large amount accumulates[1-2].

**Practice 7: Pair Programming**

To ensure high quality, XP employs pair programming, where two developers work side-by-side at one computer to work on a single task, like writing the code for a required feature. In regular code reviews, a developer writes some code, and after completion, another developer reviews it. Instead, in pair programming, the code is reviewed all the time by always having another pair of eyes to consider each edit. This brings together complementary skills and perspectives when solving a problem or generating alternatives. Riskier but more innovative ideas can be pursued, aided by the courage brought on by having a partner. Pairing a senior and junior developer can foster learning, where the senior one imparts strategic advice and the junior one offers a fresh perspective. In general, the pairs are not static but change dynamically. So, in the extreme, do code reviews all the time[1-2].

**Practice 8: Collective Code Ownership**

Although a specific pair of developers might initially implement a particular piece of the product, they do not “own” it. To encourage contributions, other team members can add to it or any other part of the product. Thus, the produced work is the result of the team, not individuals. So, project success and failure reside with the team, not on specific developers[1-2].

**Practice 9: Continuous Integration and Daily Build**

In XP, developers combine their code often to catch integration issues early. This should be at least once daily, but it can be much more frequent. With tests written first, the tests can also be run frequently to highlight pending work or unforeseen issues.For the Microsoft Daily Build, each “iteration” of the construction phase is laid out in a day,hence the “daily build.” The point of the Microsoft Daily Build is to ensure that theprogrammers are all in sync at the beginning of each build activity. By following a process that makes the developers integrate their code into the larger system at the end of every day, incompatibilities are detected sooner rather than later. To control this, Microsoft uses a system of continuous integration. When a developer writes a piece of code and wants to share that code remotely with anyone else on the team, the code must first be put through an automatic testing process, which ensures that it will work with the project as a whole. All the developers can easily see how their work fits into the product as a whole, and how their work affects other members of the team.Continuous integration not only keeps developer morale up, but it also increases the quality of the product. The daily build does this by giving developers the ability to catch errors quickly before they become a real problem. A successful build allows overnight testing to proceed, with test results reported in the morning. At Microsoft, as an incentive for successful daily builds, a developer whose code breaks the build must monitor the build process until the next developer to break the build comes along [1-2].

**Practice 10: 40-Hour Work Week**

XP aims to be programmer friendly. It respects the developer’s balance of work and life. At crunch time, up to one week of overtime is allowed, but multiple weeks of overtime would be a sign of poor management or estimation[1-2].

**Practice 11: On-Site Customer**

The client is situated near the development team throughout the project to clarify and answer questions[1-2].

**Practice 12: Coding Standards**

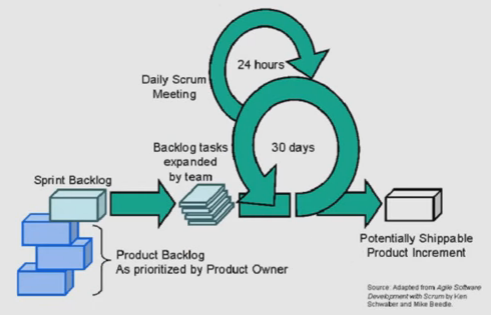
All the developers agree to and follow a coding standard that specifies conventions on code style, formatting, and usage. This makes the code easier to read, and it encourages the practice of *collective ownership*[1-2].

**Limitation of XP Practices**

One limitation is that XP is intended and suited for small development teams, for example, no more than ten people. As well, having a client available on-site with the development team may not be possible to arrange. Also, XP does not offer practices to define the architectural design of the software. Instead, this design emerges in response to changing requirements and refactoring, using the system metaphor as a guide. This approach would produce working software early, but it might involve more rework than if some effort were spent to plan the architecture[1-2].

## **SCRUM**

Scrum is an Agile methodology consisting of lightweight management practices that have relatively little overhead. Its practices are simple to understand but very difficult to master in their entirety. Scrum uses an approach that is both iterative and incremental. There are frequent assessments of the project, which enhances predictability and mitigates risk [1-2].



Scrum is based on three pillars: Transparency, Inspection, and Adaptation . With transparency, everyone can see every part of the project, from inside the team and outside the team. Scrum encourages frequent inspection of work products and progress to detect undesirable deviations from expectations. The inspections, however, do not happen so frequently to impede development. Ken Schwaber, a Scrum trainer, says “Scrum is like a mother-in-law, it points out all your flaws.” Developers sometimes are hesitant to adopt scrum because it points out everything they are NOT doing right. When a deviation is detected, the team must adapt and adjust to correct the problem. Important is agreeing on standards for the project such common terminology and definitions such as the meaning of “done”[1-2].

**SCRUM Process and Practices**

The project timeline consists of a sequence of consecutive sprints. A sprint is an iteration, whereby an increment of working software is delivered to the client at the end. Each sprint is “time-boxed” to a consistent and fixed duration. That fixed duration is chosen at the start of the project and is typically one or two weeks. To enable the three pillars, Scrum outlines four required techniques or events that happen within a sprint[1-2]:

1. **Sprint planning** (to set expectations for the sprint)
2. **Daily scrums** (to ensure work is aligned with these expectations)
3. **Sprint review** (to gain feedback on completed work)
4. **Sprint retrospective** (to identify what to do better in the future)

*Spring planning*occurs at the beginning of a sprint to determine what will be completed in that sprint. This planning sets a sprint goal, a high-level view of what will be done. The plan also identifies the particular requirements to complete and the associated developer tasks. To keep the team focused on the plan, *daily scrums* are brief meetings held early each day, when each team member outlines what they did previously, what they will work on, and what may be blocking their progress. To encourage brevity and avoid long meetings, the daily scrums are conducted with the team standing up. So, the daily scrum is also known as the daily stand-up meeting. A *sprint review* happens at the end of a sprint to show the working software to the client and gain feedback. A sprint retrospective gives an opportunity to reveal lessons learned that can be addressed to improve future work. Once sprint planning is done, work proceeds according to the sprint goal, determined requirements, and identified tasks.To avoid disturbing work in progress during the sprint, major new requirements changes are noted for later[1-2].

**SCRUM Roles**

Scrum defines some key roles for a scrum team. Depending on the situation, there are various mappings for a client and software product manager to these roles. In a large organization or a large product, the client, software product manager, product owner, and scrum master can be all different people. In a small start-up company, there may be significant overlap [1-2].

* **Product owner** is the one person who is responsible for the product. All product requirements for the developers must come through this role. The list of requirements for the product is gathered in a product backlog, which the product owner is responsible for. The product owner also determines the priorities of these requirements and ensures the requirements are clearly defined [1-2].

Naturally, the client could be the product owner, who determines the requirements and accepts the completed work. However, with clients inexperienced with defining clear requirements for developers, a software product manager may represent the client and take on the product owner role instead. If there is not an actual client, as in a mass-market product, then the software product manager may represent the end users and serve in the product owner role [1-2].

* **Scrum master** makes sure the scrum team adheres to scrum practices, by assisting the product owner and the development team. For the product owner, this help includes suggesting techniques to manage the product backlog, to attain clear requirements, and to prioritize for maximum value. For the development team members, this help includes coaching the team to self-organize and remove roadblocks. The scrum master also facilitates the four events listed above within a sprint [1-2].

A team lead could be the scrum master to facilitate scrum practices by the team. In a small company, a software product manager with leadership qualities may need to take on the scrum master role. In any case of involvement, it is important for software product managers to understand the scrum roles and their responsibilities [1-2].

* **Scrum Development Team** are the developers on a scrum team, also known as the scrum development team, must be self-organizing. No one outside the development team, not even the scrum master or product owner, tells them how to turn the backlog of requirements for a sprint into developer tasks to produce an increment of working software. Scrum development teams are small, ideally between three and nine people. They are self-contained, consisting of everyone and everything needed to complete the product. Also, each member generally takes on mixed tasks, like doing both coding and testing, rather than having dedicated coders and testers. There are no special sub-teams. Everyone on the team is responsible for the team’s work products. Accountability rests with the entire team[1-2].

## **Lean Software Development**

The origin of Lean software development is inspired by the manufacturing industry where Lean production grew out of the “Toyota Production System.” At Toyota, their approach was used effectively to reduce waste in the production process and increase the quality of their vehicles. Lean is based on seven principleswhich can be effective for both small and large projects [1].

**Principle 1: Eliminate Waste**

How can software development be wasteful? Consider the potential waste of time and effort that can arise from: unclear requirements, process bottlenecks, product defects, and unnecessary meetings. These are deficiency wastes. Waste can also be disguised as efficiency because being “busy” does not always equate with being “productive.” A busy team not focused on developing core, required features can easily produce “extra” features that interfere with the core functionality of the end product. Coding can be wasteful if it is not planned to be a released. Anything that does not add value to the product is considered waste and should be identified and eliminated[1].

**Principle2: Amplify Learning**

Explore all ideas sufficiently before proceeding with actions. Do not settle and focus on a single idea before fully exploring other options. This principle seeks to find the best solution from a number of alternatives. Lateral thinking generates alternative approaches, and the morealternatives that are considered, the greater the likelihood that a quality solution will emerge.The biological mechanism of natural selection is a good metaphor for amplified learning. The most successful solutions emerge from the widest variety of alternatives. This exploration will lead to building the right product. The principle also encourages running tests after each build of the product. Consider failures as opportunities to amplify learning on how to improve[1].

**Principle3: Decide as Late as Possible**

Unless you actually need to make a decision *right now*, don’t make the decision yet. Deciding as late as possible allows the exploration of many alternatives (amplify learning) and selection of the best solution based on the available data. Early, premature decisions sacrifice the potential for a better solution and the “right product”[1].

**Principle4: Deliver as Fast as Possible**

There is a relationship between Lean principles. Eliminating waste requires thinking about what you are doing; therefore, you must amplify learning. Amplified learning requires adequate time to consider alternatives; therefore, you must decide as late as possible. Deciding as late as possible requires focused productive work; therefore, you must deliver as fast as possible. Delivering as fast as possible is chiefly concerned with evolving a working product through a series of rapid iterations. Each release can focus on core product features, so time and effort are not wasted on non-essential features. Frequent releases provide opportunities for the client to give feedback for further refinements[1].

**Principle5: Empower the Team**

The first four Lean principles deal with the process of developing the right product. The remaining three principles describe *how* the process can be made efficient[1].

*The best executive is one who has sense enough to pick good people to do what he wants done, and self-restraint enough to keep from meddling with them while they do it*.

— Theodore Roosevelt

Lean aims to empower teams that follow its practices. It encourages managers to listen to their developers, instead of telling the developers how to do their work. An effective manager lets the developers figure out how to make the software[1].

**Principle6: Build Quality In**

Aim to build a quality product. Ways to build quality software include conducting reviews of the work products, preparing and running automated tests, refactoring the code to simplify its design, providing useful documentation, and giving meaningful code comments. Developing a quality solution reduces the amount of time developers must devote to fixing defects. By proactively applying this principle to build quality into the product, the development team eliminates waste in time and effort[1].

**Principle6: See the Whole**

Maintain focus on the end-user experience. A quality product is cohesive and well designed as a whole. Individual components must complement the entire user experience[1].

## **Kanban**

Kanban involves a technique to organize and track project progress visually, which is widely used even outside Agile or Lean software development. *Kanban* is a Japanese word, which loosely translated means board or signboard. So, the Kanban technique uses a board, with a set of columns labeled by the stages of completion [1].

**Tracking Tasks**

A Kanban board can track the status of tasks completed by a team. So, for simple tasks, suitable column labels for the stages of completion would be “To Do”, “Doing”, and “Done.” Initially, all the required tasks are written on sticky notes and placed in the “To Do” column. As work proceeds and a task to do enters the “doing state,” its sticky note is pulled from the “To Do” column to the “Doing” column. When that task is done, the sticky note then moves from the “Doing” column to the “Done” column. For the required tasks, the board easily shows their states at a glance to the entire team. The team has full visibility of what needs to be done, what is being done, and what has been done. It is motivating to see the tasks march steadily across the board and useful to see when a task appears blocked. Typically, a requirement for a feature is considered done when it is coded, tested, documented, and accepted[1].

**Tracking Product Requirements**

A Kanban board can be used effectively with Scrum. For example, the board above can track the status of the developer tasks that are planned to be done within a sprint. As well, a different Kanban board can track the status of individual requirements on the product backlog through the stages of being considered “done” in Scrum. Here, example column labels for the stages of completion could be: “Backlog,” “Analysis,” “Design,” “Tests,” “Coding,” “Review,” “Release,” and “Done,” following the phases of some software development process. Initially, the requirements on the backlog are written on sticky notes and placed in the “Backlog” column. As work proceeds, the requirements march across the board, eventually landing in the “Done” column. The requirements will move at different rates across the board toward completion[1].

In general, a requirement stays in a certain intermediate column until the team actually has the capacity to start the next stage for it. That is, a requirement is generally “pulled” into the next column. However, one tactic is to have a special “Next” column just after “Backlog”, so that a client or product owner can take a high-priority requirement off the backlog and explicitly “push” it into the development process to be worked on in a sprint. In this way, Kanban can be used to organize work as well as tracking work [1].

## **EXERCISE -4: Selection of Process Model**

**Objective:**Selection of appropriate process model for the software development

**Tools/ Device:**Protégé Ontology Development tool to check rational and consistency of your process model selection.

**Procedure:**

1. Study the Software Engineering process model to appropriate process model to develop your proposed system solution.
2. Present your arguments based on your analysis about why your selected method(s) is the best choice among all other methods to develop your proposed software.
3. Identify all the roles in the project management activities in software development. Describes the responsibilities of the role in the software development.
4. Your Design Specification must address the evaluation rubrics mentioned in the software engineering course outline.