DevNet Associate v1.0

Software Development

3.1.1

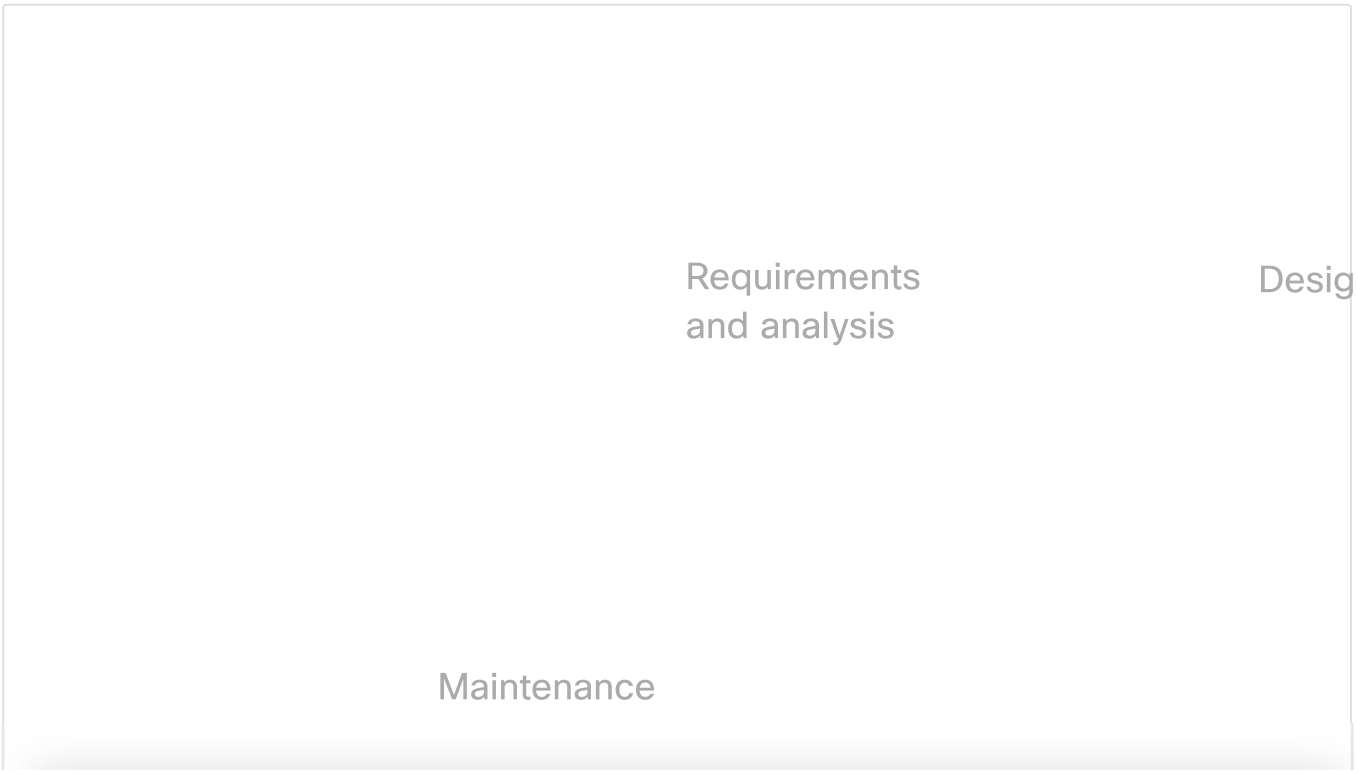
Introduction

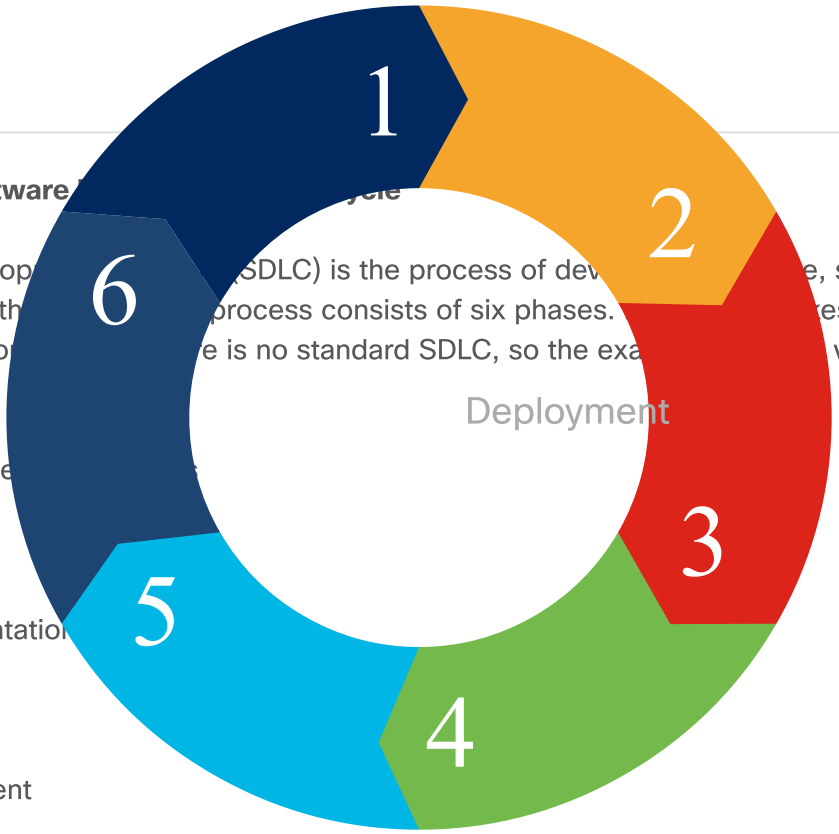
The software development process is also known as the software development life cycle (SDLC). This process is more than just coding. It also includes gathering requirements, creating a proof of concept, testing, and fixing bugs.

In this module, we discuss phases in the software development life cycle followed by methodologies for managing the real-world requirements of software projects. The methodologies discussed here begin with the waterfall method, which emphasizes up-front planning. Other methodologies in this topic, such as Agile and Lean, are more dynamic and adaptive than the waterfall method.

3.1.2

Software Development Life Cycle (SDLC)





Phases of the Software

The software develop (SDLC) is the process of dev e, starting from an idea and ending with process consists of six phases. es input from the results of the previo e is no standard SDLC, so the exa vary, but the most common are:

Phase 1. Requireme

Phase 2. Design

Phase 3. Implementatio

Phase 4. Testing

Phase 5. Deployment

Phase 6. Maintenance

Historically, development teams usually followed these phases in order in the waterfall method. The goal of waterfall was to complete each phase of SDLC down to the last detail before proceeding to the next, never returning to a prior phase, and writing everything down along the way.

Although the waterfall methods is still widely used today, it's gradually being superseded by more adaptive, flexible methods that produce better software, faster, with less pain. These methods are collectively known as “Agile development.”

It is important to understand that the SDLC can be applied many different ways. Its phases can be repeated, and the order reversed. Individual phases can be performed at many levels in parallel (for example, requirements can be gathered separately for user interface details, back-end integrations, operating and performance parameters, etc.).

We'll look at the phases of the SDLC in greater detail, and in their classic order (just remember: this is a description, not a prescription)

3.1.3

Requirements and Analysis Phase

The goal of the requirements and analysis phase is to answer several tiers of questions. These begin with exploring stakeholders' current situation, needs and constraints, present infrastructure, etc. They are determining what problem(s) this new software needs to solve.

After the problem is better-defined, more concrete issues can be explored to determine where, and in what context, the new software will need to live.

When answers to such questions are compiled, it's time to begin exploring more precise requirements by focusing on desired features and user experience (UX).

Finally, the team begins assessing architectural options for building the software itself. For most enterprise applications, this means many iterations of defining requirements for the software's front end and back end. You will also need to provide points of integration for other applications, as well as services for lifecycle management.

After gathering the requirements, the team analyzes the results to determine the following:

- Is it possible to develop the software according to these requirements, and can it be done on-budget?
- Are there any risks to the development schedule, and if so, what are they?
- How will the software be tested?
- When and how will the software be delivered?

At the conclusion of this phase, the classic waterfall method suggests creating a Software Requirement Specification (SRS) document which states the software requirements and scope, and confirms this meticulously with stakeholders.

3.1.4

Design and Implementation Phases

Design

The design phase classically takes the SRS document from the Requirements & Analysis phase as input. During the design phase, the software architects and developers design the software based on the provided SRS.

At the conclusion of the design phase, the team creates High-Level Design (HLD) and Low-Level Design (LLD) documents. The HLD gives a "10,000-foot view" of the proposed software. It describes the general architecture, components and their relationships, and may provide additional detail. The LLD, based on the HLD document, describes in much greater detail the architecture of individual components, the protocols used to communicate among them, and enumerates required classes and other aspects of the design.

Implementation

The implementation phase classically takes the HLD and the LLD from the design phase as an input.

The implementation phase is often called the coding or development phase. During this phase, the developers take the design documentation and develop the code according to that design. All of the

components and modules are built during this phase, which makes implementation the longest phase of the life cycle. During this phase, testing engineers are also writing the test plan.

At the conclusion of the implementation phase, functional code that implements all of the customer's requirements is ready to be tested.

3.1.5

Testing, Deployment, and Maintenance Phases

Testing

The testing phase classically takes the software code from the implementation phase as input. During this phase, the test engineers take the code and install it into the testing environment so they can follow the test plan. The test plan is a document that includes a list of every single test to be performed in order to cover all of the features and functionality of the software, as specified by the customer requirements. In addition to functional testing, the test engineers also perform:

- Integration testing
- Performance testing
- Security testing

When code doesn’t pass a test, the test engineer identifies the bug, which gets passed to the developers. After the bug is fixed, the test engineers will re-test the software. This back and forth between the test and development engineers continues until all of the code passes all of the tests.

At the conclusion of the testing phase, a high quality, bug-free, working piece of software is ready for production, in theory. In practice, this rarely happens. Developers have learned how to test more efficiently, how to build testing into automated workflows, and how to test software at many different levels of detail and abstraction: from the tiniest of low-level function definitions to large-scale component aggregations. They've also learned that software is never bug-free, and must instead be made observable, tested in production, and made resilient so it can remain available and perform adequately, despite issues.

Deployment

The deployment phase takes the software from the testing phase as input. During the deployment phase, the software is installed into the production environment. If there are no deployment issues, the product manager works with the architects and qualified engineers to decide whether the software is ready to be released.

At the end of the deployment phase, the final piece of software is released to customers and other end users.

Maintenance

During the maintenance phase, the team:

- Provides support for customers
- Fixes bugs found in production
- Works on software improvements
- Gathers new requests from the customer

At the conclusion of the maintenance phase, the team is ready to work on the next iteration and version of the software, which brings the process back to the beginning of the SDLC and the requirements and analysis phase.

3.1.6

Software Development Methodologies

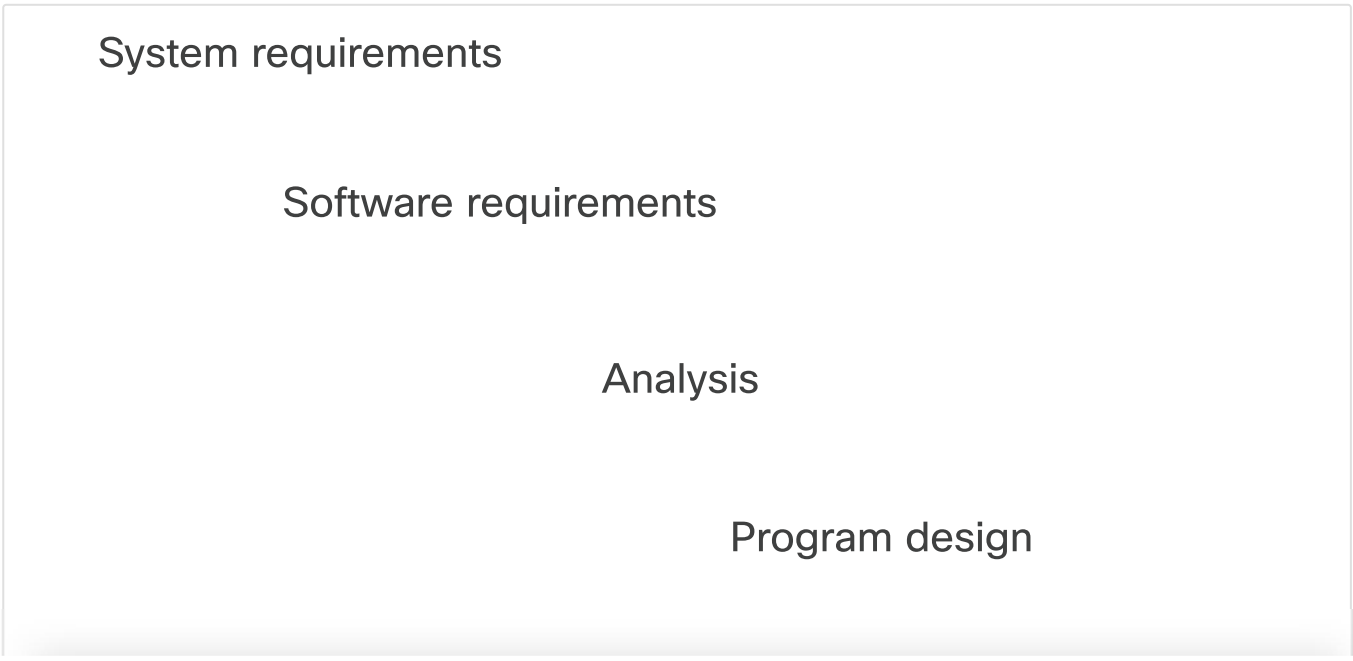
A software development methodology is also known as a Software Development Life Cycle model. These methodologies are nothing more than a set of rules, steps, roles and principles. Many different methodologies exist, but we will focus on the three most popular:

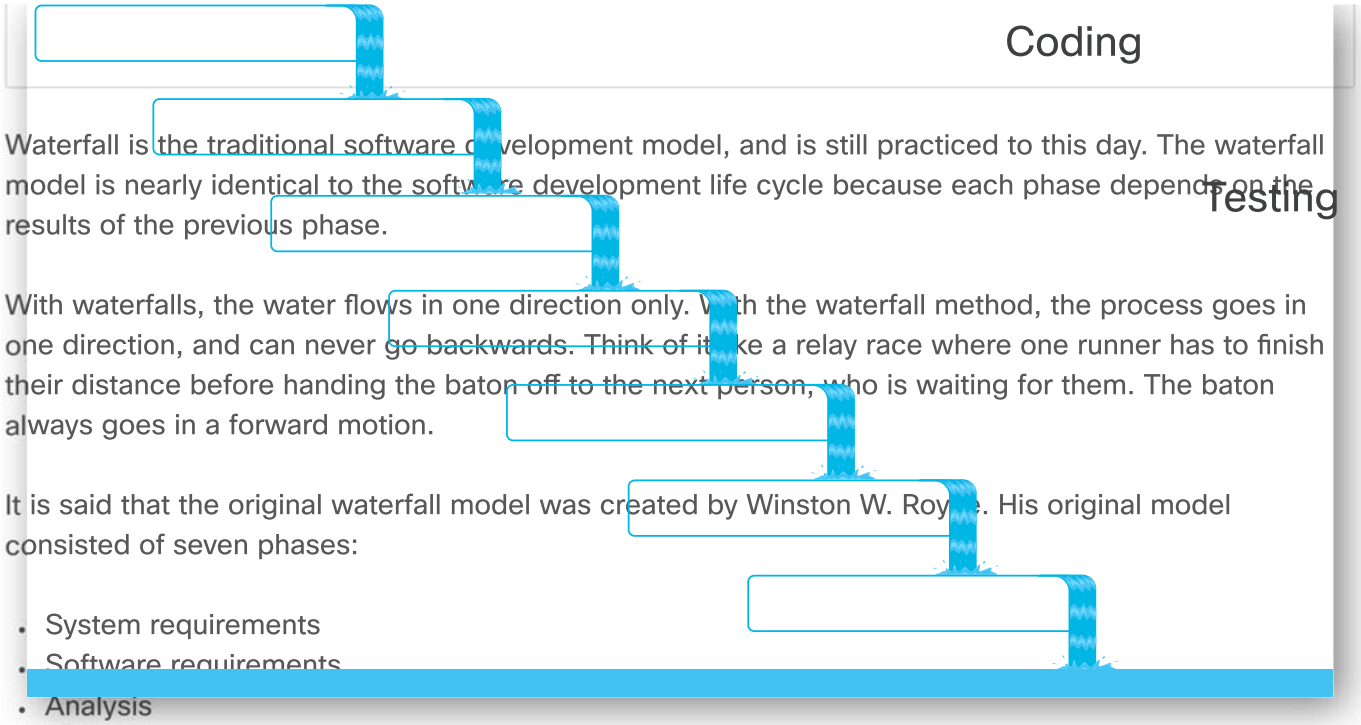
- Waterfall
- Agile
- Lean

Each methodology has its own pros and cons. Deciding on which to use depends on many factors, such as the type of project, the length of the project, and the size of the team.

3.1.7

Waterfall Software Development





Waterfall is the traditional software development model, and is still practiced to this day. The waterfall model is nearly identical to the software development life cycle because each phase depends on the results of the previous phase.

With waterfalls, the water flows in one direction only. With the waterfall method, the process goes in one direction, and can never go backwards. Think of it like a relay race where one runner has to finish their distance before handing the baton off to the next person, who is waiting for them. The baton always goes in a forward motion.

It is said that the original waterfall model was created by Winston W. Royce. His original model consisted of seven phases:

- System requirements
- Software requirements
- Analysis
- Program Design
- Coding
- Testing
- Operations

As you can see, the waterfall model is really just one iteration of the software development life cycle. There are now many variations of the phases in the waterfall model, but the idea that each phase cannot overlap and must be completed before moving on remains the same.

Because the outcome of each phase is critical for the next, one wrong decision can derail the whole iteration; therefore, most implementations of the waterfall model require documentation summarizing the findings of each phase as the input for the next phase. If the requirements change during the current iteration, those new requirements cannot be incorporated until the next waterfall iteration, which can get costly for large software projects, and cause significant delays before requested features are made available to users.

3.1.8

Agile Software Development

The Agile method is flexible and customer-focused. Although methodologies similar to Agile were already being practiced, the Agile model wasn't official until 2001, when seventeen software developers joined together to figure out a solution to their frustrations with the current options and came up with the Manifesto for Agile Software Development, also known as the Agile Manifesto.

Agile Manifesto

According to the Agile Manifesto, the values of Agile are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The manifesto lists twelve different principles:

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

3.1.9

Agile Methods

Agile Methods

The Agile Manifesto by design, wasn't precise about how Agile should work. After forging the Manifesto, its originators (and many others) kept evolving these ideas, absorbing new ideas from many sources, and testing them in practice. As a result, over the past few decades, many takes on Agile have emerged, and some have become widely popular. These include:

- **Agile Scrum -** In rugby, the term scrum describes a point in gameplay where players crowd together and try to gain possession of the ball. The Scrum focuses on small, self-organizing teams that meet daily for short periods and work in iterative sprints , constantly adapting deliverables to meet changing requirements.
- **Lean -** Based on Lean Manufacturing, the Lean method emphasizes elimination of wasted effort in planning and execution, and reduction of programmer cognitive load.
- **Extreme Programming (XP) -** Compared with Scrum, XP is more prescriptive about software engineering best-practices, and more deliberately addresses the specific kinds of quality-of-life issues facing software development teams.

- **Feature-Driven Development (FDD) -** FDD prescribes that software development should proceed in terms of an overall model, broken out, planned, designed, and built feature-by-feature. It specifies a process of modeling used to estimate and plan feature delivery, and a detailed set of roles for both the core development team and support people.

Of the methodologies described above, Agile Scrum is probably the most popular. We'll discuss some Scrum terms and concepts below that have been more or less universally adopted by the Agile community across all methodologies.

Sprints

In the Agile model, the software development life cycle still applies. Unlike the waterfall method, where there is one long iteration of the SDLC, Agile is many quick iterations of the SDLC.

These quick iterations are called sprints, and the purpose of sprints is to accomplish the frequent delivery of working software principle of the Agile manifesto. A sprint is a specific period of time (time-boxed) which is usually between two weeks and four weeks, but preferably as short as possible. The duration of the sprint should be determined before the process begins and should rarely change.

During a sprint, each team takes on as many tasks, also known as user stories, as they feel they can accomplish within the time-boxed duration of the sprint. When the sprint is over, the software should be working and deliverable, but that doesn't necessarily mean that it will be delivered; a sprint doesn't always lead to a release, but Agile requires the software to remain deliverable.

Backlog

It is the role of the product owner to create the backlog. This backlog is made up of all of the features for the software, in a prioritized list. The features in the backlog are a result of the Requirements & Analysis phase, and include features that won't necessarily be in the immediate release. New features can be added to the backlog at any time, and the product owner can reprioritize the backlog based on customer feedback.

User stories

When a feature gets close to the top of the priority list, it gets broken down into smaller tasks called user stories. Each user story should be small enough that a single team can finish it within a single sprint. If it's too large to be completed in a single sprint, the team should break it down further. Because the software must be deliverable at the end of each sprint, a user story must also abide by those rules.

A user story is a simple statement of what a user (or a role) needs, and why. The suggested template for a user story is:

As a `<user|role>`, I would like to `<action>`, so that `<value|benefit>`

Completing a user story requires completing all of the phases of the SDLC. The user story itself should already have the requirements defined by the product owner, and the team taking on the user story needs to come up with a design for the task, implement it and test it.

Scrum Teams

Scrum teams are cross-functional, collaborative, self-managed and self-empowered. Ideally, these scrum teams should not be larger than 10 individuals, but they should be big enough to finish a user story within a sprint.

Every day, each scrum team should have a daily standup. A standup is a meeting that should last no longer than 15 minutes, and should take place at the same time every day. In fact, it's called a "standup" because ideally it should be short enough for the team to accomplish it without having to sit down.

The goal of the daily standup is to keep all team members in sync with what each person has accomplished since the last standup, what they are going to work on until the next standup, and what obstacles they have that may be preventing them from finishing their task. The scrum master facilitates these standups, and their job is to report and/or help remove obstacles.

3.1.10

Lean Software Development

Lean software development is based on Lean Manufacturing principles, which are focused on minimizing waste and maximizing value to the customer.

In its simplest form, Lean Software Development delivers only what the customers want. In the book Lean Software Development: An Agile Toolkit , there are seven principles for lean:

- Eliminate waste
- Amplify learning
- Decide as late as possible
- Deliver as fast as possible
- Empower the team
- Build integrity in
- Optimize the whole

Eliminate waste

Waste is anything that does not add value for customers. The definition of value is subjective, however, because it's determined by the customer. Eliminating waste is the most fundamental lean principle, the one from which all the other principles follow.

To eliminate waste, you must be able to understand what waste is. Waste is anything that does not add direct value to the customer. There are seven wastes of software development:

- Partially Done Work
- Extra Processes
- Extra Features
- Task Switching
- Waiting
- Motion

- Defects

Partially done work

Partially done work is a waste because:

- It doesn't add any value to the customer
- The time and resources spent doing this work could have been used on something that is of value to the customer
- The work usually isn't maintained, so it eventually becomes obsolete

Extra processes

Extra processes are just like a bunch of paperwork. As a result, they are a waste for pretty much the same reasons as partially done work.

Extra features

If the customer didn't ask for it, it doesn't bring them value. It might be nice to have, but it's better to use the resources to build exactly what customers want.

Task switching

Humans need time to switch their mind to focus on another task, and that time spent switching contexts is a waste. Task switching wastes a resource's (person’s) time, so it's a waste to assign a resource to multiple projects.

Waiting

Many people would agree that by definition, waiting is a big waste of time. So, any type of delay is a waste. Examples of a delay in software development are delays in:

- starting the project
- getting the right resources (staff)
- getting the requirements from the customer
- approvals of documentation
- getting answers
- making decisions
- implementation
- testing

Motion

Lean software development defines motion for two things: people and artifacts. Motion for people is when people need to physically walk from their desk to another team member to ask a question, collaborate, and so on. When they move from their desk, it is not only the time it takes for them to get to the destination that is a waste, but also the task switching time.

The motion of artifacts is when documents or code are moved from one person to another. Most of the time, the document doesn't contain all of the information for the next person, so either that person has to gather the information again, or the hand-off requires time, which is a waste.

Defects

Unnoticed defects (otherwise known as bugs) are a waste because of the impact of the defect. The defect can cause a snowball effect with other features, so the time it takes to debug it is a waste. Also, for a customer, the value of the software is reduced when they run into issues, so the feature ends up being a waste.

3.1.11

Lean Software Development (Cont.)

Amplify Learning with Short Sprints

To be able to fine tune software, there should be frequent short iterations of working software. By having more iterations:

- Developers learn faster
- Customers can give feedback sooner
- Features can be adjusted so that they bring customers more value

Decide as Late as Possible

When there is uncertainty, it is best to delay the decision-making until as late as possible in the process, because it's better to base decisions on facts rather than opinion or speculation.

Also, when a decision isn't yet made, the software is built to be flexible in order to accommodate the uncertainty. This flexibility enables developers to make changes when a decision is made--or in the future, if requirements change.

Deliver as Fast as Possible

Delivering the software faster:

- Enables customers to provide feedback
- Enables developers to amplify learning
- Gives customers the features they need now
- Doesn't allow customers to change their mind
- Makes everyone make decisions faster
- Produces less waste

You'll notice that each of these reasons practices at least one of the previously discussed lean principles.

Empower the Team

Each person has their own expertise, so let them make decisions in their area of expertise. When combined with the other principles such as eliminating waste, making late decisions, and fast

deliveries, there isn't time for others to make decisions for the team.

Build Integrity In

Integrity for software is when the software addresses exactly what the customer needs. Another level of integrity is that the software maintains its usefulness for the customer.

Optimize the Whole

Although one of the principles is empowering the team, each expert must take a step back and see the big picture. The software must be built cohesively. The value of the software will suffer if each expert only focuses on their expertise and doesn't consider the ramifications of their decisions on the rest of the software.

3.1.12

Lab - Explore Python Development Tools

In this lab, you review Python installation, PIP, and Python virtual environments.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Review the Python Installation
- Part 3: PIP and Python Virtual Environments
- Part 4: Sharing Your Virtual Environment

Explore Python Development Tools

3.0 Introduction

3.2 Software Design Patterns