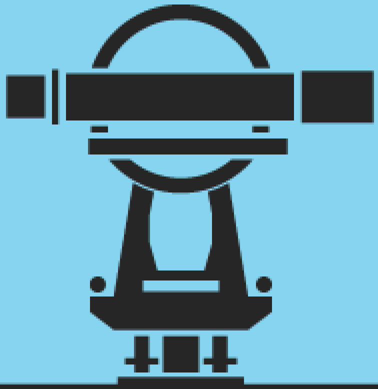

Agile Software Development Succinctly

by Stephen Haunts

Syncfusion | Technology Resource Portal

Agile Software Development Succinctly

By

Stephen Haunts

Foreword by Daniel Jebaraj

Technical Reviewer: Peter Shaw
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Stephen Haunts has been developing software and applications professionally since 1996, and as a hobby since he was 10 years old. Stephen has worked across many different industries including computer gaming, online banking, retail finance, healthcare, and pharmaceuticals. Stephen started out programming in BASIC on machines such as the Dragon 32, Vic 20, and the Amiga, and moved onto C and C++ on the IBM PC. He has been developing software in C# and the .NET framework since first being introduced to it in 2003.

In addition to being an accomplished software developer, Stephen is an experienced development leader and has led, mentored, and coached teams to deliver many high-value, high-impact solutions in finance and healthcare.

Outside of Stephen's day job, he runs a popular blog called "Coding in the Trenches" at www.stephenhaunts.com, and he is a training course author for the popular online training company Pluralsight. Stephen also runs several open-source projects including SafePad, Text Shredder, Block Encrpytor, and Smoke Tester—the post-deployment testing tool.

Stephen is also an accomplished electronic musician and sound designer.

Introduction

Who Is This Book For?

This book will appeal to many different audiences. If you are a developer, then this book will give you a good understanding of why Agile is beneficial to you, your team, and your employer. This might be the first Agile project that you've worked on, and you want to understand why you're using Agile over Waterfall. This book will also be a good refresher on why you're using Agile if you're already on an Agile project.

If you're a project manager, then this book will help you understand the difference between an Agile project and the more traditional Waterfall project. As teams become more self-directed when working on a project, a project manager is still crucial to help ensure the teams are run correctly and deliver on time and budget.

If you're an IT or business leader and your company is considering adopting Agile, this book will help you understand how this will work and what the benefits are to your organization. This book covers six main areas:

- Waterfall development and its problems
- What is Agile all about?
- Common Agile misconceptions
- Advantages and disadvantages
- Extreme Programming (XP)
- Scrum

Chapter 1 Waterfall Development and its Problems

History of the Waterfall Model

The Waterfall software development process was introduced computer scientist Winston Royce in 1970. Royce first wrote about Waterfall in an article called, Managing the Development of Large Software Systems. Although Royce didn't directly refer to his model as Waterfall, the article was actually about a process that was flawed for software development. Royce's model allowed for more repetition between stages of the model, which Waterfall doesn't allow you to do.

Royce's actual model was more iterative in how it worked and allowed more room to maneuver between stages. We will discuss a more iterative way of working when we discuss Agile later on in this book. Although Royce didn't refer to his model as the Waterfall model directly, he is credited with the first description of what we refer to as the Waterfall model.

Royce's original article consists of the following stages, which we'll go into more detail on in a moment. Those stages are:

- Requirements Specification
- Detail Design
- Construction, where developers start crafting code
- Integration, where all the code is brought together and compiled into a run-able solution
- Testing and Debugging, where your testing will try to find defects that the developers will fix
- Installation, where you deploy your system so that it can be used by your end users.
- Maintenance, where you fix any issues that are raised by the users.

How Does Waterfall Work?

The Waterfall process is split into separate stages, where the outcome of one stage is the input for the next stage. In the first stage, Requirements Specification, all possible requirements for the system to be developed are captured and documented in a requirement specification document. This document normally requires sign-off by key project and business stakeholders.

This part of the Waterfall model is typically organized by the business analysts, but depending on the size of your project, team, or organization, other members of your development team may be involved. This stage is about teasing out the requirements of the system from your stakeholders. This would include the required functionality, documentation of business rules and processes, and capturing any regulatory and compliance requirements that will affect the overall system.

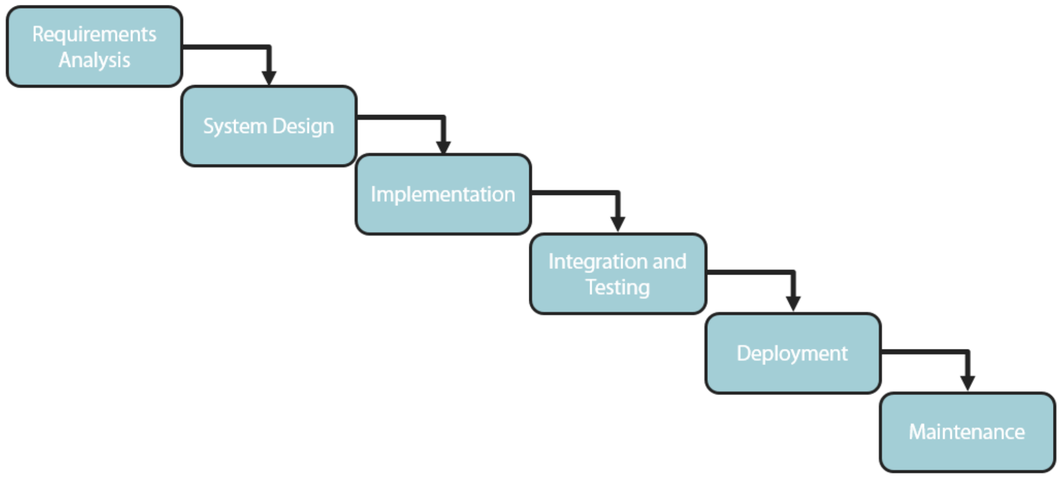

Figure 1 The Waterfall software development process

The next stage is System Design. The requirement specifications from the first stage are inspected, and the system design is put together. This design helps in specifying the system design requirements, and also helps with designing the overall system's architecture. It is this stage where architects, solution designers, and developers will work together to decide how the overall system will be constructed. This is from a code perspective, and also a technology choice and infrastructure perspective.

The next phase is Implementation. This is the phase where the developers take the design and start producing code to turn the design into a reality. The developers may also write automated unit and integration tests at this stage.

After the Implementation phase, we have the Integration and Testing phase. This is where all the deliverables from the implementation phase are brought together and tested as a whole. The testing team should be working to a defined test plan. Once the system has been tested and signed off by the test team, the next stage is deploying the solution to your end users. Your end users may be internal customers within your organization, or customers.

Once the solution has been deployed, it goes into the Maintenance phase, where any issues that are reported will need fixing and re-deploying. This would generally be in the form of release patch fixes to your system. You may also perform small enhancements to the system at this phase. If an enhancement is quite large in scope, then you might start the Waterfall process again and start capturing further requirements.

All of these phases are cascaded, where progress is seen as flowing steadily downwards like a waterfall. The next phase is started only after a pre-defined set of goals are achieved from the previous phase. In this model, the phases do not overlap.

Where Is Waterfall Suitable?

Every software development project is different and requires a suitable software development lifecycle approach that is employed based on your team and organization's internal and external factors. There are some project situations where the Waterfall model is appropriate, but as we look at these, you may feel from experience that this doesn't always work out.

First, Waterfall is suitable if your requirements are well documented, clear, and fixed. How often is that the case though? From my own experience as a software developer, I can't remember any of the many projects I've delivered where the requirements have been clear from the start so that they can be captured in a document that doesn't change as the project rolls on.

Next, the product definition must be stable. Again, I can't think of a single project where this has been the case, as external factors like a change in the marketplace or a shift in business priorities mean that your product will change over time. I have worked on many projects where the final delivered product was quite different from what was initially specified. Under Waterfall this shouldn't happen, but in reality, what you are building can change.

There is nothing wrong with this, but it does fight against the software delivery process. Hold this thought in your mind, as this will form a big part of our core theme when we discuss Agile in detail later on.

Next, the technology should be well understood. This means that developers should understand the technologies that they're going to be using and how they work. Once you enter the implementation and construction phase of the project, developers normally have to work toward very rigid and set-time scales. In my experience, a lot of effort is expelled on the requirements and design phases, which normally eat into the time needed to actually develop the code.

Next, Waterfall works best on projects that are short, and by short I mean projects that take around two-to-four months in total. The longer a project runs, the more chance there is of the requirements and product definition becoming out-of-date.

Finally, Waterfall works best when all the members of your project team are available. It is quite normal for a development team to have a pool of resources that might be shared out between many different projects. If another project is overrun for any reason, you may not have all your people available at the time when they are required. This can greatly impact a project's time scale and put delivery dates at risk.

Advantages and Disadvantages of Waterfall

In a minute, we'll take a look at a number of pros and cons of the Waterfall model. But before we do, I first want to cover some of the main high-level advantages and disadvantages to this development process.

The first advantage is that by splitting your project deliveries into different stages, it is easier to maintain control over the development process. This makes it much easier for schedules to be planned out in advance, making the project manager's life much easier. It’s for this reason I've found that experienced project managers tend to favor the Waterfall process. By splitting a project down into the various phases of the Waterfall process, you can easily departmentalize the delivery of your project, meaning that you can assign different roles to different departments and give them a clear list of deliverables and time scales. If any of these departments can't deliver on time for various reasons, it’s easier for a project manager to adjust the overall plan.

Unfortunately, in reality I've seen a plan adjusted where the implementation phase gets squeezed more and more, which means the development team has less time to deliver a working solution. Shortcuts tend to be taken, and the quality can suffer as a result. It's normally code-base unit integration testing that gets affected first. The testing teams in the test phase get a solution that contains more problems, which makes their lives very hard. So while departmentalization is seen as an advantage, it can easily become a disadvantage if another team is late delivering their part of the project.

Now, let's take a look at some of the high-level disadvantages. The Waterfall model doesn't allow any time for reflection or revision to a design. Once the requirements are signed off on, they're not supposed to change. This should mean that the development team has a fixed design that they're going to work towards. In reality, this does not happen, and changes in requirements can often result in chaos as the design documents need updating and re-signing off on by stakeholders.

By the time the development team starts its work, team members are pretty much expected to get it right the first time, and they're not allowed much time to pause for flaws and reflection on the code that they have implemented. By the time you get to the point where you think a change of technical direction is required, it’s normally too late to do anything about it unless you want to affect the delivery dates. This can be quite de-motivating for a development team, as they have to proceed with technical implementations that are full of compromises and technical debts. Once a product has entered the testing stage, change is virtually impossible—whether to the overall design or the actual implementation.

Now we've seen some of the high-level advantages and disadvantages. Let's take a deeper look at more of the benefits of the Waterfall model. Waterfall is a simple process to understand, and on paper it looks like a good idea for running a project. Waterfall is also easier to manage for a project manager, as everything is delivered in stages that can be scheduled and planned in advance. Phases are completed one at a time, where the output from one phase is fed into the input of the next phase. Waterfall generally works well for smaller projects where the risk of changing requirements and scope is lower. Each stage in Waterfall is very clearly defined. This makes it easier to assign clear roles to teams and departments who have to feed into the project. Because each stage is clearly defined, it makes a milestone set up by the project manager easier to understand. If you're working on a stage like Requirements Analysis, you should clearly understand what you need to deliver to the next phase, and by when.

Under Waterfall, the process and results of each stage are well documented. Each stage has clear deliverables that are documented and approved by key project stakeholders. And finally, tasks in a Waterfall project are easy to arrange and plan for a project manager. The Waterfall model fits very neatly into a Gantt chart, so a project manager is generally happiest when they can plan everything out and view a project timeline in an application like Microsoft Project.

The biggest disadvantage of the Waterfall model is you don't get any working software until late in the process. This means that your end users don't get to see their vision come to life until it’s too late to change anything. It can be very hard for non-technical people to be really clear about how they want an application to operate, and it isn't normally until they can visualize an application that they can really give good feedback. You can mitigate this a bit by doing some prototyping in the system design phase to help users visualize their system, but there is nothing like giving them actual working code to try out.

The Waterfall model can introduce a high level of risk and uncertainty for anything but a small project. Just because a set of requirements and a design has been approved does not mean that the requirements won't change. Waterfall is all about getting the requirements, design, and implementation right the first time. This is a grand idea, but in the real world it is very rarely the case, and this is a big risk to a project. We have talked about how Waterfall is better for small projects, but it is possible to have a small, but very complex project. The more complexity that is involved, the more likely it is that change will be needed further down the line. Complexity in the system is also very hard to implement and test, and can often cause delays in the later stages of the Waterfall software development lifecycle.

If you're working on a project where change is expected, then Waterfall is not the right model for you. I've worked on projects for a financial services company where changes in the law were causing compliance regulations to change. Unfortunately, these rules are very open to interpretation, which meant the legal team was involved at a very early stage. This meant that the interpretation changed a few times during the course of the project. If this had been a Waterfall project, we would have been in big trouble, as projects normally come with very hard and fixed set of deadlines.

This project was a perfect fit for an Agile project. If you are working on a large project and the scope changes, the impact can be so expensive and costly that the original business benefit for the project can evaporate, and then the project is cancelled. I've seen this happen a couple of times, and it's a real shame, as projects that show promise are stopped due to restrictions in the process.

Finally, the integration and delivery of a project is done as a "big bang" on a Waterfall project. This means you're introducing huge amounts of change all at once. This can very easily overwhelm testing teams and your operational teams.

History of the V-Model

Now that we've finished taking a look at the traditional Waterfall model, let's take a look at a model that builds on Waterfall. This is called the V-Model. The V-Model is a modified version of Waterfall. As opposed to the Waterfall method, the V-Model was not designed run in a linear fashion. Instead, the process is turned upwards after the implementation or coding stage is complete, making the V shape, and therefore the name V-Model. The V-Model is based on the idea of having a testing stage for each development stage. This means that for every single stage in the development cycle, there is a directly associated testing phase. This is a strict model, and the next stage starts only after the completion of the previous stage. Now let's take a look at how the V-Model works.

How Does the V-Model Work?

With the V-Model, the related testing of the development stage is planned in parallel, so there are verification stages on one side of the V, and the validation stages on the other side. The coding phase joins the two sides of a V-Model together.

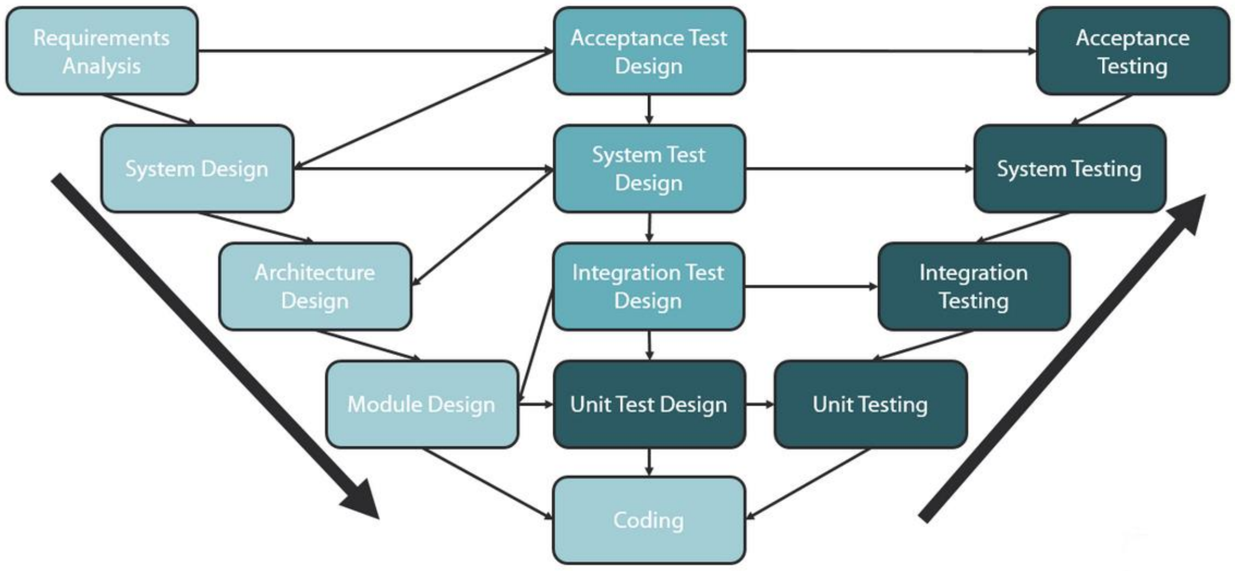

Figure 2 The V-Model software development process

When you draw out the V-Model, it can look a bit complicated, but once you break it down, it's actually quite straightforward. First is the Requirements Analysis phase. This is the first phase in the development cycle where the customer requirements are understood, which requires collaboration with the customer to understand his or her expectations. As most customers are generally not totally sure about what they require, the Acceptance Test Design planning is done at this stage, and business requirements can be used as input for the Acceptance testing.

Next is System Design. Once you understand the requirements, you put together a more complete design of the entire system. This design will include both software and hardware/infrastructure design. At the same time that you prepare the system design, you would normally put the system test design together too so that test teams can pre-plan their testing activities.

Next we have the Architectural Design phase. The architectural design will look at a much-wider design focus than the system design. This may even result in multiple designs being proposed that balance vendors, costs, and other factors.

Unit tests are an important part of any development process, and help identify bugs in the code early so that the team has early visibility of any breakages caused by other dependencies. Once all of these design phases are completed, you can then proceed into the coding stage. The language used in the coding stage along with the architecture should already have been agreed upon by this point, allowing developers to start immediately.

Next, we enter the Validation phase of the V-Model. First, we have Unit Testing. Unit tests designed in the Module Design stage are run against the code during its Validation stage. Unit testing happens at code level and helps eliminate bugs at an early stage. Although all defects cannot be uncovered by unit testing alone, they do give a good indication quickly as to any breakages that may or may not occur. This means that the developers have to be quite disciplined in writing good unit tests that add value, and don't just test language features. Next, we have the Integration Testing phase. Integration tests are performed to test the co-existing of different modules or components within the system. Here we are basically making sure that all integration between various components within the system as a whole are working as expected.

After Integration Testing, we have System Testing. System tests check the entire system's functionality and the communication between all other external systems. If you have integrated with third-party payment providers, for example, they will be tested at this stage. Most of the software and hardware compatibility issues you're likely to face can be uncovered during system test execution.

The final phase is Acceptance Testing. Acceptance testing is associated with the business requirements analysis phase, and involves testing the products in a user's environment. Acceptance tests uncover the compatibility issues with other systems available in the user environments. Acceptance testing also discovers a non-functional issue such as load and performance defects in the user environments.

Where Is the V-Model Suitable?

The V-Model is similar to Waterfall, as both models follow a defined path through their stages. To make the V-Model as successful as possible, your project requirements need to be well-defined, documented, and clearly thought out so that they don't change over time. You also need to be sure that a product definition is stable. This is much easier to describe than put into practice. Project changes over time due to changes in priorities or market conditions are the main sources of problems here. The technology being used must be well understood; before you get to the coding phase, there's often no margin for your developers to learn on the job.

Once you leave the Requirements Analysis and Definition phases, you cannot have any ambiguous requirements, because like Waterfall, there is simply no margin for changing them later without causing a lot of disruption.

Finally, as with Waterfall, the V-Model is ideally suited to shorter project time scales. The longer the project is running, the more risk there is of the requirements changing.

Advantages and Disadvantages of the V-Model

The V-Model shares very similar advantages and disadvantages of the more traditional Waterfall model, but it is worth covering them again, as it helps set the scene for our discussion about Agile software development.

The first advantage is that the V-Model is quite easy to understand and apply. It fits well with companies that have different departments all feeding into the development process. The V-Model is also easy to manage, as you only proceed to the next phase of the model once the current phase is complete. Again, as with Waterfall, the V-Model is not flexible to changes in requirements. This means you'll have to repeat phases in the model to make sure all your documentation is in place. A shift in requirements can be quite disruptive to a project, so you need to ensure the requirements are right from the start.

In reality, what often happens is that if there are any changes to the requirements, to mitigate the cost of disruption, the process is just bypassed to get the changes through quicker. But if you're going to bypass a process, what’s the point in having the process in the first place?

The V-Model works best on small projects where the risk of changing requirements is less than that of a larger project. Generally the V-Model is easy to understand and the actual validation phases are good for mature test departments. Project managers generally tend to like the V-Model, as it is easier to manage against the plan. The rigidity of the model maps well to a project manager's view of the world, and makes their job a bit easier.

The V-Model can introduce a high level of risk and uncertainty for anything but small projects. Just because a set of requirements in a design has been approved does not mean the requirements cannot change. The V-Model is all about getting the requirements, design, and implementation right the first time. Again, this is a nice ideal, but in the real world this is very rarely the case—and this is a big risk to a project. I talked about how the V-Model is better for small projects, but it is possible to have a small but very complex project. The more complexity that is involved, the more likely that change will be needed further down the line. Complexity in a system is also very hard to implement and test, and can often cause delays in the later stages of the V-Model software development lifecycle.

If you're working on a project where change is expected, then a V-Model is not the right model for you. Once you're starting to test your developed solution, going back to make changes in the code (other than to fix defects) can be very difficult and expensive. The biggest disadvantage of the V-Model is that you don't get any working software until late in the process. This means that your end users don't get to see their vision come to life until it is too late to change anything. It can be very hard for non-technical people to be really clear about what they want an application to do, and it isn't normally until they can visualize the application that they can give really good feedback.

Chapter 2 What Is Agile?

Agile is a group of software development processes that promote evolutionary design with self-organizing teams. Agile development inspires adaptive planning, evolutionary development, and early delivery of value to your customers.

The word "agile" was first associated with software development back in 2001 when the Agile Manifesto was written by a group of visionary software developers and leaders. You choose to become a signatory on the Agile Manifesto website, which stamps your intention to follow the principles.

Unlike traditional development practices like Waterfall, Agile methodologies such as Scrum and Extreme Programming are focused around self-organizing, cross-discipline teams that practice continuous planning and implementation to deliver value to their customers.

The main goal of Agile development is to frequently deliver working software that gives value. Each of these methods emphasize ongoing alignment between technology and the business. Agile methodologies are considered lightweight in that they strive to impose a minimum process and overhead within the development lifecycle.

Agile methodologies are adaptive, which means they embrace and manage changes in requirements and business priorities throughout the entire development process. These changes in requirements are to be expected and welcomed. With any Agile development project, there is also a considerable emphasis on empowering teams with collaborative decision-making. In the previous chapter, I talked about how the Waterfall-based development process follows a set series of stages, which results in a "big bang" deployment of software at the end of the process.

One of the key ideas behind Agile is that instead of delivering a "big bang" at the end of the project, you deliver multiple releases of working code to your business stakeholders. This allows you to prioritize features that will deliver the most value to the business sooner, so that your organization can start to realize an early return on your investment. The number of deliveries depends on how long and complex a project is, but ideally you would deliver working software at the end of each sprint or iteration.

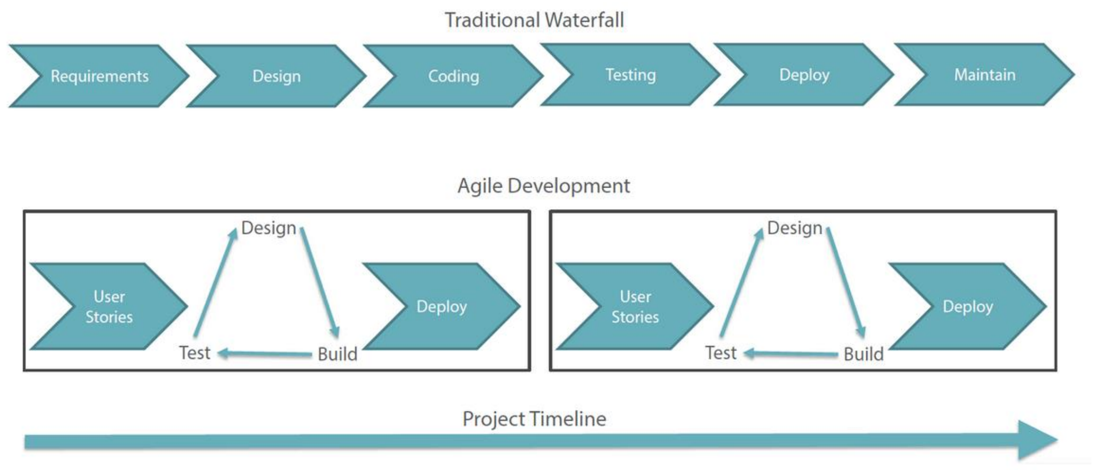

Figure 3 Agile versus Waterfall

Another good way to visualize the premise of Agile is with the diagram in Figure 3. What this diagram shows is that with Agile, you deliver incrementally instead of all at once. You should hold this thought in your mind as we progress through the rest of the book.

A Brief History of Agile

There have been many attempts to try and improve software development practices over the years, and many of these have looked at working in a more iterative way. These new practices didn't go far enough when trying to deal with changing requirements of customers.

In the 1990s, a group of industry software thought leaders met at a ski resort in Utah to try and define a better way of developing software. The term "Agile software development" emerged from this gathering. The term was first used in this manner and published in the now-famous Agile Manifesto. The Agile Manifesto was designed to promote the ideas of delivering regular business value to your customers through the work of a collaborative, cross-functional team.

The Agile Manifesto Core Values

The Agile Manifesto is built upon four core values:

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

Individuals and interactions over processes and tools

Software systems are built by people, and they all need to work together and have good communications between all parties. This isn't just about software developers, but includes QA, business analysts, project managers, business sponsors, senior leadership, and anyone else involved in the project at your organization. Processes and tools are important, but are irrelevant if the people working on the project can't work together effectively and communicate well.

Working software over comprehensive documentation

Let's face it—who reads hundred-page collections of product specs? I certainly don't. Your business users would much prefer to have small pieces of functionality delivered quickly so they can then provide feedback. These pieces of functionality may even be enough to deploy to production to gain benefit from them early. Not all documentation is bad, though. When my teams work on a project, they use Visio or a similar tools to produce diagrams of employment environments, database schemas, software layers, and use-case diagrams (and this is not an exhaustive list). We normally print these out on an A3 printer and put them up on the wall so they are visible to everyone. Small, useful pieces of documentation like this are invaluable. Hundred-page product specs are not. Nine times out of 10, large items of documentation are invalid and out-of-date before you even finish writing them. Remember, the primary goal is to develop software that gives the business benefit—not extensive documentation.

Customer collaboration over contract negotiation

All the software that you develop should be written with your customer's involvement. To be successful at software development, you really need to work with them daily. This means inviting them to your stand-ups, demoing to them regularly, and inviting them to any design meetings. At the end of the day, only the customer can tell you what they really want. They may not be able to give you all the technical details, but that is what your team is there for: to collaborate with your customers, understand their requirements, and to deliver on them.

Responding to change over following a plan

Your customer or business sponsor may change their minds about what is being built. This may be because you've given them new ideas from the software you delivered in a previous iteration. It may be because the company's priorities have changed or new regulatory changes come into force. The key thing here is that you should embrace it. Yes, some code might get thrown away and some time may be lost, but if you're working in short iterations, then the time lost is minimized. Change is a reality of software development, a reality that your software process must reflect. There's nothing wrong with having a project plan; in fact, I'd be worried about any project that didn't have one. However, a project plan must be flexible enough to be changed. There must be room to change it as your situation changes; otherwise, your plan quickly becomes irrelevant.

Note: The Agile Manifesto also contains 12 principles that underpin the four core values. You can read more about it here.

Agile Methodology Overview

Which Agile project methodologies are commonly in use today? First, we'll take a look at Scrum. Scrum is a lightweight project management framework that is based around an iterative working model. Ken Schwaber, Mike Beedle, and Jeff Sutherland, among others, contributed significantly to the evolution of Scrum over the last decade and a half. Over the last few years in particular, Scrum has earned increasing popularity in the software community due to its simplicity, proven success, improved productivity, and its ability to act as a wrapper for various engineering practices promoted by other Agile methodologies.

Next we have Extreme Programming, or XP. Extreme Programming was originally devised by Kent Beck, and has emerged as one of the more popular and controversial Agile methods. XP is a disciplined approach at delivering high quality software quickly and continuously. It promotes high customer involvement, rapid feedback loops, continuous testing, continuous planning, and close teamwork to deliver working software at very frequent intervals, typically one-to-three weeks. Where as Scrum is a project management framework, XP is more of an engineering discipline. It is very common for teams to adopt Scrum, yet borrow different engineering practices from XP.

The original XP recipe is based on four simple values:

- Simplicity
- Communication
- Feedback
- Courage

There are 12 supporting practices:

- Planning game
- Small releases
- Customer acceptance tests
- Simple design
- Pair programming
- Test-driven development
- Refactoring
- Continuous integration
- Collective code ownership
- Coding standards
- Metaphors
- Sustainable pace

Next we have the Crystal methodology, which is a lightweight, adaptable approach to software development. Crystal is actually comprised of a family of methodologies like Crystal Clear, Crystal Yellow, and Crystal Orange, whose unique characteristics are driven by factors like team size, system criticality, and project priorities. The Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project's unique characteristics.

Next we have the Dynamic Systems Development Method, or DSDM, as it is most commonly known. DSDM dates back to 1994, and evolved to provide an industry standard project framework for what was commonly called Rapid Application Development (RAD).

While RAD was popular in the early 1990s, the RAD approach was quite unstructured, and as a result of this, DSDM was formulated to add structure around the idea of rapid application development. Since 1994, the DSDM methodology has evolved and matured to provide a comprehensive foundation for planning, managing, executing, and scaling Agile and iterative software development projects. DSDM is based on nine key principles that revolve around business need and value, active user involvement, empowered teams, frequent delivery, integration testing, and business stakeholder collaboration.

Next, we have Feature-Driven Design (FDD), developed by Jeff De Luca. FDD is a model-driven, short-iteration process. It begins with establishing an overall model shape and continues with a series of two-week, design-by-feature, built-by-feature iterations. The features are small and useful in the eyes of the client. FDD designs the rest of the development process around feature delivery using the following practices:

- Domain object modeling
- Developing by feature
- Components and class ownership
- Feature teams
- Inspections
- Configuration management
- Regular builds
- Visibility of progress
- Results

Next, we have Lean software development, an iterative method developed by Mary and Tom Poppendieck. Lean software development owes its heritage to the Lean enterprise movement of companies like Toyota. Lean software development focuses the team on delivering value to the customer, and on the efficiency of the value stream and the mechanisms that deliver that value.

The main principles of Lean include:

- Eliminating waste
- Amplifying learning
- Deciding as late as possible
- Delivering as fast as possible
- Empowering the team
- Building in integrity
- Seeing the whole

Lean eliminates waste by focusing on only the parts of a system that deliver true value to a business. It emphasizes the speed and efficiency of development and relies on rapid and reliable feedback between programmers and customers.

Roles Within an Agile Team

Agile teams, while part of a department or company, are primarily focused on their software development goals. Each team should also be focused on their team's overall vision. This means a team should be very reactive in doing whatever is required to get the job done. Team members may have to do work that is outside their normal skill set, and this should be embraced and encouraged. A cross-functional and adaptive team is much more likely to succeed.

Most teams will, of course, have some standard areas of expertise and specialties, and you may also have people with specific domain or product knowledge, but generally, there should be flexibility in team players' expected roles and responsibilities. It should also be common for team members to have access to the business as a whole—and this shouldn't just be limited to a select few. You should have people who are tasked with making sure the team follows the development process, and someone who co-ordinates requirements gathering with the business; this would typically be referred to as the product owner if you are working within the Scrum framework.

Teams will normally have some form of leadership role within the team. In Scrum, this person is the Scrum Master. On Agile teams, the role of this person is to enable and ensure the success of the team. This type of leader is normally referred to as a servant leader. This role is quite different to the direct transactional leader on a Waterfall project.

One goal of an Agile team should be to improve every day. The larger the organization, the more complex team structures can get. Cross-project teams, shared services, operations, configuration management, and database administration can all come into play, but the goal remains the same: define a software project and cross-functional team capable of delivering on that project, and empower the team to do so.

Summary

In this chapter we have taken a high-level look at what Agile is about by exploring the Agile Manifesto and introducing two of Agile's most popular methodologies, Scrum and Extreme Programming. We then took a brief look at some other methodologies like Crystal, DSDM, Rapid Application Development, and Feature-Driven Design. Finally, we talked about some typical roles in an Agile team, from the development team itself, through to Agile leadership.

Chapter 3 Common Agile Misconceptions and Mistakes

Common Agile Misconceptions

When a team is new to Agile, it can be hard to adjust to a new way of working, especially if team members are used to working under a Waterfall-based methodology. When a team is faced with changing how they work, it's common for excuses to be made by team members as they resist the change. Not all teams are like this, but in my experience, it's quite common to hear many different misconceptions. In this chapter, we'll discuss many of these misconceptions and why they come about.

Agile is ad-hoc, with no process control: To be agile, you need to adhere to the Agile Manifesto, but following the manifesto doesn't mean you are using a defined process. The manifesto describes a set of ideals. There are various different processes and project management templates that you can apply to your projects to help them become agile. Extreme Programming and Scrum are the two most popular, but Lean and Kanban are also becoming very popular.

When you try to implement the manifesto items, you generally need to apply lots of common sense and pragmatism to get to your goal. If you want to wrap a more formal process around the "how" of Agile, as opposed the "why," then you would need to apply something like Scrum or Extreme Programming, which gives you more formal processes like stories, iterations, stand-ups, demos, retrospectives, test-driven development, and pair programming.

Agile is wasteful without upfront planning: This assumes that your customer knows the details of all their requirements in advance. If this is true, then by all means, undertake comprehensive upfront planning. However, in reality this is rare, and usually leads to greater waste of having undertaken unnecessary design and development work.

Agile development is not predictable: When working with an established Agile team, you can bring a level of predictability to your development life-cycle and business, as you will be regularly delivering working software to your customers. The frequency of these releases will be set with your stakeholders, but in the ideal situation you should have releasable code at the end of each sprint.

Agile is faster and cheaper: Running an Agile team doesn't mean you will finish a project quicker or for less money. It isn't a direct money-saver in that respect—it's about delivering value to the business sooner. You head towards working versions of the software quicker. At the end of each development iteration, you're supposed to have working software to demo to the business. You may not have all the requirements in place, but what is there will work. This means re-thinking how you plan your workload in each iteration. Instead of delivering horizontal slices—such as the data access layer in this iteration, and the user interface in the next iteration—you think in vertical segments.

This means you deliver defined pieces of functionality in an iteration that may encompass work on the user interface and data access layer. It's a mind shift that I've seen teams struggle with if they are used to working horizontally, but when they finally get it, the efficiency of a team is increased remarkably. Being agile is also about being able to respond to change. Requirements can change, and businesses can change partway through a delivery. I've worked with teams who treat this is a real negative thing. If you want to be agile, you need to expect and embrace that things will change. The tools and processes of Scrum, for example, are designed to help you react to these changes in a more efficient manner.

Agile teams don't write documents or do planning: Practicing Agile is not an excuse to avoid planning or documentation writing. Agile is an act of doing what is needed at the time of requiring it, and encourages continuous planning and documentation—but only when it is needed for specific customer requirements. This allows teams and their customers to decide if the planning or document adds value to the product. Depending on what type of company you work for, formal documentation may not be something that you can avoid. For example, if you work in a very heavily regulated environment, then there's lots of upfront documentation that may be needed for evidence and submission to a regulatory body. If this is the case, then the delivery team will need to take this documentation into account.

I personally prefer to work with large diagrams instead of large documents of text. If you can, get these diagrams printed out onto A3 paper, and then put them up all over the walls so you have something to refer to in your stand-ups. With the planning side of this, you still need to do it. At the beginning of each iteration or sprint, you should have a planning session where you allocate user stories for iteration. The number of stories you allocate will be based on the estimates given and the velocity of the previous iteration.

Agile means no commitments: It's a common belief that people on Agile teams do not want to make commitments, and that you have a team of developers churning away until someone shouts, "we're done!" A successful Agile team should be very transparent about what they intend to deliver to their users. Using methodologies like Scrum and Extreme Programming, you have a backlog, which contains all your high-level user stories and tasks for a given sprint or iteration. As you define the workload for a sprint, this should be seen as a guide to what the team intends to deliver. Generally, once a sprint or iteration is set up, it will not change, but you may have to change your plan partway through. This could result in a partial re-plan in that iteration, or waiting until the next iteration. XP doesn't like changing an iteration once it is in flight, but this is more acceptable under Scrum. However, there is no law that says you can not change the commitment if required. What is important is that a level of trust is built between the team and the business stakeholders.

An Agile project will never end: This might be true in some situations. You should continue to work on a project while the customer continues to get business value, and that value is worth more than the cost of developing it. Most products in any industry have a point of diminishing returns; this is the ideal time for an Agile project to end. This decision should come from the business though, since it's for them that you are delivering value. Agile works for projects, teams, and organizations of any size, not just small projects. This doesn't mean it'll necessarily work for all organizations, but size is rarely a factor. Large, complex projects and organizations are often excellent candidates for an Agile transformation, where it is difficult or impossible to know all of your customer's requirements in advance.

Agile is not the solution to all your problems: Agile is a change in approach and culture that comes with its own set of benefits and issues. If you're working in a well-established team that has not been following any Agile processes, then changing them over will not be an instant transformation. You need to do it slowly, and make sure everyone has a say in the decision-making process. If you don’t, you may get resistance from team members who fear change, which is a perfectly normal human characteristic. Convincing your team isn't the biggest hurdle though—your biggest challenge is making sure your leadership team understands and wants to adopt Agile as a way of working. Once you have buy-in from the leadership, then the rest of the adoption just takes time and patience as everyone adjusts.

There's only one way to do Agile: The Agile Manifesto consists of four core values and 12 principles. It doesn't document any actual implementation details. There are many interpretations of Agile that form different methodologies, like Scrum, Extreme Programming, Kanban, and Feature-Driven Development, to name a few. Each style has its own benefits and weaknesses, and you must evaluate your own situations to decide which methodology is the best fit for your team. As long as you're sticking to the Agile Manifesto's values and principles and delivering high-value software regularly to your customers, you should be considered Agile.

Agile development doesn't require upfront design: It is a common misconception that Agile teams just make it up as they go along. What is more realistic is that Agile teams should make sure design happens at the latest possible point in time. For coding activities, it is more acceptable that the code is designed as the developer works on it, and refactors to a better design as they go along; this is what evolutionary design is all about. More system-wide and architectural design can be scheduled in one or more iterations ahead of time. By only designing as you need to, you can react to changes in requirements more efficiently. When you try to design the entire system upfront, any design decisions that you make are likely to be redundant by the time you implement them.

Chapter 4 Advantages and Disadvantages

Advantages of Agile

As you've seen in the past chapters, Agile software development is a completely different approach to software development compared to the more traditional Waterfall development model. Let's take a look at some of the advantages to using the Agile approach.

Customer satisfaction by rapid, continuous delivery of useful software

Your customers and users will be satisfied because you are continually delivering value with usable software. This is a stark contrast with the traditional Waterfall product delivery process. If your customers are used to Waterfall, they may find it strange to have working software sooner. The big downside of Waterfall is that you deliver large pieces of functionality towards the end of the project life-cycle. This means that all throughout the development stages of Waterfall, your project is incurring cost with no return on investment. By delivering working pieces of functionality sooner and more regularly, you're giving your users an opportunity to get a return on their investment sooner. Sure, they may not have all the functionality they need upfront, but they can start to make use of the solution to make their lives easier and start realizing the benefits sooner. People and interactions are emphasized rather than process and tools.

Agile is focused very heavily around people and the interactions between people rather than the processes and tools

This is a core value of the Agile Manifesto. This is important because it is the input from your team and customers that will ultimately make your project a success, as opposed to what tools you use. Continual collaboration throughout the entire development cycle of your project enables everyone involved to build a good working relationship based on trust. This trust-based working relationship is crucial when building software incrementally.

Continuous attention to high quality code and design

When working with Agile, you're working short iterations and only build what is necessary to satisfy the requirements for that iteration, and nothing else. This forces you to keep your design simple, which helps you design testable and more reliable systems. Developers understand and choose many solutions to solving a businesses problems, and these are choices that reflect a craft that balances design, use, and support. Developers provide the technical support to the team that enables them to always move forward at rapid pace and keep code quality high. Developers like to use the latest techniques for keeping their implementations simple and clean without having to rework any of their solutions.

One of these techniques is refactoring. Refactoring is the process of improving the design of existing code without changing its behavior. In order to make changes to the structure of the code, refactoring uses a quick succession of small, well-defined steps that can be verified as safe or functionally equivalent. Refactoring is most often done in conjunction with test-driven development where unit tests and simple design make it easier to refactor safely.

Simple design

Keeping your design simple (and not repeating code) helps you keep your code maintainable. If you design your code to be modular and interface-driven, then you can reduce coupling between objects, which leads to a more robust system.

Test-driven development

Test-driven development (TDD) is a way of driving the design of code by writing a test, which expresses what you intend the code to do, making that test pass, and continually refactoring to keep the design as simple as possible. TDD can be applied at multiple levels, for example, unit tests and integration tests. Test-driven development follows a rigorous cycle. You start by writing a failing test, and then implement the simplest solution that will cause that test to pass. Next, you search for duplication in the code and remove it. This is often called Red-Green-Refactor, and has become almost a mantra for many test-driven design practitioners. Understanding and even internalizing this cycle is key to being able to use test-driven design to solve your problems.

Embracing changes in requirements

Your customer or business sponsor may change their mind about what is being built. This may be because you have given them new ideas from software you delivered in a previous iteration, or because the company's priorities have changed or new regulatory changes have come into force. The key thing here is that you should embrace it. Yes, some code may get thrown away and some time lost, but if you're working in short iterations, then this lost time is minimized. Change can be very scary at first for clients and partners alike, but when both sides are prepared to take the leap, it can be mutually rewarding. In some ways, Agile is a simple idea, but the reality is that it can mean different things to different people, depending on their role in the software development process. One of the key things, though, is to be open to change, not just in order to move in traditional ways of organizing projects, but to adapt your use of Agile itself.

Early return on investments

Another advantage to releasing higher-value features early is you start to get a return on your investment. Running a software delivery team is expensive. You have permanent developers and testers, as well as consultants with expensive day rates. There's also business analysts and project managers, as well as other hardware and software costs. These are all costs to the business. By releasing early and generating revenue from your product, you can start to offset some of the initial investments and development costs. On the flip side of that, if you have a more Waterfall-based approach where you end up with a "big bang" deployment after a year or so, you will have already spent large amounts of money to fund the development with nothing to show until the end.

Feedback from your customers

If you release early, you can start to solicit feedback from your customers a lot sooner. These customers could be public-facing customers or business sponsors. I've worked on many projects where the business customer specifies requirements, which you then build, only for the customer to want changes once they have something they can use. This always seems to happen—it’s very hard for someone to specify a system without having something to play with. You can use prototyping software to help, but there really is nothing like giving them actual functionality early on to start using. One of the principles of Agile is to embrace change in the requirements. This should be expected, so giving your customer something they can give feedback on earlier in the process will give them an opportunity to make changes sooner without causing much disruption.

Feedback from real customers

Once you start getting feedback from real customers, you can start incorporating changes and new ideas from the feedback into the product. It is much more cost effective to make changes early on in a product's development cycle than it is to wait until the end after a large release has been made. It's not just customer feedback that helps you build the right product—by testing your product early in the marketplace, you can gauge customer uptake and see how popular the product will be, and continually deliver better quality.

Everything we have discussed so far has business benefits or culminates in the fact that you should be delivering a better quality product with every release. By releasing earlier and soliciting feedback, you can learn from the product performance earlier, and use this information to create something better. Product and system development is all about continuous learning and improvement, which is much easier to do when you're delivering a project by being agile. It doesn't matter whether you're using Extreme Programming, Scrum, DSDM, Crystal, or any of the other project management frameworks. If you adhere to the core values in the Agile Manifesto and routinely deliver high-value functionality early to your customers, monitor their usage, and listen to their feedback, you can apply this learning to the ongoing development and increase quality as you go along.

Disadvantages of Agile

Now that we have examined some of the advantages of Agile development, let's take a look at some disadvantages.

Difficult to assess the effort required at the beginning of the software development life cycle

One complaint I have often heard from business leaders and project managers alike, is that compared to Waterfall, it’s hard to quantify the total effort and cost to deliver a project. On one hand, I can see why they would think this, especially when they come from a regimented Waterfall process world. Indeed, it is harder to fully quantify how long the total project will take, but the mitigation for this is that a product will be delivered incrementally by giving the users the most valuable requirements first, meaning you can plan for the coming sprint (and maybe a few sprints ahead) to deliver a specified amount of functionality.

It can be very demanding on a user’s time

Active user participation and collaboration with the users of your system are required throughout the development cycle with Agile. This can be very rewarding, and ensures you deliver the right product to your users. It's a key principle with Agile to ensure that a user’s expectations are well-managed, and the definition of failure is not meeting your user’s expectations. However, this level of participation can be very demanding on the user, and requires a big commitment for the duration of the project. I have been in this situation many times where the business users love the idea of what Agile can bring to them, but they don't like the extra amount of time they have to spend on the project in addition to their current workloads.

Costs can increase as testers are required throughout a project instead of at the end of a project

Testing is a key part of an Agile project during sprints or iterations. This helps to ensure quality throughout the project without the need for a lengthy and unpredictable test phase at the end. However, this does mean that testers are needed throughout the entire product development life cycle, and this can dramatically increases the cost. This extra upfront cost does save you money in the long run though, as you are continually having people test your code. Having a combination of manual testing and automation testing is the best way to drive up the quality of your product. The cost of a long and unpredictable test stage at the end of a Waterfall project can, in my experience, cause huge, unexpected costs when a project overruns—and they frequently do overrun.

What Are Your Department's Biggest Challenges?

Let's now take a look at whether Agile is right for your team, and if you are prepared for moving to a more agile way of working. If you’re working at a new company or on a brand-new team, starting out with Agile can be very easy, but if you're working in a larger, well-established organization that has been using more of a Waterfall-based approach, the switch to Agile can be difficult to do.

Let's start off by looking at possible challenges faced by your department.

Is your department under pressure to achieve difficult deadlines? Are there too few people to get the work done, or insufficient budget allocations? Are staff members not as productive as they could or should be? Are the business processes, equipment, or communication channels slowing them down? Is there too much corporate knowledge in the heads of a handful of employees? Are low quality outputs creating the need for constant fire-fighting and damage control?

Every IT team can benefit from using an Agile approach, but the teams that have the most significant issues also have the most to gain from Agile approaches that specifically target these issues. Agile is ideally suited to teams where there are ongoing issues with the quality of delivered solutions, delivering software solutions within agreed timelines or budgets, and delivered solutions not adequately supporting business requirements, or there are high staff-turnover rates or low staff-productivity levels.

The amount of benefit your team will get from implementing Agile is also linked to a number of risks:

- The likelihood for requirements changing while the product is being developed, including changes in user requirements, staff departures, business priority shifts, and funding
- External changes factors like changes in market demand, announcements from competitors, and the availability of new technologies
- The sustainability of your current overheads, including development costs, implementation costs, maintenance, and support

If your products are based on predictable and replicable business processes with a minimum likelihood of changing requirements, then your team will not achieve the same level of benefit from Agile as one that is more susceptible to solution requirements that are likely to change over time. The same is true for teams where the current software solutions are delivered on time, align well with the business requirements, and require minimal ongoing support to address quality and usability issues.

In each of these situations, Agile methodologies can provide some degree of benefit to the team, but not the dramatic benefit that the teams with more dynamic and less sustainable software solutions can achieve. Ultimately, the more your team is faced with changing requirements and unsustainable IT overheads, the better positioned you are to receive returns on your Agile investments.

Are You Prepared for Agile?

For some organizations, particularly larger and older ones, the answer to this question is likely to be “no.”

Methodologies that encourage the evolution of business requirements (instead of relying on upfront documentation) empower the project team to self-organize instead of controlling their daily activities. Replacing reams of documentation with face-to-face communication may seem a bit daunting for some staff members, something that is especially true for those that have grown comfortable in their normal day-to-day routines and just live with the problems in their code and the solution they are developing. A big debilitation when trying Agile is people saying, "This is the way we’ve always done it.” These types of people are normally very resistant to change.

If your staff is hesitant at first, you may find that giving Agile a try on a donor project in your team will help get them familiar with and motivated by Agile. If after trialing one or two Agile projects, your staff are still uncomfortable working directly with the business areas, supporting changing requirements as the project progresses, and self-managing their work, it may be that Agile approaches are just not suited to your organization’s working culture.

If, on the other hand, your team reacts well to the trial projects, then this paves the way for you more fully adopting these methodologies. Going Agile requires a change in attitude for managers and leadership, too. Traditionally, it might have been more common to have direct control over what your team members are working on, but with Agile, you need to take a different approach.

Management style needs to be more like servant leadership, where managers are there to remove any barriers form the teams' progress, and encourage the team to think for themselves and organize their own workload. After all, developers are generally paid very well, so you need to have a more realistic level of trust that they will do the right thing.

Another interesting thing about the dynamic of self-organizing teams is that as they progress, they improve ongoing motivation for employees. Project team members know that their continued ability to self-manage their work depends on their regular delivery of higher-value business outcomes. Additionally, because they are the ones who identify what work can and cannot be achieved in each iteration, they are motivated by their personal responsibility to achieve these outcomes. This combination of factors is heightened by the satisfaction and pride that staff members feel when they produce tangible outputs that truly meet the needs of the organization.

Chapter 5 Extreme Programming (XP)

Now that we have looked at some theory behind Agile software development, it's time to take a look at some Agile methodologies.

In this chapter, we'll look at Extreme Programming, or XP for short. As we look at Extreme Programming, we'll first look at its history and an overview of the methodology. Next, we'll look at common XP activities, values, principles, and practices. Finally, we'll finish up by looking at the different rules of XP, which are split into five categories: planning, managing, designing, coding, and testing.

History of Extreme Programming

Extreme Programming is a software development methodology that is intended to improve software quality and responsiveness to changing customer requirements. As a type of Agile software development, it advocates frequent releases and shorter development cycles, which are intended to improve productivity and introduce checkpoints where new customer requirements can be adopted. Other elements of XP include programming in pairs or doing extensive code reviews, unit testing all of the code and avoiding programming of features until they are actually needed. XP has a flat management structure with simplicity and clarity in code, and a general expectation that customer requirements will change as time passes. The problem domain will not be understood until a later point, and frequent communication with the customer will be required at all times.

Extreme Programming takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to extreme levels. As an example, code reviews are considered a beneficial practice. Taken to the extreme, code can be reviewed continuously with the practice of pair programming.

XP was created by Kent Beck during his work at the struggling Chrysler Comprehensive Compensation System payroll project, or C3, as it was known. In 1996, Chrysler called in Kent Beck as an external consultant to help with its struggling C3 project. The project was designed to aggregate a number of disparate payroll systems into a single application.

Initially, Chrysler attempted to implement a solution, but it failed because of the complexity surrounding the rules and integration. From this point of crisis, Kent Beck and his team took over, effectively starting the project from scratch. The classic Waterfall development approach had failed, so something drastic was required. In Kent Beck's own words, he just made the whole thing up in two weeks with a marker in his hand and a white board. Fundamentally, the C3 team focused on the business value the customer wanted, and discarded anything that did not work towards that goal. Extreme Programming was created by developers for developers.

The XP team at Chrysler was able to deliver its first working system within a year. In 1997, the first 10,000 employees were paid from the new C3 system. Development continued over the next year, with new functionality being added through smaller releases. Eventually, the project was cancelled because the prime contractor changed, and the focus of Chrysler shifted away from C3. When the dust settled, the eight-member development team had built a system with 2,000 classes and 30,000 methods. Refined and tested, XP was now ready for the wider development community.

Overview of Extreme Programming

Extreme Programming can be described as a software development discipline that organizes people to produce high-quality software more productively. XP attempts to reduce the cost of changing requirements by having multiple short development cycles rather than one long cycle, as is seen in Waterfall.

With XP, changes are a natural, inescapable, and desirable aspect of software development projects, and should be planned for instead of attempting to define a stable set of requirements upfront. XP is built around its own set of activities, values, practices, and rules, which we will examine in detail in this chapter.

Activities

Extreme Programming describes four basic activities that are performed within the software development process. These activities are:

- Coding
- Testing
- Listening
- Designing

Coding

Coding is the most important product of the XP process. Without code, there is no working product. Coding can be performed using an on-screen designer that will generate code, scripting a web-based system, or coding a program that needs to be compiled.

A programmer dealing with a complex problem and finding it hard to explain the solution to fellow programmers, might code it and use the code to demonstrate what he or she means. This coding can involve many different languages, such as C#, Java, Python, C, C++, F#, JavaScript, and many more.

Testing

With Extreme Programming, the developer will practice what is called test-driven development. This is where you write a failing test first and implement just enough code to pass the test, and then refactor the code to a better structure, while tests still pass. Programmers will strive to cover as much of their code in unit tests as they can to give them a good level of overall code coverage. This code coverage will help build trust that the system will operate as expected.

You cannot be certain of having a working system or product unless you have tested it. With XP, you ideally want to automate as much of your testing as possible so that you can repeat the testing frequently. This can be done by writing unit tests. Unit tests will test a small block of code in isolation from any external dependencies like databases or the file system.

Listening

Programmers must listen to what the customers need the system to do and what business logic is required. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem can or cannot be solved. The requirements from the customer are documented as a series of user stories.

These user stories help to drive out a series of acceptance tests, which help determine when a user story is completed and working as expected. Once user stories and acceptance tests are written, the developers can start their planning and estimating.

Designing

The final activity is designing. To create a working system or product, requirement gathering, coding, and testing should be all you need. In reality, however, software systems are very complicated, so you'll often need to perform a level of overall system design that you may not have expected. This doesn't mean that you'd need to create a several-hundred-page design document, as that could be quite wasteful, but there is definite value in producing an overall system design where you look at the overall structure of the system and its dependencies.

Ideally, you want to create a system where all of the components are as decoupled from each other as they can be, so that a change in one component doesn't require sweeping changes across the rest of the system.

Values

Extreme Programming is based on five core values. Although XP defines many rules, which we'll look at in a bit, it’s more wired to work in harmony with your personal and corporate values. The five values are:

- Communication
- Simplicity
- Feedback
- Respect

- Courage

Communication

Good communication is essential to any project. Honest, regular communication allows you to adjust to change. This is how developers know what to do, and how the customer knows when it will be done. XP puts developers and customers in constant communication. A customer sets business priorities and answers any questions about how they perceive a project, both as a real user and from a business point of view.

The customer sees the team's progress every day, and can adjust the work schedule as needed, as the customer works with the developers to produce tests to verify that a feature is present and works as expected.

When you have a question about a feature, you should ask the customer directly. A five-minute face-to-face conversation, peppered with body language, gestures, and white board drawings, communicates more than an email exchange or conference call can, so removing the communication barriers between customers and developers greatly increases your flexibility.

Clear communication about goals, status, and priorities not only allows you to succeed, but makes everything else in the project run smoothly too.

Simplicity

Simplicity means building only the parts of the system that really need to be built. It means solving today's problems today and tomorrow’s problems tomorrow. Complexity is expensive, and predicting the future is very hard. Once you’re armed with communication and feedback, it's much easier to know exactly what you need. If you practice simplicity, it should be easy to add a feature when it becomes necessary.

Feedback

Feedback means asking questions and learning from the answers. The only way to know what a customer really wants is to ask them. The only way to know if the code does exactly what it should do is to test it. The sooner you can get feedback, the more time you have to react to it. XP provides rapid, frequent feedback. Every XP practice is part of building a feedback loop. The best way to reduce the cost of change is to listen to and learn from all of those sources as often as possible. This is why XP concentrates on frequent planning, design, testing, and communication. Rapid feedback reduces the investment of time and resources into ideas with little payoff.

Failures are found as soon as possible—within days or weeks, rather than months or years—and this feedback helps you to refine your schedule and plan even further than your original estimates may have ever allowed you to. It allows you to steer your project back on track as soon as someone notices a problem or identifies when a feature is finished, and very importantly, where it will cost more or less than previously believed. It builds confidence that the system does just what the customer really wants.

Courage

Courage means making the hard decisions when necessary. If a feature isn't working, fix it. If some code is not up to standard, improve it. If you're not going to deliver everything you promised on schedule, be upfront and tell the customer as soon as possible. Courage is a difficult virtue to apply—no one wants to be wrong or to break a promise. The only way to recover from a mistake, though, is to admit it and fix it.

Delivering software is challenging, but meeting that challenge, instead of avoiding it, leads to better software.

Respect

Respect underlies the other values previously mentioned. Every member of the team must care about the project. Intrinsic rewards like motivation, enjoyment, and job satisfaction beat extrinsic reward, like employee-of-the-month awards or physical rewards, every time. Everyone gives and feels the respect they deserve as a valued team member. Everyone should contribute value to the team, even if it's simply enthusiasm. Developers should always respect the expertise of the customers (and vice-versa), and managers should always respect the developer’s right to accept responsibility and receive authority over their work.

Principles

Extreme Programming sees feedback as most useful if it is done frequently and promptly. It stresses that minimal delay between an action and its feedback is critical to learning and making changes. Unlike traditional system development methods, contact with the customer occurs more frequently. The customer has clear insight into the system that is being developed, and can give feedback and steer development as needed.

With frequent feedback from the customer, a mistaken design decision made by the developer will be noticed and corrected quickly, before the developer spends too much time implementing it.

Unit tests contribute greatly to the rapid feedback principle. When writing code, running the unit tests provides direct feedback as to how the system reacts to any changes. This includes running not only the unit tests that test the developer's code, but running all the unit tests in a project against the software using an automated process that can be initiated by a single command as part of a build. If a developer's change causes a failure in some other portion of the system that the developer making the change knows little or nothing about, the automated unit test suites will reveal the failure immediately, alerting the developer of the incompatibility of a change within other parts of the system, and the necessity of removing or modifying this change.

With other traditional development practices like Waterfall, the absence of an automated, comprehensive unit test suite means that such a code change, thought to be harmless by the developer, would have been left in place, appearing only during integration testing—or even worse, showing up once the product has been put into production.

Determining which code change caused the problem, among all the changes made by the developers during the weeks and months previous to integration testing, is an immensely difficult task, and not one you want to perform very often.

Simplicity is about treating every problem as if its solution were extremely simple. Traditional system development methods say to plan for the future and to code for reusability. XP rejects these ideas, and applies incremental changes.

For example, a system might have small releases every three weeks. When many small steps are made, the customer has more control over the development process and the system that is being developed. The principle of embracing change is about not about working against changes, but embracing them.

For instance, if at one of the iteration planning meetings it appears the customer's requirements have changed dramatically, programmers are able to embrace this and plan new requirements for the next iteration. Under Waterfall, changes in requirements are seen as very bad and costly. Even small changes can have a very large impact to a program at work. If any of the main fundamental requirements change under Waterfall, it could put the entire project at risk of being cancelled. This risk is drastically minimized under an Agile development framework like XP.

Practices

In Extreme Programming, 12 practices are followed. These are split into four main groups that aim to define software development best practices. These are:

- Fine-scale feedback
- Continuous process
- Shared understanding
- Programmer welfare

Fine-scale Feedback

Let’s examine the practices of fine-scale feedback. First of all, there's pair programming, which means that all code is produced by two people programming on one task at one workstation. One programmer has control over the workstation and is thinking mostly about the coding in detail. The other programmer is more focused on the big picture and is continually reviewing the code that is being produced by the first programmer.

Programmers trade roles after short periods of time; the pairs are not fixed. Programmers switch partners frequently so that everyone knows what everyone else is doing, and everybody remains familiar with the whole system, even the parts outside their skill sets. This way, pair programming can also enhance team-wide communication.

The main planning process within Extreme Programming is called a planning game. The game is a meeting that occurs once per iteration, typically once a week or every two weeks. The planning process is divided into two parts. The first part is release planning; this is focused on determining what requirements are included in which near-term releases, and when they should be delivered. The customers and the developers are both part of this meeting.

Release planning consists of three phases. The first is the exploration phase. In this phase, the customer will provide a short list of high-value requirements for the system. These will be written down on user story cards. Then, there's the commitment phase. Within the commitment phase, the business and developers will commit themselves to the functionality that will be included and the date of the next release.

Then, there's the steering phase. In this phase, the plan can be adjusted, new requirements can be added, and existing requirements can be changed or removed. After release planning, we have the iteration planning, when we plan the activities and tasks of the developers. In this process, the customer is not involved. The purpose of the planning game is to guide the product into delivery. Instead of predicting the exact dates when deliverables will be needed and produced, which is difficult to do, the aim is to steer the project to completion.

Next, we have test-driven development. Unit tests are automated tests that test the functionality of pieces of the system being developed. Within XP, unit tests are written before the code is written. This approach is intended to stimulate the programmer to think about the conditions in which his or her code could fail. A programmer is finished with a certain piece of code when he or she cannot come up with any further conditions in which the code may fail. Test-driven development proceeds by quickly cycling through a series of steps, with each step taking minutes at most, but preferably much less.

First, programmers write a minimal test that should break the code because the functionality hasn't been fully implemented. After the programmers verify that the code does indeed fail the test, they will write the minimum amount of code to make the test pass. The unit tests are run to make sure that they pass. Then, you should modify or restructure the code to a better design while the tests still pass.

Within XP, the customer is the one who really uses the system being developed. The customer should be on hand at all times and available for questions. For instance, a team developing a healthcare dispensing system should include a pharmacy business partner to answer questions and assist with the design.

Continuous Process

Now let's take a look at the practices for continuous process. First, we have continuous integration. The development team should always be working on the latest version of the software. Since different team members may have versions saved locally with various changes and improvements, they should try to upload their current version to the code repository at least every hour, or when a significant break presents itself.

The source code repository should ideally run an automated build against the code as it is checked in, and then run the automated unit tests. This will test the integrity of the code being checked in. Continuous integration will avoid delays later on in the project cycle caused by integration problems.

Next we have refactoring or design improvements. XP advocates programming only what is needed today, and implementing it as simply as possible.

Another symptom is that changes in one part of the code affect lots of other parts. The XP approach states that when this occurs, the system is telling you to refactor your code by changing the architecture, making it simpler and more generic.

The delivery of the software is done by frequent releases of live functionality, creating value for the end user. The small releases help the customer gain confidence in the progress of the project. Once you're building quality software, team members can feel good about the accomplishments they've achieved.

Shared Understanding

Coding standards are an agreed-upon set of rules that the entire development team adheres to throughout the project. The standards specify a consistent style and format for source code within a chosen programming language, as well as various programming constructs and patterns that should be avoided in order to reduce the probability of defects.

The coding standards may be a set of conventions specified by the language vendor or custom-defined by the development team. These days, it's common to use a coding productivity tool such as ReSharper, CodeRush, or JustCode to help enforce these standards. These tools will be set up with a pre-defined set of rules, and as the developer is writing code, these tools will highlight violations of the coding standards, and in most cases offer suggestions for fixes. They are excellent for ensuring consistency within a code base.

Next, we have collective code ownership. Collective code ownership means that everyone is responsible for all of the code. This in turn means that everybody is allowed to change any part of the code. Pair programming contributes to this practice by working different pairs. All the programmers get to see all of the parts of the code.

A major advantage of collective code ownership is that it speeds up the development process, because if an error occurs in the code, any programmer can fix it. By giving every programmer the right to change code, there is a risk of errors being introduced by programmers who think they know what they're doing, but do not foresee certain dependencies. Sufficiently well-defined unit tests help to address this problem. If unforeseen dependencies create errors, then when the unit tests are run, they will show up as failures.

Next up, we have simple design. Programmers should take a "simple is best" approach to software design. Whenever a new piece of code is written, the developer should ask themselves: "Is there a simpler way to introduce the same functionality?" If the answer is "yes," the simpler course should be chosen. Refactoring should also be used to make complex code simpler.

Finally, with shared understanding, we have the system metaphor. The system metaphor is a story that everyone—customers, programmers, and managers—can tell about how the system works. It's a naming concept for classes and methods that should make it easier for a team member to guess the functionality of a particular class or method from its name only. For example, a pharmacy healthcare system may create a dispensable drugs class for a dispensing system, and if the drug goes out of stock, then the system will return a warning when a check stock availability method is called on the dispensing drug's class. For each class or operation, the functionality is obvious to the entire team.

Programmer Welfare

For the final principle, we'll take a look at programmer welfare, and start with sustainable pace. The concept is that programmers or software developers should not work any more than 40-hour weeks, and if there is overtime one week, then the next week should not include any overtime. Since the development cycles are short cycles of continuous integration, and full development release cycles are more frequent, the projects in XP do not follow the typical crunch time that other projects require (which requires overtime).

Also included in this concept is that people perform best and most creatively if they are rested. A key enabler to achieve sustainable pace is to frequently code-merge and always have executable and test-covered, high-quality code. The intense, collaborative way of working within a team drives the need to recharge over weekends.

Well-tested, continuously integrated, frequently-deployed code and environments also minimize the frequency of unexpected production problems and outages, and the associated after-hours, nights, and weekend work that is required.

Rules

The first version of the rules for XP were published in 1999 by Don Wells. Twenty-nine rules are given in the categories of:

- Planning
- Managing
- Designing
- Coding
- Testing

Planning Rules

The first category is planning. The first rule in this category is that user stories are written. User stories document the use cases for the system being built and create time estimates for the release planning meeting. User stories replace the need for large requirements documents, and are written by the customers as things that the system needs to do for them.

User stories are in the format of about three sentences of text written by the customer in the customer's language, and are not meant to be technical. User stories also help drive the creation of the acceptance tests. One or more automated acceptance tests should be created to verify the user story has been correctly implemented.

A release planning meeting is used to create a release plan, which lays out the overall project. The release plan is then used to create plans for each individual iteration. It is important in this meeting for technical people to make technical decisions, and business people to make business decisions. The essence of the release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks. An ideal week is how long you imagine it would take to implement that story if you had absolutely nothing else to do. The customer then decides which story is the most important or has the highest priority.

The development team needs to release iterative versions of the system to customers often. Some teams deploy new software into production every day. At the very least, you'll want to get new software into production every week or two. At the end of every iteration, you will have tested, working, production-ready software to demonstrate to your customers. The customers will then decide whether to put that release into production. Iterative development adds agility to the development process. Divide your development schedule into a series of iterations of one-to-three weeks in duration. You should keep the iteration length consistent, as this sets the pace for your project. It is this constant that makes measuring progress and planning simple and reliable in XP.

You shouldn't schedule your programming tasks in advance. Instead, have an iteration planning meeting at the beginning of each iteration to plan out what will be done. Just-in-time planning is an easier way to stay on top of changing user requirements. An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks. Each iteration is between one and three weeks in length, and user stories are chosen for this iteration by the customer from the release plan, in order of the most valuable to the customer.

Any failed acceptance tests from the previous iteration are also selected to be fixed. The customer selects user stories with estimates that total out to the project velocity of the last iteration. The user stories and failed tests are broken down into the programming tasks that will support them. Programming tasks are written for each user story. While user stories are in the customer's language, tasks are in the developer's language.

Managing Rules

Communication is very important to an XP development team. You can make communication on your team more effective just by removing any dividing barriers between desks to make it easier for people to talk. The ideal working environment is an open-plan area where desks and computers are arranged to make pair programming easier. The team should either use shared computers or have their computers are set up with a consistent development environment, so that code can be worked on any machine with minimal disruption. Try to include a large area for daily stand-up meetings, and add a conference table that gives you a home for group discussions that occur spontaneously throughout the day.

Being able to see the discussions encourages people to listen or join in the discussion when they have a stake in the outcome. Adding white boards for design sketches and important notes, or blank walls where user story cards can be taped, creates additional channels for communication.

To set your pace for a project, you need to take your iteration seriously. You want the most complete, tested, integrated, production-ready software you can get at each iteration. Incomplete or buggy software represents an unknown amount of future effort, so you can't measure it. If it looks like you're not going to be able to get everything finished by the iteration end, have an iteration planning meeting and re-scope the iteration to maximize your project velocity.

Even if there is only one day left in the iteration, it’s best to get the entire team refocused on a single completed task than on many incomplete tasks. Working lots of overtime sucks the life and motivation out of your team. When your team becomes tired and demoralized, they will get less work done—no matter how many hours are worked. You can't make realistic plans when your team does more work this month and less work next month, so instead of pushing people to do more than humanly possible, use a release planning meeting to change project scope or timing. The purpose of a stand-up meeting is to get the whole team to communicate so everyone knows what everyone else is doing.

A stand-up meeting every morning is used to communicate problems, solutions, and promote team focus. Everyone stands in a circle to avoid long discussions. It is more efficient to have one short meeting that everyone is required to attend than many meetings with a few developers each. During a stand-up meeting, developers report at least three things: what was accomplished yesterday, what will be attempted today, and what problems are causing delays. The daily stand-up meeting is not another meeting to waste people's time; it will replace many other meetings, giving a net savings several times its own length.

The project velocity is a measure of how much work is getting done in your project. To measure the project's velocity, simply add up the estimates of the user stories that were finished during the iteration. You also total up the estimates of tasks finished during the iteration. Both of these measurements are used for the iteration planning. During the iteration planning meeting, customers are allowed to choose the same number of user stories equal to the project's velocity measured in the previous iteration. Those user stories are broken down into technical tasks, and the team is allowed to sign up to the same number of tasks equal to the previous iteration’s project velocity.

This simple mechanism allows developers to recover and clean up after a difficult iteration, and averages out estimates. Your project velocity goes up by allowing developers to ask the customers for another story when their work is completed early and no clean-up tasks remain. You should try to move people around in the team to avoid serious knowledge loss and coding bottlenecks. If any one person on your team can work in a given area, and that person leaves or just has too much work to do, you'll find that your project progress reduces to a crawl.

Cross-training is often an important consideration in companies trying to avoid islands of knowledge, which are so susceptible to loss. Moving people around the code base in combination with pair programming does your cross-training for you. Instead of one person who knows everything about a given section of code, everyone on the team knows about the whole system. A team is much more flexible if everyone knows enough about every part of the system to work on it. Instead of having a few people overloaded with work while other team members have little to do, the entire team can be productive.

The Extreme Programming methodology isn't perfect, and it won't fit for all organizations and teams. Follow the XP rules to start with, but do not hesitate to change what doesn't work for you.

This doesn't mean the team can do whatever they want, though—rules have to be followed until the team decides to change them. All of your developers must know exactly what to expect from one another, and having a set of rules is the only way to set these expectations.

Design Rules

A simple design always takes less time to finish than a complex one, so always do the simplest thing that could possibly work next. If you find something that is complex, replace it with something simple. It's always faster and cheaper to replace complex code now, before you waste more time on it. A system metaphor is a story that everyone, including your customers, programmers, and managers, can tell about how the system works. It's a naming concept for classes and methods that should make it easier for a team member to guess the functionality of a particular class or method from its name only.

The metaphor should be helpful in figuring out the overall design of the system. The metaphor should also help the team find a common vocabulary, and help everyone reach agreements about your requirements.

When developers are faced with a problem they don't know the answer to straightaway, create spike solutions to figure out the answer. A spike solution is a very simple program designed to explore potential solutions. Build a spike only to address the problem under examination, and ignore all other concerns. Most spikes are not good enough to keep, so expect to throw them away.

The goal is reducing risk of a technical problem or increasing the reliability of the user story's estimate. When a technical difficulty threatens to hold up the system's development, put a pair of developers on the problem for a week or two to help reduce potential risk. You should aim to keep the system uncluttered with extra code that you think may be useful later on. We're all tempted to add functionality now rather than later, because we see exactly how to add it, or because it would make the system so much better. However, we need to constantly remind ourselves that we are not going to actually need it.

Extra functionality will always slow us down and squander our resources. Keeping our code ready for unexpected changes is about simple design. Adding extra flexibility beyond what you need now always makes the design more complex. Refactoring is a control technique for improving the design of existing code bases. This is essentially about applying a series of small behavior-preserving transformations to your code. The cumulative effect of making lots of these code transformations is quite significant. By making them in small steps, you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring, which allows you to gradually refactor a system over an extended period of time.

Coding Rules

Code must be formatted to agree to coding standards. It's these coding standards that keep the code consistent and easy for the entire team to read and refactor. Code that looks the same also helps to encourage collective code ownership. It used to be quite common for a team to have a coding standards document that defined how the code should look, including the team's best practices for styling and formatting. The problem with this is that people rarely read them, let alone follow them. These days, it's much more common to use a developer productivity tool to automatically guide the user in best practices.

Popular tools in use today, certainly from a .NET perspective, are ReSharper from JetBrains, CodeRush from Dev Express, and JustCode from Telerik. These are all paid-for solutions, though. If you want to use a free alternative, then you can look at StyleCop for the .NET platform. Visual Studio also has its own versions of some of these tools built in, but it’s quite common to supplement Visual Studio with an additional add-on.

Other development platforms will have their own variants of these tools, either as separate additions to their environments, or built in to their IDEs. These tools are so unbelievably powerful that it really makes it frictionless to write code that conforms to a set of coding standards.

When you create a unit test before writing out your code, you'll find it much easier and faster to create the code. The combined time it takes to create a unit test, and then create some code to make it pass that test, is about the same as just coding it out straightaway. Creating unit tests helps the developer to really consider what needs to be done, and then the system's requirements are firmly nailed down by the tests. There can be no misunderstanding the specification written in the form of executable code, and you have immediate feedback while you work.

It’s often not clear when a developer has finished all the necessary functionality, and scope creep can occur as extensions and error conditions are considered, but if you create your unit tests first, then you know when you are done. A common way of working while pairing with another developer is to have one developer write a failing test, and then the other developer to write just enough code to make that test pass. Then, the second developer writes the next failing test, and the first programmer writes just enough code to make that test pass. It almost feels like a game when you work in this way. I worked this way for quite a while when I was working for an internet bank, and once you get a good pace with your programming pair, you can become really productive really quickly.

Under XP, all code to be sent to production should be created by two people working together at a single computer. Pair programming increases software quality without impacting delivery time. It can feel counter-intuitive at first, but two people working at a single computer will add as much functionality as two people working separately, except that it will be much higher in quality, and with increased code quality comes big savings later on. The best way to pair programming is just to sit side-by-side in front of the monitor, and slide the keyboard back and forth between the two. Both programmers concentrate on the code being written.

Pair programming is a social skill that takes time to learn when you’re striving for a cooperative way to work that includes give and take from both partners, regardless of corporate status.

Without force-controlling the integration of code, developers test their code and integrate on their machines, believing all is well. But because of parallel integration with other programming pairs, there's a combination of source code that has not been tested together, which means integration problems can happen without detection. If there are problems, and there is no clear-cut, latest version of the entire source tree, this applies not only to the source code, but to the unit test suite, which must verify the source code's correctness.

If you cannot get your hands on a complete, correct, and consistent test suite, you'll be chasing bugs that do not exist and overlooking bugs that do. It is now common practice to use some form of continuous integration system integrated with your source control repository. When a developer checks in some code, the code is integrated with the main source code tree and built, and the tests are executed. If any part of this process fails, the development team will be notified immediately so that the issue can be resolved.

It's also common to have a source control system fail at check-in if the compile and test run fails. In Team Foundation Server, for example, this is called a gated build. Once you submit your code to the repository, the code is compiled on a build server and the tests are executed. If this process fails for any reason, the developer would not be able to check-in their code. This process helps to ensure your code base is in a continual working state, and of high quality. Developers should be integrating and committing code into the source code repository at least every few hours, or when they have written enough code to make their whole unit test pass. In any case, you should never hold onto changes for more than a day.

Continuous integration often avoids diverging or fragmented development methods, where developers are not communicating with each other about what can be reused or can be shared. Everyone needs to work with the latest version, and changes should not be made to obsolete code, which causes integration headaches. Each development pair is responsible for integrating their own code whenever a reasonable break presents itself.

A single machine dedicated to sequential releases works really well when the development team is co-located. Generally, this will be a build server that is controlled by checking commits from a source control repository like Team Foundation Server. This machine acts as a physical token to control releasing, and also serves as an objective last word on what the common build contains. The latest combined unit test suite can be run before releasing, when the code is integrated on the build machine, and because a single machine is used, the test suite is always up-to-date. If unit tests pass 100 percent, the changes are committed. If they fail for any reason, then the check-in is rejected, and the developers have to fix the problem.

Collective code ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, improve designs, or refactor. No one person becomes a bottleneck for changes. This can seem hard to understand at first, and it can feel inconceivable that an entire team can be responsible for systems design, but it really makes sense not to have developers partitioned their own particular silos. For starters, if you have developers who only own their part of the system, what happens if that developer decides to leave the company? You have a situation where you have to try and cram a transfer of a lot of knowledge into a short space of time, which in my experience never works out too well, as the developers taking over are not building up a good level of experience in the new area.

By spreading knowledge throughout the team, regularly swapping pairs, and encouraging developers to work on different parts of the system, you minimize risks associated with staff member unavailability.

Testing Rules

Unit tests are one of the cornerstones of Extreme Programming. To start, you should decide on what unit testing framework you want to use. For .NET, for example, this might be NUnit or MSTest. Then you should test all the classes in your system, except trivial getters and setters, which are usually omitted. You would also create your tests first before writing the actual application code. This doesn't mean you have to write all of the tests for the entire system upfront, but before you tackle a new section, module, or class, you would develop a set of tests as you go along with the coding.

While building up your tests and writing code to make the tests pass, you will, before you know it, have created a robust testing suite that can be executed over and over again. Unit tests are released into the code repository along with the code they test, and code without tests should not be released into production. If a unit test is found to missing, then it should be created at that time and checked in. Normally, the biggest resistance to dedicating this amount of time to unit tests is a fast-approaching deadline, but during the life of a project, an automated test can save you hundreds of times the cost it takes to create it by finding and guarding against bugs.

The harder the test is to write, the greater your savings will be—and the more you need it. Automated unit tests offer payback far greater than the cost of their creation.

Another common misconception is that a unit test can be written in the last few months of a project. Unfortunately, without unit tests, the development drags on and eats up those last few months of the project, and then some. Even if the time is available, a good unit test suite takes time to evolve. Just having a suite of unit tests is meaningless if any of the tests fail for any reason. If you find a test is failing, you should fix it straightaway and not continue coding until all the tests are passing. It doesn't matter if it's your test or someone else's—strive to get it fixed there and then.

If you have an automated, continuous integration system set up, then you should be alerted straightaway if any of your tests start to fail. Even better, you'll be blocked from checking in the code if you have any check-in policies in force. When a bug is found, tests should be created to detect the bug and guard against it coming back. The debugging process also requires an acceptance test to be written to guard against it. Creating an acceptance test before debugging helps customers concisely define the problem and communicate that problem to the programmers. Given a failed acceptance test, developers then create unit tests to show the defects from a more source-code-specific point of view.

Failing unit tests give immediate feedback to the development effort when the bug has been repaired. When the unit test runs 100 percent, then the failing acceptance test can be run again to validate that the bug is fixed. Acceptance tests are created from user stories. During an iteration, the user story selected during the iteration planning meeting will be translated into acceptance tests. A customer specifies scenarios to test when the user story has been correctly implemented. A story can have one or many acceptance tests, whatever it takes to ensure the functionality works. Acceptance tests are black-box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are the highest priority. Acceptance tests also use regression tests prior to the production release. A user story is not considered complete until it has passed all of its acceptance tests.

Extreme Programming Diagram

Now that we have covered Extreme Programming in detail, let's express some of what we have seen in an easy-to-understand diagram.

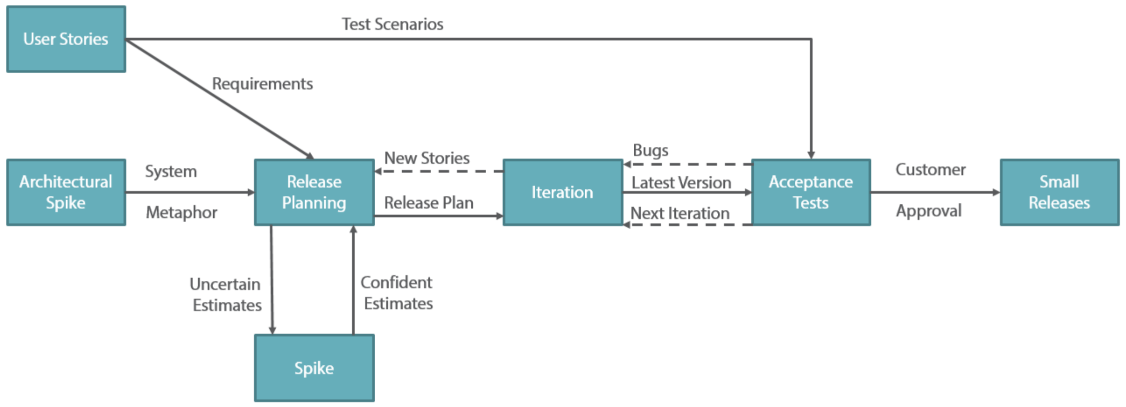

Figure 4 Extreme Programming (XP) Diagram

Here we have some of the different stages of XP: User Stories, Architectural Spikes, Release Planning, Development Spikes, Iterations, Acceptance Tests, and Small Releases. From the user story-writing stage, we end up with a set of requirements and a series of test scenarios that form our acceptance tests.

From the architectural system spike, we end up with a system metaphor, which is a story that everyone—customers, programmers, and managers—can tell about how the system works. During the release planning phases, we determine what requirements are included in which near-term releases, and when they should be delivered. The customers and developers are both part of this.

If we are uncertain about particular estimates, we can create a spike application where developers spend a constrained amount of time writing a small example program to quickly solve the problem, and therefore provide a more confident estimate.

The release plan then feeds into a development iteration where the code and unit tests are developed. Any new stories that come out during the iteration feed back into the release planning. From the iteration, you should have a working piece of software. This software should pass the acceptance tests set out from the user stories. If there are any bugs, then they are fixed by the developers.

There will typically be multiple iterations for a project. Once the acceptance tests pass from the iteration and the customer approves the system that has been developed in the iteration, a small release can take place, which gives the users access to real working code where they can start to reap the benefits early.

Chapter 6 Scrum

In this chapter, we'll take a look at the Scrum methodology. Scrum is an iterative development framework where value is delivered to the customer and users regularly. Let's explore an overview of Scrum, starting with its history.

Definition and History of Scrum

Scrum is an iterative and incremental Agile software development framework for managing product development. It defines a flexible, holistic product development strategy where a development team works as a unit to reach a common goal. Scrum is an agile way to manage a project, usually software development. In the Scrum world, instead of providing complete detailed descriptions about how everything is to be done on a project, much of it is left up to the software development team. This is because the team will know better how to solve the problem they are presented with.

Scrum relies on a self-organizing, cross-functional team. The Scrum team is self-organizing in that there are no overall team leaders who decide which person will be doing which task and how the problem will be solved. Those are issues that are decided by the team as a whole.

Scrum was conceived by Ken Schwaber and Jeff Sutherland in the early 1990s, who published a paper to describe the process. The term "scrum" is borrowed from the game of rugby to stress the importance of teams, and illustrates some analogies between team sports like rugby, and being successful in the game of new product development.

The research described in their paper showed that outstanding performance in the development of new, complex products is achieved when teams (small, self-organizing groups of people) are fed with objectives, not with tasks. The best teams are those that are given direction within which they have room to devise their own tactics on how to best move towards their shared objective.

Teams require autonomy to achieve excellence. The Scrum framework for software development implements these principles for developing and sustaining complex software projects. In February of 2001, Jeff and Ken were among 17 software development leaders who created a manifesto for Agile software development.

In 2002, Ken Schwaber founded the Scrum Alliance with Mike Cohn and Esther Derby, with Ken chairing the organization. In the years to follow, the highly successful, certified Scrum Master programs and its derivatives were created and launched In 2006. Jeff Sutherland created his own company, Scrum Inc., while continuing to offer and teach certified Scrum courses. Ken left the Scrum Alliance in the fall of 2009 and founded scrum.org to further improve the quality and effectiveness of Scrum, mainly through the Professional Scrum series. With the first publication of the Scrum Guide in 2010, and its incremental updates in 2011 and 2013, Jeff and Ken established a globally recognized body of knowledge.

Overview of Scrum

Scrum is an Agile process most commonly used for product development, especially software development. It’s a project management framework that is applicable to any project with aggressive deadlines, complex requirements, and a degree of uniqueness. In Scrum, projects move forward by a series of iterations called sprints. Each sprint is typically two-to-four weeks in length. When describing the Scrum framework, it is easy to split it into three main areas. They are:

- **Roles**, which include the product owner, Scrum Master, and Scrum team.
- **Ceremonies**, which include the sprint planning meeting, sprint review, and sprint retrospective meetings
- **Scrum artifacts**, which include the product backlog, sprint backlog, and the burn down chart.

Let's take a high-level look at these terms before we go into more detail.

The product owner is a project's key stakeholder and represents the users for whom you are building the solution. The product owner is often someone from the product management, a key stakeholder, or a user of the system. It is quite common for a business analyst with domain experience to take on the product owner role for the development team who will engage regularly with the customers.

The Scrum Master is responsible for making sure the team is as productive as possible, and achieves this by removing impediments to progress, by protecting the team from the outside, and so on. Their role is very much facilitating the team to steer their product to completion, and they act very much as a servant leader, fulfilling the needs of the team. The typical Scrum team has between five and nine people. A Scrum project can easily scale into the hundreds; however, Scrum can easily be used by one-person teams, and often is.

This term does not include any of the traditional software engineering roles such as programmer, designer, tester, or architect. Everyone on the project works together on tasks they are collectively committed to completing within a sprint. Scrum teams tend to develop a deep form of camaraderie and a feeling that “we're all in this together.” At the start of each sprint, a sprint-planning meeting is held, during which the product owner presents the top items on the product backlog to the team. The Scrum team then selects what they can complete during the coming sprint. That selected work is then moved from the product backlog to a sprint backlog, which is a list of tasks needed to complete the sprint.

At the end of each sprint, the team demonstrates the completed functionality at the sprint review meeting, during which the team shows what they have accomplished during the sprint. Typically this takes the form of demonstration of new features, but in an informal way. This meeting doesn't need to be very long or onerous to the development team, but is a good forum to demonstrate the work completed in the sprint. Also at the end of each sprint, the team conducts a sprint retrospective, which is a meeting where the team, including the Scrum Master and product owner, reflects on how well Scrum is working for them, and what changes they may wish to make for it to work even better.

Each day during the sprint, a brief meeting, called the daily Scrum, is conducted. This meeting helps set the context for each day's work and helps the team stay on track. All team members are required to attend the daily Scrum. Ideally, everyone in the team stands in a circle. Everyone is made to stand so that their update is brief. The team needs to answer three questions:

- What did I achieve yesterday?
- What do I plan to achieve today?
- Is there anything that is blocking me from achieving it?

The product backlog is a prioritized list of features containing every desired feature or change in a product. The term “backlog” can get confusing, because it's used for two different things; the product backlog is a list of the desired features for the product, and a sprint backlog is a list of tasks to be completed within that sprint.

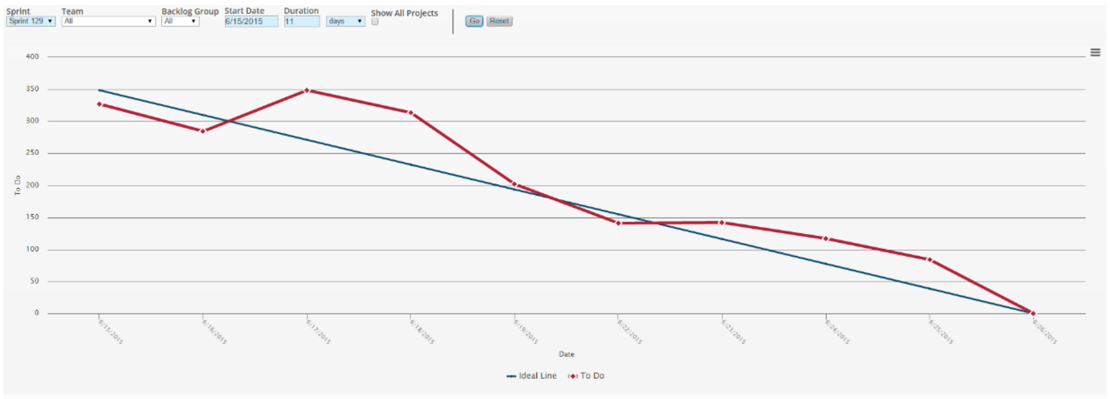

Figure 5 Example Burn down chart

On a Scrum project, the team tracks its progress against a release plan on the burn down chart, as you can see in Figure 5. The burn down chart is updated at the end of each sprint by the Scrum Master. The horizontal axis of the chart shows the teams, and the vertical axis shows the amount of work remaining at the start of each sprint. Work remaining can be shown in whatever unit the team prefers, such as story points, ideal days, or team days. Before we examine the roles, ceremonies, and artifacts in more detail, let's look at a visual representation of the Scrum process.

Scrum Diagram

The product backlog is a prioritized feature list containing every desired feature or change to the product. When you have a sprint planning meeting, items from the product backlog are selected to be implemented in the next sprint and placed into the sprint backlog. Once the sprint backlog has been identified from the product backlog, the team enters a two-to-four week sprint where they implement the items in the sprint backlog.

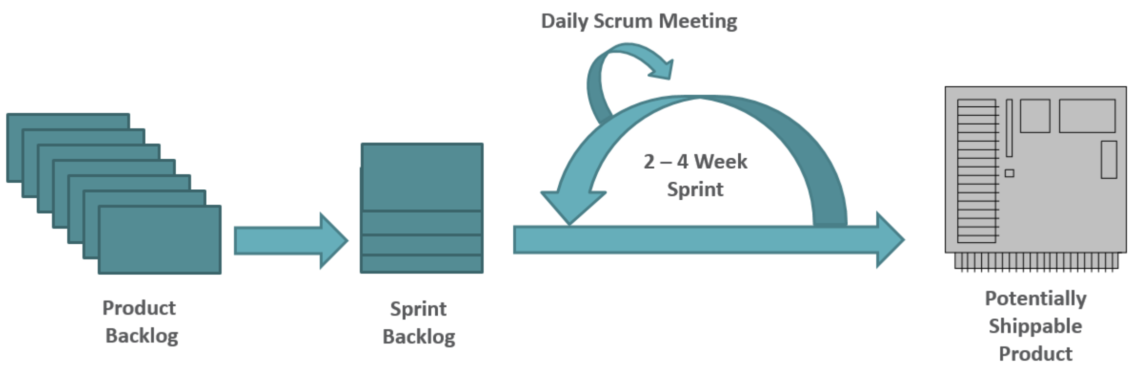

Figure 6 The Scrum Development Process

Each day during the sprint, the team holds a brief meeting, called the daily Scrum. This meeting helps set the context for each day's work and helps everyone stay on track. All the team members are required to attend the daily Scrum. At the end of the sprint, the team should have a potentially shippable product that could go into production and give value to the end user.

Now that we've had an overview of Scrum, let's look at each of the roles, artifacts, and ceremonies in more detail.

Scrum Roles

Scrum defines three main roles:

- Product owner
- Scrum master
- Scrum team

Normally, the Scrum Team's product owner is the project's key stakeholder, but it could also be a business analyst who works closely with the business and the users of the system. Part of the product owner's responsibility is to have a vision of what he or she wishes to build, and convey that vision to the rest of the Scrum team. The product owner is key to successfully starting any Agile software development project. The product owner works by maintaining the product backlog, which is a prioritized feature list for the product.

The product owner is commonly a lead user of the system or someone from marketing or product management, or anyone with a solid understanding of the users, the marketplace, the competition, and the future trends for the domain or type of system being developed. This could also be a business analyst who has excellent grasp of the business domain. The product owner prioritizes the product backlog during the sprint planning meeting.

It is the development team that selects the amount of work that they believe they can do during each sprint and how many sprints will be required. It is not the responsibility of the product owner to tell the development team how much work they should do in a sprint or how many sprints are required to complete the work. This should come from the rest of the development team, who will be doing the actual estimates. Requirements are allowed to change within Scrum—and this change is encouraged—but these changes should come outside the sprint, and be ready for the next sprint planning meeting. Once a team starts on a sprint, it should remain completely focused on delivering the work for that sprint.

The product owner role requires an individual with certain skills, including availability to the team, business and domain knowledge, and good communication skills. It is important that the product owner is available to his or her team all the time, and that they are committed to doing whatever is necessary to build the best product. Business and domain knowledge is important for Agile product owners, because he or she is the decision-maker regarding what features the product will have. That means a product owner should understand the market, the customer, and the business, in order to make the right decisions. Communication is a large part of the product owner's responsibilities.

The product owner role requires working closely with the key stakeholders throughout the organization, so he or she must be able to communicate different messages to different people about the product at any given time. The Scrum Master is responsible for making sure the Scrum team lives by the values and practices of Scrum.

The Scrum Master is like a coach for the team, helping the team do the best work they possibly can. The Scrum Master can also be thought of as a process owner for the team, creating a balance with the project's key stakeholder, who is referred to as a product owner. The Scrum Master does anything possible to help the team perform at the highest level. This involves removing any impediments to progress, facilitating meetings, and doing things like working with the product owner to make sure the product backlog is in good shape and ready for the next sprint. The Scrum Master role is commonly filled by a former project manager or a technical team leader, but it can be anyone.

People who are new to the Scrum Master role sometimes struggle with the apparent contradiction of the Scrum Master, who is both servant leader to the team, and also someone with no authority as a team leader or manager. This contradiction disappears when we realize that, although the Scrum Master has no authority over Scrum team members directly, the Scrum Master does have authority over the process.

The Scrum Master is there to help the team in its use of Scrum. They're a bit like a personal trainer who helps you stick with an exercise workout. A good trainer will provide motivation, while at the same time making sure you don't cheat by skipping the hard exercise. A trainer cannot make you do any exercise you don't want to; instead, the trainer reminds you of your goals and how you've chosen to meet them. To the extent that the trainer does have the authority that has been granted by the client, Scrum Masters are much the same—they have authority, but the authority is granted to them by the team.

The Scrum Master can say to the team: “Look, we're supposed to deliver potentially shippable software at the end of each sprint. We didn't do it this time. What we can do is make sure we do better on the next sprint.” This is the Scrum Master exerting authority over the process. Something has gone wrong with the process if the team has failed to deliver something potentially shippable. But because the Scrum Master's authority does not extend beyond the process, the same Scrum Master should not say, for example: “Because we failed to deliver something potentially shippable after the last sprint, I want Kevin to review all the code before it gets checked in.”

Having Kevin review the code might be a good idea, but the decision is not the Scrum Master's to make. Doing so goes beyond authority over process and enters into how the team works. With authority limited to ensuring the team follows a process, the Scrum Master's role can be more difficult than that of a typical project manager.

Project managers often have the fall-back position of “do it because I say so.” The times when a Scrum Master can say that are limited, and restricted to ensuring that the Scrum process is being followed.

The team in a Scrum environment does not include any of the traditional software engineering roles such as programmer, designer, tester, or architect. Everyone on the project works together to complete the set of work they’ve collectively committed to complete within the sprint. Because of this cross-disciplinary nature, Scrum teams develop a deep form of team spirit.

Scrum Ceremonies

There are four ceremonies that the Scrum team will be involved with. These are the:

- Sprint planning meeting
- Sprint review meeting
- Sprint retrospective
- Daily Scrum

The sprint planning meeting is attended by the product owner, Scrum Master, and the entire Scrum team. Outside stakeholders and users may attend if they are invited by the team, but generally they won't be attending this meeting. During the sprint planning meeting, the product owner describes the highest priority features to the team. The team should then ask enough questions so they can turn a high-level user story of the product backlog into a more detailed set of tasks for the sprint backlog.

The product owner doesn't have to describe every item being tracked in the product backlog. A good guideline is for the product owner to come to the sprint planning meeting prepared to talk about two sprints’ worth of product backlog items. This means that if the team is likely to finish what they thought they would get done in one sprint, the product owner is prepared with details of additional work and priorities.

Each sprint is required to deliver a potentially shippable product by the end of its duration. This means that at the end of each sprint, the team has to produce a coded, tested, and usable piece of software. At the end of each sprint, a sprint review meeting is held, and during this meeting, the Scrum team shows what they have accomplished during the sprint. Typically, this takes the form of a demo of the new features. This meeting should be quite brief and not take up too much of everyone's time, as it'll also be attended by product customers and management, whose time can be limited.

Participants in the sprint review typically include the product owner, the Scrum team, the Scrum Master, management, customers, and developers from other products. The product is assessed against the sprint goal determined during the sprint planning meeting. Ideally, the team has completed each product backlog item brought into the sprint, but it's more important that they achieve the overall goal of the sprint. No matter how good a Scrum team is, there is always opportunity to improve.

Although a good team will be constantly looking for improvement opportunities, the team should set aside a brief, dedicated period at the end of each sprint to deliberately reflect on how they are doing and find ways to improve. This takes place during the sprint retrospective meeting. The retrospective is normally the last thing to be done on a sprint. The entire team, including both the Scrum Master and the product owner, should participate. A retrospective meeting should last for up to an hour, which is usually quite sufficient. However, occasionally a hot topic will arise or a team conflict will escalate, and a retrospective could take longer.

During a retrospective meeting, the team should answer the following questions:

- What should we start doing?
- What should we stop doing?
- What should we continue doing?

The Scrum Master can facilitate the sprint retrospective meeting by asking everyone to just shout out ideas during the meeting. The Scrum Master can go around the room asking each person to identify any one thing to start, stop, or continue. After an initial list of ideas has been brainstormed, teams will normally vote on specific items to focus on during the next sprint.

The daily Scrum meeting is held every day, preferably in the morning. This meeting is very important, as it allows the team to understand where everyone else is within the sprint. Everyone stands in a circle during the meeting, ensuring their updates are kept brief. The team members have to answer three questions:

- What did you achieve yesterday?
- What will you achieve today?
- Is there anything blocking you?

If anything is blocking you, then you can work with the Scrum Master to resolve the issue to enable you to continue.

Scrum Artifacts

As part of the Scrum process, there are three main artifacts you will use, besides the actual delivered product. These are the:

- Product backlog
- Sprint backlog
- Burn down chart

The product backlog in Scrum is a prioritized feature list containing short descriptions of all the functionality desired in the product. When applying Scrum, it's not necessary to start a product with a lengthy upfront effort to document all the requirements, as you would in Waterfall. Typically, a Scrum team and its product owners will begin by writing down everything they can think of for the backlog prioritization. This product backlog is almost always more than enough for the first sprint.

The Scrum product backlog is then allowed to grow and change as more is learned about the product and its customers. A typical Scrum backlog comprises the following different types of items:

- Features
- Bugs
- Technical work
- Knowledge acquisition

The main way for a Scrum team to express features on the product backlog is in the form of user stories, which are short, simple descriptions of the desired functionality told from the perspective of the user. For example, a pharmacist can dispense products from a customer's prescription, which then appear on the customer's dispense items record.

There's also no difference between a bug and a new feature. Each describe something different that the user wants, so bugs are also put into the product backlog.

Technical work and knowledge acquisition activities also belong in the backlog. An example of technical work would be upgrading all developers' workstations to Windows 8, or migrating to a continuous integration server for continuous delivery.

An example of knowledge acquisition could be a backlog item about researching various JavaScript libraries and then making a technical decision. This may result in a small technical spike to solidify this knowledge. The product owner shows up at the sprint planning meeting with a prioritized product backlog and describes the top items to the team. The team then determines which items they can complete during the next sprint, and moves items from the product backlog to the sprint backlog. In doing so, they expand each product backlog item into one or more sprint backlog tasks so they can more effectively share work during the next sprint.

The sprint backlog is a list of tasks identified by the Scrum Master to be completed during the sprint. During the sprint planning meeting, the team selects some number of product backlog items, usually in the form of user stories, and identifies the tasks necessary to complete each story. Most teams also estimate how many hours each task will take for someone on the team to complete. It's important that the team selects the items and the size of the sprint backlog. Because there are other people committing to completing the tasks, they must be the people to choose what they are committing to during the sprint.

The sprint backlog can be maintained as a spreadsheet, but it's also possible to use your bug-tracking system or any number of software products designed specifically for Scrum or Agile. Team Foundation Server with the Scrum template, Jira, and VersionONE are common options to choose from.

During the sprint, team members are expected to update the sprint backlog as new information is available, but minimally, once per day. Many teams will do this during the daily Scrum. Once each day, the estimated work remaining in the sprint is calculated and graphed by the Scrum Master, resulting in a sprint burn down chart.

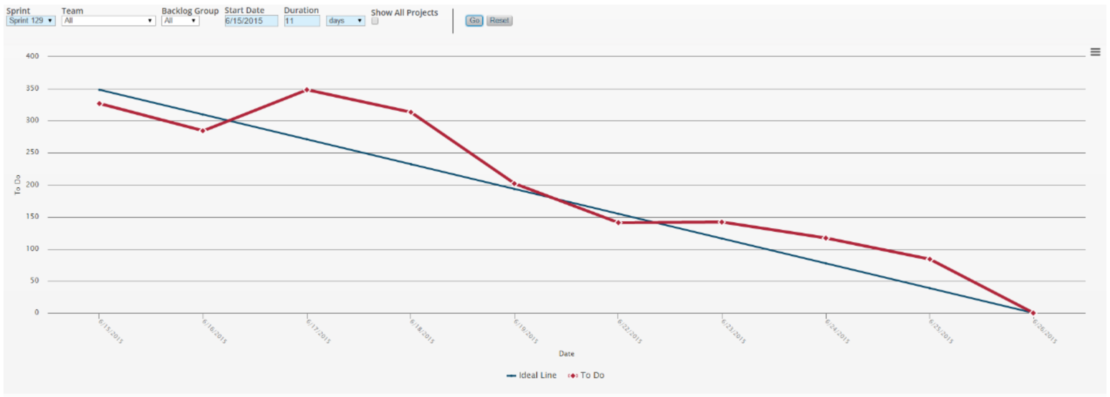
Sprint: Sprint 129
Team: All
Backlog Group: All
Start Date: 6/15/2015
Duration: 11 days
Show All Projects

Figure 7 Example Burn down chart

The team does its best to pull the right amount of work into the sprint, but sometimes too much or too little work is pulled in during the planning. In this case, the team needs to add or move new tasks. On a Scrum project, the team tracks its progress against a release plan on a release burn down chart.

The Scrum Master publishes the burn down chart at the end of each sprint by the Scrum Master. The horizontal axis of the chart shows the sprints, and the vertical axis shows the amount of work remaining at the start of each sprint. Work remaining can be shown in whatever unit the team prefers, such as story points, ideal days, or team days.

The burn down chart is an essential part of any Agile project, and is a way for the team to clearly see what is happening and how progress is being made during each sprint. One issue that may be noticed in the burn down chart is whether the actual work line is above or below the ideal work line, and this depends on how accurate the original estimates were. This means that if your team constantly overestimates time requirements, the progress will always appear ahead of schedule. If they constantly underestimate time requirements, they will always appear behind schedule.

Extreme Programming vs. Scrum

Now that we have taken a look at both Extreme Programming and Scrum, let's take a look at some of the main differences between the two. Scrum teams work in iterations called sprints, and these sprints are generally between two and four weeks in duration, although there's nothing stopping you from having a one-week sprint if you have a small team.

Having such a short sprint can be problematic, though, if you have to fit planning meetings, sprint reviews, and retrospectives all into one week. Extreme Programming teams work in iterations, and these iterations usually last for a week or two. Once a sprint has started under Scrum, the Scrum team doesn't allow any changes to that sprint until they are finished.

The team will continue as planned to the end of the sprint, and then do any pre-planning as necessary for the next sprint. With XP, teams are much more amenable to change in their iteration. If a change is required, the team will hold another planning session and adjust their iteration accordingly. In Scrum, the product owner prioritizes the product backlog, but the team determines the sequence in which they will develop the backlog items.

The team is trusted and expected to set their own pace and workload within the sprint. The backlog will be prioritized, which does allow the team to work on the high-value items first, but the order for these high-value items to be implemented is chosen by the team.

In XP, the teams work in a strict order of priorities as set out in the planning sessions, and tend not to deviate from that order. Scrum does not prescribe any engineering practices for the developers, as it is more of a lightweight project management framework. XP, on the other hand, is a very engineering-based methodology that defines many engineering practices, like test-driven development, pair programming, and continuous integration.

XP comes with many rules that can be hard for new teams to adopt. In my experience, what tends to happen is that teams adopt Scrum, as it is a lightweight framework for managing your Agile project, and then introduce different engineering practices from XP as deemed necessary. For example, at the time of writing this book, I am working on a Scrum team where we do test-driven development and continuous integration and delivery.

Closing Summary

In this book, we first looked at the more traditional Waterfall and V-Model development methodologies and discussed how they don't work very well for modern, large-scale projects, due to the "big bang" nature of deployments.

We then looked at the Agile development philosophy and how it focuses on delivering value to the customer in smaller increments.

Agile software development practices are based on four guiding values:

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

Next, we took a detailed look at the Extreme Programming development methodology. XP is an engineering-based discipline and incorporates many rules that need to be followed. It's a good framework, but teams can be put off by its initial complexity, and it can be difficult to follow for a team trying to transition into Agile from Waterfall.

Scrum, on the other hand, is a more lightweight project management framework that doesn't contain any engineering practices. What is quite common is for a team to adopt Scrum and then pick various engineering disciples from XP that suit the team, such as test-driven development and continuous integration.