# 1

# 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.*

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

**Individuals and interactions over processes and tools.** People are the most important ingredient of success. A good process will not save the 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 that raw programming talent. A team of average programmers who communicate well are more likely to succeed than a group of superstars who fail to interact as a team.

The right tools can be very important to success. Compilers, IDEs, source code control systems, etc., 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.

My advice is to start small. Don't assume you've outgrown a tool until you tried it and found 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 CASE tools, use white boards 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.

**Working software over comprehensive documentation.** 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 their 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, then they turn into huge lies, and become a significant source of misdirection.

I have no problem with a short rationale and structure document that the team produces and keeps in sync from month to month. But I want that document to be short and

salient. 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 hard 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 their heads. There is no way to put that roadmap down on paper and transfer it to others that is faster and more efficient than interaction with the team.

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.*

**Customer collaboration over contract negotiation.** 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 the managers of a company to tell their development staff what their needs are, and then expect that staff to go away for awhile and return with a system that satisfies their 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 upon a contract, or a statement of work, the customer of the software works closely with the development team, providing frequent feedback on their 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. The best contracts are those that govern the way the development team and the customer will work together.

As an example of a successful contract, in 1994 I negotiated a contract for a large, multi-year, half-million-line, project. We, the development team, were paid a relatively low monthly rate. Large payouts were made to us when we delivered certain large blocks of functionality. Those blocks were not specified in detail by the contract. Rather the contract stated that the payout would be made for a block when the block passed the customer's acceptance test. The details of those acceptance tests were not specified in the contract.

During the course of this project we worked very closely with the customer. We released the software to him almost every Friday. By Monday of Tuesday of the following week he would have a list of changes for us to put into the software. We would prioritize those changes together, and then schedule them into subsequent weeks. The customer

worked so closely with us that acceptance tests were never an issue. He knew when a block of functionality satisfied his needs because he watched it evolve from week to week.

The requirements for this project were in a constant state of flux. Major changes were not uncommon. There were whole blocks of functionality that were removed, and others that were inserted. And yet the contract, and the project, survived and succeeded. The key to this success was the intense collaboration with the customer; and a contract that governed that collaboration rather than trying to specify the details of scope and schedule for a fixed cost.

**Responding to change over following a plan.** It is the ability to respond to change that often determines the success or failure of a software project. When we build plans, we need to make sure that our plans are flexible and ready to adapt to changes in the business and technology.

The course of a software project cannot be predicted far into the future. There are too many variables to account for. We simply aren't very good at estimating the cost of a large project. The business environment that the software must serve is likely to change during the course of development. It is difficult to write reliable requirements. Customers are likely to alter the requirements once they see the system start to function.

It is tempting for novice managers to create a nice PERT or Ghant chart of the whole project, and tape it to the wall. 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 their 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 just changes in dates.

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

This decreasing resolution of the plan means that we are only investing in a detailed plan for those tasks that are immediate. Once the detailed plan is made, it is hard to change since the team will have a lot of momentum and commitment. But since that plan only governs a few weeks worth of time the rest of the plan remains flexible. The lower resolution parts of the plan can be changed with relative ease.

## Principles

The above values inspired the following twelve principles. These principles are the characteristics that differentiate an agile process.

- *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

  The MIT Sloan Management Review published an analysis of software development practices that help companies build high quality products[1]. The article found a number of practices that had a significant impact upon the quality of the final system. One was a strong correlation between quality, and the early delivery of a partially functioning system. The article reported that the less functional the initial delivery, the higher the quality in the final delivery.

  Another finding of this article was a strong correlation between final quality and frequent deliveries of increasing functionality. The more frequent the deliveries, the higher the final quality.

  An agile process is one that delivers early and often. We strive to deliver a rudimentary system within the first few weeks of the start of the project. And we strive thereafter to continue to deliver systems of increasing functionality every few weeks.

  Customers may choose to put these systems into production if they think they are functional enough. Or they may choose simply to review the existing functionality and report on changes they want made.

- *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

  This is a statement of attitude. The participants in an agile process are not afraid of change. They view changes to the requirements as *good* things, because they mean that the team has learned more about what it will take to satisfy the market.

  An agile team works very hard to keep the structure of their software flexible, so that when requirements change, the impact to the system is minimal. Later in this book we will learn the principles of object oriented design that help us to maintain this kind of flexibility.

- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.*

  We deliver *working* software. And we delivery it early and often. We are not content with delivering bundles of documents, or plans. We don't count those as true deliveries. Our eye is on the goal of delivering software that satisfies the customer's needs.

---

1. *Product-Development Practices That Work: How Internet Companies Build Software*, MIT Sloan Management Review, Winter 2001, Reprint number 4226

- *Business people and developers must work together daily throughout the project.*

   In order for a project to be agile, there must be significant and frequent interaction between the customers, developers, and stakeholders. An agile project is not like a fire-and-forget weapon. An agile project must be continuously guided.

- *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

   An agile project is one in which people are considered the most important factor of success. All other factors, process, environment, management, etc., are considered to be second order effects, and are subject to change if they are having an adverse effect upon the people.

   For example, if the office environment is an obstacle to the team, the office environment changes. If certain process steps are an obstacle to the team, the process steps change.

- *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

   In an agile project, people *talk* to each other. The primary mode of communication is conversation. Documents may be created, but there is no attempt to capture all project information in writing. An agile project team does not demand written specs, written plans, or written designs. They may create them if they perceive an immediate and significant need, but they are not the default. The default is conversation.

- *Working software is the primary measure of progress.*

   Agile projects measure their progress by measuring the amount of software that is working. They don't measure their progress in terms of the phase that they are in, or by the volume of documentation that has been produced, or by the amount of infrastructure code they have created. They are 30% done when 30% of the necessary functionality is working.

- *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

   An agile project is not run like a 50 yard dash; it is run like a marathon. The team does not take off at full speed and try to maintain that speed for the duration. Rather they run at a fast, but sustainable, pace.

   Running too fast leads to burnout, shortcuts, and debacle. Agile teams pace themselves. They don't allow themselves to get too tired. They don't borrow tomorrow's energy to get a bit more done today. They work at a rate that allows them to maintain the highest quality standards for the duration of the project.

- *Continuous attention to technical excellence and good design enhances agility.*

  High quality is the key to high speed. The way to go fast is to keep the software as clean and robust as possible. Thus, all agile team-members are committed to producing only the highest quality code they can. They do not make messes and then tell themselves they'll clean it up when they have more time. If they make a mess, they clean it up before they finish for the day.

- *Simplicity--the art of maximizing the amount of work not done--is essential.*

  Agile teams do not try to build the grand system in the sky. Rather they always take the simplest path that is consistent with their goals. They don't anticipate tomorrow's problems and try to defend against them today. Rather they do the simplest and highest quality work today, confident that it will be easy to change if and when tomorrows problems arise.

- *The best architectures, requirements, and designs emerge from self-organizing teams.*

  An agile team is a self organizing team. Responsibilities are not handed to individual team members from the outside. Rather, responsibilities are communicated to the team as a whole, and the team determines the best way to fulfill them.

  Agile team members work together on all aspects of the project. Each is allowed input into the whole. No single team member is responsible for the architecture, or the requirements, or the tests, etc. The team shares those responsibilities and each team member has influence over them.

- *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

  An agile team continually adjusts its organization, rules, conventions, relationships, etc. An agile team knows that its environment is continuously changing, and knows that they must change with that environment to remain agile.

## Conclusion

The professional goal of every software engineer, and every development team, is to deliver the highest possible value to our employers and customers. And yet, our projects fail, or fail to deliver value, at a dismaying rate. Though well intentioned, the upward spiral of process inflation is culpable for at least some of this failure. The principles and values of agile software development were formed as a way to help teams break the cycle of process inflation, and to focus on simple techniques for reaching their goals.

At the time of this writing there were many agile processes to choose from. These include SCRUM[1], Crystal[2], Feature Driven Development[3], Adaptive Software Development (ADP)[4], and most significantly, Extreme Programming[5].

1.  www.controlchaos.com

2.  crystalmethodologies.org

3.  *Java Modeling In Color With UML: Enterprise Components and Process*, Peter Coad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999

4.  [Highsmith2000]

5.  [BECK99], [Newkirk2001]

# Bibliography

**[BECK99]:** *Extreme Programming Explained: Embracing Change*, Kent Beck, Addison Wesley, 1999.

**[Newkirk2001]:** *Extreme Programming in Practice* , James Newkirk and Robert C. Martin, Addison Wesley, 2001

**[Highsmith2000]:** *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems* , James A. Highsmith, Dorset House, 2000