Agile Processes

The weather-cock on the church spire, though made of iron, would soon be broken by the storm-wind if it did not understand the noble art of turning to every wind.

-- Heinrich Heine

Many of us have lived through the nightmare of a project with no process to guide it. The lack of a process leads to unpredictability, repeated error, and wasted effort. Customers are disappointed by slipping schedules, growing budgets, and poor quality. Developers are disheartened by working ever longer hours to produce ever poorer software.

Once we have experienced such a fiasco, we become afraid of repeating the experience. A common response to fear is to create a process that we believe eliminates what we are afraid of. We are afraid that

- The project will produce the wrong product.
- The project will produce a product of inferior quality.
- The project will be late.
- We'll have to work 80 hour weeks.
- We'll have to break commitments.
- We won't be having fun.

Our fears motivate us to create a process that constrains our activities and demands certain outputs and artifacts. We draw these constraints and outputs from past experience, choosing things that appeared to work well in previous projects. Our hope is that they will work again, and take away our fears.

But projects are not so simple that a few constraints and artifacts can reliably prevent error. As errors continue to be made, we diagnose those errors and put in place even more constraints and outputs in order to prevent those errors in the future. After many projects we may find ourselves overloaded with a huge cumbersome process that greatly impedes our ability to get projects done.

A big cumbersome process can create the very problems that it is designed to prevent. It can slow the team to the extent that schedules slip and budgets bloat. It can reduce responsiveness of the team to the point that they are always creating the wrong product. Unfortunately this leads many teams to believe that they don't have enough process. So, in a kind of runaway process inflation, they make their process ever larger.

Runaway process inflation is a good description of the state of affairs of the software industry circa 2000 A.D. Though there were still many teams operating without a process. The adoption of very large heavyweight processes was rapidly growing; especially in large corporations.

The Agile Alliance

In early 2001, motivated by the observation that software teams in many corporations were stuck in a quagmire of ever increasing process, a group of industry experts met to outline the values and principles that would allow software teams to develop quickly and respond to change. They called themselves *the Agile Alliance* . Over two days they worked to create a statement of values. The result was the manifesto of the Agile Alliance. Over the next three months they continued to work together to create the principles of agility.

The Manifesto: a statement of agile values

<u>Manifesto for Agile Software Development</u>

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.

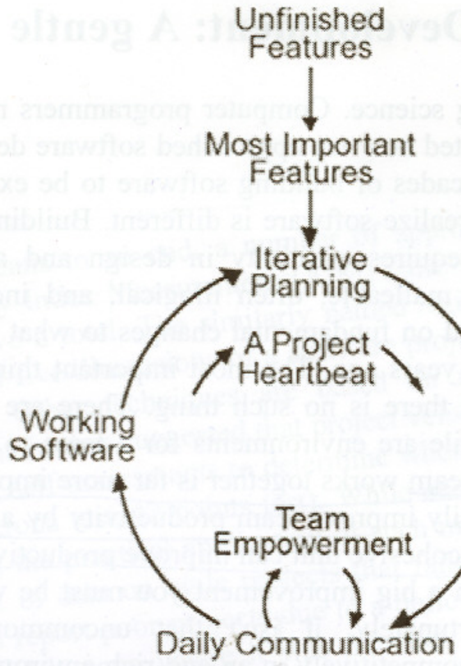
Agile Software Development: A gentle introduction

Computer science is a young science. Computer programmers my age were trained by engineers. That training dictated how we approached software development for an entire generation. But now after decades of building software to be expensive, unwanted, and unreliable we have come to realize software is different. Building software is more like creating a work of art, it requires creativity in design and ample craftsmanship to complete. Software remains malleable, often illogical, and incomplete forever. Agile software development is based on fundamental changes to what we considered essential to software development ten years ago. The most important thing to know about Agile methods or processes is that there is no such thing. There are only Agile teams. The processes we describe as Agile are environments for a team to learn how to be Agile.

We realize the way a team works together is far more important than any process. While a new process can easily improve team productivity by a fraction, enabling your team to work effectively as a cohesive unit can improve productivity by several times. Of course to be eligible for such a big improvement you must be working at a fraction of your potential now. Unfortunately, it isn't that uncommon. The most brilliant programmers alive working competitively in an ego-rich environment can't get as much done as ordinary programmers working cooperatively as a self disciplined and self-organizing team. You need a process where team empowerment and collaboration thrive to reach your full potential. The second change is making the customer, the one who funds the software development, a valuable and essential team member. When the dead line gets close a traditional approach to reducing scope is to let the developers decide what will work properly and what won't. Instead let the customer make scope decisions a little at a time throughout the project. When your customer, or domain expert works directly with the development team everyone learns something new about the problem. True domain expertise and experience is essential to finding a simple, elegant, correct solution. A document can have plenty of information, but real knowledge is hard to put on paper. Left alone programmers must assume they know everything they need. When asking questions is difficult or slow the knowledge gap grows. The system will get built, but it won't solve the problem like one guided by an expert on a daily basis.

Perhaps the biggest problem with software development is changing requirements. Agile processes accept the reality of change versus the hunt for complete, rigid specifications. There are domains where requirements can't change, but most projects have changing requirements. For most projects readily accepting changes can actually cost less than ensuring requirements will never change.

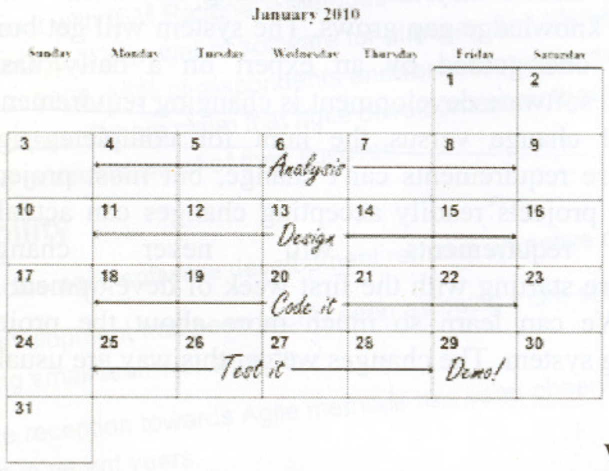
We can produce working software starting with the first week of development so why not show it to the customer? We can learn so much more about the project requirements in the context of a working system. The changes we get this way are usually the most important to implement.



Agile also means a fundamental change in how we manage our projects. If working software is what you will deliver then measure your progress by how much you have right now. We will change our management style to be based on getting working software done a little at a time. The documents we used to create as project milestones may still be useful, just not as a measure of progress.

Instead of managing our activities and waiting till the project ends for software, we will manage our requirements and demonstrate each new version to the customer. It is a hard change to make but it opens up new ways to develop software.

Manage Your Goals Instead of Activities



With a traditional process we schedule our activities one after another until the project end date. Design always starts when analysis is finished. Coding starts when the design is done, etc. We make a plan to get everything finished on time. At the end of the schedule we demonstrate our new software and see if the customer likes it.

Agile Process Proverbs

Here are some things to think about. I have summarized them into pithy sentences, added bullet points and called them proverbs to make them sound important. Don't be fooled. These are ideas, not rules. Ideas to help you think about making your current process, what ever it is, a more Agile process.

Iterative Planning

- The best way to meet customer objectives is to explicitly schedule their completion.
- Trying to catch up is the fastest way to get further behind.
- The changes you get late in a project are very valuable because you paid the most to get them.
- One third of your requirements will represent two thirds of the project's value.
- Showing someone something they can change is helpful, showing something they can't change is spiteful.
- Iterative development gives you a few "oh drat"s during development instead of one big "AW SHIT" at the end.

Honest Plans

- An estimate not based on a measurement is a guess; an estimate based on a measurement is a prediction.
- He who does the work sets the estimate.
- If software is what you want to deliver than measure progress by how much you have working right now.

Team Empowerment

- The most important thing on a project is good leadership; the least important thing is who leads.
- If you can see a need for leadership you know enough to lead.
- When you only have responsibility you will know worry, but when you also have authority you will know opportunity.
- The organization of your team will be reflected in the code. (Conway's Law)
- Calm and relaxed plus confident yields decisive.
- Count how many people are on the project, now communicate like a group that size.
- Hermits don't share, communities are based on sharing.

Documentation/Models

- Agile models are paintings, not photographs.
- The creation of a document is an implicit agreement to keep it up to date or destroy it.
- Without the use they are just cases.
- Ideas move faster than documents.

Customers/Product Owners

- There is more to owning software than just paying for it.
- Negotiating is basic to being human, don't give it up that right before the project even begins.

Managers/Scrum Masters

- Challenge your team intellectually or they will challenge themselves in ways you wouldn't have chosen.
- Demanding estimates change seems like taking control, but you lose control by not

making a decision.

- People don't scale, teams do.
- The only way to guarantee bad decisions is to make all of them yourself.

Developers/Team Members

- Big changes cause big problems.
- Egoless programming doesn't work; expand your ego to include everyone's code.
- Don't guess, measure it.
- Don't theorize, try it.
- Everything you did today can be done over tomorrow in half an hour and be better.
- Measure twice cut once saves wood, but software isn't made of wood.
- The best way to get help is to offer help.

Testing

- The first test is the hardest.
- Your test suite is more valuable than your code.
- The harder the test is to create the greater your savings.
- Where there is a will there is a way to test.
- If you want a good suite of tests next year you must start collecting them today.

Process

- If you don't use your process it can't help you; if your process doesn't help, you won't use it.
- The one essential ingredient that turns repetitive development into iterative development is feedback.
- If less doesn't work try more, if more doesn't work try less, if neither works stop doing it.
- If 10 people slow the project down by just 10% each it will take twice as long.
- If it isn't fun you're doing something wrong.
- A hammer without a good process is a sore thumb waiting to happen.
- Process is something you do, not something you do to someone.
- The person least likely to save time is the person in the biggest hurry.
- The least disciplined team member has the greatest control over your process.

Simplicity

- Simple is subjective so judge it subjectively as testable, browsable, understandable, and explainable.
- The best way to guarantee it costs too much is to add something you don't need.
- If it takes you too long to refactor than you are not refactoring enough.
- A simple solution always takes less time than a complex one.
- If you always try the simplest thing next you will always work as simply as you can.

Design

- Who ever finds a problem knows enough to design a solution.
- Flexibility isn't building for every imaginable possibility; it's having as little to change as possible.
- A good design can be explained to someone else using four blank cards.
- A design which doesn't meet business needs is bad, no matter how pretty.
- UML was designed to document complexity, not expose it.

all unit tests to pass (100%) before any new code is released into the code repository.

If one or two developers have become bottlenecks because they own the core classes in the system and must make all the changes, then try collective code ownership. (You will also need unit tests.) Let everyone make changes to the core classes whenever they need to.

You could continue this way until no problems are left. Then just add the remaining practices as you can. The first practice you add will seem easy. You are solving a large problem with a little extra effort. The second might seem easy too. But at some point between having a few XP rules and all of the XP rules it will take some persistence to make it work. Your problems will have been solved and your project is under control. It might seem good to abandon the new methodology and go back to what is familiar and comfortable, but continuing does pay off in the end. Your development team will become much more efficient than you thought possible. At some point you will find that the XP rules no longer seem like rules at all. There is a synergy between the rules that is hard to understand until you have been fully immersed.

This up hill climb is especially true with pair programming, but the pay off of this technique is very large. Also, unit tests will take time to collect, but unit tests are the foundation for many of the other XP practices so the pay off is very great.

XP projects are not quiet; there always seems to be someone talking about problems and solutions. People move about, asking each other questions and trading partners for programming. People spontaneously meet to solve tough problems, then disperse again. Encourage this interaction, provide a meeting area and set up workspaces such that two people can easily work together. The entire work area should be open space to encourage team communication.

WATERFALL vs. AGILE METHODOLOGY

There is no IT meeting that does not talk and debate endlessly about Waterfall vs. Agile development methodologies. Feelings run strong on the subject with many considering **Agile** *'so of the moment'*, just so right, while **Waterfall** is thought to be passé! But, before deciding which is more appropriate, it is essentially important to provide a little background on both.

Waterfall

A classically linear and sequential approach to software design and systems development, each waterfall stage is assigned to a separate team to ensure greater project and deadline control, important for on-time project delivery. A linear approach means a stage by stage approach for product building, e.g.

1. The project team first **analyses,** then determining and prioritisingbusiness requirements / needs.

2. Next, in the **design phase** business requirements are translated into IT solutions, and a decision taken about which underlying technology i.e. COBOL, Java or Visual Basic, etc. etc. is to be used.

3. Once processes are defined and online layouts built, code**implementation** takes place.

4. The next stage of data conversion evolves into a fully **tested** solution for implementation and testing for evaluation by the end-user.

5. The last and final stage involves **evaluation** and **maintenance,** with the latter ensuring everything runs smoothly.

However, in case a glitch should result, changing the software is not only a practical impossibility, but means one has to go right back to the beginning and start developing new code, all over again. That's Waterfall for you! Now, as for minimal risk Agile, it is a low over-head method that emphasizes values and principles rather than processes. Working in cycles i.e. a week, a month, etc., project priorities are re-evaluated and at the end of each cycle. Four principles that constitute Agile methods are:

1. The reigning supreme of individuals and interactions over processes and tools.

2. As does, working software over comprehensive documentation.

3. Likewise, customer collaboration over contract negotiation.

4. And again, responding to change over plan follow-throughs.

To synopsise the difference between the two, one can say the classic waterfall method stands for predictability, while Agile methodology spells adaptability. Agile methods are good at reducing overheads, such as, rationale, justification, documentation and meetings, keeping them as low as is possible. And, that is why Agile methods benefit small teams with constantly changing requirements, rather more than larger projects.

Agile, based on empirical rather than defined methods (Waterfall) is all about light maneuverability and sufficiency for facilitating future development. By defined methods what one means is that one plans first and then enforces these plans. However, Agile methods involve planning what one wants and then adapting these plans to the results. Extreme Programming (XP) is an excellent example of Agile methodology i.e.:

1. Communication between customers and other team members;

2. Simple, clean designs;

3. Feedback given on Day 1 of software testing;

4. Early delivery and implementation of suggested changes.

Agile methodology means cutting down the big picture into puzzle size bits, fitting them together when the time is right e.g. design, coding and testing bits. So, while there are reasons to support both the waterfall and agile methods, however, a closer look clarifies why many software and web design firms make the more appropriate choice of employing Agile methodology. The following table enumerates the raison d'être for choosing Agile methodology over the Waterfall method.

1. Once a stage is completed in the **Waterfall method,** there is no going back, since most software designed and implemented under the waterfall method is hard to change according to time and user needs. The problem can only be fixed by going back and designing an entirely new system, a very costly and inefficient method. Whereas, **Agile methods** adapt to change, as at the end of each stage, the logical programme, designed to cope and adapt to new ideas from the outset, allows changes to be made easily. With Agile, changes can be made if necessary without getting the entire programme rewritten. This approach not only reduces overheads, it also helps in the upgrading of programmes.

2. Another **Agile method** advantage is one has a launchable product at the end of each tested stage. This ensures bugs are caught and eliminated in the development cycle, and the product is double tested again after the first bug elimination. This is not possible for the **Waterfall method,** since the product is tested only at the very end, which means any bugs found results in the entire programme having to be re-written.

3. **Agile's** modular nature means employing better suited object-oriented designs and programmes, which means one always has a working model for timely release even when it does not always entirely match customer specifications. Whereas, there is only one main release in the waterfall method and any problems or delays mean highly dissatisfied customers.

4. Agile methods allow for specification changes as per end-user's requirements, spelling customer satisfaction. As already mentioned, this is not possible when the waterfall method is employed, since any changes to be made means the project has to be started all over again.

5. However, both methods do allow for a sort of departmentalization e.g. in waterfall departmentalization is done at each stage. As for Agile, each coding module can be delegated to separate groups. This allows for several parts of the project to be done at the same time, though departmentalization is more effectively used in Agile methodologies.

In conclusion, though on the plus side, waterfall's defined stages allow for thorough planning, especially for logical design, implementation and deployment, Agile methodology is a sound choice for software development and web design projects. More and more firms are becoming Agile!

Introduction to Scrum - An Agile Process

What is Scrum?

Scrum is an agile approach to software development. Rather than a full process or methodology, it is a framework. So instead of providing complete, detailed descriptions of how everything is to be done on the project, much is left up to the software development team. This is done because the team will know best how to solve the problem they are presented. This is why, for example, a sprint planning meeting is described in terms of the desired outcome (a commitment to set of features to be developed in the next sprint) instead of a set of Entry criteria, Task definitions, Validation criteria, and Exit criteria (ETVX) as would be provided in most methodologies.

Scrum relies on a self-organizing, cross-functional team. The scrum team is self-organizing in that there is no overall team leader who decides which person will do which task or how a problem will be solved. Those are issues that are decided by the team as a whole. The team is cross-functional so that everyone necessary to take a feature from idea to implementation is involved.

These agile development teams are supported by two specific individuals: a ScrumMaster and aproduct owner. The ScrumMaster can be thought of as a coach for the team, helping team members use the Scrum framework to perform at their highest level. The product owner represents the business, customers or users and guides the team toward building the right product.
Scrum projects make progress in a series of sprints, which are timeboxed iterations no more than a month long. At the start of a sprint, team members commit to delivering some number of features that were listed on the project's product backlog. At the end of the sprint, these features are *done*--they are coded, tested, and integrated into the evolving product or system. At the end of the sprint a sprint review is conducted during which the team demonstrates the new functionality to the product owner and other interested stakeholders who provide feedback that could influence the next sprint.

What are the main activities in Scrum?

The sprint itself is the main activity of a Scrum project. Scrum is an iterative and incremental process and so the project is split into a series of consecutive sprints. Each is timeboxed, usually to between one week and a calendar month. A recent survey found that the most common sprint length is two weeks. During this time the team does everything to take a small set of features from idea to coded and tested functionality.

The first activity of each sprint is a sprint planning meeting. During this meeting the product owner and team talk about the highest-priority items on the product backlog. Team members figure out how many items they can commit to and then create a sprint backlog, which is a list of the tasks to perform during the sprint.
On each day of the sprint, a daily scrum meeting is attended by all team members, including the ScrumMaster and the product owner. This meeting is timeboxed to no more than fifteen minutes. During that time, team members share what they worked on the prior day, will work on today, and identify any impediments to progress. Daily scrums serve to synchronize the work of team members as they discuss the work of the sprint.
At the end of a sprint, the teams conducts a sprint review. During the sprint review, the team demonstrates the functionality added during the sprint. The goal of this meeting is to get feedback from the product owner or any users or other stakeholders who have been invited to the review. This feedback may result in

changes to the freshly delivered functionality. But it may just as likely result in revising or adding items to the product backlog.
Another activity performed at the end of each sprint is the sprint retrospective. The whole team participates in this meeting, including the ScrumMaster and product owner. The meeting is an opportunity to reflect on the sprint that is ending and identify opportunities to improve in the new sprint.

What are the main artifacts of a Scrum project?

The primary artifact of a Scrum project is, of course, the product itself. The team is expected to bring the product or system to a potentially shippable state at the end of each sprint.

The product backlog is a complete list of the functionality that remains to be added to the product. The product backlog is prioritized by the product owner so that the team always works on the most valuable features first. The most popular and successful way to create a product backlog is to populate it with user stories, which are short descriptions of functionality described from the perspective of a user or customer.
On the first day of a sprint and during the sprint planning meeting, team members create the sprint backlog. The sprint backlog can be thought of as the team's to-do list for the sprint. Whereas a product backlog is a list of features to be built (often written in the form of user stories), the sprint backlog is the list of tasks the team needs to perform in order to deliver the functionality they committed to deliver during the sprint.
Two other primary artifacts are the sprint burndown chart and release burndown chart. Burndown charts show the amount of work remaining either in a sprint or a release. They are a very effective tool for determining at a glance whether a sprint or release is on schedule to have all planned work finished by the desired date.

Main roles on a Scrum team

Even if you are new to Scrum, you may have heard of a role called ScrumMaster. The ScrumMaster is the team's coach and helps team members achieve their highest level of performance. A ScrumMaster differs from a traditional project manager in many key ways, including that the ScrumMaster does not provide day-to-day direction to the team and does not assign tasks to individuals. A good

ScrumMaster shelters the team from outside distractions, allowing team members to focus maniacally during the sprint on the goal they have selected. While the ScrumMaster focuses on helping the team be the best that it can be, the product ownerworks to direct the team at the right goal. The product owner does this by creating a compelling vision of the product and then conveying that vision to the team through the product backlog.
The product owner is responsible for ensuring that the product backlog remains prioritized as more is learned about the system being built, its users, the team, and so on.

The third and final role on a Scrum project is the team itself. Although individuals on a Scrum team may come to that team with various job titles, while on the team those titles are insignificant. Each person contributes in whatever ways they best can to complete the work of each sprint. This does not mean that a tester will be expected to rearchitect the system; individuals will spend most (and sometimes all) of their time working in whatever discipline they worked before adopting Scrum. But on a Scrum team, individuals are expected to work beyond their preferred disciplines whenever doing so would be for the good of the team.

One convenient way to think of the interlocking nature of these three roles is as a race car. The team is the car itself, ready to speed along in whatever direction it is pointed. The product owner is the driver, making sure that the car is always going in the right direction. The ScrumMaster is the chief mechanic, keeping the car well-tuned and performing at its best.

Essentials of Scrum

- Overview of Scrum
- Daily scrum
- Product backlog
- Product owner
- Release burndown
- ScrumMaster
- Scrum team
- Sprint backlog
- Sprint planning meeting
- Sprint review meeting
- Using a task board
- Scrum illustrations and wallpapers
- A Presentation on Scrum

A typical scrum team has between five and nine people, but Scrum projects can easily scale into the hundreds. Scrum can easily be used by one-person teams and often is. The team does not include any of the traditional software engineering roles such as programmer, designer, tester, or architect. Everyone on the project works together to complete the set of work they have collectively committed to complete within a sprint. Scrum teams develop a deep form of camaraderie and a feeling that "we're all in this together."

The product owner is the project's key stakeholder and represents users, customers and others in the process. The product owner is often someone from product management or marketing, a key stakeholder or a key user.

The ScrumMaster is responsible for making sure the team is as productive as possible. The ScrumMaster does this by helping the team use the Scrum process, by removing impediments to progress, by protecting the team from outside, and so on.

The product backlog is a prioritized features list containing every desired feature or change to the product.

At the start of each sprint, a sprint planning meeting is held during which the product owner presents the top items on the product backlog to the team, and the Scrum team selects the work they can complete during the coming sprint. That work is then moved from the product backlog to a sprint backlog, which is the list of tasks needed to complete the product backlog items the team has committed to complete in the sprint.

Each day during the sprint, a brief meeting called the daily scrum is conducted. This meeting helps set the context for each day's work and helps the team stay on track. All team members are required to attend the daily scrum.

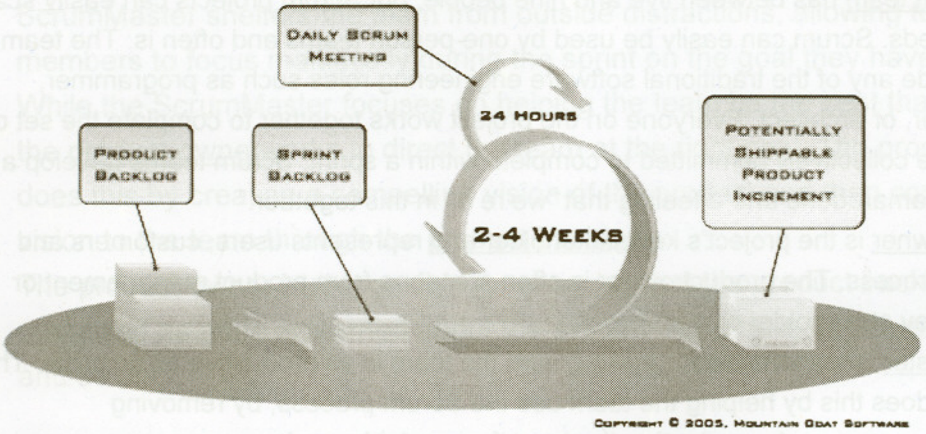
At the end of each sprint, the team demonstrates the completed functionality at a sprint review meeting, during which, the team shows what they accomplished during the sprint. Typically, this takes the form of a demonstration of the new features, but in an informal way; for example, PowerPoint slides are not allowed. The meeting must not become a task in itself nor a distraction from the process.

Also at the end of each sprint, the team conducts a sprint retrospective, which is a meeting during which the team (including its ScrumMaster and product owner) reflect on how well Scrum is working for them and what changes they may wish to make for it to work even better.

Note

The term "backlog" can get confusing because it's used for two different things. To clarify: the product backlog is a list of desired features for the product. The sprint backlog is a list of tasks to be completed in a sprint.

Graphically, Scrum looks something like this:


Daily Scrum Meeting
24 Hours
Product Backlog
Sprint Backlog
Potentially Shippable Product Increment
2-4 Weeks
Copyright © 2005, Mountain Goat Software

This graphic is an introduction to the essential elements of using Scrum for agile software development. On the left, we see the product backlog, which has been prioritized by the product owner and contains everything desired in the product that's known at the time. The 2-4 week sprints are shown by the larger green circle.

At the start of each sprint, the team selects some amount of work from the product backlog and commits to completing that work during the sprint. Part of figuring out how much they can commit to is creating the sprint backlog, which is the list of tasks (and an estimate of how long each will take) needed to deliver the selected set of product backlog items to be completed in the sprint.

At the end of each sprint, the team produces a potentially shippable product increment — i.e. working, high-quality software. Each day during the sprint, team members meet to discuss their progress and any impediments to completing the work for that sprint. This is known as the daily scrum, and is shown as the smaller green circle above.

Agile Unified Process (AUP) is a simplified version of the IBM Rational Unified Process (RUP) developed by Scott Ambler.[1] It describes a simple, easy to understand approach to developing business application software using agile techniques and concepts yet still remaining true to the RUP. The AUP applies agile techniques including test driven development (TDD), Agile Modeling, agile change management, and database refactoring to improve productivity.

Disciplines

Unlike the RUP, the AUP has only seven disciplines:

1. **Model**. Understand the business of the organization, the problem domain being addressed by the project, and identify a viable solution to address the problem domain.
2. **Implementation**. Transform model(s) into executable code and perform a basic level of testing, in particular unit testing.
3. **Test**. Perform an objective evaluation to ensure quality. This includes finding defects, validating that the system works as designed, and verifying that the requirements are met.
4. **Deployment**. Plan for the delivery of the system and to execute the plan to make the system available to end users.
5. **Configuration Management**. Manage access to project artifacts. This includes not only tracking artifact versions over time but also controlling and managing changes to them.
6. **Project Management**. Direct the activities that take place within the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.), and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget.
7. **Environment**. Support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.

Philosophies

The Agile UP is based on the following philosophies:

1. **Your staff knows what they're doing**. People are not going to read detailed process documentation, but they will want some high-level guidance and/or training from time to time. The AUP product provides links to many of the details, if you are interested, but doesn't force them upon you.
2. **Simplicity**. Everything is described concisely using a handful of pages, not thousands of them.
3. **Agility**. The Agile UP conforms to the values and principles of the agile software development and the Agile Alliance.
4. **Focus on high-value activities**. The focus is on the activities which actually count, not every possible thing that could happen to you on a project.
5. **Tool independence**. You can use any toolset that you want with the Agile UP. The recommendation is that you use the tools which are best suited for the job, which are often simple tools.
6. **You'll want to tailor the AUP to meet your own needs**.

Releases