REVIEWS AND METRICS FOR SOFTWARE IMPROVEMENTS

COURSE NOTES

Table of Contents

Course Overview

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

- Understand the continuous feedback gained from different reviews and metrics that can be applied to any of the three goals of software development in order to create improvements for the work and product.
- Explain Sprint Review Meetings and what are part of Sprint Review Meetings.
- Explain the concept of user studies and how they can subjectively and objectively provide feedback.
- Provide example measures of product success.
- List and explain industry examples of processes used to create successful products.
- Explain peer review techniques used to improve project work.
- Define metrics and desirable properties of metrics, including popular metrics and defect-related metrics.
- Explain the Agile practice of Daily Scrum.
- Use various techniques such as velocity, Release Burndown Charts, Task Boards, and Iteration Burndown Charts to help monitor product development.
- Define retrospectives and exercises used in retrospectives.
- Apply retrospectives at the end of a sprint or project to improve the overall product or future products.
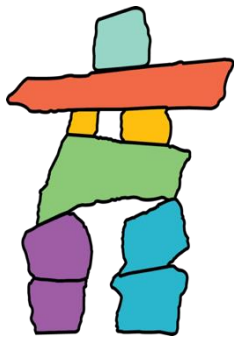
Introduction

Welcome to the **Reviews and Metrics for Software Improvements** course, within the Software Product Management Specialization. This course will focus on what happens once development has begun, in order to keep things Agile and on target.

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. The third course, **Agile Planning for Software Products** serves as the body of the inukshuk, which builds on the first two courses. This fourth course, **Reviews and Metrics for Software Improvements**

serves as the arms of the inukshuk, which builds upon not only the first two courses, but the third one as well. This means that **Reviews and Metrics for Software Improvements** brings together many of the fundamentals of the prior courses.

In other words, the fundamentals of this course – ensuring that the project is Agile and on target – begin after a product opportunity has been identified, potential users have been talked to, requirements have been defined, releases have been mapped out, and developers are in the process of designing and implementing the product.

Over the past three courses, you learned that building better software involves three goals:

- the right product
- done right
- managed right

This course outlines techniques used to determine if these goals are aligned. Specific types of reviews and metrics should be used for each goal to gain continuous feedback. This feedback can take the form of subjective or objective data.

Based on this data, improvements to the product itself, the underlying process, and the overall project can be continuously determined. As issues are addressed, the software becomes better and better.

For the *right product*, the software product manager and development team must:

- demo the product early to gain feedback early.
- observe users and how they engage with the product.
- gauge the effectiveness and efficiency of the product.
- collect data on user satisfaction.
- collect information about the product success in the market.

All of this can be used to identify what needs to improve in the product's features.

For the product to be *done right*, the development team must:

- review and inspect the work products created in order to identify any repair issues as soon as possible.
- monitor data about important quality factors of the work (e.g., defect rates).

All this information can be used to identify what needs to improve in product implementation and what needs to be adjusted for the product to achieve its desired quality.

To *manage right*, the development team must:

- meet briefly each day to update and synchronize with other members.
- track measurements, such as velocity.
- monitor the completion of planned user stories or tasks within a sprint, in a way that is transparent to everyone.

This information can used to coordinate the team and the work the team produces. It can also be used to identify work that needs improvement in order to become more efficient and less wasteful.

There are four modules in this course.

Module 1 covers reviews and metrics related to creating the *right product*, including:

- Sprint Review Meetings, where the product is demoed after a sprint and reviewed by stakeholders and other project members to produce feedback
- user studies that suggest improvements based on observations on how users interact with a product
- example measures for product success
- processes used in the industry to create successful, well-designed products

Module 2 covers reviews and metrics related to making sure the product is *done right*, including:

- peer review techniques used by developers to inspect and improve their work
- metrics and how they fit into addressing specific questions involving product goals or processes
- the desirable properties of good software metrics
- popular metrics and defect-related metrics

Module 3 covers reviews and metrics related to *managing a product right*, including:

- the common Agile practice of Daily Scrum
- a review of the concept of velocity
- Release Burndown Charts, which track completed user stories over successive sprints
- Task Boards and Iteration Burndown Charts, which track completed tasks within a sprint and help identify work that is only half done

Module 4 covers reviews and metrics related to learning from experiences on the project, including:

- retrospectives or practices based on lessons learned, which can be applied at the end of a sprint or project
- exercises related to project retrospective, which identify successes, reveal shortcomings, revisit what happened, recognize all contributors, and produce constructive lessons to carry into future projects

Module 1: Right Product

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

- Explain the term monitoring, and the goals of monitoring.
- Explain the term feedback.
  - Identify different types of feedback.
- Explain the term verification, the term validation, and the difference between the two.
- Explain the terms Sprint Review Meeting, Client Demo, Definition of Done, and Stakeholder.
  - Explain the difference between a Sprint Review Meeting and a Sprint Retrospective Meeting.
- Explain the three main parts of a Sprint Review Meeting: the client demo, the Product owner approval, and the stakeholder feedback.
- Understand that although clients can suggest improvements or requirements mid-sprint, these can only be added to the backlog during a sprint.
- Explain the terms user study and usability.
- Summarize different types of user studies, and how to gather data for them.
- Explain why both objective and subjective measures in user studies are important.
- Recognize that the above practices are grounded in the industry.
- Identify the different processes used by different companies.

Introduction to Monitoring

This module focuses on ensuring the *right product* is being delivered by the software product manager and development team. It will cover techniques related to this goal, including:

- Sprint Review Meetings
- User studies
- Industry examples

Terminology Review

Some terms from previous courses are key to understanding these concepts. They are reviewed below:

A **verified** product refers to one that works the way it was intended to, according to a set of requirements. This concept has been addressed before in both the courses **Introduction to Software Product Management** and **Software Processes and Agile Practices**.

Verification is commonly fulfilled through the use of unit and acceptance tests. **Unit tests** are tests that verify specific, low-level functionalities. In other words, they ensure the software goal of being built right. In test-driven development, unit tests are written before source code for the feature.

A **validated** product, on the other hand, refers to a released software product that satisfies its stakeholders. Validation ensures the software goal of the right product. A good product has validation from all **stakeholders**, which includes anyone affected by or who has an effect on the success of the product. Both clients and end-users are examples of stakeholders. Validation aligns with the Agile Manifesto principle,

"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

This concept has also been addressed before in both the courses **Introduction to Software Product Management** and **Software Processes and Agile Practices**.

Validation can be obtained from clients through frequent demos and open communication. Validation can be obtained from users through techniques such as user studies, surveys, or focus groups. In both cases, the information obtained is used to sure the right product is being built.

New Terminology

Monitoring refers to the tracking, reviewing, and evaluation of the product and process of the software development being undertaken. It is an essential part of meeting the goals of software development (the right product, done right, and managed right).

Monitoring is important to Agile, as it allows everyone to know the status of a project at any given time. Therefore, project transparency is also an important aspect of Agile.

Metrics are used in development to measure various aspects of a product, process, or project. Metrics provide quantifiable data and results to accomplish this. They are actually a means of monitoring.

However, it is important to remember not all important aspects of a project are quantifiable, and not everything that can be quantified is necessarily important to a project.

> You will examine metrics in more detail in the second module of this course!

Both monitoring and metrics help software product managers and development teams obtain **feedback**. Feedback is information or criticism that can be used to identify improvements in a product or process or to affirm that the project is on the right track. In fact, feedback can be used to improve or verify not only the product or process but also the communication or other practices used in the project.

Feedback can come in many forms, both qualitative and quantitative. Calculations, results from user studies, suggestions from the team or clients, and client reactions are all different forms of feedback.

Feedback is important, as Agile is committed to constantly improving a project or a product as often as possible. Continuous feedback allows this. In Lean methodology, feedback is also used when considering different alternatives to decide which option is the best for the project.

Sprint Review Meeting

Sprint review meetings are meetings held at the end of a sprint. They are opportunities for the development team to demonstrate their product and collect feedback from stakeholders. Sprint review meetings are a good technique to use to help determine if the right product is being made.

Sprint review meetings are Scrum events. Scrum is an iterative and incremental Agile software development methodology for managing product development. A review of Scrum terminology used in this lesson follows below:

- **Scrum events** are events that the Scrum Master is responsible for facilitating. They generally occur throughout the development process of a project. Examples of Scrum events include sprints, daily stand-ups, sprint review meetings, sprint retrospective meetings, etc.
- A **Scrum Master** is the role that requires the facilitation of Scrum practices.
- A **Product owner** is the role that is responsible for product backlog.
- **Sprints** are synonymous with iterations in this lesson.

Sprint review meetings are the first of two distinctive Scrum events that take place at the end of a sprint. In the Scrum review meeting, the product is demonstrated and discussed. The second Scrum event is known as a sprint retrospective meeting. In the sprint retrospective meeting, the development's team process is re-evaluated and discussed. Sprint retrospective meetings will be discussed in more detail in a later lesson.

All Scrum events are formal opportunities for the software development manager and development team to gain feedback on their work and product and to evaluate their progress on the project. They are also all time boxed, which means that they cannot exceed a predetermined amount of time.

Sprint review meetings are therefore both opportunities to gain feedback and a time-boxed event. The suggested time for sprint review meetings is an hour for each week of the sprint the review is for. For example, if a sprint took three weeks, then the sprint review meeting should be three hours long.

Roles of a Sprint Review Meeting

There are a number of important roles played by various attendees at a sprint review meeting. Sprint review meetings can be attended by any stakeholder in the project—in other words, any one with interest in the product is invited to the meeting. Typical sprint review meetings are attended by the Scrum master, the Product owner, the Scrum team, management, some customers or end-users, and sometimes developers from other projects.

The Scrum Master, as outlined above, facilitates the sprint review meeting. This involves making sure that the conversations during the meeting stay focused and on task. For example, Scrum masters ensure that suggestions are addressed at the end of the meeting, and they remind other meeting members of that when necessary. The Scrum master is also responsible for ensuring that people only voice relevant comments and that questions are asked or feedback provided appropriately.

The development team also plays an important role in sprint review meetings. The development team is responsible for running the meeting. Everyone on the team should have a turn to speak or give input in the meeting, because everyone on the team has played a part in creating the product.

The Product owner, as outlined above, is responsible for the product backlog. This means that the Product owner must approve features of the product and ensure that the Scrum team is following the Product owner's vision. Other stakeholders, such as users, also contribute to sprint review meetings by providing feedback and improvements at the end of a meeting.

Sprint Review Meeting Outline

Sprint review meetings are informal events. They are focused on demonstrating the product as finished by the end of the sprint. The general rule is that the development team should not be distracted or spend extra work creating slide presentations or other products for the meeting. Instead, demonstrations should need minimal preparation outside of the work that went into developing the software.

A sprint review meeting begins when the Scrum master introduces the outline for the meeting, which includes information such as the order of speakers and when members are allowed to make suggestions (generally at the end of the meeting). The goal of the sprint that is finishing is then reviewed, to help give perspective to the meeting.

Sprint review meetings generally consist of three main parts, which follow after the meeting outline and sprint goals have been explained.

1. The product is demonstrated. During this part, the Scrum team demos the product as realistically as possible on the platform the product was meant for. For example, a mobile app should be demonstrated on a mobile device.

   The product demonstrated should also be authentic. This means that what is demoed should legitimately be the product as it currently exists. It should not appear to work when it actually does not work. For example, features should not be hard coded for the demonstration so that they only work with specific input values. Only features that are truly ready and have met the definition of done should be demoed. Anything incomplete should be saved for a future sprint review, when they are done.

2. The product and features demonstrated are approved by the Product owner. Once features have been approved by the Product owner, they should also be removed from the product backlog. Removing items from the backlog can take place at the Sprint Review Meeting. Alternatively, the Product owner may also wish to take more time to fully evaluate the product, and features can be removed from the product backlog after the meeting, once the Product owner has had a chance to examine it in detail.

3. Stakeholders provide feedback on the product after it has been approved by asking questions, suggesting features, or providing other commentary.
   In general, feedback takes three different forms:
   - praise for the feature if it is on track
   - suggested areas for change or improvement
   - identifying areas or issues that lead to questions, new problems, or a revision of assumptions

   Stakeholders can also suggest as many new features as they like during this time. Remember that the product backlog is maintained and prioritized by the Product owner, not by the development team. Ultimately, the Product owner is the one who decides if and when a feature will be developed for the product.

Sprint review meetings allow the Scrum team to showcase their hard work while also increasing project transparency. The meetings provide a good opportunity to gather feedback on the work produced so far. Sprint review meetings also serve as something for the Scrum team to work towards, as it encourages the creation of working software.

User Studies

User studies are another means of validating that the right product is being created, by focusing on end users.

In the **Client Needs and Software Requirements** course of this specialization, usability was discussed in the context of creating a product that works well for users but also appeals to users. Users will not use a product that they do not like or that they cannot use.

DID YOU KNOW?

The desire to assess the ability of computer systems in their usefulness to humans has existed in many forms since computer systems have been designed. It has appeared under the terms ergonomics, ease of use, and then more recently, usability.

The International Organization for Standardization (ISO) has defined **usability** as the measure of "effectiveness, efficiency, and satisfaction with which specified users can achieve goals in particular environments" (ISO, 1998, p. 2). User studies measure the usability of a product.

Usability testing can take place on a user interface, such as a website, but can also test any kind of product. Because of that, in some cases, it is also known as product testing.

Key Aspects of User Studies

There are a number of key aspects of user studies that the definition of usability touches upon.

As explained in the previous section, user studies measure usability. This is done by measuring the three different elements of usability:

- Effectiveness
- Efficiency
- Satisfaction

Most often user studies focus on effectiveness and efficiency and assume that satisfaction will result from doing well in both those elements. However, even though there can be a correlation, this is not always true.

Various types of research methods can be employed in user studies. Research methods tend to depend on product design and the type of information the team desires to obtain.

Examples of research methods include exploratory studies, direct observation, etc. User studies are also conducted alongside other processes; such as interface design or product conceptualization.

In general, user studies give a certain number of users a set of goals to achieve by using the software product in question. This must all be conducted in a particular environment. All software products are created to achieve goals. Even products like games have an intended use. Through observation or collecting data on users, usability studies can determine if the software product is achieving its goal in the intended way and how easy it is to do so. They can also uncover if the user is using the product in an unintended way to achieve that goal, or to achieve a new goal entirely, and why this might be happening.

Whether or not a user achieves a specified goal measures the **effectiveness** of the product. **Efficiency** can be determined by how quickly the user accomplishes their goals. **Satisfaction**, on the other hand, can be more directly measured through a short interview after the study by asking users if they liked the product and felt confident about achieving goals. These answers might differ from what measured effectiveness or efficiency suggest, which is why satisfaction should be evaluated separately. For example, even if it takes a short amount of time to achieve a task, satisfaction might be low if users did not trust the result.

It is important that user studies select the right end-users for the study. This means those selected must be representative of the target audience of the software product and share similar characteristics. Examples of shared characteristics might be socio-economic, educational, or cultural background, or computer skill level. In fact, identifying target users, their needs and wants are the first things a good project will undertake, often through user research. Users in user studies should be drawn from this same pool of target users.

In general, the family and friends of the product manager and development team should not be involved in user studies because they may provide biased feedback.

User studies must also be performed in particular environments. The environment must be controlled, as well as the test method. This is covered in more depth in the next section.

There are actually many different kinds of common usability studies. Some are discussed below, but for a more thorough examination on different kinds of usability studies, see Hornbæk, 2006 in the **Course Resources** section. There, you can find a link to a study from the University of Copenhagen that explores different kinds of common usability studies. This paper also focuses on the challenges that usability studies and research address!

Different Types of User Studies

There are two major different types of user studies:

- In some studies, users are provided goals to accomplish.
- In other studies, users are instructed to navigate the software how they would like.

In cases where users must meet specific goals, user studies validate the ability of the product to allow users to achieve those goals. It also reveals parts of the software that may be problematic and prevent users from reaching the goals.

In cases where users are asked to use the product as they like, testers are usually asked to think out loud. The user study seeks to determine what the product naturally encourages users to do, and whether or not that use is different from the intended goal of the software. It may be that different purposes for the product are identified and consequently the product might be adjusted going forward—either to return to its original goal or to change the focus of the product.

In both kinds of user studies, users are usually observed and their actions or comments recorded. Observation is often bolstered through questions asked to users. These questions are meant to help determine how well the product met needs or to identify any negative experiences created by the product. This kind of observation can take place in a laboratory or through on-location testing and can be accomplished through note-taking. However, observation can also be done remotely through computer applications or digital capture. Many different software are available to aid in digital capture of user studies, such as through monitoring usage of an app or a web service to determine which features are most used.

> **TIP:** If you are interested in getting better, first-hand knowledge of user studies, you can often sign up to participate in a user study through local tech companies. This also has the added benefit of helping out local industry!

Both kinds of user studies can also give feedback on:

- How enjoyable it is to use the product
- The aesthetics of the product
- The product flow

Other notable types of user studies include:

Type of User Studies	Description
Shadowing users	This involves observing a user's average day. Testers can then identify what tasks a user completes in a day and if there is a need for a product to help accomplish those tasks.
Cognitive walkthrough	This involves usability experts using the product to complete a task and then using their skills to assess its usability. In essence, they ask questions and attempt to act like an end-user would.
A/B Testing	This type of user study compares two similar versions of a product against each other, in order to obtain quantifiable results informing which version of the product fares best. The difference between the versions of the product is usually just one feature.
Longitudinal studies	These kinds of studies actually assess user behaviour over a long period of time. Data for longitudinal studies can come from interviews or from monitoring software logs. They can reveal how quickly a user becomes familiar with using a product over time, what features are most used, and what work-arounds users develop in order to accomplish goals.

Results from user studies can be determined using both objective and subjective measures.

> **Objective measures** involve quantifiable usability metrics or in other words, anything involving numbers, such as time, the number of page clicks, etc.
>
> **Subjective measures** involve qualitative usability metrics, which are more opinion based.

Both objective and subjective measures of usability are important, because they might lead to different conclusions regarding a product. For example, if users spend a long time accomplishing a goal, this might seem to be undesirable on paper. However, by using subjective measures, spending a long time accomplishing a goal may have been something users enjoyed, which lead to increased confidence, involvement, satisfaction, and usability.

Subjective measures may also involve qualities that are not obviously related to the task the product sets out to accomplish. These are hedonic qualities, such as originality of the product, beauty, sociability, etc.

Industry Examples

Major companies such as Apple, Google, Intuit, and IBM have also developed their own means of producing the "right product."

Apple

Apple notably uses prototyping to create their products. Their methods are known as "**10-to-3-to-1**" or "**pixel perfect**" prototyping.

"10-to-3-to-1" refers to a type of prototyping where designers initially develop 10 different prototypes without restrictions. This is similar to Lean development, as discussed in the course **Software Processes and Agile Practices**, where many prototypes are examined before a decision is made on which to pursue. From the 10 prototypes, the top three are chosen and explored further. Then, the top prototype of the three is selected and designed to a "pixel perfect" level.

Apple's late stage prototypes are "pixel perfect," as they have been designed to the point of each individual pixel. In other words, the development team does not just design the interface, they go to the highest level of detail. This method of prototyping is quite expensive. However, Apple yields good results from these methods, so for the company, they are worthwhile.

The Apple company also holds regular reviews through a weekly executive team review. At this meeting, the team goes over its production process. Peer design meetings are also held between design teams and engineers. These meetings focus on improving design.

Google

Google is noted for creating the "**Design Sprint**" process.

The "design sprint" is a rapid and iterative process. It compresses the work of several weeks or months into a single week. Sometimes, "Design Sprints" are even shorter. This method is in line with the Lean principle to "deliver as fast as possible," as outlined in the **Software Processes and Agile Practices** course.

There are five major stages in a "design sprint." Each stage is meant to take a single day. These stages are:

1. **Understand**. During this stage, engineers and designers evaluate the problem their product is meant to address and identify their target audience.

2. **Diverge**. During this stage, participants are encouraged to develop as many solutions as possible for the problem, no matter how far-fetched they may seem.
3. **Decide**. During this stage, participants vote and discuss which ideas should be pursued in more detail from the previous stage.
4. **Prototype**. During this stage, participants create sketches and prototypes quickly. These prototypes focus on user interface and experience.
5. **Validate**. During this stage, products are given or demonstrated to users in order to elicit feedback for designers and engineers.

The "design sprint" has also been used in major companies such as Airbnb and Blue Bottle Coffee.

Intuit

Intuit is a company that creates accounting software. It was founded by Scott Cook, who also developed the process known as "**follow-me-home.**" This process helps to ensure that the right product is built.

The "follow-me-home" process involves watching a customer install and use the software. It provides developers with the ability to observe the user's environment first hand, which helps them better meet user needs.

In its initial use, Cook himself would wait for customers in stores, and observe them in their own homes with permission. Modern use of "follow-me-home" is more controlled. Now, Intuit arranges 20 visits per year with different customers. These visits are about an hour long. They still allow developers to take note of the user's environment and how the software fits into that environment.

In addition to helping to ensure that the right product is built, "follow-me-home" provides validation for the product. This is because it tests the satisfaction of the user, instead of testing whether or not specific functions work as intended.

IBM

International Business Machines (IBM), one of the largest information technology companies, uses the practice "Design Thinking." This process combines prototyping, user observation, and Agile development. It starts out by identifying the needs of users and comes up with solutions for those needs.

"Design Thinking" was initially created at Stanford University. However, IBM adapted the process to meet their own needs.

The four stages of "design thinking" traditionally include:
- Understand
- Explore
- Prototype
- Evaluate

IBM added three additional stages among those above:

Design Sprint Stages
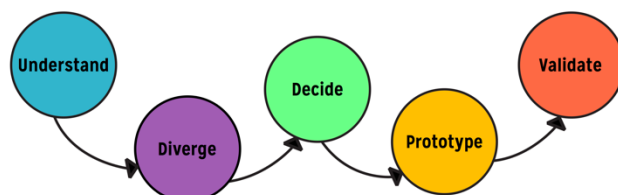
- Sponsor Users
- Hill Statements
- Playbacks

These stages unfold as follows in the IBM process. Developers and designers first seek to **understand** their users, as well as those users' needs or the gaps in those needs. Following that, a candidate is identified who would be interested in the product and who can provide insights throughout the development of the product. This candidate, or **sponsored user**, must share most of the characteristics of the target users.

The sponsored user works with the development team to identify a problem in the product. This is known as the **Hill Statement**. This problem is then taken to the larger team to solve. A number of solutions are **explored**, and any ideas that do not work are scrapped.

After solutions have been narrowed, sketches and **prototypes** are created. **Playbacks** occur if a few iterations are needed to determine the best idea.

Finally, a working model has been created and can be **evaluated** with end-users. The model is iteratively refined based on the feedback of the sponsored user. In "design thinking," a product is actually continually refined based on continuous tests and user needs, even after the product is launched.

Module 2: Done Right

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

- Recall the requirements criteria: correct, complete, clear, consistent, verifiable, feasible, traceable, manageable, and simple.
- Explain the different review techniques: software walkthroughs, software technical reviews, software inspections, and requirements technical reviews.
- Summarize important monitoring issues, such as missing metrics, knowing why you are measuring, and quantifying everything.
- Explain the concept of Goal Question Metric, including associated terms: question, goal, and metric.
- Create a metric for an appropriate quality.
- Differentiate between metric, measure, and indicator.
- Explain the desirable properties of a good metric.
- Recognize that popular metrics have a process
- Explain popular metrics used to measure the non-functional requirements of maintainability, complexity, performance, reliability, and success.
- Explain defect analysis.
- Determine rate of defects found and fixed.
- Summarize the concept of software barrier, subsystem, pre-release, post-release, and defect density.
- Explain the relationship of defects with subsystems.
- Recognize when a software is good enough to be released and that it's better to find defects early.

Review Techniques

The previous module **Right Product** focused on the software development goal of producing the *right product*. This module, **Done Right** focuses on the software development goal of ensuring the product is *done right*.

This lesson examines important **review techniques** used to examine the work products created throughout a software project. Review techniques are meant to help identify defects early, when they are cheaper to fix, and to help make sure the product is created right and the project is on track.

There are many different types of reviews. Some have already been explored throughout this specialization, such as sprint review meetings. Reviews examined in this module are more internal to the development team; they are actually **peer reviews**. **Peers** refer to those who work on the same "level" as another. For example, two developers are peers.

DID YOU KNOW?

The use of peer reviews falls in line with the Agile software development technique known as pair programming, or peer programming. In this technique, two programmers work together in developing a feature. Both write the code, while both review the code as it is created. This helps catch errors in the code and improve it. Peers can be a valuable resource in software development.

Reviews can be formal, informal, involve detailed technical code analysis, involve high-level requirements design, etc.

In this module, the following types of reviews are covered:

- software walkthroughs
- software technical reviews
- software inspections
- requirements technical reviews
- requirements inspection

These reviews focus on creating better software, although there also exist reviews that are more management focused. These management-focused reviews are not covered here.

> It is of note that review meetings should not be confused with other kinds of meetings that have different goals, like elicitation meetings (see the **Client Needs and Software Requirements** course). Review meetings focus on improving products and processes already in progress, while elicitation meetings are meant to help identify client needs at the outset of a project.

Software Walkthroughs

Software walkthroughs are the least formal of all types of software peer reviews. In a software walkthrough, the developer provides a short "show-and-tell" for a small audience of peers. Basically, the code is demonstrated, and during the walkthrough, peers are allowed to ask questions or point out potential issues. Software walkthroughs tend to be quick with the focus being on major issues; the minor issues are often skipped over.

Software walkthroughs tend to be informal because they are quick, easy ways of helping the developer improve the quality of the software. No work product is expected to come out of a walkthrough.

When reviewers watch the code demonstration, they check:

- for reliability of the code
- the security of the software
- the readability of the code
- that the code meets the team's coding standards

There are many advantages to software walkthroughs. As outlined above, they are a good technique for harnessing the power of peers, and they allow suggestions and improvements to be quickly applied to code. Additionally, they provide the developer's peers with the opportunity to learn about the feature's code. This knowledge can be used to enhance other projects that the developer's peers might work on in the future. Software walkthroughs are therefore also a good way to train new team members.

Software Technical Review

Software technical reviews are more formal reviews than software walkthroughs, although the level of formality used can vary depending on the organization implementing the review and the organization's goals. For example, military companies are likely to use software technical reviews in a much stricter manner than a mobile gaming company.

As the name suggests, software technical reviews specifically address technical aspects of the product. Because of the technical focus, software technical reviews are often used to create more robust designs or to form more well-defined requirements from a client. Of course, they are also used to make technical improvements in the code.

Software technical reviews are discussion oriented. There are three main roles for this review:

- The **decision maker** who sets the goals for the review and determines if those goals were met.
- The **review leader** who keeps the review orderly and on track.
- The **recorder** who documents the issues identified over the course of the review. A recorder therefore does not take "meeting notes" but instead makes notes about the findings of the meeting.

One person can have multiple roles in a software technical review. For example, a decision maker can also be a recorder. Reviews may also be attended by other development team members and contributors.

Before the review begins, all participants must prepare in some way:

- The decision maker creates a list of objectives for the meeting.
- The review leader makes sure that participants have access to the software product being reviewed.
- Other parties must review the material.

Once the review meeting starts, the review leader facilitates the conversation between all relevant parties. The goals for the meeting may cover issues such as technical processes, resolving ambiguities, creating more efficient code, improving the production process, improving or creating new requirements, etc. By the end of the meeting, the goals outlined by the decision maker should be met, and a list of recommendations should be created to address those goals.

Software Inspections

Software inspections are the most formal type of review discussed in this module. They are also known as "formal technical review." Software inspections follow rigid structures and involve multiple roles. The goal of an inspection is to find and fix defects in some work product, such as a set of requirements or a module of code.

The roles in a software inspection are:

- The **author** who created the work product being inspected.

- The **moderator** who oversees the inspection and ensures it goes smoothly (the moderator is similar to the review leader in a software technical review).
- The **reader** who reads the document to the inspectors.
- The **inspectors** who point out defects in documents.
- The **recorder** who take notes of the major findings in the meeting.

One person can have multiple roles in a software inspection.

Software inspections also have multiple stages. Sometimes, some of these stages are repeated until the work product is deemed to meet appropriate standards by the team.

In general, strict procedures apply for determining if a stage is finished. For example, each stage might have a checklist that needs to be completed before the review can move on to the next stage.

The stages of a software inspection are:

Stage	Description
Planning Stage	In the **planning stage**, the moderator familiarizes themselves with the work product being reviewed and primary areas of concern to the author.
Overview Meeting	In the **overview meeting**, the people filling the roles outlined above come together. In this meeting, the participants are briefed on the work product by the moderator. At the end of this stage, inspectors are given the product to review.
Preparation Stage	In the **preparation stage**, the inspectors review the product. They identify defects and make notes on them. This stage can take anywhere from a few minutes to a few days, depending on the work product under review.
Inspection Meeting	In the **inspection meeting**, readers go through the document in small pieces. For each piece, inspectors comment and detail the defects they found in that piece, and the recorder makes note. The inspection meeting has a rigid process: the author can answer questions to clarify on any issues the inspectors bring up, but discussion is not generally encouraged.
Rework	During the **rework** stage, the author uses the feedback gained from the inspection meeting and makes changes to the work product as necessary.
Follow-up	The final stage is the **follow-up** during which review participants make sure that all changes have been completed, are appropriate, and address the defects previously identified in the inspection meeting.

The rework and follow-up stages are the most likely to be repeated, especially if it is found that not enough work has been done to address identified defects. Such cases are generally recognized in the follow-up stages. The author goes back to finish work, and another follow-up is performed, until the work product meets desired standards.

Requirements Technical Review

Requirements technical reviews are popular techniques used to ensure high quality requirements. They help to make sure the product is *done right.*

The roles in a requirements technical review are:

- **Reviewers** who try to identify as many defects as possible in a set of requirements written by the author. Ideally, reviewers are not part of the project. Distance from the project makes it easier for reviewers to determine if there are issues with the requirements.
- The **autho**r**(s)** who created the requirements being reviewed.
- The **moderator** who is responsible for making sure the discussion meeting stays on topic and that criticism is presented in a way that does not feel like an attack on the author.
- The **recorder** who makes notes on the important findings of the discussion meeting. These notes will help author(s) to fix identified issues.

The role of moderator is one well suited to Scrum Masters, as they already moderate other meetings in a similar way. As well, the recorder does not necessarily need to be someone on the project. More than one reviewer can be involved in conducting the review at a time, alongside the development team.

There are two main stages to a requirements technical review. These stages are the **review** stage and the **discussion meeting**.

In the **review** stage, reviewer(s) go over a set of requirements written by the author(s) in order to identify as many defects as possible. This may include errors, omissions, etc., that would prevent the product from working. Sometimes reviewers are aided by a glossary or other important information sources.

Reviewer(s) also identify high-risk areas in terms of requirements criteria or any other related concerns. For example, requirements may be incomplete or could be improved in some way.

In a review, requirements are specifically assessed by reviewers against criteria as discussed in the **Client Needs and Software Requirements** course. This criteria suggests requirements should be:

- **correct** (each requirement represents what the client envisioned)
- **complete** (there are no missing requirements)
- **clear** (requirements are free of ambiguities)
- **consistent** (requirements do not conflict with one another)
- **verifiable** (requirements are measurable or testable in some way)
- **feasible** (requirements are able to be completed by the development team with the available resources)

- **traceable** (each requirement is connected to the associated design and implement artifacts in some trackable way)
- **manageable** (requirements are independent, meaning they are expressed in a way that allows them to change without excessive impact on other requirements)
- **simple** (requirements should be kept minimalistic and focus on one core idea)

Reviews may also go over requirements according to a set of guidelines that outline specific issues reviewers need to consider.

After requirements are assessed and defects are identified, reviewer(s) classify their findings by level of severity:

- **major**: when the repair for the problem is not obvious and a careful, further exploration is required
- **moderate:** when the repair for the problem is straightforward but needs to be reviewed further
- **minor**: when the fix for the problem is either unnecessary or obvious, and there is no need for further review

In the **discussion meeting**, the reviewer(s), the author(s), the moderator, and the recorder meet to discuss the findings of the review and reclassify the level of severity of those findings, if necessary.

Discussion meetings are generally advised to be time-boxed meetings, so they have a strict time frame that cannot be exceeded. This helps the discussion to stay on track and ensures that neither reviewer(s) nor author(s) lose too much time to this stage. Because of this, minor issues should take as little time as possible to go over.

In a discussion meeting, the author does not need to respond to every question or comment raised in the findings. In fact, the author is primarily at the meeting to better understand the issues and obtain clarification where needed, instead of there to address issues. The recorder, however, should make note of questions, comments, and findings. These practices help keep the meeting within the time box.

After the Requirements Technical Review process is a **repair** process. In this process, the author of the requirements uses the feedback from the review to adjust the requirements. Not all suggested changes are required to be incorporated into requirements because some changes may not be valid or have already been addressed. This is up to the author's discretion.

Conducting a Requirements Technical Review helps to reinforce the concepts behind quality requirements and development.

Requirements Inspection

A **requirements inspection** is a specific type of software inspection. A popular technique, a requirements inspection is a formal process that helps ensure that requirements are clear, consistent, and complete. This makes the project more straightforward and easier to understand.

The roles in a requirements inspection are:

- **inspectors** who review the user stories.

- **author(s)** who created or help create the user stories. Product owners, software product managers, clients, and even development team members could all be authors.

During the inspection process, inspectors look for issues in the requirements, usually in the form of a list of user stories, against the criteria of:

- **ambiguity**
- **consistency**
- **completeness** (see **Requirements Technical Review** above, or the **Client Needs and Software Requirements** course)

Ambiguity in particular is an important issue to address, and third-party inspectors are often in the best position to identify unclear wording.

Once requirement issues are identified, the author(s) must fix them. The amended requirements are then presented in a follow-up.

Monitoring Issues

As you have seen, many different kinds of approaches can be used to monitor a project and track the progress, productivity, and quality of a work product, including velocity (see the **Agile Planning for Software Products** course), code walkthroughs, requirements technical reviews, etc. However, a number of issues also exist related to monitoring projects. For example, many projects skip metrics, and some metrics are not effective.

Skipped Metrics

Many software projects avoid measuring or using metrics at all. There are many reasons why metrics might be skipped, but it can be problematic if a project fails to identify issues that need to be caught and fixed properly. .

One of the most common reasons metrics are skipped is because of time. Developers have a limited amount of time to code, so any time spent on non-coding tasks, such as metrics, can be seen as an impediment on development. This issue is particularly complicated in an Agile framework. Agile focuses on fast and efficient development, so using metrics can sometimes seem like taking a step back for an Agile team. However, well-used metrics can prevent expensive mistakes down the road.

Metrics are also often skipped due to a lack of industry standards. Very few industry standards exist because many studies on metrics are inconclusive or cannot be generalized. Some studies suggest that metrics work better in certain situations or only for certain organizations. But there are many metrics to choose from, and since there is a lack of industry standards, it can be difficult to determine which metric is best to select and invest the time in for the project at hand.

The main issue behind skipped metrics is the lack of development of manager knowledge in this area, which often leads to the misuse or dismissal of metrics.

Ineffective or Misused Metrics

Why Are You Measuring?

Another common issue is the use of ineffective metrics. Often metrics are calculated, but they are not useful to the project or do not inform the project in any way. This misuse can lead to work that does not benefit the project, which indicates that there is a lack of understanding as to why things should be measured.

This can be well illustrated by the common metric in the industry, known as lines of code or LOC. This metric tracks the number of lines of code written for a piece of software. This number is then used to generate data related to the project. The data could include information such as the price per line of code (the cost of the project divided by the number of lines of code), or time per line of code (the average time it takes to write a line of code for the project).

Very quickly, however, it becomes apparent that this metric is problematic and unreliable. For example, different parts of the project, or even different projects can vary drastically from each other, leading to very different numbers of lines of code. Different programming languages can also influence the number of lines of code needed, even for the same feature. This makes it impossible to use lines of code as a means of assessing a project because circumstances prevent the numbers from being meaningful or comparable, even for the same coding language.

Metrics are also susceptible to misuse. For example, measurements can change a developers' behaviour. Once a measurement is known, it is human nature to want to maximize the number. This can lead developers to write more lines of code than they would have if they had not known about the metric. Focusing on writing more can actually lead to a decrease in productivity, as the goal shifts towards creating quantity rather than on delivering the product. For this reason, it is important to understand why something is being measured.

Quantifying Everything

Metrics are rendered ineffective when they are used to quantify things that do not need to be. Sometimes, it is tempting for projects using metrics to quantify everything. For example, you could calculate the number of times the letter "a" shows up in a piece of code, but this metric would not improve the project and is not useful.

It is important when using metrics to know that what you are measuring is important, and why it is being measured. Is the metric related to the product in a way that is useful? Can it help improve the product? This way, the trade-off can be assessed between the value and insight the metric can provide the project and the time and other associated costs it takes to perform the metric. Measuring and monitoring a project can lead to improved software products and processes if it is done in a smart, informed manner.

Sociologist William Bruce Cameron once noted:

"Not everything that can be counted counts, and not everything that counts can be counted."

This phrase is a useful way of summarizing why it is important for projects to be conscious of not over quantifying everything.

Goal, Question, Metric (GQM)

Goal, Question, Metric (GQM) is a method of choosing metrics for software projects. It helps to ensure that the metrics used are measuring a part of the project that is valuable to the product manager and development team. In other words, it helps make sure metrics are used effectively on a project.

> The GQM method was developed by Victor Basili from the University of Maryland in 1994. In his own words,
>
> > For an organization to measure in a purposeful way, it must first specify the goals for itself and its projects. Then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals.

The philosophy behind GQM begins with the idea that before you can be sure you are using the right metrics on a project, you need to understand what you want to measure first. This is the **goal**. Goals are generally used to improve objects. Objects in a software project can be classified into categories that help identify how to measure them.

Objects might include:

- work products, such as code or documentation
- processes, such as testing or elicitation processes
- resources, such as office space, computers, salaries, work effort, etc.

Once you know what you want to measure, you must assess what defines the goal. In other words, what **question** do you need to ask to ensure the goal is being met? Questions should be specific enough that their answers address the goal. Another useful practice is to break down big questions into smaller ones that focus on specific facets of the goal. Together, all the questions should address the entire goal.

Only once the goals are defined with questions can you move on to implementing **metrics** and collecting data. The metric is the quantitative way to address the question asked.

> The GQM method is similar to the process of eliciting requirements, which was discussed in the course **Client Needs and Software Requirements**. Many comparisons can be drawn:
>
> In eliciting requirements, requirements must be discussed with the client so that needs and wants are clearly defined. They are not simply gathered. In GQM, you do not simply develop questions; instead, the goal of the metric should be well understood first.
>
> In eliciting requirements, once the requirements have been identified and expressed as user stories, you do not build them right away. Instead, user stories are broken down into smaller, manageable developer tasks. In GQM, questions should also be broken down into smaller ones focusing on specific aspects of the goal of the metric, but when added together, the information addresses the entire goal.
>
> When working with requirements, work in a software project should only begin once the tasks and user stories have been planned with a release plan and an iteration plan. In GQM, metrics and data can only be used once the goal and questions are clearly defined.

Although similar to metrics as described in other sections of this course, in GQM, metrics are made more valuable because of the planning process preceding their use. The questions the metrics are designed to answer were thought out ahead of time, which help to avoid issues of ineffective of misused metrics.

Metrics in GQM can be objective or subjective. Objective metrics are based on data that does not rely on any individual's viewpoint or interpretation. They generally involve numbers, such as the number of downloads, sale numbers, processing time, the number of complete user stories, etc. Subjective metrics depend on interpretation, however, such as ratings or developer morale.

A simple example of the use of GQM helps illustrate the process. Imagine that a product has been created and released to the public, and you would like to know how satisfied customers were with the product. The goal in this case would be customer satisfaction. With the goal identified, a number of questions can be created to address the goal. For example, is the product reliable? Is the product fast? How do users rank their satisfaction? How many times do users engage with the product over a given period of time? From here, metrics can be identified to answer those questions. In this case, this might include defect density, response times, collecting customer ratings on a scale of one to five, or collecting data on the number of times the software is used.

> **TIP:** Using the FURPS+ acronym can help you develop questions in GQM.
>
> As a model by Robert Grady at Hewlett-Packard to classify software quality attributes, FURPS+ stands for:
>
> - **Functionality**, where questions could be related to how well the product works, its security, or if it meets customer needs and wants
>
> - **Usability**, where questions relate to effectiveness, aesthetics, the quality of documentation, etc.
>
> - **Reliability**, where questions relate how the system reacts to failure, the accuracy of product results, etc.
>
> - **Performance**, where questions relate to how quickly the product works, throughput, the memory consumption of the product, etc.
>
> - **Supportability**, where questions relate how serviceable the product is, whether or not it can be configured, whether or not there is some means of monitoring the system, etc.
>
> - **Plus questions** related to design constraints, implementation requirements, interface requirements, or physical requirements

Desirable Properties of Metrics

In order to use metrics that can provide trustworthy results, it is important to understand the desirable properties of metrics. The following key terms are properties that can be used to help identify what is a "good" or "bad" metric or, in other words, a metric's desirable properties.

A **measure** is a standard or unit of measurement. Each unit is one instance of that measurement. For example, inches are a measure. "Inch" is a standard; each inch is a unit, comparable to another one. Without consistency, metrics cannot be meaningful.

A **metric** is a combination of two or more measures that provides a meaningful result. For example, kilometres per hour (km/h) is an example of a metric. Kilometres and hours are two measures that when combined are meaningful. Not all metrics are meaningful, since almost anything can be measured. For example, number of keystrokes per hour is not a helpful metric, because it does not necessarily provide any meaningful information about the project. Keystrokes are only loosely related to the creation of software—different programming languages may require more or less keystrokes to write out the same feature. Further, progress can still made on a project, even when little typing occurs, and vice versa—a large amount of typing does not necessarily mean progress was made on the project or that all the typing done by one person was one-hundred-percent project related.

An **indicator** is a measure that draws the attention of the person making the measurements. The value of the indicator informs the numbers in some way, which would be meaningless alone. For example, if a person's oral temperature is between 36.5 and 37 degrees Celsius, this is an indicator the person is healthy. If their temperature is over this range, then it is an indicator that they have a fever. Indicators therefore also signify if there is something to be aware of in the measures or metrics.

As outlined in the lesson **Goals, Questions, Metrics (GQM)**, the right metric will help answer the questions about the goal of the metric. Understanding desirable properties of metrics helps identify if the right metric has been chosen.

Desirable Property	Description
Simple and computable	The more complex a metric is, the more likely it is for the person using it to make a mistake. Further, if a mistake is made in a complex metric, it can take a long time to track down and fix. For these reasons, simple and computable metrics are more desirable.
Intuitively persuasive	The metric should make sense to use, even for someone outside of the team. This means it is intuitive.
Objective	The metric should not depend on any one person's opinion. It should instead be based on some empirical data so that it can be applied to a broad number of issues. For example, measuring completed story points over a period time is a better metric for productivity than simply asking a developer to guess what their own productivity is.
Consistent in the use of units and dimensions	The metric should have a steady, consistent definition of what is being measured. This makes the metric easier to interpret, comparable, and meaningful. For example, switching from a metric of story points per

	sprint to hours per sprint halfway through a project prevents the first sprints from being comparable or usable in planning future sprints.
Programming language-independent	The metric should be reliable across different programming languages. Programming languages can be very different from one another, and even within a language, the coding style of developers can vary. If the metric relies on the programming language, it will not be a consistent unit of measurement.
Provide an effective mechanism for improving software quality	The numbers the metric tracks should actually improve the project, or they are not useful. The metric should therefore be an effective means of identifying whether or not improvement has been made, or conversely, if quality has declined in the product.

Although other abilities of metric might be seen as desirable, such as reducing hours, this is not considered a broader desirable property of metrics. It is actually a goal of metrics because it a measurable outcome, instead of a property, which is more of a quality of a metric.

An example of a good metric is the number of defects found in a product per week. This is a simple calculation, as all that must be done is track the number of defects reported in a week through bug reports or logged issues and then add them up at the end of every week. It is intuitively persuasive because defects are clearly negative and decreasing defects creates a better product. It is a consistent metric across units and programming languages, because assuming developers are about equally skilled in one language as another, the number of defects per week should remain fairly consistent across different languages. It is an objective metric because defects in code are not based on someone's opinion—a logged defect is an issue that must be addressed. It is also an effective mechanism for improvement. Tracking defects each week over time should allow you to see if defects are increasing or decreasing as the project progresses. If processes are used to help increase code quality or the number of defects that can be fixed, then the number of defects per week should decrease over time, and the product should improve.

Number of defects per week is a metric that also provides the added bonus of being scalable to specific parts of a product, as well as applicable to the product as a whole. Problem areas can then be focused on, if desired.

It is important to remember that metrics are not always absolute; they may be relative only to a specific project. For example, velocity (see the course **Agile Planning for Software Products**) is the metric of story points per sprint. Story points are relative estimates that can vary from project to project. Using relative metrics is fine, as long as you remember that two metrics based on slightly different measures are not a hundred percent comparable. Velocity for one project cannot be compared with the velocity of a very different project.

Popular Metrics to Improve Software Projects

This lesson focuses on popular metrics that are used to assess the product and project based on non-functional requirements, so the goals for the metrics discussed in this lesson are based on non-functional requirements. Non-functional requirements are those that describe how well a product must perform (see the course **Client Needs and Software Requirements**). They are

also known as quality requirements, which addresses issues such as accuracy, dependency, security, usability, efficiency, performance, and maintainability.

Metrics generally become popular if process is involved. This means they are able to provide insight into improving the process of creating the product.

Goals for metrics according to non-functional requirements and the popular metrics associated with them are explained in the table below.

Non-Functional Requirement	Popular Metric
Maintainability	To measure maintainability, complexity metrics are often used. Complexity metrics measure the complexity of a product's source code. If the code is too complex, it may need to be reduced so that the product can be better maintained. Detailed explanations of the means of measuring complexity in product code goes beyond the scope of this lesson. However, what is important to understand is that complexity is related to the maintainability of a product.
Performance	To measure performance, metrics like response time are often used. Response time is a metric that describes how long it takes for a task to complete once it has started. For example, the time it takes for an application to receive data from a server is a response time. If this measurement is averaged over time, it can be used to determine if performance has improved.
Reliability	To measure reliability, measurements like project uptime are often used. Uptime measures in percentages how long a product is "up" and available to users. For example, if a file-sharing app is available to users and can send and receive information 99% of the time, this is the project's uptime.
Product Success	Product success is often assessed through factors such as customer satisfaction. As discussed earlier, this can be measured through product ratings averaged over the number of users.

Other aspects of a product can be measured too, such as correctness. However, this is expressed more through functional requirements. Correctness is often measured using defect analysis, which is explored in the following lesson.

A number of popular metrics were outlined above. There also exist unpopular metrics, which fail to offer improvements to the process used in the project. These might include lines of code delivered, size of documentation, etc.

> The McCabe number, part of the metric known as cyclomatic complexity, is another example of a metric that can be used to indicate the complexity of a program. It is a measure of the number of logical branches there are in a product's source code. This is a very code-specific metric, so more details extend beyond the scope of this specialization, but it is still worthy of mention.

Defect Analysis

Defect analysis is a way to assess the quality of a product by analyzing the number of errors in the product. An error is something that fails in the product. Often product errors are associated with the source code, but errors can also occur elsewhere. Defect analysis a common tool in the industry, and it is considered one of the most useful.

Traditionally, defect analysis is performed by tracking the number of defects in the code and comparing that number to the amount of defects that have been fixed. The number of defects can be found by adding up all the errors reported in the system. These reports may come from testers before the product is released, known as **pre-release**, or users after the product is

released, known as **post-release**. There are also many tools available that can help track bugs and other errors, as well as updates on whether or not those issues have been addressed.

The number of errors can then be tracked and compared over time. The rate of defects found is compared to the rate of defects fixed. These numbers can serve as an indicator of whether or not more focus should be placed on fixing current problems instead of creating new features. If more bugs are found than fixed in comparing the two rates, then more work needs to be done to fix problems. Consequently, the number of defects fixed should rise. This serves as a means of tracking improvement. Over time, the number of defects fixed should increase, and fewer defects should be found.

When the number of defects created and found is the same as the number of defects fixed, this is known as a **software barrier**. Once this software barrier is reached, it also indicates a need for change in the project. More time should go into fixing bugs, just like when the number of errors exceeds the number of fixed defects. It also means the product is not ready for release.

Subsystems

Defect analysis can also be performed on **subsystems**. Subsystems are different components of a product that usually have a specific purpose. Together, subsystems make up a whole system. In order to determine defects in subsystems, each subsystem associated with a defect must simply be noted. This applies for both defects found pre-release and post-release. Once defects have an associated subsystem, the numbers can be tallied for each subsystem. Subsystems with more errors will likely need more attention. It should be noted, however, that a fair comparison will need to normalize for subsystem size, since bigger subsystems will have more defects due to their size.

Defect Density

Defect analysis can also be performed using the metric of **defect density**. Defect density operates under the idea that rigorous testing and revision can bring a product closer to a perfect status. In order to calculate defect density, the ratio of the number of defects found in the source code is compared against the size of the code. The size is measured in "thousands of lines of code."

Defect density is the standard way of determining how any defects the product has as a whole. The industry average for defect density is between 15 and 50 defects per thousand lines of code.

Defect density can be used to help determine if a product is ready for release. This is because defect density can be predicted based on the defect density of previous releases.

For example, imagine you have created a product with two previous releases. You are trying to determine if the current and third version of the product is ready for release using defect density.

In the first release, 900 defects were found pre-release, and 100 defects post-release in a total of 100,000 lines of code. The first release therefore had a total of 1,000 defects per 100,000 lines of code, or 10 defects per thousand lines of code.

In the second release, only 400 defects were found pre-release, and 45 defects post-release, in an added 50,000 lines of code. A total of 445 defects were found. The defect density is therefore 8.9 defects per thousand lines of code.

In the third version, 100,000 lines of code was added to the product. 600 defects have been detected so far, so the current defect density is 6 defects per thousand lines of code. Although this number seems smaller than the previous two releases, the product is still not ready for release. In the previous versions, 90% of defects were found in pre-release. Unless a significant change has happened in the development process, the 600 defects found in this version should also represent 90% of the overall errors in the code. This means you should expect to find another 120 to 255 more defects. It is best practice to find defects before a product is released, so an effort still needs to be made to find those remaining errors.

Subsystems with high post-release defect numbers can be improved through more pre-release testing, or by bringing in more senior developers to work on the team, who have more experience to catch these problems.

> **DID YOU KNOW?**
>
> One famous example of defect analysis is that of the space shuttle flight software developed originally by IBM. This software is famous for being error-free. When IBM developed the software, it did not just fix defects that were found, the defects and the circumstances surrounding it were also analyzed. Then, similar areas where the defect could happen again were identified and tested. Further, the entire development process was adjusted to prevent similar defects in the future.
>
> You can read more about defect analysis in the **Course Resources**.

Module 3: Managed Right

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

- Explain the scrum process of the **daily stand-up meeting** and the three questions of the stand-up (what did you do?; what are you going to do?; what is blocking you?).
- Map out and explain **estimated velocity** against **actual velocity**.
- Explain how to predict velocity.
- Recognize how velocity is an Agile practice and that velocity changes depend on factors such as learning curves, bugs, risks, etc.
- Define the term **velocity driven**.
- Create and describe a **release burndown**.
- Define the term **burn up**, and **burn down**.
- Create and describe a **total work done release burndown** and an **adjustable floor**, and how these can be used to demonstrate changes in requirements and actual velocity.
- Create and describe an **iteration burndown**, which is generated daily
- Summarize the concept of a **whiteboard task board** and how to create an iteration burndown from the task board.

Stand-ups

Daily Scrum

The previous two modules focused on the software development goals of producing the *right product*, and ensuring the product is *done right*. This module focuses on the software development goal of *managed right*, which includes keeping a project on schedule and ensuring development is effective and occurring at a sustainable pace.

This lesson examines the process known as **Daily Scrum**. The Daily Scrum is a daily meeting that occurs with the development team, usually at the beginning of the day. It is meant to synchronize the development team.

> **DID YOU KNOW?**
>
> The Daily Scrum meeting is also known as a **daily stand up, a daily huddle,** or **morning roll-call**. This lesson will use Scrum terminology.

Daily Scrum is what this meeting is called in Scrum methodologies. However, this type of meeting is common to many other methodologies as well. In Extreme Programming (XP), this meeting is known as the **daily stand-up**. XP is a software development methodology that provides practices for management and development.

Daily Scrums are usually attended by developers, the Scrum Master, and any other interested participants. The **Scrum Master** is the role in Scrum who organizes and facilitates the development team and Product owner (or client). Other interested participants might include developers from other teams, business executives interested in a project update, or Product owners. Product owners are not required to attend every daily Scrum meeting, however.

This meeting usually occurs at the same time and place each day. Often, development teams choose to hold the meeting in front of a task board or a **Kanban board**, so teams can update their progress at the same time as the meeting. Kanban is a scheduling system that organizes production in a logical chain. Task boards are discussed more in the **Sprint Monitoring** lesson of this module.

Importantly, the Daily Scrum is a time-boxed meeting. This means the meeting has a strict time limit that has to be observed.

During the Daily Scrum, meeting participants stand up in a circle and each person answers three questions:

1. What did you accomplish yesterday?
2. What will you do today?
3. Are there any impediments in your way?

To ensure the meeting stays within 15 minutes, often only the people who are substantially committed to the project answer these questions, such as developers and Product owners if they have a major update.

If the conversation during the Daily Scrum starts to go off topic, it is the responsibility of the Scrum Master to cut the conversation short. The "stray" topic is then added to a list of issues to discuss after the meeting is over or at the beginning of the next meeting. In the case of moving the discussion to the end of the Daily Scrum, those involved with the "stray" topic, or who have interest in it, are invited to wait until after the meeting to continue talking This practice helps to keep the Daily Scrum to 15 minutes, while still encouraging open communication. The list of topics to discuss is commonly known as a **parking lot**, a **sidebar**, or is referred to as discussing **offline**.

It is important that even if there is extra time in a meeting before 15 minutes is up, off-topic discussions should still be ended and moved until after the meeting or to the next meeting. Even if there is time in a short meeting, ending off-topic discussions reinforces focused conversations for future Daily Scrums.

The Scrum Master is also responsible for recording impediments brought up in the meeting that were not resolved. The Scrum Master has to make sure these impediments are resolved or that someone is assigned to resolve them.

The end of a meeting should be signaled by some kind of action. This symbolizes a clear end to the meeting and prevents people from lingering or wondering if the meeting is over. Catch phrases or team cheers are actions commonly used to signal the end of a meeting.

It is important to understand that Daily Scrums are synchronization meetings. They are for the benefit of the development team, encouraging communication and commitment from team members to one another. This means that Daily Scrums are not the place for suggesting improvements, nor are they status meetings where members simply report their progress to their superiors. Tracking progress is covered by practices such as Kanban boards or iteration burndowns instead.

In a Daily Scrum, the commitments from team members to each other means that the team should expect goals for the day to be met and reported at the following Daily Scrum. If goals were not accomplished, some impediment should explain why, or the team should offer to help somehow. These kinds of practices reinforce the team nature of Daily Scrum.

Daily Scrums should also be an enthusiastic meeting that makes team members excited to work for the day. Using the practice of Daily Scrum improves team support and a sense of teamwork, encourages work, and open communication. This makes Daily Scrum a good starting point and introductory practice for companies and teams interested in bringing in Agile practices into the workplace.

Common Issues in Daily Scrum

Starting the Meeting and Order of Speakers

There are a number of common issues that can come up in the use of Daily Scrums. The most common occurrence is that no one wants to start the meeting or knows the speaking order. In these cases, the team usually looks for direction from the Scrum Master.

Although the Scrum Master might call the meeting together, it is important that they do not always start it. If the Scrum Master always starts the meeting, Daily Scrums run the risk of becoming status meetings.

A good strategy to use is **last arrival, speaks first**. This means that the last person to arrive at the meeting is responsible for starting it. This strategy encourages punctuality, so it is also good to use if team members are frequently late. However, sometimes the last person to arrive at Daily Scrum is the least prepared to speak, so this strategy can have problems.

Some teams make use of a **token that they pass around** instead. This means that a token such as a ball, plush toy, or some kind of office supply is passed around. The person with the token speaks and then he or she passes the token to another in the circle who then speaks. Participants are more likely to pay attention in a Daily Scrum with this strategy, and it solves picking who speaks next in the meeting.

Another strategy to deal with this issue is the **round robin**. In this technique, one person volunteers to start the meeting and then the meeting goes around the circle from that person in one direction, with each person giving their update where applicable.

Exceeding the Time Box

Another common issue encountered in Daily Scrums is that the meeting exceeds the 15-minute time box.

One strategy, suggested above, is making sure just the "pigs" of the meeting speak. The Scrum Master should also be sure to add off-topic conversations to the parking lot.

Another helpful practice is to make sure the team is actually standing for the meeting. People who are standing are more likely to engage in quicker conversations and they are less likely to hang around after the meeting is done. If meetings still go over 15 minutes, then a modified pass-the-token technique can be helpful. Instead of using a small token, a heavy exercise ball should be used. Holding the ball will encourage participants to be concise and pass the ball to the next person. If necessary, participants can also hold the ball with their arm stretched out in front of them, making it even harder to hold, which encourages even faster meetings.

Lack of Impediments

Sometimes Daily Scrums encounter problems when no one brings forward impediments at the meeting. This is especially problematic if deadlines are missed, but no impediments are listed, as it becomes difficult to know why development is affected.

In order to counter this problem, the Scrum Master can include impediments in their own updates, which may encourage others to discuss theirs. The Scrum Master can also ask questions at meetings, such as "Is there anything that I, or the team, can do to make your day easier?" Remember, the Daily Scrum should not feel like an attack; it should be a team effort.

Team Members' Schedules

In work environments that offer flexible hours, a common problem for Daily Scrum is that not all the team members are present in the morning. In these cases, the Daily Scrum needs to be held at later time in the day, but one that does not make it feel like the start of the day has moved later, which can lead to lost work hours. A good alternative time is after lunch, so the Daily Scrum still feels like it is in front of a block of work.

Becoming a Status Meeting

When holding Daily Scrums, it is important to be careful that the meetings do not begin to turn into status meetings. In status meetings, members just report to the Scrum Master or the Product owner and not to the rest of the team. A good rule to avoid this is to ask the person speaking to not look at the supervisor, so it forces people to look at other team members and engage with them more.

See the **Course Resources** for a list of tips for holding Daily Scrum meetings!

Velocity

The concept of **velocity** has already been described in the course **Agile Planning for Software Products**, but it is reviewed in this lesson as a metric used to manage a project right.

Velocity is the measure of the number of units of work a team completed within a time interval. Software projects commonly use story points completed per sprint as a measure. In fact, velocity should work at the level of the user story and not tasks, which make up user stories.

Velocity can be calculated as an estimate or an actual number. An **estimated velocity** sets a "goal velocity" for the sprint, while the **actual velocity** is determined by how many story points were actually completed in an iteration. In both cases, it is very important that a user story is **done** before it can count towards actual velocity. This means that the work product or task should not only be developed but verified through testing.

Estimating velocities is not always an accurate endeavour. In ideal situations, the same development team has worked on a similar project in the past, so it can use this as a basis for new velocities. However, the development team does not always have such previous experience. In those cases, velocity can also be estimated by taking the user stories that are planned to be completed in a sprint and creating a work breakdown structure (WBS).

A WBS is a representation that takes one large work product or task and breaks it down into smaller, manageable work products or tasks in a hierarchical fashion. Each of those small tasks is easier to assign an estimate, and then those estimates can be added together to make an estimate for the sprint. If the estimated time does not match the amount of time the development team has available, then user stories may need to be added or removed from the sprint. Once all the user stories have been decided upon, their totalled story points will be the estimated velocity for the sprint.

Actual velocities may vary for each sprint as well. This usually happens at the beginning of a project, when the development team is still learning to work together or with new technologies. Over time, however, velocities should stabilize. A development team following the Agile principle of maintaining a sustainable development pace should have a stable velocity.

Actual velocity is also used in the creation of release and iteration burndowns, as explored in the following lessons.

Release Burndown

A **release burndown chart** is a visual monitoring technique common in Scrum. It helps ensure a product is *managed right*.

There are many advantages to using a release burndown chart. Release burndown charts allow an entire development project to be visualized, including:

- How much work the development team has completed
- The work left to finish

- The actual velocity of each sprint
- When the product is expected to be finished

Release burndown charts can also provide early warning that a project is going off plan. However, in spite of providing all this information, release burndown is still a straightforward concept.

Here is a simple example of a release burndown chart:
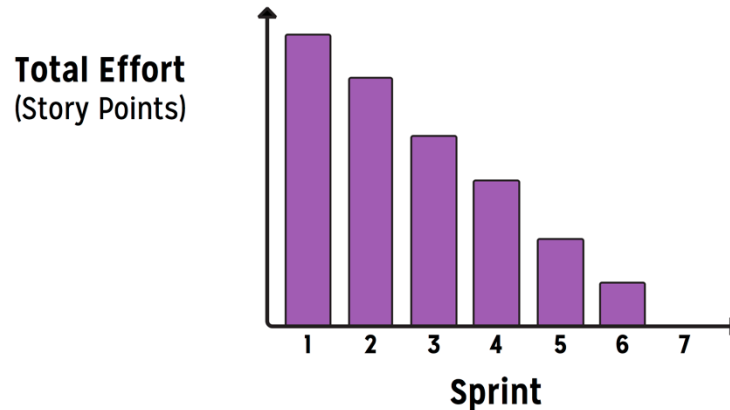
Each bar in a release burndown chart shows the total number story points remaining at the beginning of a sprint. Sprints are listed horizontally—in the example above, there are six sprints. The vertical axis represents total effort. This should be in the measurement the development team has decided to use to size requirements, like hours. In this lesson, story points are used.

Sprint 1, or the first bar, has all the story points remaining to be completed. This is the **total effort**, or the combined total of the story points for all the user stories intended to be completed in the project. As requirements are completed, their story points are removed from the burndown chart, so the next sprint does not include the finished story points. This is illustrated in the example below.

In this example, the total effort for the first sprint was 90 story points. During Sprint 1, 15 story points were finished. These points were taken off in Sprint 2, so that 75 points remain.
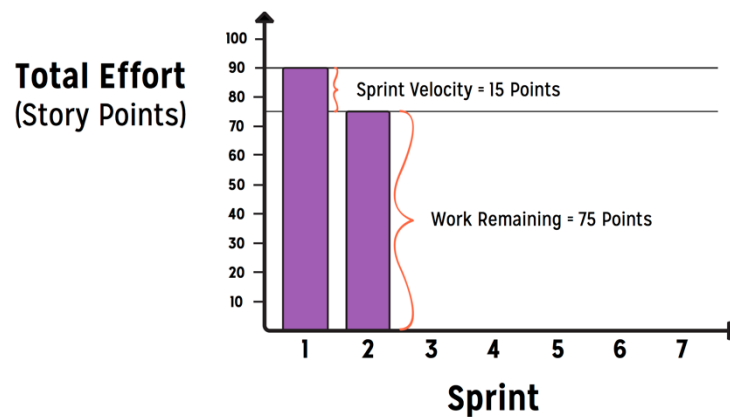
Once two or more sprints are available, release burndown charts can be used to make predictions about how many sprints it will take to finish the project. This is done using a **prediction line**, which is a line drawn through the middle of the top of the bars. This is illustrated on the next page.

This prediction line shows that this project will take six sprints to finish. Even though the line ends at Sprint 7, that sprint actually starts with zero story points (remember the bars represent story points remaining), so Sprint 7 does not need to happen.
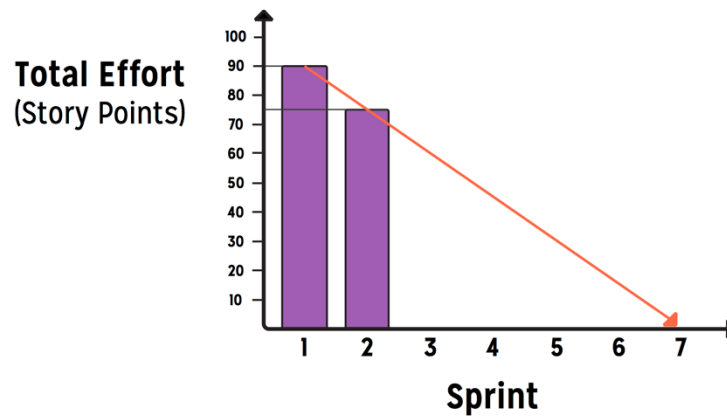
Prediction lines help development teams determine if project development is on track. This is illustrated in the example below:
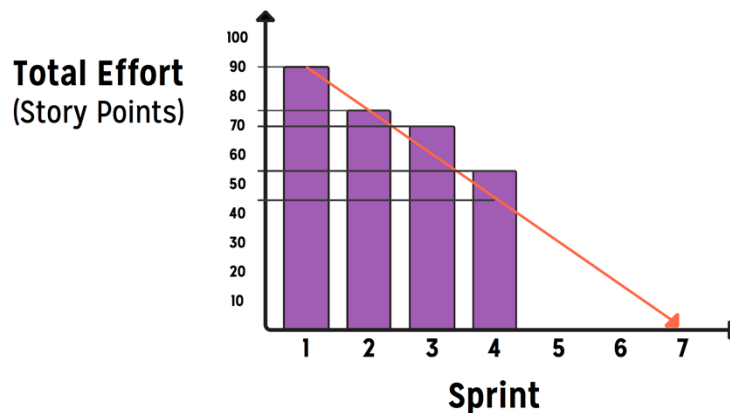
In the example, 15 story points were completed in Sprint 1. A prediction line was generated that suggested that 15 points should be finished in Sprint 2. However, only 5 story points were completed, and Sprint 3 started with 70 points. This is above the prediction line. In Sprint 3, 15 story points were completed again. Velocity was back to its initial value, but the project was still behind schedule at the start of Sprint 4.

If this happens, projects will sometimes hire developers to help get a project back on schedule.
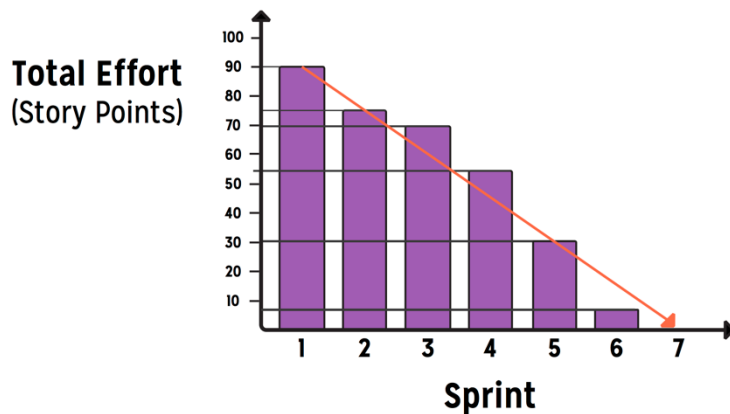
In this example, after extra developers were brought onto the project, 25 story points were finished in Sprint 4. This put the project back on schedule. If the team maintains a velocity of 25 story points per sprint in Sprint 5, then the project will actually be ahead of schedule, as Sprint 6 will begin with only 5 story points, as illustrated above. In this case, extra requirements can be added to meet the prediction line.

> **TIP:** Release burndown charts almost never perfectly follow the prediction line. The first simple example chart in this lesson follows a prediction line perfectly. In real life, however, release burndown charts are more likely to look like the example chart immediately above this tip.

The **definition of done** is very important for release burndown charts, because the release burndown only measures completed story points. In order for story points to be complete in

Agile, which measures progress with working software, the development team must decide upon a definition of done, and only user stories which meet this definition should be considered done.

In cases where many requirements are worked on at once but are not completed over several sprints, release burndown charts will look like the example below.
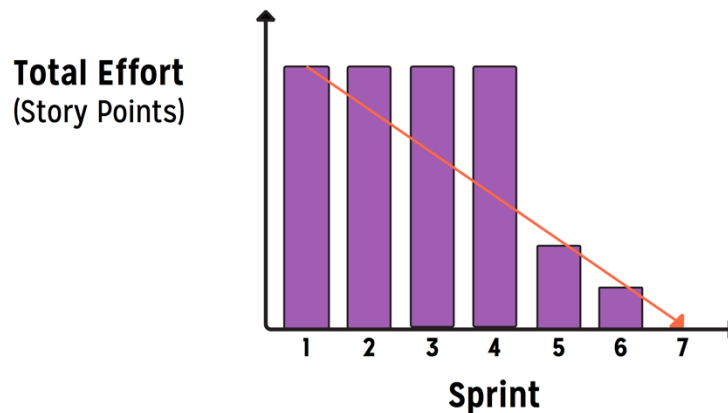
In this example, it took until Sprint 4 for requirements to finally reach the definition of done. At first glance, it might look like nothing was worked on for three sprints, and then a lot suddenly done in the fourth, even though this is not necessarily true. A release burndown chart can therefore inform a team if they are starting too many new tasks before completing old ones, and whether or not tasks are truly being completed. The chart focuses on "done" stories, to depict the real progress.

Changing Requirements

In Agile, change is an expected part of software development. Requirements change constantly, with new ones added, or estimates for story points might change for a requirement. Although it might be tempting to just add story points to the basic bar charts shown in this lesson, this can give a false representation of how much work was done. If 15 story points were finished, but 10 are added, it might look like only 5 story points were completed in a sprint.

There are two ways of showing change in a release burndown chart.

The first shows the total amount of work in each sprint, by adding bars above the work remaining bars. This is illustrated in the example below.
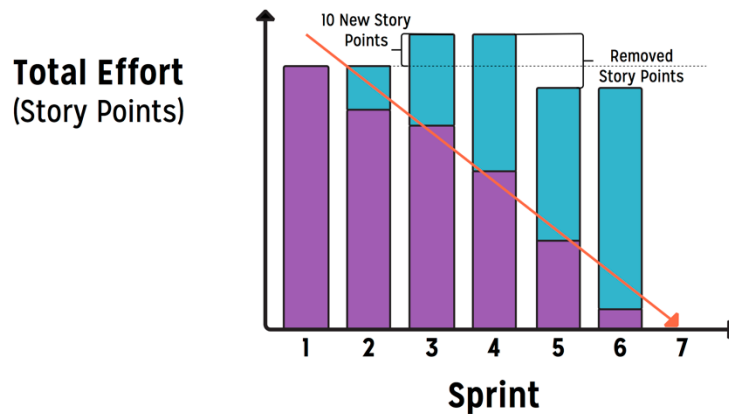
The bottom bar represents the work remaining, as before. However, a bar is also added on the top, representing completed work. In this example, the number of story points in the project has remained the same from Sprint 1 to Sprint 2, and completed story points are represented in turquoise at the start of Sprint 2. In Sprint 3, however, 10 story points were added. This is visible in the chart, because the bar is now taller than the original total effort bar. In Sprint 5, story points were removed, resulting in the sum of the total amount of work remaining and work completed as smaller than both the sprint before and the total effort at the beginning of the project.

This system therefore shows the team's actual velocity, as well as any changes in story points. When calculating actual velocity, it is important to remember to take into account any story points that have been added or removed, as well as the number of remaining story points. This is illustrated in the example on the next page.
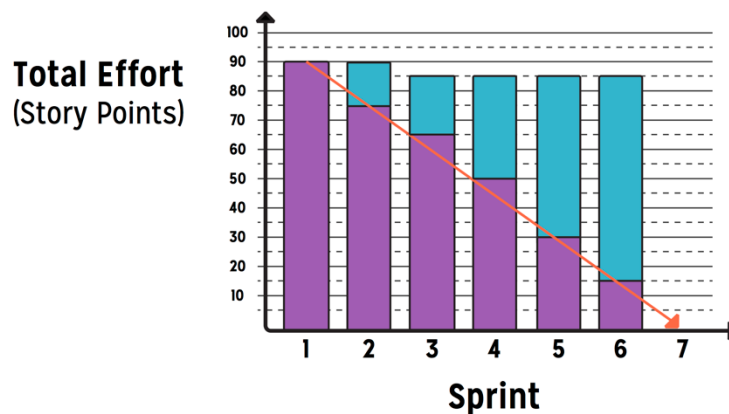
In this example, Sprint 2's velocity is 5 points. The number of remaining story points went from 75 at the beginning of Sprint 2, to 65 at the start of Sprint 3. This makes it seem like 10 story points were completed, and the sprint velocity was 10 story points per sprint. However, the top bars indicated that 5 story points were removed from Sprint 2 to Sprint 3. These points must be subtracted from the 10 story points that seemed to have been completed, making the velocity 5 story points per sprint.

The second way of showing change in story points in release burndown charts is by using an **adjustable floor**. This demonstrates a change in story points on the bottom of the chart, instead of by adding a bar on top. This is illustrated in the example below.
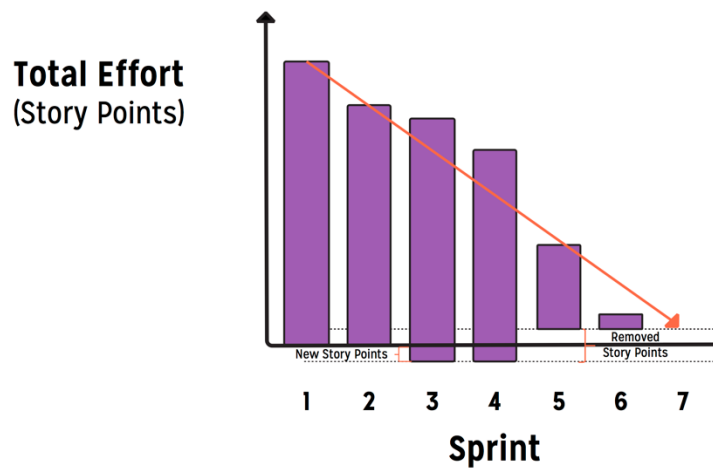
In this example, parallel to the example above with added bars, new story points are added in Sprint 3. The bar in Sprint 3 is extended below the x-axis. In Sprint 5, story points removed, and the bar is then made higher than the previous sprint.

Prediction lines should also be adjusted to align with adjustable floors. This is illustrated in the example below.
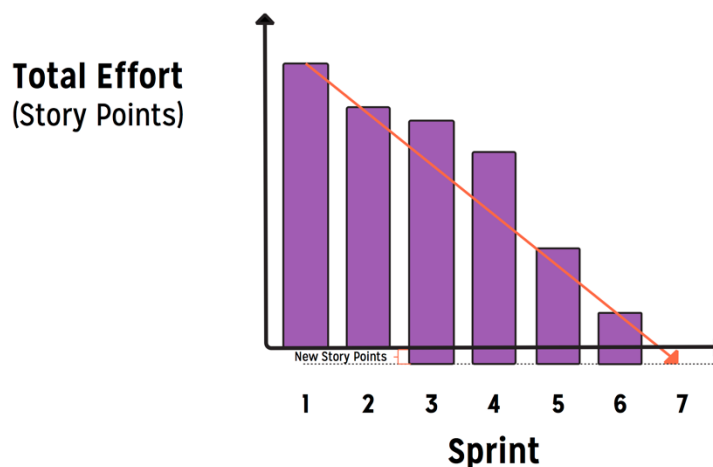
In this example, since the "floor" has moved below the x-axis, the prediction line should also stretch to the same level. This project should take six sprints to finish.

In general, the basic form of release burndown charts is a good tool to for projects that wish to show how much total work is left to complete. It is also used when tracking velocity is not necessarily important to the project. However, in cases where the development team would like to track more information, such as changes in story points, or the actual velocity, using total work done release burndowns or adjustable floor release burndowns is better. However, these

last two techniques are fairly interchangeable, so it is up to team preference on which to use between the two.

Release Burndowns as Line Charts

Sometimes release burndowns are drawn as line charts instead of bar graphs. This is illustrated through examples below.

Simple charts from a bar graph to a line chart will appear as below.
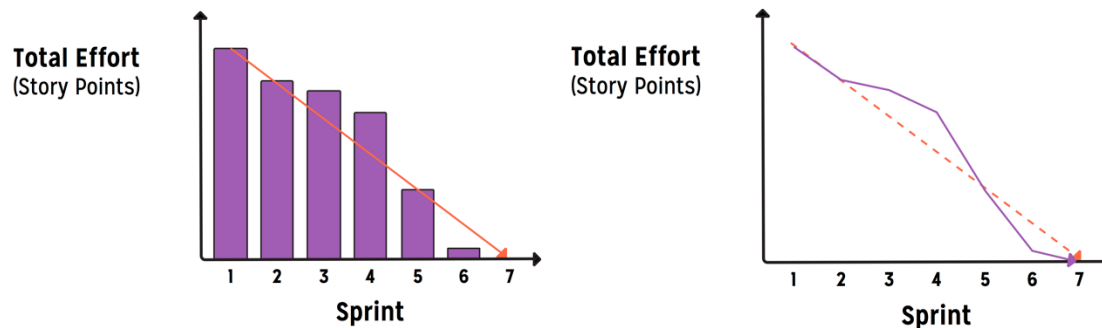
Total work done release burndown charts from a bar graph to a line chart will appear as below.
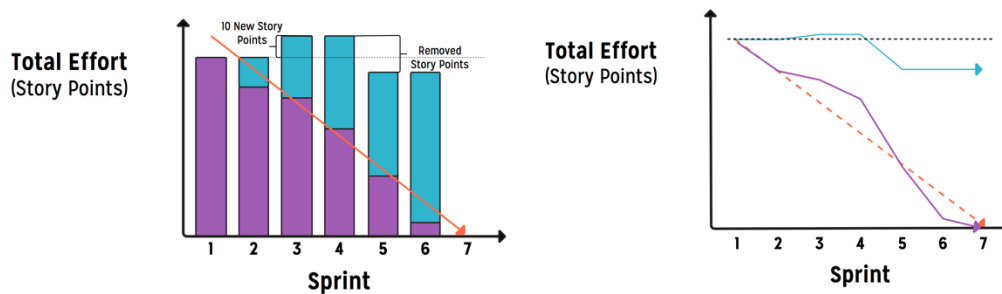
Adjustable floor release burndown charts from a bar graph to a line chart will appear as below.
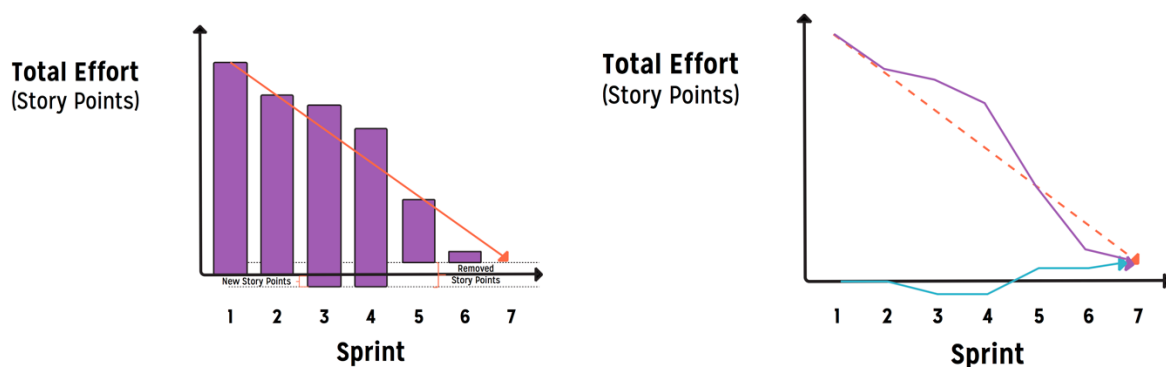
Sprint Monitoring

Burndown charts discussed in the previous lesson were used at a release level. However, they can also be used at an iteration or sprint level. These are known as **iteration burndown charts**. An iteration burndown chart is illustrated in the example below.

Sprint 3

Total Effort
(Hours)

02/22 02/23 02/24 02/25 02/26 02/29 03/01 03/02 03/03 03/04

Date
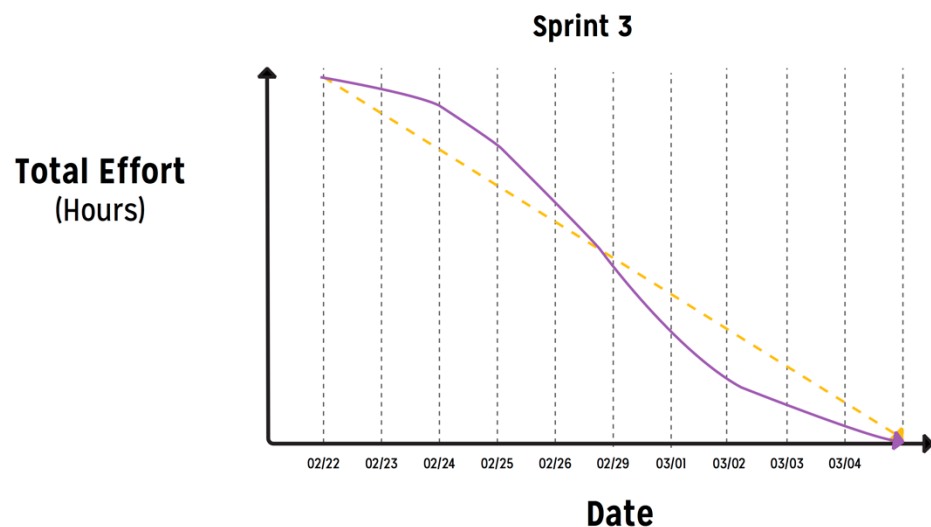
As with release burndown charts, iteration burndown charts have a **prediction line** that demonstrates what sustainable completion of tasks should look like.

Iteration burndown charts can be line graphs or bar graphs, just like release burndowns. It is a matter of preference for which type of chart the team will use.

TIP: It is helpful to consistently use bar graphs for release burndowns, and line charts for iteration burndowns, or vice versa. This makes a chart easily distinguishable as the release level or iteration level at a glance!

Below is the corresponding bar graph for the line graph example above.

Sprint 3
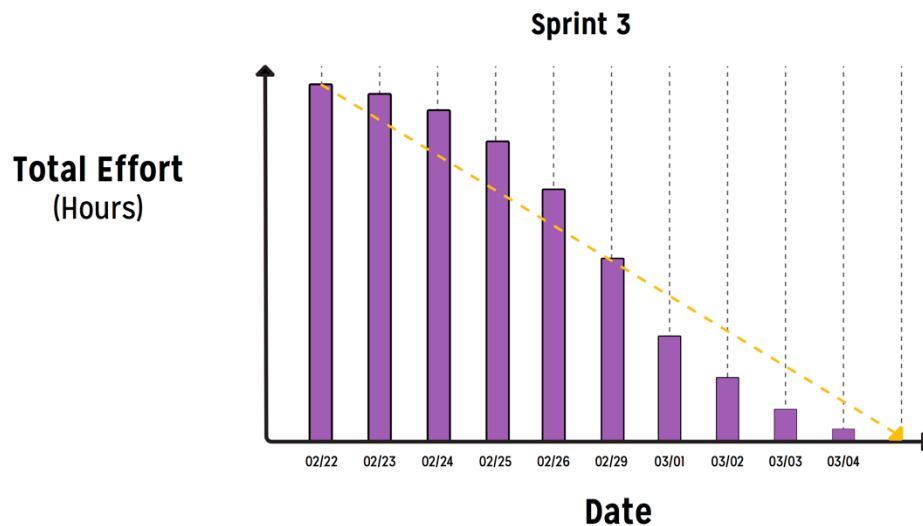
In an iteration burndown chart, the horizontal axis represents days, while the vertical axis represents total effort, usually in hours. An iteration burndown tracks the completion of tasks, instead of user stories and story points as done on the release level. Remember that user stories are made of several tasks. This means the bars or points on the graph represent the total work remaining at the beginning of each day.

The dates listed along the horizontal axis represent the working days of the project. Weekends, holidays, and other days taken off of work are not included, unless work is scheduled on those days. In the example above, you can see that the chart goes from February 26 to February 29. This means no work was done on February 27 or 28, which may have been a weekend. Non-working days are simply skipped in the chart.

Iteration burndown charts should be updated daily. This is typically done during the Daily Scrum, when the previous day's work is plotted as updated in the meeting.

When the iteration burndown chart is updated, this usually corresponds to an update with a **Scrum task board**, also known as a **whiteboard task board**, or a **task board**. A Scrum task board is similar to a Kanban board—it is a means of tracking progress. A Scrum task board is illustrated on the next page.

User Story	To-Do	In Progress	To Verify	Verifying	Done
■	■ ■ ■ ■	■	■	■	■ ■
■	■ ■ ■	■ ■	■	■	■
■	■ ■	■ ■	■	■	
■	■	■ ■	■ ■		■ ■ ■
■	■ ■	■	■ ■	■	■

Task #20

Write Tests for Profile Page

10 Hours

In a Scrum task board, user stores that are intended to be completed are written on post-it notes and placed in the first column, as seen the example above. Then, each user story is broken into developer tasks, and estimates are created for those tasks. These are also written on post-it notes.

Developers will often write their name on a task as well, after they have been assigned the work. This helps the project track who is doing what, and it makes it easier to identify who to talk to about certain tasks.

Tasks can then be placed in a number of different categories on the board, according to what stage the task is in and in the row of the user story they belong in. Task board categories are:

- To-do
- In progress
- To verify
- Verifying
- Done

As tasks are finished according to those stages, they move across the task board into their appropriate categories, similar to a Kanban board. This is illustrated in the example below, with the dotted arrow that shows the movement of one task (or post-it note) across categories.

User Story	To-Do	In Progress	To Verify	Verifying	Done
■	■ ■ ■	■	■	■	■ ■ ■
■	■ ■ ■	■ ■	■	■	■
■	■ ■	■ ■	■	■	
■	■	■ ■	■ ■		■ ■ ■
■	■ ■	■	■ ■	■	■

Once a task is "done," then it is complete and can be "burned" on the iteration burndown chart when it is updated during the Daily Scrum the next day. Hours are "burned" by being deducted from the total effort remaining on the chart. Only tasks that are completed are included in the iteration burndown.

After all the tasks for a user story are in the done column, then the entire user story should meet the definition of done.

A task board is helpful to use alongside an iteration burndown, because it clearly shows what tasks are completed and verified, and can be "burned."

For example, imagine three tasks were finished on day 6 of a sprint.

Task #3 *Josh*
Write Code for Search Feature
15 Hours

Task #15 *Woo*
Write User Manual for Account Creation
3 Hours

Task #10 *Natalie*
Write Tests for the Home Page
7 Hours

Sprint 3
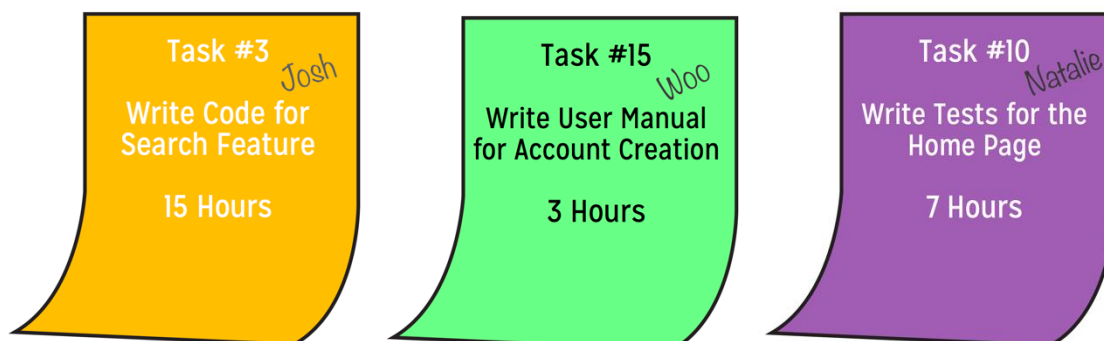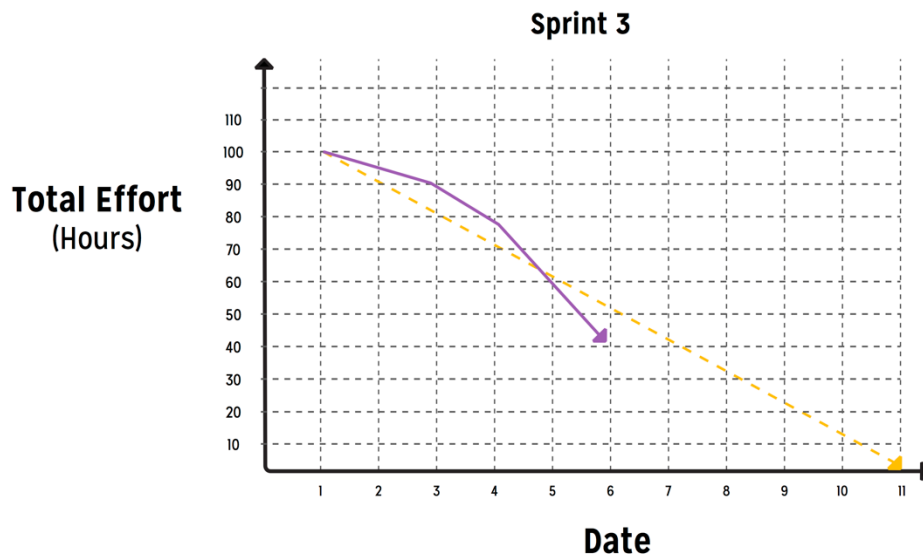
On day 6, the arrow points to 40 hours. Since 25 hours were completed on day 6, the arrow would move to 15 hours on day 7. This is how a task board can help inform an iteration burndown chart.

Adjustable floor charts and total work done iteration burndown charts are likely not necessary in most iteration burndowns, although if desired, these can be easily implemented. Iteration burndowns focus on providing a visualization of the development team's progress within a sprint. They are not used to calculate velocities like release burndowns are, so removing or adding tasks does not have a big impact on information portrayed.

Iteration burndowns can help a team realize quickly (within a day or two) if they are off target, however. This is very useful, as it is better to know mid-sprint than at the end of the sprint that things are not going right. This is primarily realized through two different processes that can show up on an iteration burndown chart: burning up and burning across.

Burning up happens when more tasks are added to a sprint than are completed or removed. This results in remaining work hours increasing over time, instead of decreasing. This is illustrated in the example below.

Sprint 3

Total Effort (Hours)

Date

Burning up usually happens when the team did not do careful iteration planning. User stories were not carefully broken into appropriately sized tasks, or some tasks were overlooked or forgotten. This can be problematic, especially late in a sprint, since it means that tasks may not be completed. One of the best strategies to deal with this issue is to remove or prioritize some tasks. This helps ensure that tasks are not half done at the end of the sprint.

Burning across happens when the team is not completing tasks. This results in remaining work hours staying consistent over time, instead of decreasing. This is illustrated in the example below.

Sprint 3
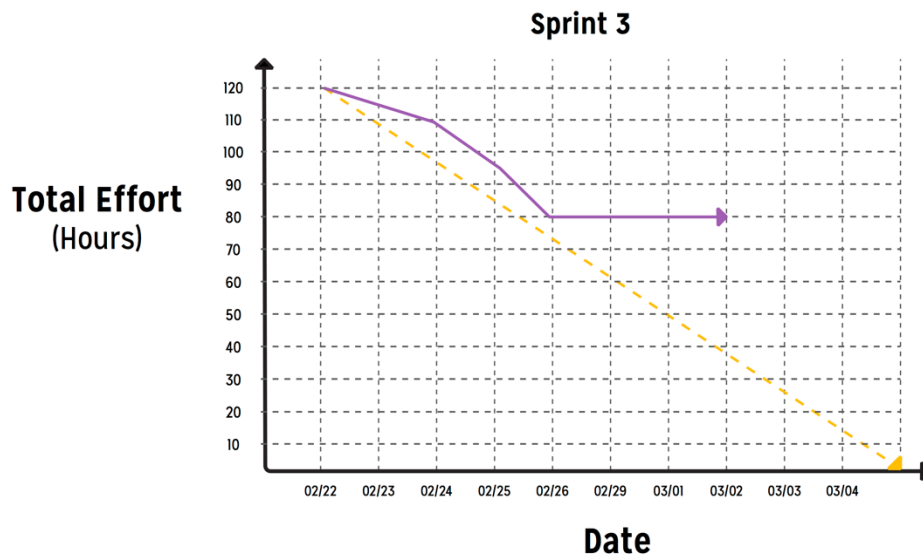
Burning across usually happens when the team is:

- Adding as many tasks as it is completing or removing.
- Not working towards the full completion of tasks, although they may still be working.
- Not working at all.

The second scenario is the most common. One of the best strategies to deal with this is to have a meeting with the development team, where emphasis is placed on completing tasks before starting new ones. It should also be suggested that team members with free time should help someone else finish a task, instead of starting something new. This helps avoid having half-done user stories at the end of the sprint, which cannot count towards burndown.

Iteration burndown charts are useful tools for assessing quickly what might be going wrong in a project and how to fix it. They ensure a project is *managed right* on a day-to-day basis.

Module 4: Project Retrospectives

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

- Explain the terms: retrospective, postmortem, and lessons learned.
- Summarize what a retrospective is used for.
- Recognize that retrospectives address both what went wrong and right.
- Explain what a safe environment is and how to create it.
- Explain the difference between a functional and non-functional culture.
- Describe the role and benefit of an outside facilitator.
- Summarize how to prepare a retrospective.
- Identify what happens at the beginning, middle, and end of a retrospective, and list steps of a retrospective meeting, including the readying course, past course, and future course.
- Explain examples of retrospective exercises in a project retrospective, at the readying course, past course, and future course.
- Identify good questions an outside facilitator could ask in retrospective exercises.

What are Retrospectives?

Retrospective is a term that describes the act of reflecting on work done, or not done, over the course of the project. This reflection examines what went well and what did not go well in the work. Usually, retrospectives lead to a set of improvements.

This module covers two different types of retrospectives:

- **sprint retrospectives**
- **project retrospectives**

Sprint retrospectives have been touched upon before in the course **Agile Planning for Software Projects** with respect to time boxing and iteration planning. Both those practices involved breaking up time into small, manageable pieces in which work can be accomplished.

In Agile methodologies, it is recommended to have a planned self-appraisal session at the end of those time-boxed iterations. This self-appraisal allows insights to be applied to the following iteration or sprint, allowing the project to improve. This self-appraisal is the sprint retrospective.

Project retrospectives, also known as **postmortems**, are team reviews of the entire project after it is complete, instead of just a review of a time-boxed sprint. This means that although project retrospectives also examine what went well and what did not, as well as identify how the project could be improved, there are no "next sprints" to apply these **lessons learned** from a retrospective. Lessons learned are insights the development team has discovered through reflecting on the work done and how it could be done better the next time.

Project retrospectives can be applied to future work that people on the development team might participate in, whether together on the same time or not.

Retrospectives also allow the development team to address any personality differences, disagreements, or compromises made over the course of the work. If such issues are discussed

in a collaborative, open atmosphere, without judgment, then teams can release energy and mend any differences.

The members of development teams are responsible for selecting what is important to discuss and what should be focused on in a retrospective, so the team feels they contribute to their own improvement. This increases motivation without external forces pushing the team to improve.

Sometimes, retrospectives are avoided. In such cases, the development team is likely to go ahead and start planning the next sprint and then jump directly into development. However, without retrospectives, there is little sense of what is going well or not well in a project. Further, even if team members voice concerns without retrospectives, there is no official way to capture these feelings about the project and channel them into structured change in a project (or not, if the team decides upon reflection that things are better as is). Retrospectives should always be encouraged, because there is always room for improvement in a project.

Retrospective Issues

There are a number of issues a development team might face regarding retrospectives.

Often, a team may decide to skip retrospectives. This can be for many reasons, including:

- lack of time
- a desire to look ahead instead of reflect on the past
- some people may not like to speak about themselves
- some people may be burned out and wish to move on from the project

Sometimes, retrospectives are carried out for the wrong reasons, like being required to do one for an audit.

Feeling Safe

The most common reason retrospectives are not implemented properly, however, is that the development team does not feel safe enough to participate in reflective discussion, either emotionally or professionally. Feeling safe refers to a person feeling as if they can express themselves in a genuine and truthful manner without fearing a negative response from peers or management. In other words, people feel safe when they can communicate openly without judgment and if they are supported regardless of what they have to say.

For example, imagine a developer named Pierre is working on a software development project with a team. For the past few days he has been unproductive due to burn out. In an environment where he feels safe, he can express this openly in the Daily Scrum. On the other hand, in an environment where he feels unsafe, Pierre might try to embellish his work to make himself seem more productive, because he is not comfortable expressing himself.

As the example above helps illustrate, it is important that team members feel safe. If they do not, important issues may be ignored and cannot be resolved. In Pierre's case, if no one knows he is burned out, then it will be hard to help him. Pierre will remain unhappy, and his productivity may lower still.

Here are some major techniques software project managers can use to help create safe environments:

- Let the team know that the retrospective happens in an open environment, where new ideas and perspectives are encouraged.
- Show appreciation for contributions made by team members. Feedback in the form of appreciation lets team members know their input is valued. Often, when no feedback is given, people will assume input is not appreciated. Project managers should remember that this doesn't mean they have to agree with the contribution, but they should still acknowledge that contributions are valuable. There are several methods managers can use to show appreciation by:

  - telling each person directly their input is appreciated. However, if not done well, this method can seem insincere.
  - encouraging feedback when the manager themselves offer perspectives and ideas. This helps team members feel comfortable giving feedback in general and to other members.
  - building or expanding on another team member's idea, and then attributing the original idea back to the person who thought of it. This provides validity to the idea and affirms the person who thought of it.

- Show positive leadership. This means the manager should be a role model for the team by showing appreciation and having a positive attitude. As mentioned before, managers should always show team members high levels of respect even if they do not agree with the idea presented.

Returning to the example above, if Pierre reveals he has not been productive lately, the manager should not chastise him, or the environment may no longer feel safe. Instead, the project manager should express appreciation towards Pierre, especially since he is being brave. They could also suggest that Pierre take some time away from his current problem to refresh his mind, perhaps by trading work with someone else, or perhaps he could take some time off. In this example, it would also be helpful if the manager complimented Pierre on the work he has done so far and expressed an interest in hearing how things turn out over the next few days. These practices contribute to a safe environment. In such an environment, Pierre is more likely to be honest about future impediments and to be a more productive team member overall.

> **TIP:** In cases where team members do not feel safe in front of a manager, the manager can help change this by avoiding criticizing developers' work in front of the group, or by not attending future retrospectives. Remember, retrospectives are not about criticism; they are used to reflect on the work the team has done and to determine what changes should be made.

Functional and Dysfunctional Team Cultures

Retrospective meetings are also facilitated by team culture. Team cultures can either be **functional** or **dysfunctional**.

A functional team culture is similar to a safe environment, except it is an environment where people actively contribute instead of just feeling they can. In a functional culture, everyone tries

to make everyone else look good, and the ownership of work is shared among the team. In this kind of culture, there are no secrets or guarded language, as success belongs to everyone, not just an individual. Meetings are constructive.

In functional team cultures, the productivity and decision-making of the retrospective meeting is driven by the team members and not by managers. This is because the team is motivated and takes ownership of the project.

A dysfunctional team culture, on the other hand, results when more pressure is placed on producing instead of improving. As a result, team members may become secretive and competitive in an unhealthy way in order to "look good" to management.

Decision-making becomes driven by self-interest instead of what's best for the team. This can lead to a decrease in productivity.

Outside facilitators are sometimes brought in to help teams collaborate effectively. Outside facilitators are people who guide the retrospective process by creating a safe and trusting environment for discussions. They are also responsible for ensuring that all team members, including managers, contribute equally in meetings and do not avoid tough issues. Outside facilitators should be well acquainted with the goals of the retrospective.

It is good practice to do a retrospective at every major junction in a project, as they can help developers see a project objectively, so that improvements can be made.

Good retrospectives are:

- organized and motivated by the development team
- driven by objective data, such as velocity or defect analysis
- done regularly by the development team
- a safe place for team members to mend feelings and build trusting relationships in the team

> **TIP:** It is important to remember that team members do not usually start on projects to ruin them. Instead, they usually want the project to do well. Blaming problems on developers is likely inaccurate, and it is more productive to reflect on why things might have gone badly. This helps keep observations objective and improvements easier to identify.

In-Depth Sprint Retrospective

This lesson outlines an example of a successful sprint retrospective. A sprint retrospective should be the last thing that occurs in a sprint. It usually happens after a **sprint review meeting**, which demos the product to the client or Product owner—it is a review of the product itself. As outlined before in this module, a **sprint retrospective meeting** is where the process is reviewed, so what is working in the product and what can be improved can be identified. This is keeping within Agile methodologies, which requires both product and process to be reviewed and adapted.

To that end, sprint retrospective meetings should focus on issues about the process of the work, such as underestimated tasks in the last sprint, highlighting good work, or the implementation of an iteration burndown chart. Issues related to the product or features should not be discussed in a sprint retrospective meeting.

Sprint retrospectives should be time-boxed meetings set for an hour. However, these meetings do not need to be as strict as other Scrum events that are time boxed. If issues arise that should be discussed, the meeting may go over an hour.

Three major questions are addressed in a sprint retrospective meeting. These are:

1) What went well this sprint?
2) What did not go well this sprint?
3) What could be improved for the next sprint?

This is the best order to ask the questions. It allows the meeting to start on a positive note, and it gives the Scrum Master an opportunity to show their appreciation and recognition of the development and their work. Then, the second question allows criticism, but this should be facilitated by the Scrum Master so that team members do not feel attacked. The meeting should be honest but respectful. Finishing with the third question allows the team to feel constructive. The team should not finish with negative feelings, but instead should have something to look forward to, such as improvements for the next sprint. Using this order is especially important if the sprint did not go well. It prevents the team from starting the next sprint with negative feelings.

Sprint retrospectives are excellent opportunities to help the development avoid larger issues down the line.

Retrospective Exercises

Project retrospective meetings address the entire project instead of only a sprint. However, like sprint retrospectives, project retrospectives hope to discover what went well in a project, what did not go well, and how to improve future work. This means that what specific features do is not the focus of retrospectives. Instead, how features are implemented and the success of those features is addressed. For example, analyzing the fact that users can customize their avatar in a gaming app is not the purpose of a retrospective. But, how the team created this feature, such as how the team worked together to create the code can be examined. Because of this broader scope, project retrospectives tend to take quite a bit longer than an hour to discuss. Remember, some projects last months or even years, so there may be many issues to reflect upon and address.

This lesson is largely based on the book *Project Retrospectives – A Handbook for Team Reviews* (2001) by Norman L. Kerth (see the **Course Resources**). In his book, Kerth (2001) suggests project retrospective meetings should take three days. Although this seems like quite a long period of time, it is likely the three days will yield many ways time can be saved in future projects. The meeting should therefore save more than three days of time in the future. It is an investment.

Kerth (2001) compares the three-day project retrospective to a three-course meal, where each day corresponds to a course.

- The appetizer is the **readying course**. In this course, ground rules are set out for the retrospective. The purpose of the readying course is to build up trust among team members and to help the team realize what they have accomplished.
- The main course is the **past course**. During this course, the team reviews everything that occurred during the lifecycle of the product. It is also an opportunity to share all aspects of the project with the team, as some members may have been specialized and unaware of certain aspects. This may also help identify root causes of any issues.
- The dessert is the **future course**. This course allows the team to look forward to future work and to suggest ideas for improvement.

Although three days is the suggested meeting length, it is important that courses are not extended just to fill the time. If all issues are addressed, then it is possible to move forward to the next course.

All of these events require preparation. This lesson shares a number of exercises a retrospective facilitator could use for each of the courses to help the meetings go as smoothly as possible. These exercises are generally implemented in the appropriate meetings. However, not all exercises need to be executed in every project retrospective.

Teams should be able to assess which exercises would be most beneficial to their projects. In all cases, safe environments should be maintained.

If you are interested in more information on the retrospective exercises outlined here, you can read more about them in Kerth's book.

Preparation Exercises

Before any of the retrospective courses happen, sometimes a **retrospective pre-work handout** is used. A retrospective pre-work handout is a handout that is emailed to everyone invited to participate in the retrospect. It contains a list of questions that can help identify general patterns that might arise during the meeting.

Once participants have a copy of the questions, they are encouraged to write down answers before attending meetings. If people write things down, they are more likely to share them. A retrospective pre-work handout also encourages participants to think about issues before the meetings begin.

Kerth suggests the following questions:

- What topics should be discussed in the retrospective, in order to make sure we learned the most from this experience?
- What do you hope will happen during the retrospective?
- What long-term impact do you hope this retrospective will have?
- What reservations, concerns, or worries do you have about this retrospective?
- What emotions do you feel as you think about this meeting?
- What else should I ask, and how would you respond?

Once team members have responded, their answers go to the facilitator, usually by email. Responses should remain private and are only used to help the facilitator identify general trends that may need to be discussed.

Outside facilitators may want to introduce themselves to the team beforehand. This helps establish a basis of trust and gives facilitators an opportunity to discuss the project in person with people. This can help them further identify trends with the pre-work handout. Facilitators can also use this opportunity to remind people to complete the pre-work handout and to bring artifacts to meetings (artifacts will be discussed later in this lesson).

Readying Course Exercises

Readying course exercises focus on establishing trust and designing ground rules that create safe environments. They should also convince the team that the retrospective meeting will result in positive outcomes.

Exercise	Description
Introduction Exercise	**Introduction exercises** are 30-minute exercises, which occur at the beginning of the retrospective. During this exercise, the facilitator should make sure everyone on the team knows each other, and a general schedule and agenda should be set for the retrospective, with the understanding that this might change as the meeting progresses. Introduction exercises often begin by asking the development team to define the word "wisdom." This usually leads the discussion towards learning from experience, which is the purpose of a retrospective. The meeting is a safe environment where no one is blamed; this should also be reinforced in this exercise.
"Defining Success" Exercise	The **"defining success" exercise** seeks to define what the development team considers success. Drawing from this, the team can then assess whether or not they believe the project was successful, or what could have been done to make the project a success. These exercises should take between 30 minutes and an hour. A good question to begin "defining success" exercises with is: "Was this project a success?" Answers are then recorded. The focus should then shift to what the team accomplished. Even if the project failed, there is a lot of the team should be proud of in their work. The following definition of success can then be proposed, **A successful project is one on which everybody says, "I wish we could do that over again – the very same way."** Using this standard, the team can then focus on the successes of the project, while also determining how it could have been improved.
"Creating Safety" Exercise	The **"creating safety" exercise** seeks to make sure everyone on the team feels safe enough to share their feelings about the project. This exercise can be skipped if the development team is already very comfortable with each other. There are many ways to create safety. First, the team should be informed that retrospective exercises are optional. If people are forced to participate, they may say what they think they should say, instead of what they actually think. A secret-ballot poll can also be used to determine how safe the team feels. Team members write a number from a scale between 1 and 5 on a ballot, where 1 means they do not feel safe enough to share feelings, and 5 means they feel safe enough to

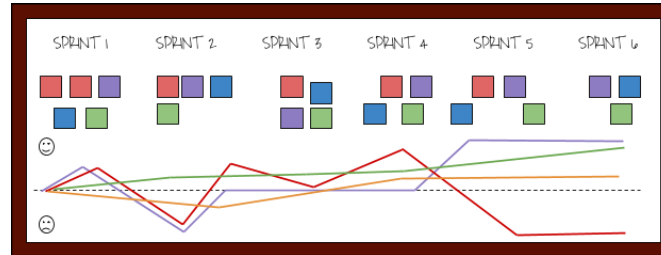|	
	share anything. If each team member indicates a 3 or above, then the meeting can move forward. If not, then the facilitator should focus on other ways to make the team feel safe.
	One way to do this is by establishing ground rules for the retrospective, which the safety exercise should also do. The team should establish rules, as this gives them control and makes them feel more safe. If the following rules were not established in the discussion, they should be added: • **all participation in the retrospective is optional** • **jokes will not be made about anyone in the room** • **ground rules can be amended after any break** It is important that team members do not make jokes about each other, as it can be sometimes difficult to tell if the joke is playful or hurtful. Jokes can also discourage people from sharing, if they are worried about being teased.
"Artifacts Contest" Exercise	The **"artifacts contest" exercise** is one where participants bring any and all artifacts related to the project, so they might be discussed. Artifacts are any object that someone on the team believed was important or significant, and can truly be any kind of object. For example, memos, schedules, burndown charts, random post-it notes, mementos, pictures, sketches of the user interface, employee contracts, small trophies or gifts someone was given after a major problem was solved, a movie that inspired the product, etc., could be considered artifacts. This exercise should take one to two hours, although this depends on the number of team members and the number of artifacts they choose to bring. Participants should bring as many artifacts as possible to this exercise. Each person should explain the artifacts they brought, and their importance. Facilitators should encourage questions about artifacts, as this helps engage the team. After artifacts are presented, the team usually votes on them according to the following categories: largest collection, most creative artifact, most unusual artifact, and most important artifact. Small prizes should be given to the winners. This exercise helps the team remember good work and memories associated with the project. It also encourages the team to reflect on the entire project, not just the final push.

Past Course Exercises

There are a number of past course exercises used as well.

Exercise	Description
"Create a Timeline" Exercise	The **"create a timeline" exercise** is one where the development team creates a timeline for all the significant events of the project. This allows the team to reflect on the whole project. To create the timeline, a wall or whiteboard is cleared and covered in paper. People then use index cards or post-it notes to place significant events along the timeline. A good place to start for team members is by asking them to place a note on the timeline for when they joined the project. If some team members left the project before the end, then they can also place their end date cards on the timeline. From there, significant events can be filled in. As this is an inclusive process, any event someone thinks is significant should be added to the timeline. In the end, a visual representation of the entire project should be the result of the exercise.

"Emotions Seismograph" Exercise	An **"emotions seismograph" exercise** seeks to explore events that trigger emotion among team members. In this exercise, participants simply draw a line below the timeline, reflecting their emotions at that time. This helps bring meaning to the project, and identify significant events
"Mine for Gold" Exercise	The **"mine for gold" exercise** usually follows the "create a timeline" exercise. In the "mine for gold" exercise, the team examines the timeline in periods chronologically. For each period, the significant events are reviewed. Ideally, this exercise takes between five and eight hours, over two days, because it is so in-depth. As eight-hour chunks can be quite long, spreading those hours over two days is preferred. The extra day also allows team members more time for reflection. Five whiteboards or pieces of chart paper are created in this exercise. They are labelled with the topics: **What worked well that we don't want to forget** **What we learned** **What we should do differently next time** **What still puzzles us** **What we need to discuss in more detail** As each period of time is discussed, items can be added to these topics. Good questions to help review time periods include: **What card is most significant?** **What card surprises you?** **What cards don't you understand?** **Do you see any patterns emerging?** **What card should we discuss next?** **What card still needs discussion?**
"Offer Appreciations" Exercise	An **"offer appreciations" exercise** allows team members to express appreciations for each other and to mend relationships. It is also a useful exercise for building team morale. It is an exercise that can be conducted as many times as necessary, and at any point in the retrospective meeting. To conduct this exercise, the team members should stand in a circle, and one person starts by offering an appreciation to someone else. The person who received the appreciation then offers an appreciation for another team member. This cycle continues until everyone has received an appreciation. Appreciations should be worded like, "Pierre, I appreciate you for teaching me Java," instead of "I appreciate Pierre for teaching me Java". The phrase "I appreciate you for" is more personable than the alternative.

Future Course Exercises

A couple of future course exercises are outlined in the following table.

Exercise	Description
"Making the Magic Happen" Exercise	The **"making the magic happen" exercise** helps team members talk about sensitive subjects that might otherwise be avoided. A good opening phrase facilitators can use to start this conversation is, "This retrospective is coming to an end. This would be a good time to bring up anything that you were hoping we would get to but haven't." Facilitators can also reference or bring up issues that they have heard hinted at throughout the retrospective. A good phrase to use in that case is, "We have spent days talking about everything except one really important issue." In general, once the facilitator invites difficult conversation, people will tend to share what they have been avoiding previously. If conversation becomes aggressive and hurtful, the facilitator must bring it back to one of respect. This can be done by reviewing the guidelines established at the beginning of the meeting, or asking how certain actions made the aggressor feel, or why they acted a certain way.
"Closing the Retrospective" Exercise	**"Closing the retrospective" exercise** ties up loose ends and wraps up the retrospective. In this exercise, team members use a pen and paper to write down their hope or wish for what will happen after the retrospective on a card. This is meant to be shared anonymously with the team. Cards are then shuffled, and one is passed to each team member. Members then read the card. After, cards are passed around the team until everyone has read each card. Once all the cards have been read, the meeting should be concluded with a positive statement, such as, "I think we made a lot of progress over the last few days. I think a lot of these hopes and wishes will become true. Now it's up to you. Make it happen." Additionally, a final phrase such as "This retrospective is officially over" should conclude the meeting.