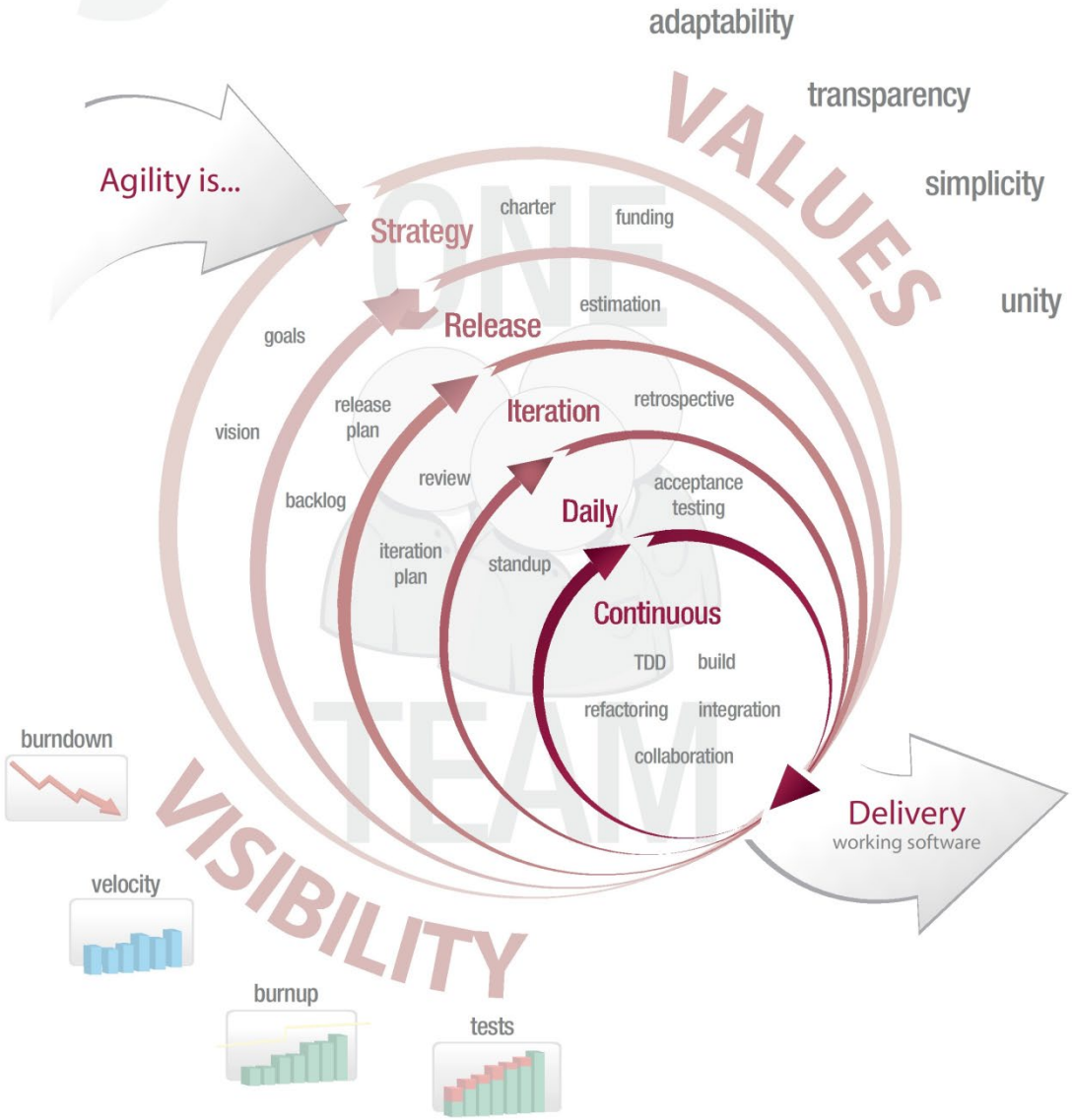
Agile Development



Accelerate Success

Author & Instructor: Vijay Nathani

Agile Software Development

Table of contents

Introduction

In the 1960s, customer involvement and collaboration weren't problems. Then, the computers were less powerful, the users were fewer, the applications were simpler, and the idea of milestones was still unknown. Developers used short iterations of one or two days. They did meet with the customer and together sketch out on paper what he or she wanted. They'd discuss the problem until developers understood it. Then developers would go to their desk, design and code the solution, punch up the cards, and compile the program. Once the compile and link were clean, they did run some test data against the program. Then they'd return to the customer and ask, "Is this what you wanted?" People didn't realize it at the time, but this was heaven.

As the applications and technology became more complex and the number of stakeholders in a project increased, practices were inserted to coordinate the communication among the increased numbers of participants. For instance, because many stakeholders were involved, we began to collect all of their requirements prior to starting development. We felt that the system should implement the sum of their respective requirements. Because documentation was such an inadequate medium of communicating, we started to use pictures to communicate, supporting these pictures with text. And because pictures were imprecise, we developed modeling techniques to formally represent the pictures. Each step drove a wedge between the stakeholders and the developers. We went from face-to-face communication to documentation. We went from quick turnaround to lengthy requirements-gathering phases. We went from simple language to artifacts that were arcane and highly specialized.

In retrospect, the more we improved the practice of software engineering, the further we widened the gap between stakeholders and developers. The last step in the estrangement was the introduction of *waterfall methodology*, which embodies all the flaws of sequential development. Waterfall methodology gathers all the requirements, then creates the design, then writes the code, then develops and runs tests, and finally implements the system. Between each of these steps, or phases, were review meetings. Stakeholders were invited to these meetings to review the progress to date. At these meetings, developers and managers would ask customers "Do these requirements that we've gathered and the models that demonstrate them constitute a full and accurate representation of what you want? Because once you say yes, it will be increasingly expensive for you to change your mind!" As you can see, this wording implies a contract between the customers and the developers. By this point, there was little in-person collaboration; in its place were contracts that said, "If you agree that what I showed you is the complete description of your requirements, we will proceed. If you don't agree, we'll continue to develop requirements until you give up!"

Einstein's definition of insanity: doing the same thing over and over and expecting different results. Surprisingly, this approach is common. If a project being managed by a defined approach (Waterfall) fails, people often assume that the project failed because the defined approach wasn't adhered to rigorously enough. They conclude that all that is needed for the project to succeed is increased control and project definition.

Iterative Development

Software development is a cooperative game of invention and communication.

Humans working together, building something they don't quite understand. Done well, the result is breathtaking; done poorly, dross.

Purely people factors predict project success, overriding choice of process or technology.
There is still a lot of resistance in our industry to this idea.
A well-functioning team of adequate people will complete a project almost regardless of the process or technology they are asked to use (although the process and technology might help or hinder them along the way).
Dave A. Thomas, founder of Object Technology International, a company with a long record of successful projects, summarized his success formula: "Some people deliver software; some don't. I hire those that have delivered."

The very reason for incremental and iterative strategies is to allow for people's inevitable mistakes to be discovered relatively early and repaired in a tidy manner.
That people make mistakes should really not be any surprise to us. And yet, some managers seem genuinely surprised when the development team announces a plan to work according to an incremental or iterative process. I have heard of managers saying things like

> "What do you mean; you don't know how long it will take?"
> Or
> "What do you mean; you plan to do it wrong the first time? I can go out and hire someone who will promise to do it right the first time."

In other words, the manager is saying that he expects the development team not to make any major mistakes or to learn anything new on the project.

One can find people who promise to get things right the first time, but one is unlikely to find people who actually get things right the first time. People make mistakes in estimation, requirements, design, typing, proofreading, installing, testing . . . and everything else they do. There is no escape. *We must accept that mistakes will be made and use processes that adjust to the fact of mistakes.*

Given how obvious it is that people make mistakes, the really surprising thing is that managers still refuse to use incremental and iterative strategies. This is because

- Humans prefer to fail conservatively rather than risk succeeding differently
- Humans have difficulty changing working habits.

Consider a manager faced with changing from waterfall to incremental or iterative scheduling. The waterfall strategy is accepted as a normal, conservative way of doing business, even though many people think it is faulty. The manager has used waterfall strategy several times, with varying success. Now, one of his junior people comes to him with a radically different approach. He sees some significant dangers in the new approach. His reputation is riding on this next project. Does he use the normal, conservative strategy or try out the risky new strategy?
Odds are that he will use the normal, conservative strategy, a "guaranteed" standard outcome, rather than one that might work but might blow up in strange ways.

This characteristic, "preferring to fail conservatively rather than to risk succeeding differently," gets coupled with people's fear of rejection and the difficulty they have in building new work habits. So managers continue to use the long-abused one-pass waterfall development process. Based on this line of thinking, I expect that people will continue to use

the waterfall process even in the presence of mounting evidence against it and increasing evidence supporting incremental and iterative development. Use of the waterfall process is anchored in a failure mode.

What is Agility?

Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battening down the business hatches to ride out fearsome competitive 'storms.' It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many companies now fear.

Most development teams are so far away from doing agile development that it is easy to give them a meaningful, "more agile" direction:

- Decentralize decision making
- Integrate the code more often
- Shorten the delivery cycles
- Increase the number of tests written by the programmers. Automate those tests
- Get real users to visit the project
- Reflect

An agile team

1. Take responsibility for failures as well as successes.
2. Can introduce you to their stakeholders, who actively participate on the project.
3. Write high-quality code that is maintained under CM (Configuration Management) control.
4. Produce working software on a regular basis.
5. Welcome and respond to changing requirements.
6. Can show and run their regression test suite.
7. Automate the drudgery out of their work.

Agile Manifesto

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.

A good way to think about the manifesto is that it defines preferences, not alternatives, encouraging a focus on certain areas but not eliminating others. The Agile Alliance values:

1. **Individuals and interactions over processes and tools**

   Teams of people build software systems, and to do that they need to work together effectively – including but not limited to programmers, testers, project managers, modelers, and your customers. Who do you think would develop a better system: five software developers and with their own tools working together in a single room or five low-skilled "hamburger flippers" with a well-defined process, the most sophisticated tools available, and the best offices money could buy? If the project was reasonably complex my money would be on the software developers, wouldn't yours? The point is that the most important factors that you need to consider are the people and how they work together because if you don't get that right the best tools and processes won't be of any use. Tools and processes are important; don't get me wrong; it's just that they're not as important as "working together effectively".

   Remember the old adage: a fool with a tool is still a fool. As Fred Brooks points out in The Mythical Man Month, this can be difficult for management to accept because they often want to believe that people and time, or men and months, are interchangeable.

   People are the most important ingredient of success. A good process will not save a project from failure if the team doesn't have strong players, but a bad process can make even the strongest of players ineffective. Even a group of strong players can fail badly if they don't work as a team.

   A strong player is not necessarily an ace programmer. A strong player may be an average programmer but someone who works well with others. Working well with others, communicating and interacting is more important than raw programming talent. A team of average programmers who communicate well are more likely to succeed than is a group of superstars who fail to interact as a team.

   The right tools can be very important to success. Compilers, interactive development environments (IDEs), source code control systems, and so on, are all vital to the proper functioning of a team of developers. However, tools can be overemphasized. An overabundance of big, unwieldy tools is just as bad as a lack of tools.

   Our advice is to start small. Don't assume that you've outgrown a tool until you've tried it and found that you can't use it. Instead of buying the top-of-the-line, mega expensive source code control system, find a free one and use it until you can demonstrate that you've outgrown it. Before you buy team licenses for the best of all computer-aided software engineering (CASE) tools, use whiteboards and graph paper until you can unambiguously show that you need more. Before you commit to the top-shelf behemoth database system, try flat files. Don't assume that bigger and better tools will automatically help you do better. Often, they hinder more than they help.

   Remember, building the team is more important that building the environment. Many teams and managers make the mistake of building the environment first and expecting the team to gel automatically. Instead, work to create the team, and then let the team configure the environment on the basis of need.

2. **Working software over comprehensive documentation**

   When you ask a user whether they would want a fifty-page document describing what you intend to build or the actual software itself, what do you think they'll pick? My guess is that 99 times out of 100 they'll choose working software. If that is the case, doesn't it make more sense to work in such a manner that you produce software quickly and often, giving your users what they prefer? Furthermore, I suspect that users will have a significantly easier time understanding any software that you produce than complex technical diagrams describing its internal workings or describing an abstraction of its usage, don't you?

   Documentation has its place, written properly it is a valuable guide for people's understanding of how and why a system is built and how to work with the system. However, never forget that the primary goal of software development is to create software, not documents – otherwise it would be called documentation development, wouldn't it?

   Software without documentation is a disaster. Code is not the ideal medium for communicating the rationale and structure of a system. Rather, the team needs to produce human-readable documents that describe the system and the rationale for design decisions. However, too much documentation is worse than too little. Huge software documents take a great deal of time to produce and even more time to keep in sync with the code. If they are not kept in sync, they turn into large, complicated lies and become a significant source of misdirection.

   It is always a good idea for the team to write and maintain a short rationale and structure document. But that document needs to be short and salient. By short, I mean one or two dozen pages at most. By salient, I mean that it should discuss the overall design rationale and only the highest-level structures in the system.

   If all we have is a short rationale and structure document, how do we train new team members about the system? We work closely with them. We transfer our knowledge to them by sitting next to them and helping them. We make them part of the team through close training and interaction.

   The two documents that are the best at transferring information to new team members are the code and the team. The code does not lie about what it does. It may be difficult to extract rationale and intent from the code, but the code is the only unambiguous source of information. The team holds the ever-changing roadmap of the system in its members' heads. The fastest and most efficient way to put that roadmap down on paper and transfer it to others is through human-to-human interaction.

   Many teams have gotten hung up in pursuit of documentation instead of software. This is often a fatal flaw. There is a simple rule that prevents it: ***Produce no document unless its need is immediate and significant.***

3. **Customer collaboration over contract negotiation**

Only your customer can tell you what they want. Yes, they likely do not have the skills to exactly specify the system. Yes, they likely won't get it right the first. Yes, they'll likely change their minds. Working together with your customers is hard, but that's the reality of the job. Having a contract with your customers is important, having an understanding of everyone's rights and responsibilities may form the foundation of that contract, but a contract isn't a substitute for communication. Successful developers work closely with their customers, they invest the effort to discover what their customers need, and they educate their customers along the way.

Software cannot be ordered like a commodity. You cannot write a description of the software you want and then have someone develop it on a fixed schedule for a fixed price. Time and time again, attempts to treat software projects in this manner have failed. Sometimes, the failures are spectacular.

It is tempting for company managers to tell their development staff what their needs are and then expect that staff to go away for a while and return with a system that satisfies those needs. But this mode of operation leads to poor quality and failure.

Successful projects involve customer feedback on a regular and frequent basis. Rather than depending on a contract, or a statement of work, the customer of the software works closely with the development team, providing frequent feedback on its efforts.

A contract that specifies the requirements, schedule, and cost of a project is fundamentally flawed. In most cases, the terms it specifies become meaningless long before the project is complete, sometimes even long before the contract is signed! The best contracts are those that govern the way the development team and the customer will work together.

4. Responding to change over following a plan

People change their priorities for a variety of reasons. As work progresses on your system your project stakeholder's understanding of the problem domain and of what you are building changes. The business environment changes. Technology changes over time, although not always for the better. Change is a reality of software development, a reality that your software process must reflect. There is nothing wrong with having a project plan; in fact, I would be worried about any project that didn't have one. However, a project plan must be malleable; there must be room to change it as your situation changes otherwise your plan quickly becomes irrelevant.

The ability to respond to change often determines the success or failure of a software project. When we build plans, we need to make sure that they are flexible and ready to adapt to changes in the business and technology.

The course of a software project cannot be planned very far into the future. First, the business environment is likely to change, causing the requirements to shift. Second, once they see the system start to function, customers are likely to alter the

requirements. Finally, even if we know what the requirements are and are sure that they won't change; we are not very good at estimating how long it will take to develop them.

It is tempting for novice managers to create and tape to the wall a nice PERT or Gantt chart of the whole project. They may feel that this chart gives them control over the project. They can track the individual tasks and cross them off the chart as they are completed. They can compare the actual dates with the planned dates on the chart and react to any discrepancies.

But what really happens is that the structure of the chart degrades. As the team gains knowledge about the system and as the customer gains knowledge about the team's needs, certain tasks on the chart will become unnecessary. Other tasks will be discovered and will need to be added. In short, the plan will undergo changes in shape, not only in dates.

A better planning strategy is to make detailed plans for the next week, rough plans for the next 3 months, and extremely crude plans beyond that. We should know the individual tasks we will be working on for the next week. We should roughly know the requirements we will be working on for the next 3 months. And we should have only a vague idea what the system will do after a year.

This decreasing resolution of the plan means that we are investing in a detailed plan only for those tasks that are immediate. Once the detailed plan is made, it is difficult to change, since the team will have a lot of momentum and commitment. But since that plan governs only a week's worth of time, the rest of the plan remains flexible.

The interesting thing about these value statements is that they are something that almost everyone will instantly agree to, yet will rarely adhere to in practice. Senior management will always claim that its employees are the most important aspect of your organization, yet insist they follow ISO-9000 compliant processes and treat their staff as replaceable assets. Even worse, management often refuses to provide sufficient resources to comply with the processes that they insist project teams follow. Everyone will readily agree that the creation of software is the fundamental goal of software development, yet insist on spending months producing documentation describing what the software is and how it is going to be built instead of simply rolling up their sleeves and building it. You get the idea – people say one thing and do another. This has to stop now. *Agile developers do what they say and say what they do.*

Agile Principles

To help people to gain a better understanding of what agile software development is all about, we can refine the philosophies captured in their manifesto into a collection of twelve principles. These principles are:

1. **Our highest priority is to satisfy the customer through early and continuous delivery of valuable software**. For some reason many people within IT have seem to

have forgotten that the goal of software development should be the development of software. Or, perhaps the problem is that they think that they need to define everything up front before they can start building software, whereas an evolutionary approach to development seems to work much better.

2. **Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage**. Like it or not, requirements will change throughout a software development project. Traditional software developers will often adopt change management processes which are designed to prevent/reduce scope creep, but when you think about it these are really change prevention processes, not change management processes. Agilists embrace change and instead follow an agile change management approach which treats requirements as a prioritized stack which is allowed to vary over time.

3. **Deliver working software frequently, from a week to a couple of months, with a preference to the shorter time scale**. Frequent delivery of working software provides stakeholders with concrete feedback, making the current status of your project transparent while at the same time providing an opportunity for stakeholders to provide improved direction for the development team.

4. **Business people and developers must work together daily throughout the project**. Your project is in serious trouble if you don't have regular access to your project stakeholders. Agile developers adopt practices such as on-site customer and active stakeholder participation, and adopt inclusive tools and techniques which enable stakeholders to be actively involved with software development.

5. **Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done**. Too many organizations have a vision that they can hire hordes of relatively unskilled people, provide them with a CMMI/ISO/...-compliant process description, and that they will successfully develop software. This doesn't seem to work all that well in practice. Agile teams, on the other hand, realize that you need build teams from people who are willing to work together collaboratively and learn from each other. They have the humility to respect one another and realize that people are a primary success factor in software development.

6. **The most efficient and effective method of conveying information to and within a development team is face-to-face conversation**. For software development team to succeed its members must communicate and collaborate effectively. There are many ways which people can communicate together, and as you can see face-to-face communication at a shared drawing environment (such as paper or a whiteboard) is the most effective way to do so.

7. **Working software is the primary measure of progress**. The primary measure of software development should be the delivery of working software which meets the changing needs of its stakeholders, not some form of "earned value" measure based on the delivery of documentation of the holding of meetings.

8. **Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely**. Just like you can't

sprint for an entire marathon, you can't successfully develop software by forcing people to work overtime for months at a time. My experience is that you can only do high-quality, intellectual work for 5-6 hours a day before burning yourself out. Yes, the rest of the day can be filled up with email, meetings, water cooler discussions, and so on, but people's ability to do "real work" is limited. Yes, you might be able to do high-quality work for 12 hours a day, and do so for a few days straight, but after a while you become exhausted and all you accomplish is 12 hours of mediocre work a day.

9. **Continuous attention to technical excellence and good design enhances agility**. It's much easier to understand, maintain, and evolve high-quality source code than it is to work with low-quality code. Therefore, Agilists know that they need to start with good code, to keep it good via refactoring, and take a test-driven approach so that they know at all times that their software works. We also adopt and follow development guidelines, in particular coding guidelines and sometimes even modeling guidelines.

10. **Simplicity – the art of maximizing the amount of work not done – is essential**. Agile developers focus on high value activities, we strive to maximize our stakeholder's return on investment in IT, and we either cut out or automate the drudge work.

11. **The best architectures, requirements, and designs emerge from self-organizing teams**. This is one of the most radical principles of the agile movement. The Agile Model Driven Development (AMDD) and test-driven design (TDD) methods are the primary approaches within the agile community for ensure the emergence of effective architectures, requirements, and designs.

12. **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly**. Software process improvement (SPI) is a continual effort, and techniques such as retrospectives should be adopted to enable you to improve your approach to software development.

Stop for a moment and think about these principles. Is this the way that your software projects actually work? Is this the way that you think projects should work?
Re-read the principles once again. Are they radical and impossible goals as some people would claim, are they meaningless motherhood and apple pie statements, or are they simply common sense? My belief is that these principles form a foundation of common sense upon which you can base successful software development efforts.

Agile vs. Defined Process

Agile Methods	Plan Driven Methods
Embraces scope change. Believes scope is not 100% known until you get into development.	Tries to predict what will happen with the entire project during planning. Scope change can cause delay.
Status of the project is determined by the demonstration of functioning code.	Status is determined by the percent of tasks completed.

Agile Methods	Plan Driven Methods
The team determines the level of documentation needed.	Methodology dictates documentation requirements.
Methods are people oriented.	Methods are process oriented.
Looks for a different kind of customer, more like a partner who participates in the cycle.	Customer view is frequently associated with fixed scope/price contract, which can make the customer seem like an adversary.
Believes the knowledge of the team is more important than rigid process.	Process must be followed.
Believes team members are responsible professionals.	Methods do not take team skill level into account.
Teams have huge influence on decisions.	Management makes the decisions.
Best with smaller teams.	Best with larger teams.
Primary objective – rapid value.	Primary objective – high assurance.

Whenever I discuss Agile processes with friends they frequently tell me about the issues with using it. I hear things like:

- Agile must be used across all processes or not at all.
- Agile leads to cowboy coding.
- Agile projects have poor quality because there is no documentation.

Scrum

Scrum is an iterative, incremental process for developing any product or managing any work. It produces a potentially shippable set of functionality at the end of every iteration. Its attributes are:

- Scrum is an agile process to manage and control development work.
- Scrum is a wrapper for existing engineering practices.
- Scrum is a team-based approach to iteratively, incrementally develop systems and products when requirements are rapidly changing
- Scrum is a process that controls the chaos of conflicting interests and needs.
- Scrum is a way to improve communications and maximize co-operation.
- Scrum is a way to detect and cause the removal of anything that gets in the way of developing and delivering products.
- Scrum is a way to maximize productivity.
- Scrum is scalable from single projects to entire organizations. Scrum has controlled and organized development and implementation for multiple interrelated products and projects with over a thousand developers and implementers.
- Scrum is a way for everyone to feel good about their job, their contributions, and that they have done the very best they possibly could.

Scrum is an empirical process. Rather than following outdated scripts, Scrum employs frequent inspection and adaptation to direct team activities toward a desired goal. The Sprint review inspection is particularly powerful because real functionality is being inspected. When they use Scrum, teams are empowered to find their own way through complex situations. This freedom, along with the creativity that results from it, is one of the core benefits of Scrum.

There are only three Scrum roles: the Product Owner, the Team, and the ScrumMaster. All management responsibilities in a project are divided among these three roles.

The product owner (PO) represents the customer(s), stakeholder(s) or funder(s) of the product.

The PO is a part the team delivering the product. The product owner is responsible for:

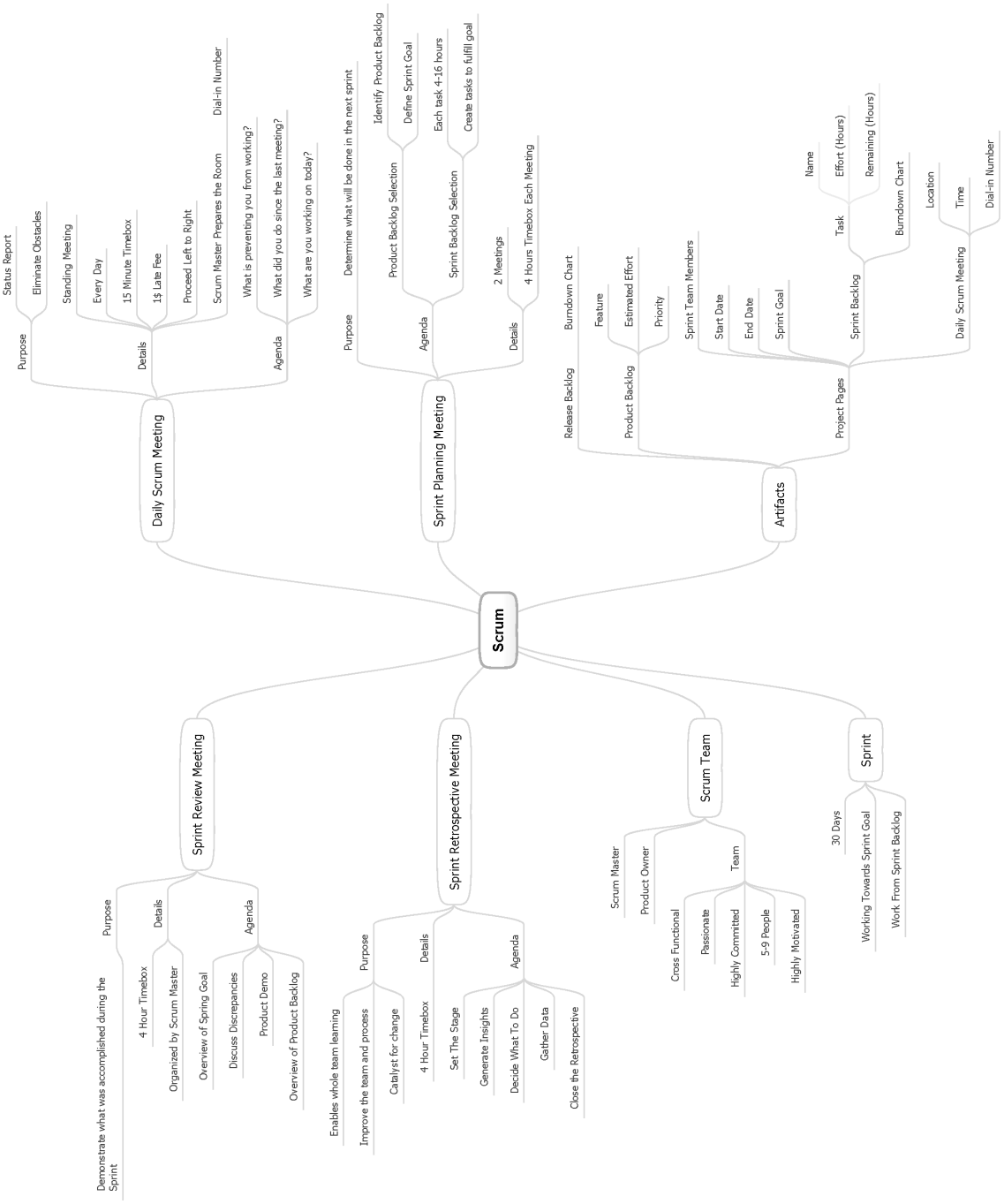
- setting the product vision
- managing the return on investment (ROI)
- present initial and ongoing product requirements to the team
- prioritize each requirement relative to business value
- manage new requirements and their prioritization
- release planning
- act as mediator when there are multiple customers
- ensure subject matter experts (SME's or business domain experts (BDE's) are available to the team

The ScrumMaster or project manager (PM) role differs from traditional command and control methods. The ScrumMaster in scrum works with and for the team. The ScrumMaster is responsible for:

- allowing the team to self-organize to get the work done
- ensure that communication paths are open and efficient
- ensure and help the team to follow the scrum process
- remove any impediments (issues) that the team encounter
- protect the team from outside interferences to ensure that their productivity is not affected
- facilitate the daily stand-up meeting

A team member is an individual that commits to getting the work done. Team members are cross-functional. What do we mean by that? In traditional methods we have distinct roles e.g. tester or architect. In a scrum team we have people with architecture skills, testing skills, however many individuals for example an architect could have sufficient secondary skill to help out in testing. These cross-functional skills help the team to get the work done. The team member is responsible for:

- setting the iteration goal
- committing to the work and doing it to the highest quality
- work to the product vision and iteration goal
- collaborating with team members and help the team to self-organize
- estimating of requirements and ensuring that the work effort remaining is up to date
- attending the daily stand-up meeting
- raise impediments


Scrum
Daily Scrum Meeting
Purpose
Status Report
Eliminate Obstacles
Details
Standing Meeting
Every Day
15 Minute Timebox
1$ Late Fee
Proceed Left to Right
Agenda
Scrum Master Prepares the Room
Dial-in Number
What is preventing you from working?
What did you do since the last meeting?
What are you working on today?
Sprint Planning Meeting
Purpose
Determine what will be done in the next sprint
Agenda
Product Backlog Selection
Identify Product Backlog
Define Sprint Goal
Sprint Backlog Selection
Each task 4-16 hours
Create tasks to fulfill goal
Details
2 Meetings
4 Hours Timebox Each Meeting
Artifacts
Release Backlog
Burndown Chart
Product Backlog
Feature
Estimated Effort
Priority
Project Pages
Sprint Team Members
Start Date
End Date
Sprint Goal
Sprint Backlog
Name
Task
Effort (Hours)
Remaining (Hours)
Burndown Chart
Daily Scrum Meeting
Location
Time
Dial-in Number
Sprint Review Meeting
Purpose
Demonstrate what was accomplished during the Sprint
Details
4 Hour Timebox
Organized by Scrum Master
Agenda
Overview of Spring Goal
Discuss Discrepancies
Product Demo
Overview of Product Backlog
Sprint Retrospective Meeting
Purpose
Enables whole team learning
Improve the team and process
Catalyst for change
Details
4 Hour Timebox
Set The Stage
Agenda
Generate Insights
Decide What To Do
Gather Data
Close the Retrospective
Scrum Team
Scrum Master
Product Owner
Team
Cross Functional
Passionate
Highly Committed
5-9 People
Highly Motivated
Sprint
30 Days
Working Towards Sprint Goal
Work From Sprint Backlog

Scrum offers many opportunities to make changes during the process and to respond to new opportunities. Scrum also keeps everything highly visible. The Product Backlog and its prioritization are open to everyone so that they can discuss them and come to the best way to optimize ROI. The Daily Scrum keeps all team activities visible so that the ScrumMaster can enforce the rules and help the team stay on track.
By keeping everything in full view, the type of backroom politicking and influence swapping normal in most organizations is minimized. These mechanisms are useful in bureaucratic organizations as a way to get particular things done. But when Scrum is already getting things done, these behind-the-scenes pressures are counterproductive.

The ScrumMaster often has to work against 20 years of history during which customers and Teams have drifted apart. Each sees the other as a source of something that is of great value but that is also extremely hard to get. The customer knows from experience how unlikely it is that the Team will deliver a system, much less a system that meets the customer's needs. The Team knows from experience that the customer is fickle; always changing his or her mind just when the Team thinks it knows what to build. Together, they believe there is little opportunity to work closely together to each other's mutual benefit.

Stakeholders tend to be skeptical when first told about Scrum. Customers have had so many "silver bullets" imposed on them that they can be forgiven for this, especially since each of the silver bullets involved more work for them but fewer results. A primary tool the ScrumMaster can use to improve customer involvement is the delivery of quick results that customers can potentially use in their organization. The Sprint planning and Sprint review meetings are the bookends to initiate and fulfill this expectation. If the Team is at all capable and the project is technologically feasible, the stakeholders and Product Owner can't wait to collaborate more with the Team.

After several Sprints, most managers are more than satisfied with the visibility Scrum provides them into the project and its progress. The trick is how to get over these first few Sprints. ScrumMasters have had to devise any number of ancillary reporting mechanisms to Scrum to do so. This could potentially be viewed as a weakness of Scrum. But keep in mind that Scrum represents a major shift in thinking and acting, and many people don't really understand Scrum until they have experienced it. In the meantime, these interim reports bridge the gap between when the first Sprint starts and when management feels comfortable with the visibility it has into the project through Sprint reviews and Scrum reports.

During the transition to Scrum, these ancillary and customized reports are necessary. You don't want Scrum rules to be broken and the team to be disrupted. On the other hand, you don't want management to withdraw or boil over. A ScrumMaster's job is the Scrum process and its adoption by the organization. They are responsible for figuring out and delivering these interim- reporting mechanisms whenever they are needed.

Scrum works only if everything is kept visible for frequent inspection and adaptation. To be empirical, everyone must know that about which they are inspecting. Practices such as the Sprint review meeting, the Daily Scrum, the Sprint Backlog, and the Product Backlog keep everything visible for inspection. Rules such as not being able to interrupt a team during a Sprint keep the adaptations from turning meaningful progress into floundering, as over adaptation overwhelms the project.
The ScrumMaster must keep everything visible at a meaningful level of detail
The Sprint Backlog is the team's Sprint plan.

A ScrumMaster must be vigilant. If the ScrumMaster is unclear about what's going on, so is everyone else. Make sure everything is visible.

Scrum's productivity stems from doing the right things first and doing those things very effectively. The Product Owner queues up the right work by prioritizing the Product Backlog. How does the Team maximize its productivity, though? Assuming that lines of code per day or function points per person-month are good productivity measurements, who tells the Team how to maximize them? In Scrum, the Team figures out how to maximize its productivity itself; the job of planning and executing the work belongs solely to the Team. The ScrumMaster and others can guide, advise, and inform the Team, but it is the Team's responsibility to manage itself.

At the heart of the solution is the Team working without interruption for the (say) 2-week Sprint. Having selected the Product Backlog for a Sprint, the Team has mutually committed to turning it into an increment of potentially shippable product increment in 14 calendar days. Once the Team makes this commitment, the clock starts ticking. The Sprint is a time-box within which the Team does whatever is necessary to meet its commitment. At the end of the Sprint, the Team demonstrates the working functionality to the Product Owner.

An increment is a potentially shippable product functionality. Each Sprint, the Team commits to turning selected Product Backlog into such an increment. For the functionality to be potentially shippable, it has to be clean. Clean code is not only having to be free from bugs, but must also adhere to coding standards, have been refactored to remove any duplicate or ill-structured code, contain no clever programming tricks, and be easy to read and understand. Code has to be all of these things for it to be sustainable and maintainable. If code isn't clean in all of these respects, developing functionality in future Sprints will take more and more time. The code will become more turgid, unreadable, and difficult to debug. Scrum requires transparency. When the Team demonstrates functionality to the Product Owner and stakeholders at the Sprint review, those viewing the functionality have a right to presume that the code is complete, meaning not only that the code is written but also that it is written according to standards, easy to read, refactored, unit tested, harness tested, and even functionality tested. If this isn't true, the Team isn't allowed to demonstrate the functionality, because in that case, the viewer's assumption would be incorrect.

Scrum requires complete transparency. Every day, the team has to synchronize its work so that it knows where it stands. Otherwise, team members might make incorrect assumptions about the completeness and adequacy of their work. Scrum relies on empirical process control, which in turn is based on frequent inspections and adaptation. If the Team couldn't inspect its status at least daily, how could it adapt to unforeseen change? How could it know that such change had even occurred? How could the team avoid the traditional death march of pulling everything together at the end of a development cycle— in this case, a Sprint—if it didn't pull everything together at least daily?

Many people love Scrum's frequent, regular delivery of working functionality, the high morale of the team members, the improved working conditions, and the excellent quality of the systems. But phrases such as "the art of the possible" drive them crazy when they see its implications. Some hit at the heart of the misuse of the word "estimate." I saw this misuse recently in a meeting, when a manager shouted at a developer, "How can I ever trust you when you never meet your estimates?" To estimate means to form an approximate judgment or opinion of the value of a measure, but that isn't the definition many managers use. All

people involved with Scrum need to understand that an estimate is not a promise. The job of the stakeholders is to accept the imprecision. The imprecision is worrisome, but it is inherent in the problem of software development.

People in software development teams are the same. When management tells them that it wants them to improve the accuracy of their estimates, what they hear is that management doesn't want any surprises. Some organizations try to improve estimates by first building databases of actual and estimated hours worked and then deriving statistics of the variances. For example, such statistics might show that a team worked 24 percent more hours than it estimated across four Sprints. Management naturally sees this as a problem. Management might then create a system of rewards if the team can reduce this imprecision. Management might tell the team that part of its performance review will depend on improving this variance to less than 20 percent. Once this target has been established, I guarantee that the team members will meet it because their salaries depend on improving this metric. Their success will cause management to view them favorably and perhaps promote them or give them more interesting work. Regardless, good things will come if the team members do what management wants. *The typical way that team members then improve estimating accuracy is to drop quality or to implement the functionality with less quality.* They might stop refactoring out duplicate code. They might not follow standards. They might implement a control that is less difficult but that isn't as user friendly. They might not rationalize the database. None of these actions are visible to management. All of these tricks are employed to meet the measurements and for the team members to do well in management's eyes.

If we tell a Team that we want it to improve the accuracy of its estimates, it will improve this metric regardless of the shortcuts it takes. Scrum asks management to focus on the overall delivery of functionality and eschew suboptimal measurements.

A friend of mine, thinking about the idea of collocated Team work areas, remarked that when he was a child and had been bad, his parents put him in a corner. He had to face away from everyone and be in isolation. He couldn't help but notice the parallel between this punishment for being bad and what we do to our most valuable assets, our employees. Scrum relies on high-bandwidth, face-to-face communication and teamwork; cubicles and unneeded artifacts promote isolation and misunderstandings. I recommended removing the cubicles and setting up collocated team spaces.

When people are asked to achieve the possible, they will often try. When people are asked to try to do a little more than the possible, they will continue to try if they aren't punished for not achieving everything. When people are given all the help they need and ask for, when people are encouraged and treasured, they almost always respond by doing their best.

When people work by themselves, they can achieve great things. When people work with others, they often achieve synergy, where the joint effort far exceeds the sum of the individual efforts. In my experience, this exponential increase in productivity continues until a Team reaches seven people, give or take two. At that point, the shared work, vision, and concepts start to require additional support, such as documentation. Regardless of the scaling mechanism, above a modest number like seven, the productivity of a Team starts to decline, the miscommunications increase, the mistakes proliferate, and frustration grows.

Deploy all the tools, technologies, and processes you like, even our processes, but in the end it's the people that are the biggest difference between success and failure. We realize that

however hard we work in coming up with process ideas, the best we can hope for is a second-order effect on your project. So it's important for you to maximize that first-order people factor.
The hardest thing for many people to give is trust. Decisions must be made by the people who know the most about the situation. For managers, this means trusting in your staff to make the decisions about the things they're paid to know about.
Unfortunately, most managers mistrust their own people.

I have the greatest respect for command & control Project Managers. I used to be one. Felt powerful. And "in charge." I also used to be a work-a-holic and had no work-life balance to speak of. But now I consider that style of management as childish.

Scrum Team

A Team is cross-functional: in situations where everyone is chipping in to build the functionality, you don't have to be a tester to test, or a designer to design.

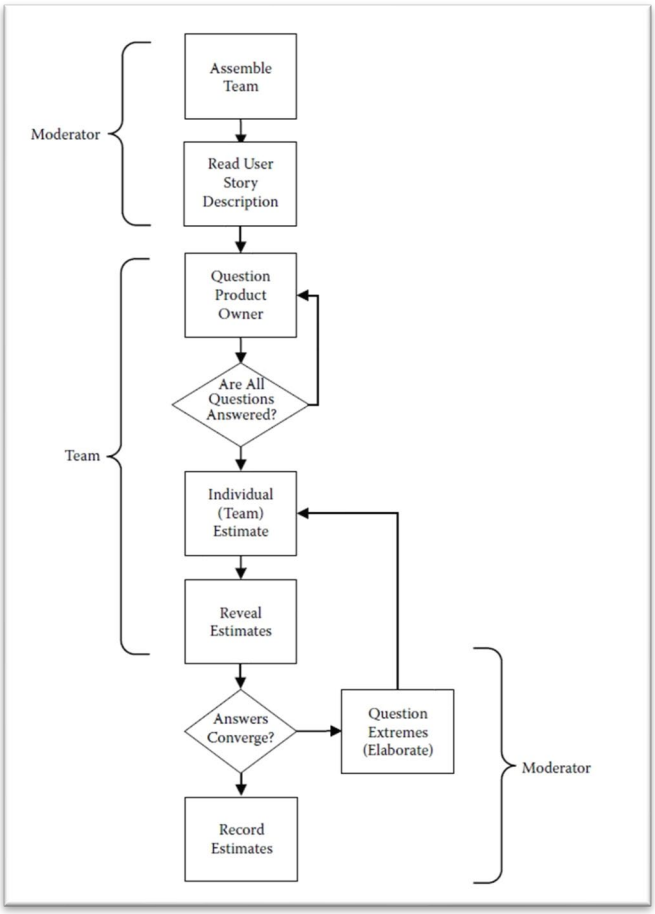
A Scrum team is self-organizing. It assumes responsibility for planning its own work. The Sprint Backlog is the visible manifestation of the team fulfilling this responsibility. Teams need to think through what they are doing and write it down so that team members can refer back to a plan as they work. The Daily Scrum synchronizes everyone's work only if the work has been thought through. Otherwise, the Daily Scrum is useless.

Self-organization and self- management are easy to grasp on an intellectual level, but they often prove difficult to implement. Such concepts are counterintuitive to the culture of many workplaces, and many teams veer off track.

As an organization transitions to Scrum, the Team bears the brunt of the change. Whereas before the project manager told the Team what to do, now the Team has to figure out what to do on its own. In the past, team members worked on their own, but now they work with each other.

Being managed by someone else is totally ingrained in our life and work experience. Parents, teachers, and bosses who teach us to self-manage instead of striving to fulfill their expectations are rare. Why should we expect that when we tell a Team that it is responsible for managing itself, it will know what we are talking about? "Self-management" is just a phrase to them; it isn't yet something real. A Team requires concrete experience with Scrum before it can truly understand how to manage itself and how to take the responsibility and authority for planning and conducting its own activities. Not only must the ScrumMaster help the Team to acquire this experience, but the ScrumMaster must also do so while overcoming his or her own tendencies to manage the Team. Both the ScrumMaster and the Team have to learn a new how to approach the issue of management.

For Scrum to work, the team has to deeply and viscerally understand collective commitment and self-organization. Scrum's theory, practices, and rules are easy to grasp intellectually. But until a group of individuals has made a collective commitment to deliver something tangible in a fixed amount of time, those individuals probably don't get Scrum. When the team members stop acting as many and adopt and commit to a common purpose, the team



becomes capable of self-organization and can quickly cut through complexity and produce actionable plans. At that point, the members of a team no longer accept obstacles, but instead scheme, plot, noodle, and brainstorm about how to remove them. They figure out how to make their project happen despite the different backgrounds and skill sets each person brings to the job.

The natural tendency of managers is to figure out how to do things right and tell the workers to do it that way; teams expect this. But in Scrum, the teams are responsible for their own management. The ScrumMasters were only there to act as advisors or to help the conversation along. Once teams realize this, they generally start looking for their own solutions to their problems.

This is one of the great truths of Scrum: constant inspection and adaptation are necessary for successful development. It is not the ScrumMaster's job to manage the Team. The Team has to learn to manage itself, to constantly adjust its methods in order to optimize its chances of success. The Sprint retrospective provides a time for such inspection and adaptation. As with many other Scrum practices, the Sprint retrospective is time-boxed to stop the Team from spending too much time searching for perfection, when no such thing exists in this complex, imperfect world.

Let the Team figure things out on its own. The ScrumMaster role ensures that this will happen, since the role includes no authority over the Team. The ScrumMaster is responsible for the process and removing impediments but is not responsible for managing the development of functionality. ScrumMasters can help by asking questions and providing advice, but within the guidelines, conventions, and standards of the organization, the Team is responsible for figuring out how to conduct its work. The ScrumMaster's job is to ensure that the Scrum practices are followed. Working together, the ScrumMaster and the Team shape the development process so that it will bring about the best possible results and won't let things get too far off track.

ScrumMaster

ScrumMasters have no authority over the development teams; they are present only to ensure that the Scrum process is adhered to and that the teams' needs are met.

The shift from project manager to ScrumMaster eludes many a traditional project manager. These people believe that Scrum was merely a series of practices and techniques for

implementing iterative, incremental development. They miss the subtle but critical shift from controlling to facilitating, from bossing to coaching. They also miss out on the importance of a self-organizing team.
They and the team commit to a Sprint goal, but the team never self- organized or truly committed to the Scrum goal. The productivity that emerges when a team figures out the best way to accomplish its goals isn't realized. Neither do team members have the deep personal commitment that emerges when people puzzle their way through their work on their own. The team's ability to tackle its problems and solve them is the heart of Scrum and the basis of the Scrum team's extraordinary productivity. Some people are so embedded in their familiar ways that they have trouble seeing what they have to change, no matter how many training sessions they undergo and books they read.

The behavior of the ScrumMaster is dramatically different from that of people staffing a formal project management office that assigns work and controls its completion. The shift from having authority to being a facilitator is too much for some traditional managers. Not only is the ScrumMaster role one without authority, but it also potentially represented a career change that some traditional managers don't want to make. The ScrumMaster is a leader, not a manager. The ScrumMaster earns the team's respect because he or she fulfills the duties of the role and not simply because he or she was assigned the role in the first place.

ScrumMasters have to make a personal commitment to their teams. A ScrumMaster would no more delegate his responsibilities than a sheepdog would lie down for a nap while herding the flock. The team needs to sense that someone is deeply invested in its work and will protect and help it no matter what. The ScrumMaster's attitude should reflect the importance of the project. If anything else is more important for the ScrumMaster, that ScrumMaster should be replaced.

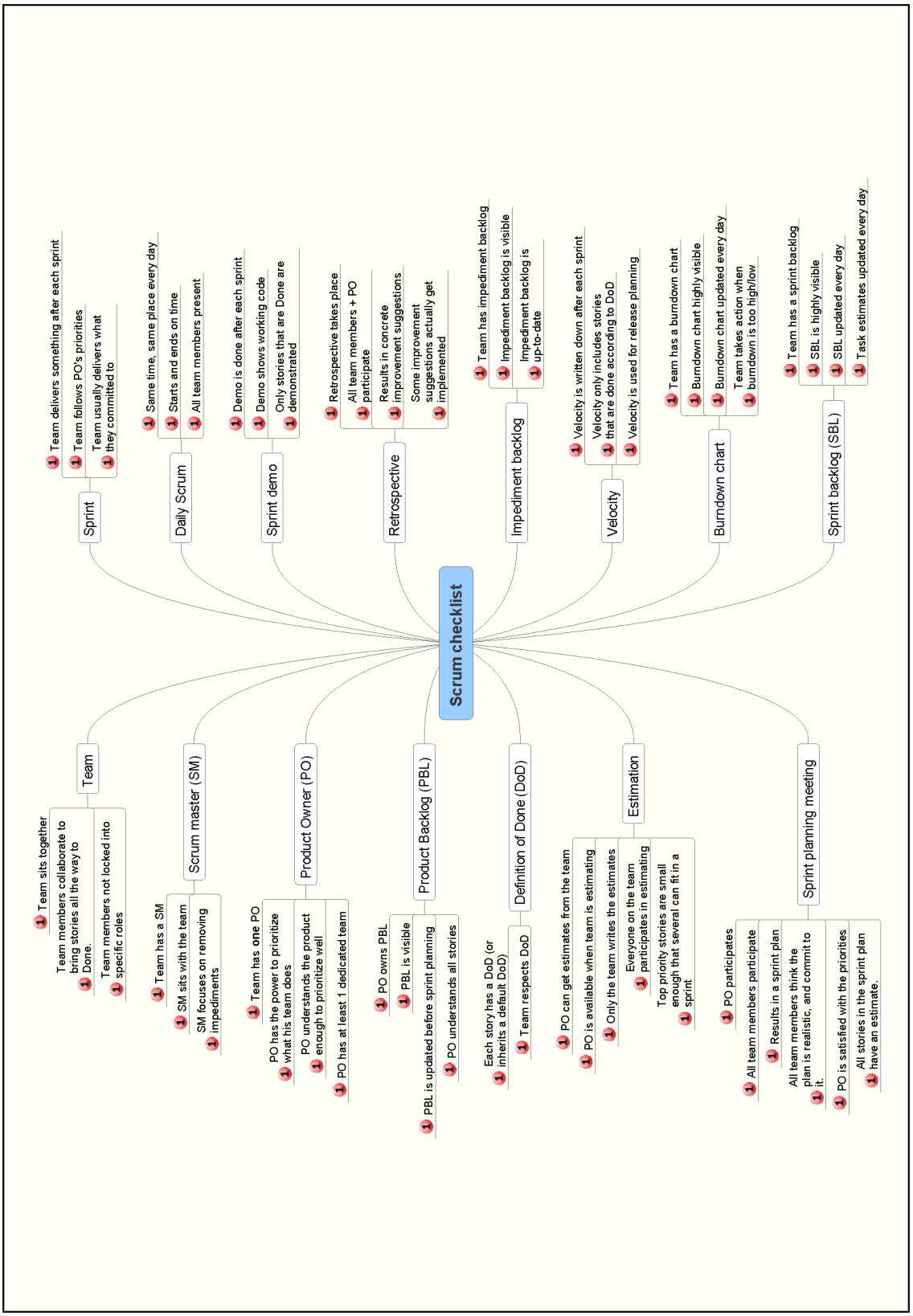
The shift from delegating to being personally responsible is difficult for some people to accept. The "hands-on" aspect of Scrum scares some people. These people are unfit to become ScrumMasters.

The responsibilities of the ScrumMasters can be summarized as follows:

- Remove the barriers between development and the Product Owner so that the Product Owner directly drives development.
- Teach the Product Owner how to maximize ROI and meet his or her objectives through Scrum.
- Improve the lives of the development team by facilitating creativity and empowerment.
- Improve the productivity of the development team in any way possible.
- Improve the engineering practices and tools so that each increment of functionality is potentially shippable.
- Keep information about the team's progress up-to-date and visible to all parties.

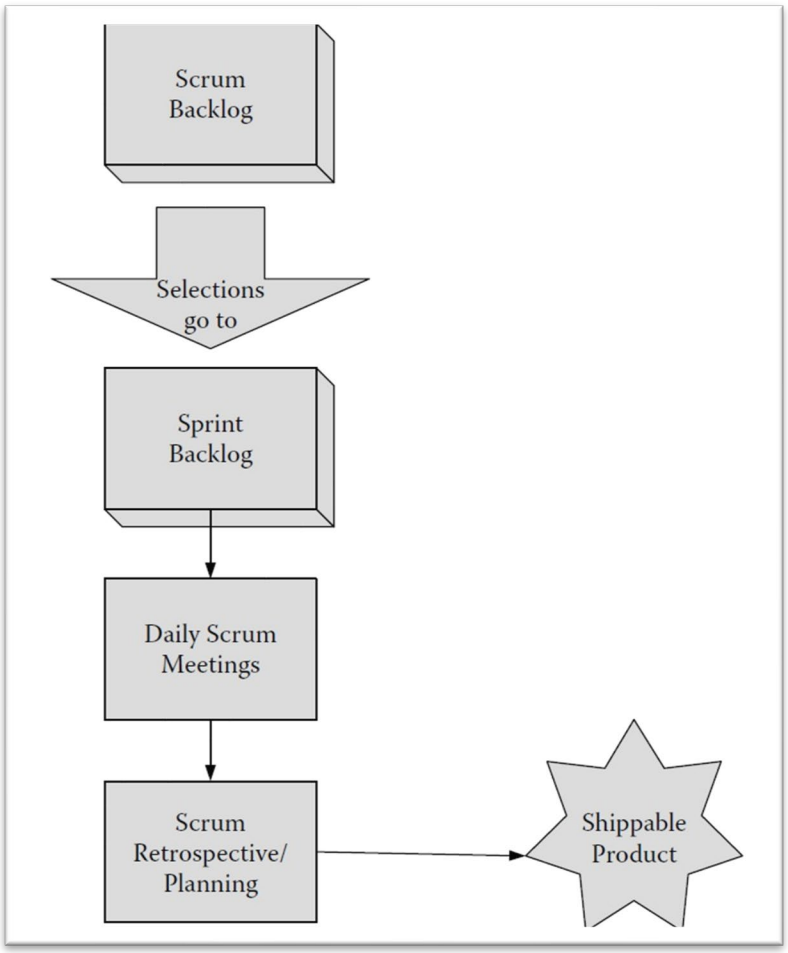
When the ScrumMaster fulfills these responsibilities, the project usually stays on track. These responsibilities should be enough to keep the ScrumMaster busy; no ScrumMaster should have any time left over to act like a typical boss. Indeed, a ScrumMaster who acts like a program manager probably isn't fulfilling all of his or her duties as a ScrumMaster.

The ScrumMaster has to teach, enforce, and reinforce the rule of sashimi. Sometimes teams try to cut corners. Sometimes teams are so used to waterfall development processes that they view testing as someone else's problem. The mechanism for detecting whether the team is doing all necessary work is the Sprint Backlog.


Scrum checklist
Team
Team sits together
Team members collaborate to bring stories all the way to Done.
Team members not locked into specific roles
Sprint
Team delivers something after each sprint
Team follows PO's priorities
Team usually delivers what they committed to
Scrum master (SM)
Team has a SM
SM sits with the team
SM focuses on removing impediments
Daily Scrum
Same time, same place every day
Starts and ends on time
All team members present
Product Owner (PO)
Team has one PO
PO has the power to prioritize what his team does
PO understands the product enough to prioritize well
PO has at least 1 dedicated team
Sprint demo
Demo is done after each sprint
Demo shows working code
Only stories that are Done are demonstrated
Product Backlog (PBL)
PO owns PBL
PBL is visible
PBL is updated before sprint planning
PO understands all stories
Retrospective
Retrospective takes place
All team members + PO participate
Results in concrete improvement suggestions
Some improvement suggestions actually get implemented
Definition of Done (DoD)
Each story has a DoD (or inherits a default DoD)
Team respects DoD
Impediment backlog
Team has impediment backlog
Impediment backlog is visible
Impediment backlog is up-to-date
Estimation
PO can get estimates from the team
PO is available when team is estimating
Only the team writes the estimates
Everyone on the team participates in estimating
Top priority stories are small enough that several can fit in a sprint
Velocity
Velocity is written down after each sprint
Velocity only includes stories that are done according to DoD
Velocity is used for release planning
Burndown chart
Team has a burndown chart
Burndown chart highly visible
Burndown chart updated every day
Team takes action when burndown is too high/low
Sprint planning meeting
PO participates
All team members participate
Results in a sprint plan
All team members think the plan is realistic, and commit to it.
PO is satisfied with the priorities
All stories in the sprint plan have an estimate.
Sprint backlog (SBL)
Team has a sprint backlog
SBL is highly visible
SBL updated every day
Task estimates updated every day

Rules of Scrum

The ScrumMaster is responsible for ensuring that everyone related to a project, whether chickens or pigs, follows the rules of Scrum. These rules hold the Scrum process together so that everyone knows how to play. If the rules aren't enforced, people waste time figuring out what to do. If the rules are disputed, time is lost while everyone waits for a resolution. These rules have worked in literally thousands of successful projects. If someone wants to change the rules, use the Sprint retrospective meeting as a forum for discussion. *Rule changes should originate from the Team, not management.* Rule changes should be entertained if and only if the ScrumMaster is convinced that the Team and everyone involved understands how Scrum works in enough depth that they will be skillful and mindful in changing the rules. No rules can be changed until the ScrumMaster has determined that this state has been reached.



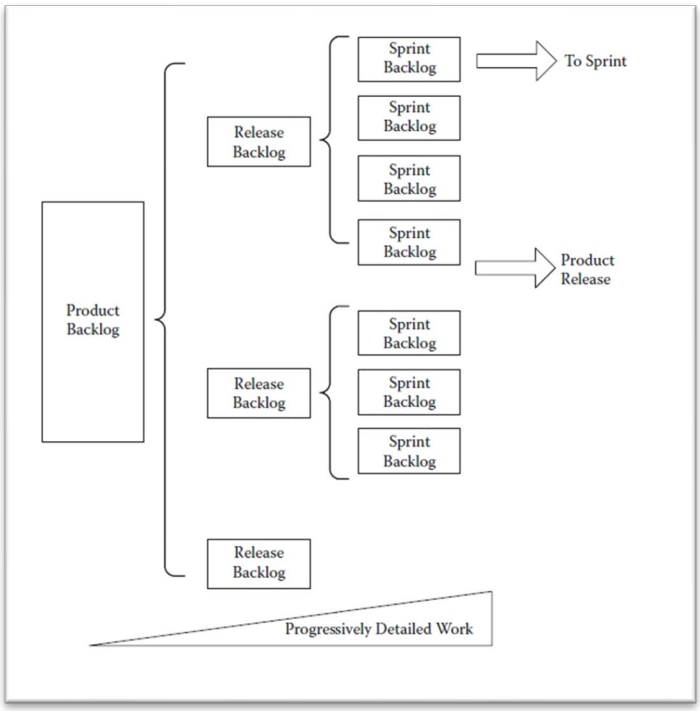
Sprint Planning Meeting

The Sprint planning meeting is time-boxed to 8 hours and consists of two segments that are time-boxed to 4 hours each. The first segment is for selecting Product Backlog; the second segment is for preparing a Sprint Backlog.

- The attendees are the ScrumMaster, the Product Owner, and the Team.
- The Product Owner must prepare the Product Backlog prior to the meeting.
- The goal of the first segment, or first 4 hours, is for the Team to select those Product Backlog items that it believes it can commit to turning into an increment of potentially

shippable product functionality. The Team will demonstrate this functionality to the Product Owner and stakeholders at the Sprint review meeting at the end of the Sprint.

- The Team can make suggestions, but the decision of what Product Backlog can constitute the Sprint is the responsibility of the Product Owner.
- The Team is responsible for determining how much of the Product Backlog that the Product Owner wants worked on the Team will attempt to do during the Sprint.
- Time-boxing the first segment to 4 hours means that this is all of the time that is available for analyzing the Product Backlog. Further analysis must be performed during the Sprint.
- The second segment of the Sprint Planning meeting occurs immediately after the first segment and is also time-boxed to 4 hours.
- The Product Owner must be available to the Team during the second segment to answer questions that the Team might have about the Product Backlog.
- It is up to the Team, acting solely on its own and without any direction from outside the Team, to figure out during the second segment how it will turn the selected Product Backlog into an increment of potentially shippable product functionality. No one else is allowed to do anything but observe or answer questions seeking further information.
- The output of the second segment of the Sprint planning meeting is a list, called the Sprint Backlog, of tasks, task estimates, and assignments that will start the Team on the work of developing the functionality. The task list might not be complete, but it must be complete enough to reflect mutual commitment on the part of all Team members and to carry them through the first part of the Sprint, while the Team devises more tasks in the Sprint Backlog.



Daily Scrum Meeting

The Daily Scrum meeting is time-boxed to 15 minutes regardless of the number of Team members.

- Hold the Daily Scrum in the same place at the same time every work day. The Daily Scrum is best held in the mornings.
- All Team members are required to attend. If for some reason a Team member can't attend in person, the absent member must either attend by telephone or by having another Team member report on the absent member's status.
- Team members must be prompt. The ScrumMaster starts the meeting at the appointed time, regardless of who is present.
- Each Team member should respond to three questions only:
    - What have you done since the last Daily Scrum regarding this project?
    - What will you do between now and the next Daily Scrum meeting regarding this project?
    - What impedes you from performing your work as effectively as possible?
- Team members should not digress beyond answering these three questions into issues, designs, discussion of problems, or gossip. The ScrumMaster is responsible for moving the reporting along briskly, from person to person.
- During the Daily Scrum, only one person talks at a time. That person is the one who is reporting his or her status. Everyone else listens. There are no side conversations.
- When a Team member reports something that is of interest to other Team members or needs the assistance of other Team members, any Team member can immediately arrange for all interested parties to get together after the Daily Scrum to set up a meeting.
- Chickens are not allowed to talk, make observations, make faces, or otherwise make their presence in the Daily Scrum meeting obtrusive.
- Chickens stand on the periphery of the Team so as not to interfere with the meeting.
- Chickens are not allowed to talk with Team members after the meeting for clarification or to provide advice or instructions.
- Pigs or chickens who cannot or will not conform to the above rules can be excluded from the meeting (chickens) or removed from the Team (pigs).

Sprint

The Sprint is time-boxed to (say) 15 consecutive calendar days.

- The Team can seek outside advice, help, information, and support during the Sprint.
- No one can provide advice, instructions, commentary, or direction to the Team during the Sprint. The Team is utterly self-managing.
- The Team commits to Product Backlog during the Sprint planning meeting. No one is allowed to change this Product Backlog during the Sprint. The Product Backlog is frozen until the end of the Sprint.
- If the Sprint proves to be not viable, the ScrumMaster can abnormally terminate the Sprint and initiate a new Sprint planning meeting to initiate the next Sprint. The ScrumMaster can make this change of his or her own accord or as requested by the Team or the Product Owner. The Sprint can prove to be not viable if the technology proves unworkable, if the business conditions change so that the Sprint will not be of value to the business, or if the Team is interfered with during the Sprint by anyone outside the Team.
- If the Team feels itself unable to complete all of the committed Product Backlog during the Sprint, it can consult with the Product Owner on which items to remove from the current Sprint. If so many items require removal that the Sprint has lost its

value and meaning, the ScrumMaster can abnormally terminate the Sprint, as previously stated.
- If the Team determines that it can address more Product Backlog during the Sprint than it selected during the Sprint planning meeting, it can consult with the Product Owner on which additional Product Backlog items can be added to the Sprint.
- The Team members have two administrative responsibilities during the Sprint: they are to attend the Daily Scrum meeting, and they are to keep the Sprint Backlog up-to-date and available in a public folder on a public server, visible to all. New tasks must be added to the Sprint Backlog as they are conceived, and the running, day-to-day estimated hours remaining for each task must be kept up-to-date.

Sprint Review Meeting

The Sprint review meeting is time-boxed to 4 hours.

- The Team should not spend more than 1 hour preparing for the Sprint review.
- The purpose of the Sprint review is for the Team to present to the Product Owner and stakeholders functionality that is done. Although the meaning of "done" can vary from organization to organization, it usually means that the functionality is completely engineered and could be potentially shipped or implemented. If "done" has another meaning, make sure that the Product Owner and stakeholders understand it.
- Functionality that isn't "done" cannot be presented.
- Artifacts that aren't functionality cannot be presented except when used in support of understanding the demonstrated functionality. Artifacts cannot be shown as work products, and their use must be minimized to avoid confusing stakeholders or requiring them to understand how systems development works.
- Functionality should be presented on the Team member workstations and executed from the server closest to production—usually a quality assurance (QA) environment server.
- The Sprint review starts with a Team member presenting the Sprint goal, the Product Backlog committed to, and the Product Backlog completed. Different Team members can then discuss what went well and what didn't go well in the Sprint.
- The majority of the Sprint review is spent with Team members presenting functionality, answering stakeholder questions regarding the presentation, and noting changes that are desired.
- At the end of the presentations, the stakeholders are polled, one by one, to get their impressions, any desired changes, and the priority of these changes.
- The Product Owner discusses with the stakeholders and the Team potential rearrangement of the Product Backlog based on the feedback.
- Stakeholders are free to voice any comments, observations, or criticisms regarding the increment of potentially shippable product functionality between presentations.
- Stakeholders can identify functionality that wasn't delivered or wasn't delivered as expected and request that such functionality be placed in the Product Backlog for prioritization.
- Stakeholders can identify any new functionality that occurs to them as they view the presentation and request that the functionality be added to the Product Backlog for prioritization.
- The ScrumMaster should attempt to determine the number of people who expect to attend the Sprint review meeting and set up the meeting to accommodate them.

- At the end of the Sprint review, the ScrumMaster announces the place and date of the next Sprint review to the Product Owner and all stakeholders.

Sprint Retrospective Meeting

The Sprint retrospective meeting is time-boxed to 3 hours.

- It is attended only by the Team, the ScrumMaster, and the Product Owner. The Product Owner is optional.
- Start the meeting by having all Team members answer two questions:
  - What went well during the last Sprint?
  - What could be improved in the next Sprint?
- The ScrumMaster writes down the Team's answers in summary form.
- The Team prioritizes in which order it wants to talk about the potential improvements.
- The ScrumMaster is not at this meeting to provide answers, but to facilitate the Team's search for better ways for the Scrum process to work for it.
- Actionable items that can be added to the next Sprint should be devised as high-priority nonfunctional Product Backlog. Retrospectives that don't result in change are sterile and frustrating.

Scrum Definitions

Item	Definition
Burndown graph	The trend of work remaining across time in a Sprint, a release, or a product. The source of the raw data is the Sprint Backlog and the Product Backlog, with work remaining tracked on the vertical axis and the time periods (days of a Sprint or Sprints) tracked on the horizontal axis.
Chicken	Someone who is interested in the project but does not have formal Scrum responsibilities and accountabilities (is not a Team member, Product Owner, ScrumMaster, or another stakeholder).
Daily Scrum meeting	A short status meeting held daily by each Team during which the Team members synchronize their work and progress and report any impediments to the ScrumMaster for removal.
Done	Coded, Tested, Checked-in, Automated tests exists for Unit testing and functional testing, all automated tests have been checked in and are running, Functionality has been shown to or discussed with the QA and Product owner.
Estimated work remaining	The number of hours that a Team member estimates remain to be worked on any task. This estimate is updated at the end of every day the Sprint Backlog task is worked on. The estimate is the total estimated hours remaining, regardless of the number of people that perform the work.
Increment	Product functionality that is developed by the Team during each Sprint.

Item	Definition
Increment of potentially shippable product functionality	A completely developed increment that contains all of the parts of a completed product, except for the Product Backlog items that the Team selected for this Sprint.
Iteration	One cycle within a project. In Scrum, this cycle is of say 15 sequential calendar days, or a Sprint.
Pig	Someone occupying one of the three Scrum roles (Team, Product Owner, ScrumMaster) who has made a commitment and has the authority to fulfill it.
Product Backlog	A prioritized list of project requirements with estimated times to turn them into completed product functionality. Estimates are in days and are more precise the higher an item is in the Product Backlog priority. The list evolves, changing as business conditions or technology changes.
Product Backlog items	Functional requirements, nonfunctional requirements, and issues, which are prioritized in order of importance to the business and dependencies and then estimated. The precision of the estimate depends on the priority and granularity of the Product Backlog item, with the highest priority items that can be selected in the next Sprint being very granular and precise.
Product Owner	The person responsible for managing the Product Backlog so as to maximize the value of the project. The Product Owner represents all stakeholders in the project.
Scrum	Not an acronym, but mechanisms in the game of rugby for getting an out-of-play ball back into play.
ScrumMaster	The person responsible for the Scrum process, its correct implementation, and the maximization of its benefits.
Sprint	A time-box of say 15 sequential calendar days during which a Team works to turn the Product Backlog it has selected into an increment of potentially shippable product functionality.
Sprint Backlog	A list of tasks that defines a Team's work for a Sprint. The list emerges during the Sprint. Each task identifies those responsible for doing the work and the estimated amount of work remaining on the task on any given day during the Sprint.
Sprint Backlog task	One of the tasks that the Team or a Team member defines as required to turn committed Product Backlog items into system functionality.
Sprint planning meeting	A one-day meeting time-boxed to 8 hours that initiates every Sprint. The meeting is divided into two 4-hour segments, each also time-boxed. During the first segment, the Product Owner presents the highest priority Product Backlog to the Team. The Team and the Product Owner collaborate to help the Team determine how much Product Backlog it can turn into functionality during the upcoming Sprint. The Team commits to this Product Backlog at the end of the first segment. During the second segment of the meeting, the Team plans how it will meet this commitment by detailing its work as a plan in the Sprint Backlog.

Item	Definition
Sprint retrospective meeting	A meeting time-boxed to 3 hours and facilitated by the ScrumMaster at which the Team discusses the just-concluded Sprint and determines what could be changed that might make the next Sprint more enjoyable or productive.
Sprint review meeting	A meeting time-boxed to 4 hours at the end of every Sprint at which the Team demonstrates to the Product Owner and any other interested parties what it was able to accomplish during the Sprint. Only completed product functionality can be demonstrated.
Stakeholder	Someone with an interest in the outcome of a project, either because he or she has funded it, will use it, or will be affected by it.
Team	A cross-functional group of people that is responsible for managing itself to develop software every Sprint.
Time-box	A period of time that cannot be exceeded and within which an event or meeting occurs. For example, a Daily Scrum meeting is time-boxed to 15 minutes and terminates at the end of those 15 minutes, regardless.

Introduction to Extreme Programming

The goal of Extreme Programming (XP) is outstanding software development. Software can be developed at lower cost, with fewer defects, with higher productivity, and with much higher return on investment. The same teams that are struggling today can achieve these results by careful attention to and refinement of how they work, by pushing ordinary development practices to the extreme.

Prepare for success. Don't protect yourself from success by holding back. Do your best and then deal with the consequences. That's extreme. You leave yourself exposed. For some people that is incredibly scary, for others it's daily life. That is why there are such polarized reactions to XP.

XP is a style of software development focusing on excellent application of programming techniques, clear communication, and teamwork which allows us to accomplish things we previously could not even imagine. XP includes:

- A philosophy of software development based on the values of communication, feedback, simplicity, courage, and respect.
- A body of practices proven useful in improving software development. The practices complement each other, amplifying their effects. They are chosen as expressions of the values.
- A set of complementary principles, intellectual techniques for translating the values into practice, useful when there isn't a practice handy for your particular problem.
- A community that shares these values and many of the same practices.

XP can be described this way:

- XP is lightweight. In XP you only do what you need to do to create value for the customer. You can't carry a lot of baggage and move fast.
- XP is a methodology based on addressing constraints in software development.
- XP can work with teams of any size. XP has had success with both large and small projects and teams. The values and principles behind XP are applicable at any scale. The practices need to be augmented and altered when many people are involved.
- XP adapts to vague or rapidly changing requirements. XP is still good for this situation, which is fortunate because requirements need to change to adapt to rapid shifts in the modern business world. However, teams have also successfully used XP where requirements don't seem volatile, like porting projects.

You may have enough time, money, or skills on your team or you may not; but it is always best to act as if there is going to be enough. This "mentality of sufficiency" is movingly documented by anthropologist Colin Turnbull in The Mountain People and The Forest People. He contrasts two societies: a resource-starved tribe of lying, cheating backstabbers and a resource-rich, cooperative, loving tribe. I often ask developers in a dilemma, "How would you do it if you had enough time?" You can do your best work even when there are constraints. Fussing about the constraints distracts you from your goals. Your clear self does the best work no matter what the constraints are.

Software development is capable of much, much more than it is currently delivering. Defects should be rare. Initial deployment of software should come after a small percentage of the project budget is spent. Teams should be able to grow and shrink without catastrophic consequences. XP is a way to get to that place. When we work with human nature in our development process, we have the opportunity to make these big leaps in effectiveness.

No book of gardening, however complete, makes you a gardener. First you have to garden, then join the community of gardeners, then teach others to garden. Then you are a gardener. So it is with XP. Reading these pages won't make you an extreme programmer. That only comes with programming in the extreme style, participating in the community of people who share these values and at least some of your practices, and then sharing what you know with others.

XP Practices

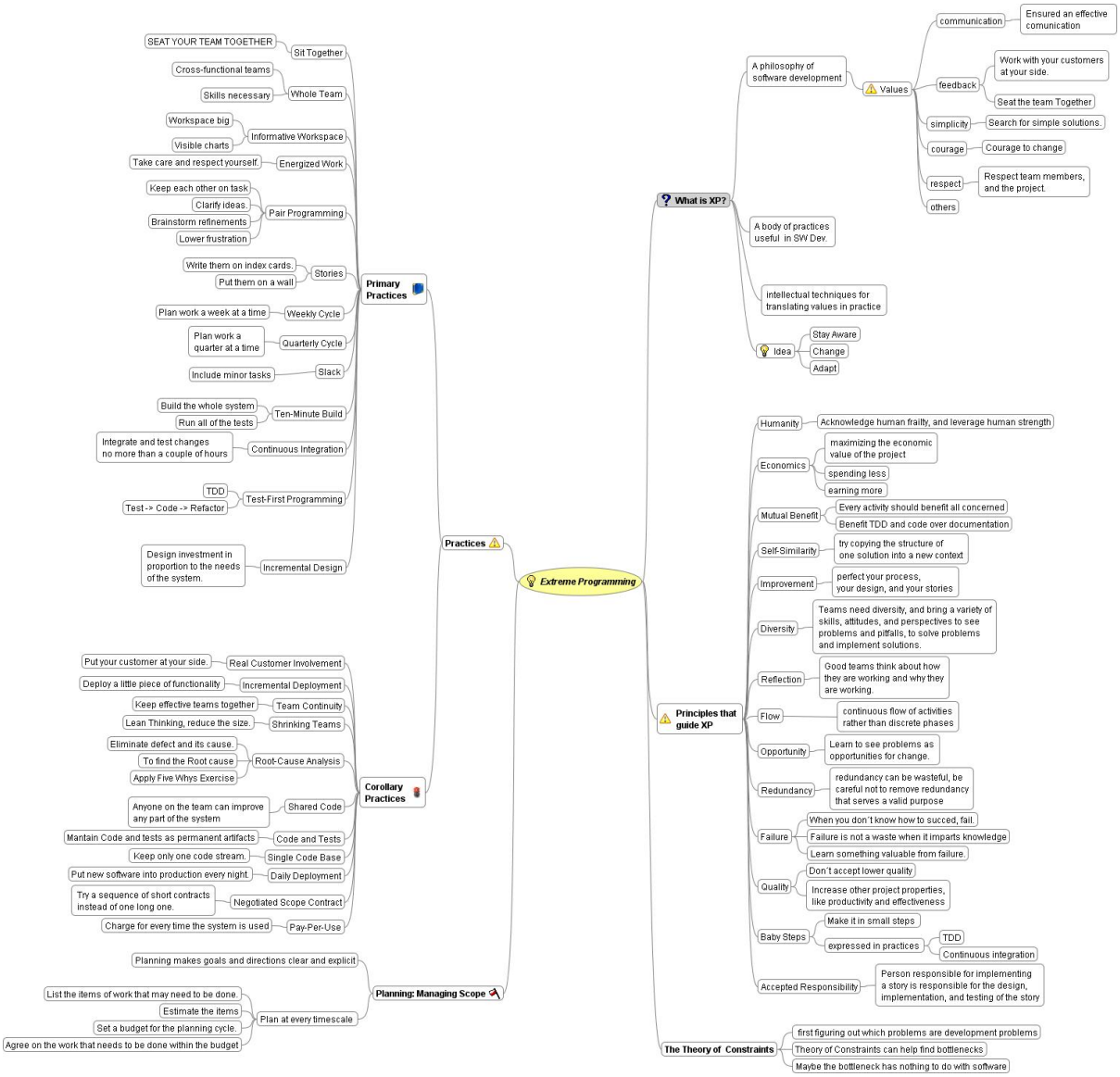
The practices of XP are the kind of things you'll see XP teams doing day-to-day. Practices by themselves are barren. Unless given purpose by values, they become rote. Pair programming, for example, makes no sense as a "thing to do to check off a box". Pairing to please your boss is just frustrating. Pair programming to communicate, get feedback, simplify the system, catch errors, and bolster your courage makes a lot of sense.

If you find yourself with a problem not covered by one of the practices, that's the time to look back at the XP values and the XP principles to come up with an appropriate solution for your team.

XP Principles

Principles are the link between XP values and XP practices.

You can use the principles to understand the practices better and to improvise complementary practices when you don't find one that suits your purpose. The principles give you a better idea of what the practice is intended to accomplish.

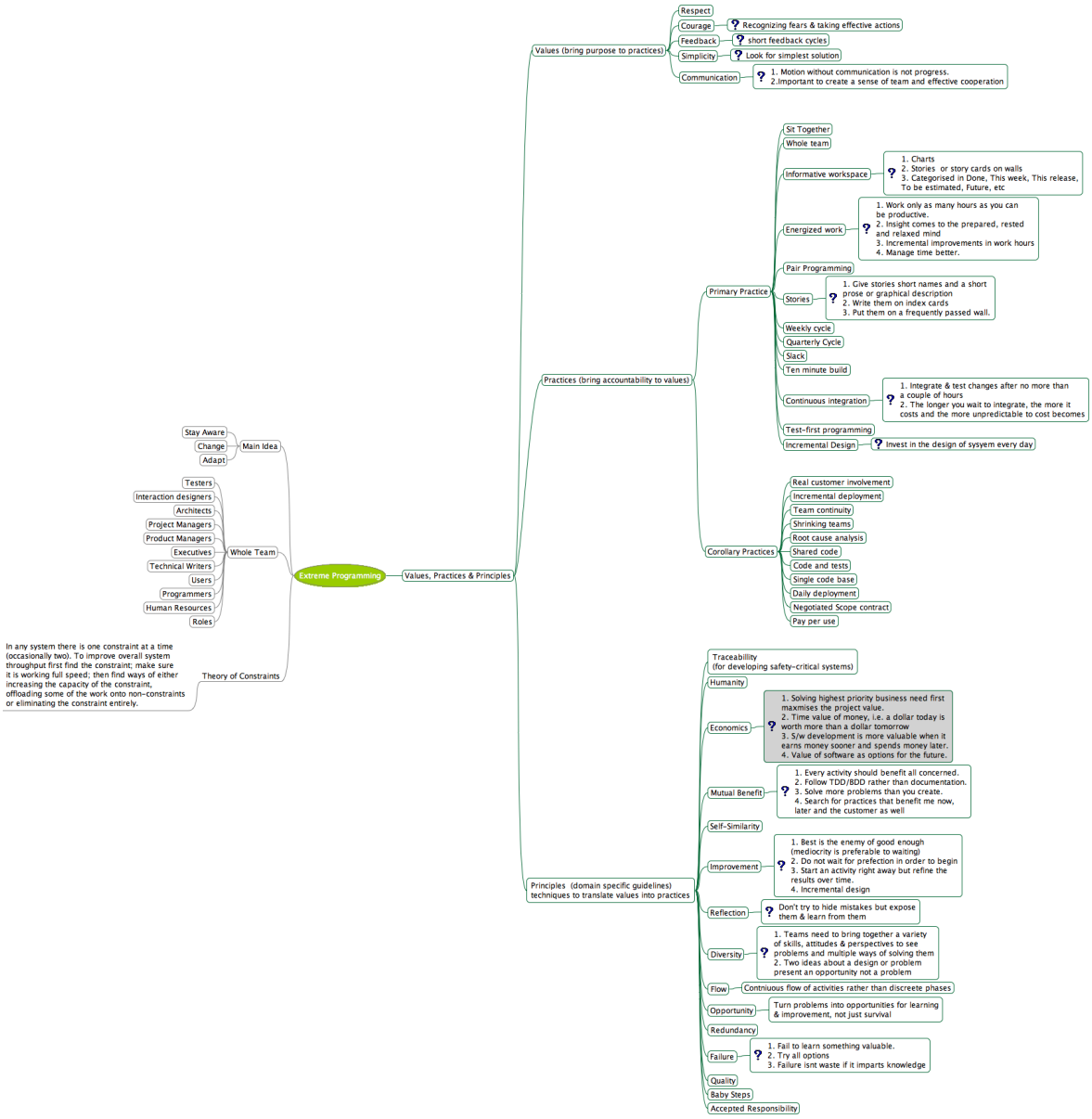


XP Values

Values bring purpose to the practices. Practices are the evidence of the values.

As Will Rogers said, "It isn't what you don't know that gets you in trouble. It's what you know that isn't so." The biggest problem I encounter in what people "just know" about software development is that they are focused on individual action. What actually matters is not how any given person behaves as much as how the individuals behave as part of a team and as part of an organization.

If everyone on the team chooses to focus on what's important to the team, what is it they should focus on? XP embraces five values to guide development: communication, simplicity, feedback, courage, and respect.



Roles in XP

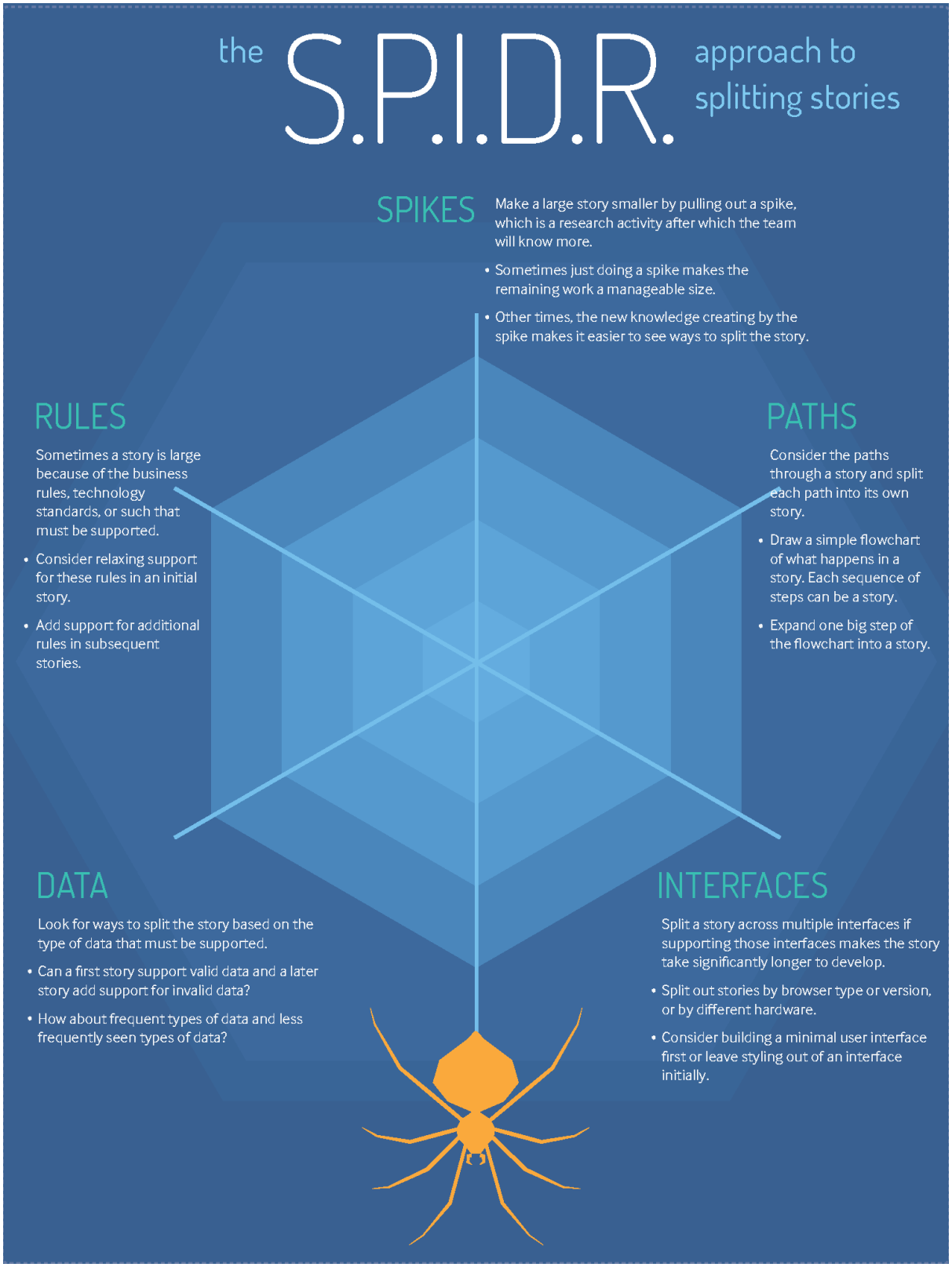
Roles on a mature XP team aren't fixed and rigid. The goal is to have everyone contribute the best he has to offer to the team's success. At first, fixed roles can help in learning new habits, like having technical people make technical decisions and business people make business decisions. After new, mutually respectful relationships are established among the team members, fixed roles interfere with the goal of having everyone do his best. Programmers can write a story if they are in the best position to write the story. Project managers can suggest architectural improvements if they are in the best position to suggest architectural improvements.

Architects on an XP team look for and execute large-scale refactorings, write system-level tests that stress the architecture, and implement stories. Architects apply their expertise a little bit at a time throughout the project. They direct the architecture of the project throughout its evolution. The architecture for a little system should not be the same as for a big system.

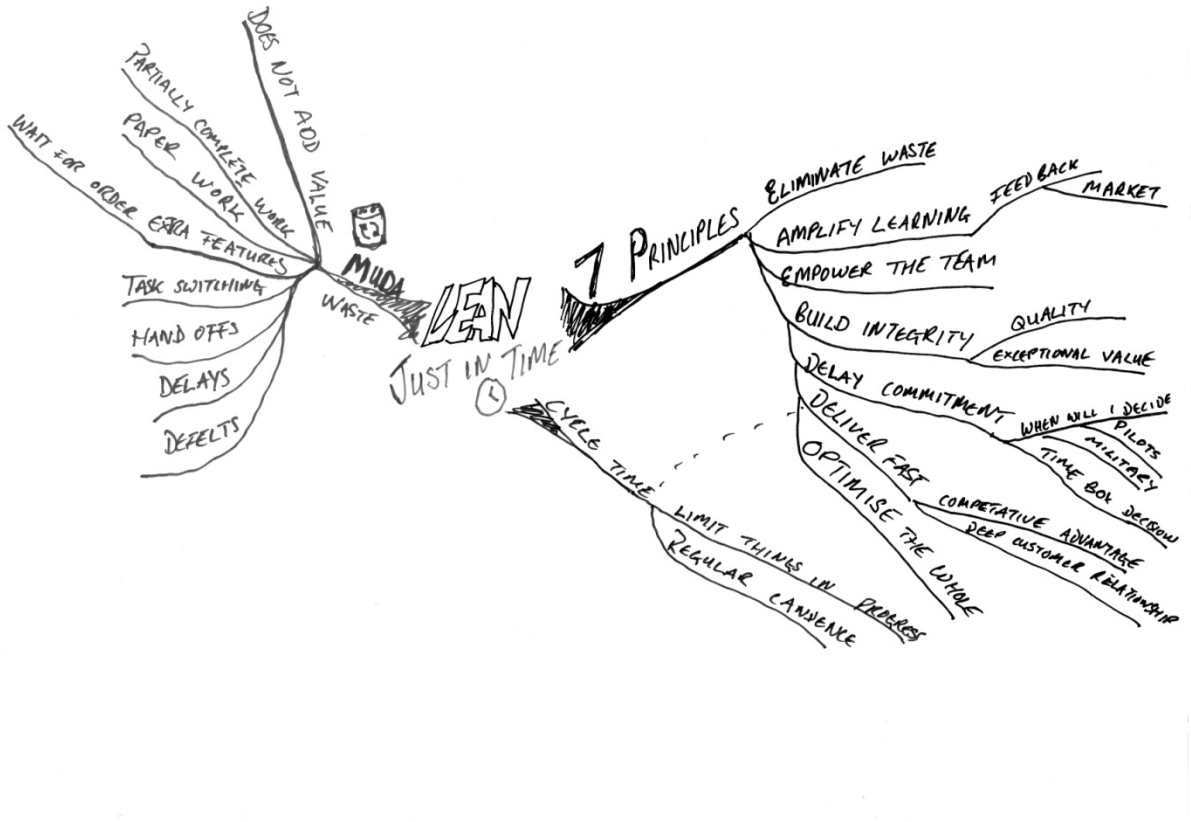
While the system is little the architect makes sure the system has just the right little architecture. As the system grows, the architect makes sure the architecture keeps pace.

Making big architectural changes in small, safe steps is one of the challenges for an XP team. The principle of the alignment of authority and responsibility suggests that it is a bad idea to give one person the power to make decisions that others have to follow without having to personally live with the consequences. Architects sign up for programming tasks just like any programmer. However, they are also on the lookout for big changes that have big payoffs

Another task for architects on an XP team is partitioning systems. Partitioning isn't an up-front, once-and-for-all task, though. Rather than divide and conquer, an XP team conquers and divides. First a small team writes a small system. Then they find the natural fracture lines and divide the system into relatively independent parts for expansion. The architects help choose the most appropriate fracture lines and then follow the system as a whole, keeping the big picture in mind as the groups focus on their smaller section.


the S.P.I.D.R. approach to splitting stories
SPIKES
Make a large story smaller by pulling out a spike, which is a research activity after which the team will know more.
• Sometimes just doing a spike makes the remaining work a manageable size.
• Other times, the new knowledge creating by the spike makes it easier to see ways to split the story.
RULES
Sometimes a story is large because of the business rules, technology standards, or such that must be supported.
• Consider relaxing support for these rules in an initial story.
• Add support for additional rules in subsequent stories.
PATHS
Consider the paths through a story and split each path into its own story.
• Draw a simple flowchart of what happens in a story. Each sequence of steps can be a story.
• Expand one big step of the flowchart into a story.
DATA
Look for ways to split the story based on the type of data that must be supported.
• Can a first story support valid data and a later story add support for invalid data?
• How about frequent types of data and less frequently seen types of data?
INTERFACES
Split a story across multiple interfaces if supporting those interfaces makes the story take significantly longer to develop.
• Split out stories by browser type or version, or by different hardware.
• Consider building a minimal user interface first or leave styling out of an interface initially.

Lean Software Development



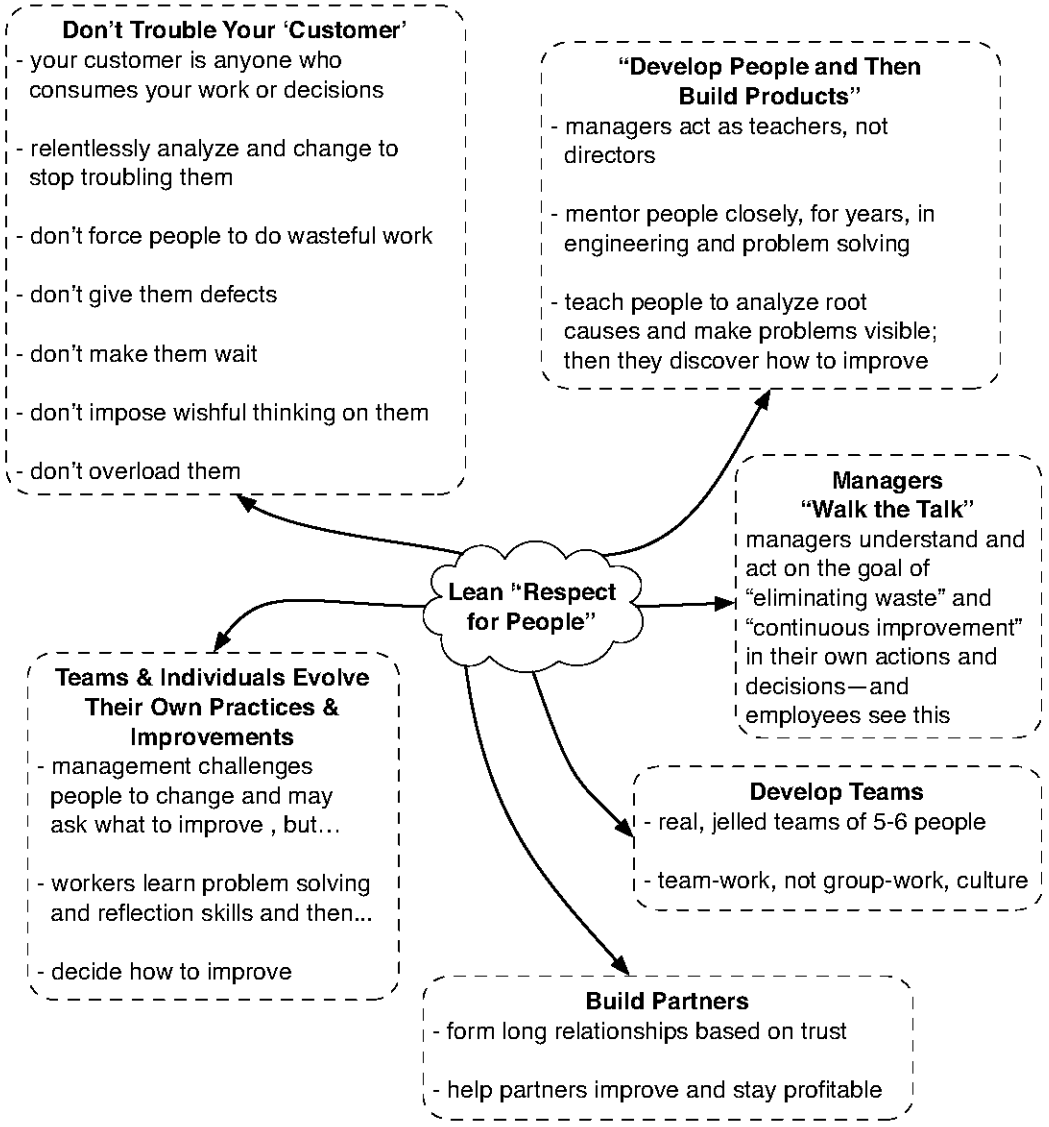
This is a 21-step program for implementing lean software development. Each step contains just a brief sketch of what might be done. Consider these as suggestions; try them and see if they produce the results that you are looking for in your organization.

Optimize the Whole

1. Implement lean across an entire value stream and the complete product: Appoint a value stream leader or leadership team that accepts responsibility for the entire value stream starting and ending with customers. Focus on the whole product, not just the software. Draw a value stream map and look for interruptions in flow, loop-backs or churn, and areas in the flow that are either not available when needed or not capable of delivering the needed results. Fix broken processes, and provide missing system capabilities.
2. Restructure the measurements: Chances are very high that there are local measurements within the value stream, and chances equally high that these measurements lead to sub-optimization or worse dysfunction. Stop using the local measurements, and use value stream measurements instead.
3. Reduce the cost of crossing boundaries: If there are big delays in a value stream, they probably occur at department or company boundaries. Look at these boundaries carefully and evaluate how much they are actually costing. Anything you can do to speed the flow of value across these boundaries will almost certainly reduce the cost of crossing the boundary.

Respect People

4. Train team leaders/supervisors: Instead of focusing on process leaders, give natural team leaders the training, the guidance, the charter, and the time to implement lean in their areas.
5. Move responsibility and decision making to the lowest possible level: Your lean initiative should be implemented by the work teams that create value under the guidance of their existing leaders. Have work teams design their own lean processes, with guidance from properly trained team leads and supervisors. Expect the work teams to ask for what they need to make the new processes work.
6. Foster pride in workmanship: Many things impede pride in workmanship: individual over team rewards, sloppy workspaces and work practices, impossible deadlines, no time for proper testing or refactoring, imposed processes, routine or robot-like jobs, etc. Root out and eliminate these practices: never create conflicting incentives among team members, automate routine tasks, and let workers decide the best approach to their job, never ask for sloppy work in the interest of meeting deadlines. Encourage passionate commitment and expect top-quality results. This will have a more sustained positive impact by far than individual incentives and bonuses.



Deliver Fast

7. Work in small batches: Reduce project size. Shorten release cycles. Stabilize. Repeat. Clean out every list and queue, and aggressively limit their size going forward.
8. Limit work to capacity: Have teams work at a repeatable cadence to establish capacity. Expect teams to pull from queues based on their proven velocity and to completely finish the work before they start on more work. Limit queue sizes, and accept no work unless there is an empty slot in a queue.
9. Focus on cycle time, not utilization: Stop worrying about resource utilization and start measuring time-to-market and/or customer response time.

Defer Commitment

10. Abolish the notion that it is a good practice to start development with a complete specification: Concurrent development means allowing the specification to emerge from the development process. Concurrent development saves money, it saves time, it produces the best results, and it allows you to make decisions based on the most current data. There is nothing not to like about concurrent development, so why cling to the idea that there ought to be a complete specification before getting started with development? If the answer is driven by up-front funding, then move to incremental funding. If it is driven by fixed-price contracting practices, move to another contracting model.
11. Break dependencies: Instead of worrying about dependencies, do everything possible to break them so that any feature can be added in any order. A divisible systems architecture is fundamental. Ruthlessly question dependencies, and tolerate them only very sparingly.
12. Maintain options: Develop multiple options for all irreversible decisions and never close off an option until the last responsible moment. Critical design decisions are tradeoffs. Invest enough in understanding the impact of each option to have the data and the confidence to make the best choice. For all other decisions, create change-tolerant code, keep it clean and simple, and do not hesitate to change it.

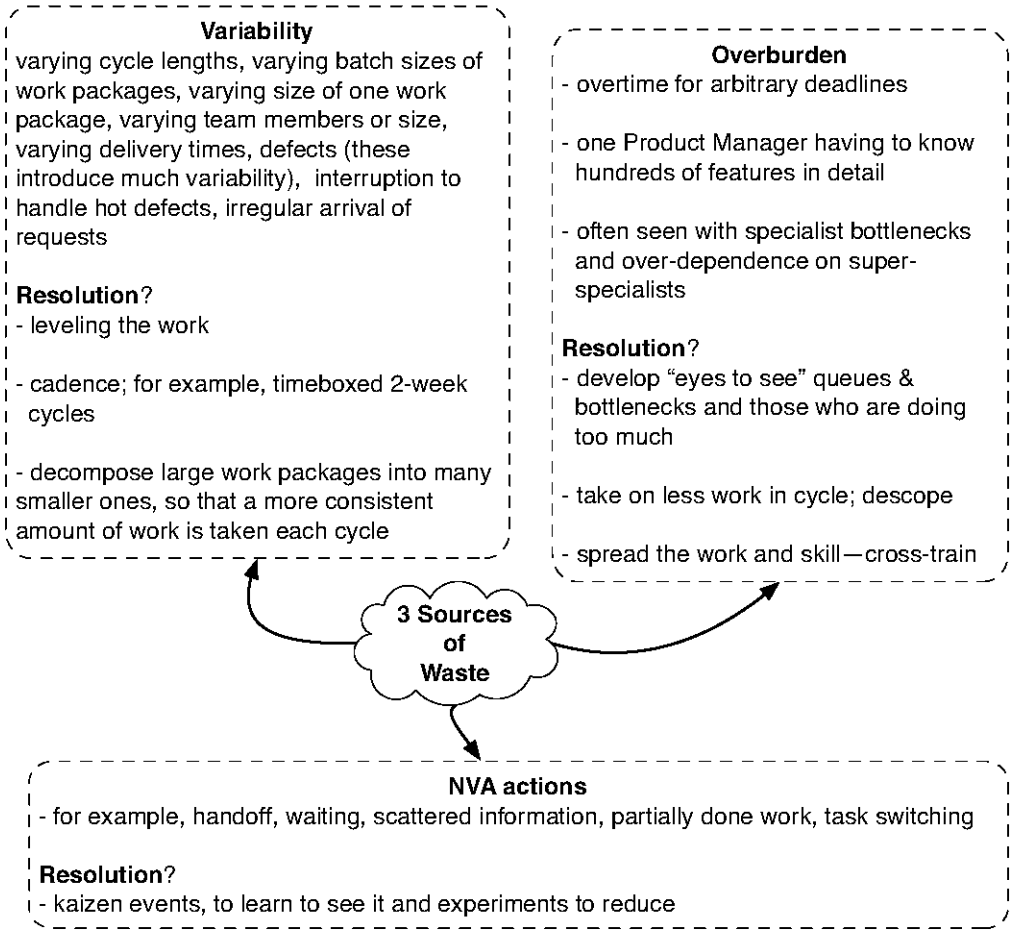
Create Knowledge

13. Create design-build teams: Assure that the systems architecture allows products to be broken into logical modules that can be addressed by cross-functional teams representing the interests of all steps in the value stream. Provide each team with the appropriate leadership and incentives to maintain engagement, transparency, and intensive feedback. These teams must be encouraged to share early and often, fail fast, and learn constantly.
14. Maintain a culture of constant improvement: Create the time and the expectation that every team and every function will continually examine and improve its processes and test its assumptions. Hold cross-team and cross-functional events to identify accommodations and constraints in the flow of value and replace these with practices and policies that will improve overall results.
15. Teach problem-solving methods: Teach the PDCA cycle or the scientific method or some variation of these problem-solving methods. Expect teams to establish hypotheses, conduct many rapid experiments, create concise documentation, and implement warranted changes.

Build Quality In

16. Synchronize: Aggressively reduce partially done work. Write tests first. Test code as soon as possible. Don't put defects on a list; stop and fix them the moment they are detected. Integrate code as continuously and as extensively as possible. Use nested synchronization to integrate with ever larger and more complex systems.
17. Automate: Accept that people make mistakes, and mistake-proof through automation. Automate every process you can as soon as early as possible. Automate testing, builds, installations, anything that is routine, but be sure to automate in a way that supports people and keeps them thinking about how to do things better.
18. Refactor: Keep the code base clean and simple, and the minute duplication shows up, refactor the code, the tests, and the documentation to minimize complexity.

Eliminate Waste



19. Provide market and technical leadership: Assure that there is clear responsibility for developing a deep understanding of what customers' value and a closely associated deep understanding of what the technology can deliver. Be sure the team builds the right thing.
20. Create nothing but value: Assure that all steps in all processes are focused on value-creating activities or improved capability to deliver value in so far as this is possible. Measure process cycle efficiency and keep on improving it.
    Waste is the difference between what is valuable and what we think is valuable.

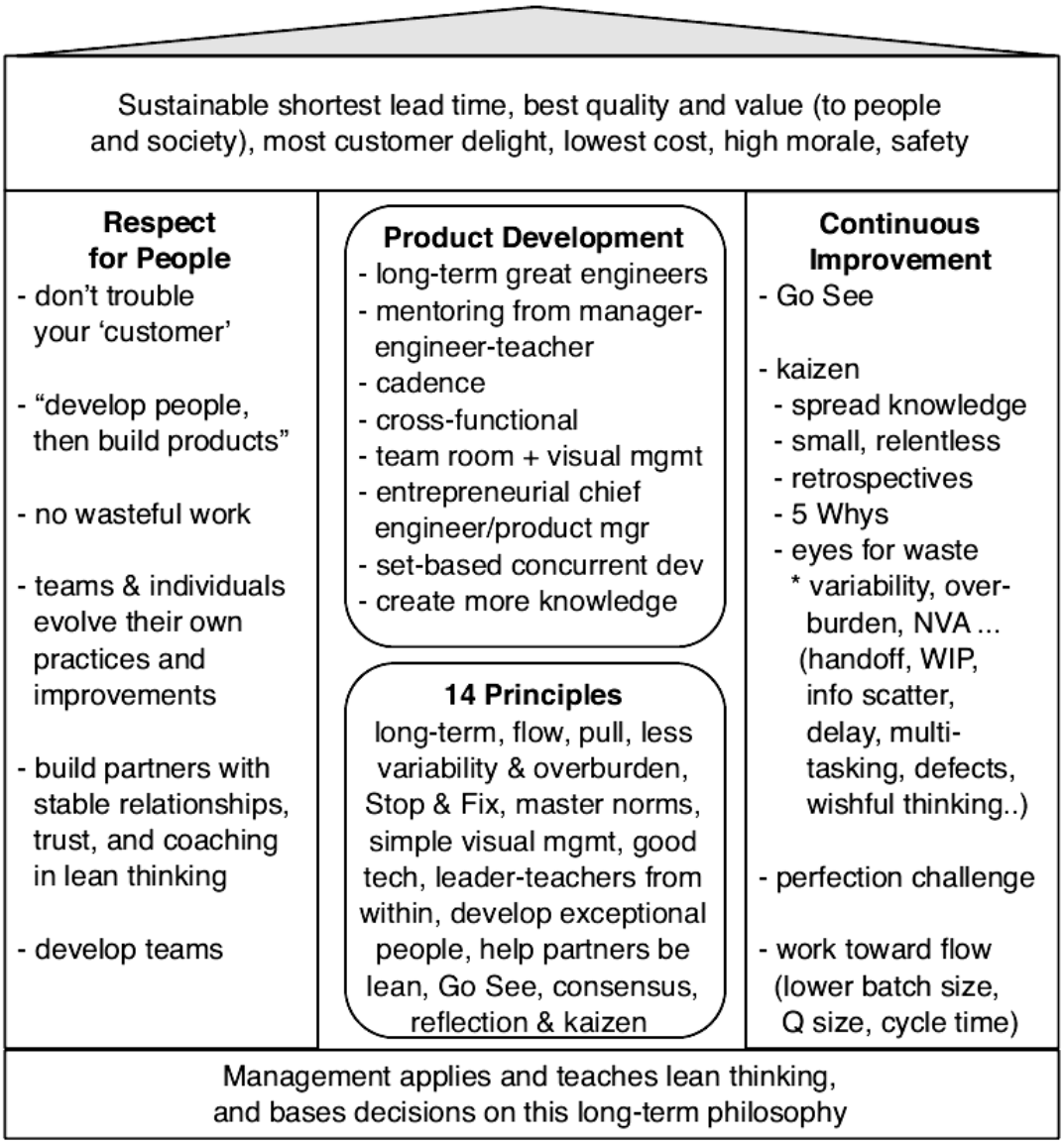
21. Write less code: Aggressively limit the features in a system to only those that are absolutely necessary to add value. Develop an organizational intolerance for complexity.

Lean Journey

Lean development is a journey that begins where you are and takes you far into the future. We offer a brief roadmap for getting started on your journey:

1. Begin where you are: How do you create value and make a profit?
2. Find your biggest constraint: What is the biggest problem limiting your ability to create value and make a profit?
3. Envision your biggest threat: What is the biggest threat to your ability to continue creating value and making a profit over the long-term?
4. Evaluate your culture: Establish and reinforce a culture of deep respect for Developers, Testers and for partners. Remove barriers that get in the way of pride in workmanship.
5. Train: Train team leads and managers how to lead, how to teach, and how to help people use a disciplined approach to improving work processes.
6. Solve the biggest problem: Turn the biggest constraint over to work teams. Expect many quick experiments that will eventually uncover a path to a solution.
7. Remove accommodations: Uncover the rules that made it possible to live with the constraint. Decide what the new rules should be.
8. Measure: See if end-to-end cycle time, true profitability and real customer satisfaction have improved.
9. Implement: Adopt changes supported by results.
10. Repeat the cycle: With the biggest problem addressed, something else will become your biggest problem. Find it and repeat the cycle.

Lean Thinking House



How to go agile?

The Agile Manifesto provides a philosophical foundation for effective software development. Contrary to some of the criticisms you may have heard from the traditional community, the fact is that the agile movement is based on some very solid concepts and the methodologies clearly reflect that. Unfortunately, the agile community currently suffers from low-end hackers claiming to be agile (e.g. the "we don't document therefore we're agile" crowd) and many traditionalists jump on that and say that agile is a bad idea. Yes, the "code-and-fix" approach to development is a bad idea, but code-and-fix isn't agile, regardless of what these clowns claim.

Some guidelines while going agile:

- Don't make an executive proclamation that all development will be done using the agile approach and then purchase training for everyone in the company. This approach will burn up a lot of money quickly and fuel the resistance movement from the beginning.
- Do seek support at both the highest and lowest levels of the organization. The programmers, at least, have to want the change, and they need the input and support of some of the executive team. Either group without the other will become isolated and unable to proceed.
- Do get one project team working successfully.
- Do staff that team with at least two people who are competent, respected, and can pull off the agile approach.
- Don't shower that team will an undue amount of support and attention. The other teams will just feel left out and get annoyed at that team.
- Do get experts in to help. Place them in the teams, not outside as advisors.
- Do consider the users and the executive sponsors part of the team ("There's only us").
- Do reflect on what you are doing each sprint/iteration and change your working convention to get better. If you are not changing something, trying out something new, each month for the first few months, then you probably aren't getting a taste of it yet.
- Do deliver real software to real users (at least) monthly.
- Finally, one word of comfort: The higher-level executives probably want what the agile approach offers, which is "Early and regular delivery of business value."

Over the last five years, people have been quoting agile rhetoric without adopting the elements that make the agile approach work. This section is my little attempt to correct some of the misquotations of the agile model that overzealous would-be agile developers inflict upon their managers.

Before getting into ways to misconstrue the message, let's first review what the agile message is supposed to be.

- First, create a safety net for the project, one that increases the chances of a successful outcome. This safety net consists of the following:
    - Everyone on the team sits within a small physical distance of each other.
    - The team uses an automated tool suite that integrates the code base every short while (continuously, each half hour or each hour), runs the test suite, and posts the result on an information radiator.
    - The team, working in small increments so that they check in code and tests several times a day, creates unit and system tests for the automated integration system to run.
    - The sponsor and key users visit the project team often to offer insights, feedback, and direction.
    - The team delivers running, tested features to real users every week or month.
    - Following each delivery, the sponsors, key users, and development team get together and review their mission, the quality of their product, and their working conventions.

- After the safety net is in place, see what can be removed from the development process. Typically, requirements documents can be lightened, planning documents put into a simpler form, and design documentation done in a different way. Typically, more training also needs to be brought in, more test automation done, designs simplified, and community discussions increased.

The point I wish to make is that simplifying the process is a reward the team gets for putting the safety net into place. Many would-be agile developers take the reward without ever putting the safety net in place

Some teams claim they're "going Agile" when all they're really doing is compressing the schedule, tossing out the documentation, and coding up to the last minute.
As Abraham Lincoln said, "If you call a tail a leg, how many legs does a dog have? Four. Because calling it a leg doesn't make it a leg."
Similarly, just calling a project or team "Agile" doesn't make it so.

Testing in Agile Projects

Agile methods pose three key challenges for testing:

- The software is a moving target:
  In traditional testing, system testers typically push for an early code freeze so they have time to test the feature-complete system. This simply isn't possible in an agile context. There may be some short period of time in which new stories are not being implemented, but it is not the month or more that system test teams are accustomed to having for a full regression cycle. Agile testing must take that continuous change into account. That means finding ways to test earlier.

- Short iterations and frequent releases increase time pressure
  Traditional test teams accustomed to having 4 – 6 weeks at the end of a release cycle to do nothing but end-to-end system or regression tests will discover that simply isn't possible in an Agile context. The iterations are too short.
  Agile testing must provide feedback not only sooner but faster. The solution to these first two challenges is not to find ways to do the same type of testing only faster, but to reevaluate the testing processes entirely.

- The risks are shifting
  Agile projects still have risks, of course. Agile practices, particularly XP programming practices, mitigate risks associated with late breaking change. Other risks remain. The Agile tester's challenge is to detect those new risks and determine how to respond to them.

  For example, a tester realized on the XP projects had significant risks around:
  - The parts of the system that stayed mocked out the longest.
  - Team's assumptions about production data we didn't control.
  - Areas of the system that didn't have many automated acceptance-level tests.

> By understanding the risks, the tester was able to focus my test efforts more effectively. Testers found a number of issues by testing around each area where the team had interfaced to other software and data that the team didn't control.

Test documentation can account for a large percentage of the test effort. Polls were conducted to understand exactly how much time test documentation takes. 135 testers across 57 organizations revealed that testers spend about one third of their time just documenting test cases.
Most of this documentation can probably be avoided if testers write automated test scripts.

In addition to supplementing the team's test effort, Agile testers can support the team by:

- Asking "what if" questions of both the programmers and business stakeholders in the planning game. For example, "What if the migrated data has null values?" or "What if a user decides to…?"
- Analyzing risks and providing information early. For example, "A similar Web application that used a similar mechanism was hijacked by spammers. What safeguards do we have in place to ensure this won't become a spam machine?"
- Offering information about external dependencies or requirements that the team might not otherwise know about. For example, "I would expect this application to look and feel like this other related application that was just released. Will it?"

If programmers change how they create the software, testing needs to change as well. Traditional test processes are not agile; heavyweight documentation, fragile automation, and extensive test tracking slow down the test effort and make it more difficult for the test process to adapt to extensive changes in the software.

Testers can become more agile, and thus adapt better to Agile methods if they:

- Streamline test processes
- Focus on providing feedback not assessments
- Shift their role from last line of defense to team supporter

Without these changes, testers find themselves at odds with Agile programmers. But by becoming more agile in their approach, testers can have a huge positive impact, helping Agile project be even more Agile

Testers on an Agile team help customers choose and write automated system-level tests in advance of implementation and coach programmers on testing techniques. On Agile teams much of the responsibility for catching trivial mistakes is accepted by the programmers. Test-first programming results in a suite of tests that help keep the project stable. The role of testers shifts too early in development, helping define and specify what will constitute acceptable functioning of the system before the functionality has been implemented.

In Weekly Cycle, the first thing that happens to the chosen stories is that they are turned into automated system-level tests. This is one leveraged place for strong testing skills. Customers may have a good idea of the general behavior they want to see, but testers are good at looking at "happy paths" and asking what should happen if something goes wrong. "Okay, but what if login fails three times? What should happen then?" In this role testers amplify communication. They ensure that the system-level tests succeed only when the stories are fully implemented and ready for deployment.

Once the tests for the week are written and failing, testers continue to write new tests as implementation uncovers new details that need to be specified. Testers can also work to further automate and tune tests. Finally, when a programmer gets stuck on a knotty testing problem, a tester can pair with the programmer to help solve the problem

Reward Schemes

Companies die because their managers focus on the economic activity of producing goods and services, and they forget that their organizations' true nature is that of a community of humans.
People are committed to a company if they feel that the company is committed to them.
When an employee isn't performing, the first question a manager should ask is, "What am I doing wrong?"

Rewards actually reduce the intrinsic joy and output quality of an otherwise fun activity.

- Young children, who are rewarded for drawing, are less likely to draw on their own than are children who draw just for the fun of it.
- Teenagers offered rewards for playing word games enjoy the games less and do not do as well as those who play with no rewards.
- Employees who are praised for meeting a manager's expectations suffer a drop in motivation.
- In one study, girls in the fifth and sixth grades tutored younger children much less effectively, if they were promised free movie tickets for teaching well.
- The study, by James Gabarino, now president of Chicago's Erikson Institute for Advanced Studies in Child Development, showed that tutors working for the reward took longer to communicate ideas, got frustrated more easily, and did a poorer job in the end than those who were not rewarded.

If rewarding intrinsically motivated behavior destroys intrinsic motivation, what rewards might retain a person's intrinsic motivation?

- Pride-in-work
- Pride-in-accomplishment
- Pride-in-contribution

People who have pride in their work do a better job than those who do not, and they are also more likely to step outside of their own job descriptions to repair or report some other problem that they notice. Even though their only reward may be that they have done a good deed, we continually encounter people for whom this is sufficient.

In Scrum, the team is treated as one. The team is rewarded and not individuals on the team. Annual merit ratings for individuals create competition rather than cooperation and kill pride in workmanship. Companies that measure individual performance often wonder, “Why team work is so poor?”
HR departments of many companies needs to evolve their policies to accommodate Scrum teams.

XP vs. Toyota Production System

Toyota is one of the most profitable large auto manufacturers. It makes excellent products, grows fast, has high margins, and makes lots of money. Toyota's goal of going fast is not achieved by straining. Toyota eliminates wasted effort at every step of the process of producing cars. If you eliminate enough waste, soon you go faster than the people who are just trying to go fast.

Their alternative social structure of work is critical to the success of TPS (Toyota Car Production System). Every worker is responsible for the whole production line. If anyone spots a defect he pulls a cord that stops the whole line. All the resources of the line are applied to finding the root cause of the problem and fixing it. Unlike mass-production lines where someone "down the line" is responsible for quality, in TPS the goal is to make the quality of the line good enough that there is no need for downstream quality assurance. This implies that everyone is responsible for quality.

Individual workers have a lot of say in how work is performed and improved in TPS. Waste is eliminated through kaizen (continuous improvement) events. Workers identify a source of waste, either quality problems or inefficiency. Then, they take the lead in analyzing the problem, performing experiments, and standardizing the results.

TPS eliminates the rigid social stratification found in Taylorist factories. Industrial engineers begin their careers working on the line and always spend a considerable amount of time in the factory. Ordinary workers perform routine maintenance instead of an elite caste of technicians. There is no separate quality organization. The whole organization is a quality organization.

Workers are accountable for the quality of their work because the parts they create are immediately put to use by the next step in the line. If you use a part immediately you get the value of the part itself as well as information about whether the upstream machine is working correctly. This view, that parts aren't just parts but also information about their making, leads to pressure to keep all the machines in a line working smoothly and also provides the information necessary to keep the machines working smoothly.

The greatest waste is the waste of overproduction. If you make something and can't sell it, the effort that went into making it is lost. If you make something internally in the line and don't use it immediately its information value evaporates. There are also storage costs: you have to haul it to a warehouse; track it while it is there; polish the rust off it when you take it back out again; and risk that you'll never use it at all, in which case you have to pay to haul it away.

Software development is full of the waste of overproduction: fat requirements documents that rapidly grow obsolete; elaborate architectures that are never used; code that goes months without being integrated, tested, and executed in a production environment; and documentation no one reads until it is irrelevant or misleading. While all of these activities are important to software development, we need to use their output immediately in order to get the feedback we need to eliminate waste.

Requirements gathering, for instance, will not improve by having ever more elaborate requirements-gathering processes but by shortening the path between the production of

requirements detail and the deployment of the software specified. Using requirements detail immediately implies that requirements gathering isn't a phase that produces a static document; but an activity producing detail, just before it is needed, throughout development.

Assignments

Exercise: Existing Processes
How effective are the existing processes and development practices within your organization?

10 == couldn't be better; living the dream!
1 == failing on multiple levels; train wreck waiting to happen

Exercise: Defined vs. Empirical Processes
Part 1

1. Form pairs.
2. Assign one person the boss, the other is the worker.
3. The boss can give the following commands. Go, Stop, Right, Left, Faster, Slower
4. The worker must follow the boss' commands.
5. The boss is responsible for having the worker proceed 60 normal paces to a common destination within two minutes, from the time "go" is said, until "stop" is said, by the moderator.
6. The boss can command the worker but not touch the worker.

Part 2

1. With the same teams as before, except everyone is a worker and responsible for figuring out how to proceed during the exercise by him or herself.
2. Each team proceeds 60 normal paces to a common destination within two minutes, from the time "go" is said, until "stop" is said, by the moderator.

Exercise: MLBTix

Overall attendance at baseball games has increased over the last 10 years. In some cities, such as Boston, almost all games are sold out, and obtaining tickets through normal channels is nearly impossible. MLB rules prohibit the resale of tickets at a profit. Scalping (resell something for a quick profit) is illegal and has been cracked down on recently. The primary distribution channel for buying tickets is an online auction site, xAuction. Although all auctions for tickets on xAuction are supposed to be capped at the retail price plus expenses, MLB has learned that, through a variety of workarounds, these tickets are being scalped for prices of up to 1000 percent of the retail price.

Project Plan

The MLB commissioner's office hired an external consulting organization, Denture, to plan a project to manage the resale of baseball tickets. Denture delivered the final plan on November 15, and it was subsequently approved. Excerpts of the plan are provided here.

Project Background New legislation mandates that as of the 2016 baseball season all ticket resales must take place through MLB-authorized facilities. MLB has decided to develop such a facility on the Web; the site will be known as MLBTix. Through functionality similar to the online auction site, xAuction, but specific to MLB, the public will be able to buy and sell MLB tickets online. Sellers will auction the tickets to the highest bidder, setting an initial bidding price of their own choice without floor or ceiling conditions established by MLBTix. The seller can also limit the duration of the auction by setting a start and an end date and time. If the ticket(s) are successfully sold, the buyer pays the seller through the MLBTix credit card facilities, and the seller mails the tickets to the buyer. Sellers will automatically be notified when buyers receive their tickets, at which point MLBTix will mail a check for the proceeds (less the 25 percent MLB fee) to the seller.

The commissioner will be announcing MLBTix at a news conference on January 15. He hopes to have some functionality available by opening day, March 30, 2016, and for the site to be fully functional by the All Star break, which begins July 18, 2016. Therefore, March 30, 2016 is the anticipated release date. On this date, the MLBTix site will be up, and buyers and sellers will be able to register. Sellers will be able to make tickets available at a fixed price, which buyers will be able to pay in full via credit card. MLBTix is a go-between, but all tickets are transferred directly from sellers to buyers. The release schedule mandates that on June 30, 2016, auction capability be added to site. Finally, on August 30, 2016, buyers will be able to buy groups of collocated tickets, view the locations of seats being sold, and check on inventory.

Funds for the project are ample and should not be considered an unreasonable constraint. The deliverables are the date and functionality. Facilities or packaged software to support MLBTix can be either bought or developed— whichever helps meet the date. The commissioner needs a heads-up on the likelihood that the MLBTix will be available by the above dates prior to his press conference.

Product Backlog These are the functional requirements:

- Customers can register as potential sellers of tickets and be assigned a user ID and password.
- Customers can register as potential buyers of tickets and be assigned a user ID and password.
- Customers can maintain a profile under the user ID, including e-mail address, street address, preferences, and credit card information.
- Customers can place tickets up for auction, declaring a floor price, start of auction time/date, and end of auction time/date. Sufficient information should be provided so that buyers can ascertain that the tickets meet their requirements (for the right days, right teams, right number of seats located next to each other, and the seat locations in the ball park).
- Customers can cause an auction to be conducted for the tickets to registered buyers.
- Customer can have MLBTix successfully conclude the auction by awarding the tickets to the highest bidder by the end date and, at the same time, debiting the buyer's credit card and placing the funds in an MLBTix account.
- MLBTix will notify the seller of the successful sale of the tickets and provide the delivery information for the buyer.

- MLBTix will provide the buyer with a mechanism for indicating that the tickets were not successfully received by the selling date plus a specified period of time (for example, one week).
- MLBTix will transfer the funds for the ticket sale less 25 percent to the seller at the end of the specified delivery time, unless the buyer has indicated otherwise.
- MLBTix will transfer the 25 percent plus any interest to a corporate MLB account from the MLBTix account automatically.
- MLBTix will provide customers with inventory and inventory search capabilities for teams, tickets, dates, and seats.
- MLBTix will provide for promotions on MLBTix.
- MLBTix will be able to identify and ban abusers of MLBTix.

These are nonfunctional requirements. MLBTix must be able to

- Handle 250,000 simultaneous users with sub-second response time.
- Be secure at the anticipated level of financial activity (2,000 tickets per day at an average selling price of $50).
- Be scalable to 1,000,000 simultaneous users if necessary.
- Be 99 percent available, 24 hours a day, 7 days a week.

This is the development context for bidders: The system will be created in a development environment for building Open Source products, using Intel technology and software running on an OpenSQL database server. The development Team members will all live within easy commuting distance of the development site. There are currently cubicles at the development site. The development environment is wireless and has all power and networking capabilities already operating. The development environment uses Open Source development tools such as Eclipse. The development Team is required to use a source code library, check in code every time it's changed, build the software at least daily, and unit and acceptance test the software every time that it is built. Scrum will be used as the development practice. Use of any other aspects of Extreme Programming or any other engineering practices, such as coding standards, is up to the Team. All of the developers on the Team must have excellent engineering skills and at least be familiar with Scrum and Extreme Programming. The Team must consist of development engineers with excellent design and coding skills. These engineers are responsible for all testing and user documentation, although they can hire contractors to assist with this. The engineers on the Team must average 10 years of progressive experience on software projects using complex technology and Open Source software products. All Team members must be baseball aficionados.

The Project

Imagine that after a quick request for proposal (RFP) process, the commissioner of MLB selected your organization to develop MLBTix. In your response to the RFP, you assured the commissioner that you can meet the release schedule. You were present with the commissioner at a press conference on January 15 when he announced MLBTix, and at this press conference, you demonstrated the functionality completed during the first Sprint. This Sprint began on December 7, 2015, and the Sprint review meeting was held on January 7, 2016.

Your Team has just completed its third Sprint, which ended on March 7, 2016 (assume today is March 8, 2016). You have demonstrated the functionality developed during this Sprint to the commissioner. All the functionality necessary for the first release is in place. You intend to pull everything together into the production environment for the planned initiation of MLBTix on March 30, 2016, the start of the MLB 2016 season.

Uh-Oh!

At the Sprint planning meeting for the fourth Sprint, you and the Team become concerned about the capability of MLBTix to handle the kind of volumes that might be encountered. MLB has hired a public relations firm to market MLBTix, and it's done almost too good of a job: MLBTix has been the rage of every sports page and sports magazine. Everyone who knows about baseball knows about MLBTix and knows that it will be available as of 12:00 p.m. Eastern time on March 30, 2016. There are over 40 million baseball fans, and you know that almost no system could handle 40 million simultaneous hits.

You provide the commissioner with the following background information: The Team contacted several e-commerce retailers and determined that there would be on average 100 visits for every sale. The Team is unable to estimate the exact number of hits that will occur when the Web site first goes up but is worried that it will be more than it can handle. The MLB commissioner's research indicates that the site will likely sell 2,000 tickets per day in April 2016 and 5,000 per day thereafter for the rest of the season. The average price that will be charged by a seller above retail is $30, of which 25 percent will go to MLBTix. You have previously alerted the commissioner that the database technology Denture recommended the Team use is an iffy proposition at best, and scaling tests have shown the application to be database intensive. Even with all the tuning efforts from the consultants that have been brought in, and even running OpenSQL on the fastest RAID devices possible, the maximum number of simultaneous transactions that can be served with sub-three-second response time is 100 per second. Loads are expected to reach significant peaks at lunchtime and after dinner. The Team is concerned that peak volumes during normal production might overwhelm the server and that the knee of the performance curve will be very close to the 110-transactions-per-second rate. You have determined that the Miracle database will readily support the scaling requirements predicted by the commissioner, but it will take one more Sprint to trade out OpenSQL and implement Miracle database. The upshot? The application won't be ready until a month after the season opener.

What Advice Should You Provide?

You tell the commissioner all of this. You notice that the commissioner gets increasingly agitated during your presentation, tapping his feet, spitting at the floor, and uttering muffled expletives. He appears to be very unhappy. The commissioner tells you to knock off all of this technology mumbo-jumbo and tell him what he should do. He wants to know whether he should call in his public relations people and tell them to announce that he can't get MLBTix up. What should you advise the commissioner based on the above risk/reward model and your best instincts?

Exercise – Complete Teams

1. Do you have operations people on your team?
2. Is someone from tech support regularly consulted when designing new features?
3. Are testers involved in development right from the start?
4. When do technical writers get involved?
5. Are customers or customer proxies treated as full team members?

Exercise – Processing in Large Batches vs. Small Batches

Let 5 people sit in a circle. Give one person a box of coins.
Round 1: Each person flips ALL coins. When done with the entire batch, gives all the coins to the next person.
Round 2: Each person flips one person and passes the flipped coin to the next person. Keep flipping and passing until done.

Exercise – Deming

Deming's System of Profound Knowledge says:

a. It's the whole product, the whole team, the whole system that matters.
b. When something goes wrong, in all probability it was caused by the system, which makes it a management problem.
c. Use the scientific method to change and improve.
d. With people, the things that matter are skill, pride, expertise, confidence, and cooperation.

Imagine that Deming was scheduled to tour your organization next week. Prepare a presentation for him on how each of these points is addressed in your organization. What do you think his advice would be? Would you be prepared to act on it?

Exercise – Fourteen Points
Review Deming's 14 points with your team. For each point think:

- Is this point relevant today in our organization? Is it important?
- If we regard it as relevant and important, what does it suggest we should do differently? What would it take to make the change?

The 14 points are mentioned below:

1. Provide for the long-range needs of the company; don't focus on short term profitability.
2. The world has changed, and managers need to adopt a new way of thinking. Delays, mistakes, defective workmanship, and poor service are longer acceptable.
3. Quit depending on inspection to find defects, and start building quality into products while they are being built. Use statistical process control.

4. Don't choose suppliers on the basis of low bids alone. Minimize total cost by establishing long-term relationships with suppliers that are based on loyalty and trust.
5. Work continually to improve the system of production and service. Improvement is not a one-time effort; every activity in the system must be continually improved to reduce waste and improve quality.
6. Institute training. Managers should know how to do the job they supervise and be able to train workers. Managers also need training to understand the system of production.
7. Institute leadership. The job of managers is to help people do a better job and remove barriers in the system that keep them from doing their job with pride. The greatest waste is failure to use the abilities of people.
8. Drive out fear. People need to feel secure in order to do their job well. There should never be a conflict between doing what is best for the company and meeting the expectations of a person's immediate job.
9. Break down barriers between departments. Create cross-functional teams so everyone can understand each-other's perspective. Do not undermine team cooperation by rewarding individual performance.
10. Stop using slogans, exhortations, and targets. It is the system, not the workers, that creates defects and lowers productivity. Exhortations don't change the system; that is management's responsibility.
11. Eliminate numerical quotas for workers and numerical goals for people in management. (We add: Eliminate arbitrary deadlines for development teams.) This is management by fear. Try leadership.
12. Eliminate barriers that rob the people of their right to pride of workmanship. Stop treating hourly workers like a commodity. Eliminate annual performance ratings for salaried workers.
13. Encourage education and self-improvement for everyone. An educated workforce and management is the key to the future.
14. Take action to accomplish the transformation. A top management team must lead the effort with action, not just support.

Exercise – Visual Workspace

In column 2, answer the question posed in column 1 as it relates to your organization.
In column 3, rate how self-directing your organization is.
Give your organization a score of 0 to 5, with

0 = people are told what to do and
5 = people figure out among themselves what to work on next.

Question	Current Practice	Score
1) How do people know what customers really want?		
2) How do people get technical questions answered?		
3) How do people know what features to work on next?		
4 How do people know what defects to work on next?		

Question	Current Practice	Score
5) How do people know if tests are passing?		
6) How do people know their progress toward meeting the overall goal of their work?		

Exercise – Employee Survey

Do you have an employee survey? Does your employee survey have these questions?

a. Do you feel that the compensation system is fair?
b. Do you feel that the promotion system is fair?
c. Rate how the compensation affects your dedication to your job (Scale of 1 to 5):
    1. It angers me and gets in the way of doing a good job.
    2. It sometimes annoys me and occasionally affects my performance.
    3. It doesn't make much difference.
    4. It occasionally motivates me to work harder.
    5. It motivates me to work hard every day.

How would it impact your organization if these questions are added?

Exercise – Problem
What, exactly, is our biggest problem? And what, exactly, are we going to do about it?

Exercise - Refactoring

1. What is your team's practice regarding refactoring?
2. Before you add any new features, do you first change the design to simplify it, without making any feature changes, and test the new design to be sure nothing has changed?
3. Do you refactor to simplify immediately after getting a new feature working?

Exercise – Testing

On a scale of 0-5 (with 0 = low and 5 = high) rate your organization on:

a. Standardized architecture
b. Standardized tools
c. Coding conventions
d. Configuration management
e. Automated unit tests
f. Automated acceptance tests
g. One-click build and test

h. Continuous integration
i. Automated release
j. Automated installation

For any score that is below 3, is there room for improvement?
What stops your team from doing this improvement?

Exercise – Cycle Time

As per Little Law:

$$\text{Cycle time} = \frac{Number\ of\ items\ in\ the\ process\ (work\ in\ progress)}{Throughput\ (How\ many\ items\ are\ produced\ per\ unit\ of\ time)}$$

A simpler definition is, "Cycle time is the time elapsed between the customer asking for a feature and development team delivering it."

What is the cycle time?

1. A team is producing production grade code every sprint. Sprint length is 4 weeks. Sprint backlog, on an average, is 50 user stories.
2. In the above example, what if the sprint length was 2 weeks (instead of 4) and average user stories done every sprint were 20 (instead of 50).
3. A team is producing integration tested code every 2 weeks' sprint. They do a hardening cycle of 4 weeks after typically 10 sprints and then produce production grade code.

Exercise – The 5S

Ask the team to take a look at your team room. Rate its general appearance on a scale of 0-5 (high). Now rate the general neatness and simplicity of your code base on the same scale. Are the results similar? If you have a score of 3 or lower, propose to the team that you do a 5S exercise, first on the room and then on the code base.

The 5S for Java

1. Sort: Reduce the size of the code base. Throw away all unneeded items immediately. Remove:
    - Dead code
    - Unused imports
    - Unused variables
    - Unused methods
    - Unused classes
    - Refactor redundant code
2. Systematize: Organize the projects and packages. Have a place for everything and everything in its place.
    - Resolve package dependency cycles
    - Minimize dependencies
3. Shine: Clean up. Problems are more visible when everything is neat and clean.
    - Resolve unit test failures and errors (passed == 100%)

- Improve unit test coverage (> 80%)
- Improve unit test performance
- Check AllTests performance
- Resolve checkstyle warnings
- Resolve PMD warnings
- Resolve javadoc warnings
- Resolve TODO's

4. Standardize: Once you get to a clean state, keep it that way. Reduce complexity over time to improve ease of maintenance.
5. Sustain: Use and follow standard procedures.

Exercise – Standards

Who is responsible for setting standards? Who should be? How closely are standards followed? How easily are they changed? Is there a connection between the last two answers in your organization?

Exercise – Contracting

What is the primary purpose of outsourcing in your company? What kinds of activities does the company keep inside rather than outsource? Are the incentives of outsourcing agreements aligned with the best interests of your company and your customers? Can you imagine ways in which your outsourcing arrangements might create questions of allegiance in the minds of workers in your company? In the other company?

For contracting companies: What kinds of contracts do you routinely use? What do you see as the key benefit of that kind of contract? What is the key risk? If you were in the shoes of your contractors, what would you see as the benefits and risks of that kind of contract?

For contractors: What kinds of contracts do you routinely engage in? Are some types better for you than others? Consider your favorite contracting format and reconsider it from the point of view of the contracting companies. What do you see as the benefits and risks for them?

Exercise – Scrum Product Backlog item

Product backlog item	**Estimate**
Read a high level, 10-page overview of agile software development in a business magazine.	
Read a densely written 5-page research paper about agile software development in an academic journal	
Write the product backlog for a simple eCommerce site that sells only clocks	
Recruit, interview and hire a new member for your team	
Create a 60-minute presentation about agile estimating and planning for your coworkers	
Read a 150-page book on agile software development	
Write an 8-page summary on this session for your boss	

In the above table, mark any item as 5 units in estimate. Give relative estimate for other items. You can choose only one of the following units: 0, 1, 2, 3, 5, 8, 13, 20, 40, 100 and "?".

If a task is above 40, then try breaking it into smaller tasks. Use the poker card method for estimate. Total time for this exercise is 10 minutes for a group of about 5 people.

Exercise – Scrum – Available time exercise
This exercise is to be performed by a group of 6 persons. If there are lesser people in any group, one person can perform two roles.

Aishwarya	Out of office on vacation the 12th and 13th. 5 total productive hours each day, but average 1 hour of maintenance responsibilities each day.
Abhishek	In training all day on the 20th. 6 total productive hours each day, and no maintenance responsibilities, but divides time 50/50 between this team and another
Shah rukh	No days out of the office planned. 6 total productive hours each day, but average of 2 hours of bug-fixing responsibilities each day.
Amitabh	Out of vacation the week of the 11th. 7 total productive hours each day, but average of 2 hours of maintenance responsibilities each day.
Hrithik	Out of office on the 13th. 5 total productive hours each day, and no maintenance responsibilities
Preeti	No days out of office planned. 6 total productive hours each day, but averages 1 hour of operational responsibilities each day.

The 4-week sprint is marked in the calendar below.

Mon	Tue	Wed	Thu	Fri
				1
4 Sprint Planning Meeting	5	6	7	8
11	12	13	14	15
18 Office Closed. (state holiday)	19	20	21	22
25	26	27	28	29 Sprint Review and Retro

Calculate the available working hours during the Sprint for the team.
Sprint Length – 4 weeks
Available working days during the Sprint – 17

Team Member	Available days during Sprint (Net of holidays & planned days out of office)	Available hours per day for Sprint	Total available hours during Sprint
Aishwarya			
Abhishek			

Shah rukh			
Amitabh			
Hrithik			
Preeti			

Exercise – Scrum Release Planning Exercise

You are a team that's planning a release. Start by calculating the team's velocity, based on the last 3 sprints. Then look at the product backlog and answer the following questions

- How much of the product backlog could you commit to release by May 15? (Date-driven release)
- On what date could you commit to a release with product backlog items 1-15? (Feature-driven release)
- Could you commit to a release with product backlog items 1-32 by June 1? (both date-driven and feature-driven)

Data:

- Sprint begins on 1-Jan-2016
- All sprints are 4 weeks in length
- Before each release, you will do a 2-week pre-release Sprint of final integration, testing, etc.

Velocity data from first 3 sprints is given below. These 3 sprints are over. The last sprint completed on 31-Dec-2015.

	Backlog Item	Estimated Size
Sprint 1	Implement the ___ functionality	20
	Fix the ___ bug	50
	Rearchitect the ___ module	20
	Investigate the ___ solution	60
	Upgrade the ___ servers	60
Sprint 2	Implement the ___ functionality	50
	Fix the ___ bug	60
	Rearchitect the ___ module	20
	Investigate the ___ solution	20
	Upgrade the __ servers	10
	Implement the ___ functionality	10
Sprint 3	Fix the ___ bug	40
	Rearchitect the ___ module	60
	Investigate the ___ solution	40
	Upgrade the ___ servers	40
	Implement the ___ functionality	40
	Fix the ___ bug	20

2016 Calendar

Mon	Tue	Wed	Thu	Fri

				01-Jan
04-Jan	05-Jan	06-Jan	07-Jan	08-Jan
11-Jan	12-Jan	13-Jan	14-Jan	15-Jan
18-Jan	19-Jan	20-Jan	21-Jan	22-Jan
25-Jan	26-Jan	27-Jan	28-Jan	29-Jan
01-Feb	02-Feb	03-Feb	04-Feb	05-Feb
08-Feb	09-Feb	10-Feb	11-Feb	12-Feb
15-Feb	16-Feb	17-Feb	18-Feb	19-Feb
22-Feb	23-Feb	24-Feb	25-Feb	26-Feb
29-Feb	01-Mar	02-Mar	03-Mar	04-Mar
07-Mar	08-Mar	09-Mar	10-Mar	11-Mar
14-Mar	15-Mar	16-Mar	17-Mar	18-Mar
21-Mar	22-Mar	23-Mar	24-Mar	25-Mar
28-Mar	29-Mar	30-Mar	31-Mar	01-Apr
04-Apr	05-Apr	06-Apr	07-Apr	08-Apr
11-Apr	12-Apr	13-Apr	14-Apr	15-Apr
18-Apr	19-Apr	20-Apr	21-Apr	22-Apr
25-Apr	26-Apr	27-Apr	28-Apr	29-Apr
02-May	03-May	04-May	05-May	06-May
09-May	10-May	11-May	12-May	13-May
16-May	17-May	18-May	19-May	20-May
23-May	24-May	25-May	26-May	27-May
30-May	31-May	01-Jun	02-Jun	03-Jun
06-Jun	07-Jun	08-Jun	09-Jun	10-Jun
13-Jun	14-Jun	15-Jun	16-Jun	17-Jun
20-Jun	21-Jun	22-Jun	23-Jun	24-Jun
27-Jun	28-Jun	29-Jun	30-Jun	01-Jul
04-Jul	05-Jul	06-Jul	07-Jul	08-Jul
11-Jul	12-Jul	13-Jul	14-Jul	15-Jul
18-Jul	19-Jul	20-Jul	21-Jul	22-Jul
25-Jul	26-Jul	27-Jul	28-Jul	29-Jul
01-Aug	02-Aug	03-Aug	04-Aug	05-Aug
08-Aug	09-Aug	10-Aug	11-Aug	12-Aug
15-Aug	16-Aug	17-Aug	18-Aug	19-Aug
22-Aug	23-Aug	24-Aug	25-Aug	26-Aug
29-Aug	30-Aug	31-Aug	01-Sep	02-Sep
05-Sep	06-Sep	07-Sep	08-Sep	09-Sep
12-Sep	13-Sep	14-Sep	15-Sep	16-Sep
19-Sep	20-Sep	21-Sep	22-Sep	23-Sep
26-Sep	27-Sep	28-Sep	29-Sep	30-Sep
03-Oct	04-Oct	05-Oct	06-Oct	07-Oct
10-Oct	11-Oct	12-Oct	13-Oct	14-Oct
17-Oct	18-Oct	19-Oct	20-Oct	21-Oct
24-Oct	25-Oct	26-Oct	27-Oct	28-Oct

31-Oct	01-Nov	02-Nov	03-Nov	04-Nov
07-Nov	08-Nov	09-Nov	10-Nov	11-Nov
14-Nov	15-Nov	16-Nov	17-Nov	18-Nov
21-Nov	22-Nov	23-Nov	24-Nov	25-Nov
28-Nov	29-Nov	30-Nov	01-Dec	02-Dec
05-Dec	06-Dec	07-Dec	08-Dec	09-Dec
12-Dec	13-Dec	14-Dec	15-Dec	16-Dec
19-Dec	20-Dec	21-Dec	22-Dec	23-Dec
26-Dec	27-Dec	28-Dec	29-Dec	30-Dec

Product Backlog as of today i.e. Jan 1 2016.

Id	Backlog Item	Estimated Size	Cumulative total
1.	Implement the ___ functionality	20	20
2.	Fix the ___ bug	10	30
3.	Rearchitect the ___ module	40	70
4.	Investigate the ___ solution	10	80
5.	Upgrade the ___ servers	10	90
6.	Implement the ___ functionality	50	140
7.	Fix the ___ bug	60	200
8.	Rearchitect the ___ module	40	240
9.	Investigate the ___ solution	10	250
10.	Upgrade the __ servers	40	290
11.	Implement the ___ functionality	60	350
12.	Fix the ___ bug	60	410
13.	Rearchitect the ___ module	30	440
14.	Investigate the ___ solution	40	480
15.	Upgrade the ___ servers	30	510
16.	Implement the ___ functionality	20	530
17.	Fix the ___ bug	10	540
18.	Implement the ___ functionality	10	550
19.	Fix the ___ bug	60	610
20.	Rearchitect the ___ module	10	620
21.	Investigate the ___ solution	50	670
22.	Upgrade the ___ servers	60	730
23.	Implement the ___ functionality	10	740
24.	Fix the ___ bug	20	760
25.	Rearchitect the ___ module	60	820
26.	Investigate the ___ solution	30	850
27.	Upgrade the __ servers	60	910
28.	Implement the ___ functionality	50	960
29.	Fix the ___ bug	60	1020
30.	Rearchitect the ___ module	10	1030
31.	Investigate the ___ solution	10	1040
32.	Upgrade the __ servers	40	1080

33.	Implement the ___ functionality	10	1090
34.	Fix the ___ bug	20	1110
35.	Implement the ___ functionality	20	1130
36.	Fix the ___ bug	20	1150
37.	Rearchitect the ___ module	10	1160
38.	Investigate the ___ solution	50	1210
39.	Upgrade the ___ servers	20	1230
40.	Implement the ___ functionality	30	1260
41.	Fix the ___ bug	30	1290
42.	Rearchitect the ___ module	20	1310
43.	Investigate the ___ solution	30	1340
44.	Upgrade the ___ servers	20	1360
45.	Implement the ___ functionality	30	1390

Exercise – Self Organization of Team

You are the ScrumMaster. What would you do under the following circumstances?

1. The team has four developers, two testers and a DBA. Developers and Testers are not working well together. Developers work in isolation. Two days before the end of sprint, they throw the code "over the wall" to testers.
2. The team is failing to deliver potentially shippable software at the end of sprint. Nothing is 100% production grade. They are close, but work needs to be done on the same user story in the next iteration.
3. The team seems to be consistently under committing during sprint planning. They finish the work that they commit, but it does not seem much. The Product owner has not yet complained about this issue.
4. The organization has all its code in Java. The team likes to work on new hot technologies. The team has chosen Ruby and Rails for development. The Product owner is not technical and cares only about the application running.
5. The organization has 20 agile projects. Each team has its own testers, who are starting to go in different directions in terms of preferred tools and approaches.
6. Jeff, a senior developer, is very dominating. During iteration planning, the team defers to him on every decision even though he is a horrible estimator. You notice the glances that other team members exchange when he suggests low estimates on some user stories.
7. You are the ScrumMaster for two teams. One team discusses all sides of various issues raised before making a decision. With the other team discussions drag on endlessly because they pursue absolute consensus in all cases.

Exercise – TeamWork Survey

Objectives:
To identify the present stage of the teamwork model that your team is presently operating in.

Directions

Below is mentioned a questionnaire that contains statements about teamwork. Next to each question, indicate how often your team displays each behavior by using the following scoring system:

- Almost never - 1
- Seldom - 2
- Occasionally - 3
- Frequently - 4
- Almost always - 5

Part 1 - Questionnaire

1. _____ We try to have set procedures or protocols to ensure that things are orderly and run smoothly (e.g. minimize interruptions, everyone gets the opportunity to have their say).
2. _____ We are quick to get on with the task on hand and do not spend too much time in the planning stage.
3. _____ Our team feels that we are all in it together and shares responsibilities for the team's success or failure.
4. _____ We have thorough procedures for agreeing on our objectives and planning the way we will perform our tasks.
5. _____ Team members are afraid or do not like to ask others for help.
6. _____ We take our team's goals and objectives literally, and assume a shared understanding.
7. _____ The team leader tries to keep order and contributes to the task at hand.
8. _____ We do not have fixed procedures, we make them up as the task or project progresses.
9. _____ We generate lots of ideals, but we do not use many because we fail to listen to them and reject them without fully understanding them.
10. _____ Team members do not fully trust the others members and closely monitor others who are working on a specific task.
11. _____ The team leader ensures that we follow the procedures, do not argue, do not interrupt, and keep to the point.
12. _____ We enjoy working together; we have a fun and productive time.
13. _____ We have accepted each other as members of the team.
14. _____ The team leader is democratic and collaborative.
15. _____ We are trying to define the goal and what tasks need to be accomplished.
16. _____ Many of the team members have their own ideas about the process and personal agendas are rampant.
17. _____ We fully accept each other's strengths and weakness.

18. _____ We assign specific roles to team members (team leader, facilitator, time keeper, note taker, etc.).
19. _____ We try to achieve harmony by avoiding conflict.
20. _____ The tasks are very different from what we imagined and seem very difficult to accomplish.
21. _____ There are many abstract discussions of the concepts and issues, which make some members impatience with these discussions.
22. _____ We are able to work through group problems.
23. _____ We argue a lot even though we agree on the real issues.
24. _____ The team is often tempted to go above the original scope of the project.
25. _____ We express criticism of others constructively
26. _____ There is a close attachment to the team.
27. _____ It seems as if little is being accomplished with the project's goals.
28. _____ The goals we have established seem unrealistic.
29. _____ Although we are not fully sure of the project's goals and issues, we are excited and proud to be on the team.
30. _____ We often share personal problems with each other.
31. _____ There is a lot of resisting of the tasks on hand and quality improvement approaches.
32. _____ We get a lot of work done.

Part 2 - Scoring

Next to each survey item number below, transfer the score that you give that item on the questionnaire. For example, if you scored item one with a 3 (Occasionally), then enter a 3 next to item one below. When you have entered all the scores for each question, total each of the four columns.

Item	Score	Item	Score	Item	Score	Item	Score
1.	_____	2.	_____	4.	_____	3.	_____
5.	_____	7.	_____	6.	_____	8.	_____
10.	_____	9.	_____	11.	_____	12.	_____
15.	_____	16.	_____	13.	_____	14.	_____
18.	_____	20.	_____	19.	_____	17.	_____
21.	_____	23.	_____	24.	_____	22.	_____

27. _____	28. _____	25. _____	26. _____
29. _____	31. _____	30. _____	32. _____
TOTAL _____	**TOTAL _____**	**TOTAL _____**	**TOTAL _____**
Forming Stage	**Storming Stage**	**Norming Stage**	**Performing Stage**

This questionnaire is to help you assess what stage your team normally operates. It is based on the "Tuckman" model of Forming, Storming, Norming, and Performing. The lowest score possible for a stage is 8 (Almost never) while the highest score possible for a stage is 40 (Almost always).
The highest of the four scores indicates which stage you perceive your team to normally operates in. If your highest score is 32 or more, it is a strong indicator of the stage your team is in.
The lowest of the three scores is an indicator of the stage your team is least like. If your lowest score is 16 or less, it is a strong indicator that your team does not operate this way.

If two of the scores are close to the same, you are probably going through a transition phase, except:

- If you score high in both the Forming and Storming Phases then you are in the Storming Phase
- If you score high in both the Norming and Performing Phases then you are in the Performing Stage

If there is only a small difference between three or four scores, then this indicates that you have no clear perception of the way your team operates, the team's performance is highly variable, or that you are in the storming phase (this phase can be extremely volatile with high and low points).

Exercise – Teamwork
Take a coin. Choose a partner. Your partner also has a coin. Both you and your partner choose either head or tail from you coin, without showing the coin to your partner. Both show their choice of the coin side at the same time to each other. Each person calculates his/her score by using the table:

Coin Side	Points	Comments
Head Tail	-5 +5	Both the person have different choice
Head Head	+1 +1	Both have chosen head
Tail Tail	-1 -1	Both have chosen tail

Repeat the above sequence multiple times. Keep a total of your score. The game will have a fixed time. You can change your partner at any time. You may also stop playing. The goal is to have maximum score at the fixed time.

Exercise – Requirements in Waterfall vs. Iterative.
We have a bag of coins. It needs to be sorted and total amount of money computed. Give your lowest bid in terms of time. The person with the lowest bid wins the contract and has to perform in the given time period.

Exercise – We are having a Party
Ingredients:

- At least 10 pages (8.5' X 11') per participant
- 1 marker per participant
- Stickers

Directions:

We are having a party, and we need to enlist everybody in the room to create invitation cards (3 per person). Begin by showing an example of what the finished card should look like:

1. fold page in half
2. draw a happy face on the front,
3. write a message on the inside,
4. sign the card,
5. stamp the back (sticker) and,
6. mail the card by dropping it in a box.

Once everybody is comfortable with all of the steps, start the timer and have participants build 3 cards each by completing each step to completion before moving on to the next step; this is known as batch & queue. Stop production about half way through and ask everybody what would happen if we decided to change the color of the paper. How much wasted effort would there be? How does this map to software? Let production continue and note the time when the first card is delivered to the customer and again when all cards are complete.

Run the process again. This time, have participants complete a card before moving on to the next; this is known as single piece flow. Again, stop production about half way through and ask the same questions as before. Let production continue and compare the times with the first method. Obviously, the second method is much faster at getting something to the customer, but more surprisingly, the second method is also faster over all.

Discuss why this is; if the participants say that it is because they have become more efficient, then run it again with the first method and challenge them to beat their time.

Learning Points:

- By taking a smaller set of requirements all the way to completion, you get something to the customer faster. Conversely, if all the requirements are processed at the same time, changes later in the cycle become more costly.

- Single piece flow is often faster than batch and queue. This is due to the fact that each cross-functional participant can take ownership of a module all the way to completion, reducing overall task-switching and hand-offs.

Exercise – Football Scrum

Timing: 15 mins

Ingredients:

- Football (or some other kind of ball)

Directions:

In order to enforce the rules of the daily stand-up meeting, a football can be used. Only those holding the football can speak. Once they have completed answering their questions, they can throw the football to another team member who has not yet spoken. This continues until all members have spoken. It is up to each individual to remember who has not spoken. You can implement penalties for violations of the rules (scoreboard, $1 to the happy hour or lunch fund, etc.)

Learning Points:

- Tracking who has not yet spoken and the expectation of receiving the football keeps everybody alert and involved.
- Only one member speaks at a time.

Exercise – You are in control (not)

Timing: 30-45 mins

Ingredients:

- Four unique paper airplane instructions, one set for each team.
- A big stack of standard size white papers and a smaller stack of yellow papers.

Directions:

In teams of 4 or more, have participants create as many paper airplanes as possible. When thrown from behind a table at one end of the room, airplanes must cross the room and touch the opposite wall. The facilitator, playing the role of the customer, can reject any planes that do not meet their quality standards. Track the number of planes created/approved, time to get the first plane approved, time to absorb a new team member, time to incorporate a new requirement (first yellow plane) First pass: Self organizing and cross functional teams No roles or responsibilities. No prep time. Provide each team one paper airplane instruction, the

same for all teams. Go! Second pass: command and control with specialists. Create new teams. Team members may only perform one function:

- Folder: Can fold paper so that the surfaces remain in contact.
- Bender: Can bend the paper into a new angle, so long as the surfaces are not touching (this would be the job of the Folder).
- Pilot: May do final adjustments to elevators.

A team member may change their role; however they must leave for 1 minute to attend training. A Project Manager is in charge of overall quality. They must establish the steps and tasks for each team member. Give the team 5 minutes to prepare their process. Go! Run exercise for 5-10 minutes for each pass. After two-four minutes, swap one team member from each team, After three - six minutes, put in a special order for yellow planes. Offer bonus of ten points per yellow plane.

Learning Points:

- Self organizing and cross functional teams are better able to adapt to changes in than those driven by command and control with specialists.
- In addition, they are faster at getting to market and more productive.

Exercise – The Story of Sprints

Timing: 10 mins

Ingredients:

- People and space
- 1 stop watch
- Optionally, something to record the audio with.

Directions:

Have the team sit/stand in a circle. You want to get a story of the last sprint that is told by the entire team. You start by saying 'Once upon a time , we had a X (insert sprint length here!) week sprint...'. Then, the next person to your left adds to your sentence and this carries on until the last person has spoken or if the story is developing in an interesting direction, until all the points appear to have been made and there is nothing of value coming through. You might want to strictly enforce the time limit for a large team.

After the retrospective, you could run the game again to tell the story of the next sprint, and this should galvanize the improvements that will take place and nicely summarize the lessons learnt and help the team visualize how the next sprint could be better. This game helps to create an ongoing shared goal and represents an oral history of the software process.

Learning Points:

- Discover a consensus view of the success/failures from the last sprint.
- Empower everyone to add value to a collective goal through participation.
- Exercise the 'responding to change' learning point from the word-at-a-time letter game.

Exercise – Ball Point Game

Playing the Game
In order to play the Ball Point game, you'll need a large open space with enough room for everyone to stand. You'll also need about 20 brightly colored tennis balls and you may want a whiteboard to do the debriefing. We play the game in the following way:

1. Provide an overview of the game and the rules.
    - Everyone is part of one big team.
    - Each ball must have air-time.
    - Each ball must be touched at least once by every team member.
    - Balls cannot be passed to your direct neighbor to your immediate left or right.
    - Each ball must return to the same person who introduced it into the system.
    - There are a total of five iterations.
2. Allow the team two minutes of preparation time to determine how they will organize themselves.
3. Get an estimate from the team of how many balls they can pass through the system.
4. Run a two-minute iteration.
5. Allow the team one minute to discuss how to improve the process.
6. Repeat for five iterations. Make the fifth iteration a challenge.

The objective of the Ball Point game is to get as many balls through the team as possible within two minutes. Each ball must be touched at least once by every team member and must end with the same person with whom it began. After two minutes the team is allowed an additional minute to discuss the process and how it could be improved. The game is played a total of five times.

Initially, the Scrum Trainers will have difficulty getting a single ball through the system, but, after adapting our process, the interactions became more predictable and the participants started to make visible progress. It usually takes four cycles for participants to reach a point where they were both productive and having fun.

The fifth (and final) iteration is a challenge Sprint. The Trainer challenges the participants to get as many balls through the system as possible. Usually the participants will reorganize themselves.

Learning: All systems have a natural velocity and if we want to increase this natural velocity, we need to change the boundaries of the system.

Exercise – Spec Writer
Duration: 30 minutes

In this team-based exercise, each team is divided into “Developers” and “Spec-writers.” The “Developers” are separated from the “Spec-writers” and only allowed to communicate using written specifications. “Spec-writers” are then presented with a diagram that they need to communicate to the “Developers,” who, in turn, must interpret the written specifications and reproduce the diagram. The exercise is run twice with two different diagrams and a retrospective is held at the end of each run.

1. Find two “Developers” in each team. (I usually do that by asking who is a PM or non-developer since I want them to feel how hard it can be to understand a specification.)
2. The rules of the exercise are:
    - a) The originals cannot leave the room.
    - b) Specifications must be written. Diagrams, symbols and numbers are not permitted.
    - c) As many specifications as desired can be delivered, as often as desired.
    - d) The only allowed communication is for “Spec-writers” to hand over the written specifications to developers.
    - e) “Spec-writers” can look at what the “Developers” are doing, but not communicate verbally or with body language.
3. Move the “Developers” out of the room and place them so that the “Developer” teams cannot see what other teams are doing.
4. Distribute copies of the first original to the “Spec-writers.” It’s important that the “Developers” don’t see the original.
5. Run the exercise. Often “Spec-writers” have a hard time not communicating with the “Developers” with body language, so pay attention to this.
6. After 12 minutes stop the teams and collect the results. Show them to class.
7. Let the teams do a retrospective under the given rules.
8. Run the exercise again with the second original.

Diagrams: http://kanemar.files.wordpress.com/2008/03/oppgaveomkravformidling1.pdf

Learning Points: It show the difficulties that the “Developers” have understanding the written specifications and the importance of verbal communication in communicating a vision. It also demonstrates some of the difficulties that teams have when there is a separation between the “Spec-writers” and the “Developers.”

The separation of “Spec-writers” and “Developers” greatly reduces the team’s performance.

Exercise – Play word association

The goal of the game is to provoke thought. This is not a test, not a test of our skill.

Given the list of the words on the left, find the most appropriate match on the right. Since this a game, even though both “ATDD” and “TDD” are listed on the left, we won’t find “Testing” anywhere on the right.

ATDD	Bad
Backlog	Coordinate

Daily Stand-up / Daily Scrum	Design
Product Owner	Facilitate Collaboration
Scrum Master	Iteration
Sprint	Specify
TDD	To Do List
WIP	Trade-offs

References

1. Extreme Programming Explained - Kent Beck, Cynthia Andres
2. Enterprise and Scrum – Ken Schwaber
3. User Stories Applied – Mike Cohn
4. Maverick & Seven day weekend - Ricardo Semler
5. The Agile Samurai - Jonathan Rasmusson
6. Agile Testing - Lisa Crispin and Janet Gregory
7. https://www.scruminc.com/scrum-blog/
8. www.mountaingoatsoftware.com
9. www.scrumalliance.org
10. www.agilealliance.org
11. www.agilemodeling.com
12. www.infoq.com/presentations/agile-quality-canary-coalmine
13. Scrum - http://www.itconversations.com/audio/download/itconversations-350.mp3
14. Developer Testing - http://www.itconversations.com/audio/download/itconversations-301.mp3
15. Lean Primer - http://www.leanprimer.com
16. Agile process with offshore development - www.martinfowler.com/articles/agileOffshore.html#ExpectToNeedMoreDocuments
17. Roots of Scrum - http://www.infoq.com/presentations/The-Roots-of-Scrum
18. Value in Agile vs. Traditional - http://www.youtube.com/watch?v=OWvSnYjqOTQ
19. http://www.infoq.com/minibooks/agile-patterns
20. Introduction to Agile: www.infoq.com/presentations/role-of-testing-in-agile-scott-ambler
21. Estimation in Agile projects: www.youtube.com (search for “Mike Cohn Estimation”)
22. Pretending to be Agile: http://www.informit.com/articles/printerfriendly.asp?p=25913&rl=1
23. Agile Puzzle: http://www.netobjectives.com/podcasts/last20070509_podcasts.mp3
24. Introduction to Lean and DSDM - http://www.infoq.com/presentations/agile-styles-lean-dsdm
25. Agile Leadership - http://www.infoq.com/presentations/agile-leadership-tim-lister
26. Agile Retrospectives – http://www.infoq.com/presentations/Heartbeat-Retrospectives-Boris-Gloger
27. Discipline in Agile
    a. https://www.ibm.com/developerworks/community/blogs/ambler/entry/bureaucracy_isn_t_discipline?lang=en
    b. http://www.ddj.com/article/printableArticle.jhtml?articleID=201804241&dept_url=/architect/
    c. http://www.ddj.com/article/printableArticle.jhtml?articleID=207401013&dept_url=/architect/

28. Distributed team working together. Eclipse project over 7 years.
    http://www.infoq.com/presentations/Eclipse-Lessons-Erich-Gamma
29. Tips for successful agile transitions –
    http://www.infoq.com/presentations/10-tips-for-agile-transitions
30. Continuous Improvement - http://www.infoq.com/presentations/kanban-for-software
31. Leadership - http://www.infoq.com/presentations/agile-cultures
32. Self Organizing teams - http://www.infoq.com/presentations/scaling-up-by-scaling-down
33. Defined vs. Empirical - http://www.ted.com/talks/dan_pink_on_motivation.html
34. Distributed Scrum - http://www.infoq.com/presentations/fully-distributed-scrum
35. Agile Metrics - http://www.infoq.com/presentations/agile-project-metrics
36. Scrum Checklist - http://www.crisp.se/scrum/checklist
37. Free online Scrum tools - http://www.scrumexpert.com/tools/free-online-scrum-tools/
38. Nexus for large scale Scrum - https://www.youtube.com/watch?v=wuc3NPsL844

The End.
Please send your feedback on this document to `Vijay_nathani@yahoo.com`.
Thank you.