# Table of Contents

Chapter 1: Agile Software Delivery Methods and How They Fit the Manifesto

**Understanding Scrum**
- Background
- Introduction to the mechanics of Scrum
  - Sprint Planning – part 1
  - Sprint Planning – part 2
- The Daily Scrum
- The Sprint Review
- The Sprint Retrospective
- Additional events

**XP - Extreme Programming**
- Background
- Introduction to the mechanics of XP
  - The planning game
    - Part 1 – Release planning
    - Part 2 – Iteration planning
  - Implementing the iteration plan
- Iteration demo
- Iteration retrospective

**Kanban and Lean Software Development**
- Background
  - Reducing waste
  - Single-piece flow
  - How Kanban/Lean fit into the Agile movement
- Introduction to the mechanics of Lean/Kanban
  - Getting started with Kanban
    - Step 1 – Make the team’s work visible
    - Step 2 – Make the work policies of the team explicit
    - Step 3 – Improve flow
    - Step 4 – Kaizen or continuous improvement

**Choosing the right framework**
- Designed for small teams
- They don't include product discovery phases
- Not all frameworks prescribe technical practices
- There are similarities with subtle differences
- Mixing and matching Agile methods

**Summary**

Index
Some people like a bit of background before they get started, if that’s you then you’re in the right place. In this chapter, we’re going to take a look at the various strands of the modern Agile movement and see how they’ve come together.

Alternatively, if you’re the kind of person who likes to get stuck in first and then get some context later, once you’ve tried a few things out, skip this chapter and go directly to the next one. In it, we’ll discover how to get your team up-and-running using the most popular Agile framework, Scrum. If that’s you, see you there.

Here we’re going to take a look at several Agile methods, including their backgrounds and how they fit the Agile Manifesto. Or perhaps, more importantly, how the Manifesto fits them because many Agile methods were developed before the Manifesto was written—the practical experience gained by the original members of the Agile Alliance is what gave the Agile Manifesto its substance.

Some have a number of prescribed artifacts and ceremonies; others are much less prescriptive. In most cases, there is no one-size-fits-all approach and most Agile practitioners will mix and match, for example, Scrum and XP. This chapter aims to help you decide which method might work for you.
We'll cover the following topics in this chapter:

- A detailed look at the most common methods: Scrum, Kanban, and XP
- A comparison of specific Agile methods
- Kanban for software is included in this chapter. Although it's technically a Lean approach to software development, Agile and Lean approaches are often combined
- How you can choose the right Agile framework

When the original 17 signatories to the Agile Manifesto came together in 2001 to form the Agile Alliance, they each brought with them ideas about how the industry could be changed for the better based on actual experiences. You see, many of them had already started shifting away from what they deemed heavyweight practices, such as the ones encouraged by Waterfall. Instead, they were putting new ideas into practice and creating SDLC frameworks of their own.

Among the signatories, that weekend were the creators of XP, Scrum, Dynamic Systems Development Method (DSDM), Crystal, Adaptive Software Development (ASD), Feature-Driven Development (FDD), and so on.

They initially called them "light" frameworks, to distinguish them from their heavyweight counterparts, but they didn't want the world to consider them to be lightweight. So, they came up with the term Agile, because one thing all of these frameworks had in common was their adaptive nature.

They noted at the time that some of their thinking was influenced by industrial product development and manufacturing.

The industrial heritage came from, predominantly, three sources:

- Product development and in particular how product development companies in the 1980s had been reducing the time to market for new products
- Engineering technical practices, which provided for better and, in some cases, fully automated quality assurance on a production line.
- Lean manufacturing as developed by Toyota Industries
In the following sections we’re going to look at three of the Agile methods, first up is Scrum.

**Understanding Scrum**

Scrum is the most popular framework among Agile teams; 58% of respondents to VersionOne’s 11th Annual State of Agile Report use pure Scrum. A further 10% are using a Scrum/XP hybrid.

**Background**

The following timeline shows a brief history of Scrum:

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1943</td>
<td>Lockheed form a special programs group to develop the XP-80 jet fighter. The group is called Skunk Works.</td>
</tr>
<tr>
<td>1986</td>
<td>“The New New Product Development Game” by Nonaka &amp; Takeuchi, published by HBR. A cross-functional, self-organizing, bureaucracy-free, product development team given 150 days to develop a new aircraft around a jet engine prototype. They did it 143.</td>
</tr>
<tr>
<td>1995</td>
<td>Scrum presented at OOPSLA. “The rugby approach” and “moving the Scrum downfield”</td>
</tr>
<tr>
<td>2001 2002</td>
<td>Scrum Alliance</td>
</tr>
<tr>
<td>2009</td>
<td>Scrum.org</td>
</tr>
</tbody>
</table>

In the paper, the two professors describe efforts by companies to try to speed up their product development life cycles to decrease their time to market. They observed that companies that were successfully doing this were employing some interesting alternative approaches.

These companies were assembling small teams of highly capable people with the right skills, setting the vision for them to build the next-generation product, giving them a budget and timeframe, and then getting out of the team’s way to let it do its thing.

Some observed characteristics of these teams included having all the skills necessary to carry out the job they were being asked to do—the essence of a cross-functional team. They were allowed to determine how they best carried out the work, so were self-organizing and autonomous. They used rapid, iterative development cycles to build and validate ideas.

Nonaka and Takeuchi called it the *rugby approach* because they observed product teams passing the product back and forth among themselves as it was being developed, much like a rugby team passes the ball when moving upfield. In a rugby game, the team moves as a unit and even though each team member has a specialty regarding position on the field and gameplay, any member of the rugby team can pick up the ball, carry it forward, and score a try or goal. The same was true of these product development teams—their contribution to the product development was specialist and highly collaborative.

In the section of their paper titled *Moving the Scrum downfield*, they list the common characteristics of the teams they observed as follows:

- **Built-in instability**: Some aspect of pressure was introduced, which encouraged the product development teams to think out-of-the-box and use an innovative approach to solving the problem.
- **Self-organizing project teams**: The teams were given the autonomy to decide how they carried out the task of solving the problem handed to them.
- **Overlapping development phases**: Instead of the normal sequential-phased development that you get with processes such as Waterfall, the teams worked iteratively, building quickly and evolving their product, with each iteration. Multiple phases overlapped, such that the following steps might be informed by the discoveries made in the previous one. In this way, the teams were able to gain fast feedback about what would and wouldn’t work.
• **Multilearning**: A trial-and-error learning culture is fostered, which allows team members to narrow down options as quickly as possible. They are also encouraged to diversify their skill sets, to create team versatility. Nonaka and Takeuchi called this multi-learning because they said it supported learning along two dimensions: traversing different layers of the organization (individual, team, unit, and group) and across various functions. This cross-pollination of skills is an aspect of cross-functionality we encourage today.

• **Subtle control**: The approach to managing these projects was very different. To create a space for the team to innovate, they realized command-and-control supervision wouldn't work. Instead, management would check in with the team regularly to check progress and give feedback, leaving the team to manage its work how it saw fit.

• **Organizational transfer of learning**: If and when the development life cycle began to move towards mass manufacture, the product development team would often be strategically placed in the wider organization to seed knowledge and assist with the preparation for production.

The approach described by Nonaka and Takeuchi has similarities to the Skunk Works projects started in World War II by Lockheed’s Advanced Development Programs division.

The Skunk Works team were originally tasked with designing and building a highly secret prototype jet fighter aircraft around a new jet engine developed by a British company, deHavilland. The work commenced on little more than a handshake, and the team was formed in a location separate from the rest of the group and given relatively free rein on how to proceed. They were given 150 days to complete their prototype; they finished it in 143.

Skunk Works was an official alias that originated with the Lockheed special projects development team during World War II. Due to a lack of room at its California premises, the team worked out of a well-guarded Circus Tent next to a manufacturing plant from which strange smells wafted in. The team associated the odor with a famous comic strip of the time called Li’l Abner, in which a mysterious moonshine factory deep in the woods brewed a terrible smelling concoction. The moonshine factory in the comic strip was called Skonk Works; over time this evolved into Skunk Works.
Lockheed’s Skunk Works took on many secret projects after the war finished. Their approach gained notoriety among other companies, including Apple who built the Macintosh in a Skunk Works type operation behind a restaurant in Cupertino. It also seems to have permeated through to the product development teams that Nonaka and Takeuchi were observing when they wrote their paper.

Everett Rogers (in *Diffusion of Innovations, 4th Edition*) points to the reason for isolating a project team and allowing them to take this crash approach to product development: it’s because companies operate as bureaucracies. The *stability and continuity* that a bureaucratic organization seeks are at odds with the instability needed to foster innovation. Most find it undesirable to disrupt their own, currently successful, business model. The Skunk Works approach fosters maximum creativity by isolating the teams away from the organizational mainstream, allowing them to innovate around both their process and product.

Some of these ideas would go on to influence Jeff Sutherland and Ken Schwaber when they started working on the Scrum framework in the early 1990s. In 1995, they formalized and presented it as a paper at the Business Object Design and Implementation Workshop held as part of *OOPSLA ’95 (Object-Oriented Programming, Systems, Languages, Programming, and Applications)* in Austin, Texas. In the following section we’ll introduce Scrum and talk through the basics with an overview of its characteristics, roles and events.

### Introduction to the mechanics of Scrum

Here are the basic characteristics and features of Scrum:

- **Planning style**: Empirical/adaptive
- **Delivery style**: Iterative/incremental
- **Iteration length**: Ranges between 1 and 4 weeks; a length within this range is initially chosen by the team and then fixed. The most popular is 2 weeks.
- **Values**: Commitment, courage, focus, openness, and respect
- **Roles**: Product Owner, Scrum Master, Development Team
- **Team size**: small, 5-9
- ** Artefacts**: Product Backlog, Sprint Backlog, Sprint progress tracking
- **Events**: Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective
- **Special features**: All events are time-boxed
- **Lacks**: A product/project/feature initiation phase and doesn’t specify technical practices
There are three roles in Scrum: the **Product Owner**, the **Scrum Master**, and the **Development Team**, which are defined as follows:

### The Scrum Team - Roles

#### Product Owner
- Holds the vision for the product and controls the budget
- Works to maximize value delivered by the team
- Clearly expresses what's to be done, makes the Product Backlog visible and transparent to all
- Sets priorities for the team in terms of which Product Backlog items to work on next
- Should be a single person, not a committee

#### Development Team
- Create working increments of "done" work
- Self-organizing - team decides how to deliver
- Cross-functional - have all the skills on the team necessary to do the job
- Individuals may have specialist skills, but are accountable as a team for delivery
- Scrum only recognises the title "developer" within the team
- Scrum doesn’t ask for or recognise sub-teams within the team

#### Scrum Master
- Coaches the team in the use of Scrum
- Coaches the organization how to get best value from its interactions with the team
- Facilitates events as requested or needed (Daily Scrum, Sprint Planning)
- Removes impediments to the team's progress
- Acts as a servant leader to the team

Team size is essential in a Scrum. Face-to-face communication is preferred by Agile teams; it's open and has a high bandwidth. The larger the team gets, the harder it becomes for each team member to know everything that is going on.
Scrum recommends a team of no fewer than five and no more than nine. Fewer than five (which includes the **Product Owner** and **Scrum Master**) and it's believed that the team will be limited in what it will be able to achieve. More than nine and the team's communications will become strained, and information may fall through the cracks.

The Scrum Guide defines three artifacts—**Product Backlog**, **Sprint Backlog**, and **Increment**:
Scrum uses an incremental approach to delivering. It achieves this by working in iterations known as Sprints. The recommended length for a Sprint is between a minimum of 1 week to a maximum of 4 weeks. Most teams opt for a 2-week Sprint:

![A Scrum Sprint](image)

The Sprint isn't the only aspect of Scrum that has a time limit, also known as *time-box*. All of events, such as Sprint Planning, the Daily Scrum, and so on, are also time-boxed. The aim of time-boxing an event is to ensure focus is maintained. The time-box for each event is proportional to the length of the Sprint and is set out in the Scrum guidelines.

**Sprint Planning – part 1**

The Sprint starts with Sprint Planning, a meeting where the whole team comes together. The aim of the first part of Sprint Planning is for the Development Team to forecast which items, from the top of the already prioritized Product Backlog, they think they can achieve in the coming Sprint.
Several factors influence their decision including the following:

1. The latest product increment.
2. The items in the Product Backlog.
3. The team capacity for the upcoming Sprint; for example, are any members on leave?
4. How did the team perform in the last Sprint?
5. Are there any work items being carried over from the last Sprint?

The usual process will involve the Development Team taking the next story from the top of the Product Backlog and discussing whether they think they can complete it as part of the coming Sprint. Usually, this begins with one of the team reading the story aloud, including the acceptance criteria. The Product Owner should be available for questions if the Development Team needs any clarifications. If, after discussion, our team believe they can complete it in the coming Sprint, they put it on the Sprint Backlog. They then take the next story from the Product Backlog and repeat the process.

Once the Development Team has determined the items from the Product Backlog they think they can achieve, the Scrum team as a whole works to set the Sprint Goal.

**Sprint Planning - part 2**

The second part of Sprint Planning is for the Development Team to determine how they will do the work. Deciding how involves discussion and breaking down the work into tasks, for each Product Backlog item in the Sprint. At the end of the Sprint Planning Meeting, the resulting set of work items, including the Sprint Goal is collectively known as the Sprint Backlog.

Once ready, the Sprint Backlog is added to the team's Scrum Board. For Scrum teams, this usually takes the form of physical board, such as a whiteboard or similar, and would look something like as follows:
**The Daily Scrum**

Once the Sprint is in progress, the Scrum team meets each day to coordinate their work. This meeting is called the Daily Scrum.

The Scrum team will congregate around the Scrum Board and discuss what they've achieved since they last met, what they will be working on until they next meet, and whether there are any problems, or if there is anything in their way.
The Development Team will update the board as necessary, moving tasks from left to right as applicable. This Scrum Board’s primary function is to help the team to coordinate their work, ascertain progress, and quickly uncover any assumptions or identify any risks in their plan. The team should remain focused on meeting the Sprint Goal; as such, this is a key inspect-and-adapt meeting for the team.

The meeting is timeboxed to 15 minutes to keep it purposeful and focused. Although the Scrum Guide doesn’t indicate this should be a standing meeting, many Scrum teams will stand for their Daily Scrum. This is something that’s been adopted from the Extreme Programming community. The aim of making it a standing meeting is to keep it short; people are less inclined to talk at length if they’re standing.

**The Sprint Review**

The Sprint Review is the first meeting held at the end of the Sprint cycle. The attendees include the Scrum team and stakeholders for the product. The working software for each completed User Story is demonstrated to the group. The Sprint Review aims to inspect and adapt the business value delivered by this latest increment, to see if we can optimize it further. Based on the feedback gained, the Product Owner can then adjust the backlog (adapt the plan) accordingly.

**The Sprint Retrospective**

The Sprint Retrospective usually follows on immediately from the Sprint Review and is the last meeting of the iteration. It’s an opportunity for the Scrum team to inspect and adapt its process. In general, the Scrum team will ask itself what went well during the Sprint, what didn’t go well, and what it can improve. They should consider aspects such as the team dynamics as well as processes and tools. The outcome of this meeting is for the team to come up with actionable improvements that it can carry out during the next Sprint.
Additional events

Most Scrum teams add additional events to their Scrum workflow. A good example is called backlog refinement, an event which some teams hold as regularly as once per week, or at least once per Sprint.

The backlog refinement meeting aims to look at the top stories on the backlog and prepare them to be worked on in an upcoming Sprint. Part of this preparation will include estimating the User Stories, which will tell us whether:

1. We have enough information to able to start working on the User Story
2. We've broken the story down into a small enough chunk; a User Story must be small enough to be completed in one Sprint

Estimates are usually given in Story Points; we'll talk more about those in the following two chapters.

XP - Extreme Programming

Extreme Programming (XP) is the second most popular framework, used by roughly 10% of Agile teams. XP stands out as one of the few Agile frameworks that prescribe technical practices.
Background

The following timeline shows a brief history of Extreme Programming (XP):

The 1990s was the beginning of another paradigm shift for the software industry as Object-Oriented Programming began to replace Structured Programming. As a way to explore how they would use this new approach, the Chrysler motor company decided to build their payroll system in Smalltalk, an OO programming language.

Initially invited to performance-tune the system because of his knowledge of Smalltalk, Kent Beck was asked to lead their software team in 1996. In Kent’s potted history of XP, he says that, before joining Chrysler, he had asked teams to do things he thought were important, such as testing and reviews. This time, at Chrysler, he felt there was a lot more at stake, so Kent asked the team to turn all the dials up to 10 on the things he thought were essential and not to bother with the ones he thought weren’t.

For instance, peer review is considered so important; when dialed up to 10 we do peer review all of the time. If all code is written while working with another programmer, we are code reviewing continuously. So XP makes the rule that all software that is destined for a production environment must be Pair Programmed.
He applied the same thinking to unit testing. XP deems unit testing so valuable it makes it a rule to write the unit test first before any code is written.

Writing tests first, also known as Test-Driven Development (TDD), is a practice that supports the creation of simpler code designs because just enough code is written to fulfill the test specifications. The resulting automated test suite also inspires the confidence to make subsequent changes to the specification tests and code, in the knowledge that if the tests still pass, other parts of the system are unaffected by the change.

So this is how XP got its name. It prescribes core programming practices and turns the volume on each up to maximum, taking them to the extreme.

Introduction to the mechanics of XP

Here are the basic characteristics and features of Extreme Programming:

- **Planning style**: Adaptive
- **Delivery style**: Iterative/Incremental, sustainable pace
- **Iteration length**: Ranges from 1 to 3 weeks, with a preference for the shortest possible
- **Values**: Communication, simplicity, feedback, courage, and respect
- **Roles**: Customer, Development Team
- **Team size**: small, 2-10
- **Artifacts**: Release plan, iteration plan, User Stories, tasks, CRC cards
- **Technical practices**: Pair Programming all production code, TDD, metaphor, refactoring, collective code ownership, Continuous Integration, daily builds, Spikes, sustainable pace
- **Events**: Release planning/iteration planning, daily standup
- **Special features**: Prescribes technical practices, gathers requirements with User Stories, promotes working at a sustainable pace
- **Lacks**: Compromise—it’s a fully committed, all-or-nothing approach
These are the roles XP recommends for a team:

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>Estimates stories. Breaks stories down into tasks. Works closely with the customer and other team members to implement stories and tests. Is responsible for ensuring the implemented story meets team standards.</td>
</tr>
<tr>
<td>Programmer</td>
<td>The tester role often works closely with other team members to help write a range of different tests, including functional and acceptance tests. They will also define and run exploratory tests. The tester is the test champion and advocates for sensible testing plans amongst the team. The tester role will hold the team to account if tests are failing or they aren’t up to standard.</td>
</tr>
<tr>
<td>Tester</td>
<td>Tracks the team progress against their current iteration plan. Will keep charts such as the iteration burn-down up-to-date. Will take action if things aren’t going to plan. These actions include enlisting the support of various team members, including the coach and customer to determine what can be done to get things back on track.</td>
</tr>
<tr>
<td>Tracker</td>
<td>Coaches the team. Schedules and facilitates meetings. Will record results of a meeting, particularly any actions that need to be taken. Will pass this information to the relevant people. Ensures the proper process is followed. Doesn’t tell the team what to do or when to do it.</td>
</tr>
<tr>
<td>Coach</td>
<td></td>
</tr>
</tbody>
</table>
XP requires that the **Customer** is part of the team; their involvement throughout the product development life cycle is key to its success. In XP, it's assumed that the customer has the most information about the value of what is being built. At the same time, it's expected that the Development Team has the best idea of how to implement that value and how much it will cost.

While this may be a radical shift from your current way of working, you’ll be amazed by the results of having the customer on the team—I’ve seen this work well many times. We may be the experts at building software, but our customer is the expert in their domain and is the person closest to the real-world problem that we’re trying to solve.

XP uses iterations to deliver working software incrementally. The iteration length is anywhere from 1 to 3 weeks. The team should pick an iteration length and fix it, only changing it if it's necessary:
The planning game

We start the beginning of each XP iteration with the planning game, an event in which the whole team participates. It is split into two phases, Release Planning and Iteration Planning, which we'll take a look at now.

Part 1 – Release planning

A new phase of work begins with release planning, a whole team meeting which is used to create a release plan. It commences with the customer/business introducing new problems for the team to solve. At this early ideation phase, the customer will often be thinking in the form of software features. These are written on index cards in the format of users stories.

XP emphasizes that business people are to make business decisions and technical people are to make technical decisions. The Development Team, therefore, leaves the customer to write the User Stories.

The following is an example User Story from a cinema ticketing system:

````text
As a cinema goer
I want to be able to purchase tickets for a particular film and session time
so that I can see the film I want to see at the time I want to see it.

Acceptance criteria:
- I can select a particular film
- I can select a particular session
- I can choose where to sit (Show available seats)
- I can pay using credit card
- I receive my tickets with the confirmation email
````

On the top half of the index card, the narrative of the User Story is written. It typically follows the format given here:

As a <certain type of user> I want to <perform some action> so that I get <some outcome>.

In the example User Story given here, the precise type of user is the cinema goer. The action they want to perform is to purchase tickets and reserve seats for a particular session. Finally, the outcome the Cinema goer would like is to watch the movie.
On the bottom half of the index card, the acceptance criteria are written. These help the team to understand what expected behavior is to be delivered as part of this User Story. These are the criteria that the customer will use to determine if the resulting software works as they were expecting.

User Stories are business problems formulated to fit on an index card. They are deliberately kept in this small format to enable the 3 Cs to happen: Card, Conversation, and Confirmation. In this way, active discussion occurs between the customer and the Development Team during implementation. Leading to fewer assumptions being made when compared to traditional requirements gathering. We'll discuss User Stories in more detail in Chapter 4, Gathering Agile User Requirements.

For each User Story, the Development Team will gather enough information from the customer so that they can estimate it. At the release planning level, estimate in ideal weeks. An ideal week is a week where we can focus entirely on implementing the User Story without any distractions. Estimates should fall between one and three ideal weeks. If the User Story is bigger than three weeks, it should be broken down. If it is smaller than one week, then it may need to be combined with others.

It's not unusual in a release planning meeting to see the User Stories spread around the table. Viewing all the stories like this helps the team absorb the bigger picture and will ultimately make it easier for the customer to identify any gaps.

With index cards, it's easier for both the customer and the Development Team to move the cards around and start to formulate a plan. Priority is given to stories with the highest business value.

The outcome of release planning is the release plan, a deck of User Stories written on index cards, estimated by the Development Team and prioritized by the business.

Work on the release plan begins with the preparation for the next iteration; this starts with the iteration planning meeting.
Part 2 – Iteration planning

Phase 2 of the planning game focuses on planning the next iteration. The release plan is the principal ingredient to this process. If this is the first iteration, that’s all we need. If, however, this isn’t the first iteration, we’ll also need the following:

1. The team’s project velocity. Velocity tells the team how much stuff we can get done in one iteration. To calculate it we take the estimates for the work completed in the last iteration and total them. This total is then used by the team to help them determine the right amount of work to commit to for this iteration. User stories are estimated in ideal weeks, so if we completed User Stories with a total of five ideal weeks, our velocity would be 5.

   If the team doesn’t have a project velocity, that is, it’s their first iteration, they will have to use a degree of gut feeling. It will take at least 3-4 iterations before their velocity begins to stabilize.

2. As with Scrum, any User Stories that are still in flight from the previous iteration are carried over. This includes any unfulfilled acceptance tests or bugs. Work that is carried over usually takes precedence as it was previously prioritized higher than the other items on the Release Plan. These user stories are placed at the top of the Release Plan for this round of iteration planning.

With all these factors available to it, the team goes about planning the next iteration. It does this by selecting the next story from the top of the release plan and starts to break it down into smaller, more manageable chunks, known as tasks.

Tasks are estimated in terms of ideal days, which is how many days it would take to complete the task if there were no distractions. Half days can be used if necessary. The breakdown of tasks should be done by the people carrying out the work.

The team keeps selecting User Stories from the top of the release plan deck and breaking them down into tasks until they hit their project velocity, or they decide they can fit no more into this iteration.

Some teams assign tasks to developers during iteration planning, some teams don’t. It does depend on how you break down User Stories into tasks and whether your team operates as specialists or generalists. It’s more fluid if we don’t assign tasks and we just let the next available pair pick up the next item to be done.
Implementing the iteration plan

Once the planning is complete, the iteration begins. If a programmer doesn't already have a task, they take the next available one and find a partner. If necessary, design work will be carried out at this stage, including the use of Class Responsibility Collaborator (CRC) modeling of Object-Oriented Design (OOD), and other design tools such as UML Sequence Diagrams.

The pair will begin coding in the TDD way by writing a failing test. They then write the code that fulfills the test. Once the test is running, they then look to refactor the code, as the following diagram illustrates:

This cycle continues until the task is complete.

Each day the team will meet in the form of a Daily Standup; this is a 15-minute meeting where, as the name suggests, everyone stands. The aim is to coordinate the team's work. Each team member or pair will talk about what they've implemented since yesterday's standup, what they'll look to achieve before the next standup, and what, if anything, is in their way.

XPers aim to integrate early and often; they favor a practice called Continuous Integration (CI). They will typically commit work every few hours, providing, of course, all of their unit tests pass. Modern CI practices offer an automated way for the latest changes to be incorporated. Teams will often set up their CI server so that it automatically checks out the newest version, builds it, and runs the tests. Their practices and workflow shouldn't allow them to commit code and proceed unless all tests pass.
Once the story is complete, the software is made available for acceptance testing. The customer executes the scenarios they've created around the acceptance criteria of the User Story, so they can determine whether the software is complete. Many teams now automate their acceptance testing as part of their TDD approach, a technique known as **Acceptance Test Driven Development (ATDD)**.

**Iteration demo**

As with Scrum, the demo is an opportunity to invite stakeholders along so that they can see progress and give feedback. The focus is on a demonstration and inspection of the working software completed during the iteration. Information gathered from the demo can be fed into the release planning ahead of the next iteration, and if necessary changes can be made and the release can be re-planned.

**Iteration retrospective**

Similar to the idea of the Scrum Retrospective, the iteration retrospective is an opportunity for the team to take stock of how things have gone during the past iteration and determine whether it can make changes to its process for the better.

**Kanban and Lean Software Development**

In this section, we look at Lean Software Development and its origins in Lean Manufacturing. We'll first discuss the thinking that led to significant breakthroughs in responsiveness and quality on the Toyota production line. We'll then look at how we can apply those to software development.
Background

The following timeline shows a brief history of Lean and Kanban:

Lean Software Development and Kanban have their history in manufacturing and the work done by one company, in particular, Toyota Motor Corporation. In its effort to economically mass-produce affordable motor vehicles for people after the Second World War, Toyota made profound changes to the way it organized its production line.

Toyota realized early on that there was a significant waste in manufacturing, which added to the overall cost. They implemented two notable changes to typical mass production lines, which had a profound effect on reducing costs and significantly increasing quality.

1. Reduce all waste which didn't add value to the customer  
2. Focus on single-piece flow through the production line using a pull system
Reducing waste
Waste was identified at multiple levels, for example, overproduction of a particular part was seen as waste for two reasons. Firstly, space was needed to stockpile the part until it could be used. This cost money not only for storing it but also took up floor space that could be used for production. Secondly, sometimes problems in the manufactured parts weren’t found until they were combined with other parts and put to use further down the production line. If a problem were discovered that required the entire batch to be re-machined or re-manufactured, this would add significant time and money.

Single-piece flow
In a production line environment, there are multiple workstations each of which takes the work of preceding stations and combines it or enhances it before passing it on to the next workstation. In this way, each station adds value and passes it further down the production line until the product being built is complete.

In a traditional product line environment, the work was pushed from one station to the next. Sometimes the work was sent in batches. For example, the machine that built widget A, with some amount of retooling, also built widget B. Rather than lose time due to retooling, if you built batches of say 100 at a time, the machine operation was more efficient.

This can create unevenness in the production line, which can lead to fluctuations in the flow; sometimes a station further down the line could be waiting for the previous station to complete a batch of Widget As and retool because it needs two Widget Bs.

To solve these problems, Toyota perfected a system called Kanban. A Japanese word meaning signboard, Kanban is used in lean manufacturing to signal when a piece of work needs to be done. The profound change that Toyota’s employee Taiichi Ohno implemented in this signaling system was to reverse the flow of information on the production line. This means that stations further down the production line would send Kanban signals to the stations behind them that they needed certain parts made.
A Lean Production Line waits until there are orders for its product because the communication channel is reversed. The order being placed is a signal to the end of the production line that it needs to deliver more finished products. It does this by sending requests to the stations preceding it in the production line so that they can make and assemble the necessary parts for it to deliver a finished product.

The reversed flow creates a pull rather than push approach to manufacture and changes the dynamic of the system to focus on the whole system’s end-to-end flow. In layman's terms, this means that each station on the production line is concerned with carrying out the request it receives as promptly as possible to ensure the next station ahead of it is also able to do so.

Each station only carries out the work when it is requested; this is the Just in Time (JIT) of lean manufacturing. In any time between servicing requests, the workers at the station can clean their station and prepare themselves for any potentially busy periods ahead. They can also use this time to see if there is anything they can do to improve their process.
It is the responsibility of the entire production line to smooth out any unevenness in the flow. Workers should not try to compensate by deliberately operating their station even though they have no work requests, as this leads to unnecessary overproduction and stockpiling of inventory.

**How Kanban/Lean fit into the Agile movement**

Kanban, although not technically an Agile method, shares many similar attributes. The Toyota Production System (TPS) was founded on the principle that people were at its heart.

You may find some people are initially skeptical that it is possible to translate practices from Lean manufacturing to software product development. They argue that there are very few similarities between a production line and a software development process. Production lines are doing the same repetitive tasks over and over again in sequence, which is often associated with a predictive planning approach. However, using *Just In Time* manufacturing, Toyota has created and evolved an approach that makes their assembly line much more responsive and adaptive than any other.

The Lean approach to process management has already strongly influenced the thinking of those who formed the Agile Alliance. Many of the methods the practitioners created incorporate aspects of Lean already, such as Scrum's use of the Scrum Board to make work visible. XP and Scrum's use of iterations to manage batch size and create JIT thinking about the requirements and implementation of software products. These are just a couple of examples of how Taiichi Ohno and his work with others at Toyota have influenced how we build software.
Introduction to the mechanics of Lean/Kanban

Here are the basic characteristics and features of Lean/Kanban:

- **Planning style**: Lean/Adaptive
- **Delivery Style**: Flow/Incremental
- **Iteration length**: Doesn’t have time-boxed iterations
- **Principles**: Eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity in, see the whole
- **Roles**: Not prescribed, existing roles
- **Team size**: Not prescribed
- **Artefacts**: Kanban (message board)
- **Events**: Andon (stop the line), Gemba (go and see)
- **Special features**: Kaizen (Continuous Improvement), makes work visible, limits work in progress, works with your existing process, makes policies explicit, evolves empirically
- **Lacks**: A clear process framework of its own; instead it asks that we make our existing process visible and use the principles of Lean to improve it

There are no explicit roles defined by Kanban; assuming you have everyone necessary to do the job you’ve been asked to do, this is sufficient.

Start with your existing process. Map the workflow on a wall, making it visible so that everyone sees it. Use this sudden wealth of information on how your system works to makes changes and bit by bit evolve it into something that works better.

Getting started with Kanban

There are four steps to implementing Kanban within your team, which we’ll cover in the following sections.

Step 1 – Make the team's work visible

This involves creating a board that reflects the current process the team uses to deliver software. The work that the team currently has in progress and the stage it’s at also needs to be shown. The easiest way to do this is to write down each work item on an index card and place it in the appropriate area of the board.
For example, a Kanban board for a team with a simple workflow would look something like this:

```
<table>
<thead>
<tr>
<th>To-do</th>
<th>In progress</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Work item</td>
</tr>
<tr>
<td></td>
<td>Work item</td>
<td>Work item</td>
</tr>
<tr>
<td>Work item</td>
<td>Work item</td>
<td></td>
</tr>
<tr>
<td>Work item</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Step 2 – Make the work policies of the team explicit**

Teams often handle this by placing entry and exit criteria for each column to make them transparent. This will look something like the following:

<table>
<thead>
<tr>
<th>To-do</th>
<th>In progress</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>User story defined</td>
<td>Automated tests written</td>
<td>Released to staging</td>
</tr>
<tr>
<td>Estimated</td>
<td>Code reviewed/paired</td>
<td>Acceptance tested</td>
</tr>
<tr>
<td>Initial user scenarios</td>
<td>Exploratory tested</td>
<td>Ready to release</td>
</tr>
</tbody>
</table>

The policies form what is commonly known as the **Definition of Done (DoD)**. The DoD is a checklist that the team applies to each work item. It is a way for the team to ensure they’ve completed the work item to a satisfactory standard and therefore ensure they are the baking quality of their product.
Step 3 – Improve flow

With Kanban, as with the other Agile approaches, we want to add value as soon as we can, by delivering useful software as fast as possible. The reason for this is two-fold:

1. Most people don’t know what they need until they see it. It’s hard to imagine how the software will look and behave until it’s built. Everyone has a different version of it in their head. Once you have something working, you can start to get feedback from your customer. This is when the real work begins.
2. The domain we’re operating in is rapidly evolving. In business terms, 6 to 12 months is too long to wait for something. The way the business works and the rules that govern it could have easily changed within such a timeframe.

For work to start to flow across the board, we have to do two things:

1. The first step is to reduce the size of each work item so that it is as small as possible. Taiichi Ohno called this reducing the waste of unevenness (Muri). For us in the software industry, unevenness is reduced by managing and reducing probable variability, which we can also manage by keeping the work item as small as possible. This doesn’t mean that we’re eliminating all variability, just reducing the amount.
2. The second step is to switch to small batches. This can either be done as Scrum or XP does, using Sprints or Iterations. Alternatively, the more granular way is to start to manage Work In Progress (WIP) limits so that the team can focus their effort on the items currently being worked on and avoid the loss of time caused by context switching when multitasking. Assuming the items have been prioritized by value, this allows them to focus on completing the high-value items first.

Rather than thinking and working in iterations, the focus in Kanban is on optimizing the flow of work items through the system. To optimize flow two shifts in thinking have to happen. Firstly, we need to think of the system as a whole, the end-to-end process from idea to delivery of that idea. Secondly, instead of pushing the work through the system, we have to think of the system as pulling the work through it. When the system has the capacity, it pulls the next piece of work into it to be worked on.
This requires careful control; the aim is to avoid large batches of work that move as one cohesive block as this only encourages optimization at the local level. Instead, we carefully manage WIP and prevent overcapacity by limiting the number of items being worked on at any given moment.

To identify where you might have flow issues in your process, you can map your entire process out so that the entire process is explicit. This is similar to a technique known as Value Stream Mapping, but where a Value Stream Map is a snapshot in time, modeling your Kanban board precisely to your Value Stream allows you to observe and iron out any problems in flow in real time.

The following shows a Kanban board where our team has mapped out every step of the process:

<table>
<thead>
<tr>
<th>To-do</th>
<th>UX / Design</th>
<th>Code</th>
<th>Unit Test</th>
<th>Code Review</th>
<th>Test</th>
<th>Merge</th>
<th>Deploy Staging</th>
<th>UAT</th>
<th>Deploy Prod</th>
<th>Confirm</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td>Work item</td>
<td></td>
</tr>
</tbody>
</table>

Visualizing your work in this way will soon help you identify when there is too much work in progress.
For example, the team using the Kanban board in the following figure have identified that they have too much work to test and this is causing a logjam:

Blocks like this can have quite subtle effects. One observed result of allowing work to accumulate is to put all of the work items in the column into a single batch. We can then create a single software deployment and carry out testing on all the items at once. We think this will create efficiency in our testing, which it may do at a local level. However, with five work items in that column, the reality is that each work item will delay the others as we wait for testing to complete. If we find any problems with one or two them, the whole batch will be delayed and the overall flow will be significantly reduced.

To tackle this, determine if this is a one-off or whether a pattern is emerging. Either way it is important to take pressure off the people who are currently testing, by swarming on the work as a team. Once the workload is back to normal, the team can prevent this from happening again by placing a **WIP** limit on the test column. The following figure shows a Kanban board with WIP limits set. We'll talk about WIP in more detail in Chapter 8, *Tightening Feedback Loops in the Software Development Life Cycle*. 
**Step 4 – Kaizen or continuous improvement**

Once the team is up and running with the preceding three steps, the next thing for them to implement is Kaizen.

Kaizen is a Japanese word meaning continuous improvement. It is similar in concept to Scrum and XP’s retrospective event. The team is encouraged to take time to reflect on their work regularly, and where possible identify improvements to the way they carry it out.

**Choosing the right framework**

It’s no surprise that most Agile transformations start at the team level. Agile is a grassroots movement led by a bunch of software developers who knew, through their trials and tribulations, that there had to be a better way to build software.

In choosing the right framework for your team it will depend a lot on where you are on your journey.

If you’re a product/project team, Scrum is the place to start. Especially if you’re new to Agile, or if you have a mix of Agile understanding in your team members.

Scrum is simple to pick up because the framework gives you everything you need to get started. Plus, once you begin to master some of the basics of Scrum, it’s easy to add practices which enhance your agility.
This is the path that we will follow in this book, in it we will give clear guidance on how to build up our team’s practice of Scrum.

What follows in this section is a commentary on the three methods we’ve just overviewed where we took a quick look at the similarities and differences.

**Designed for small teams**

All of the Agile frameworks described in this chapter are designed for use by small teams. For Kanban, there is no prescribed team size, but if clear communication and coordination among team members is needed, then keeping the team size small is desirable.

An Agile team’s ideal size is often referred to as a two-pizza team, that is, the team is just big enough that two pizzas would be enough to feed it. This obviously depends on the size of the pizza!

**They don't include product discovery phases**

The other thing you’ll notice about each of these frameworks is they don’t explicitly define phases for product discovery/ideation. Few Agile methods do, DSDM being the standout. Instead, most prefer the team to manage this themselves. There are a number of techniques for doing so, including Design Thinking, User Story Mapping, and Impact Mapping. We’ll discuss these techniques later in the book in Chapter 10, *Using Product Roadmaps to Guide Software Delivery*.

**Not all frameworks prescribe technical practices**

Scrum, for example, doesn't specify technical practices. It's rumored that it did initially, to make the framework more effective, but Ken and Jeff pulled them before formally announcing it.

As a result around 10% of Scrum teams, according to VersionOne's 11th *Annual State of Agile Report*, incorporate some or all of the technical practices from Extreme Programming (XP).
There are similarities with subtle differences

Scrum and XP both explicitly involve the customer, Scrum with the Product Owner role, XP with the customer/business representative on the team. Lean Software Development emphasizes people are at its heart.

Scrum doesn't explicitly mention release planning, it assumes that the Product Owner will manage the backlog and the items nearest the top of the backlog are the ones of most value to the team. It assumes that the Product Owner will manage the backlog and the items nearest the top of the backlog are the ones of most value to the team.

Scrum and XP form batches of work using iterations. The following figure shows the Sprint Backlog—this is the batch in Scrum:
XP encourages its practitioners to err towards the smallest iteration possible. This stimulates incremental delivery thinking and moves teams away from the pitfalls of a mini-waterfall, where you execute a User Story in a series of handoffs just as you would a waterfall project.

Kanban doesn’t make use of iterations to batch work. Instead, it focuses on flow; achieved by keeping each work item as small as possible and by limiting the work in progress so that people can focus on one task at a time.

It is designed to work with your existing process, and because of its value-driven mindset, it will start to shift the Development Team’s thinking towards delivering something sooner rather than later.

The advantage of using Kanban at a more granular level, work item versus iteration, means that we can be even more adaptive and responsive to change. However, with great power comes great responsibility, and the Kanban approach does require more attention to detail.

For example, to remain this adaptive requires that the customer and team have an innate understanding of the overall objective and whether they are currently tracking towards it.

I often hear people discussing where and when you should use Scrum versus Kanban. For example, because of Scrum’s history in product development, it seems the logical choice when developing a new software product or making significant enhancements to an existing one.

For most people, Kanban seems better suited to a more ad hoc backlog, where there is typically little or no coherency between work items. This is often the case when the product is in maintenance mode (some would call this BAU or business as usual). However, we shouldn’t let Kanban’s apparent simplicity fool us; of the two approaches, Scrum and Kanban, Kanban is probably the more nuanced.

As we’ll see in Chapter 8, *Tightening Feedback Loops in the Software Development Life Cycle*, it can be used just as effectively as Scrum to build products, if not more so. Applying Lean thinking to product development increases flow and works particularly well when the top portion of the Product Backlog is prioritized and well defined.

**Mixing and matching Agile methods**

As we’ve seen in this chapter, there are a lot of similarities between Scrum, XP, and Kanban. No matter where we start as a team, most will start to combine the practices and thinking of one or more of these methods.
Sometimes we do it without realizing, for example, XP’s User Stories have become a universally accepted way to gather Agile requirements. Sometimes we do it explicitly because we want to enhance our Agile adoption with the benefits of a particular practice. An example is when Scrum teams enhance their workflow using Lean principles, something we’ll discuss in Chapter 8, *Tightening Feedback Loops in the Software Development Life Cycle*.

When we look at the practices that each method presents, we see a spectrum form. At one end of the spectrum, we have Kanban, a simple but nuanced approach for making work visible and then improving our workflow. At the other end, we have XP, where years of practical experience led the founders of that movement to insist on following a specific, disciplined approach.

XP can be too extreme for some as there are a lot of practices to adopt. This is why in his 2nd Edition book *Extreme Programming Explained*, Kent Beck re-categorized some of those disciplines as primary (ones we can introduce immediately) and others as a corollary (ones that we introduce later when we better understand the context for them).

Lean Software Development and Kanban could be seen as too light—just do what you’re doing now, but make it visible and seek to improve it by continuously eliminating waste in our process. Sounds simple, but understanding and optimizing our system isn’t as easy as it sounds and requires much practice.

Scrum can be seen as somewhere in the middle. It provides enough of a framework for people to start their journey to becoming great inspectors and adaptors. At the same time, it holds back from prescribing a huge change in technical practices that may cause change phobia.

So, start with Scrum and get used to incremental delivery. Then begin to apply good disciplines that bake quality in from the beginning rather than test for quality later. Do this by conducting a number of experiments to see what works for you in your context. We’ll look at how can do this in Chapter 7, *Software Technical Practices are the Foundation of Incremental Software Delivery*.

One final thought, we need to consider that most Agile frameworks focus on the delivery of software. Few include explicit phases, either at the beginning for the initiation of product ideas or at the end of release and deployment. We’ll cover techniques for both of these phases in this book, in Chapter 10, *Using Product Roadmaps to Guide Software Delivery* and Chapter 7, *Software Technical Practices are the Foundation of Incremental Software Delivery* respectively.
Summary

The modern Agile movement is the coming together of three different timelines from the software industry's industrial heritage: Product Development, Engineering, and Lean Manufacturing.

We looked at the three most popular Agile methods, and how they represent each of these timelines.

We discussed how all three methods have a degree of crossover or similarity. For instance, Scrum encourages a high degree of transparency, and Scrum teams often communicate this using a visible workspace. This is very similar to the concept in Kanban of making work visible.

We also discussed how many companies are improving their results by mixing and matching from two or more approaches.

In the next chapter, we're going to give a practical overview of how to get our team started with Scrum.
A
Acceptance Test Driven Development (ATDD) 22
Agile
    methods 2
C
characteristics, Scrum
    artefacts 6
    delivery style 6
    events 6
    lacks 7
    planning style 6
    roles 6
    special features 6
    team size 6
    values 6
Class Responsibility Collaborator (CRC) 21
Continuous Integration (CI)
    about 21
D
Dynamic Systems Development Method (DSDM) 2
E
Extreme Programming (XP)
    about 13, 14
    characteristics 15
    game, planning 18
    iteration plan, implementing 21
    iteration planning 20
    iteration retrospective 22
    mechanism 15
    release planning 18, 19
F
framework
    designed, for small teams 34
    product discovery phases, avoiding 34
    selecting 33
    technical practices 34
K
Kanban
    about 22, 27
    background 23
    continuous enhancement 33
    explicit team work policies, creating 29
    flow, enhancing 30
    mechanics 27
    team work 27
L
Lean Software Development (LSD)
    about 2, 22
    background 23
    mechanics 27
    single-piece flow 24, 26
    waste, reducing 24
O
Object-Oriented Design (OOD) 21
Object-Oriented Programming, Systems,
    Languages, Programming, and Applications
    (OOPSLA) 6
R
rugby approach 4
Scrum
about 3
background 3, 6
characteristics 6
Daily Scrum 11
events 13
mechanism 6, 9
Sprint Planning 9, 10
Sprint Retrospective 12
Sprint Review 12
Test-Driven Development (TDD)
about 15, 21
time box 9
Work in Progress (WIP)
about 31, 32