



intelliware
software development




eBook: Agile
Methodology Series



Table of Contents

| | |
|--|-----|
| <u>Chapter 1: What is Agile Development?</u> | 3 |
| <u>Chapter 2: 7 Myths of Agile Development</u> | 32 |
| <u>Chapter 3: Agile Story Writing</u> | 43 |
| <u>Chapter 4: Agile Estimating</u> | 65 |
| <u>Chapter 5: Agile Release & Iteration Planning</u> | 80 |
| <u>Chapter 6: Agile Room (Team) Dynamics</u> | 100 |





intelliware
software development






What is Agile Development?

Once primarily the domain of early adopters, Agile flavoured methodologies have steadily gained acceptance as a mainstream approach to software development.^{1,2,3} Although Agile has come to mean many different things to different people, at its core, it is a philosophy that guides a set of technically rigorous development practices. It has been adopted by a growing number of organizations to lower the risk that is inherent in software development and to deliver better software as defined by the end user. As more teams adopt Agile practices and more companies transform to Agile cultures, we continue to learn together about the best practices and benefits of developing software in an Agile manner.

This introductory paper is designed to provide a basic understanding of Agile software development.



The Foundations of Agile

A quick lesson in software development history provides context for both how and why Agile is important. Agile is typically seen as an evolutionary step forward from the “Waterfall” method, which is the traditional – and still most prevalent – way of developing software.



Top 10 Agile Links

1. agilemanifesto.org
2. en.wikipedia.org/wiki/Agile_software_development
3. agilealliance.org
4. codeproject.com/Articles/604417/Agile-software-development-methodologies-and-how-t
5. objectmentor.com/omSolutions/agile_what.html
6. agilemethodology.org/
7. mountaingoatsoftware.com/agile/new-to-agile-or-scrum
8. allaboutagile.com/what-is-agile-10-key-principles/
9. versionone.com/Agile101/Agile-Development-Overview/
10. i-proving.com

The Waterfall Method

Waterfall is a sequential approach to software development. As Figure 1 illustrates, it is a multi-step process with phases executed in a stepwise manner. First, software requirements are defined. This is followed by software design, then development and finally, integration and testing. Often steps are defined by gates, at which point some sort of approval, often involving a document, is required before the project can proceed to the next step. Hence, the approach is also often referred to as, “document driven development”.

Waterfall has obvious appeal. It provides an easily understandable process that seems to progress in a logical sequence. Moreover, it's a familiar process because it is similar to how we build other things, such as buildings and cars. The gates provide a form of risk management, in that the process cannot proceed to the next step without an authorization of the preceding step.

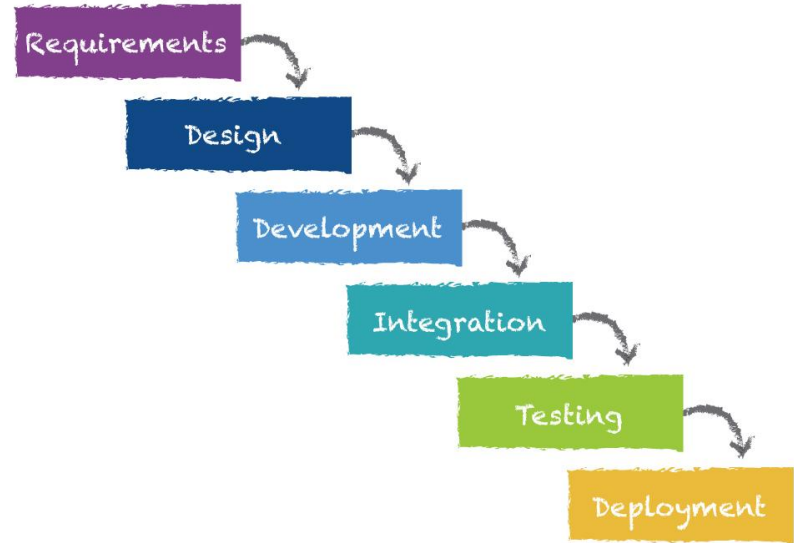


Figure 1: The Waterfall Process of Software Development

The Problem With Waterfall

The Waterfall approach works well if everything is straightforward, simple and predictable. But therein lies the problem: software development is rarely straightforward, simple and predictable. Waterfall poses the following risks:

- 1. Startup risk:** Many projects fail to get off the ground because of the “analysis paralysis” that is symptomatic of the Big Design Up Front (BDUF) approach. Projects get stuck at a gate while various stakeholders hesitate to commit to a sign off.
- 2. Integration risk:** The sequential approach of Waterfall development leaves integration until near the end of development. Consequently, there is significant risk that the software will have greater integration issues since it will be done all at once.
- 3. Cost risk:** The cost of change in Waterfall projects is relatively high (see Figure 2) because it tries to minimize change by using BDUF (Big Design Up Front).

The Cost of Change

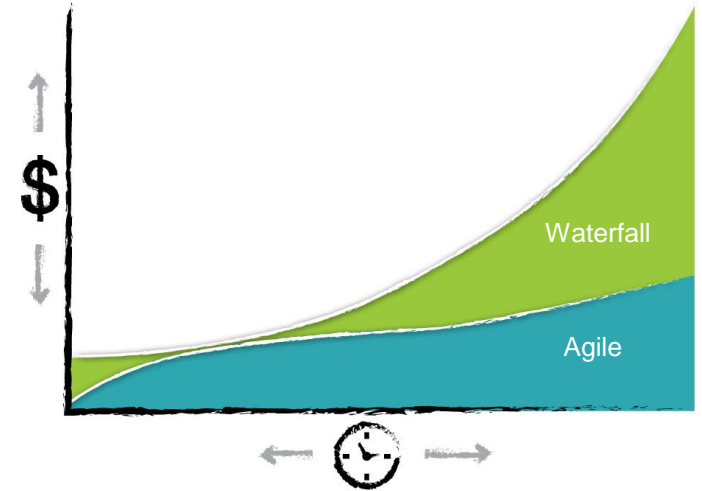


Figure 2: The Cost of Change: Traditional (Waterfall) vs. Agile⁴

The Problem With Waterfall (continued)

4. Delivery risk: Waterfall has a relatively high risk of failing to meet deadlines or accommodate customer needs. Business needs inevitably change throughout the development process (which can range from a few months to years) and the Waterfall approach, by nature, does not take this into account. Also, in a document driven development environment, a development team may place more focus on delivering documentation than on customer features.

5. Technical risk: Waterfall lacks rigorous practices of more modern methodologies. These practices include unit testing, collaborative code-base and frequent builds, all of which help ensure technical excellence.

To understand how these risks are avoided in modern approaches, it is helpful to review a few key historical milestones in software development methodologies.



From Waterfall to Agile

The Waterfall method has been prevalent in software development since the mid '50s. It wasn't until the late '60s and early '70s that references to iterative software development were made.⁵ In 1970, Winston W. Royce was credited as one of the first people to describe the Waterfall approach and also be one of the first to highlight its flaws, arguing for an iterative, Agile-like approach to software development.⁶

In the mid '80s, after decades of experience with software developed using the Waterfall methodology, change was needed. Software projects were earning the reputation of often being over budget, behind schedule, and failing to meet the needs of end users. Overly complex software was the norm (Figure 3 pokes fun at how this tends to happen when using BDUF). Then, in 1986, Frederick P. Brooks stated the following in his landmark paper, *"No Silver Bullet-Essence and Accident in Software Engineering"*:

...it is really impossible for clients, even those working with software engineers, to specify completely, precisely, and correctly the exact working requirements of a modern software product before having built and tried some versions of the product they are specifying.

Therefore one of the most promising of the current technological efforts...is the development of approaches and tools for rapid prototyping of systems as part of the iterative specification of requirements.⁷ – Frederick P. Brooks

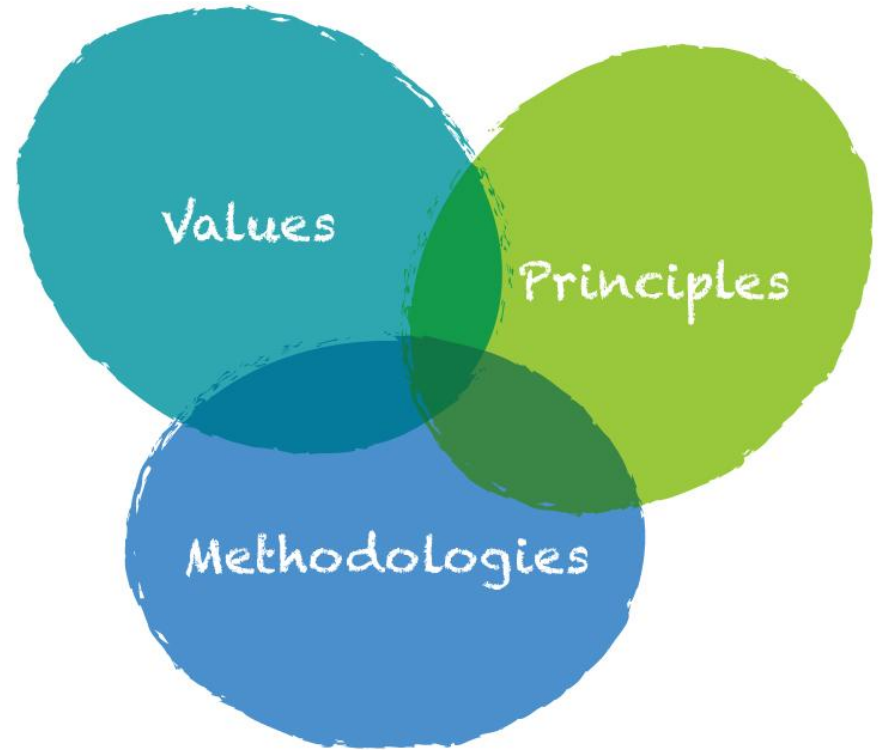
Over the next fifteen years, several "lightweight methodologies" were developed. In 2001, a group of software leaders, in a field known at the time as lightweight methods, met to bring these new methodologies together to find a better way to build software. The result was the Agile Manifesto.



Figure 3: The problem with guessing customer requirements

The Basics of Agile

Agile software development is comprised of Values, Principles and Methodologies. The 4 Agile Values serve as the foundation of Agile philosophy. The 12 Agile Principles embody the Values and provide more concrete examples of what Agile means at a lower level. They form the Agile philosophy that guides Agile Methodologies.



Agile Values

Agile is a somewhat loose term. If it had a motto, it would be embrace change.⁸ The Agile Manifesto is accepted as the official definition of Agile and expresses the four foundational Values of the philosophy:

Manifesto for Agile Software Development⁹

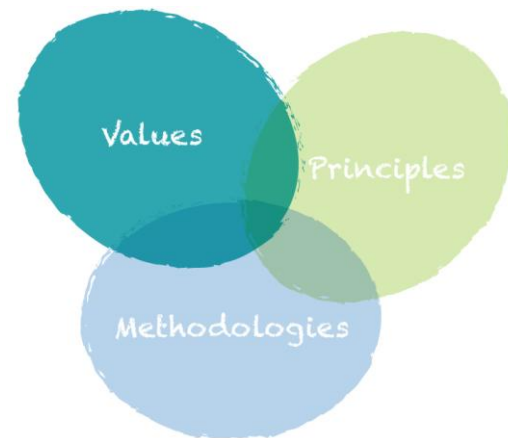
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas.

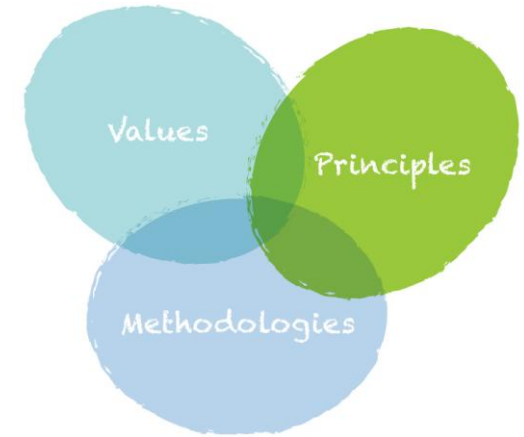
© 2001, the above authors this declaration may be freely copied in any form, but only in its entirety through this notice.



Agile Principles

The 4 main Values of the Agile Manifesto guide the twelve Principles of Agile development, which can be summarized as follows:

1. Satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently.
4. Bring business people and developers together to work throughout the project.
5. Build projects around motivated individuals.
6. Communicate face-to-face when possible.
7. Measure progress primarily through working software.
8. Develop at a sustainable pace.
9. Focus on technical excellence and good design.
10. Simplicity, maximizing the amount of work not done, is essential.
11. Enable teams to self-organize.
12. Reflect at regular intervals on how the team can become more effective.



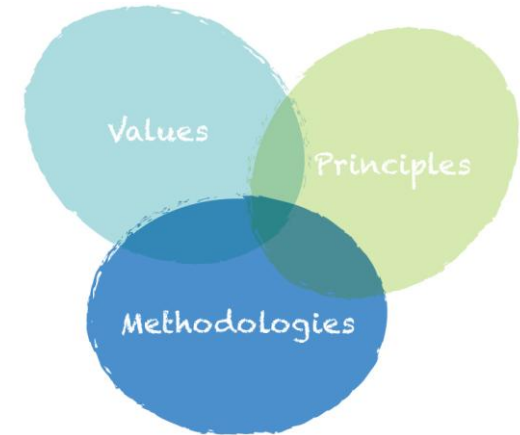
Agile Methodologies

Agile software development is perhaps best understood as a philosophy that guides several groups of practices known as Methodologies. These Methodologies, which are like recipes to follow to help you be Agile, include Scrum, Kanban, XP / Extreme Programming, Lean Programming and Agile Modeling.¹⁰ Collectively, they comprise only a selection of the “lightweight methodologies” that can be used to be Agile. While each Methodology has its nuances, the important thing to understand is there is more than one way to practice Agile development.

While ground-breaking at the time, the Agile Manifesto has led to misconceptions about Agile as a method-less, process-less approach. These claims are false. Many are based upon straw man fallacies – oversimplifications of arguments to make the Agile Manifesto easier to attack. People claim that “Agile has no documentation”. It’s a fallacious argument. Agile has documentation but places a greater value on working software. As Agile co-creator Jim Highsmith said:

“The Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment.”¹¹ - Jim Highsmith, for the Agile Alliance

In addition to the Values, Principles and Methodologies, there are additional characteristics that are common to organizations that have adopted Agile philosophies.



Agile Process

The Agile process takes a large software project and divides it into many smaller pieces to be developed incrementally and iteratively. Studies have found that project size and success are negatively correlated, i.e. the shorter the project the higher the rate of success.¹²

The Agile approach helps reduce project size by effectively making several mini-projects. It is the iterative approach that distinguishes Agile from other methods.

Unlike traditional approaches, Agile has multiple planning and development phases, known as iterations. Each iteration tends to be a week or so of work. The development team and the customer team work together to prioritize what will be included in each iteration. The end result of the relatively short sprint is working software delivered to a production-like environment where it can be tested by the customer. This iterative process is repeated until the software project is completed. Along the way, depending on the customer's needs, there can be a series of releases – software released to the end-user.



Agile Practices

There are dozens of Agile practices. Not all are used by Agile practitioners, but those looking to be Agile should be aware of these key practices. Here are some examples to help illustrate how the Agile values have been applied into more concrete practices.

Daily Standup (Stand-Up Meetings)

Also referred to as a Daily Scrum, Agile teams have daily meetings to briefly share need-to-know information. These meetings are held to keep the development team on the same page, not to provide a status update to the project manager. Brevity is the key to these meetings. In Daily Scrum meetings, participants answer three questions¹³:

1. What did I do yesterday?
2. What will I do today?
3. What problems are preventing me from making progress?



Agile Practices

User Stories

A User Story is a short description of a function that an end-user would want. There are three elements to a User Story:

- a written description for planning (card),
- a conversation about the story to better understand it (conversation), and,
- a series of tests to confirm the story (confirmation).¹⁴

Figure 4 provides an example of a User Story card. These are written from the perspective of the end-user in language that is understandable to them. Stories serve as a form of currency between customers and developers that both parties understand.

Show Account Balance

As a Private Investor, show me my current account balance so I can make decisions on future transactions that will affect my account.

Figure 4: An example of a User Story for a Financial Services System

Agile Practices

Automated Testing

With Agile, a key aspect is the implementation of automated tests. Formal and thorough automated tests help ensure the delivery of working software throughout development and eliminate defects at the source. Developers write test code using any number of available frameworks (e.g., JUnit is the go-to Java framework) at the same time as they develop the code. In this manner they build a safety net for the working code that allows them to make changes and have confidence that other features haven't been broken. It also greatly reduces the length of time between the origin and the discovery of a bug.

Acceptance tests are functional tests for a User Story. In other words, they are programmed checks to ensure that existing features continue to work as defined as new features are added. Ideally, acceptance tests are defined by the customer and coded by developers. Running acceptance tests help prove to the customer that the Story is done. After acceptance tests are created, they are run automatically and repeatedly so that the newly developed code works.



Agile Practices

Automated Builds

A key principle for Agile methodologies is to have running software at all times. In practice, the only way to do this is by ensuring that all software development is regularly and automatically compiled, built, deployed and tested. This is usually done many times a day and at least once every time a developer “checks in” code as a main part of the development branch.

The advantages for automating builds are huge. They reduce troublesome deployment and integration problems and they provide a reliable environment for demonstrations and testing.



Agile Practices

Agile Planning: Release, Iteration and Task

In Agile development, planning is divided into three levels: release, iteration and task. At the start of the project, the developers and the customer come together to discuss the major User Stories (features) that are needed in the software. They focus on identifying must-have features and roughly prioritizing and estimating them, which helps facilitate Release, Iteration, and Task planning.

Release planning: A customer-driven planning session. The developers and the customer decide on a date for the first in a series of product releases. They decide what stories to incorporate in the release. The Developers drive the effort estimates for the stories while the customer drives the selection of the stories. Effort estimates take various forms depending on the preferences of the development and customer teams.

Iteration planning: A joint effort between the customer and developers to do part of the release plan. As in release planning, the customer defines and prioritizes the User Stories and the developers estimate the effort it will take to develop them. Naturally, the timeline is shorter for an iteration than that of a release. It is often a matter of weeks, not months, for an iteration.

Task planning: After planning the iteration, the development team breaks down the stories for the iteration into a series of tasks. A list of the tasks is documented in the project room where it is highly visible; Post-it® notes and whiteboards are common tools to help with task planning. Developers sign up for tasks and assign estimates to them.



Agile Practices

Pair Programming

This practice has two developers working together on a programming task. Typically, one person takes the role of entering the code (the driver) while the other thinks about the next steps and how this code will fit into the bigger picture (the navigator).

A common objection to pair programming is that it is wasteful to have two people doing the work of one. While it may take more developer time to program this way, the output often justifies it. As one study found, pairing takes 15% more effort than one person working alone but produces results more quickly and with 15% fewer defects.¹⁶ Results will vary on a case-to-case basis, but when considering pair programming, ask whether a reduction in defects is worth added effort and resources. Also, pairing is not a full-time requirement. Teams often set their own working rules around when it is advantageous to pair.



Agile Practices

Continuous Integration

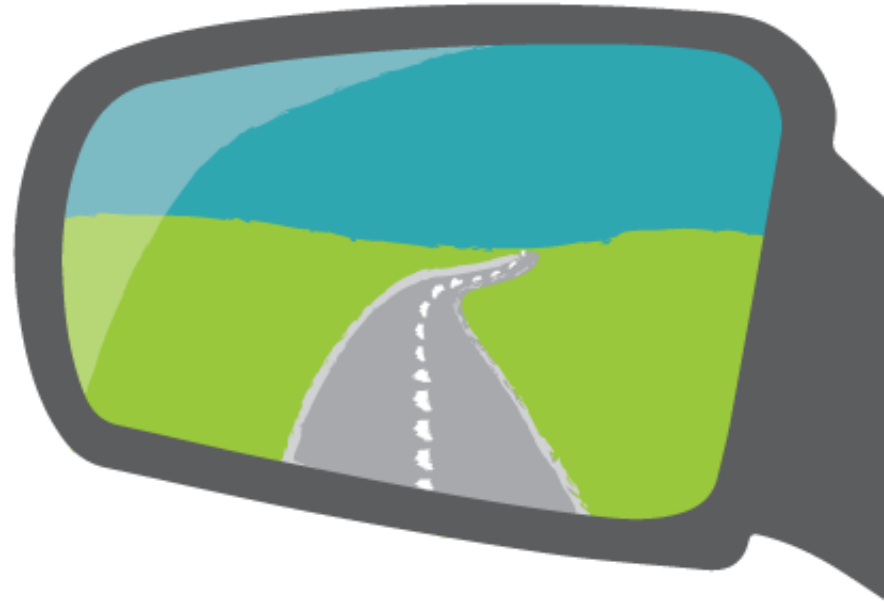
This practice has the development team integrate their code to the system several times per day. Before integrating completed code, the developer runs a series of tests to make sure the code to be integrated won't break any existing functionality or tests in the system. To do so, the developer runs all of the tests for the system and makes any necessary fixes. The more frequently code is integrated, the less time it takes to merge and find errors.



Agile Practices

Retrospectives

A retrospective is a meeting to look back over an iteration, release, or project, specifically to discuss what worked well, what could be improved, and most importantly, how to translate the lessons learned into actionable change. They are a forum for the team to improve upon their process.



The Business Benefits of Agile

Industry View

The appeal of an Agile approach to develop software incrementally and iteratively continues to grow. Agile helps lower risks associated with delivery, scope and budget that are common to every software development project.

One *MIT Sloan Management Review* study on software development practices found that early delivery of a partially functioning system and frequent deliveries throughout development are both positively correlated with higher quality software.¹⁷ Agile enables the collaboration between the development team and the customer, which offers the mutual benefit of mitigating the inherently high risk of software development.



The Business Benefits of Agile

Industry View

Version One's 7th Annual State of Agile Development Survey found that 90% of those who implemented Agile reported an improved ability to manage changing priorities¹⁸. In addition, most (70%) believed that Agile projects have a faster time to completion.¹⁹

Finally, in 2009, Dr. David F. Rico's research and synthesis of available data on Agile versus traditional methods²⁰ identified many compelling business benefits. In an analysis of 23 Agile versus 7,500 traditional projects²¹, Agile projects stacked up impressively well:

- 41% were better in terms of overall business value
- 83% achieved a faster time-to-market
- 50% were deemed to have higher quality
- 50% cost less
- 83% were seen as more productive

41%

were better in terms
of overall business value

50%

were deemed to
have higher quality

83%

achieved a faster
time-to-market

83%

were seen as
more productive

50%

cost less

The Business Benefits of Agile

Our View

Intelliware has been involved with Agile since the very early days of Kent Beck's articulation of his Extreme Programming principles in the late 1990's, at a time pre-dating the Agile Manifesto. For us, Beck and the rest of the Agile pioneers articulated and validated many of the frustrations and challenges we had experienced using Waterfall and BDUF. XP, and by extension, Agile, offered an alternative that made sense and put simply, worked.

Over the years, different projects have enjoyed a range of benefits as a result of the application of Agile philosophies. However, there are four key business benefits that our clients view as particularly important.



The Business Benefits of Agile

Our View

1. Get unstuck.

In all organizations, it is often very difficult to just get started. With Waterfall, there are unreasonable demands to “nail the requirements”, which fuels analysis paralysis. With Agile, a project can get started with high-level requirements and a simple gating of the first level of useful functionality. You need to have a high level map of where you are headed, but the nitty-gritty detail work of fine-grained requirements, design, coding and testing can be done more efficiently and with a higher degree of quality when they are done in parallel.

Agile helps you get going. If you don't start, you won't get to the business benefits of your project until it's too late.



The Business Benefits of Agile

Our View

2. Change is built into the process.

The biggest fallacy of Waterfall is that you can “nail the requirements”. All projects have change, and the longer they run, the more changes your project will require. If all your software project plans are tied up in an elaborate Microsoft Project®-style plan with dependencies, you and your organization are in for a rough ride.

Agile software development builds change into the process from the get-go.



The Business Benefits of Agile

Our View

3. Risk is managed, your project is delivered.

Because change is built right into the process, and Agile depends on robust technical practices such as automated testing and automated builds, you have a running production system as early as day one. As the system grows, and the project endures changes in requirements and design, the core application is kept running. The risks associated with change are dealt with daily, and so, at the end of the project, there isn't a big bang release with a mountain of hidden problems.

When steered properly, and with discipline, Agile projects get delivered.



The Business Benefits of Agile

Our View

4. Deliverable quality is higher.

Key drivers for better application quality include a number of the rigorous Agile technical practices such as automated tests and builds. But another, less talked about contributing factor to overall project quality is the effect Agile processes have on the people involved with the project. Because of the cross-functional nature of Agile teams, questions get answered quickly, roadblocks can be removed, and the team is motivated to work together towards a common end goal. It's not a year-long project; it's a series of one or two week sprints with real deliverables in a short period of time. It's a satisfying and energizing manner in which to work.

Stronger technical practices and greater overall team motivation all combine to result in higher quality project deliverables.





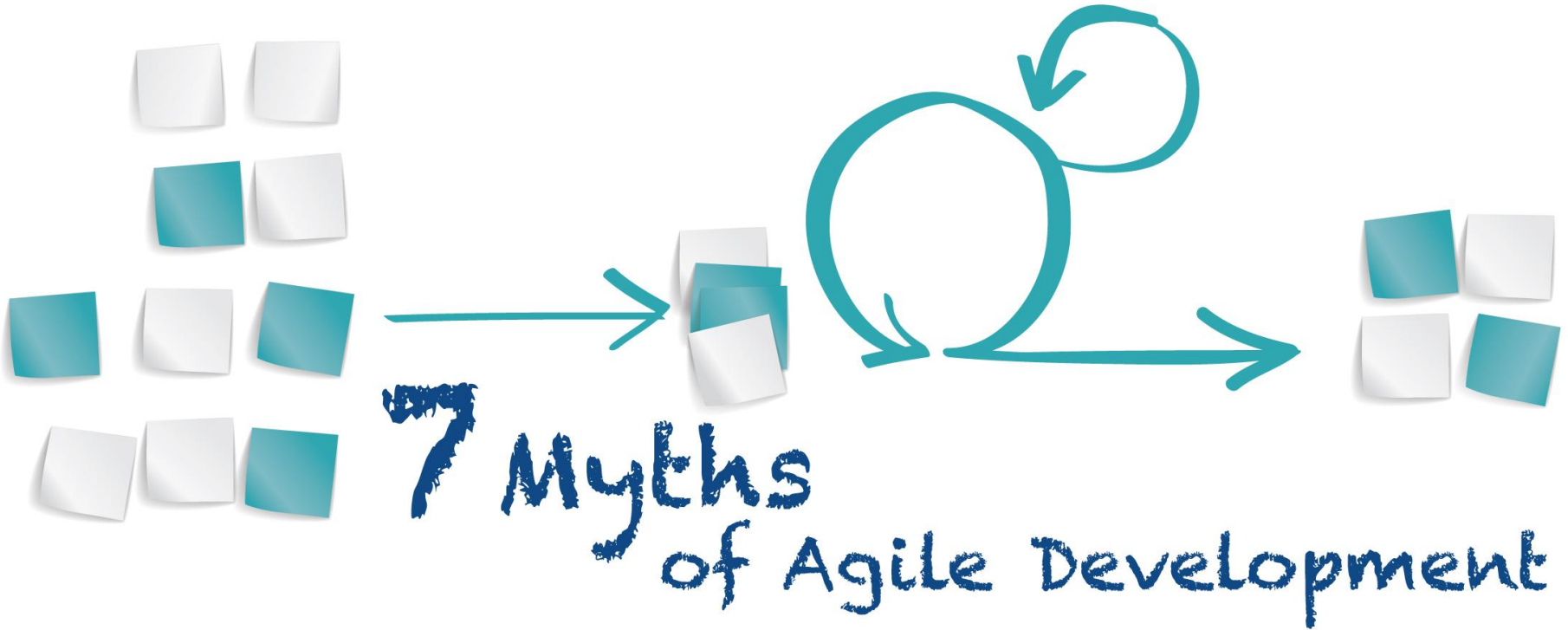
Is Agile right for you?

A transition to Agile is not a trivial effort, nor is it necessarily for everyone. A common pitfall is to try a few Agile practices that seem convenient and ignore those that seem difficult. This often results in a failed transition based on faulty assumptions and missing pieces. A truly sustainable Agile transformation requires a significant change in culture within your organization. Take time to really appreciate what Agile entails, understand the main barriers to adoption and consider its compatibility with your organization before starting implementation.



Sources

- ¹ Scott M. Ambler and Matthew Holitza. *Agile for Dummies*. Hoboken: John Wiley & Sons, 2012. Print.
- ² Krill, Paul. "[Agile software development is now mainstream](#)". *InfoWorld*. The IDG Network. 22 Jan. 2010. Web.
- ³ Heusser, Matthew. "[Has Agile Software Development Gone Mainstream?](#)" *CIO.com*. CXO Media Inc. 12 Aug. 2013.
- ⁴ Ambler, Scott. "[Examining the Agile Cost of Change Curve](#)". *Scott W. Ambler + Associates*. Ambyssoft Inc. 2002. Web.
- ⁵ Larman, Craig and Basili, Victor R. "[Iterative and Incremental Development: A Brief History](#)". *C2.com*. 21 Apr. 2011. Web.
- ⁶ Winston, Royce, W. "[Managing the Development of Large Software Systems](#)", *Proceedings of IEEE WESCON 26* (1970). Web.
- ⁷ Brooks, Frederick P. "[Silver Bullet: Essence and Accidents of Software Engineering](#)". Pgs. 13-14 *Computer In Computer*, Vol. 20, No. 4. (1987). Web.
- ⁸ Craig Larman. "Agile & Iterative Development: A Manager's Guide". Boston: Pearson Education. 2004. Print.
- ⁹ Beck, Kent et al. "[Manifesto for Agile Software Development](#)". *Agilemanifesto.org*. 2001. Web. 15 Sept, 2013.
- ¹⁰ Scott M. Ambler and Matthew Holitza. *Agile for Dummies*. Hoboken: John Wiley & Sons, 2012. Print.
- ¹¹ Highsmith, Jim. "[History: The Agile Manifesto](#)". *Agilemanifesto.org*. 2001. Web.
- ¹² Craig Larman. "Agile & Iterative Development: A Manager's Guide". Boston: Pearson Education. 2004. Print.
- ¹³ Shore, James, Warden, Shane. *The Art of Agile Development*. Sebastopol: O'Reilly Media, Inc., 2008. Print.
- ¹⁴ Mike Cohn. "User Stories Applied: For Agile Software Development". Boston: Pearson. 2004. Print. Reference also to Ron Jeffries (2001)
- ¹⁵ Wake, Bill. "[INVEST in Good Stories, and SMART Tasks](#)". *XP123*. Web.
- ¹⁶ Shore, James, Warden, Shane. *The Art of Agile Development*. Sebastopol: O'Reilly Media, Inc., 2008. Print.
- ¹⁷ MacCormack, Alan. "[Product-Development Practices That Work: How Internet Companies Build Software](#)". *MIT Sloan Management Review*. 2001. Web.
- ¹⁸ VersionOne®, 7th Annual State of Agile Development Survey © 2013.
- ¹⁹ VersionOne®, 7th Annual State of Agile Development Survey © 2013.
- ²⁰ Dr. David F. Rico, PMP, CSM, "[Business Value of Agile Methods – Cost and Benefit Analysis](#)" [davidfrico.com/rico09e.pdf](#), 2009.
- ²¹ Mah, M. (2008). Measuring agile in the enterprise: *Proceedings of the Agile 2008 Conference, Toronto, Canada*.





Introduction

There are dozens of myths about Agile development. But before jumping into specific misconceptions, let's have a look at some common business challenges:


For senior-level execs: do you value revenue growth or cost containment?

For project managers: do you value team efficiency or effectiveness?

For developers: do you value code quantity or quality?

In each scenario, you probably struggled to make a choice given that your two options were not mutually exclusive.

Posing the question this way creates a false dilemma since you likely value both options but to varying degrees. So the better question is, of the two options, which do you value *more*?



Introduction (continued)

The Agile Manifesto evolved through dilemmas like those just mentioned. Often two opposing approaches, such as responding to change versus following a plan, were deliberated upon until the authors of the Manifesto decided that it would be best if they valued one approach more than the other, instead of choosing one over the other.

Unfortunately, many of the myths about Agile are based upon straw man fallacies: oversimplifications of arguments to make them easier to attack. For example, the Agile Manifesto states that Agile values working software over comprehensive documentation. Agile detractors often oversimplify this idea as an either/or state: working software or comprehensive documentation.

What follows are the most common Agile Myths and a rebuttal to these false claims.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

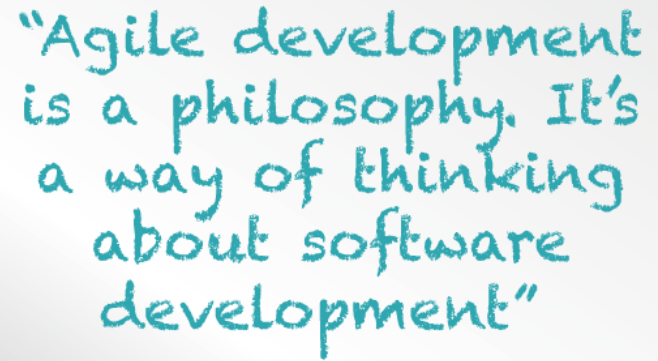
Figure 1: The Agile Manifesto
Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas.

Myth #1

Agile development is a methodology

While not as common as the other 6 myths, this one needs to be addressed first. Agile development is not a methodology – it is a set of shared values and principles that guide a set of technically rigorous development methodologies. These methodologies include, but are not limited to: Scrum, Kanban, XP (Extreme Programming) and Lean Software Development.

Authors of “The Art of Agile Development”, James Shore and Shane Warden, argue, “Agile development is a philosophy. It’s a way of thinking about software development.” We view Agile as a philosophy that addresses the forces at work in organizations that are striving to deliver high quality software applications to the customer and their often evolving needs.



“Agile development is a philosophy. It’s a way of thinking about software development”

Myth #2

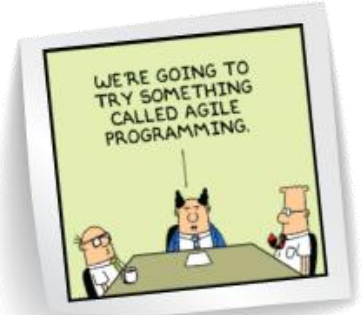
Agile is undisciplined

This myth is based on a perceived lack of process. Agile places greater value on individuals and interactions than on processes and tools. Therefore, some conclude illogically that Agile lacks process and therefore discipline.

This myth comes in many flavours: “Developers get to do what they like.”² Agile = anarchy.³ Agile is “cowboy programming”.⁴ Agile means “code and fix”.⁵

Incomplete Agile transitions help keep this myth alive, as some companies have adopted the easier parts of Agile while ignoring the harder parts.⁶ This is sometimes referred to as “Agilefall”.

There is process to Agile, though it isn't the one-track, sequential process of traditional development. Agile processes often occur in parallel with one another and are repeated.⁷ In fact, many argue that Agile actually requires a greater level of discipline than more traditional approaches.^{8,9} In our experience the ‘agility’ in application delivery is enabled through a very disciplined approach to technical application development practices.



Myth #3 Agile has no planning

"Plans are worthless, but planning is everything."¹⁰ - Dwight D. Eisenhower

Eisenhower made the above comment to a National Defense Committee in the context of planning for emergencies. But there is a reason why it has been quoted by Agile proponents: both national defense emergencies and end-user software needs are unforeseen.

To be agile is to be able to move quickly and easily. Planning can inhibit agility. Some illogically conclude that all planning inhibits agility. As English author Lewis Carroll originally said, "if you don't know where you are going [no plan], any road will get you there." The right amount of planning is essential to properly guide your 'agility'.

In Agile development, Big Design Up Front (BDUF) is avoided; planning occurs throughout the development cycle and is spread across the entire team.¹¹ It avoids the situation where overly detailed plans made at the start of a project become out of sync with the technical and business needs as the project progresses.¹² Agile aims to work the plan, not work to the plan. The result is a constant focus on business value.

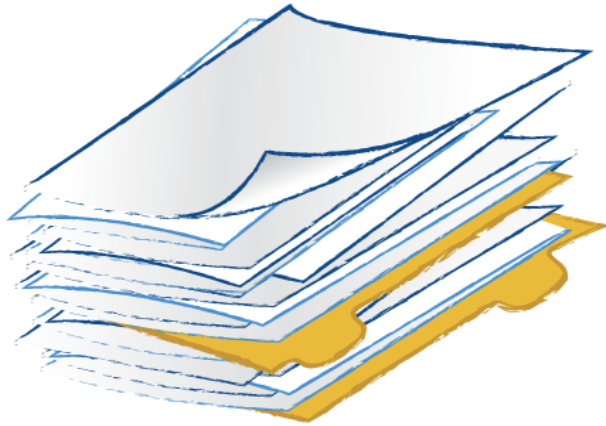
Throughout development, the team adapts to make sure the plan reflects the current needs of the customer. Therefore, Agile welcomes the changes that are inevitable in software development and plans accordingly. Release and iteration plans detail both what needs to be done and how it will be done. Delivering working software throughout the development process at agreed upon deadlines requires planning.



Myth #4

Agile has no documentation

The Agile founders met to create an alternative to document driven development.¹³ They did not set out to remove documentation from software development. Agile simply places more value on working software than on *comprehensive* documentation because of the dynamic nature of software development. As requirements are modified, the development changes course and the software evolves. This does not preclude any development team from generating as much documentation as the project requires. Indeed, the natural process of Agile development tends to generate a greater amount of (and more accurate) documentation than BDUF methodologies.



Comprehensively documenting a system (particularly when it's done “up front”) can be a poor use of time since changing requirements renders documents obsolete or inaccurate. Also, there is a risk of misunderstanding between the customer and developer when relying on written documentation to express software requirements. However, there are situations (documenting interfaces between systems, for instance) in which documentation is absolutely required. There is nothing inherent in Agile that prevents you from creating as much documentation as your project requires, especially if the customer values it. Agile just suggests you be smart about it and that documentation not take on a life of its own. Documentation that provides value to the customer is much different than documentation that is produced for the sake of documentation or to support a process.

To quote Mike Cohn, in a Waterfall approach, “Customers will get the developers’ interpretation of what was *written down*, which may not be what they *wanted*.”¹⁴ The iterative delivery of working software effectively replaces much, though not all, of the comprehensive upfront requirements documentation. A picture is worth a thousand words. Working software is worth even more.

Myth #5

Agile has no upfront design/architecture

Agile stresses a simplification of upfront design, not the elimination of upfront design. As argued by Robert C. Martin, one of the founders of the Agile Manifesto, Big Design Up Front (BDUF) is “harmful” but little upfront design (LUFD) is “absolutely essential”.¹⁵

The harm from BDUF can take shape in an overly complex product, which is a typical outcome for projects developed using the Waterfall approach. Agile development stresses simple upfront design to focus on the foundation and general structure of the software. Agile developers avoid building software features that may or may not be needed; they build for the current need and get feedback in the iterative delivery of software to the client.

In XP, an Agile method, there is a principle called YAGNI (You Aren’t Gonna Need It) to help focus on designing at the right time in the process.¹⁶ As with drawing, Agile recommends beginning with a sketch to explore the presentation of a concept. If the concept is validated, then the details are added.



Myth #6

Agile does not scale

Scaling software development is difficult, regardless of approach. Some believe that Agile may work fine for small projects but not for large, complex projects.

Agile encourages breaking large, complex projects into many small, manageable pieces. This means that it can indeed scale – even for big projects. Of course, it really is a matter of approach. Some Agile practices need to be tweaked for the realities of large projects. For example, the larger the team, the shorter the development cycle should be. By keeping the development cycles short, the project remains a series of small, manageable projects. As well, face-to-face conversations will likely be limited in large projects, so technologies that enable the closest representation of this form of communication, such as videoconferencing, should be used. Finally, since continuous integration can be a significant challenge on large Agile projects, there should be an integration team.¹⁷



Myth #7

Agile is just another fad

A fad is, by definition, short-lived. Agile has been around for over a decade. Some of the central tenets have been in practice since the '70s. Agile has been around for too long to be a fad, especially when one compares it to the relatively short history of software development. This myth may be propagated by those who comprise the laggards of Agile adoption – many of whom tend to resist change.

Agile, as a philosophy (not a methodology), was created in response to the inherent complexities of software development; therefore, as long as there is software development, Agile will exist.

An alternate explanation may be that Agile is still in a “hype cycle” and therefore, still subject to possibility of being a fad. In Gartner’s 2010 report, “Hype Cycle for Application Development”, Agile Development Methods were considered as, “sliding into the trough (of disillusionment)”.¹⁸ That may sound ominous, but for those who believe in this model, it is merely a growing pain. Java went through this trough before reaching its plateau.¹⁹ Agile continues to develop. In 2012, Gartner also stated that Agile’s move through the Trough of Disillusionment is a “normal part of any IT trend that is going mainstream” and that “the long term trend of agile is working well in more and more companies, so the future of agile is still promising.”²⁰

Gartner’s Hype Cycle

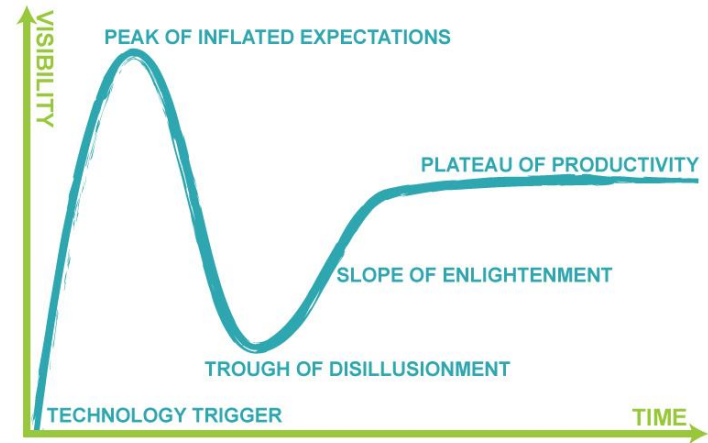


Figure 4: Gartner's Hype Cycle



Conclusion

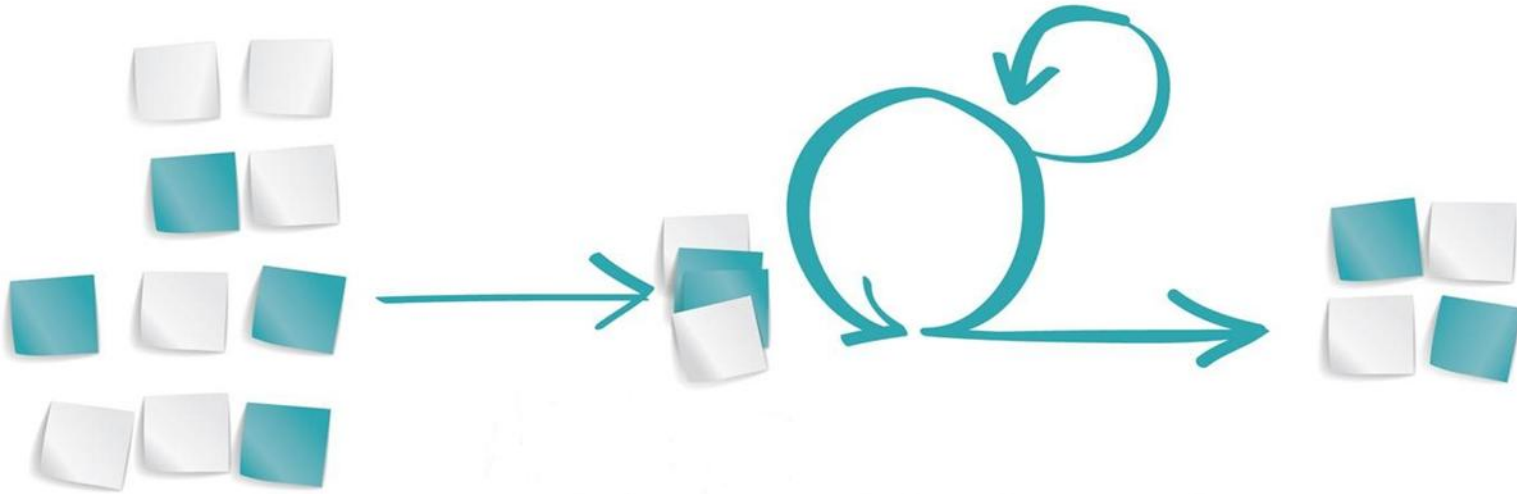
Myths and criticisms about Agile software development abound. Don't let them impede your progress. Agile is not a fad – the forces that have led us to Agile are not going away. Regardless of your proficiency with Agile, it is important to be mindful of these common myths and criticisms because Agile requires organizational buy-in. Therefore, you will need your colleagues to understand Agile, which will likely require you to debunk a few of these myths yourself. If you can succeed at this task, you will increase your chances of fully realizing the business benefits of Agile, such as faster time-to-market, higher quality software, lower costs and a greater ability to adapt to changing priorities.

Sources

- ¹ Shore, James, Warden, Shane. *The Art of Agile Development*. Sebastopol: O'Reilly Media, Inc., 2008. Print.
- ² Kelly, Allan. "[Top Twelve Myths of Agile Development](#)." *The Agile Connection*, TechWell Corp. 27 Mar, 2013. Web.
- ³ Löffler, Marc. "[7 Agile Myths](#)". *The Agile Zone*. 29 Jan, 2013. Web.
- ⁴ O'hEocha, Colm. "[Agile – Adoption: Agile Myths](#)". *AgileInnovation Ltd.* www.agileinnovation.eu. 2010. Web.
- ⁵ Holler, Robert. "[Five Myths of Agile Development](#)". *VersionOne*. 2010. Web.
- ⁶ Rasmusson, Jonathan. "[Agile Myths](#)". *Agile in a Nutshell*. Web.
- ⁷ Gregory S. Smith, "[What an Agile Process Looks Like](#)". CIO.com. 23 Jan., 2008. Web.
- ⁸ Scott M. Ambler and Matthew Holitza. *Agile for Dummies*. Hoboken: John Wiley & Sons, 2012. Print.
- ⁹ Holler, Robert. "[Five Myths of Agile Development](#)". *VersionOne*. 2010. Web.
- ¹⁰ From a speech to the National Defense Executive Reserve Conference in Washington, D.C. (November 14, 1957) ; in *Public Papers of the Presidents of the United States, Dwight D. Eisenhower, 1957*, National Archives and Records Service, Government Printing Office, p. 818 : [ISBN 0160588510](#), 9780160588518
- ¹¹ Kelly, Allan. "[Top Twelve Myths of Agile Development](#)." *The Agile Connection*, TechWell Corp. 27 Mar, 2013. Web
- ¹² Holler, Robert. "[Five Myths of Agile Development](#)". *VersionOne*. 2010. Web.
- ¹³ Highsmith, Jim. "[History: The Agile Manifesto](#)". *Agilemanifesto.org*. 2001. Web.
- ¹⁴ Mike Cohn. *User Stories Applied For Agile Software Development*. Boston: Peasron Education, 2004. Print.
- ¹⁵ Martin, Robert C. ("Uncle Bob"), "[The Scatology of Agile Architecture](#)". *Uncle Bob Consulting LLC*. April 25, 2009. Web.
- ¹⁶ Rasmusson, Jonathan. "[Agile Myths](#)". *Agile in a Nutshell*. Web.
- ¹⁷ Jutta Eckstein and Nicolai Josuttis. [Scaling Agile Processes: Agile Software Development in the Large](#). Agility Days 2002. Web.
- ¹⁸ Janes, Andrea, and Succi, Giancarlo. "[The Dark Side of Agile Software Development](#)". *Darkagilemanifesto.org*. Free University of Bolzano/Bozen. 2012. Web.
- ¹⁹ Janes, Andrea, and Succi, Giancarlo. "[The Dark Side of Agile Software Development](#)". *Darkagilemanifesto.org*. Free University of Bolzano/Bozen. 2012. Web.
- ²⁰ Wilson, Nathan. "[The Trough of Disillusionment](#)". *Gartner*. 27 July, 2012. Web.



intelliware
software development



Agile Story Writing

A decorative graphic on the left side of the slide featuring three overlapping sticky notes: a white one at the top, a light green one in the middle, and a darker green one at the bottom. A large green arrow points from the bottom of these notes down to a single white sticky note at the bottom left.

What You'll Learn in this Presentation:

- The basics of user stories.
- How user stories fit into the overall Agile planning process.
- How to write a user story.

An example of a user story card, which is a white rectangular card with rounded corners and a horizontal line near the top. The title 'Show Account Balance' is written in blue cursive at the top. Below the line, the user story is written in black cursive. The card is slightly offset to the right and has a soft shadow.

Show Account Balance

As a Private Investor, show me my current account balance so I can make decisions on future transactions that will affect my account.

A story card example

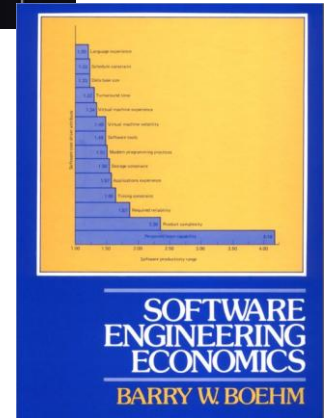
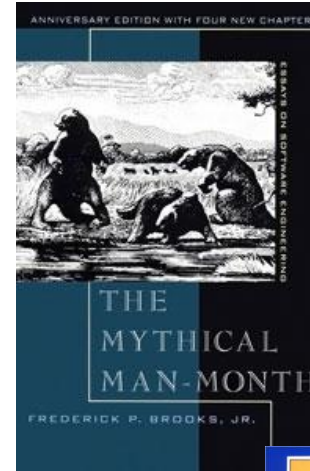
Why is it so Difficult to Determine Software Requirements?

- **Requirements gathering is when informal ideas become formal concepts:**
 - Converting a concept into something concrete is almost always more difficult than it is initially believed to be.
 - The concept of what the concrete version needs to look like changes frequently.



Why are Requirements So Important?

- According to Fred Brooks (the author of *The Mythical Man Month*):
 - *“The hardest single part of building a software system is deciding precisely what to build. No other part is as difficult...No other part of the work so cripples the resulting system if done wrong.”*¹
- According to Barry Boehm (*Software Engineering Economics*) and other software engineering experts, around 75-80% of all errors found in software projects can be traced back to the design and requirements phases.²

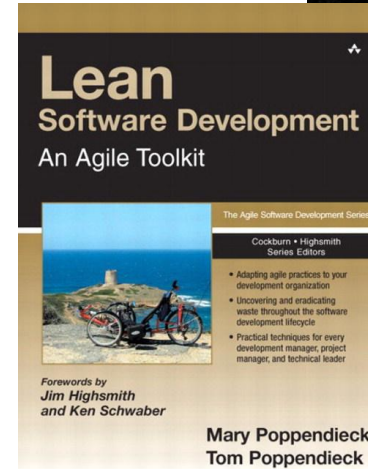
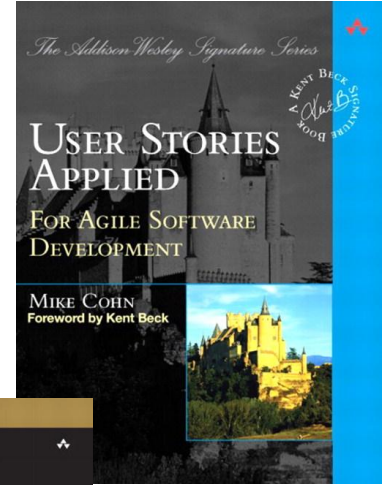


¹ Brooks, Frederick P., 1987. "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer*, Volume 20, No. 4.

² Boehm, W. Barry. "Software Engineering Economics". Prentice Hall; 1 edition (Oct. 22 1981).

The Challenges with Written Requirements

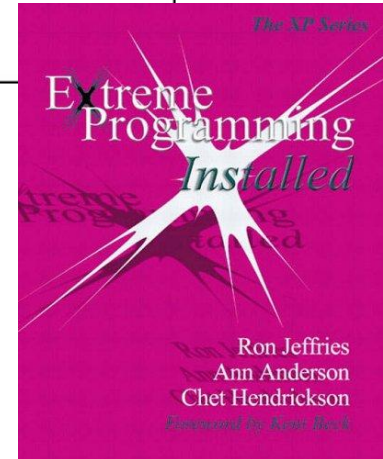
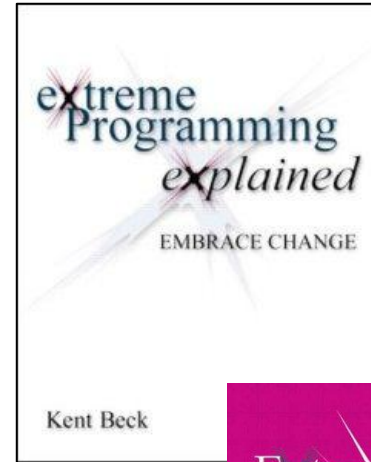
- According to Mike Cohn, author of *User Stories Applied*:
 - “Writing things down is no guarantee that customers will get what they want; at best they’ll get what they wrote down.”³
- From *Lean Software Development* by Mary & Tom Poppendieck, the Seven Wastes of Software Development:⁴
 1. Partially Done Work
 2. **Extra Processes (paperwork)**
 3. Extra Features
 4. Task Switching
 5. Waiting
 6. Motion
 7. Defects



What is a User Story?

There are numerous definitions for stories.

- A common definition:
 - ***A short description of a function that an end-user would want.***
- From Kent Beck:
 - ***“One thing the customer wants the system to do...(it) should be testable.”***⁵
- From Ron Jeffries:
 - ***“Stories are promises for...the series of conversations that will take place between the customer and the programmers.”***⁶



Why User Stories?

Stories have many advantages.

- **Easy to understand**
 - Written in non-technical language that customers / product owners can relate to.
- **Work at the right level**
 - Not too detailed, are easy to manipulate and move around, like a deck of cards.
- **Relatively easy to create**
 - Writing stories takes some skill, but experts can define entire systems for planning purposes in a matter of hours.



Story Types

Epic

- **Represents multiple features or many stories.**
- Can take months to build and works at the release level.
- What the end users tend to focus on.

Feature

- **Smaller than epics, but bigger than stories.**
- Can take weeks, possibly one or more iterations to build.
- What customers / product owners tend to focus on.

User Story

- **Are the smallest increment of value.**
- Take days, perhaps a week or two at most to build.
- What development teams tend to focus on.

← Primary focus of this presentation.

Key Players: Customers & Developers

1. Customers / product owners

- The people who know how to do what the system is going to be doing.
- They either are the end user or they are representative of the eventual user of the system.

2. Programmers

- The people who will be building, testing, deploying, documenting & training those who will use the system.

Stories are key to fulfilling the requirements in the **Customer and Programmers Bills of Rights.**

Customer Bill of Rights

- *You have the right to an overall plan*
- *You have the right to change your mind, to substitute functionality, and to change priorities*
- *You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date*

Programmer Bill of Rights

- *You have the right to know what is needed, with clear declarations of priority*
- *You have the right to make and update your own estimates*

User Story Components

At minimum, a user story has:

1. **A card** to write the story on.
2. **A name** that the customers and developers understand.
3. **A description** (should be limited to one or two sentences).
4. **Acceptance criteria** to define when the story will be considered completed.
5. **A size estimate** for time management.

Pay by Credit Card *Size = 2*

As an online customer, allow me to pay by credit card of my choice so that I can complete my purchase.

Acceptance:

Test payments with VISA, Master Card & Amex.

Reject payments exceeding \$1000

A typical story card.

User Story Components

Diagram illustrating the components of a User Story form, with labels and arrows pointing to the corresponding fields:

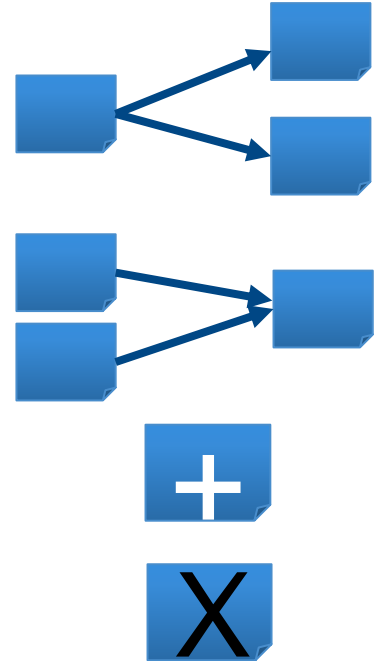
- Client Logo (optional)**: Points to the **intelliware.ca** logo and text.
- Story Name**: Points to the **Login** title.
- Description (no more than 2-3 sentences)**: Points to the **Description:** section.
- Story Notes (optional)**: Points to the **Notes:** section.
- Acceptance (list of criteria to indicate when the story will be closed)**: Points to the **Acceptance:** section.
- Story Priority (optional)**: Points to the **Priority** field (value: 2).
- Story Size**: Points to the **Size** field (value: 1).
- Release, Module and ID Fields (optional - help to identify the story)**: Points to the **ID** field (value: SMP-1-2).

| intelliware.ca software development | Priority | Size |
|--|----------|---------|
| Login | 2 | 1 |
| Description: As an authorized user, provide a login page to gain access to the secure pages of the Web site. | | |
| Notes: <ul style="list-style-type: none">• Users to provide IDs and Passwords.• A 'Forgot your password?' link will be needed. | | |
| Acceptance: To be provided by the Customer. | | |
| | ID | SMP-1-2 |

Story Actions

Stories can be:

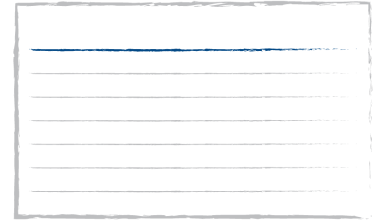
- **Split**
 - A large story can be split into two or more smaller ones of different sizes; useful for breaking up epics.
- **Combined**
 - Two or more small stories can be combined into one.
- **Added**
 - New stories can be added to an existing backlog.
- **Deleted**
 - Existing stories can be deleted from a backlog.



The 3 Cs of Stories

- **C**ard

- A token to represent some customer functionality.
- Stories represent customer requirements rather than document them.
- Using a card keeps the story short.



- **C**onversation

- Customers and developers discuss the details of the story at the time it is to be developed, not before then.



- **C**onfirmation

- The customer should provide acceptance tests for the story, and then see them run to confirm that the story has been completed.



User Roles and Description Formats

- **Identifying user roles helps with writing stories.**
- **Standard story description template:**
As a [role], provide [function] so that [business value].
- **Some simple examples:**
 - As a customer, provide a button that I can use so that I can connect directly with the call centre when my order gets stuck.
 - As a call centre rep., review orders in progress online so that I can help customers complete their orders.
 - As a manager, access stats on incomplete online orders so that I can make decisions on how to improve the ordering process.



Call Centre Rep.



Manager



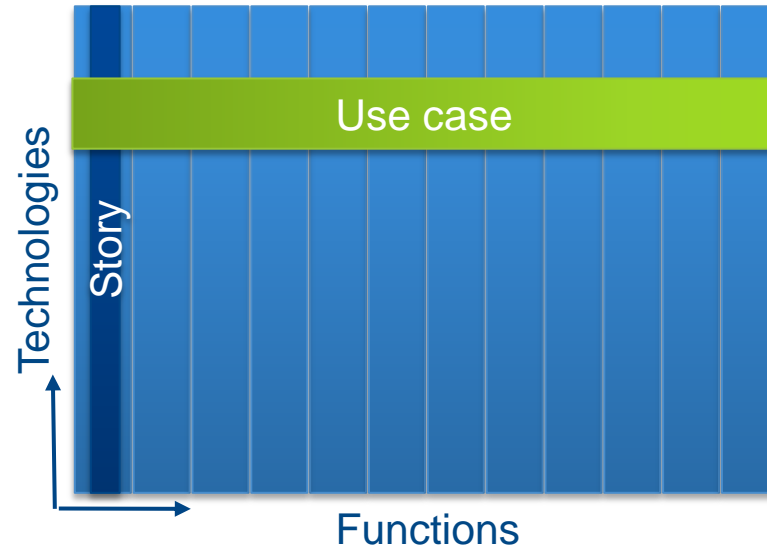
Customer

Writing Stories – The INVEST Framework⁷

- **I**ndependent – Dependencies between stories should be avoided.
- **N**egotiable – Stories should be written so that the details can be negotiated in a conversation between the customer and the development team.
- **V**aluable – The feature should have business value to the customer.
- **E**stimatable – Stories should be understood well enough by customers and should be small enough to be “estimatable”.
- **S**mall – Stories that are too big are not useful in planning.
- **T**estable – It must be possible for the development team to write tests for the story.

User Stories vs. Use Cases

- **A story** can be considered similar to a lightweight use case.
 - A story can represent a small piece of a use case.
- **Use cases** cut across many functions and may touch on many stories.



Examples of Common Story Mistakes

Easy to Use

As any user of the system, make sure the user interface is easy to use so that I can be more efficient, make more purchases and will be more likely to return.

Not testable!

Not functional.

Examples of Common Story Mistakes

Pay By Credit Card

As a customer, pay by VISA, MasterCard and American Express.

Too big...should probably be split as follows:

1. Pay by one credit card (including payment infrastructure)
2. Pay by additional credit cards

Examples of Common Story Mistakes

System Administration

As a power user or manager, provide a user interface to facilitate management of the system.

This is huge, with many functions; an epic!

Also, there's more than one user type.

Examples of Common Story Mistakes

Choose Flight

As a customer, select a flight from the list presented by the List Flights Story and confirm my selection.

This is not independent as there is a dependency on the List Flights Story.

Should be split along another dimension.

For More Information

Mike Cohn's site contains a good section on user stories:

<http://www.mountangoatsoftware.com/agile/user-stories>

The Agile Alliance site is also a good resource:

<http://guide.agilealliance.org/guide/user-stories.html>

Intelliware's Knowledge Centre contains several resources on the basics of Agile:

<http://www.intelliware.com/knowledge-centre>





A decorative graphic on the left side of the slide. It features a stack of three sticky notes at the top: a white one on top, a light green one in the middle, and a darker green one at the bottom. A large green arrow points downwards from the bottom sticky note to another white sticky note at the bottom left. In the bottom right corner, there is a single light green sticky note.

What You'll Learn in this Presentation:

- How estimates are used on Agile projects.
- How to define estimates.
- The basics of planning poker to help estimate.

Introduction – Agile Planning

- Planning occurs throughout the development cycle and is spread across the entire team.
- “***Work the plan, not work to the plan.***”
- A user story is a short description of a function that an end user would want.
- Stories serve as a form of currency between the customers and the development team.



Show Account Balance

As a Private Investor, show me my current account balance so I can make decisions on future transactions that will affect my account.

The Agile Planning Process

- Estimates are needed to support ongoing planning in Agile projects.
- Estimates help to answer the following questions:
 - How many stories can we fit into the release?
 - How many stories will be completed in the next iteration?
 - What are the impacts of adding, removing and changing stories?



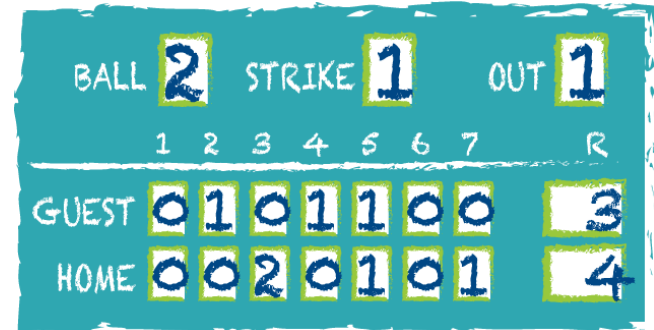
What is an Estimate?

- An estimate is a measure of the relative size, in terms of effort, of a story.
- Why focus on size?
 - Estimate size to derive duration.
 - Size can be estimated.
 - Duration is hard to determine due to meetings, non-work appointments and other distractions that cannot be estimated.



Effort vs. Duration

- **Not separating effort from duration** is a common estimating error.
- In reality, developers (and people in general) are good at estimating the effort required to complete something but not the time needed.
- For software, **development effort** can be estimated but, to determine duration, additional time needs to be factored in to account for:
 1. **non-development project time**
(e.g. standup meetings), and
 2. **non-project work time** (e.g. internal non-project meetings, medical appointments).
- Baseball analogy: the effort to complete a game is 54 outs or 9 innings, but what's the duration of these events – and those in between?



A stylized baseball scorecard with a teal background and white text. It shows the score for a game between the Guest and Home teams. The Guest team has 3 runs, 1 hit, and 1 error (R1H1E1). The Home team has 4 runs, 2 hits, and 1 error (R4H2E1). The score is 3-4 in favor of the Home team.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | R |
|--------|---|---|---|---|---|---|---|---|
| BALL | 2 | | | | | | | |
| STRIKE | 1 | | | | | | | |
| OUT | 1 | | | | | | | |
| GUEST | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 3 |
| HOME | 0 | 0 | 2 | 0 | 1 | 0 | 1 | 4 |

Ideal Time vs. Real Time

- **Ideal Time = effort**
 - Time required to complete something with no interruptions.
 - Represents effort.
 - Can be estimated relatively accurately.
- **Real Time = duration**
 - The actual time to complete something.
 - Includes breaks, distractions, delays.
 - Difficult to estimate – must be derived.



The Value of Estimates

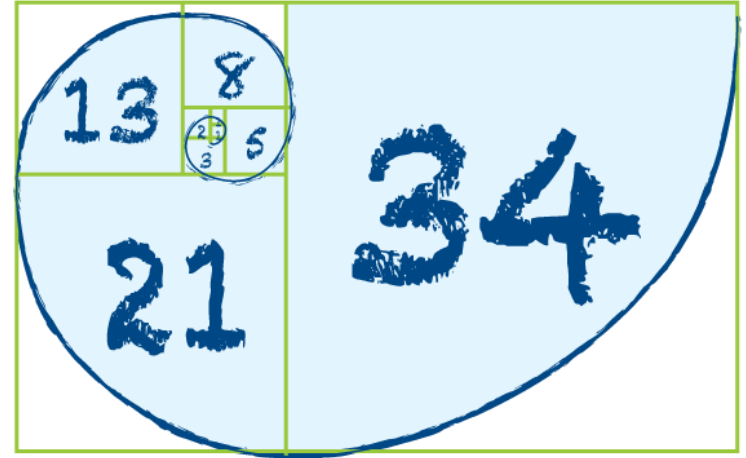
- **Address the Law of Diminishing Returns**
 - Make the most efficient use of development time.
- **Benefits from the input of several people**
 - Estimates are shared amongst the team.
- **Supports ongoing planning**
 - Facilitate planning discussions at the right level.
- **Easy to change**
 - Can be revised, split, etc.



The application of the Law of
Diminishing Returns to estimating

Estimate Values

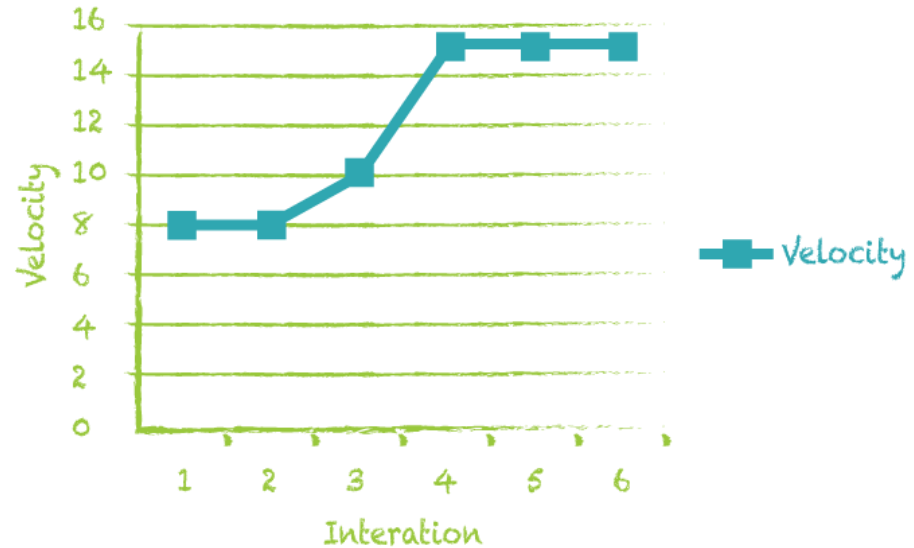
- **Estimates can be measured in terms of:**
 - Points.
 - Ideal days.
 - Any other unit of measurement that makes sense to the team.
- **Common estimating scales:**
 - Standard interval – 1, 2, 3, ...
 - Fibonacci series – 1, 2, 3, 5, 8, 13, 21, ...
 - Doubling interval – 1, 2, 4, 8, 16, ...



The Fibonacci Series

What is Velocity in Estimating?

- A measure of the rate of progress.
- Needed for iteration and release planning.
- Usually measured in terms of points or ideal days per iteration.
- Converts ideal time or estimates to duration.
- Corrects for variability and errors in estimation.



A typical velocity tracking plot

How Is Velocity Determined?

Three Primary Methods:

1. Historical Values

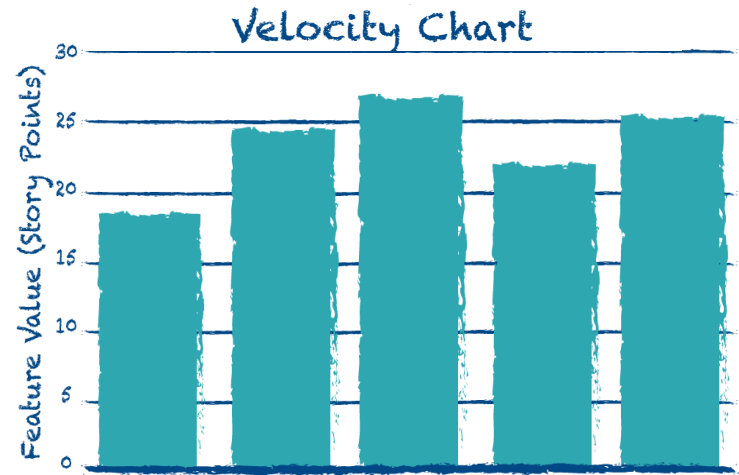
- Use observed velocity from past iterations.

2. Test Iterations

- Run a test iteration and measure actual velocity.

3. Forecast

- Compare available staff hours with story task breakdown estimates.



Another example of a velocity chart

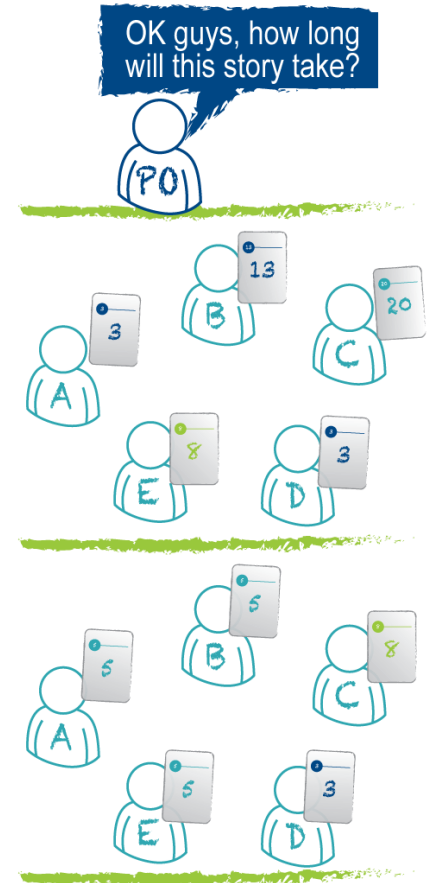
Planning Poker

- **Recognized as one of the best ways for Agile teams to estimate because it:**
 - encourages the entire development team to participate;
 - is easy, interactive, and fun; and
 - facilitates quick consensus on estimates.



Planning Poker Procedure

1. Team meets around a table with cards.
2. Moderator reads out a story.
3. Moderator answers questions.
4. Estimators privately select a size card.
5. Estimators show their cards.
6. Any discrepancies are discussed, additional questions are answered.
7. Estimators re-estimate by selecting a new size card and revealing it.
8. Repeat.



Planning Poker – Key Things to Remember

- The moderator can be anyone...product owner, scrum master, analyst, etc.
- Keep to the left side of the effort/accuracy curve.
- Limit each round of discussion to no more than a few minutes – a timer can help.
- Convergence is usually reached on the 2nd round.
- Absolute agreement is not required.
 - 5, 3, 5, 8, 5 – this would be sufficient to arrive at 5 as the estimate.



For More Information

Mike Cohn's site contains a good description of Planning Poker:

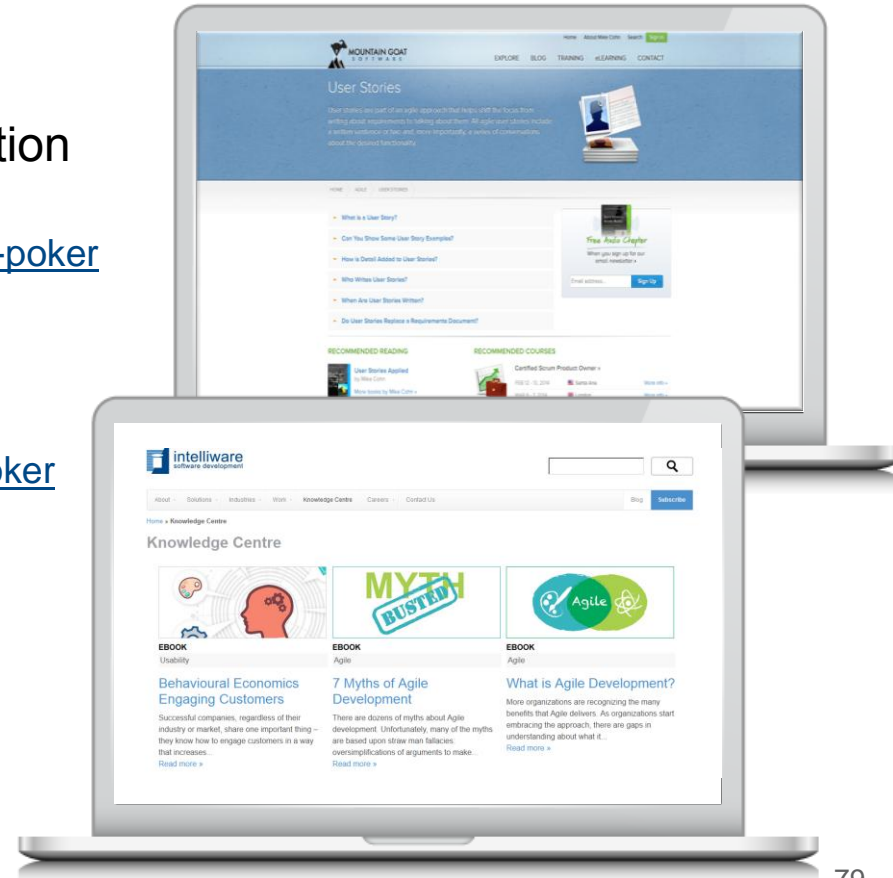
<http://www.mountangoatsoftware.com/agile/planning-poker>

The Crisp site is also a good source of Planning Poker basics:

<http://www.crisp.se/bocker-och-produkter/planning-poker>

Intelliware's Knowledge Centre contains several resources on the basics of Agile:

<http://www.intelliware.com/knowledge-centre>





intelliware
software development




A decorative graphic on the left side of the slide. It features a stack of three sticky notes at the top: a white one on top, a light green one in the middle, and a darker green one at the bottom. A large green arrow points downwards from the bottom sticky note to a single white sticky note at the bottom left. Another single green sticky note is located at the bottom right of the slide.

What You'll Learn in this Presentation:

- The basics of release and iteration planning.
- The differences between a release and an iteration.
- The basics of task planning.

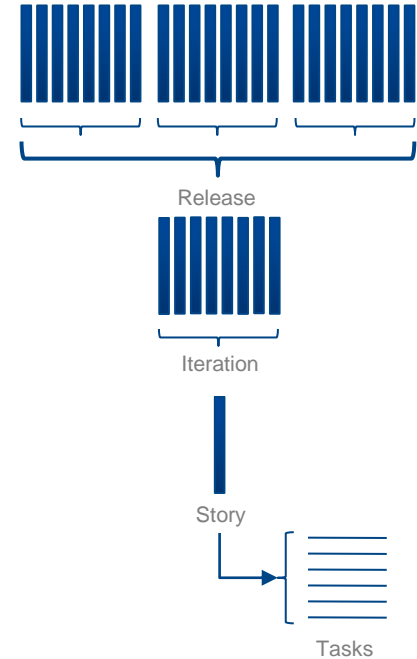


Some Things to Remember About Planning...

- **Agile planning is a continual process**
 - Development is iterative, and so planning has to respond accordingly.
 - **Our planning motto: Work the plan, not work to the plan**
 - In other words, “*Plans are worthless, but planning is everything.*”¹ - Dwight D. Eisenhower
 - Planning enables Agile teams to stay focused on business value.
- 

Agile Planning Horizons

- **Release:**
 - 1 to 3 month horizon with defined deliverables, multiple iterations, and focus on delivery to the end user.
- **Iteration:**
 - 1 to 3 week planning period with multiple user stories, greater precision than a release, and focus on delivery to the customer / product owner.
- **Task planning:**
 - 1 hour team meeting to break a story into more manageable tasks, assign tasks to developers, and focus on creating a plan to complete a user story.



The 3 levels of planning (from top to bottom):
release, iteration, and task planning for a user story

Who's Involved In Release Planning?

- **Customers / Product Owners:**

- define stories;
- set priorities; and
- put stories into iterations.



- **Developers:**

- estimate stories;
- point out significant technical risks; and
- establish velocity, a measure of team development capacity.



Defining a Release Plan

1. Establish iteration lengths

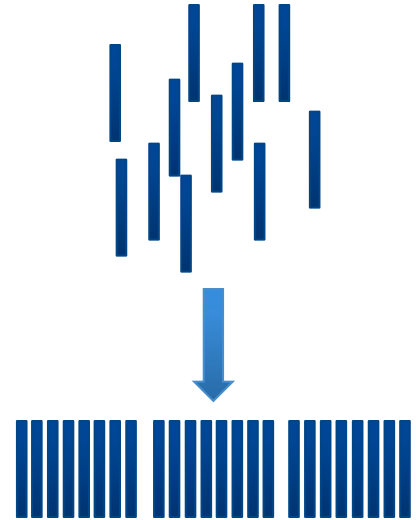
- Iterations are generally 1 to 3 weeks long.
- Developers decide what will be in the iteration in collaboration with the customer / product owner.
- Smaller projects tend to have shorter iterations.
- Size is important as release dates are usually based on external constraints.

2. Determine the velocity

- Estimate of how many estimating units can be completed per iteration.
- Developer responsibility.
- Unit of measurement used does not matter – be it points, ideal days, or something else.

3. Organize stories into iterations

- Based on business and technical priorities.



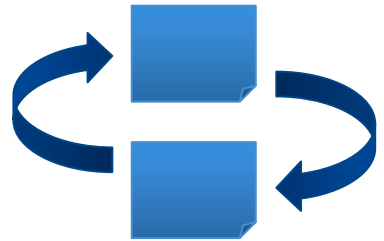
The process of deciding what goes into a release

Release Plan Variations

A release plan can (and will) be modified in numerous ways:

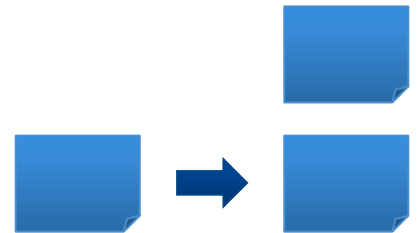
1. Change priority of stories

- Stories can be moved from one iteration to the next.
 - Decided by the customer / product owner and happens often.



2. Add new stories

- Need to determine priority and which iteration the story will fit into.
- A lower priority story may need to be bumped to make room for new stories of higher priority.



Release Plan Variations (continued)

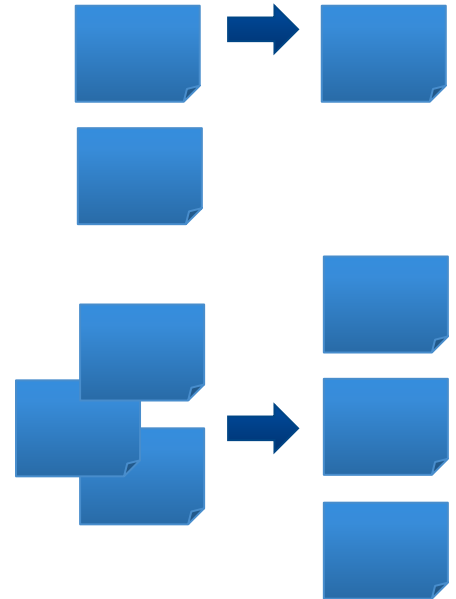
A release plan can (and will) be modified in numerous ways:

3. Delete stories

- Has the opposite effect of adding new stories; the gap will need to be filled with other stories previously excluded from the iteration.

4. Rebuild

- It's not unusual at the end of an iteration to review the remaining stories and completely rebuild the plan for the remaining iterations.
- Remember...***work the plan, don't work to the plan.***



Common Release Plan Problems & Solutions

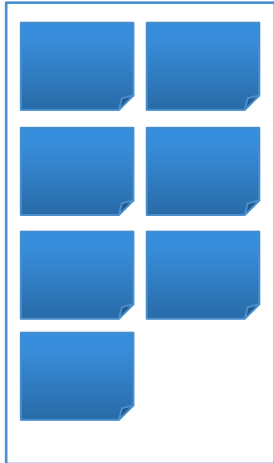
- **Story too large to estimate**
 - The story should be split into smaller parts.
- **Story spans iterations due to external dependencies**
 - Best to have the developers split the story into two parts:
 1. Finish what can be completed now.
 2. Create second story representing what is not yet done.
- **Story 'blows up' due to scope or technical issues**
 - The story needs to be halted and re-scoped or re-planned.
- **Release date slides**
 - Development team is responsible for reporting this ASAP.
 - Developers and customer / product owner need to work together to de-scope and re-plan.



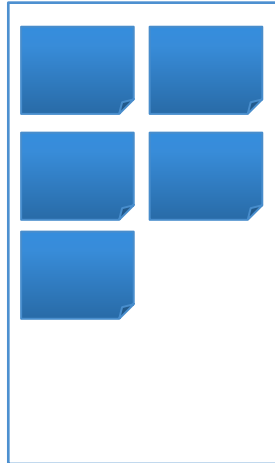
Example of a Release Plan

- Here's an example of a release that consists of 2 iterations with 12 stories.
- The release plan is presented 2 different ways.

Iteration 1



Iteration 2



| Release | Iteration | Stories | Size |
|---------|-----------|---|------|
| Alpha | 1 | Home Page | 1 |
| | | Login | 1 |
| | | Display Account Summary List | 2 |
| | | Display Account Details | 1 |
| | | List Transaction Summary for an Account | 1 |
| | | Add New Transaction | 2 |
| | | Logout | 1 |
| | | TOTAL | 9 |
| | 2 | Add New Account | 2 |
| | | Deactivate and Remove Existing Account | 1 |
| | | Modify Account Information | 1 |
| | | Delete a Transaction | 1 |
| | | Modify Transaction or Move to Different Account | 2 |
| | | TOTAL | 7 |

Iteration Planning

Characteristics of iterations to consider when planning:

- Generally 1 to 3 weeks long.
- More developer-focused than they are customer / product-owner focused.
- More precise than releases in terms of their planning focus.
- Have a specified length of time based on fixed dates.
- Start by holding an iteration planning meeting.



Iteration Planning Meeting

Objectives:

- **Acknowledge** accomplishments of last iteration.
- **Determine** the level of overall progress.
- **Review** problems and issues.
- **Establish** objectives for next iteration.
- **Task** planning for target stories (optional).



Estimating Velocity

- **Optimizing velocity is fraught with peril** – regardless of the team size, the difficulty level of the work, or the skill level of the team.
 - Predicting the future is unreliable – don't do it.
 - Velocity is about the team, not individuals.
 - Velocity is a measure of capacity, not performance.
- **Using velocity to compare teams is problematic at best**
 - No two projects, domains or teams are the same.
- **Base velocity on “yesterday's weather”**
 - *Yesterday's weather*: the velocity for the upcoming iteration is assumed to be the same as that of the previous iteration.

Timing of Task Planning

- **Two Options:**

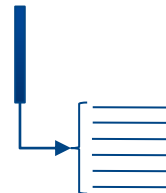
- 1. Tasking at the iteration-level**

- Scrum recommends task planning as part of iteration planning, i.e. at the start of an iteration.



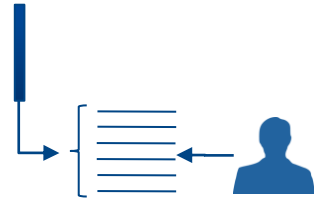
- 2. Tasking at the story-level**

- This can take place as part of iteration planning or separately.
- At Inteliware, we prefer to task stories when we open them.



Task Planning Meeting

- **Developer-focused meeting to deconstruct a story into a series of finer-grained tasks**
 - Customers / product owners can attend, but the meeting content is likely too detailed to be of interest to them.
- **Tasks are recorded in a way that makes them visible**
 - Whiteboard or chart paper are typical tools.
- **Tasking makes the story easier to develop**
 - Forces the development team to think about implementation issues.
 - Tasks can be worked on by multiple pairs or individuals simultaneously.



Example Task Board

Tasks in progress are marked with lead developer initials and start date

When a pair of developers works on a task both initials are added

| Who | Start | End | Est. | SPS-11: Show Account Balance | MH | 8.0 |
|-------|-------|-----|------|---|----|-----|
| LDL | 3 | 4 | 0.5 | Update project DB schema to add new Acc Bal field | ✓ | |
| MH/SH | 3 | | 1.0 | New batch process to update balances to new Acc Bal field | | |
| GC/BO | 4 | | 0.5 | DAO | | |
| MH | 3 | | 0.25 | Add new field to UI | ■ | |
| | | | 0.75 | Handle negative balances | | |
| | | | 0.5 | Handle \$0.00 balances for new clients | | |
| | | | 0.5 | Initial customer demo | | |
| | | | 1.0 | Update integration tests | | |
| | | | 0.5 | DAT | | |
| | | | 1.0 | Check UI changes in all browsers | | |
| | | | 2.0 | Update Stored Procedures on DB server for new Acc Bal field | | |
| | | | | | | |

Title bar with story name, ID, tasked estimate and story lead initials

Completed tasks are identified in some way

Pause symbol indicates a task on hold

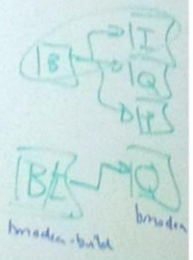
Removed task is crossed out, not erased

New task identified after tasking meeting is added to the bottom of the list

A Task List

- **Task list:** the visible list keeps the team focused on tasks.
- **Task meeting process:** the task name and size are added during the tasking meeting – the who and when are added later.
- **Designed for pull mode work assignment**
 - Pulling tasks encourages the team to work together.
 - Sometimes tasks are assigned to developers or pairs for specific reasons, but this occurs in the context of the team.

| who | start | end | DCA-000A Environment Setup | 6 1/4 |
|---------------|--------------|---------------|--|-------|
| PV/SM | 3 | 8 | 1/4 find 3 good PCs 6462341 6462341 | |
| PV/SM | 3 | 6 | 3/4 setup to run linux image (1 PC) | |
| JM | 6 | 6 | 1/4 populate new "Firmware" repo with sample app (survives 9 with) | |
| SM | 3 | | - ask backup question | |
| JM | 6 | 6 | 1/2 - remove packages ca. int. ... -> ca. int. ... ? | |
| Jm | 4 | 4 | 1/4 - set up BT de-masking? | |
| Jm | 11 | | 1/4 set up 3m de-masking } based on image on net | |
| | | | create nexus repo | |
| SM | 10 | 11 | 1/4 - set up crumble project | |
| PV | 6 | 10 | 1/4 - set up Jenkins v1 | |
| PV | 10 | 10 | 1/2 - Jenkins: build / unit | |
| PV | 11 | | 1/2 - Jenkins: web integration tests | |
| | | | 1/2 - Jenkins: deploy to 1st env. 2nd | |
| | | | 1 - setup 1st env (vm) 2nd | |
| | 10 | 10 | 1/2 - Sonar Setup RHEL6.4 | |



An example task board for a story in progress

Task Attributes

- **Tasks should be quantifiable** (in terms of time needed to complete)
 - Tasks are very specific and should have estimates in days, part-days or even hours.
 - Task estimating occurs in the task planning meeting.
- **Tasks should be small**
 - No more than a day or two per task.
- **Tasks can be technical**
 - Refactorings, system upgrades, etc. are unavoidable.

Task Tracking at Task Planning

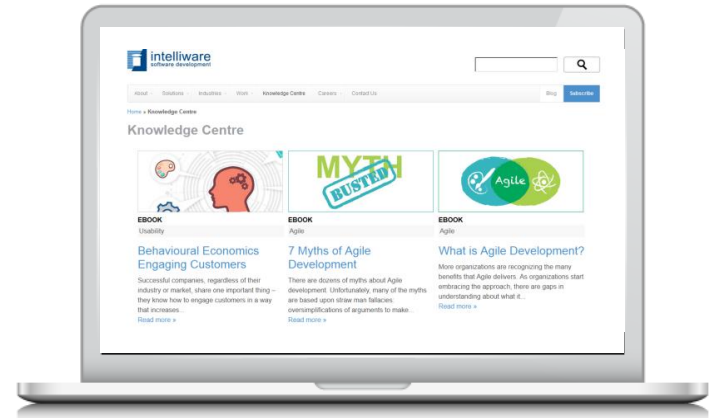
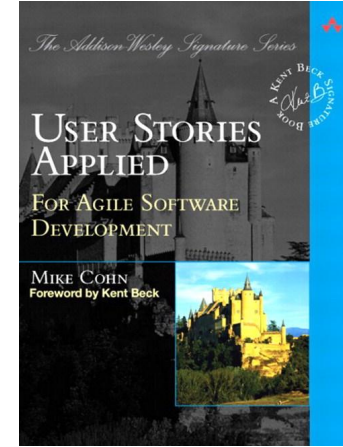
- **Task size estimates provide an early indication of potential issues with the story.**
- **Example:**
 - Story size estimate: 5 ideal days.
 - Sum of task estimates: 8 ideal days.
 - Over-estimate: 3 ideal days or 160%.
- **Outcome:**
 - Team should discuss why task estimate > story estimate.
 - Team should consider ways to develop the story within the original estimate, or split it (and report back to customer / product owner).
- **Tracking during development also provides an early indication of possible issues – don't leave things festering!**

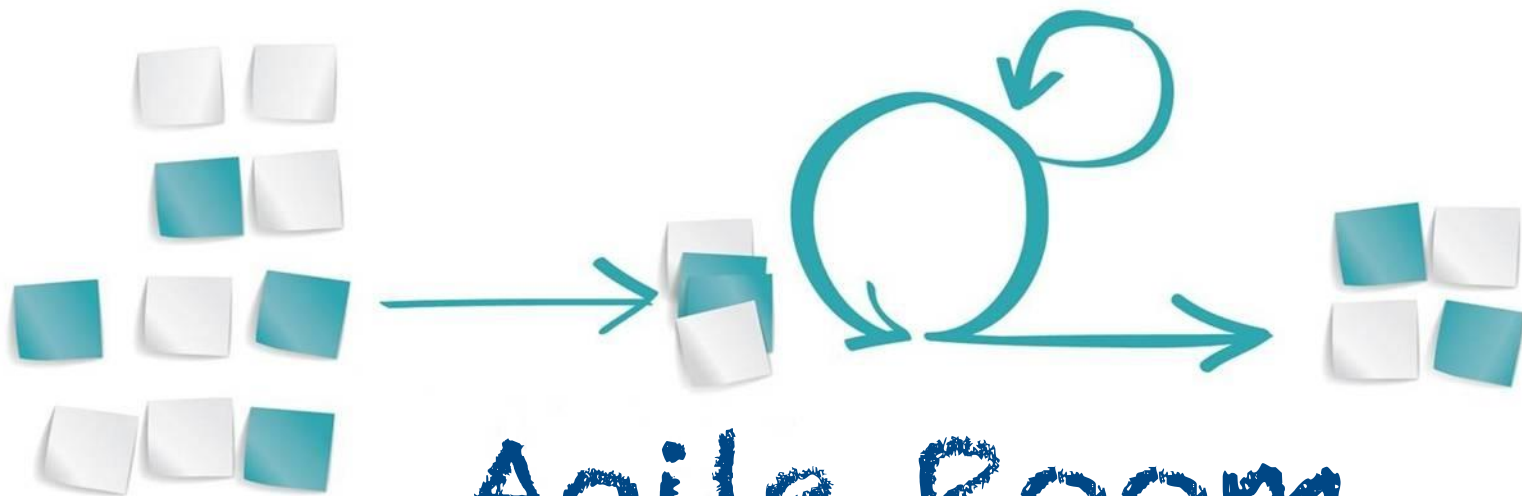
For More Information

Mike Cohn's book, "User Stories Applied" is the definitive reference on release and iteration planning.

Intelliware's Knowledge Centre contains several resources on the basics of Agile (see next slide for titles in our Agile series):

<http://www.intelliware.com/knowledge-centre>






Agile Room (Team) Dynamics

A stack of three sticky notes: a white one on top, a green one in the middle, and a larger green one at the bottom. A green arrow points from the bottom green note down to a white sticky note at the bottom left.


Agile Room (Team) Dynamics: Getting Teams Performant (and Happy)

What You'll Learn in this Presentation:

- The signs to look for in a dynamic Agile team room.
 - How to get a team performant (and happy).
- 
- A single green sticky note is positioned in the bottom right corner of the slide.

A decorative graphic on the left side of the slide featuring three overlapping sticky notes: a white one at the top, a green one in the middle, and another white one at the bottom. A large green arrow points downwards from the middle green sticky note to the bottom white sticky note.

Why (Room) Team Dynamics are Important

- Agile focus on people strongly related to teams.
 - In a team environment, team dynamics translates directly into productivity.
 - A happy team will inherently be more productive. Agile is no exception.
 - Conversely, an unhappy team can be extremely non-functional.
 - When a team isn't working well, everyone suffers.
- 
- A single green sticky note is located in the bottom right corner of the slide.

The 11 Signs of Good Room Dynamics

1. Deliverables are EVERYONE's responsibility.
2. Team Lead and Architect roles may be designated, but delivery is EVERYONE's responsibility.
3. Everyone is engaged & respected.
4. Healthy debate and conflict happens – and compromise.
5. Whiteboard sessions.
6. Members help each other.
7. Team members have confidence in each others' abilities.
8. No egos.
9. Buzz in the room.
10. Celebrations of small successes.
11. Music.

1. Deliverables are **EVERYONE's** Responsibility

- The team must be working as a team towards a common goal.
 - Everyone has the same understanding of the overall project objective.
 - No silos.
- Certain team members may be focused on specific stories or tasks, but they are not solely responsible for them.
 - Everyone on the team is aware of what everyone else is working on and are willing to assist when needed, even when not asked.
 - Refusing to help others is not an option.



2. Team Lead and Architect Roles may be Designated, but Delivery is **EVERYONE's** Responsibility

- Leadership roles, such as Architect or Team Lead, are necessary and important.
 - These roles involve responsibilities that require certain skills and don't make sense for the whole team to do.
 - They do not denote seniority over other team members.
- The whole team works together to keep the project on track.
 - There is a collective focus in the room on the overall delivery objective – success is a team objective; it is not the responsibility of one or two individuals.



3. Everyone is Engaged & Respected

- In a dynamic Agile room, everyone:
 - is an equal,
 - listens,
 - is heard, and
 - participates.
- There is no:
 - avoiding the team or
 - disrespectful behaviour.
- There are no heroes.



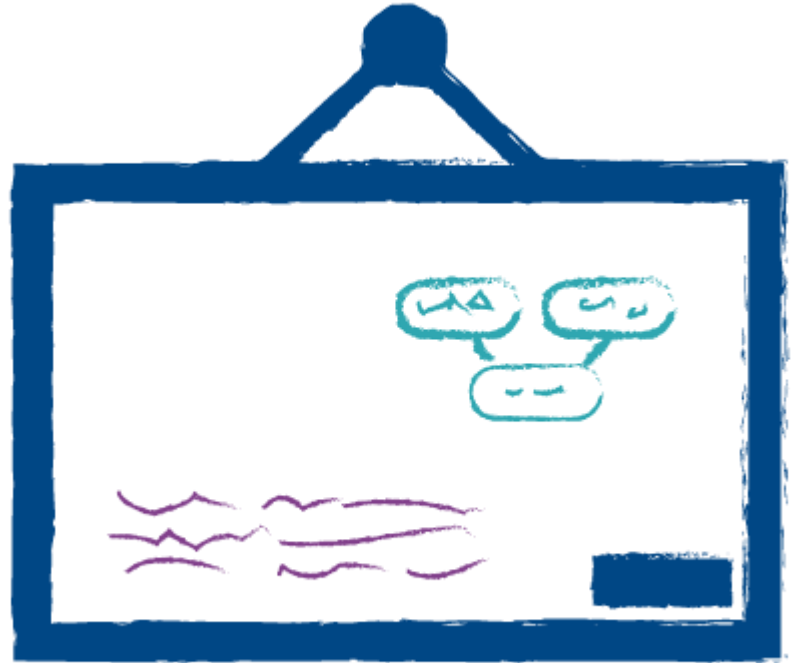
4. Healthy Debate and Conflict

- Debate and conflict are normal.
 - Debate, arguments, and conflicts happen often in the room.
 - Facilitated by the fact that everyone feels free to speak up and that their opinions will be respected.
 - The focus is on the good of the project and maximizing value to the customer.
 - Personal attacks are not tolerated in any way whatsoever.



5. Whiteboard Sessions

- Whiteboards are an important feature of the room used to communicate design diagrams, task lists, etc.
- Whiteboards are used as a common focal point for design discussions, tasking meetings, etc.
 - Everyone is allowed to participate in these discussions, at their own discretion.
 - Closely related to healthy debate.
- No whiteboard in the room is left blank!



6. Members Help Each Other

- A stuck developer is an unproductive developer.
 - Nobody is afraid to ask for help from other team members.
 - Assistance is offered without question.
- Collective sense in the room that helping each other is critical. Works in two ways:
 - If we help each other, the team will benefit.
 - I may need you to help me one day.
- Standup is a common mechanism to point out difficulties and ask for help.



7. Team Members Have Confidence in each Others' Abilities

- Everyone on the team is aware of and respects their own and other team member's abilities.
 - Varying skill sets and levels of proficiency are known and appreciated – not everyone is a rocket scientist.
 - Team members don't sign up for tasks that they're not capable of completing.
 - Likewise, when team members take on a task, this decision is respected by other team members.
- The team accepts that delivery relies on a team with diversified skills and levels of experience.



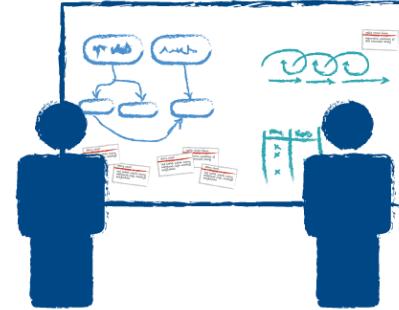
8. No Egos

- No cowboy programmers.
- No 'last minute' heroes.
- Yes Servant Leaders.



9. Buzz In The Room

- The project room immediately appears to be a hive of activity.
 - Everyone is busy and engaged.
 - The team is located around a central table.
 - No outliers.
 - There's lots of talking:
 - Pairs working together.
 - Ad hoc discussions.
 - Whiteboard sessions.
 - Whiteboards are covered with stuff.
 - It's not exactly neat and clean.



10. Celebrations of Small Successes

- In Agile, a successful project is not one event but instead is the cumulative effect of a series of small successes.
- Agile teams recognize this and celebrate small successes often by:
 - Showing appreciation for other team member's efforts.
 - Going out to lunch together.
 - Bringing food or drinks into the project room at the end of the day.



11. Music

- Music can often be heard in an Agile team room because...
 - Developers enjoy listening to music while they work.
 - The atmosphere is relaxed.
 - Everyone gets a chance to play what they like.
 - Nobody criticizes other's musical preferences (within reason 😊).
 - It's not too loud.
- No headphones!
 - This is a sign of somebody who's not fully engaged with the team.



How to Maintain Healthy Project Room Dynamics

These are the things that Agile Teams implement to maintain healthy project room dynamics:

1. Group negotiation of team rules.
2. Team lunches.
3. Storming as a given.
4. Pairing negotiation.
5. Always listen in.
6. Conflict amongst team members.
7. Decisions.
8. Engage the larger development team.
9. Incorporating new team members.
10. Humour & Food.

1. Group Negotiation of Team Rules Guidelines

- Collective confirmation regarding:
 - Stand-up.
 - Story writing structure on the board.
 - Scrum board.
 - Bug tracking and wiki usage (e.g. Jira & Confluence).
 - Retrospectives.
- Guidelines can always be changed as the team settles in. Usually done as a result of end-of-sprint retrospectives.
- First order of business: Team Lunch 😊!



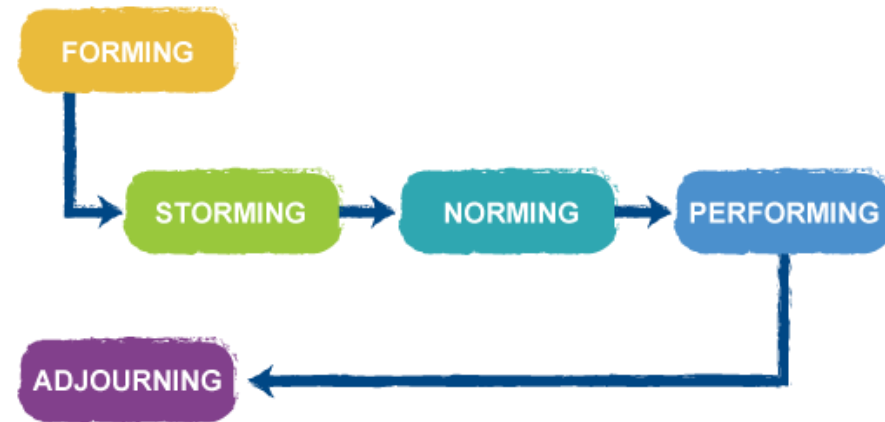
2. Team Lunches

- Scheduled in calendar in a repeating cycle (~ 3-4 weeks).
 - 1st team lunch.
 - Team building activities to break the ice.
- Initiated by any member of the team.
Important that whole team attends!!!!
- Takes about 3 lunches for team to warm-up to each other.
- Discuss 5 Stages of Team Development → Next Slides.



3. Storming as a Given...The 5 Stages of Team Development

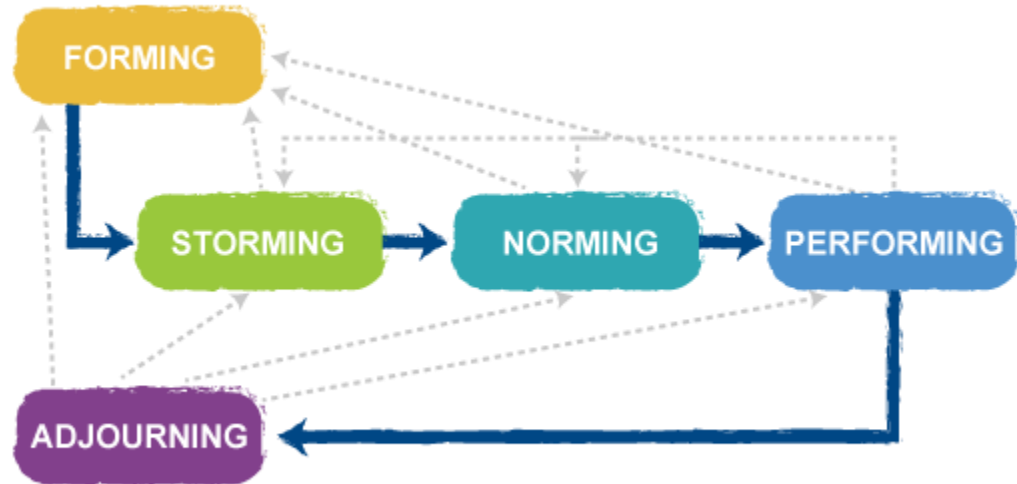
- Forming
 - Little agreement, lack of purpose.
- Storming
 - Conflict, power struggles, increased clarity of purpose.
- Norming
 - Agreement, clear roles & responsibilities.
- Performing
 - Clear vision and purpose; focused on common goal.
- Adjourning
 - Project/task complete; hopefully with good feelings about outcome.



Tuckman, Bruce, 1965. "[Developmental sequence in small groups](#)". Psychological Bulletin ,**63** (6): 384–99.

What Happens if the Team Changes

- Why might the team change?
 - New team formed for new project.
 - Maternity leave.
 - Somebody leaves the company.
 - New hire added to supplement the team.
 - Somebody is added that has a specific skill set.
 - Team members moved between teams to cross-pollinate skills and practices.



Tuckman, Bruce, 1965. "[Developmental sequence in small groups](#)". Psychological Bulletin ,**63** (6): 384–99.

You Are Expected to Storm

- Must be verbalized by the Team Lead, Project Manager or team coach to ensure team has common expectations.
- Introduce concept at first team lunch.
- Allows team members to disagree passionately (and even get annoyed with each other) and know that it is an expected part of growing pains.
- Early retrospectives review where we think we are on the 5 steps of team formation.
- The whole team storms, some are more noticeable than others.



4. Pairing Negotiation

- Discuss briefly how you like to pair.
 - Want pair to point out typos or mistakes immediately?
 - Drive for several hours and switch, or ping pong?
- What are habits you have (or not aware of that have been pointed out to you in the past)?
- What are your normal work hours?
- Give pair permission to speak-up or stop you if you are doing something they don't like.
- This is especially important at the beginning of project for all “new pairs.”
 - Additional reading: *Pair Programming Illuminated* by Williams & Kessler.



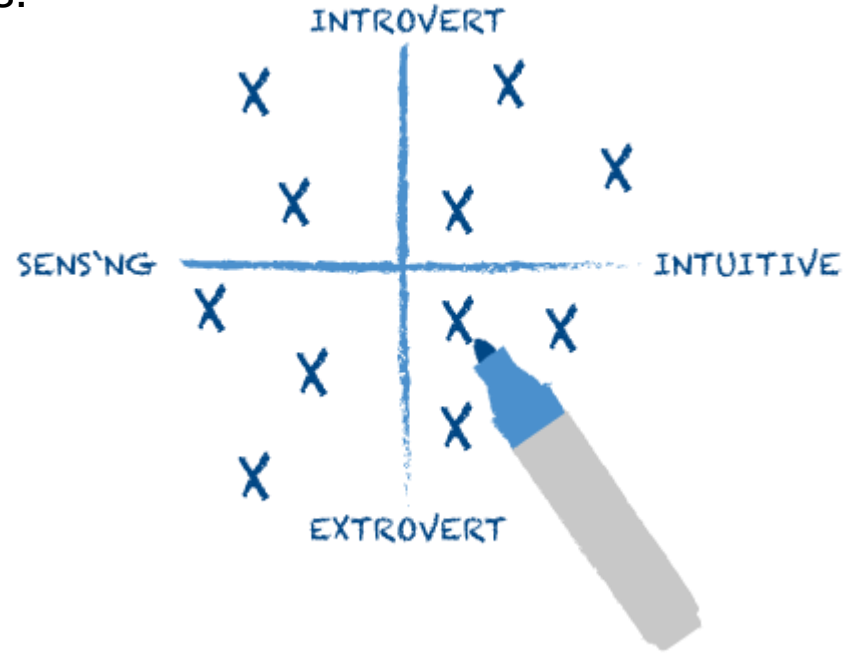
5. Always Listen In

- Pay attention to discussions going on in the room!
 - It takes a village!
 - At the end of the day, it's EVERYONE'S fault if something goes wrong (especially true if a new or junior member caused it).
- Tune in and out of conversations around you.
 - Saves time when you have to switch pairs or a task.
- Do as much pairing as possible and practical.
 - Ideally identify tasks that should be paired on during tasking.
- Verbally communicate code changes that may impact others as soon as it's pushed.
- Headphones? ☹ Seriously??



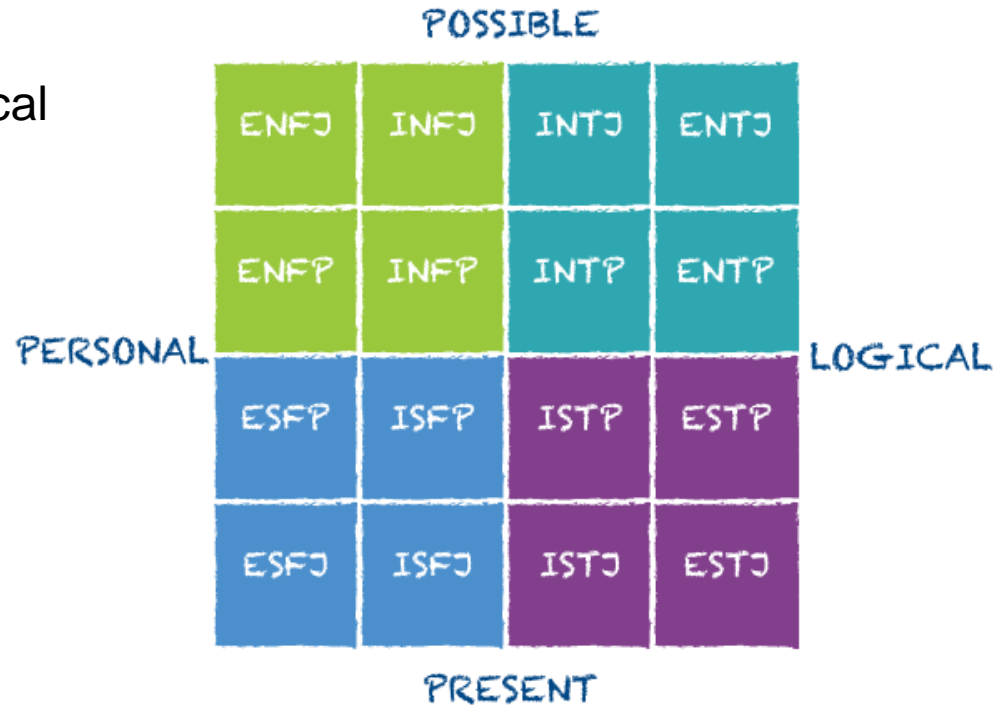
6. Conflict Amongst Team Members - Know the Personality Types

- Not easy initially, but team building helps.
- Kindergarten rules.
- Always give an opt-out option and if not possible - the lesser of two evils.
- Include everyone in their own way.
- Don't allow others to be interrupted by stronger personalities in a discussion.
- Pay attention to non-verbal cues & ask follow-up questions.



Myers Briggs Personality Test

- Based on the theory of psychological types.
 - Rational (judging) – thinking & feeling.
 - Irrational (perceiving) – sensation & intuition.
- Knowing your personality type and the types on your team will help you better interact with them.
- Online Myers Briggs Test:
www.humanmetrics.com/cgi-win/jtypes2.asp



Conflict Amongst Team Members - Storming

- Let people storm, but monitor that they move beyond that stage.
 - If two people are storming, let them work it out.
- Don't enable avoidance, just to be "nice".
 - Don't allow team members to avoid each other via not pairing when they could or should be.
- Pay attention to non-verbal cues.
 - Folded arms.
 - Raised eyebrows.
 - High pitched voice.
- Be aware of the differences among:
 - Difference of technical opinion vs.
 - Personality conflict vs.
 - Personal styles.



Conflict Amongst Team Members - Intervening

- Only step in when it becomes unhealthy/uncomfortable for the team and absolutely necessary.
 - If you must intervene, discuss with them separately & privately and provide an objective point of view, then arrange a mediation if absolutely necessary.
- Anyone on the team can step in.
- Come to consensus and then be consistent. Don't agree to disagree and then implement multiple flavours of the same solution.



7. Decisions

- The team is responsible for delivery, but technical decisions are **not** the responsibility of the whole team...
 - Some members of the team, such as PMs, BAs and QAs, do not have the skills, experience and background to be involved in these decisions.
 - Larger final decisions that have impact on overall architecture are usually arrived at as a result of discussion of one or two senior team members. These decisions are then communicated to the rest of the team to seek consensus.
 - Day-to-day technical decisions are made by the team consistent with the shared technical direction.
 - If options impact scope, budget or future feature options, PM and/or BA present to Client for the final call if necessary.



8. Engage the Larger Development Team

- It takes the development village.
- Interact with other co-workers beyond your team during your project's lifetime.
- Don't spin wheels too long.
 - Ask around if stuck. Your company's knowledge isn't limited to your project room.
 - Know and engage your options before spending 2 to 3 days on a problem 😊.
 - Document and share answer!
 - By asking around, people you talked to will remember next time they encounter a similar problem.



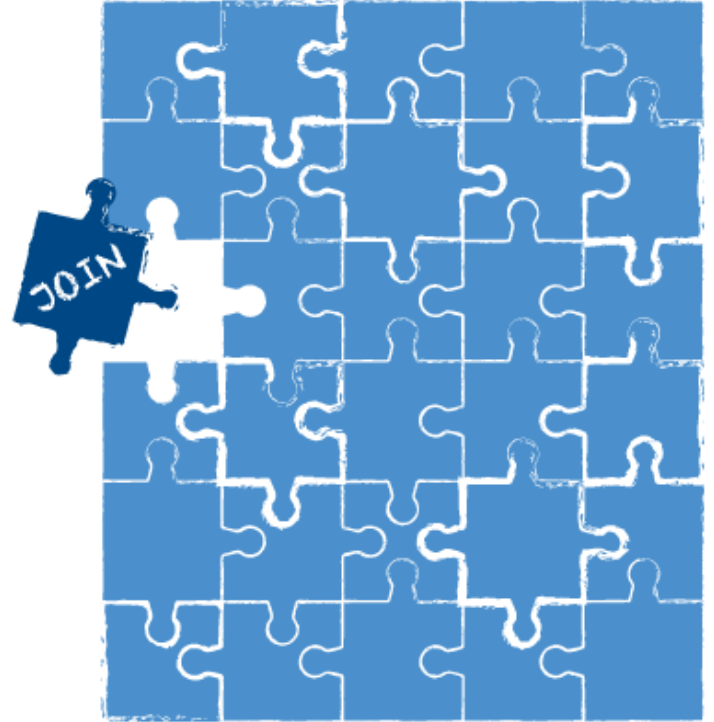
9. Incorporating New Team Members - Make New Members Feel Welcome

- When the team is disrupted, storming is expected again in addition to the other 4 stages.
 - Good time for team lunch.
- New team members are responsible for asking questions partly to learn and partly to challenge the status-quo. They are by definition “fresh eyes”.
 - This is an opportunity to learn where team’s process and documentation is lacking.
- Existing team members should be confident in the existing decisions that were made by the team.



New Team Member – Make Yourself Fit In

- Accept that you represent a disrupting force.
 - The team will storm. Don't take it personally.
- Don't be afraid to ask questions.
 - But respect history.
 - Previous decisions may seem insane, but they were probably made for reasons that made perfect sense at the time.
- Go out of your way to fit in with your new team mates.
 - It's okay to rock the boat...but don't tip it over!



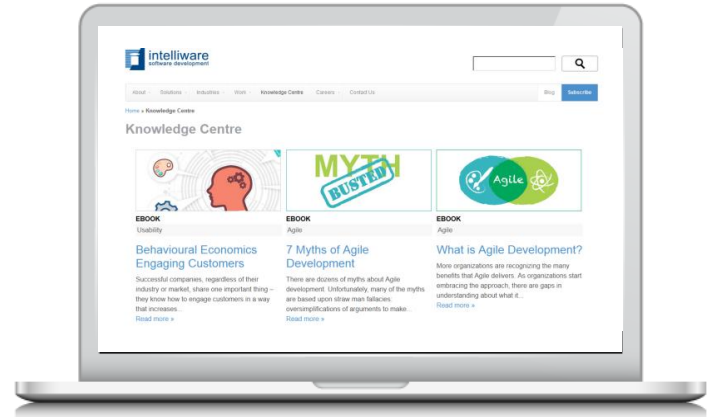
How to Make a Team Happy

- Humour & Food
- Food & Humour
- Humour & Food
- Food & Humour
- Did I mention Humour?.....
What about Food?



For More Information

- Inteliware's Knowledge Centre contains several resources on the basics of Agile (see next slide for titles in our Agile series):
<http://www.inteliware.com/knowledge-centre>
- Further reading that we recommend:
 - *The Human Side of Agile* by Gil Broza.
 - *Peopleware* by Tom DeMarco and Timothy Lister.



About Intelliware Development Inc.

Intelliware is a custom software, mobile solutions and product development company headquartered in Toronto, Canada. Intelliware is a leader in Agile software development practices which ensure the delivery of timely high quality solutions for clients. Intelliware is engaged as a technical partner by a wide range of national and global organizations in sectors that span Financial Services, Healthcare, ICT, Retail, Manufacturing and Government.



[/company/intelliware-development-inc-](https://www.linkedin.com/company/intelliware-development-inc-)



[/intelliware.inc](https://www.facebook.com/intelliware.inc)



[/intelliware_inc](https://twitter.com/intelliware_inc)



[/GooglePlusIntelliware](https://plus.google.com/+GooglePlusIntelliware)



www.intelliware.com