Agile Technologies

Module 4

Mastering Agility : Values and Principles: Commonalities, About Values, Principles, and Practices ,Further Reading, Improve the Process: Understand Your Project, Tune and Adapt, Break the Rules, Relyon People :Build Effective Relationships, Let the Right People Do the Right Things, Build the Process for the People, Eliminate Waste :Work in Small, Reversible Steps, Fail Fast, Maximize Work Not Done, Pursue Throughput

Values and Principles

To master the art of agile development, you need **experience** and **mindfulness.** Experience helps you see how agile methods work. Mindfulness helps you understand your experiences.

Experience and mindfulness are the path to mastery.

Commonalities

Can any set of principles really represent agile development?

The answer is yes: agile methods do share common values and principles.

Five themes: Improve the Process, Rely on People, Eliminate Waste, Deliver Value, and Seek Technical Excellence.

About Values, Principles, and Practices

XP's values are:

Courage
To make the right decisions, even when they're difficult, and to tell stakeholders the truth when they need to hear it.

Communication
To give the right people the right information when they can use it to its maximum advantage.

Simplicity
To discard the things we want but don't actually need.

Feedback
To learn the appropriate lessons at every possible opportunity.

Respect
To treat ourselves and others with dignity, and to acknowledge expertise and our mutual desire for success.

Principles are applications of those ideals to an industry. For example, the value of simplicity leads us to focus on the essentials of development.

"Excess methodology weight is costly," and "Discipline, skills, and understanding counter process, formality, and documentation."

Practices are principles applied to a specific type of project. XP's practices, for example, call for colocated teams of up to 20 people. "Sit Together"" and "The Whole Team" embody the principles of simplicity because face-to-face communication reduces the need for formal requirements documentation.

Improve the Process

Agile methods are more than a list of practices to follow. When your team has learned how to perform them effectively, you can become a great team by using the practices to modify your process.

As you master the art of agile development, you'll learn how and when to modify your process to take advantage of your specific situation and opportunities.

Understand Your Project

To improve your process, you must understand how it affects your project. You need to take advantage of feedback—from the code, from the team, from customers and stakeholders—so you can understand what works well and what doesn't.

Always pay attention to what's happening around you. Ask "why": why do we follow this practice? Why is this practice working? Why isn't this practice working?

Ask team members for their thoughts. There's an element of truth in every complaint, so encourage open discussion. As a team, reflect on what you've learned. When you discover something new, be a mentor; when you have questions, ask a mentor. Help each other understand what you're doing and why.

In Practice

XP is full of feedback loops of information that you should use to improve your work.

Root-cause analysis and retrospectives clearly improve the team's understanding.

For example, sitting and working together as a whole team gives team members opportunities to observe and absorb information.

When something unexpected happens. Stand-up meetings and the informative workspace contribute to an information-rich environment.

The practices of energized work, slack, and pair programming also spread useful information. When team members are under pressure, they have trouble thinking about ways they can improve their work. Energized work reduce that pressure. Pair programming gives one person in each pair time to think about strategy.

Test-driven development, exploratory testing, real customer involvement, iteration demos, and frequent releases all provide information about the project, from code to user response.

Beyond Practices

Tying an important task to a single person is a mistake. To fix it, we all agreed to divide the work. Every month, one of us takes final responsibility for creating the release; this person produces the final tested bundle, makes the appropriate announcements, and uploads the bundle to the master distribution site.

With six people available and a release schedule planned to the day, we've found a much healthier rhythm for making regular releases. If one or more developers are unavailable, several other people can perform the same function. It's worked—we've regained our velocity and started to attract new developers again.

Tune and Adapt

When you see the need for a change, modify your process. Make the change for your team alone

These changes require tuning. Think of them as experiments; make small, isolated changes that allow you to understand the results. Be specific about your expectations and about the measurements for judging success.

These changes are sources of feedback and learning. Use the results of your experiments to make further changes. Iterate until you're satisfied with the results.

Team members need to be flexible and adaptive. Your team needs to have the courage to experiment and occasionally fail.

In Practice

Tuning and adapting is implicit in XP; teams are supposed to make changes whenever they have a reason to do so.

Rely on People

Alistair Cockburn's 1999 paper, "Characterizing people as non-linear, first-order components in software development," argues that the people involved in making software affect the project as much as any method or practice.

Almost every challenge in building great software is, in some way, a people problem.

Agile methods put people and their interactions at the center of all decisions. How can we best work together? How can we communicate effectively? Successful software projects must address these questions.

Build Effective Relationships

Working relationships are built on honesty, trust, cooperation, openness, and mutual respect.

You can't force people to do this. The best your agile method can do is support these sorts of relationships. For example, one way to engender healthy interaction is to have people sit together and collaborate in pursuit of common goals.

Blame-oriented cultures also sabotage relationships. Get rid of blame by introducing collaboration and avoiding practices that indicate a lack of trust. Rather than forcing stakeholders to sign a requirements document, work together to identify and clarify requirements and review progress iteratively. Rather than telling developers that they can't produce any bugs and testers that they must find all the bugs, ask developers and testers to work together to ensure that customers don't find any bugs.

Credit isn't important. Being right isn't important. Treating your team members with respect and cooperating to produce great software is important.

In Practice

Everyday practices such as stand-up meetings, collective code ownership, ubiquitous language, the planning game, and pair programming help reinforce the idea that team members work together to achieve common goals.

Beyond Practices

XP recommends colocated teams for a reason: it's much easier to communicate and form solid working relationships when you're all in the same room. Yet some teams can't or won't sit together. How do you deal with this challenge?

Our team also instituted weekly meetings. They're virtual meetings, but they help us

understand what everyone is working on. The meetings contribute to our cohesiveness and shared direction, and they help curtail unhelpful tangents.

Let the Right People Do the Right Things

A functioning team is not enough. You need to have the right people working well together.

You need a diverse range of expertise. Once you find the right people, trust them to do their jobs. Instead of creating a process that protects your organization from its employees, create a process that enables team members to excel.

Trust them, and back up that trust by giving them authority over the project's success. If you can't trust your team, you don't have the right people. No one is perfect, but you need a team that, as a whole, you can trust.

Within the team, anyone can be a leader. Encourage team members to turn to the person or people most qualified to make a necessary decision. For example, when you need design decisions, ask your senior programmers for help. When you need business decisions, ask your most experienced businessperson to make the right choice.

Managers, rather than telling the team what to do, let the team tell you what they need you to do to help them succeed.

In Practice

This principle has two parts: first, get the right people, then give them the power do their work right. XP supports the first part by including customers and testers on the team and involving real customers when appropriate.

XP supports the second part—giving team members the power to do their work right—with many of its practices. XP has a coach, not a team lead, who helps, not directs, team members.

Build the Process for the People

Agile methods recognize the humanity at the core of software development. Agile methods are built around people, not machines.

One aspect of humanity is that we're fallible. We make mistakes, forget important practices, and refuse to do things that are good for us—especially when we're tired or under stress.

We have strengths, too. We are creative, playful, and—under the right circumstances—passionate and driven to succeed. No machine can match these characteristics.

As you modify your agile method, work with these essential strengths and weaknesses. Don't require perfection; instead, build your process to identify and fix mistakes quickly. If a task is boring and repetitive, automate it. Have fun, too.

In Practice

XP's demand for self-discipline seems to violate this principle of understanding human weakness. People aren't good at being self-disciplined all the time, so how can XP succeed?

XP handles the challenge of self-discipline in several ways. First, software developers love to produce high-quality work; once they see the quality of code that XP provides, they tend to love it.

They may not always stay disciplined about the practices, but they generally want to follow the practices.

Second, energized work and pair programming give developers the support they need to be disciplined.

Pair programming provides positive peer pressure and additional support; if one member of the pair feels like taking an ill-advised shortcut, the other often reins him in.

Finally, while XP requires that the team be generally disciplined, it doesn't require perfection.
If a pair makes a poor decision, collective code ownership means that another pair is

Beyond Practices

A friend—"Mel"—used to work for a small consulting company. The shop had three to five developers and twice that many clients at any time, so it was common for them to work on several projects during the week.

To simplify billing, the company used a custom time-tracking application that ran constantly in Windows, requiring developers to enter different billing codes whenever they changed tasks.

That single application was the only reason the developers needed to use Windows, as they deployed almost exclusively to Linux-based platforms. The lack of access to native tools occasionally caused problems. Regular task-switching—the reason for the time-tracking application—was often a more serious problem among the developers than minute-by-minute statistics.

Mel's solution had two parts. First, he dedicated his mornings to small tasks such as addressing bug reports or minor enhancements or customer requests. The minimum billable unit was 15 minutes, which was just about enough time to get into a flow state for any particular project.

This left his afternoons (his most productive time) for longer tasks of two to four hours. Very few customer requests needed immediate solutions, and most of the customers were on the East Coast with a three-hour time difference; when he returned from lunch at 1 p.m., his customers were preparing to leave for the day.

The second part of the solution was using index cards to record task times. This was often faster than finding the right billing codes in the application. It also meant that Mel could boot his computer into Linux and stay there, then enter his stats into the application on another machine just before leaving for the day. The other developers noticed Mel's productivity increase, and he was only too happy to share his ideas. When their manager realized that everyone had switched to a new system, the results were inarguable. The developers were happier and more productive.

Eliminate Waste

Agility requires flexibility and a lean process, stripped to its essentials. Anything more is wasteful. Eliminate it! The less you have to do, the less time your work will take, the less it will cost, and the more quickly you will deliver.

You can't just cut out practices, though. What's really necessary? How can you tell if something helps or hinders you? What actually gets good software to the people who need it? Answering these questions helps you eliminate waste from your process and increase your agility.

Work in Small, Reversible Steps

The easiest way to reduce waste is to reduce the amount of work you may have to throw away.

This means breaking your work down into its smallest possible units and verifying them separately.

Incremental change is a better approach. I make one well-reasoned change, observe and verify its effects, and decide whether to commit to the change or revert it. I learn more and come up with better—and cleaner—solutions.

This may sound like taking baby steps, and it is. Though I can work for 10 or 15 minutes on a feature and get it mostly right, the quality of my code improves immensely when I focus on a very small part and spend time perfecting that one tiny piece before continuing.

In Practice

The desire to solve big, hairy problems is common in developers. Pair programming helps us encourage each other to take small steps to avoid unnecessary embellishments.

Further, the navigator concentrates on the big picture so that both developers can maintain perspective of the system as a whole.

Test-driven development provides a natural rhythm with its think-test-design-code-refactor cycle.
At a higher level, stories limit the total amount of work required for any one pairing session.

The maximum size of a step cannot exceed a few days. As well, continuous integration spreads working code throughout the whole team. The project makes continual, always-releasable progress at a reliable pace.

Finally, refactoring enables incremental design. The design of the system proceeds in small steps as needed. As developers add features, their understanding of the sufficient and necessary design will evolve; refactoring allows them to refine the system to meet its optimal current design.

Fail Fast

It may seem obvious, but failure is another source of waste. Unfortunately, the only way to avoid failure entirely is to avoid doing anything worthwhile.

Instead of trying to avoid failure, embrace it. Think, "If this project is sure to fail, I want to know that as soon as possible." Look for ways to gather information that will tell you about the project's likelihood of failure. Conduct experiments on risk-prone areas to see if they fail in practice.

The sooner you can cancel a doomed project, the less time, effort, and money you'll waste on it.

Either way, invest only as much time and as many resources as you need to be sure of your results.

With these principles guiding your decisions, you'll fear failure less. If failure doesn't hurt, then it's OK to fail.

In Practice

One of the challenges of adopting XP is that it tends to expose problems. For example, iterations, velocity, and the planning game shine the harsh light of fact on your schedule aspirations.

This is intentional: it's one of the ways XP helps projects fail fast. If your desired schedule is unachievable, you should know that. If the project is still worthwhile, either reduce scope or change your schedule. Otherwise, cancel the project. This may seem harsh, but it's really just a reflection of the "fail fast" philosophy.

Cancelling a project early isn't a sign of failure in XP teams; it's a success. The team prevented a doomed project from wasting hundreds of thousands of dollars.

Beyond Practices

I once led a development team on a project with an "aggressive schedule." Seasoned developers recognize this phrase as a code for impending disaster. The schedule implicitly required the team to sacrifice their lives in a misguided attempt to achieve the unachievable.

I knew before I started that we had little chance of success. I accepted the job anyway, knowing that we could at least fail fast. This was a mistake. I shouldn't have assumed. Because there were clear danger signs for the project, our first task was to gather more information. We conducted three two-week iterations and created a release plan. Six weeks after starting the project, we had a reliable release date. It showed us coming in very late.

I thought this was good news—a textbook example of failing fast. We had performed an experiment that confirmed our fears, so now it was time to take action: change the scope of the project, change the date, or cancel the project. We had a golden opportunity to take a potential failure and turn it into a success, either by adjusting our plan or cutting our losses.

Unfortunately, I hadn't done my homework. The organization wasn't ready to accept the possibility of failure. Rather than address the problem, management tried to force the team to meet the original schedule. After realizing that management wouldn't give us the support we needed to succeed, I eventually resigned.

Maximize Work Not Done

The agile community has a saying: "Simplicity is the art of maximizing the work not done."

This idea is central to eliminating waste. To make your process more agile, do less.

Simplifying your process sometimes means sacrificing formal structures while increasing rigor.

For example, an elegant mathematical proof sketched on the back of a napkin may be rigorous, but it's informal. Similarly, sitting with customers decreases the amount of formal requirements documentation you create, but it substantially increases your ability to understand requirements.

Solutions come from feedback, communication, self-discipline, and trust

In Practice

By having teams sit together and communicate directly, XP eliminates the need for intermediate requirements documents. By using close programmer collaboration and incremental design.
XP also eliminates waste by reusing practices in multiple roles. The obvious benefit of pair programming, for example, is continuous code review, but it also spreads knowledge throughout the team, promotes self-discipline, and reduces distractions. Collective code ownership not only enables incremental design and architecture, it removes the time wasted while you wait for someone else to make a necessary API change.

Beyond Practices

ISO 9001 certification is an essential competitive requirement for some organizations. I helped one such organization develop control software for their high-end equipment. This was the organization's first XP project, so we had to figure out how to make ISO 9001 certification work with XP. Our challenge was to do so without the waste of unnecessary documentation procedures.

Nobody on the team was an expert in ISO 9001, so we started by asking one of the organization's internal ISO 9001 auditors for help. (This was an example of the "Let the Right People Do the Right Things" principle) From the auditor, we learned that ISO 9001 didn't mandate any particular process; it just required that we had a process that achieved certain goals, that we could prove we had such a process, and that we proved we were following the process.

This gave us the flexibility we needed. To keep our process simple, we reused our existing practices to meet our ISO 9001 rules. Rather than creating thick requirements documents and test plans to demonstrate that we tested our product adequately, we structured our existing customer testing practice to fill the need. In addition to demonstrating conclusively that our software fulfilled its necessary functions, the customer tests showed that we followed our own internal processes.

Pursue Throughput

A final source of waste unreleased software. It's partially done work—work that has cost money but has yet to deliver any value.

Partially done work also hurts throughput, which is the amount of time it takes for a new idea to become useful software. Low throughput introduces more waste. The longer it takes to develop an idea, the greater the likelihood that some change of plans will invalidate some of the partially done work.

To minimize partially done work and wasted effort, maximize your throughput

To maximize throughput, the constraint needs to work at maximum productivity.

In Practice

XP planning focuses on throughput and minimizing work in progress.

Every iteration takes an idea—a story—from concept to completion. Each story must be "done done" by the end of the iteration.

XP's emphasis on programmer productivity—often at the cost of other team members' productivity—is another example of this principle. Although having customers sit with the team full-time may not be the most efficient use of the customers' time, it increases programmer productivity.

Beyond Practices

Our project faced a tight schedule, so we tried to speed things up by adding more people to the project. In the span of a month, we increased the team size from 7 programmers o 14 programmers, then to 18 programmers. Most of the new programmers were junior-level.

In this particular project, management ignored our protestations about adding people, so we decided to give it our best effort. Rather than having everyone work at maximum efficiency, we focused on maximizing throughput.

We started by increasing the size of the initial development team—the Core Team—only
slightly, adding just one person. The remaining six developers formed the SWAT Team.

Their job was not to work on production software, but to remove roadblocks that hindered the core development team. Every few weeks, we swapped one or two people between the two teams to share knowledge.

This structure worked well for us. It was a legacy project, so there were a lot of hindrances blocking development.