UNIVERSITY OF ALBERTA
DEPARTMENT OF COMPUTING SCIENCE

SOFTWARE PROCESSES AND AGILE PRACTICES

COURSE NOTES

Table of Contents

Course Overview

Upon completion of this course, you will be able to:

- Explain different process models for structuring software development.
- Determine which process model works best for different software development projects.
- Apply the fundamentals of Agile software development practices, such as Extreme Programming (XP) and Scrum.

This course examines software development **processes** and Agile **practices**. The material starts with key process concepts and then transitions into established processes and practices that are successfully used in industry. For instance, you will learn the concept of **iterative processes,** followed by established **process models** such as the **Unified Process** model. The last half of this course will focus on the application of practices—specifically, **Agile** practices—to support software product development processes.

Module 1: Introduction to Processes

Upon completion of this module, you will be able to:

- Summarize the concept of a *process*.
- Define the process terms: *life cycle*, *phase*, *activity*, *task*, *work product*, *resource*, and *role*.
- Understand the hierarchical relationships of: *processes*, *phases*, *activities*, and *tasks*.
- Differentiate between how tasks *consume*, *use*, and *produce* resources and work products.
- Determine how practices align with a process.
- Define and identify a practice and a methodology.
- Summarize key software engineering activities.
- Recognize the inputs and outputs of each activity.
- Summarize the phases of an example software engineering process.
- Identify the software engineering activities in each phase.
- Define project management tasks and activities that are ongoing throughout all phases: *risk management*, *performing estimation*, and *allocating resources*.
- Understand the differences between *expressing, gathering*, and *eliciting requirements*.

Processes and Practices

The Importance of a Process

Establishing a process is a key starting point for every software development project. A **process** organizes the work on a software product into distinct **phases**. In general, the work that is undertaken for a project will transition from one phase to the next. Each phase has specific activities and tasks that must be performed. By creating distinct phases with clear steps and deliverables, the use of processes structures the software development in a manner that provides clarity for everyone involved.

Creating software is more like authoring a cookbook than following a specific recipe. There is no prescribed formula to writing a cookbook; recipes must be collected, organized, and tested. Recipe testing may lead to further refinements, experiments, and improvisations. Rejection of some recipes may require backtracking to the collection phase. And, if successful, a cookbook may have a long life cycle involving multiple editions in which corrections are made, and new recipes are added.

Similarly, software development requires continuous evaluation and decision making while moving through each phase of a given project. This continues well past the initial development project. A software product, just as with a cookbook, can have a long life cycle involving multiple versions with improved features and resolved defects.

Life Cycles and Sub-Processes

A **software life cycle process** refers to a process that covers the entire spectrum of software development, from the product's initial conception to its eventual retirement. A life cycle process is an all-encompassing approach to software development, which may include sub-processes for major releases and product evolution. Of particular interest for this course is building a new product, which requires a process that organizes how to get the product from *an idea in the client's mind* to *working*

software that an end-user experiences. The software life cycle processes presented in this course will focus on this software development problem.

Processes and Phases

As noted before, a process is divided into multiple **phases**. For example, the phases of a process to create a software product may be:

- Specification
- Design and Implementation
- Verification and Validation

Phases—or milestones—within a process need only be identified in a logical manner and organized so that the phases contribute to the completion of the process.

Phases, Activities, and Tasks

For example, consider the *Verification and Validation* phase identified previously; one of the activities within might be *Executing Tests.* Tasks related to the activity of executing tests could include:

- Writing test framework code
- Designing tests
- Writing tests
- Running tests

Tasks are where the real work gets done, no matter what process is used. Tasks are the small, manageable units of work for the project. They are the building blocks for completing an activity, finishing a phase, and ultimately completing a process.

Last Word on Phases

Adopting and adapting a suitable process will be key in structuring the work for a successful project. Designing a process can start from scratch or may be based on one of the many **process models** that have already been used successfully. A variety of process models will be presented during this course.

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:

- **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

Tasks and Task Dependencies

As introduced above, tasks are the most basic unit of work that will be managed. Tasks in software development include small pieces of work, such as:

- Write source code for a feature
- Design a feature
- Write documentation
- Install a library
- Test a feature

Tasks often have **dependencies**—for example, a task that must wait for another task to complete is said to be dependent on that other task. Dependencies will impose a specific order for tasks to be completed. It is important to understand and sequence dependent tasks appropriately.

For example, imagine a process that has been divided into three phases:

1. Specification
2. Design and Implementation
3. Verification and Validation

An activity within the second phase might be "designing user interfaces." Tasks within this activity would be ordered to reflect any identified dependencies. One task is to design the user interface look. Another task is to design the user interface behaviour, which depends on having the look established.

Although this is a small example, it does illustrate the need to identify all task dependencies, to avoid doing tasks out of order and wasting resources.

Tasks and Roles

Ultimately, it is specific people who perform the tasks. However when assigning tasks, it is good practice to assign them to roles rather than a specific person (e.g., Sam) or even a specific position (e.g., Software Engineer). There are very good reasons to approach assignment of tasks in this manner. First, even over a specific phase, people can change in an organization, so assigning tasks to roles is more constant over the life of the project. Second, setting tasks to roles will help avoid any perceived personal choices that can occur when assigning tasks to specific people instead of a given role.

A **role** is job-related activity assigned to a person. As mentioned above, a team member named Sam may have a job title "Software Engineer"; however, one of his roles is to *program*. Any given development team will have multiple members who can perform many roles, such as:

- Graphic designer
- Programmer
- Tester
- Project manager
- Product owner

Each role on a team describes specific skills that are needed to accomplish any given responsibility or task. For example, a programmer may write code for a certain feature and this might then be assigned to our example team member Sam, who is a Software Engineer. A tester will be assigned to test that feature, and this role may be filled by a Junior Developer named Theresa.

*A role **performs** a task.*

Tasks and Work Products

A **work product** is an output produced by completing a specific **task**. Work products, of course, do include the final product, but they also include other intermediate outputs such as design documentation, requirements documents, source code, test cases, and project management artifacts such as status reports and sign-off forms.

*Since work products are outputs of tasks, we can say that a task **produces** a work product.*

Work Products as Outputs and Inputs

As mentioned, some tasks depend on other tasks. So, the work product *output* of an earlier task will also be the *input* for a subsequent task. For example, the output of a task to design a feature becomes the input for a task to write the source code for that feature.

*Not only does a task produce work products as output, it also **uses** work products from another task as input.*

Tasks and Resources

Completing any given task requires resources to advance or fund it, such as time, people, technology, knowledge, and money.

*A task **consumes** resources.*

Practices

Practices are strategies used to execute processes smoothly. Practices contribute to the effectiveness of a development process. For instance, a manager can follow proven management practices for: scheduling and organizing tasks, minimizing resource waste, and tracking completed work. Additionally, estimating tasks, holding daily meetings, improving communication, and doing periodic reviews of the

project are all practices that can help a process run well. A developer can follow established development practices for: preparing effective tests, maintaining simplicity, and conducting technical reviews. This ensures a quality product with low defects.

Practices are often gathered into a **methodology**. Put another way, a methodology is a set of practices. Methodologies based on the philosophy outlined in the Agile Manifesto are known as **Agile methodologies**. **Scrum** is an example of an Agile methodology. It is an iterative and incremental software development methodology for managing product development. All the practices associated with Scrum adhere to Agile principles and philosophy.

The Agile Manifesto provides the philosophy, and an Agile methodology provides a set of day-to-day practices that align with that philosophy.

Practices and Process

Certain practices lend themselves to particular phases in a process. For example, within the *Specification* phase of a software life cycle process, practices designed to help elicit needs and express requirements are useful.

Other practices are useful throughout the entire process, such as those that pertain to project management, such as having daily meetings, monitoring how productive the team accomplishes work, and keeping everyone aware of outstanding items that still need to be completed. Specific practices will be examined later in this course as Agile methodologies are presented.

Adopting good practices reduces wasted resources. Effective planning and communication prevents duplicating work or producing a feature that is not needed. Good practices then help to keep the project on schedule and on budget.

Processes and practices are flexible and customizable. Combine various aspects of processes and practices to create something customized to your development needs. Finally, be open to making adjustments that refine the processes and practices to best fit your project and product.

Software Engineering Activities

A process is made up of phases. A phase is made up of activities. An activity is made up of tasks.

This lesson will look at activities directly related to software engineering. Understanding software engineering activities will provide a more complete picture of how software products are made.

We have already identified that tasks produce work products that can become the inputs to other tasks. Since related tasks comprise an activity, an activity too has work product inputs and outputs (i.e., the

Instructor's Note:
As the software engineering activities are discussed, they will be grouped into generalized phases to give concrete examples of how the activities can belong to the phases of a process.

initial input(s) into the related tasks and the final output(s) upon the completion of those tasks). Additionally, just as tasks have dependencies, so too do activities. For example, the activity of "creating tests" will result in a work product output that will enable the next activity of "executing tests" to begin.

Specification Activities

A **specification phase** is often a starting point for developing a software product. These activities focus on the requirements for the product. In a linear process model, specification activities might happen once at the beginning of the process; however, in iterative or parallel models, specification activities can be revisited multiple times to capture updated client needs and suggested product improvements. A software product manager plays a significant role in specification activities.

Specification activities usually include:

- Identifying ideas or needs
- Eliciting requirements
- Expressing requirements
- Prioritizing requirements
- Analyzing requirements
- Managing requirements
- Formulating potential approaches

The importance of *identifying ideas or needs* should not be underestimated. While it may seem self-evident, thoroughly understanding or exploring the idea or need that drives development will help to create a great software product.

Once the core idea of the product is understood, the next steps are to define the requirements.

1. Elicit requirements (probe for them)
2. Express requirements (write them down in a structured way)
3. Prioritize requirements (rank them by importance)
4. Analyze requirements (check them for clarity, consistency, completeness))
5. Manage requirements (organize, reuse, reference them)

Software engineering activities place an emphasis on having a good initial set of requirements to guide development work. The **Client Needs and Software Requirements** course within this specialization is devoted to the activities and practices to form effective requirements.

The final activity, *formulating potential approaches*, allows the product team to outline a strategy, consider alternatives, and further refine their next steps.

Design and Implementation Activities

These are activities accomplished by the development team to design and implement the software product. Managers need to track and monitor the work products that result from these activities.

Design and implementation activities can include:

- Designing the architecture

- Designing the database
- Designing interfaces
- Creating executable code
- Integrating functionality
- Documenting

These are all activities that happen before any programming can begin. Having a design for the software architecture, related databases, and interfaces will improve the ability of developers to work together towards a common goal. Once programming begins, programmers should be integrating or combining their code at regular intervals to ensure compatible functionality. Internal documentation will help new developers when they join the team or enable existing team members to work on development of unfamiliar aspects of the product.

Verification and Validation Activities

Constant verification and validation of the product will ensure that its development is on the right track.

Verification and validation activities include:

- Developing test procedures
- Creating tests
- Executing tests
- Reporting evaluation results
- Conducting reviews and audits
- Demonstrating to clients
- Conducting retrospectives

Verification typically consists of internal testing and technical review activities that ensure that the product does what it is intended to do, according to the documented design and requirements.

Validation activities such as *demonstrating to clients* help to check that the product meets the needs of the client and/or the end users. With iterative or parallel process models, feedback at this stage can be incorporated into new specifications to improve the product through the next cycle of development.

Both the product and the process can also be improved. *Conducting retrospectives* is an activity to identify areas to do better next time. Retrospective exercises with the development team will generate ideas for improving the process and the product.

Project Management Activities

Although the example phases represent a natural flow through software development, there are also ongoing project management activities that generally occur in parallel to the activities discussed above.

Project management activities usually include:

- Creating a process

- Setting standards
- Managing risks
- Performing estimations
- Allocating resources
- Making measurements
- Improving the process

Creating a process is an important part of managing a project because it sets the initial phases for development. A process provides a structure of expected activities and work, which help to guide a team through a product development project.

Along with a process, *setting standards* is needed to ensure the team's activities are performed consistently. The team needs a clear set of expectations on aspects of development like coding conventions, documentation levels, testing strategies, and when work should be considered *done*. A development team may set a standard that work for a feature is considered *done*, for example, when the implemented source code follows a specified convention, when the documentation has been provided to the developers and end-users, or when features has been tested.

Another project management activity is *managing risks*. This requires constant analysis and mitigation of potential risks (e.g., business, technical, managerial, scheduling, and security risks). Inputs for managing risks may require historical project records, project estimates, the software's detailed design, and the life cycle process. Any part of the project that may be considered vulnerable to failure would be an input work product for managing risks. The output work product is a **risk plan** that outlines potential risks and mitigation strategies. Ongoing risk review will identify and mitigate new risks that may emerge throughout the development process.

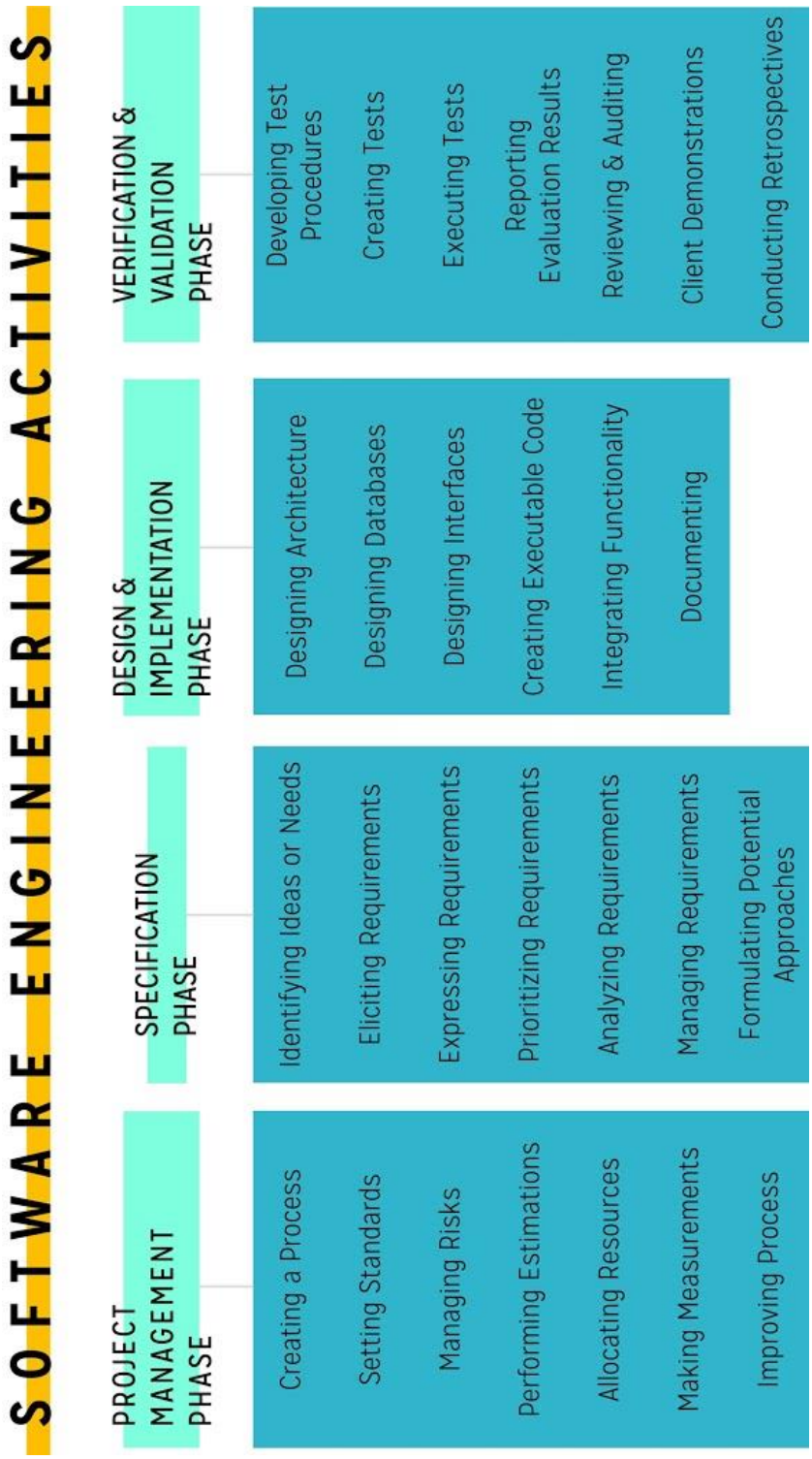
Furthermore, effective project management relies on *performing estimations*. Estimation is a critical activity in software development since development timelines are based on estimates of how long it takes to complete tasks. When estimates are too inaccurate, a schedule can be severely compromised. Tasks involved in the estimation activity include collecting data for estimates, performing estimate calculations, and evaluating estimates. For example, performing an estimate for the duration of a given task is itself a task with an input work product of collected, historical, task-duration measurements. The output work product is the estimated duration itself, which can be used in subsequent scheduling tasks.

The activity for *allocating resources* ensures that people, time, money, and equipment are deployed to do a required task. Assigning tasks and creating a work schedule are both tasks with the resource allocation activity.

Making measurements is another important project management activity. This involves tasks like defining usable metrics, calculating data, and analyzing metric data. **Metrics** are used to track and assess the quality of the product and/or the process.

Improving the process is a critical part of effective project management. It is important to recognize that, like the product, improvements can also be made to the development process. It is as important to test and review the process, as it is to test and review the product. Measurements are useful inputs for calculations to determine how the process can be improved.

Summary of Software Engineering Activities



Outlining these software engineering activities provides an idea of how they contribute toward developing a software product or managing a software project.

Module 2: Process Models

Upon completion of this module, you will be able to:

- Explain the need to consider various process models.
- Fully describe a linear model including its strengths and weaknesses, and its phases in the correct order. This includes:
    - Understanding the similarities and differences of the linear models: Waterfall, V-Model, and Sawtooth.
- Fully describe an iterative model including its strengths and weaknesses. This includes:
    - Understanding how phases relate to iteration in the iterative Spiral model.
- Fully describe a parallel model including its strengths and weaknesses. This includes:
    - Describing the specific phases associated with Unified Process models.
    - Understanding how Unified Process models can accommodate parallel activities, but that phases cannot happen in parallel.
- List and describe the five types of prototypes: illustrative, exploratory, throwaway, incremental, and evolutionary.
- Differentiate between prototypes, including:
    - Evolutionary and incremental,
    - Illustrative and exploratory, and
    - Illustrative and throwaway.

Module Overview

Process models for software development have evolved alongside advancements in computing power. While it may be tempting to simply adopt the latest innovations in software development process models, not all process models fit all project needs. It is worthwhile to gain a perspective of the relative strengths and weaknesses of each model, in order to ensure that you are musing the best tool for the job. A project may find that some process models are more complex or comprehensive than needed. For instance, developing a simple web blog for a local non-profit may be easily done with a simple process model; in fact, following a more advanced process model may result in more overhead than the organization can afford to undertake.

Linear Models

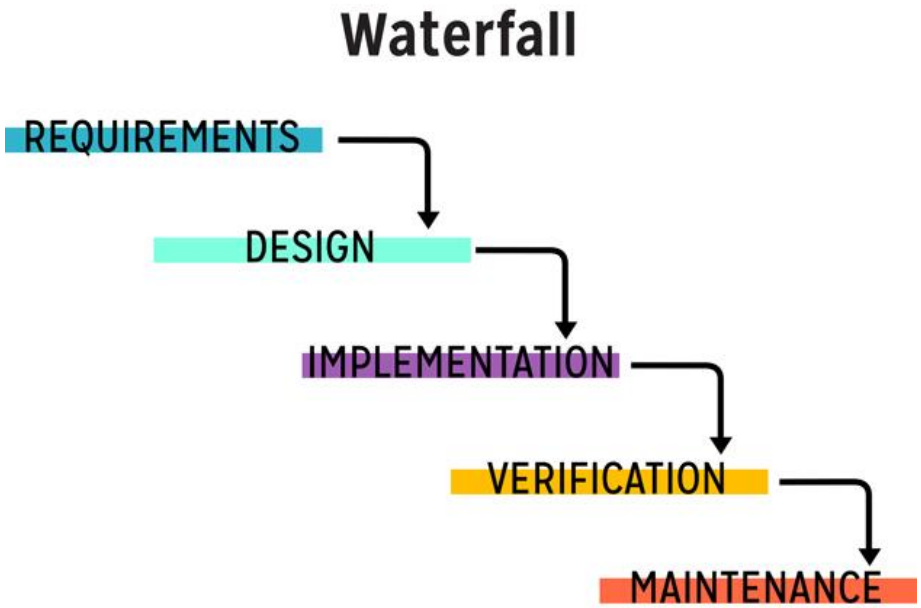
In this lesson, we will introduce three early linear process models: The Waterfall model, the V model, and the Sawtooth model. We will discuss the strengths and weaknesses of each mode, and then move into more elaborate models.

Linear process models 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.

In the early days of computing, the process of software development did not favour quick experimentation. Computers were not as prolific as they are now, and developing for them was much costlier. Computer code had to be written (or punched into cards) with meticulous attention to detail, and the time it took to compile the code was often significant and expensive. As a result, there was more emphasis on detailed, upfront design. Linear process models fit into this early way of thinking.

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. There are modified variants of the Waterfall model that allow flows to prior phases, but the dominant flow is still forward and linear.



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.

Benefits and Drawbacks

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.

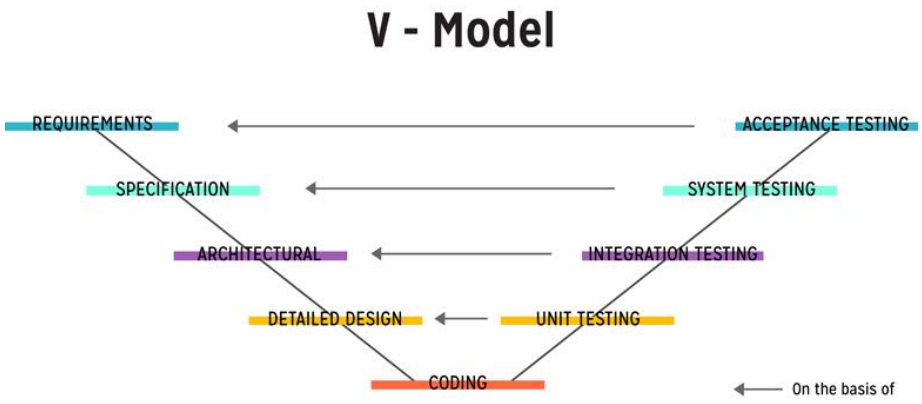
The Waterfall model faces one major flaw: It is not very adaptable to change. That is, it is not amenable to Agile principles. Modern software development is often very dynamic; flexibility with respect to

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.

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; that is, it adds a more complete focus on testing the product. The distinguishing feature—and namesake—of 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).



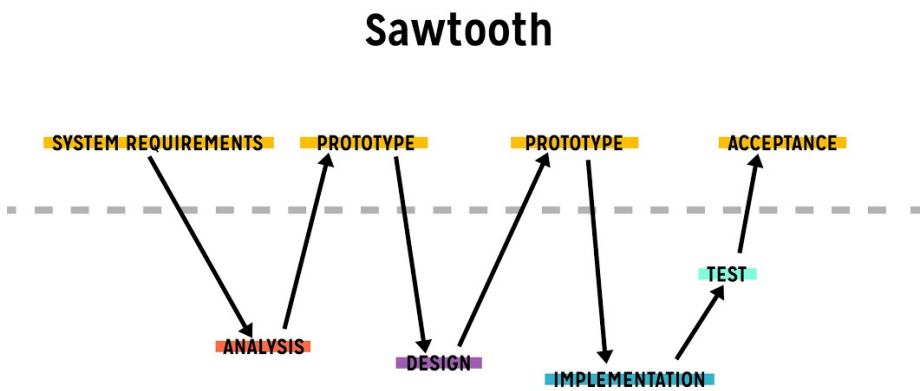
Benefits and Drawbacks

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.

Sawtooth Model

The **Sawtooth** model 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.



Benefits and Drawbacks

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.

Last Word on Linear Process Models

The main thing linear process models have in common is their sequential organization of phases. Development happens in a simple and straightforward way. Not being able to revisit early stages such as analysis or design results in the inflexibility of these models. Decisions made early in the process will constrain the final product.

Early linear process models subscribe to a *manufacturing* view of a software product. In essences, this means that it is machined and assembled according to certain requirements. Once produced, the product would only require minor maintenance upkeep, like an appliance. The emphasis falls on getting all the requirements specified "correctly" at the beginning of the process and not changing them afterwards. In reality, developing a software product is more of a creative endeavour, which necessitates experimentation and constant rework.

Linear process models are a legacy from an era when computing power was relatively expensive compared to human labour. So, programming was efficient for computers, but not necessarily for people. The time elapsed between writing a line of code and seeing its result could be hours. Trying small programming experiments to quickly test out ideas were not practical. Linear process models fit into a mode of thinking—get things right the first time and avoid rework.

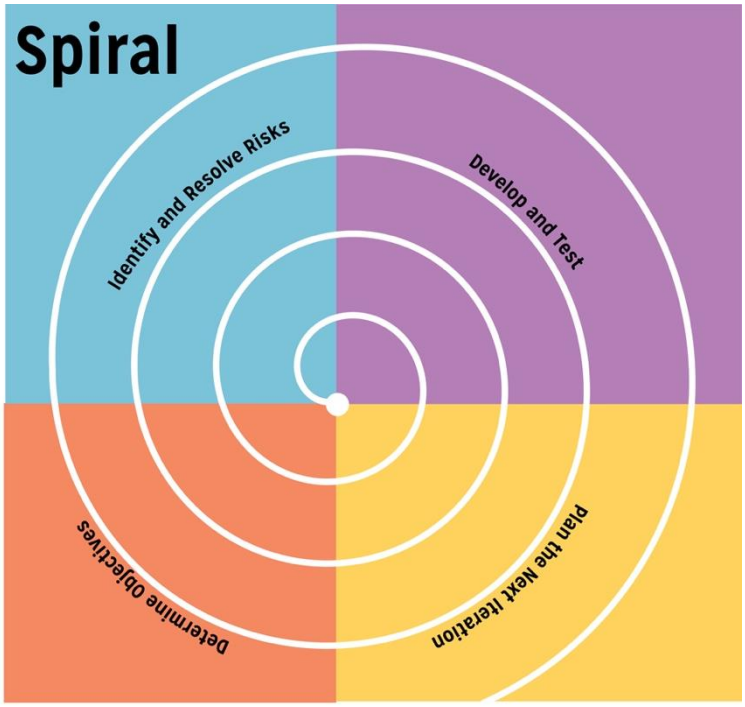
Like we said before, these linear models can still be useful for certain situations. For example, linear processes are great for developing simple, well-defined product. Indeed, more advanced process models still have certain phases that happen linearly.

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.

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.

Drawbacks of the Spiral Model

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.

Last Word on Spiral Model

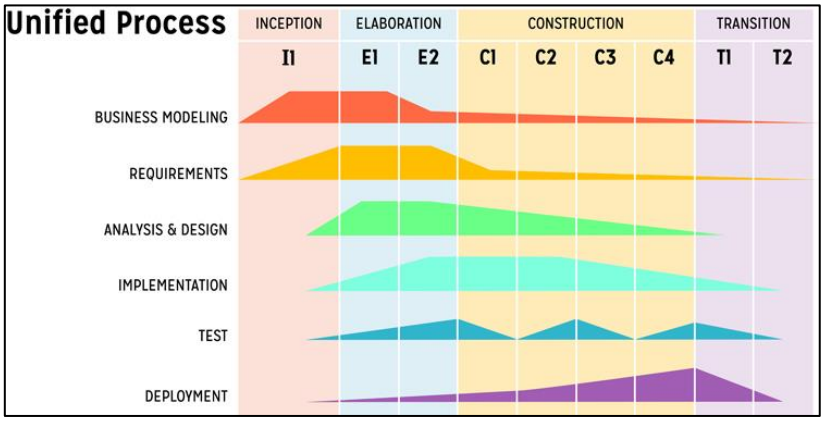
This simplified description of the Spiral model introduced the idea of an iterative process. Work is done in iterations that allow for revisions of the product at certain specific intervals. The ability to harness a repetitive process to make further refinements to a product is a clear advantage over the earlier linear models.

Parallel Models

The Spiral model is a true iterative process for software development, which means that the product is built in a repeated series of phases and product refinements are introduced at specific phases within the cycle. **Parallel processes** use a similar style of iteration—products are built in iterations—however, parallel processes allow for more concurrency of activities and tasks.

Unified Process Model

The **Unified Process** model is an iterative model of software development. 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.



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.

Phases within the Unified Process Model

The Unified Process model has four phases, with each phase having a distinct aim.

1. Inception phase
2. Elaboration phase
3. Construction phase
4. Transition phase

Inception phase

This first phase is meant to be short—enough time 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 defined are the 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.

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.

Construction phase

Similar to the preceding phase, the *construction phase* has 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.

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.

Last Word on Unified Process Model

Unified Process model is an example of a parallel process that features iterations. Activities like requirements, design, and implementation can happen at the same time, and they all drive the product to a "release" stage. However, once the product is released, the process can begin another cycle to refine and improve the product.

Iterations also happen within phases; for example, the elaboration phase may lead to several iterations of design before the work product is ready for the construction phase.
The Unified Process is well suited to large projects where iterations can allow the product to grow naturally beyond the original inception for the product.

You may notice a pattern of evolving ever-more advanced processes, but this does not mean that simpler processes are outdated! While earlier linear and iterative processes may be less sophisticated than modern parallel processes, the older processes are still frequently used in simpler situations.

Prototypes

Developing software products through a series of intermediate prototypes was a theme in both the Spiral and Unified Process models. Let's explore prototyping in more detail.

There are five types of prototypes:

1. Illustrative
2. Exploratory
3. Throwaway
4. Incremental
5. Evolutionary

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 off 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.

DID YOU KNOW?

Some prototypes go as far as "faking" their functionality by having a human control some of the functionality of the system behind the scenes.

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.

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.

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.

DID YOU KNOW?

A common feature of *illustrated, exploratory,* and *throwaway prototypes* is that none of the effort to build the prototypes will end up in the final version of the product. Lessons are learned from the effort, but the development resources put into building the prototype are essentially lost.

The final two types of prototypes will demonstrate how development effort in building a prototype can contribute to the final product. The key is to have working software for each successive prototype, any of which could be released as a version of your software product.

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.

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.

Last Word on Prototypes

Given the different types of processes, think about how prototyping might fit into the Spiral model and Unified Process model.

In the Spiral model, a first iteration may be the creation of an illustrative prototype. This could be accomplished with a few drawings on paper to sketch an idea of how the system will work. The same illustrative prototype could work for the inception phase of the Unified Process. Prototypes help to better visualize what the product does and therefore make feature decisions based on what the product might look like.

Subsequent iterations could build on the illustrative prototype with either an incremental or an evolutionary prototype approach. Using the first iteration—an idea sketched-out on a few pieces of paper—further testing of the idea, outlining some key features, and the next prototype will take shape. Before long, a prototype is ready that can be taken to a client or potential investors to prove that the product is conceptually viable.

The core idea behind all prototyping is to gain feedback on versions of the product, in order to make informed decisions about the product's future.

Continuous Delivery

In incremental or evolutionary prototyping, the product gets added to or refined iteratively over time. The idea is to start with a basic product. Over time, successive prototypes are constructed. Typically, these prototypes are delivered to the client for feedback. However, this notion of delivery may be ad hoc and labour intensive. For example, it might take a great deal of manual work for developers to build and integrate the code into a runnable prototype. This prototype may only run on the developer's environment, on their specific device, rather than as a product suitable to deliver to the client.

Continuous Delivery applies automation so that intermediate releases of working software can be more disciplined and frequent. This allows developers to more easily deliver versions of a product continuously as it is constructed. The aim is to have working software that is tested, ready to run, and releasable to others. For example, whenever a developer commits a code change, it will be automatically built, integrated, tested, released, and deployed. Ideally, the time from making a product change to having it tried by an end user can be short and frequent. Problems arising in any of these automated steps can be noticed earlier, rather than later.

Versions that are not ready are not forced to be released. Typically, releases of prototypes are placed in specific channels. For example, there can be a "developer" channel for the day-to-day releases that

developers generate, a “test” channel for a wider group internal to a company, or a “stable” channel for releases that are more “proven.” Different expectations of feedback and tolerances to issues could occur on each channel.

The Continuous Delivery practice fits well with iterative process models like the Unified Process. The most relevant phase in Unified is the construction phase, where multiple short iterations build the functionality of the product through a series of prototypes that are continuously delivered.

Within any of these iterations, the development team would work on a number of tasks to construct the next prototype for release, including detailed design, writing code, and making test cases. Determining which features are the focus for each iteration could be guided by a prototyping approach that is incremental, evolutionary, or both.

In the Unified Process, the architecture or high-level design of the product would be mostly modeled in the elaboration phase. Detailed design defines models that describe specific lower-level units, relations, and behaviours in the product. Ultimately, developers take these models, make decisions on algorithms and data structures to write or generate the code, and prepare corresponding test cases.

To support Continuous Delivery, automated tools are used to build and integrate the code, run low-level tests, package the product into a releasable form, and install or deploy it to a test environment. For example, building the code might require other dependent code to be rebuilt. To be fast, especially for large software products, the build automation has to be smart enough to only rebuild what’s really necessary.

As code is written, features start to take form within the product. Continuous Delivery ensures there is a releasable prototype ready for end users and clients to try by the end of the iteration.

Should the project be abandoned, there would in theory still be a releasable product, even if incomplete.

Product quality would also improve. Through the iterations, the product is checked in small, frequent doses, rather than all at once.

Microsoft Daily Build

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 the programmers 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.

Last Word on Continuous Delivery

Continuous delivery can be incorporated into an iterative process like the Unified Process to release incremental or evolutionary prototypes.

Combinations of approaches, like the Unified Process with prototyping and Continuous delivery, can create powerful tools for projects where regular feedback is valuable, product quality is important, and issues need to be caught early. This combination approach is a significantly more robust process than linear processes, for example, but it may not fit every situation. There will be circumstances, especially on small, short-term projects, where setting up the required infrastructure for such a process would take more time than it is worth.

Smaller projects may benefit from more simple linear processes, such as the Waterfall, V, or Sawtooth models. However, linear processes are too simplistic for very large projects.

Process as a Tool: Choose the Right Tool for the Job!

Keep an open mind when selecting a process to fit your project. It is a mistake to think that iterative or parallel software process models are always the best tools for the job. Depending on the situation, a different process could be more effective. Also, aspects of different processes can be combined to better suit the needs of the project. Do what works best for your projects!

Module 3: Agile Practices

Upon completion of this module, you will be able to:

- understand how linear, iterative, and parallel processes work with Agile principles.
- describe the 12 principles of Extreme Programming (XP), and how they fit within Agile methods.
- recognize what types of projects will work best with Extreme Programming (XP), and which ones will not.
- understand the role of a system metaphor.
- describe the difference between an acceptance test and a unit test.
- understand the goals of paired programming.
- describe the Scrum practices: sprints, product backlog, sprint review, Scrum product owner, Scrum master, and Scrum development team.
- describe how Scrum focuses on management.
- understand the process for establishing and maintaining the product backlog, and who is responsible for defining, prioritizing, and changing requirements.

Before continuing please be clear about with the terms that have already been covered.

Software **processes** structure software development work into phases. For example, the Unified Process model is an iterative process that organizes the timeline for a product into cycles of phases, where each phase typically has iterations. A process describes the expected outcomes from the activities done in the phases, so it establishes "what" needs to be done. For example, use cases are initially outlined within the inception phase of the Unified Process. Software **practices** are techniques for "how" to develop software products or manage software projects effectively. For example, early in a process, there are practices for how to prioritize the product requirements. Software **methodologies** are defined groups of practices.

Using Agile with Process Models

Given the process models to organize work in software development, how can Agile principles apply?

One Agile principle refers to early and continuous delivery, so that there is enough opportunity for close collaboration and feedback to improve the product. The Waterfall and V models only truly delivers to the client at the end of the development process, which is not early integration. As well, comprehensive documentation is not a major focus in Agile development, but the Waterfall and V models relies heavily on documentation and approving it when moving between phases. The culture of contracts and sign-offs is also not Agile. While the Sawtooth model does provide a couple of prototypes to the client during the process, it is not enough. So, the described linear process models do not work well with Agile principles and practices.

Iterative process models work much better with Agile practices because they allow for reflection and improvement. Iterative models also allow frequent and continuous releases to gather feedback. These releases are then refined and improved in the next iteration. Short iterations and small releases are

aspects of Agile. So, the Spiral model with shorter iterations and Unified Process model with its iterations within phases would work well with Agile principles and practices.

Agile Practices

Practices are techniques, guidelines, or rules that help to develop the software product or to manage the software project.

Practices based upon the Manifesto for Agile Software Development are called **Agile practices**. Agile practices align with the Agile Manifesto. For example, the Agile development practice of having small releases fits with the Agile principle of delivering working software frequently. The Agile management practice of arranging a review meeting at the end of each iteration with the client fits with the Agile principle of satisfying the customer through early and continuous delivery of valuable software.

Practices are organized into methodologies. Methodologies consisting of Agile practices are called **Agile methodologies**. The remaining of this course will examine three popular methodologies, Extreme Programming, Scrum, and Lean, as well as a management practice called Kanban that works well with these methodologies.

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.

Three Pillars of Scrum

Scrum is based on three pillars:

- Transparency
- Inspection
- 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 and author, 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.”

Sprints and Scrum Events

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:

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

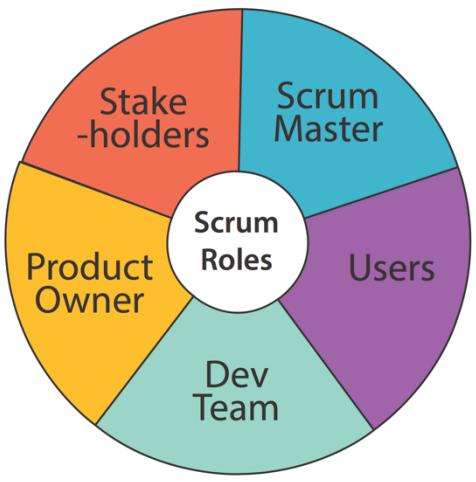
Sprint planning occurs at the beginning of a sprint to determine what will be completed in that sprint. This planning sets a sprint goal, a high-level view of what will be done. The plan also identifies the particular requirements to complete and the associated developer tasks.

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.

Scrum Roles



Scrum defines some key roles for a scrum team. Besides software developers, there is a **product owner** and a **scrum master**.

The *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.

The *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.

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.

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.

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.

Scrum Development Team

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.

Definition of "Done"

An important definition in scrum is the meaning of being **"done."** The whole scrum team must agree to that definition. Typically, a requirement for a feature is considered done when it is coded, tested, documented, and accepted. A sprint aims to get the expected requirements to be truly done, or "done done," rather than having the required features in half-done states or delivering software that does not work. To get a requirement done requires communication, integration, and synchronization by the team members. Half-done work might arise when a member works independently on something just to be busy, rather than helping to bring closure to each requirement. This may mean sometimes going slow to ensure that each requirement is truly done.

Extreme Programming (XP)

Practices

Extreme Programming (XP) is an Agile methodology consisting of effective development practices to achieve client satisfaction. The practices focus on constantly delivering software, responding to change, effective teamwork, and self-organization. These practices follow the principles outlined in the Agile Manifesto. Besides encouraging simplicity, communication, and feedback, XP fosters respect and courage.

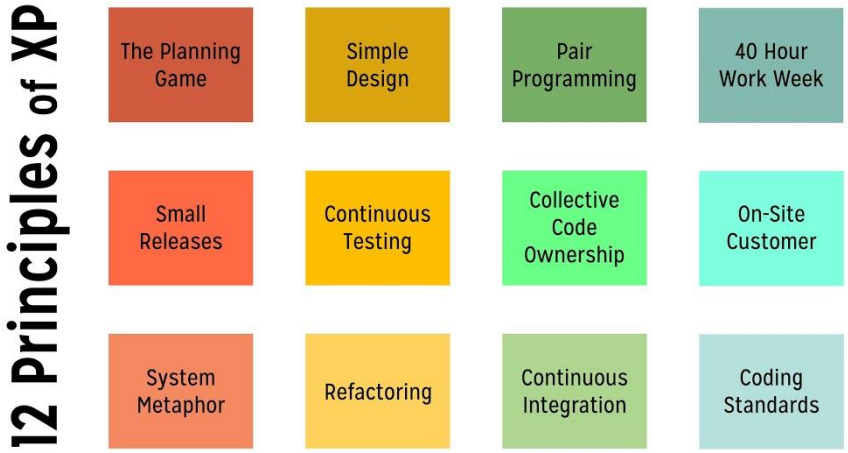
Everyone is valued and respected for what they individually bring to the project, be that business experience, domain expertise, or development skills. Everyone is considered equal, so that the client, software product manager, and development team members are all on the same level, working together.

As for courage, the following quote is very fitting.

We will tell the truth about progress and estimates. We don't document excuses for failure because we plan to succeed. We don't fear anything because no one ever works alone. We will adapt to changes whenever they happen.

— Extreme Programming, 1999

To apply XP, there are 12 practices to follow. The practices work together, so following all of the practices gives the greatest benefit.



Practice 1: The Planning Game

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.

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.

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.

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.

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 behaviour 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 follows. 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.

Practice 6: Refactoring

Refactoring is a practice to restructure the internal design of the code without changing its behaviour. 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 behaviour 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.

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.

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 resides with the team, not on specific developers.

Practice 9: Continuous Integration

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.

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.

Practice 11: On-Site Customer

The client is situated near the development team throughout the project to clarify and answer questions.

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*.

Additional Management Ideas

Beyond the 12 basic practices, XP also suggests giving the team a dedicated, open work space, with people working together in person. Multiple computers are situated in the middle of the room, available for use by everyone. A meeting table and lots of wall space for whiteboards or sticky notes allows the team to collaborate and brainstorm.

The open work space allows people to move around. It encourages communication, propagates knowledge, and enables flexible work relationships, so everyone eventually learns how to work on all parts of the product, without a heavy focus on technical documentation.

A final management idea is to change XP itself. That is, fix XP *when* it breaks. While adopting all the basic practices is touted to maximize the benefits, not all the practices will work for every project or team. When a practice does not work, change it. Retrospective meetings are a way for the team to discuss what is working and what is not. Gather data to support what practices to adopt.

Drawbacks

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 was spent to plan the architecture.

Pair programming is a major change to many developers. Some teams thrive in such a collaborative environment, while others do not. The practice requires respectful personalities that work well together. In reality, not everyone is compatible with everyone else.

Module 4: Other Practices

Upon completion of this module, you will be able to:

- Understand that Agile practices are apt to change because technology changes and evolves.
- Explain Agile Practices other than Scrum and Extreme Programming, such as: Dynamic Systems Development Method, Feature-Driven Development, Behaviour-Driven Development, and Scrumban.
- Describe the differences between Unified Process and Agile Unified Process.
- Describe the practices of Lean, including: eliminating waste, amplifying learning, deciding as late as possible, delivering as fast as possible, empowering the team, building quality in.
- Understand that "amplifying learning" is about seeking alternate solutions that vary in features, but is independent of technical aspects like programming languages.
- Understand what "seeing the whole" means.
- Setup a Kanban board, from identifying initial tasks to completing them.
- Explain what "done" means from a Kanban perspective.

Other Agile Methodologies

Software engineering methodologies have evolved over the years. While Scrum and Extreme Programming are very popular and widely used in industry, other Agile methodologies have also been proposed with practices to address particular limitations.

For example, one variation is called the Agile Unified Process, or AgileUP. As its name suggests, this Agile methodology combines Agile practices with the Unified Process model. AgileUP applies the principles of Agile to increase developer morale and improve collaboration. It uses Agile practices, such as test-driven development and small releases to supplement the basic Unified Process. Like the Unified Process, AgileUP uses the same structure of phases. The methodology emphasizes simplicity, empowering people, and customizing the methodology to suit each project's needs.

In general, use a process, methodology, or practice that suits the needs of the product, team, and project. This may entail closely following something already established or starting there and customizing it as needed.

Other Agile methodologies include:

- Dynamic Systems Development Method
- Feature-Driven Development
- Behaviour-Driven Development
- Scrumban

Lean Software Design

One Agile methodology that has gained increasing attention is **Lean Software Development**. Its origin 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.

Seven Principles of Lean

Lean Software Development, or Lean for short, provides an Agile methodology that enables the effective creation of better software. Lean is based on seven principles:

1. Eliminate waste
2. Amplify learning
3. Decide as late as possible
4. Deliver at fast as possible
5. Empower the team
6. Build quality in
7. See the whole

These principles can be effective for both small and large projects.

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.

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 more alternatives 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.

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."

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.

Empower the Team

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

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.

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.

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.

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.

Additional Principles from Lean Software Developers

Mary and Tom Poppendieck wrote about these seven principles in their 2003 book, *Lean Software Development: An Agile Toolkit*. Over time, the Lean community has added other principles.

Use the Scientific Method

Initiate experiments to collect data and test ideas.

This principle encourages software product managers to base product features on real user data, rather than relying on hunches, guesses, or intuition. Collecting and analyzing this data allows clients and software developers to make informed choices about the product. Gain credibility by backing potential decisions with data.

Encourage Leadership

Enable individual developers to be courageous, innovative, inspirational, and collaborative. This principle extends empowering the team to bring out the best in each person on the team.

Warranty

Mary and Tom Poppendieck wrote this warranty in their 2003 book, *Lean Software Development: An Agile Toolkit*:

> Lean principles are warranted to be tried and proven in many disciplines, and when properly applied, they are warranted to work for software development. Proper application

> means that all of the lean principles are employed and that thinking tools are used to translate them into agile practices appropriate for the environment. This warranty is invalid if practices are transferred directly from other disciplines or domains without thinking, or if the principles of *empower the team* and *build integrity in* [emphasis added] are ignored (p. 186).

In essence, it will not work to use controlling management practices to enable Lean. While the core of Lean is about eliminating waste and improving the product, the most effective way to achieve that is to empower the development team to implement the product the way they want and ensure quality in the product's construction.

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.

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.

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.

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.

Version 2.0.0