UNIVERSITY OF ALBERTA
DEPARTMENT OF COMPUTING SCIENCE

AGILE PLANNING FOR SOFTWARE PRODUCTS

COURSE NOTES

Table of Contents

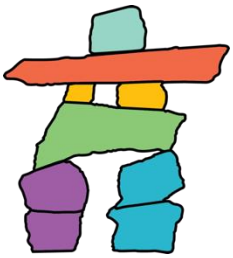
Course Overview

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

- Understand the importance of planning.
- Explain the role uncertainty plays in affecting plans and task estimates.
- Break down tasks into appropriate sizes, and create a Work Breakdown Structure (WBS) for a project.
- Explain the differences between estimates, targets, and commitments.
- Use different estimation techniques, including story points, velocity estimates and planning, time boxing, and Gantt charts.
- Generate work estimates for software products using a variety of approaches, including bottom-up, analogy, experts, and the estimate formula.
- Understand task dependencies, the different types of task dependencies, and how these affect project planning.
- Create Critical Path Method (CPM) charts and Program Evaluation Review Technique (PERT) charts.
- Create effective plans for software development on both iteration and release levels.
- Identify anti-patterns and types of risks that may affect projects.
- Assess and plan for project risks.

Introduction

Welcome to the **Agile Planning for Software Products** course, within the Software Product Management Specialization. This course will focus on agile planning techniques. These techniques ensure that the software product project is managed and organized properly while also being adaptive to change.



The first two courses of this specialization, **Software Processes and Agile Practices** and **Client Needs and Software Requirements**, serve as foundational pillars in learning Agile—they are the "legs" of the inukshuk that depicts the structure of our specialization. This course, **Agile Planning for Software Products** serves as the body of the inukshuk, and rests on the "legs." This means **Agile Planning for Software Products** brings together many of the fundamentals of those two prior courses and builds on them.

In the introductory course, you learned that building better software involves three goals. These goals are:

- Have the right product
- Ensure it is done right
- Ensure it is managed right

When a project is ***managed right***, it adopts suitable practices to organize the work of everyone involved and leads to clear next steps. This course focuses on ensuring a project is ***managed right***.

There are three key types of planning used in software projects. This course will cover all three of them. The types of planning are:

- Release planning
- Iteration planning
- Risk planning

There are four modules in this course.

Module 1 will cover:

- Major uncertainties that projects need to resolve
- How to break down work into manageable pieces
- The difference between an estimate, a target, and a commitment

Module 2 will cover how to:

- Estimate the size of a requirement using story points
- Measure a development team's productivity using velocity
- Plan work using time boxing
- Use a Gantt chart
- Construct a release plan by mapping out the delivery of user stories over upcoming sprints

Module 3 will cover how to:

- Estimate the time it will take to finish a task
- Determine task dependencies
- Use Critical Path Method (CPM) diagrams and Program Evaluation Review Technique (PERT) charts to plan work at the task level
- Construct an iteration plan by mapping out tasks to be done within a sprint

Module 4 will cover:

- Problems or risks that could cause a project to fail
- Anti-patterns (situations with negative consequences for a project and its people)
- Risk assessment and the level of action to take to address those risks
- Constructing a risk plan by mapping out what to do should a risk occur

Module 1: Introduction to Planning

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

- Explain the importance of planning in a project.
- Describe the key terms within Software Project Management: task, role, schedule, milestone, work product.
- Recognize and explain an Uncertainty Space (AUSP) diagram, and the concepts it illustrates: means and ends uncertainty, and how they can evolve over a project.
- Define work breakdown structure.
- Recognize how the steps of project initiation, hardware/equipment acquisition, development, testing, and implementation can be represented in a work breakdown structure.
- Create a work breakdown structure based on an example.
- Identify the appropriate size for a task.
- Explain the terms: estimate, target, and commitment.
- Provide examples of estimates, targets, and commitments.
- Understand what a good estimate entails.

Introduction to Planning

This course covers strategies and techniques for planning software projects within the Agile manifesto. This means the strategies and techniques covered must leave the project adaptable to change!

The key to planning software projects is breaking the project workload into small, manageable tasks. This involves describing and organizing tasks, and assigning them time and work estimates. Based on the identified tasks and their time and work estimates, an accurate schedule for the project can be made.

Software projects can be scheduled many ways on release and iteration levels. Risk planning must also be incorporated into creating schedules on both levels.

The **release** level refers to the stage where products delivered are ready for the market. **Release planning** is used to plan the project as a whole. Common release planning techniques include Gantt charts and release plans.

Iterations refer to smaller pieces of the project. They usually include a number of requirements that must be finished. A release is made of many iterations. **Iteration planning** is a technique that involves designing and developing the tasks that will be completed in a certain iteration of the project. Common iteration planning techniques include PERT charts, CPM charts, and iteration plans.

Both release planning and iteration planning must address potential **risks** that can arise in software development. **Risk planning** involves preparing for those risks in order to avoid negative repercussions in both creation and support of the product.

Important Terms

A number of important terms are used over this course that are related to planning. These should be well understood, before moving on to more complex discussions of planning software projects within Agile.

Term	Definition
Task	A **task** is a small, manageable step of a project that must be completed. Tasks are essential to a project—everything in a project can be broken down into tasks.
Role	A **role** is a duty that a person takes on or plays in some relation to the product, as explained in the course **Software Processes and Agile Planning**. Examples of roles include programmer, tester, designer, client, and product manager.
Work product	A **work product** is an output produced by a task or process, as explained in the course **Software Processes and Agile Planning**.
Schedule	A **schedule** is created when the tasks of a project are mapped to a timeline.
Milestones	**Milestones** refer to internal checkpoints used to measure progress. They are not time based, but event or action based. Milestones are most commonly used in linear and early iterative processes of a project, as explained in the course **Software Processes and Agile Practices**. In Agile, milestones are not generally used. Progress is instead measured by working software as opposed to events or actions. Releases and iterations tend to be time based as well, which do not fit with milestones.

These terms have certain dependencies on each other.

- Roles perform tasks. For example, programmers write code for selected features.
- Tasks not only create work products, but also might use them. This means that work products might be dependent on one another. An output work product of a task might be used as an input work product for a subsequent task. For example, the task of writing tests produces the output work product of tests. These tests are then the input work product for the task of executing tests.
- Tasks, once identified, can be used to create schedules. Work and time estimates can be assigned to tasks, and project schedules can be built from these estimates.

Understanding dependencies is important. Dependencies influence how tasks need to be prioritized, and the consequential project schedules. For example, work products created in one task that are then used as an input for a different task must be created before they are needed in the project. Recognizing dependencies is therefore important for planning what to complete in each iteration of the project.

Previous Concepts

A brief review of previous concepts helps show how previous courses relate to planning.

Tasks, as described above, are an important part of planning a project, since everything in a project can be broken down into tasks. Tasks also played important roles in previous courses.

In the course **Client Needs and Software Requirements**, you learned about creating requirements. After the requirements have been created, they can be prioritized and broken down into developer tasks to be completed. These developer tasks can be more technology specific than requirements, which should generally be independent of technology. Developer tasks are commonly scheduled using an iteration plan, as you will learn in this course.

In the course **Software Processes and Agile Practices**, you learned about creating processes. Tasks are essential to creating processes, as they make up activities, and activities make up phases. Each phase therefore has specific tasks that must be completed.

Tasks are also important to Agile practices. Agile practices provide ways of organizing tasks and ensuring they are effectively completed. Iteration plans, as described above, are an Agile practice and used commonly in Scrum methodology. Scrum is a popular methodology for planning and organizing work on software products that follows the Agile philosophy.

Uncertainty Space

When beginning a project, there are many uncertainties. These uncertainties are often grouped in two types. A well-managed project must identify and address both types, simultaneously. The two types of uncertainty can be best understood through the questions:

- What are you creating in the product?
- How will the product be developed?

The first question is associated with **ends uncertainty**. Ends uncertainty refers to uncertainty about what a product will do. The second question is associated with **means uncertainty**. Means uncertainty refers to uncertainty in how a product will be designed and developed.

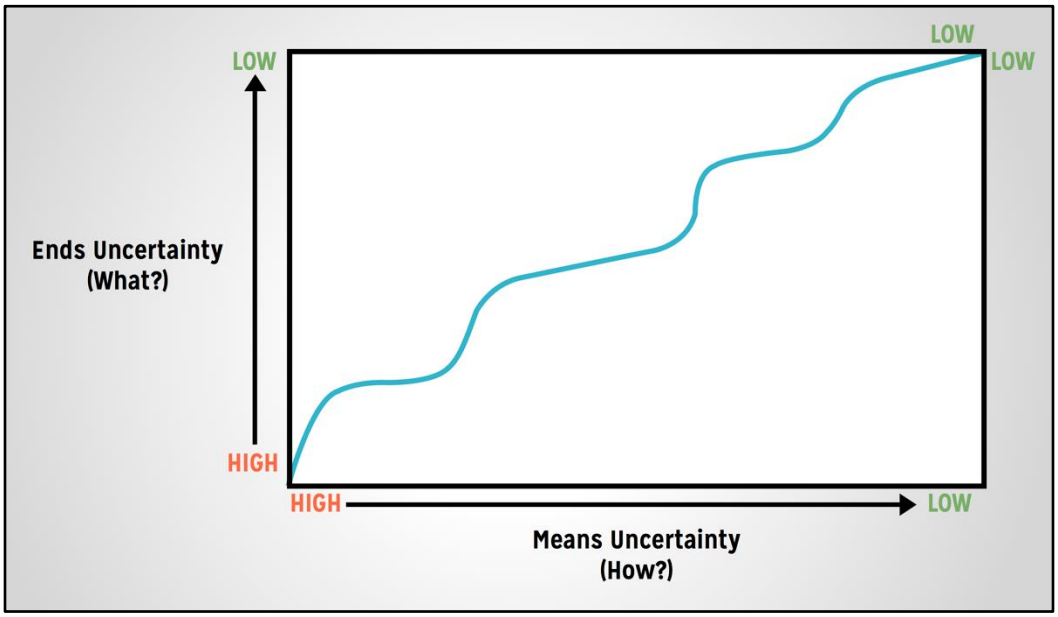
DID YOU KNOW?

The grouping of project uncertainties into the two types: means uncertainty and ends uncertainty was first proposed by the project management researcher, Alexander Laufer in 1997 in his book *Simultaneous Management*. Laufer's work is noted for using real stories from successful organizations, so his project management theories are practice based.

Mike Cohn, one of the contributors to Scrum methodology, depicted the levels of uncertainty of a product as it proceeds from concept, through development, towards an operational state in an **uncertainty space diagram**. In such a diagram:

- The **means uncertainty** should go from high to low, along the x-axis
- The **ends uncertainty** should go from high to low, along the y-axis

This is demonstrated the diagram below.



At the beginning of a project, the uncertainty for both “what” is being created and “how” it will be created are very high, especially because requirements and methods will change over time. By the end of a project, however, what will be created and how it will be created should be clear, and uncertainties are low. At any point, if there is a high uncertainty for either axis, then some uncertainty must still be solved for the product.

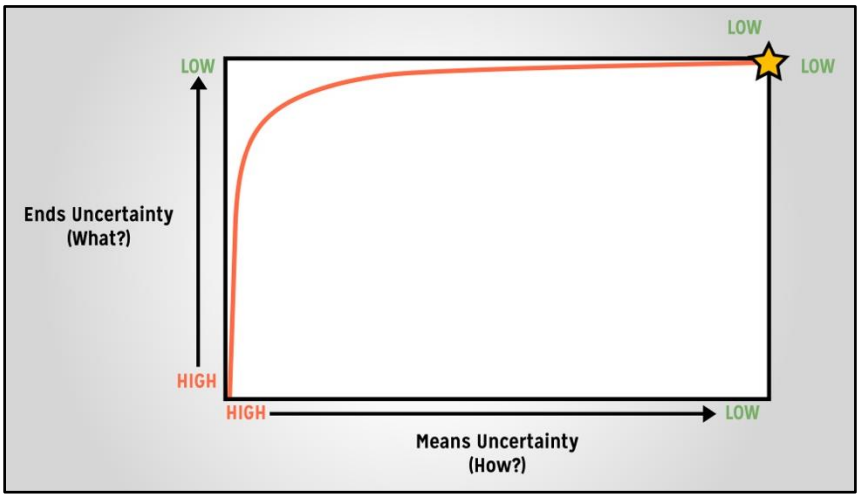
Navigating Uncertainty Space Diagrams

There are many different ways to navigate through projects using the uncertainty space diagram:

1. Determine the “what” of a project first, then the “how.”
2. Determine “how” to develop a project, and then create it as you go.
3. Determine both the “what” and “how” of a project with as much balance between the two as possible.

The first way identifies what the project will create and then how to create it. Using this method, requirements are all determined first, and in great detail, before any plans for completing them are developed.

This method is likely following a waterfall process.



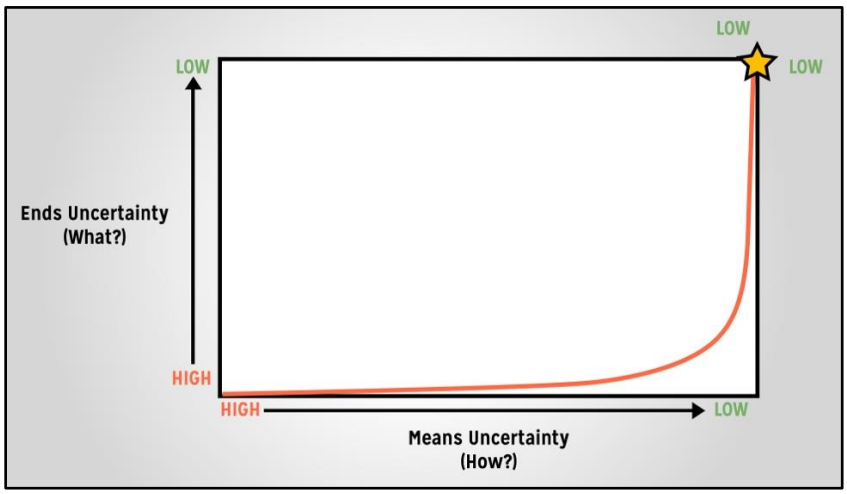
As illustrated above, this method leads to having low levels of ends uncertainty before the level of means uncertainty changes. Often, this change is significant when it happens.

This method of uncertainty planning is not very adaptable because requirements are established early and cannot change easily. It is therefore not in line with Agile methods.

The second way of navigating a project determines how to develop a product and then creates the product as the project moves along. Basically, the development team determines implementation solutions and makes the product up as they go.

Ad Hoc Development often refers to development that is "made up as you go".

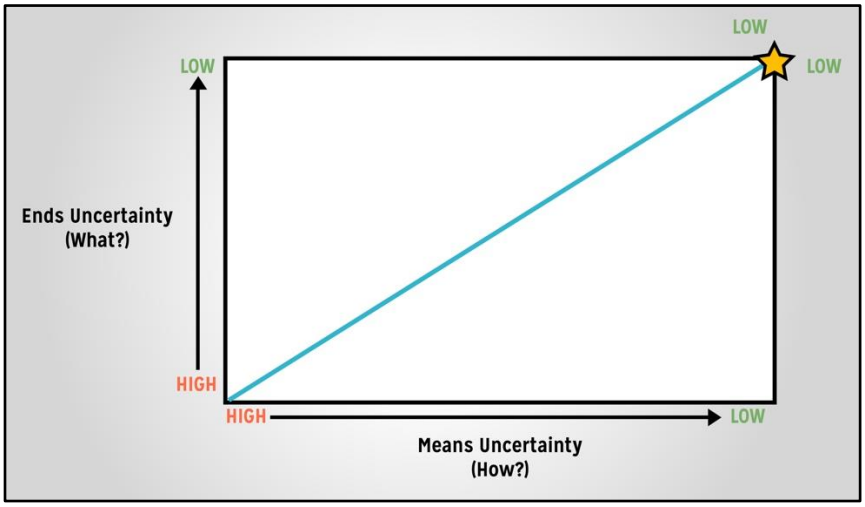
This is **ad hoc** development, which involves little planning.



The chart above illustrates that second way leads to low levels of means uncertainty before the level of ends uncertainty changes. As in the previous method, the change in uncertainty is significant when it happens.

This method of uncertainty planning can lead to wasted work if what is developed is not in line with client needs.

The third way of navigating a project is somewhere between the first two. Both the “how” and the “what” are accounted for around the same time, leading to balance between the two extremes. This is a more efficient means of navigating uncertainty than either of the previous two methods.



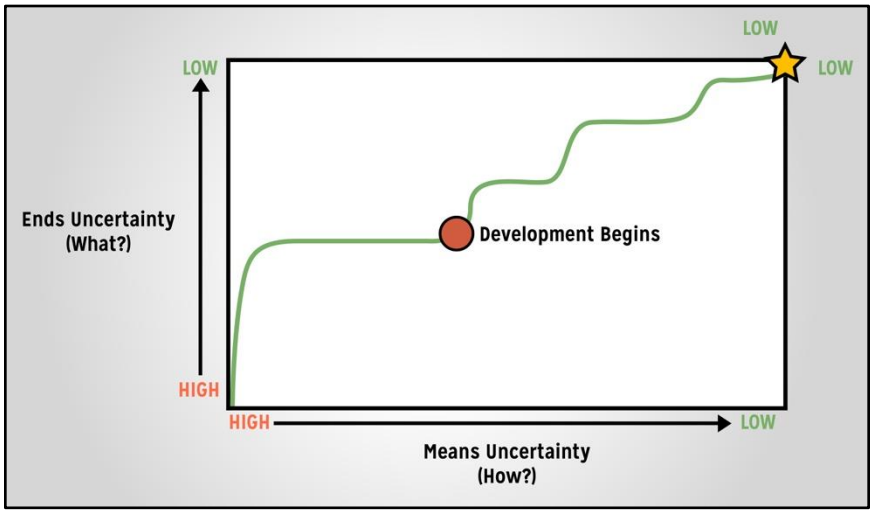
As demonstrated in the preceding graph, the levels of both ends uncertainty and means uncertainty lower at an equivalent rate.

However, development is not always this straightforward, as sometimes requirements are changed or added that will affect and create new uncertainties in both “what” the product will be and “how” it can be built.

A Final Note on Uncertainty Spaces

In most real-world cases, clients and the development team will define the majority of requirements for the project before planning begins. During this process, there might be less ends uncertainty than means uncertainty. However, once product development begins and the “how” is planned out more, the means uncertainty will lower.

As requirements are added or changed over the course of the project, there might be a slight fluctuation towards lower ends uncertainty, followed by lower means uncertainty. Development and solutions change as requirements change, so uncertainty will look more like the diagram below, instead of a straight line.



Both ends and means uncertainty should ultimately work their way towards being low. As changes or added requirements happen and any corresponding changes in development follow quickly, the project will stay adaptable and avoid wasted time and resources.

No matter the method of navigating uncertainty space, tasks are needed for the project to reach its end.

Work Breakdown Structure (WBS)

Creating a software product involves many tasks, including determining requirements and features, designing them, writing code, and testing. Often, other kinds of work are also involved, such as marketing the product and arranging technical support and operations. All of these require planning—the development team cannot simply begin developing the project without any plans. The key to being able to plan a software project is to break it down into manageable pieces.

In project management, the technique of **work breakdown structure (WBS)** creates a visual representation of the process of breaking down large project work products into smaller task work products. Tasks should be broken down until each is small and clear enough to be assigned to a single person. WBS is therefore also a hierarchical representation of project tasks and deliverables. It can take on the form of a list or other visual representation. Dates and order of execution are not included in a WBS.

Instructor's Note:

A work breakdown structure is similar to concepts you learned in the course **Software Processes and Agile Practices**. For example, a process is broken down into activities, which in turn are broken down into tasks.

Creating a work breakdown structure is a good first step in putting together a detailed plan for the project. It is important to be balanced in breaking down tasks within a work breakdown structure. Although tasks should be broken down until they are manageable, if they are too small, this will lead to micromanagement.

TIP:

The acronym SMART is sometimes applied to determine if a task or work product is a good size. The letters in SMART can stand for many things, but it generally means that a task should be:

- Specific (the task should be clear and should not overlap with other tasks)
- Measurable (the task should have a definite way of being marked as finished or done)
- Achievable (one person should be able to do the task)
- Relevant (the task should be within scope of the project)
- Time boxed (the task can fit within a certain time frame)

Some guides to creating WBS suggest other more specific rules of thumb to help know if tasks are manageable, including that the lowest level should take no more than 80 work hours to finish.

Another important tip is to remember to keep tasks within the scope of the project! This ties into the 100% rule often associated with WBS, which states that 100% of the work defined by the project scope should be included in the WBS.

WBS can be organized functionally or by phase. Common steps included are:

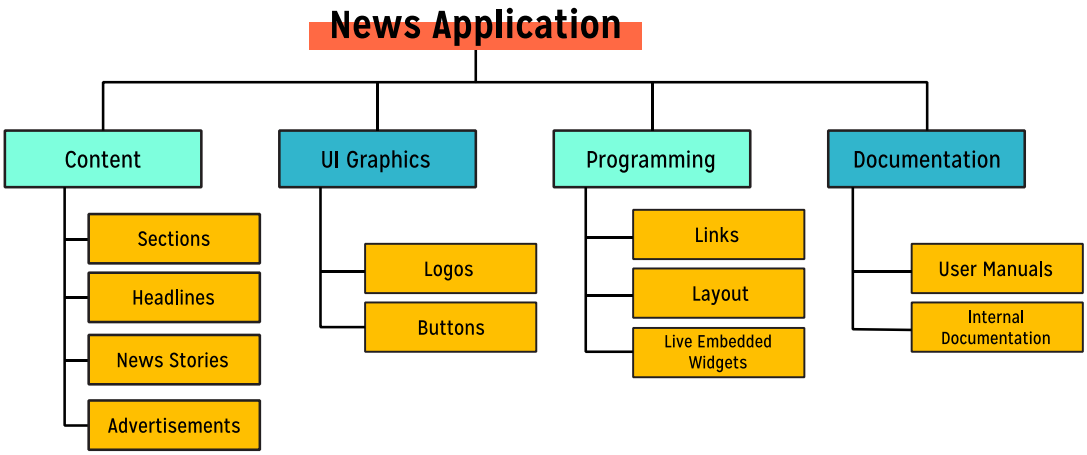
- Project initiation (when a new project is started and objectives, scope, purpose, and products are decided)
- Hardware and equipment acquisition (determining what hardware or equipment is needed to create the product, and obtaining those)
- Development (designing and creating the product)
- Testing (determining the performance and reliability of the product to identify and fix problems)
- Implementation (when the product is executed for outsiders of the project to be able to access and use)

These steps should be found at the highest level of the WBS (underneath the top level, which should be the final product or project), and then broken down appropriately in subsequent levels. A good resource for learning more about WBS levels is the Project Management Body of Knowledge Guide (PMBOK) (see **Course Resources**).

In general, final tasks identified in the work breakdown structure should be independent and unique of other tasks.

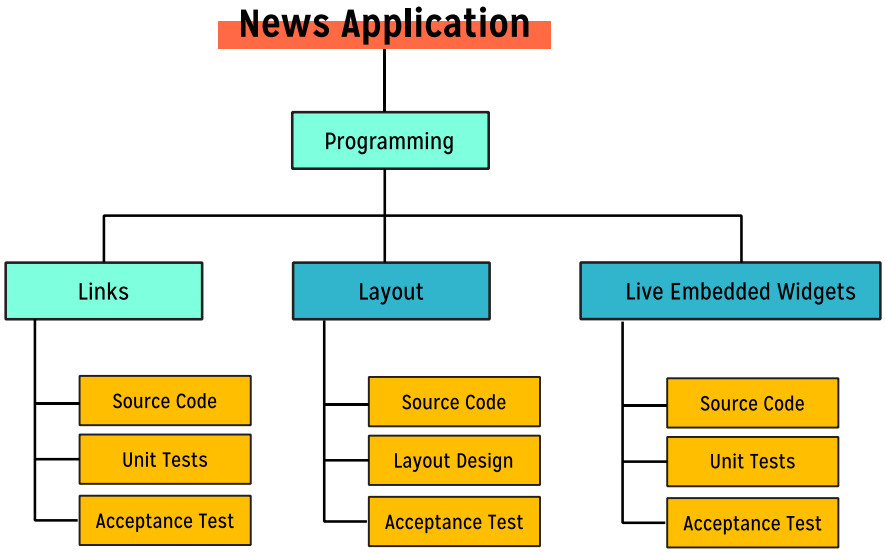
Example Work Breakdown Structure

Below is a graphic that illustrates how a work breakdown structure might look. In this example, the project is to create a News Application.



The product is the highest level of the WBS, and it is subsequently broken down into smaller tasks in a hierarchical tree. The level under the highest shows steps similar to phases, including programming and documentation.

Looking more closely at the “Programming” branch and its sub-branches, “Links/Buttons,” “Layout,” and “Live Embedded Widgets,” you can see that each of these items can be broken up into smaller tasks.



For example, all three need a source code. Any work products will also have associated tasks in the form of creating unit tests and acceptance tests.

Unit tests test the low-level functionality of source code; they should be written before source code is. For the sub-branch “Links,” a unit test could determine if links are working appropriately, for example. Acceptance tests, on the other hand, verify the application as a whole. In this same sub-branch, an acceptance test could test the interface to make sure that all the links are functional and correct, for example.

Designing the layout is another work product that should be listed, as well as any acceptance tests associated with the design that make sure the layout is compatible and functional on multiple devices.

These smaller tasks of creating designs and writing tests are good-sized work products. They are much more manageable than a large task such as “create a news application” would be. They are also in scope, unlike a task such as “create entire source code for reader chat room.” A task like the last one should be not be included in a work breakdown structure, because it is too large to tackle easily, and can easily result in missed smaller tasks and nuances.

Creating a Work Breakdown Structure

Large projects may have hundreds or thousands of tasks, and using a simple WBS tree chart similar to our example can quickly become unwieldy. Many other tools are available to help create work breakdown structures on larger scales, including Microsoft Excel and Microsoft Project. Both are better suited for large-scale projects.

Work breakdown structures are also not limited to product management concerns surrounding the Design and Implementation phase. They might also include Specification phase activities, Verification and Validation phase activities, promotion, arranging technical support, gathering analytics, setting up servers, etc.

Uses of Work Breakdown Structures

Work breakdown structures can be used many ways.

The most common use of a WBS is to determine tasks for the project based on the work products displayed. Small, manageable work products can easily be turned into small, manageable tasks. If the work product is “Source Code” for “Links,” as in the example above, then the task “Write source code for Links” can be easily created. This is very helpful in the early stages of planning a project because it considers the entire project, and tasks can then be used to create schedules.

In a similar vein, work breakdown structures can be used to help assess how long it will take a development team to complete a project. It is easier to estimate how long small tasks will take to finish, and then to add those times together to determine a project estimate, than to estimate the project as a whole.

WBS may also be used to determine potential risks the project will encounter. Once work products have been identified through a WBS, risks can be determined for each of those work

products. This is much more efficient than trying to determine risks from looking at the product as a whole.

Finally, a work breakdown structure can also be used to demonstrate a product to a client in a way that allows the client to look at features individually.

Estimates, Targets, and Commitments

The terms estimate, target, and commitment are often used interchangeably in project management. But, in fact, all three of these terms mean something different.

Estimates

An **estimate** is a guess of the time it will take the development team or worker to complete a task. A good estimate is usually based on a well-formed task, which adheres to SMART. They are also usually determined by the developers who will work on the task.

An estimate should be as accurate as possible, and it is often based on data, like the amount of time it took to accomplish a similar task. However, because a precise number of hours or days is probably inaccurate, estimates are usually ranges.

It is important to understand that although estimates are sometimes presented as ranges, they are not negotiable. This means that an estimate should *only* take into account previous work. It should not include padded or extra time or be affected by what stakeholders or managers want the estimate to be.

Other issues that might affect the ability to make a good estimation include:

- Lack of developer experience with similar tasks
- Too much attention to detail on tasks leading to over-estimates
- Not enough information available on the task to create an accurate estimate

This last issue, lack of information leading to difficulties in creating a time estimate relates to the **Cone of Uncertainty Principle**. This principle suggests that time estimates for features are less accurate the further into the future those features are intended to be developed. For example, estimating development is more difficult at the beginning of the project, when ideas are still being passed around, than towards the end of the project, closer to actual development stages.

The Cone of Uncertainty Principle is explored in larger detail in the lesson **Estimating Task Time** in **Module 3: Planning** of this course. It was also touched upon in the previous course, **Client Needs and Software Requirements**.
Developers may also express reluctance in determining an estimate if they fear not being able to meet an estimate. This could be a sign of larger organizational difficulties within the project. Clients may assume that estimates are targets or commitments, which could be problematic. Clear discussions about estimates, targets, and commitments between the development team and clients helps address this issue.

Approaches for creating an estimate include:

- Bottom-up
- Analogy
- Experts
- Using an estimate formula

More information on how to use these approaches can be found in the lesson **Estimating Task Time** of **Module 3: Planning** in this course.

Targets

A **target** is a specific point in the schedule for the project or part of the project to meet. It is almost an ideal deadline, and is usually set externally to the development team. Like estimates, targets are not negotiated. Once a target has been decided, it must be adhered to for product success and client satisfaction. In fact, targets are often contractual and can be used in arranging product promotions.

Examples of targets are ends of a sprint or the release date of a product.

Commitments

Commitments are what you agree to deliver. Commitments are where negotiations can happen. Based on both estimates and targets, a commitment can be negotiated. A good question to ask when determining commitments is, “How do I compromise on what the development team will commit to doing, based on their estimates and the time constraints (of the project)?”

In order to determine a commitment, best practice suggests that estimates should be discussed with the development team first, so accurate numbers can be determined. There should also be discussions with the client in order to determine target dates. Then, based on these discussions, commitments can be determined with both the development team and the client.

Estimate, Target, and Commitment Example

To better understand estimates, targets, and commitments, examining an example is beneficial. A theoretical project may be estimated by the development team to need 1,000 hours of work, based on the requirements the client has expressed. This is an estimate. The client, however, may want the project finished within 600 hours. This is a target. After negotiating, the client and development team may adjust the scope of the project so it seems to include about 500 hours of work. This is a commitment. The project may take more or less time to finish than the commitment, and that actual time can be used as data for future estimates.

As you can see in this example, although both estimates and targets are non-negotiable once set, it is still possible to negotiate what you commit to for the scope of the project.

It is common for estimates to be automatically converted into commitments and targets, but this is a bad practice. Drawing on the previous example, if a project is estimated to take 1,000 hours

of work, often clients will automatically assume that this is a commitment of 1,000 hours. This often leads to the temptation of inflating estimates, also a bad practice.

It is therefore important that estimates, targets, and commitments are made clear and separate for both the client and the development team in their discussions. If this can be done, it is more likely that the project and its target dates can run as planned, without the scope becoming too large, as it ensures that target dates can actually be met. Again, drawing from the above example, understanding that 1,000 hours is an estimate, and the target is 600 hours helped create a commitment of 500 hours. This commitment helped narrow the scope of the project to stay within the target.

Module 2: Estimation

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

- Define and create a story point.
- Identify different ways of sizing a task.
- Explain what velocity is, how to calculate a velocity estimate, what factors influence velocity, and what makes velocity stable.
- Explain the concept of done, and how a story is only counted towards velocity if it is done.
- Define time boxing and its relation to Scrum.
- Define release.
- Explain and create a Gantt Chart.
- Explain and create a release plan.

Story Points

As explored in the previous module, good estimates are based on good reasoning. For example, the amount of time it took to develop and finish a task in the past could be the basis for an estimate on the amount of time it would take to complete a similar task in the future.

In an ideal world, past performance would be the only thing that influenced an estimate. In reality, however, developers often feel when they give an estimate that they should "pad" their work—in other words, some extra time should be added—in case their initial estimate is mistaken for a commitment and is wrong. In fact, it is very difficult to make a reliable estimate, especially if the task is far in the future (see the Cone of Uncertainty theory discussed in **Module 3** of this course).

Estimates are easily mistaken for commitments because they are both measured in units of time. It is easy to jump to the conclusion that the time suggested in an estimate can be simply converted into work hours. However, this isn't very accurate, and should be avoided.

Story points were developed as an alternate system for estimates. Instead of estimating how much time work will take to finish, **story points** are an assigned value that look at how long it will take to finish a piece of work in relation to other pieces of work. Story points are therefore relative measurements, and unitless, unlike time estimates. In time estimates, hours or days are the units used, whereas in story points, the points are not representative of any particular measurement. This is explained more in **How to Use Story Points**, below.

How To Use Story Points

Story points build on concepts explored in the course **Agile Requirements for Software Products**, such as user stories. User stories are short, simple, uniformly formatted descriptions of a feature told from the perspective of the person who desires the feature. User stories generally take on the format:

"As a '_____,' I want to '_____,' so that '_____.'

Another concept from the same course that is relevant to story points is the Scrum product backlog, which is a set or list of software features that the software product manager and development team plan to develop for a product over the course of a project.

In order to use story points, a relatively easy to estimate product requirement or user story should first be selected from the product backlog and assigned an integer value. This value is a story point. Using this basis, story points are then assigned to the rest of the user stories. For example, if a task twice as big as the base task is found, it will be assigned approximately twice the number of story points.

After this is done, you will be left with a list of user stories whose work estimates are all relative to each other, as they are all based on the same point system. This prevents accidentally committing to a time period, but instead just determines how large each user story actually is. This is much more in tune with the true purpose of estimates.

The total number of story points can be added up for all of the user stories in the project, giving some idea of how large the project will be.

It is extremely important, however, to remember **not** to treat story points like hours, or they offer no advantages. For example, it is easy for a developer to just "translate" hours into user points. In general, story points should not be very precise; instead they should stay relative. This means that large numbers such as 10, 50, or 100 are bad as base numbers because they are prone to be viewed as percentage points.

A good practice is to use story point values that are fixed. This helps avoid story points from simply becoming interpreted as hours.

TIP

A good way to make sure story points stay fixed and relative is to use the Fibonacci sequence. The Fibonacci sequence is a sequence where any individual number is the sum of the two numbers that precede it. The sequence runs like: 1, 1, 2, 3, 5, 8, 13, 21, etc.

If the Fibonacci sequence is used, story points should be restricted to only those numbers for estimates. This means that if an estimate falls between two numbers in the sequence, you must round up the nearest available number. For example, if you have a user story that is approximately twice as much work as one with the assigned value of 2, you should assign it the number 5, not 4.

Using the Fibonacci sequence for story points keeps estimates tidy and avoids making large estimates unnecessarily precise. It also accounts for uncertainty this way. As it is not a direct correlation with hours, using the Fibonacci sequence also prevents unintended commitments from being formed.

Advantages of Story Points

Aside from the face that using story points as estimates helps to not create unintended commitments, story points offer other advantages. Creating estimates using story points tend to lead to more accurate estimates. Humans in general tend to be bad at making exact estimates from scratch, but a method of estimating based on relative sizes makes it easier.

Imagine trying to estimate the size of a building. Your gut feeling might tell you that the building is sixty metres tall. Unless you actually measured the building, you would have no idea of how tall the building exactly is. So, your estimate could be very inaccurate. If you were to use a system similar to story points, however, you would compare that building to another, smaller building nearby. If you assigned the small building a value of 1, and you believe the taller building to be about three times taller, then it would have a value of 3. It is must easier to estimate this way than to just use a best guess.

Using story points for estimations is also potentially less stressful for developers, because nothing is exact. This means it is harder to "go over the estimate" like one could go over a time "deadline."

Limitations of Story Points

Although story points offer many advantages, they are not a widely accepted practice, even for those who use Agile methods and Scrum. This is because there are several notable limitations to story points too.

One way in which story points can be difficult is that they require a shift in thought from time to points. As a result, developers might be tempted to treat points like hours. This limitation could be resolved through increased exposure and practice with using story points.

Estimates are always difficult to make accurately, whether with hours or story points. Although using relative sizes generally results in more accurate estimates than using exact numbers, this is still open to mistakes.

As an example of how bad humans are at accurate estimation, consider the vertical-horizontal illusion. It can be difficult to tell if a horizontal line and a vertical line are the same length, even if both are visible at the same time.

Story points are open to limitations on large teams as well. Sometimes, story points are inflated to give the appearance of productivity. This practice is acceptable if it's applied consistently (e.g., if the inflation of the numbers is the same across the project). If inflation of numbers starts halfway through the project due to the introduction of new features or pressures, then story points will lose value and become less useful for estimation. The important thing about story points is consistency.

Velocity Estimates

Velocity is a measurement of the work that can be taken on and finished over the length of a sprint. Velocity is represented as the number of story points completed over a given period of

time. An example of a velocity could be 15 story points per sprint. Velocity is helpful because it serves as a means of:

- Estimating the effort the development team can achieve throughout the project
- Estimating effort that can be achieved on future work
- Indirectly tracking the team's productivity

In this course, a sprint is the same thing as an iteration – they are both short, iterative, and incremental time periods. In other words, both are ways to time-box work in small chunks. Time boxing is explored more in the next lesson.

Velocity can be measured as the number of user stories or work items delivered over a sprint. However, using user stories in velocity can be problematic, because not all user stories in a project are the same size. Velocity can also be measured through the number of story points completed over a sprint. Story points make for more accurate measurements.

A project should use either user stories or story points in velocity and consistently use the same measurement. Mixing the two would be confusing, as they are interpreted in different ways. In this course, we will focus on using story points over sprints as our productivity metric.

For story points to count towards velocity, they must be finished completely. Partially finished tasks do not count towards velocity. Using story points that are considered completely "done" in velocity estimates is very important.

How can it be determined if a user story is "done"? In Scrum and Agile methods, a user story is not done when the functionality has only been coded. A user story must also pass both unit tests and acceptance tests and then must be documented before it is considered "done." The additional completion of testing and documentation to coding is sometimes referred to as "done done" in the field.

Velocity-Driven Development

Velocity serves as the foundation of **velocity-driven development**. This kind of project development happens when the velocity of previous sprints is used as a basis for planning other sprints in the project. For example, if the three previous sprints of a project had a velocity of 15 points per sprint, then this can be used to plan the next sprint. User stories can be selected based on their story points until the total number of points projected to be finished in a sprint is 15 points or less. A development team knows better how much work they can get done if they can use past data to help predict what might be done in the next sprint.

Velocity-driven development requires a certain amount of stability in past velocity. At the beginning of a project, velocity may not yet be steady as there are still a certain amount of inaccuracies, and it may take a few sprints for it to even out. In order to help calculate stable velocities, it is important to be strict about counting story points only when they are done. For example, if two stories worth 3 points each are complete at the end of the sprint, and a third story worth 4 points still needs testing, this third story should not be counted. The sprint total should be 6, and not 10. This practice also helps to force or to encourage closure to stories rather than to constantly start new ones.

When estimating velocity, will find yourself in one of two places:

- The beginning of a project, when it is difficult to estimate velocity because no prior work has happened on which estimates can be based, or
- The middle of a project, when it is easier to make estimates based on prior work.

As in estimation, in cases where no prior work can be drawn on at the beginning of a project, it is possible to:

- Consult experts who may have worked on similar projects previously and who might be able to offer insights, or
- Draw on the team's own prior experience with similar projects to make estimates.

Previous data can be used to determine estimates through a variety of techniques, including but not limited to:

- Linear extrapolation (extending graphed data beyond the known data in a straight line based on how previous data has behaved),
- Averaging the velocity of the three previous sprints, and
- Using the minimum velocity of the three previous sprints.

Some Considerations in Using Velocity Estimates

There is a little bit of controversy over using velocity estimates that are important to acknowledge. Velocity estimates are not necessarily accurate, because velocity is variable. This reinforces the importance of only counting stories that are "done" by the end of the sprint. This helps counteract variability. It is also important to remember that velocity is derived from estimates, so it should still be seen as an estimate.

Velocity still offers many advantages, as it shares the advantages offered by story points. Velocity is also considered better for long-term tracking, especially as velocities become more consistent over the course of a project. And although velocity is not considered an ideal means of tracking the individual productivity of team member, it is considered a good tool for decision-making.

For more information on some of the considerations on using velocity estimates, see Michael Cohn's "Why I Prefer Commitment-Driven Planning" in the **Course Resources**.

Time Boxing

Time boxing is creating a restricted time period in which something is to be built, or for certain tasks to be finished in. The key to time boxing is the strict time limit—the event must stay within a predetermined timeframe.

Time boxing is a popular organizational method, as it allows software teams to compartmentalize work they have planned. Good time boxing also plans out the work at the beginning of a time period and leaves room for reflection on the progress of the project. Time boxing is particularly important to Scrum.

Scrum sprints, usually between two to four weeks long, a type of time box. They consist of a relatively small amount of work that is restricted to a relatively short development time. Scrum sprints include planning and review periods before and after work is done.

Time boxes are created when a time period is set in the project during which a certain amount of work will be planned, accomplished, and reviewed. Sprints and iterations are a kind of time box. Time boxing helps keep a project on track at a regular, sustainable pace because it divides the project into manageable chunks. The goals and milestones generated by time boxing helps development stay within scope and keep things on schedule. As work is planned over small doses of time, determining upcoming targets and their time periods becomes easier. It makes a project much more efficient than working with an uncertain schedule.

Developers should beware of putting too many tasks into a time box, or they cannot be finished. Velocity helps inform how much work can be completed in one time period, as previous work informs new estimates.

The organizational ability time boxing offers projects is sustainable. Further, it allows the team to learn from work at regular intervals, because of the built-in review period. It also allows the development team to see how work will be divided throughout the project.

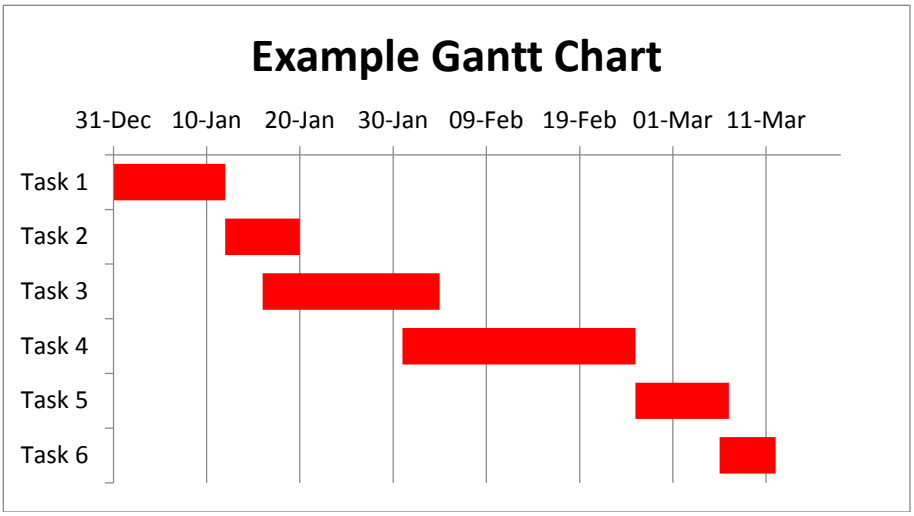
A project can be released in stages, at the end of each time box. This is because at the end of each time box, working software should be created, which can be released as a product. All work in a time box must be finished. Time boxing thus ensures that a tangible product is available at the end of a project, even if resources are lost along the way. This kind of progress also improves team morale.

Gantt Charts

Gantt charts are simple tools for visualizing project plans or schedules. They are a kind of cascading horizontal bar chart. Gantt charts can also be used to visualize task dependencies, the length of tasks, and deadlines.

Gantt charts plot tasks on the left-hand side of the chart against dates, which are displayed at the top of the chart. Each task is listed separately, one by one, and given its own horizontal bar. The first scheduled task is at the top of the chart, and progressive tasks are displayed underneath it. Each task's bar represents the duration of that task from its start date to its finish date. Once all tasks are laid out on a Gantt chart, you can see when each individual task is scheduled to begin and end.

Below, is a rough template of what a Gantt chart looks like. In a real example, the tasks would be given descriptive names, and there may be several more tasks.



Gantt charts are useful in both Agile and non-Agile projects. Because sequential tasks can be displayed in a Gantt chart, non-Agile processes, such as the Waterfall process, can make productive use of Gantt charts. In Agile environments, Gantt charts are also useful.

In order to use Gantt charts successfully with Agile methods:

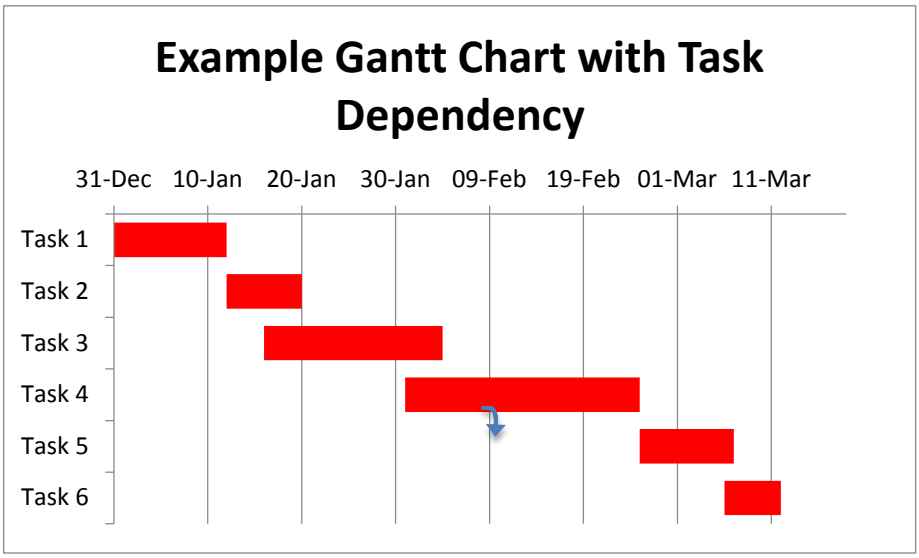
- Additional context should be provided to tasks (such as identifying if there are task dependencies)
- Parts of the chart can be sectioned off as sprints
- The charts must be used with an understanding of flexibility and adaptation to project needs (in other words, the chart could change as the project unfolds)

As detailed on the preceding page, Gantt charts allow the team to visualize when a project will be finished, when tasks are to be worked on, the length of those tasks, and potential project dependencies. Project dependencies and the different types of dependencies that can exist in projects are described more in the **Task Dependencies** lesson of the next module, but a brief outline of representing task dependencies in Gantt charts follows below.

In order to display task dependencies on a Gantt chart, first, a project backlog of tasks should be created and prioritized. Tasks that are dependent on the completion of other tasks should be identified on this list.

When a Gantt chart is produced based off of tasks and their time estimates, tasks that are dependent on others should be identified through the use of an arrow, going from a task towards the one that depends on it. In general, however, other planning techniques such as Critical Path Method (CPM) charts and Program Evaluation Review Technique (PERT) charts are favoured for demonstrating task dependencies. More information on how to visually represent task dependencies can be found in **Module 3: Planning** of this course.

In the example below, Task 5 is dependent on Task 4, as represented by the flowing arrow.



Gantt charts can be used on both the iteration level or the release level. On the iteration level, Gantt charts help provide insight by outlining tasks that should be completed within a few days in a current sprint. On the release level, Gantt charts represent user stories instead of tasks, so it can provide information on user stories over an upcoming series of sprints.

Many different tools exist to help create Gantt charts, including:

- Microsoft Excel
- Microsoft Powerpoint
- Gantt Project, an open-source Gantt chart creator (http://www.ganttproject.biz/)
- Many other subscription projects

Release Plans

Planning is a process that can occur on both the iteration level and the release level. This is known as two-level planning. As explained in the first module of this course, iteration planning involves dealing with tasks on a detailed level, while release planning deals with broader level user stories. Both levels of planning rely on time boxing in the form of sprints to organize either the tasks or user stories.

Iteration plans are generated at the beginning of a sprint in Scrum and Agile methods during the scheduled planning event. The iteration plan plans out the developer tasks to be completed during that particular sprint. Tasks tend to be actionable items, or items that a developer can complete. A user story consists of many tasks. Tasks are then self-assigned by the developers on the team before the work even begins. At the end of the sprint, working software should be created, which can then be shown to the client during the review event. Iteration plans are explored in more detail in the next module.

Release plans, on the other hand, are used to plan a few sprints at a time by determining which user stories from the project backlog should be completed and released in which sprint. User stories are chosen by priority. Generally, release planning occurs before any sprint is started—therefore, it occurs before iteration planning. User stories in release plans should be distributed logically and evenly across sprints; otherwise, they may be finished in a disorganized or haphazard manner, which may cause issue with task dependencies.

A prioritized project backlog, as described in the **Client Needs and Software Requirements** course is therefore key to release plans. User stories can be planned in a release plan from the top priority down:

1. "Must do" user stories (top priority)
2. "Should do" user stories (medium priority)
3. "Could do" user stories (low priority)

TIP

In addition to creating release plans that cover a few sprints, the product manager should also create a long-term "road map" plan, which covers the majority of the project and its user stories, if not the entire project. This helps give an idea of how the project will progress.

User stories are distributed across sprints pulling from these priorities, until each sprint is filled with an appropriate amount of work.

The number of user stories to assign during a sprint can be determined through many ways, including:

- Prior experience can help inform how many user stories can be finished in a sprint
- Data such as velocity estimates and story points can be used. Velocity can inform how many story points might be able to be finished in a sprint. User stories can be added until their story points add up to more than the projected limit.

Release planning should give an accurate projection of what the product will look like at the end of an upcoming series of sprints. A good release plan helps with client expectations and increases developer morale because tangible results are planned and expected.

Module 3: Planning

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

- Describe what makes a good estimate.
- Explain issues that may affect estimation, including the concept of the Cone of Uncertainty.
- Explain different approaches for creating estimates, such as bottom-up, analogy, and experts.
- Be able to calculate task estimation time using the estimation formula.
- Understand task dependencies and the different types of task dependencies: start-start, start-finish, finish-start, and finish-finish.
- Explain a critical path method (CPM) chart.
- Explain the concept of slack.
- Explain and create a PERT chart.
- Explain the differences between a CPM chart and a PERT chart.
- Explain and create an iteration plan.
- Understand that planning is self-assigned work done by developers; it is not assigned by clients.

Estimating Task Time

Cone of Uncertainty Principle

Before exploring the different approaches that can be used to estimate task time, it is important to understand the **Cone of Uncertainty Principle**, which informs the accuracy of an estimate.

The Cone of Uncertainty Principle works off of the fact that uncertainty is high at the beginning of a project and becomes lower as the project progresses, as discussed in the lesson **Uncertainty Space** in **Module 1: Introduction to Planning** of this course and in the course **Client Needs and Software Requirements**. The principle suggests that estimates made when uncertainty is high will be inaccurate. It also suggests conversely that as the project progresses, uncertainty is reduced because the requirements and development pace become clearer and, in turn, estimates will become more accurate.

The Cone of Uncertainty can be illustrated graphically.

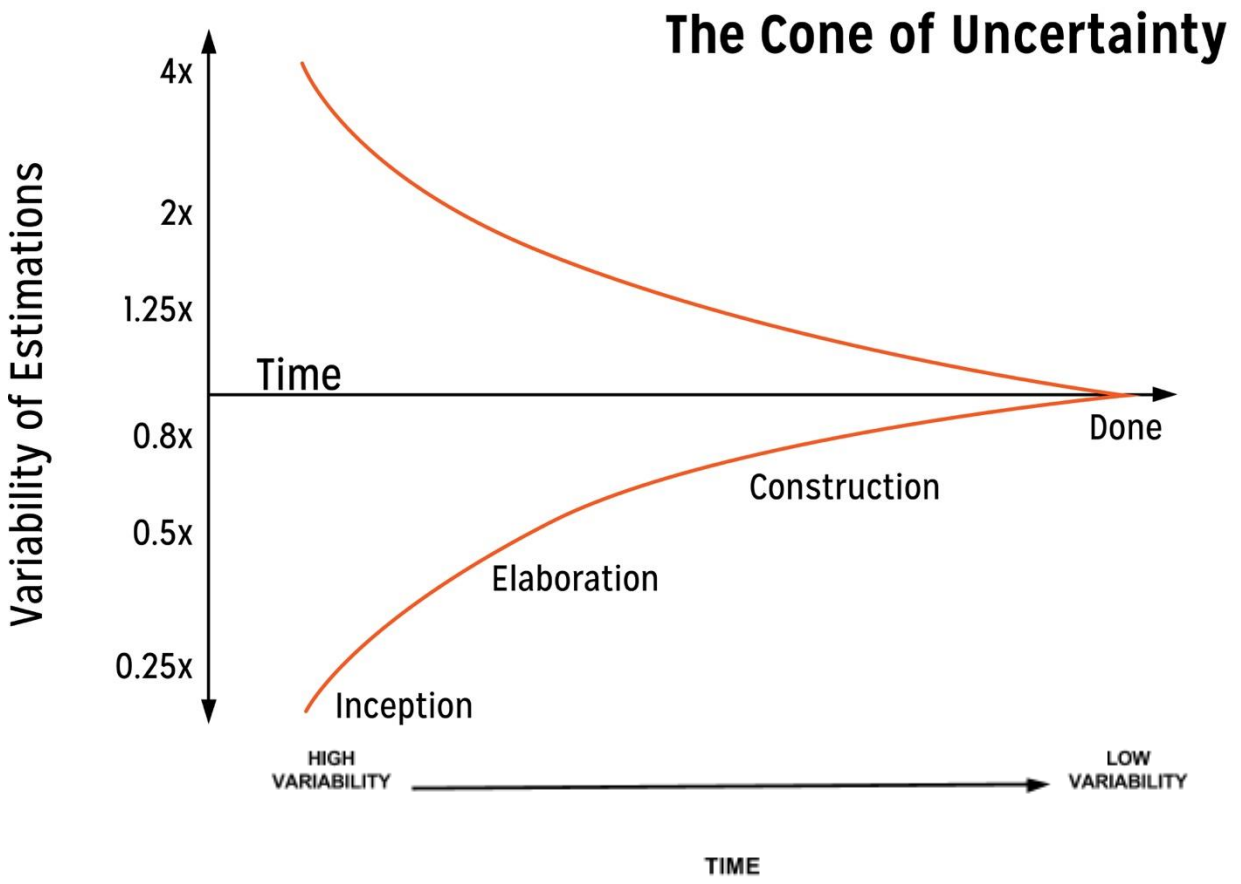
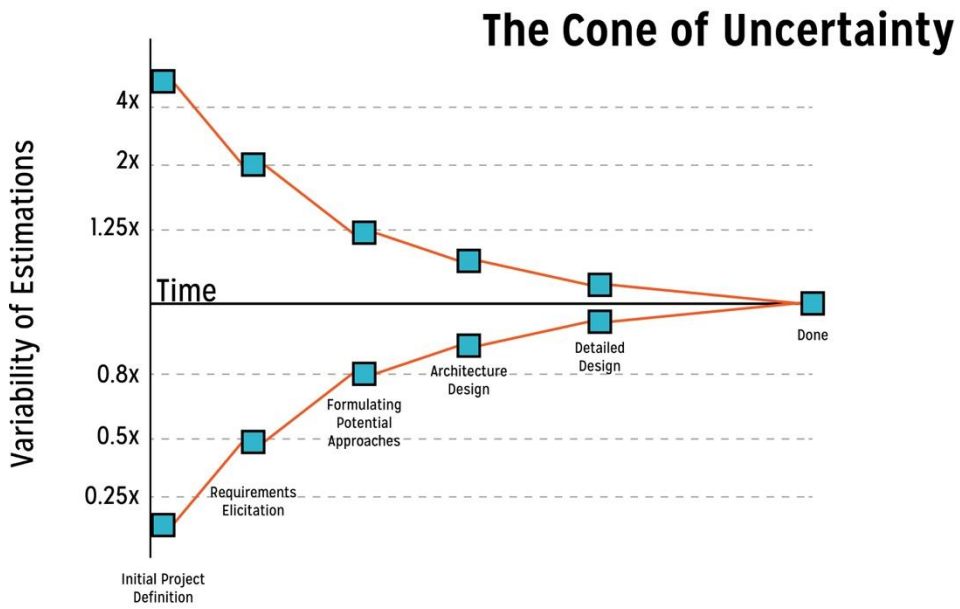


Image adapted from *Software Estimation: Demystifying the Black Art* (2006) by Steven C. McConnell. Used with Permission.

On this graph, the progression of the project over time is represented on the x-axis. Variability is represented on the y-axis. **Variability** refers to the factors or the extent by which an estimate could vary. The larger the factor, the greater the variability. Each number on the y-axis therefore represents a factor of variability, and the x represents the mathematical operation of multiplication.

The graph illustrates how at the beginning of a project, there is high variability and towards the end of the project, the variability significantly lowers. This creates a cone shape, which is where the Cone of Uncertainty principle gets its name.

The **Cone of Uncertainty** graph is illustrated below, with important points in the project included. This helps illustrate how variation changes at different points in the project over time.



The important project points are as follows:

- The **Initial Product Definition** – when the idea for the product is first formed. At this stage, variability is high (between the approximate factor of 3 and 0.375), and estimates will need a large range.
- The **Requirements Elicitation** activity – when the needs of the product and features have been identified. At this stage, variation is smaller (between approximately 2 and 0.5), and estimate ranges will be smaller too.
- **Formulating Potential Approaches** – when the methods used to do the project are chosen. At this stage, variation (between the approximate factor of 1.25 and 0.8) and estimate ranges become even smaller.
- The **Architecture Design** and **Detailed Design** stages, when information is known on how and what will be created in the end product. At these stages, the estimate becomes even more reliable as variation decreases (variation becomes closer and closer to a factor of 1).

Estimate ranges for the project can be made using the variability as illustrated in this graph. In order to do this, the estimate should be multiplied by its corresponding factor variability for where you are in the project. These will make the top and bottom numbers of the estimate range. For example, if a task is estimated to take 6 hours in the Requirements Elicitation Stage, this means it has a variability of 2 and 0.5 on the y-axis.

$$2 \times 6 = 12$$
$$0.5 \times 6 = 3$$

This gives an estimate range of 3 to 12 hours. As you can see, at this stage, there is still a great deal of uncertainty.

Using variability to create estimate ranges helps ensure a more accurate range. Estimates should also be continually revised as the project moves through the Cone.

Creating Time Estimates

There are many approaches to creating time estimates. All of these approaches, however, should be based on some sort of data. Remember also that estimates are non-negotiable figures.

This lesson will present several approaches that can be used to create time estimates:

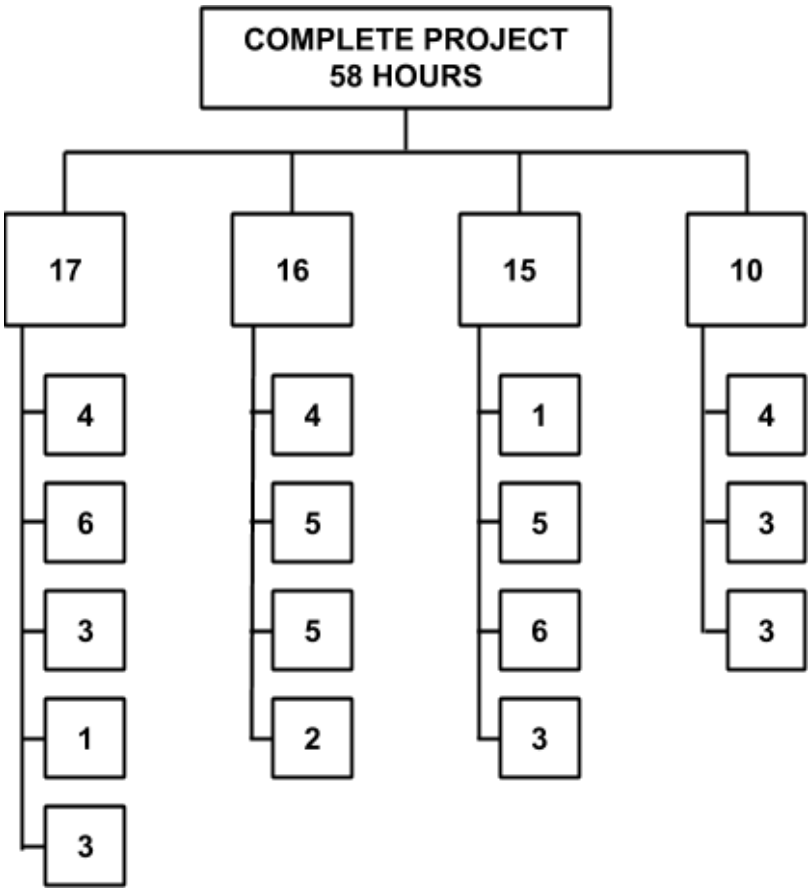
- Bottom-up
- Analogy
- Experts
- Using an 3-Point Estimate Formula

TIP: If it is necessary to create an estimate quickly, usually the bottom-up technique is the best to use, if enough information is available.

Bottom-Up

Bottom-up is a technique for creating a time estimate that involves breaking apart a project into small, manageable tasks, as done in a Work Breakdown Structure (WBS). Once the tasks have been broken up, they are much easier to estimate. If all the small tasks are given estimates, these estimates can then be added together to create an estimate for the full project.

Below is an example of a Work Breakdown Structure (WBS), where each of the tasks has been represented as hours. Once all the hours are added up, we can see how long each section of the structure is estimated to take and how long the entire project is estimated to take.



Note that even small tasks, which are easier to estimate, can be difficult to estimate accurately if there is not enough data to base that estimate on. This technique therefore works well when combined with another, especially in cases where there is not enough data to create an accurate estimate.

This technique assumes that none of the tasks in the Work Breakdown Structure (WBS) can be performed simultaneously, but instead that each task must be completed before another can start.

Analogy

Analogy is a technique for creating a time estimate that works by comparing the current task or project with a similar experience. If a similar task or project was accomplished in the past with similar scope and costs, then it can be a good basis for estimating the current project.

This technique should be used with some caution, as it can only work if the same team has worked on a similar project. This means the work, process, technologies, and team members have to be the same or similar to the current project to be a meaningful basis of data.

Experts

Time estimates can also be generated through **experts**. This technique works by converging estimates made from multiple estimators—generally developers who have prior experience with

the kind of work at hand. This technique is normally used in cases where the development team has limited experience with the work required for the project.

This technique can be costly, as it requires several estimators to look at the project. It can, however, also be very accurate.

> **TIP:** It is important to remember that all projects will have unique issues that arise over the course of the work, regardless of how similar it is to other projects. This means that any of the bottom-up, analogy, or experts techniques should be made with the understanding that these unique issues, which could not have been otherwise accounted for, may affect estimates.

The PERT 3 Point Estimate Formula

A useful tool used to estimate task time more accurately is the three-point estimation. This is known as the **PERT 3 Point Estimate Formula**, and it can create a range estimate. The formula is based on three values, which can be determined through prior experience or best-guesses:

- **Most Probable Time (Tm)** is the most likely estimate of the time it would take for the task to finish. This estimate operates under the assumption that the project or the task may encounter normal problems. Any of the bottom-up, analogy, or experts techniques can be used to generate this estimate.
- **Optimistic Time (To)** is the best-case scenario estimate. It assumes that everything will go right over the course of the project or the task and that the development team will work efficiently. It is therefore the least amount of time the task or project could be completed in.
- **Pessimistic Time (Tp)** is the worst-case scenario estimate. It assumes everything or almost everything that could go wrong over the course of the project or task will. It is therefore the maximum amount of time the task or project could be completed in.

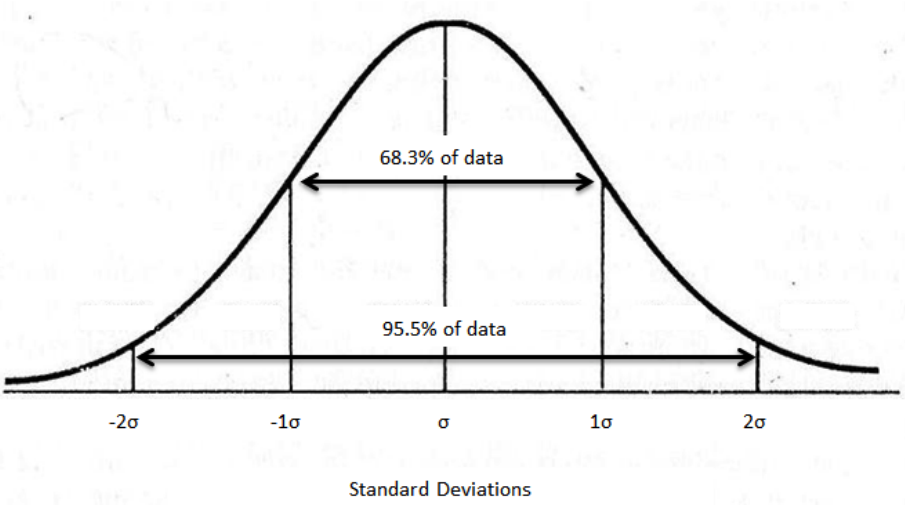
Using these three variables, the **Expected Time (Te)** can be calculated for the task. The Expected Time is a weighted average that can be calculated with this formula:

$$Te = \frac{(To+4Tm+Tp)}{6}$$

This formula places emphasis on the Most Probable Time that is multiplied by 4, giving it more weight in the formula. It is then averaged with the Optimistic Time and the Pessimistic Time.

The Expected Time is usually represented as a range. In order to create a range, we need some information about the probability that the estimate is correct. This is done by calculating the **deviations** of the Expected Time.

Deviations are represented by the Greek letter sigma (σ). Standard deviations represent how measurements can be spread out from the expected value. These are based on normal distribution plots or bell curves, which work based on percentages.



Standard Deviations

A normal distribution assumes that numbers are spread out in a certain way from standard deviation (in this case, the Expected Time), and that those values within one deviation (σ) of the standard deviation are 68.3% likely to be accurate. Within two deviations (2σ), there is a 95.5% chance the number is accurate. These numbers are drawn from bell curve and statistic theory.

To calculate deviation, the following formula is used:

$$\sigma = \frac{(Tp-To)}{6}$$

The smaller the difference between the Optimistic Time and the Pessimistic Time, the smaller the deviation will be. Once the deviation has been calculated, it can be subtracted from the Estimated Time (Te) to get the low number of the range. The deviation can also be added to the Estimated Time (Te) to get the high number of the range.

Although deviations are used to create time ranges in the course of planning, deviations can also be expressed using the ± (plus or minus) symbol after the Estimated Time. For example, if the Estimated Time is 12 hours, and the deviation is 2, this could be expressed as 10-14 hours, or in some cases, 12 hours ± 2.

If one deviation (σ) is used to create a range, we can assume that the range is 68.3% likely to be accurate. However, if it is more important for the project to have an accurate estimate than a smaller one, you can double the deviation (2σ) by multiplying it by 2, to reach 95.5% accuracy. More information on the estimate formula can be found in supplementary readings found in the **Course Resources**. A worksheet is also available to help with practice using the formula.

Sometimes a triangular distribution is also calculated using the same three variables, and the formula is:

$$Te = \frac{(To+Tm+Tp)}{3}$$

This formula places less emphasis on the most likely scenario, as it not weighted higher by being multiplied by 4. Using the variable in the example below, this equation gives us an estimate of 10 days to finish the task instead of 9.

Estimate Formula Example

Understanding the estimate formula can be easier through the use of an example. Using the hypothetical numbers for a task:

To = 5 days
Tm = 8 days
Tp = 17 days

We can calculate the answers to the estimate equation:

$$Te = \frac{(5+ (4x8) +17)}{6}$$

$$Te = \frac{(5+32+17)}{6}$$

$$Te = \frac{(54)}{6}$$

$$Te = 9 \text{ days}$$

In this example, standard deviation would be:

$$\sigma = \frac{(17-5)}{6}$$

$$\sigma = \frac{(12)}{6}$$

$$\sigma = 2$$

So, the estimation for this task would be 9 days ± 2 (a plus or minus deviation of 2 days). This would likely be expressed as an estimate of 7 to 11 days to finish the task.

> Sometimes the numbers created through using the estimate formula will not be whole numbers (in other words, they may have numbers after the decimal). In these cases, it is important to **round the estimate up** to the next highest number. For example, if the formula yields the number 6.9, it should be rounded up to 7. Similarly, if it yields the number 5.1, it should be rounded up to 6.

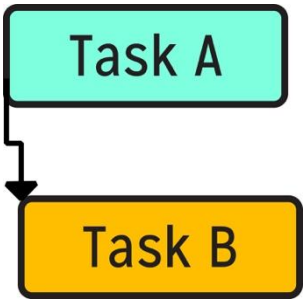
Task Dependencies

Task dependencies, as briefly discussed in the **Software Processes and Agile Practices** course, and the lesson **Introduction to Planning** at the beginning of this course, refer to those situations where one task is dependent on another task or set of tasks in some way.

Task dependencies are very important to planning and scheduling, because they shape the order in which tasks must be done.

There are four types of dependencies. They differ in how the second depends on the first:

- Start-start dependency
- Start-finish dependency
- Finish-start dependency
- Finish-finish dependency



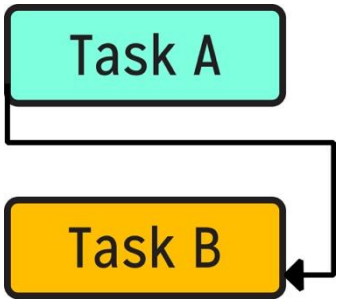
Start-Start Dependency

Start-start dependencies refer to those dependencies where the first task must start before the second task can start. When tasks finish in this dependency is not important.
Start-start dependencies are visually represented as follows, with an arrow going from the beginning of Task A to the beginning of Task B. The two tasks are shown more or less in parallel. Task B depends on Task A starting.

This method of representation can be used in any number of visual representations, including Gantt charts and PERT charts.

An example of a start-start dependency could be working on a project and creating a glossary. The project must be started before a glossary can start. These two tasks can be worked on simultaneously, however, and it does not matter which one is finished first.

Start-Finish Dependency



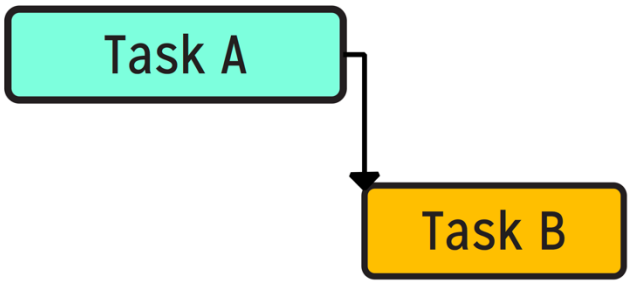
Start-finish dependencies refer to dependencies where the first task must begin before the second task can finish. These dependencies generally involve some kind of overlapping handover. In start-finish dependencies, the start of the second task is not important. In other words, the first task can start before or after the second task starts.

Start-finish dependencies are visually represented as follows, with an arrow going from the beginning of Task A towards the end of Task B. Both tasks are shown more or less in parallel. Task A must start before Task B finishes.

This method of representation can be used in any number of visual representations, including Gantt charts, and PERT charts.

An example of a start-finish dependency could be planning the next sprint during the current sprint being worked on. This allows the development team to begin working immediately upon starting the next sprint. Planning (Task B) must start before the current sprint (Task A) ends, so it is start-finish dependent on the current sprint.

Finish-Start Dependency



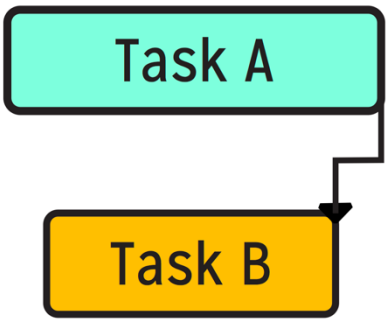
Finish-start dependencies refer to dependencies where the first task must finish before the second task can start. This is the most common type of dependency.

Finish-start dependencies are visually represented as follows, with an arrow going from the end of Task A to the beginning of Task B. Task B does not overlap with the Task A bar in any way. Task A must finish before Task B can begin.

This method of representation can be used in any number of visual representations, including Gantt charts and PERT charts.

An example of a finish-start dependency could be the need to finish designing a feature and knowing what it will do before a user or training manual can start to be written for a product.

Finish-Finish Dependency



Finish-finish dependencies occur when the first task must finish before the second task can finish.

Finish-finish dependencies are visually represented as follows, with an arrow going from the end of Task A to the end of Task B. The two tasks are parallel to each other. The arrow shows how task A must finish before Task B can finish.

This method of representation can be used in any number of visual representations, including Gantt charts and PERT charts.

An example of a finish-finish dependency could be finishing billing a customer for a product before finishing handing over the product, in order to avoid not being paid for the work.

Critical Path Method (CPM) Charts

Task dependencies are very important and, as mentioned in the previous lesson, they affect schedules and planning techniques. **Critical Path Method (CPM) charts** are an example of such a task planning method. A CPM chart is a visual way to organize task dependencies.

Creating a CPM Chart

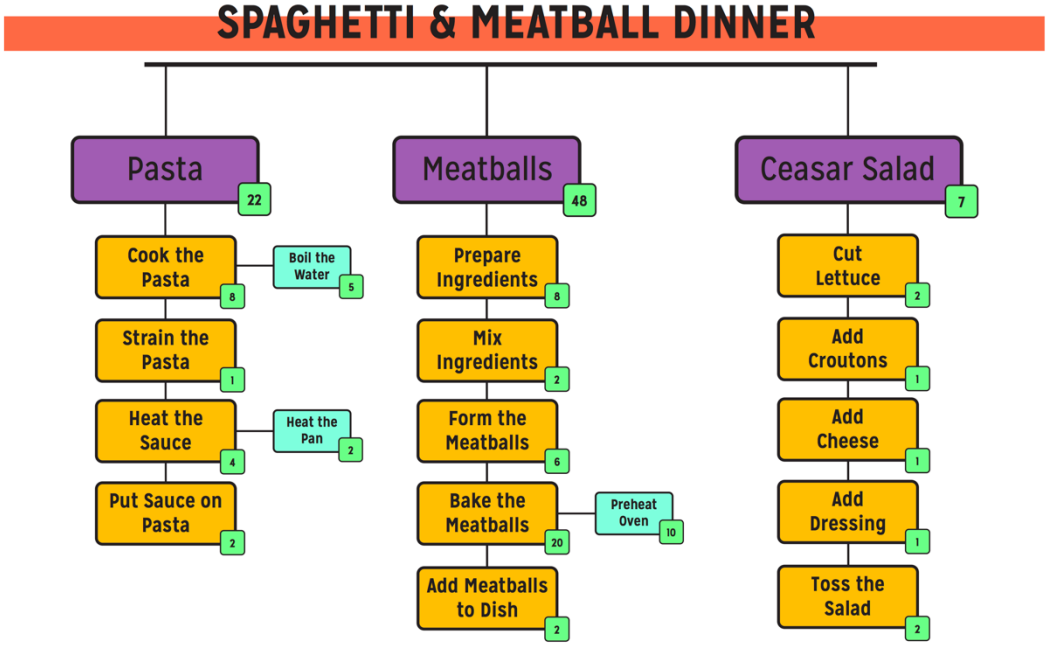
The steps involved in making a CPM chart are:

1. Make a list of tasks needed to finish the project and then create a Work Breakdown Structure (WBS) for those tasks (see the **Work Breakdown Structure** lesson in **Module 1: Introduction to Planning** of this course).

2. Add time estimates to each task.

3. Organize the tasks by grouping dependencies horizontally together. For example, a task that is finish-start dependent on another should be grouped next to the task it is dependent on. Arrows can then be used depending on which task dependency is at work to depict the dependency (see **Task Dependencies** lesson for how to graphically represent different types of task dependencies).

Here is an example of steps 1 through 3 for a project whose goal is to create a spaghetti and meatball dinner with a side salad.



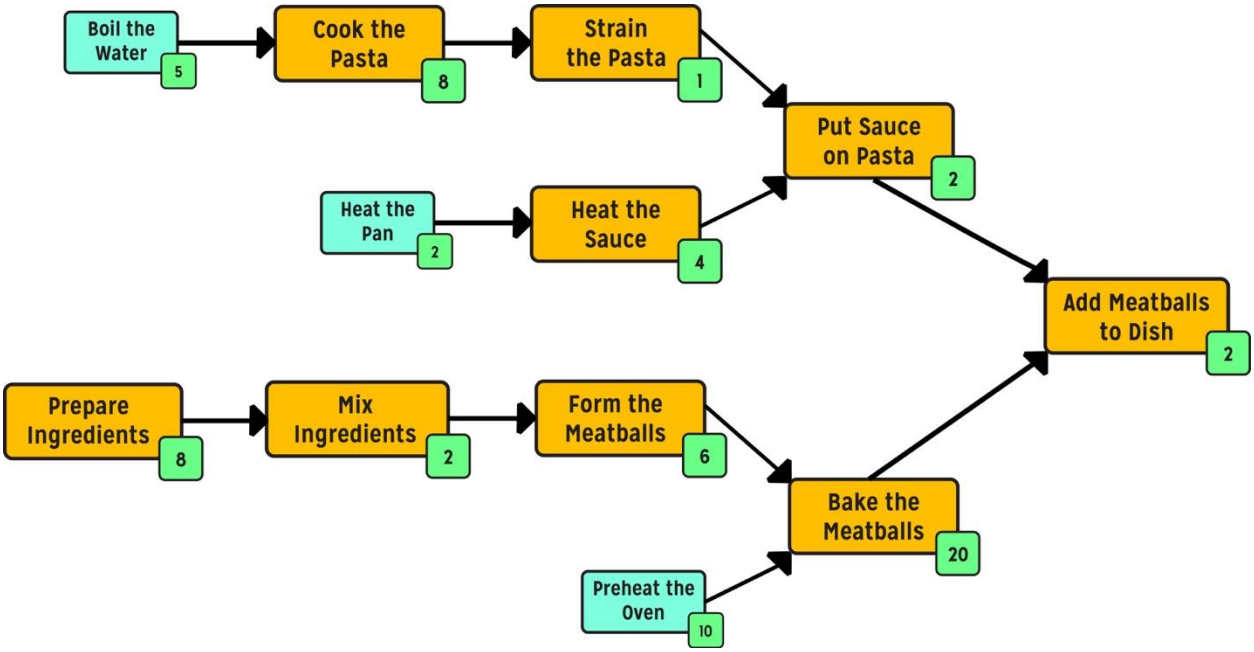
4. From these task groupings, paths can be created. A **path** is a sequence followed using arrows that you can take from one task to another (with dependencies shown as appropriate).

Paths that go from the beginning to the end of the chart are called **beginning-to-end paths**. Tasks on different beginning-to-end paths can be completed independently and, therefore, at the same time.

If two or more beginning-to-end paths meet, this is a **coordination point** for task dependencies—it means that the two or more tasks must be finished before the task they point to can begin. In other words, the tasks must be coordinated.

A CPM chart can have as many beginning-to-end paths and coordination points as needed to finish the project.

The example below shows a number of beginning-to-end paths generated from the Work Breakdown Structure (WBS) from steps 1 through 3 of the spaghetti and meatball dinner project. These paths illustrate the groupings under “Pasta” and “Meatballs” in the WBS.



There are four paths in this example. Following the arrows, these four paths are from:

- “Boil the Water” through to “Add Meatballs to Dish”
- “Heat the Pan” through to “Add Meatballs to Dish”
- “Prepare Ingredients” through to “Add Meatballs to Dish”
- “Pre-heat Oven” through to “Add Meatballs to Dish”

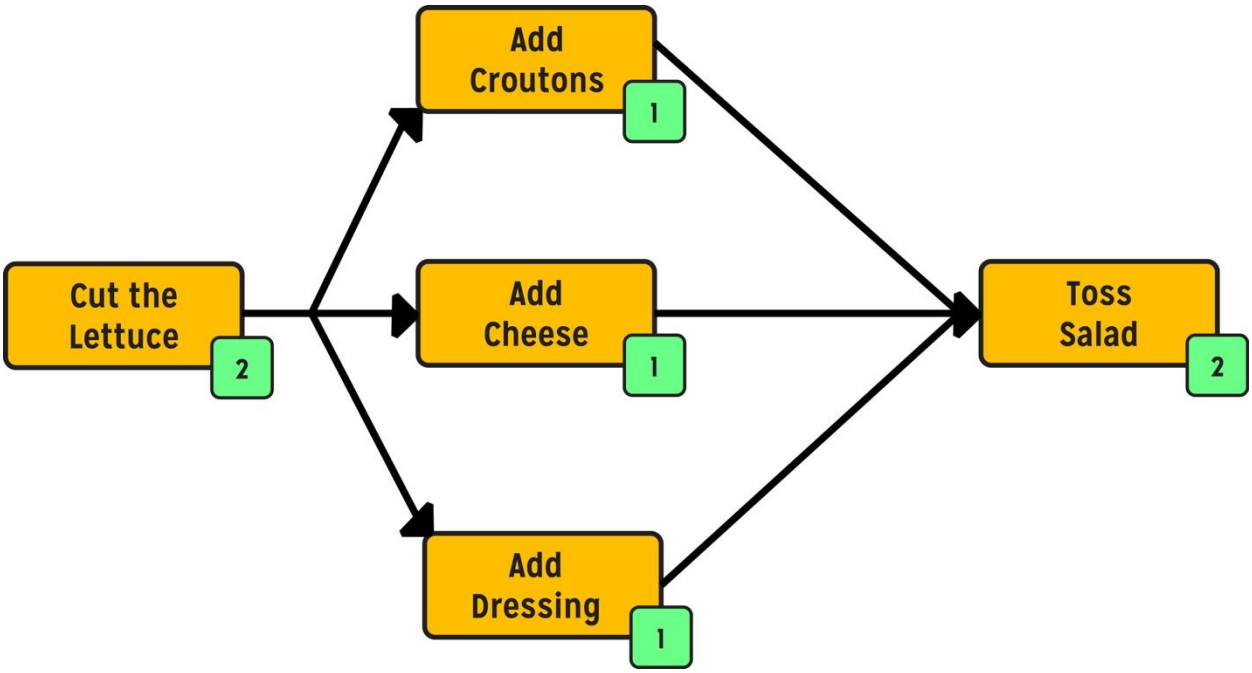
There are also three coordination points:

- When “Strain the Pasta” and “Heat the Sauce” coordinate before “Put Sauce on Pasta”
- When “Form the Meatballs” and “Pre-heat Oven” coordinate before “Bake the Meatballs”
- When “Put Sauce on Pasta” and “Bake the Meatballs” coordinate before “Add Meatballs to Dish”

If an overall time estimate is desired, it can be calculated by adding together the longest beginning-to-end path in a CPM chart. This is because all other beginning-to-end paths can happen at the same time. In the example above, “Pre-heat Oven” can occur at the same time as the path “Prepare Ingredients” → “Mix Ingredients” → “Form the Meatballs.”

A beginning-to-end path can also illustrate tasks that occur in **parallel**. Parallel tasks can occur in any order, as long as they are finished before the final task.

Drawing on the “Caesar Salad” grouping from the WBS example above, it is possible to illustrate parallel tasks.



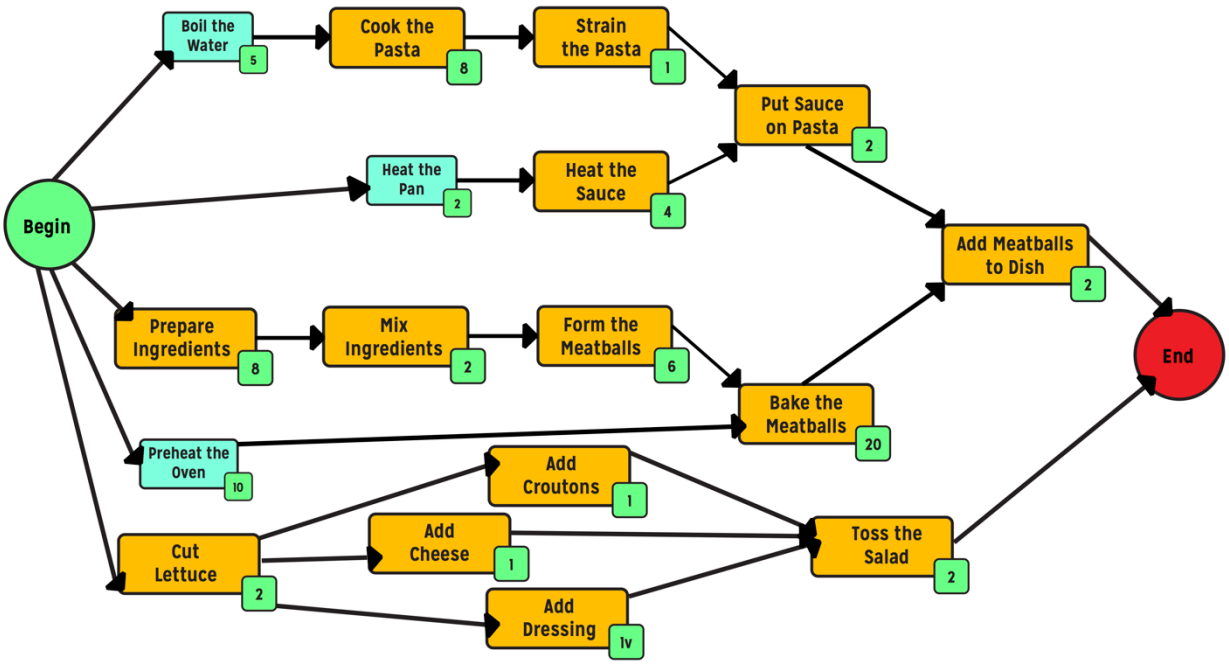
In this example, there are actually three beginning-to-end paths:

- “Add Croutons”
- “Add Cheese”
- “Add Dressing”

These can occur in any order, as long as they are done before the salad is tossed. They are in parallel.

5. Combine all of the beginning-to-end paths in order to create a full CPM chart.

Here is the full CPM chart for the spaghetti and meatball dinner project.



Critical Paths

CPM charts, once completed, are a useful tool for generating a schedule for the project and determining minimum time estimates. The major way this is done is through the use of critical paths.

A **critical path** is the longest duration path of tasks between logical points. In other words, a critical path is the beginning-to-end path with the greatest time estimate. It represents the minimum amount of time needed to complete a project, assuming there are enough resources that other beginning-to-end paths can be done at the same. The critical path in the example CPM chart above has been highlighted in blue.

In addition to providing the longest time estimate for the project, critical paths also provide information for longest and shortest time estimates for project tasks. Critical paths highlight the tasks that are **critical** for the project. These are the tasks along the longest duration path, which represent the minimum time for those tasks. But critical paths also highlight which tasks have **slack**—the ones on a different beginning-to-end path from the critical path. Because other beginning-to-end paths from the critical path have smaller time estimates, tasks on those paths

> Time estimates generated from critical paths in CPM charts may differ from estimates generated from Work Breakdown Structures (WBS). In a CPM chart, the critical path determines the minimum estimate and it is assumed (where resources are available) that other paths will be worked on at the same time. In a WBS estimate, however, each task is added as if one is done after another.

can be delayed, as long as the total estimate of the path never exceeds the critical path. This potential extra time for tasks is what gives the estimate slack.

Critical tasks in the critical path should be the first to be scheduled from a CPM chart, because tasks in other paths can be given extra time to finish (within the critical path time limit), without adding to the overall project.

Slack is useful to projects, as it provides a degree of flexibility to certain tasks, without necessarily affecting the target completion date.

Program Evaluation and Review Technique (PERT) Chart

Another task planning tool affected by task dependencies is the **Program Evaluation and Review Technique (PERT) chart**. Similar to CPM charts, PERT charts are visual representations of a project.

PERT charts use **nodes** and **edges** in their network or graphical depictions.

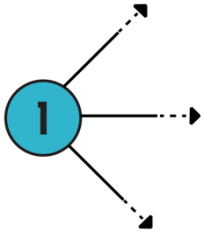
DID YOU KNOW?

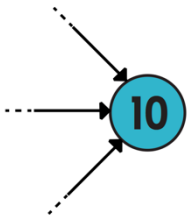
PERT charts were developed in the 1950s by the United States Navy. Their original purpose was to manage the Polaris submarine missile program!

Nodes, usually illustrated as circles, represent milestones. Milestones are not time based, so in a PERT chart, nodes tend to represent some kind of event such as the production of a work product or some other event occurring. In contrast, nodes in CPM charts represent tasks.

Edges, illustrated as lines that connect nodes, represent tasks. In a PERT chart, after lines are drawn, they are further denoted with tasks and time estimates. The lines are also given arrows, to illustrate task dependencies (see the **Task Dependencies** lesson in this module) and the order tasks must be completed in. A sequence of dependent tasks is therefore represented in a path that follows the direction of the arrows.

If multiple independent paths come out of a node (that is, if multiple lines come out of a node), it means that those tasks can be completed at the same time, assuming time and resources are available. They are in **parallel**.





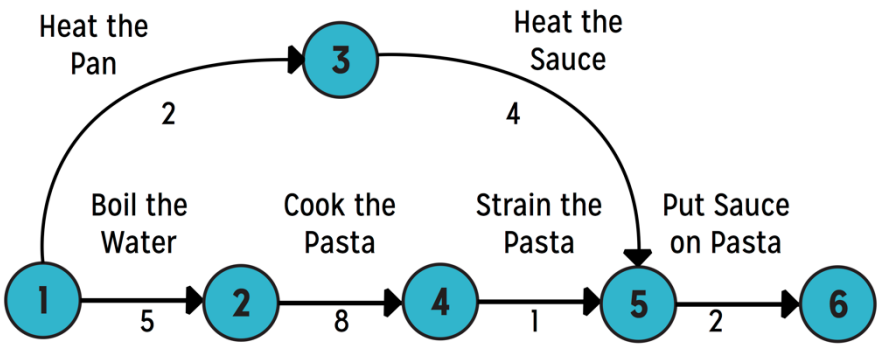
Alternately, if multiple paths lead towards a node (if multiple arrows point to a node), it means that these paths must **synchronize**. In other words, those paths must be completed around the same time, before the next path leading out of the node can start.

In general, PERT charts make dependencies easier to see.

Example of a PERT Chart

This example draws on the project discussed from the **Critical Path Method Charts (CPM Charts)** lesson preceding this one, with the hypothetical project whose goal is to create a spaghetti and meatball dinner with a side salad.

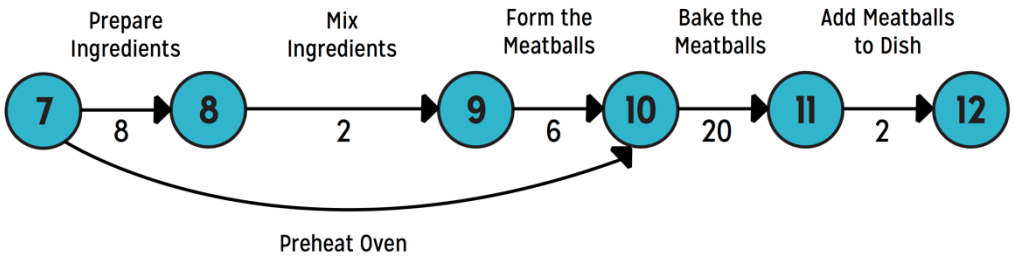
The paths for making pasta would look as follows in a PERT chart:



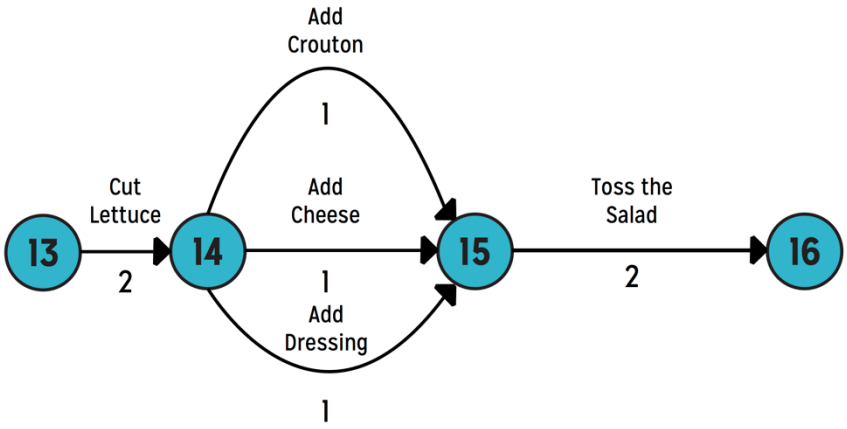
Edges, or the arrows, have been labelled with task and time. Task dependency is clear through the order of the arrows. Nodes, or milestones, have been labelled with numbers. Labelling nodes this way is effective, as numbers are a simple system, but it also allows for nodes to be easily referenced. Each node occurs after at least one event is completed.

Note in this section that there are two independent paths: 1 → 3 → 5 can be worked on during the path 1 → 2 → 4 → 5.

The paths for meatball-related tasks would look as follows in a PERT chart:

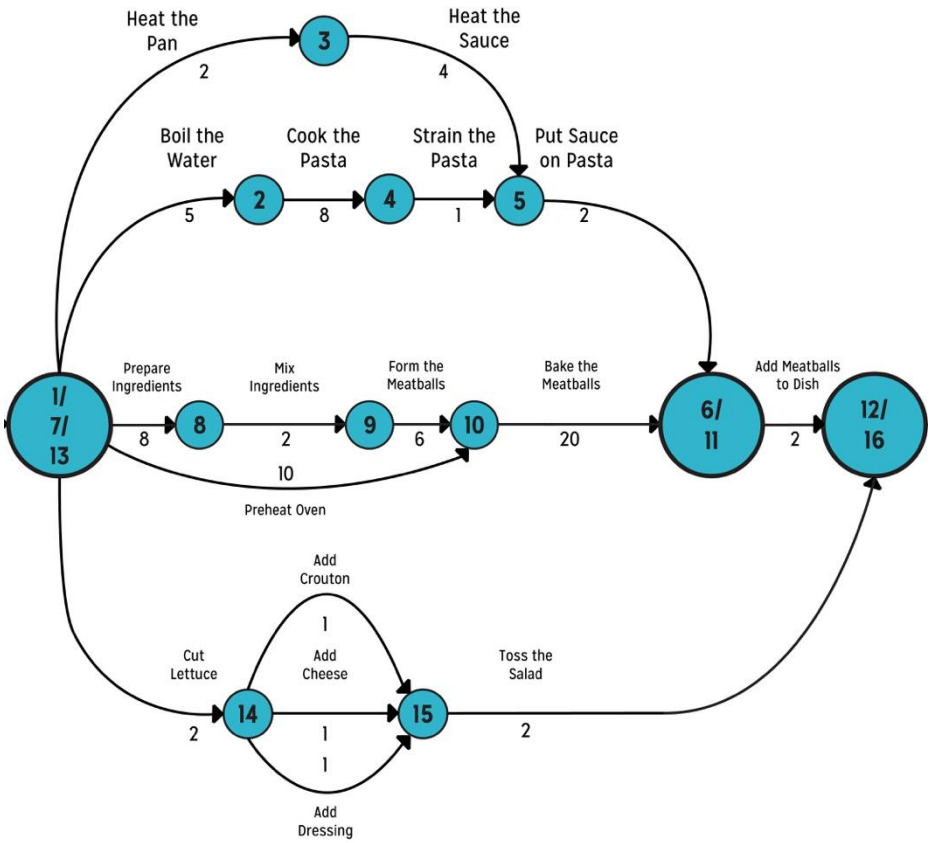


The paths for salad-related tasks would look as follows in a PERT chart:

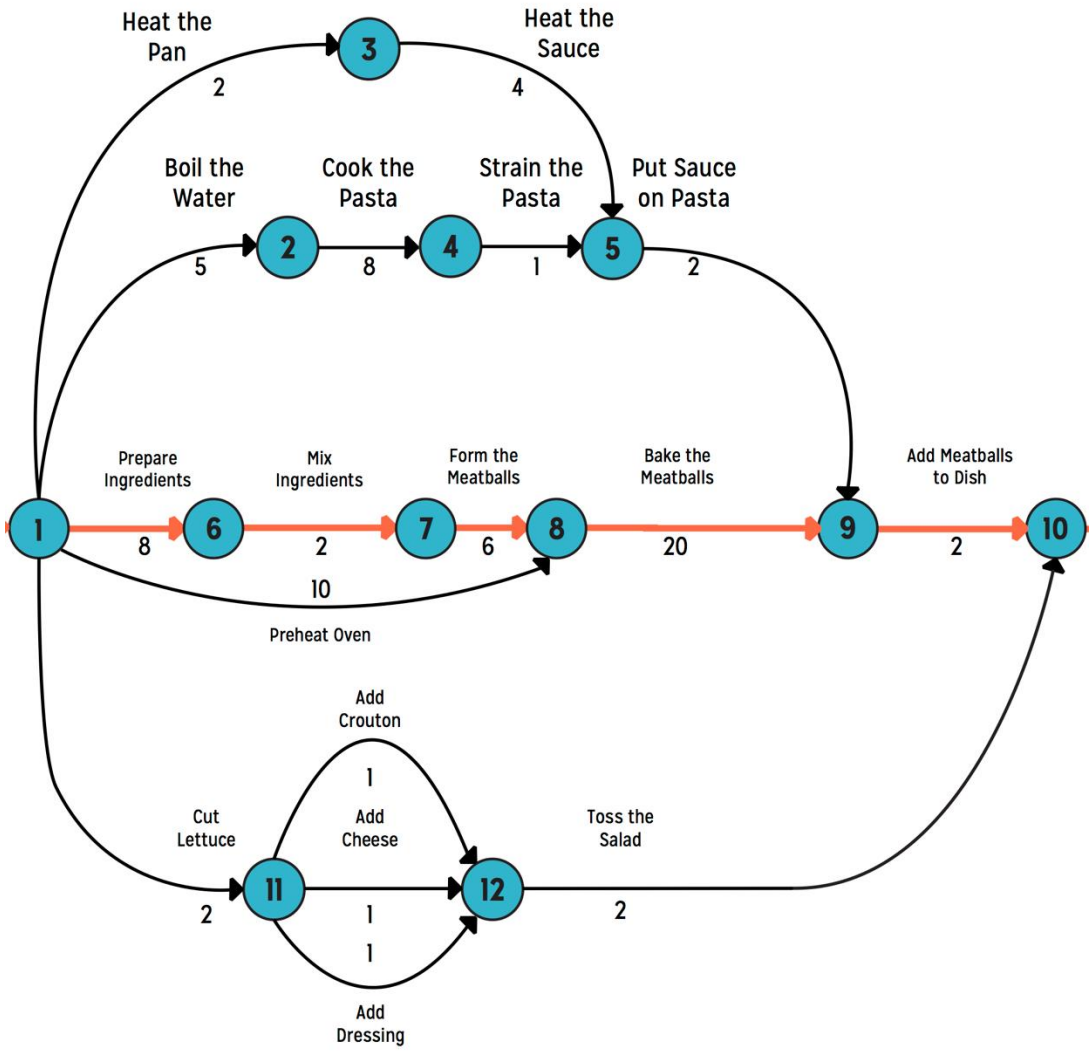


Note the parallel tasks here, of “Add Croutons,” “Add Cheese,” “Add Dressing.” These tasks can be done in any order, but all must be completed before the final task of “Toss the Salad” can begin.

Once all the paths have been combined, the complete PERT chart for this example would look like this:



As with building a CPM chart, in a PERT chart, each grouping of tasks should be charted first and then those sections combined. In a PERT chart, this may require cleaning up numbers that overlap for the same task in different sections. For example, node 6 above combined with node 11 in the final chart, and was simply labelled 11 (see below).



Nodes can be identified using any ordering scheme desired. However, using numbers increasing from left to right allows one to refer to paths in an increasing order.

Critical Paths

Critical paths are very important in PERT charts. As in CPM charts, the critical path is the path with the longest task time estimates. The critical path actually represents the fastest amount of time the project can be finished it, as all the tasks in the critical path must be finished, but it is assumed that shorter paths can be done within that same time. The critical path is therefore the minimum amount of time a project can take, assuming there are enough resources that other

paths can be completed at the same time. Let us examine the previous example to illustrate this.

The critical path from preparing ingredients through to adding meatballs to the dish (1→2→6→8→1→12) takes a total of 38 minutes. This is the longest task time estimates, as other paths take shorter to finish. 1→5→9→11→12 takes 10 minutes, 1→4→7→9→11→12 takes 18 minutes, 1→8→11→12 takes 32 minutes, and 1→3→10→12 takes 5 minutes. Even though the critical path takes the longest, all of those steps must be finished, so the overall project will take at minimum 38 minutes to finish.

Also as in CPM charts, paths other than the critical path have slack, as they have some room for estimates to change (as long as the path still takes less time than the critical path), without affecting the overall project estimate.

Again, drawing from the above example, the paths 1→5→9→11→12, 1→4→7→9→11→12, 1→8→11→12, and 1→3→10→12, which are all shorter than the critical path, can take some extra time to finish, if necessary, as long as they do not take more time than a total of 38 minutes for the path (or the overall project estimate will change). This flexibility in timing means they have slack.

Any task on the critical path that is extended in terms of time will result in an increase to the minimum time for the project, because there is no slack on the critical path.

In the above example, the critical path of 1→2→6→8→1→12 takes 38 minutes, so the minimum time the project will take is 38 minutes. However, if the original estimate of 6 minutes to finish task 6→8 is extended to 10 minutes, then the critical path estimate will also change, from 38 minutes to 42 minutes. The minimum time for the project has changed to a total of 42 minutes, as the critical path has no slack.

In a PERT chart, critical paths must be horizontal. In the example above for the project of making a spaghetti and meatball dish with a side salad, the critical path has been outlined in red to be easily identified. As critical paths are always horizontal in PERT charts, they are always easily identified, even if information such as time estimates may not be available. All other paths in the chart must be drawn either above or below the critical path.

CPM Chart or PERT Chart?

In what circumstances is it better to use a CPM chart or a PERT chart? Both can be used to demonstrate project dependencies and to generate time estimates. In other words, both work well for task planning.

In some cases, either chart can be used with success, and it is then a matter of preference for the development team and the project manager. In circumstances where the project may be more task focused, however, the CPM chart is favourable. In CPM charts, additional information can be added to tasks, such as costs. In circumstances where the project may be more event or milestone based, in contrast, the PERT chart is favourable. PERT charts show what tasks need to be completed prior to an event or milestone.

Iteration Plans

As explained in the **Release Plans** lesson in **Module 2: Estimation** of this course, planning occurs in both an iteration and a release level. Iteration planning involves dealing with tasks on a detailed level, while release planning deals with broader level user stories.

Release plans therefore plan a few sprints at a time by selecting user stories from the project backlog to be completed and released in a sprint. User stories are chosen for sprints by their priority level. Release planning occurs before iteration planning because it happens before sprints begin.

Iteration plans, on the other hand, are used to manage tasks within a sprint itself. They are based on velocities, task estimates, and available time. Iteration plans ensure that a development team has not over- or under-committed in each iteration. This keeps developers focused on their tasks, makes certain the right tasks are finished, and that the project is on schedule. Iteration plans are key for effective project development and maintaining a sustainable project pace.

Most Agile methodologies, including Scrum and Extreme programming (XP), make use of iteration plans. This lesson uses Scrum terminology such as sprints, product owner, and scrum master to explain iteration plans. For more information on these terms, see the **Software Processes and Agile Practices** course. The terms iteration and sprints are used synonymously in this lesson as well.

Iteration plans are created in Sprint Planning meetings. Sprint Planning meetings have fixed lengths, set at the beginning of the meeting, and are therefore time-boxed meetings (see the **Time Boxing** lesson in **Module 2: Estimation** of this course). As a time-boxed meeting, the length of Sprint Planning meetings must be strictly adhered to so that the meetings themselves stay on topic and that the iteration plans are not over-planned.

TIP: Four hours is considered a good amount of time for a Sprint Planning meeting.

Creating an Iteration Plan

The steps involved in creating an Iteration Plan during a Sprint Planning meeting are:

1. Create a Sprint Goal at the beginning of the Spring Planning meeting, if one does not already exist for this sprint within the release plan.

   The Sprint Goal is the general focus for the sprint being planned. Outlining the sprint goal helps keep the development team focused on relevant tasks throughout the sprint.

2. Report the project velocity from the last sprint. This is an important step, as the previous project velocity will determine the commitments made for the sprint being planned.

   In cases where the last sprint was abnormal, the lowest velocity from previous sprints should be used.

3. Determine all potential user stories to be completed within the sprint, and break them down into developer tasks.

User stories selected for completion within the sprint will be informed by the release plan. However, user stories from the previous sprint that were not considered “done” (for example, if a story did not pass acceptance tests) should also be included in the iteration plan for this sprint.

The number of user stories should be chosen based on the previous sprint’s velocity. This means that there should only be enough stories within the sprint being planned, so the sum of their story point estimates is less than or equal to the previous velocity. For example, if the previous sprint finished 15 story points, then user stories should be selected for this sprint until they equal 15 story points.

User stories are also generally chosen based on priority—the most important ones should be finished first. Any user stories not chosen must wait for a future sprint.

Once user stories have been chosen for the sprint, they must then be broken down into developer tasks by the development team. Clients are not involved with this task breakdown. A developer task should only take a day or two to complete.

After tasks have been broken down, they should be expressed using techniques such as a Work Breakdown Structure (WBS), a Critical Path Method (CPM) chart, a Program Evaluation and Review Technique (PERT) chart, a Kanban Board, etc.

Kanban Boards, first discussed in the course Software Processes & Agile Practices of this specialization, are a scheduling system developed by Toyota. This system organizes production in a logical chain.

4. Developers must create an estimate for each task.

   Any task estimates must be agreed upon by the entire development team. In Scrum, tasks are considered cross-functional. This means that any developer should be able to take on any task. However, every developer has a unique pace, so their estimates may differ for the same task. Task estimates are therefore an agreed average across the team. Even tasks that require specialized skills, and therefore needs to be done by a specific person on the development team, must have an estimate that the entire team agrees upon.

   Estimates are given in hours, half-days, or days. Any task with a smaller estimate than these measures should be grouped with other tasks. Likewise, any task estimated to take longer than three days should be broken down further.
5. Revisit chosen user stories for the sprint after estimates have been made. Once tasks have estimates, it is easier to ensure they can be realistically done within the sprint.

Upon revisiting user stories, if it is found that they cannot all be finished, then the commitment for the sprint needs to be adjusted. User stories and their associated tasks should be removed from the iteration plan until the total estimated time matches the available time of the developers.

Estimates made at the iteration level override estimates made at the release level. Iteration level estimates are more likely to be accurate, because iteration plans are more specific. If iteration level estimates change release level estimates, the release plan should be changed accordingly. Typically, release plans often only look at a few sprints, and are regularly adjusted.

On the other hand, if it is found that the selected user stories can all be finished within the sprint with time left, then more user stories can be added, as long as the total time doesn't go over the time of the sprint.

6. The development team must sign up for tasks.

   In iteration plans, tasks are self-assigned. This means that managers do not assign tasks to members of the development team, but rather, developers choose to work on what they find interesting. This generally results in more satisfied workers, and better software.

   Developers should choose tasks based on their available time, so other commitments or projects the developer is involved in can be accounted for.

Iteration plans allow both the development team and the client to know what to expect from a sprint and what the product at the end of the sprint will look like. The team and the client should be on the same page.

Sample Iteration Plan

SPRINT GOAL: In this sprint, the team is creating the user profile elements. At the completion of this sprint, users should be able to create accounts, log in, and edit their profile.

LAST SPRINT: In the last sprint, we completed 33 story points. It is based on three developers working approximately 40 hours per week for the two-week sprint.

NAME/ DESCRIPTION	PRIORITY	SIZE ESTIMATE (POINTS)	ASSIGNED TO	EFFORT ESTIMATE (DAYS)
CREATE DATABASE FOR USER ACCOUNTS	1	2	DAVID	3
DESIGN LOGIN PAGE	1	2	DAVID	1
DESIGN ACCOUNT CREATION PAGE	1	1	CLAIRE	1
DESIGN USER PROFILE PAGE	1	1	ALEX	1
PROGRAM DATABASE FUNCTIONALITY TO LOGIN PAGE	3	1	DAVID	2
PROGRAM DATABASE FUNCTIONALITY TO ACCOUNT CREATION PAGE	3	1	CLAIRE	2
PROGRAM LOGIN PAGE	2	1	DAVID	2
PROGRAM ACCOUNT CREATION PAGE	2	1	CLAIRE	2
PROGRAM PROFILE PAGE	2	2	ALEX	2
WRITE TESTS FOR ACCOUNT LOGIN	2	1	DAVID	2
WRITE TESTS FOR ACCOUNT	2	1	CLAIRE	2

CREATION				
WRITE TESTS FOR USER CHANGING PROFILE PICTURE	2	1	ALEX	2
WRITE TESTS FOR USER CHANGING INFORMATION	2	1	ALEX	2
RUN ACCEPTANCE TEST ON LOGIN PAGE	4	2	DAVID	2
RUN ACCEPTANCE TEST ON ACCOUNT CREATION PAGE	4	2	CLAIRE	2
RUN ACCEPTANCE TEST ON PROFILE PAGE	4	2	ALEX	2
WRITE USER MANUAL DESCRIPTION FOR LOGIN PAGE	4	1	DAVID	1
WRITE USER MANUAL DESCRIPTION FOR CREATING AN ACCOUNT	4	1	CLAIRE	1
WRITE USER MANUAL DESCRIPTION FOR CHANGING PROFILE	4	1	ALEX	1
TOTAL		25		33

Module 4: Risks

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

- Identify, describe, and suggest means of addressing common management anti-patterns, including: analysis paralysis, cart before the horse, groupthink, silos, vendor lock-in, overengineering, gold-plating, viewgraph engineering, fire drill and heroics, death march, micromanagement, seagull management, email as the primary means of communication, loose cannon, and intellectual violence.
- Understand the difference between group anti-patterns and individual anti-patterns.
- Define the term risk.
- Identify and define types of risks including: scope risk, technology risk, customer and stakeholder risks, and personnel risks.
- Define the terms impact, likelihood, and value.
- Know how to create and use an impact vs. likelihood matrix and a risk-value matrix to prioritize risks and features.
- Know how to address different rankings of problems along the impact vs. likelihood matrix and risk-value matrix.
- Define the terms risk management plan, indicator, action.
- Create a risk management plan.
- Understand that some risks are unforeseeable and must be dealt with as they come.

This module focuses on risks and risk planning. Risks can make even a well-planned Agile project go off track, so it is important to plan for and mitigate risks as much as possible.

Anti-Patterns

Some risks are so common, they create predictable patterns in projects. These risks are known as **anti-patterns**. Anti-patterns are defined as commonly occurring solutions or situations in a project that cause negative consequences such as project failure. They are the “classics” of risk management and are not project-specific risks.

The best way to avoid anti-patterns is to understand them, so they can be identified and addressed.

Many different types of anti-patterns exist, including:

- Anti-patterns from writing code
- Software architecture anti-patterns
- Management anti-patterns

This module focuses on management anti-patterns. This type of anti-pattern can have many different names, but the concept remains the same: management anti-patterns are ways in which projects fail due to the behaviour of individuals or groups of people.

Only a few anti-patterns are listed in this lesson. For a full list, and accompanying links, see **Course Resources**.

Group Anti-Patterns

Group anti-patterns refer to risks involving groups of people. The following group anti-patterns are covered in this module:

- Analysis paralysis
- Cart before the horse
- Groupthink
- Silos
- Vendor lock-in
- Over-engineering
- Gold plating
- Viewgraph engineering
- Fire drill and heroics
- Death march

Analysis Paralysis

Analysis Paralysis is a group anti-pattern where the development team gets stuck or stalled in some phase of the project, usually from analyzing requirements, leading to paralysis of the project.

This commonly happens in the specification phase of the project. It is during this phase that projects are likely delayed because clients and/or product managers spend a lot of time analyzing requirements and cannot decide on a direction for the project until the analysis is perfected.

Although wanting to avoid choosing the wrong path is admirable, spending long periods of time waiting for information is not in the spirit of Agile methodology. Instead, progress should be at a constant pace in a project—so as to avoid unhappy developers.

To avoid this anti-pattern, a good strategy is run a project with incremental releases, as embraced in Agile. With incremental releases, not everything needs to be known upfront because it is understood the product must be flexible and will change and refine over time.

In the course **Software Processes and Agile Practices,** you learned about the Agile practice of Lean Software Design. One of the seven principles of Lean is:

"Deliver as fast as possible"

This principle means the product should evolve through a series of rapid iterations. This is advantageous not only for avoiding analysis paralysis, but because it allows the client to give feedback and it places focus on core product features.

Cart Before the Horse

Cart before the horse is also a group anti-pattern. The name of this anti-pattern comes from the English idiom, "putting the cart before the horse." This phrase means that something was put in a counter-productive order, just like putting a cart in front a horse is less productive than having the horse pull a cart. In software product management, cart before the horse similarly means putting too much emphasis on a part of the project that should be done later.

In the course **Software Processes and Agile Practices,** you learned about the Agile practice of Lean Software Design. Lean Software Design solves the issue of groupthink by allowing individuals to come up with solutions to designs and other problems on paper, without talking. This is what generating solutions "silently" means. Then, in a group, each person's idea is discussed in turn, and a final solution combining the entire group's designs is created.

This anti-pattern might be easier understood through an example. If less time and resources are spent developing a critical feature of a product than on a less critical one, or a feature that is not required at the moment, this is an example of cart before the horse. This frequently happens when programmers assign themselves work that doesn't directly impact current features, in order to get a "head start" on future work. However, the danger of this process is that a team can find themselves with a lot of poorly developed features of varying importance.

To avoid this anti-pattern, it is important for the development team to understand development priorities and stay focused on those priorities. In other words, the team should clearly understand what work is to be done now and what work is to be done later.

Groupthink

Groupthink is a group anti-pattern that refers to how people tend to follow the general opinions of a group, regardless of whether or not their own individual opinions are different. The term comes from the social sciences. Groupthink can lead to poor decisions and judgments in project development.

The most common example of groupthink is in group discussion on projects, when ideas tend to come from only one or two people in a meeting, while most other members are quiet and go along with suggestions. This can lead to a poor end product, because often other or better options are not discussed.

To avoid groupthink, there are a couple of good measures to take. First, it is important to remember that conflict can be a good thing in group discussions, if done constructively. Creating an open atmosphere of discussion, where different perspectives are encouraged, is very helpful in combating groupthink.

In the course **Software Processes and Agile Practices,** you learned about the Agile practice of Lean Software Design. One of the seven principles of Lean is:

"Eliminate waste"

This principle means that it is important for the development team to be careful of being "busy" instead of being productive. A busy team can get distracted by "extra" features, which can ultimately interfere with the functionality of the end product, rather than focus on developing the core requirements. This is wasteful and should be avoided.

Another means of addressing groupthink is to generate solutions to problems silently, as in Lean Software Design. This means allowing a team to break up into smaller groups or individuals and encouraging these smaller groups to come up with different designs and alternatives. These suggestions are then combined together. This approach has the benefit of not putting people on the spot.

For a video demonstration of groupthink in action, see the **Course Resources.**

Silos

Silos are group anti-patterns that occur in the extreme opposite of groupthink. A team separated in smaller groups can be a good strategy, particularly if specialized groups create

more effective work, but this then runs the risk of teams losing touch with each other and limiting communication. This is called working in silos.

In essence, silos are created when a lack of communication among groups on the development team occurs, which lead to a loss of unified focus and the creation of counterproductive strategies and product features. Features developed in silos are developed in a vacuum, so the work of one team could be different or incompatible with that of another team, and bringing the work of two different teams together could thus prove difficult.

To address this problem, it is important to encourage an open and positive atmosphere where team communication and collaboration is favoured. This also encourages the development team to be less apathetic. Alternately, re-examining management structure could also be beneficial. If there is flat management where developers are able to speak directly to other developers instead of going through managers or the managers of other teams, communication can flow easier and bureaucracy can be avoided. In both cases, face-to-face communication should be encouraged.

Vendor Lock-in

Vendor lock-in is a group anti-pattern that occurs when a development team creates a product that relies heavily on a single technology solution or vendor. Vendor lock-in is different from choosing a technology because it is the best option, but rather it generally involves a lack of flexibility in moving to another technology without substantial costs.

The heavy reliance on a single technology as a project moves forward is problematic if the technology cannot cover what is needed, becomes outdated, or cannot adapt to change. In extreme cases, staff can even leave an organization over vendor lock-in.

To avoid this risk, it is important to do research before committing to a technology or a solution, regardless of recommendations made to the project.

Over-engineering

Over-engineering is a group anti-pattern that relates to how the development team creates the product itself. It occurs when a product is made more complex than necessary, such as when extra or needless features are added to a product, making it confusing for the majority of users. Over-engineering can happen on both the user interface (UI) of a product or in its internal processes.

There are many examples of over-engineering, such as digital cameras with too much features that must be set before functionality is enabled, music players with too many options, cars that can achieve speeds that it will never run at, or text editors with too many save options.

To avoid over-engineering, it is important that the development team clearly understands the needs of a product (what the product needs to be able to do) versus the wants of a product (what would be nice if the product could do but is not necessary for success). In the course **Client Needs and Software Requirements**, determining “needs” and “wants” was discussed at length, including the importance of prioritizing requirements and ensuring a project is feasible.

Gold Plating

Gold plating is a group anti-pattern that relates to how a development team creates a product, much like over-engineering. It occurs when so much effort is put into one part of a project that it reaches a point of diminishing returns. In other words, so much extra effort is focused on a part of the project that it no longer contributes value to the product.

This usually occurs when the development team adds extra features to a product in order to impress the client, particularly when the team finishes early. However, added features can add unforeseen, extra work to a project, and features that the client did not actually want in the product (remember that user requirements should be specified by the client and not by the development team). This can make the client disappointed and confused in the end product.

To avoid gold plating, the development team should stop working when the product functions well. If the team feels new features should be added, the features must first be vetted with the client. Good questions to ask when considering adding extra features include:

- Does this feature improve the product?
- Does this feature make the product confusing somehow?
- Does this feature take away from main functionality?

Another solution to avoiding gold plating could be conducting a user study. A user study involves taking a small sample of users and asking them to try out new features on a product and provide feedback. This feedback can inform the team on how useful the feature is, how it affects the interface, whether or not the feature has broad appeal, and how the product itself can be improved. Not all extra features are bad things, but these steps help to ensure that the extra features actually add value to the product.

Viewgraph Engineering

Viewgraph engineering is a group anti-pattern that occurs when too little effort is put into a project. It is the opposite of gold plating in that way. However, if gold plating is unfocused work, viewgraph engineering is working on what is not important.

Viewgraph engineering is not the same as hiring bad developers and missing deadlines, which is also a risk but a separate issue. Instead, viewgraph engineering is more directly tied to management practices and how those practices can make project work difficult for developers by requiring them to work on things other than development work, such as documentation, reports, or creating presentations. The time spent on such activities should not be greater than the time spent writing code or developing a project.

This anti-pattern commonly occurs in organizations with vertical management structures, where developers must constantly analyze the potential of a concept and prove it to management before going forward. In this way, viewgraph engineering is a kind of analysis paralysis on an organizational level.

Agile methods purposefully seek to avoid viewgraph engineering by placing focus on creating working software over comprehensive documentation. Although documentation should still be produced, it should not lead to unfocused development work.

In a similar vein, the best solution to viewgraph engineering is to remove any barriers developers might face in developing a project. Creating a basic prototype is often more informative than any report or presentation, so time spent on such documentation may be better served in development. If the prototype is good, then it can be built upon. If it is not good, then it can be scrapped, and the team will learn for going forward in both the project, and future projects.

Fire Drills (Heroics)

Fire drills and **heroics** are two types of linked grouped anti-patterns. Both are very common.

Fire drills happen when the development team does very little work throughout most of the project and then makes a big push towards the end of the project, resulting in overtime work in order to meet deadlines. This leads to a loss of quality in favour of producing a product quickly. Fire drills can have many causes, including:

- Bad management practices
- Developers wasting time
- Expectations were not properly established with the client

Fire drills can also lead to **heroics**. Heroics is an anti-pattern where the development team ends up relying on only one person's technical skills to get the project done. This requires that one developer to have almost superhuman skills, as he or she take on almost the entire project. Further, the team relies on only one person, which is bad for both the team and the developer.

To avoid fire drills and heroics, it is important to establish expectations with the client early on the project and to follow Agile software development practices. For example, using time boxes and always producing a working product at the end of those time boxes. These strategies keep the team working at a stead pace, avoiding both fire drills and heroics. They will also keep up team motivation for the project.

Death March

Death march is a group anti-pattern that happens when the development team and the management team are at odds, because the management team is enforcing a path for the project that the development team does not believe in. This leads the development team to lose conviction and passion for the project, which increases likelihood of project failure. In spite of this, everyone keeps working out of a sense of obligation.

Death marches can occur because of financial reasons, or when management is too stubborn to recognize other ideas and the potential failure of the project. In either case, if developer morale is low, then the quality of the product will suffer.

To avoid death marches, it is important for management to maintain open communication with the development team. In other words, managers should listen to what the team thinks about the direction of the project and be willing to explore other alternatives in project development. In some cases, simply reassuring the development team that you understand and recognize their concerns is all that is needed to avoid a death march.

Individual Anti-Patterns

Individual anti-patterns are anti-patterns caused by individuals instead of group behaviour. Individuals caught up in these anti-patterns usually find it hard to recognize it, but people outside the situation can identify these anti-patterns easily.
In general, work cultures that promote open mindedness, self-reflection, and improvement greatly help to counter individual anti-patterns.

The following individual anti-patterns are covered in this lesson:

- Micromanagement
- Seagull management
- Email as the primary means of communication
- Loose cannons
- Intellectual violence

The first two anti-patterns listed arise from the actions of individual managers, while the last two come from the actions of individual team members.

Micromanagement

Micromanagement is a very common individual anti-pattern. It occurs when a manager is very controlling and wants to be involved in every detail of a project, no matter how small. The manager needs to constantly know what their developers are doing. It demonstrates a lack of trust in the development team, which affects team morale and product quality. This can become exacerbated if the team is blamed for the manager's behaviour.

Micromanagers, in general, do not micromanage with the intention of bringing down developer morale. Instead, they usually think that a project will fall apart without them. Micromanagement is therefore usually born out of a manager's own internal fears, insecurities, or basic stresses. It can also be caused by overly ambitious timelines, poor product quality, or fear that developers are not up to the job.

Addressing micromanagement can be difficult, as there is no quick and easy fix. In general, micromanagers themselves have to provide a solution. Micromanager has to admit their behaviour is affecting the team and be willing to make steps towards improving their behaviour. Further management training may be needed before micromanagers can adjust their actions.

It is worth noting that wanting to stay informed on project progress should not be confused with micromanagement—in fact, asking for short daily meetings updating project progress is good Agile practice!

Seagull Management

Seagull management is an individual anti-pattern that happens when a manager is only present when problems arise or right before deadlines. This results in an otherwise absent manager swooping in, dumping a lot of information or requests on the development team, and then disappearing again. This can cause a lot of stress for the team and can easily push a project into become a fire drill project. Further, the development team will be working in constant fear of when the next dump will be.

The behaviour of seagull managers is usually caused by a number of factors, such as busy schedules, inadequate planning, or stress. Addressing seagull management behaviour involves allowing a manager to lighten their workload (although this might not always be a solution that product managers can employ, because it depends on things outside the scope of the managers themselves). In general, if you suspect you might be a seagull manager, it is a good idea to re-evaluate your schedule and how you interact with the development team.

Email as the Primary Means of Communication

Email as the primary means of communication is such a common issue in projects that it has become its own anti-pattern. The methods of communication used between managers and development teams can greatly impact a project. Email, in particular, lends itself to creating seagull managers who only write to the team with large workloads or big dumps of information, and do not communicate frequently otherwise (see above). Further, email is often not the best way for the team to communicate well with managers because it can be difficult to provide context in emails, or alternately, information can be easily misunderstood. Emails can also be too formal for most communications.

To address this anti-pattern, it is important that managers and development teams take time to communicate in person. When that is not possible, other means of communication such as chat services, video conferencing, and phone calls are good alternatives to email. All of these forms of communication have the added benefit of giving respect and attention to team members, which help to increase team motivation.

Loose Cannon

A **loose cannon** is an individual anti-pattern. The term “loose cannon” itself refers to an unpredictable person (or thing) who can negatively affect others, unless it is managed somehow. In software projects, loose cannons tend to be people who make significant project decisions without consulting the rest of the team. This behaviour is reckless and can create more work for other group members.

A loose cannon is not to be confused with someone who asserts their opinions on every topic, leading to poor project decisions. This is more closely related to the anti-pattern of intellectual violence, discussed below.
Like many individual anti-patterns, there is no quick fix to loose cannons. Often, dealing with loose cannons involves understanding the motivations of the individual’s behaviour. Are they trying to cause trouble? Are they trying to prove their work to the team?

MORE INFORMATION

The origin of the term “loose cannon” comes from the 17th and 19th centuries when wooden sailing ships used cannons as weapons. When a cannon is fired, it undergoes a large amount of recoil. To avoid damage from this recoil, cannons were tightly secured to decks using ropes.

A loose cannon refers to a cannon that has come free of these restraints, allowing it to roll around the ship and cause damage. The term has survived until this day!

Once this is understood, a manager or team can try and get the individual to understand that their behaviour is destructive and encourage the individual to make steps to change it. In some cases, organizational steps might be necessary to address the problem if the personality of the

loose cannon makes change very difficult. In any case, loose cannons should be dealt with quickly or the risk to a project will increase over time.

Intellectual Violence

Intellectual violence is an individual anti-pattern that concerns a person who asserts their opinion on every topic and impedes progress by questioning every decision and action or using their advanced knowledge in an area to intimidate or look down on other team members. These individuals tend to repeat their opinions so much that the rest of the group concedes to them just to avoid confrontation. In turn, this can lead to low team morale, with bitterness and apathy among group members that affects open communication and productivity. Loose cannons are sometimes the cause of intellectual violence.

There are a couple of strategies for addressing intellectual violence. The project manager can try talking to the individual in private about his or her behaviour and suggest any appropriate changes. Alternately, the project manager can encourage an open question policy (for example, reinforcing the idea that there is no such thing as a stupid question), and discourage prejudging opinions or inexperience. Intellectual violence should be addressed quickly to prevent a project from taking a bad direction and to keep up team morale.

Causes of Failures

Projects can fail not only from issues that present themselves as patterns, but also from project risks. **Risk** is something that could potentially cause a project to fail. Risks are found in every situation, not just in projects and project plans, so it is important to try and plan for them.

There are many different types of risks. These include:

- **Scope risks**, which refer to risks that involve expanding requirements. Agile methods are well equipped to address scope risks, because Agile has good change control measures. In other words, Agile deals well with change and changing requirements.
- **Technology risks**, which refer to risks that the technology used in a project will fail. This includes hardware failure, software protocols becoming obsolete or unsupported, lack of scalability with technology, or a lack of technological understanding on the part of the development team.
- **Customer** and **stakeholder risks**, which refers to risks that customers and stakeholders can bring to a project. These usually occur when customers promise to provide material or information to the team and they forget or deliver it late. Examples of customer or stakeholder risks include clients becoming apathetic or a change in the main point of contact between the team and the client.
- **Personnel risks**, which refer to risks posed by personnel on the development team. For example, team members may leave partway through a project. This can be especially problematic if the person who leaves has specialized skills that are not shared by other team members. Personnel risks could also refer to a lack of needed skills, conflict between team members, or communication issues, such as groupthink.
- Many other risks, such as legal issues, security problems, the destruction of physical locations, loss of interest from either management or development teams, industrial espionage, etc.

Risk Assessment, Likelihood, and Impact

In order to plan for risks, it is important not only to be able to identify potential risks, but to understand both the likelihood of a risk happening over the course of a project and the impact of that risk on the project.

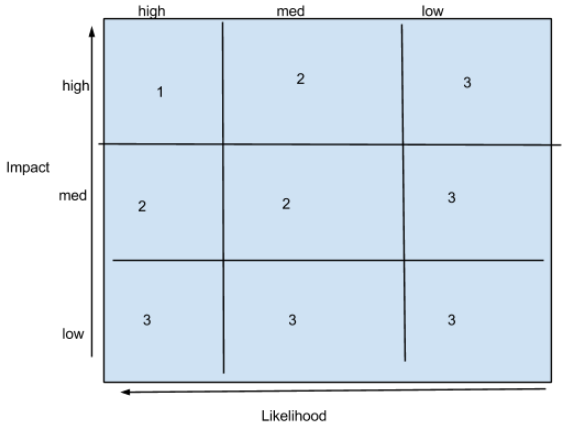
Risks may include the anti-patterns as discussed above as well as project-specific problems that may not be common. In order to assess risks, their likelihoods, and impacts, the first step is to create a list of risks. You should recognize that you can take action to avoid some risks, but this is not always possible.

Once risks have been identified, they can be prioritized. One of the most useful ways of prioritizing risks is using an impact vs. likelihood matrix.

Impact vs. Likelihood Matrix

An **impact vs. likelihood matrix** is a two-dimensional representation of the amount of impact a risk has on the project and how likely that impact is to happen. With this knowledge, risks can prioritized, and the team will know what to focus on preventing, and what to monitor.

In an impact vs. likelihood matrix, impact runs along the y-axis, from low to high. Likelihood runs along the x-axis, also from high to low. Different grids in the matrix are assigned different rankings based on how high the impact and likelihood is.



To use the matrix, the listed risks of the project should be assigned a value from the matrix based on their value along each axis. For example, if your project has identified the risk “project runs out of funding,” this is potentially a big risk. It is high on impact. But, if this only has a medium likelihood of occurring, then it would go in the middle column of the matrix, at the top of the grid.

Risks can range quite broadly, from being an immediate impact to something with a very slim likelihood, like “asteroid destroys the Earth.”

The numbers in each cell of the matrix represents a different risk ranking, as determined by each risk's value on the matrix. For the examples above, "project runs out of funding" has a risk ranking of 2, as seen in middle column of the top row. The risk "asteroid destroys the Earth" is a large impact, so it is in the top row. However, it also has a low likelihood, so it will be in the third column. It will therefore be situated in the bottom-right square, and has a risk ranking of 3.

After a list of risks has been made and assigned rankings, the team is informed of the potential magnitude of risks the project faces, so appropriate action can be taken regarding the risks.

Based on the ranking of the risk, the following actions are recommended:

- If there is a ranking of 1, then the team needs to mitigate the damage. The risk needs to be addressed as thoroughly as possible and quickly, as it is the most dangerous.
- If there is a ranking of 2, then the team should be concerned about the risk but do not need to take immediate action, since it is only moderately likely to happen. A strategy should be developed, however, for dealing with the risk should it come to pass.
- If there is a ranking of 3, then the team should spend very little time considering the risk, as it is not of great concern to the project and it is the least threatening.

Other variations on the impact vs. likelihood matrix are possible. Some projects may assign risk rankings differently. For example, some projects may assign a ranking of 1 to the high impact/high likelihood cell in the top left, as above, but also to the cells of high impact/medium likelihood and medium impact/high likelihood. In this scenario, a ranking of 2 is assigned to cells on the edges of those cells with a ranking of 1, and 3 to everything else. This puts a higher importance on some risks, which is known as decreasing a project's risk tolerance. This practice is important for **mission-critical projects**—projects that are important to a business or company, because they are tied to the success of that business or company.

Any actions taken to deal with risks adds activities to the project plan. These activities should be taken into account because they can influence both project deadlines and costs.

Risk-Value Matrix

A matrix is a useful tool that can also be applied to project features. This kind of matrix is known as a **risk-value matrix**. It is similar to an impact vs. likelihood matrix, but it focuses on showing how likely a risk is to influence the value of a project feature.

Accordingly, the y-axis of a risk-value matrix shows risk, while the x-axis shows the value of the feature to the project. Instead of assessing risks with the matrix, features are assessed. This helps prioritize which features the development team should work on first.
For example, consider the hypothetic project of building a mobile app with a database to store data. Let's say the development team is very experienced in creating databases but has little experience creating mobile apps. In this case, on a risk-value matrix, database features are low risk and high value, while app features are high risk and high value.

Using the ranking information the risk-value matrix provided, development priority can be assigned—those features that are high risk and high value should always be prioritized first. In this case, developing app features should be done first. Although instinct suggests that it is better to start with developing high value and low risk features first, as it theoretically be done with more ease, this is wrong. Tasks with high risk pose the most danger to success, and problems associated with these tasks should be tackled quickly. This way, any issues that may

cause delay or cancellation can be identified earlier in the project, rather than later, and time and work is not lost working on other features.

Risk Strategies, Contingency, Mitigation

In addition to be able to identify risks, likelihood, impact, and effects on project feature values, it important to be able to create a risk plan and implement it.

A risk plan, or a **risk management plan**, is a list of potential project risks, associated impacts, and likelihood, and any planned action for how to address the risk should it occur. Generally, risk management plans are presented in tables.

The table consists of four columns. From left to right, these columns contain:

- The risk
- The associated rank of that risk, as determined by the impact vs. likelihood matrix
- A description of the risk's **indicators** (An indicator is a sign that the risk is going to occur and is usually some sort of event or activity to look out for.)
- A description of the action to take if the risk presents itself

A risk management plan is better understood through an example. Imagine a project that seeks to create a music player app. In order to have a functional app, the client needs to obtain copyright access to the content they wish to provide users.

In an impact vs. likelihood matrix, it is determined that the risk "unable to obtain copyright to music for app" has a ranking of 2. It is high impact but only moderately likely to happen. Indicators that a failure to obtain copyright has occurred include clients or musicians communicating this to the team or if clients or musicians lose interest in the app. A number of actions can be taken to address this issue, such as:

- Ensuring the development team delivers the product in increments, so the client can see progress in the product and remain actively involved and interested in it.
- Settling financials of the project appropriately, so there is a guarantee that work done will be compensated.
- Some product managers may get involved with helping the client obtain copyright. This level of involvement largely depends on the scope of the individual position of the product manager.

In a risk management plan table, it would look like:

Risk	Risk Ranking	Indicators	Action

Unable to obtain copyright to music for app	2	• Musicians or client communicate difficulties in obtaining copyright • Musicians or client lose interest in app	• Deliver project in increments so progress is visible to client • Settle financials ahead of time so there is no risk of not being paid for work • Help client gain copyright access

Using a risk management plan for all risks ranked as 1 or 2 will greatly increase the potential success of the project.

Even with risk management plans, it is very important that the product manager and development team recognize that there will always be risks that cannot be predicted or planned for, as every project will face unique risks. However, this does not mean that risk planning should be ignored. Product managers should recognize that dealing with the unknown is a skill they will need to develop.

In cases of unforeseen risks, project success will largely depend on how those risks are dealt with in the moment.

Version 0.7.2