XP Values & Principles

Values

Communication
Facilitate communication between people who know solutions and people who have the power to make changes.
When you encounter a problem, ask if it was caused by lack of communication. If so, improve communication accordingly.

Simplicity
What is the simplest solution that could possibly work? Avoid oversimplification! Avoid overcomplication!
Simplicity is in the eye of the beholder. Respect the needs of users, designers, code authors, code readers, et. al.

Feedback
No fixed direction remains valid for long. Use feedback to change direction accordingly.
Generate as much feedback as you can handle as quickly as possible. Adapt accordingly to important feedback. Ignoring important feedback is a sign that you need to slow down.
Feedback should contribute to Communication and Simplicity.

Courage
Courage is effective action in the face of circumstances that make such action difficult.
Challenge the status quo while maintaining respect of the other values.

Respect
Care about your team, your users, and your project. Act in ways that demonstrate that care.

Principles

Humanity
Recognize that software is written by people with human needs including basic safety, accomplishment, belonging, growth, and understanding other people.

Economics
Somebody is paying for all of this. Understand the time value of money -- a dollar today is worth more than a dollar tomorrow --> other things being equal, releasing today is worth more than releasing tomorrow.
Understand the option value of the team and the system -- our ability to adapt ourselves and the system over time allows us to make money in ways we didn't necessarily imagine at the start.

Mutual Benefit
The most important XP principle. Choose actions that benefit all involved instead of actions that impose a cost on one for the benefit of another. Avoid net losses by solving more problems than you create.

Self-Similarity
Re-use the structure of good solutions in different contexts at different scales. The solution won't always be a good fit in a different context, but trying something known to work in the past is often a good start.

Improvement
Do the best you can today, striving for the awareness and understanding necessary to do better tomorrow. Perfect the process over time; don't wait for perfection in order to begin.
Gradually eliminate waste.

Diversity
Diverse ideas present opportunities. Embrace conflicting ideas and resolve disagreement productively.

Reflection
Think about how and why you are working. Expose your mistakes and learn from them. Pay attention to your emotions as well as intellectual analysis; emotions are often strong indicators of how well you are working. Reflect after doing to avoid overdoing reflection.

Flow
Maintain a continuous flow of software delivery. Release smaller increments ever more frequently. Any time you move away from flow, resolve to return. Identify and address the problems that disrupted your flow.

Opportunity
Learn to see problems as opportunities for change. Move beyond an attitude of "survival" by learning and improving in the face of challenges.

Redundancy
The critical, difficult problems in software development should be solved several different ways. Even if one solution fails utterly, the other solutions will prevent disaster.
For example, Defects corrode trust and trust is the great waste eliminator. XP addresses defects with many practices, including pair programming, continuous integration, sitting together, and daily deployment.
Eliminate redundancy that once served a valid purpose only when it is proven redundant in practice. For example, eliminate post-development testing only when you do not find any defects several deployments in a row.

Failure
If you're having trouble succeeding, take action and fail. Failure is valuable as long as it imparts knowledge. When you don't know what to do, risking failure can be the shortest, surest road to success. (Don't use this to excuse failure when you really knew better.)

Quality
Projects do not go faster by accepting lower quality. Likewise, they don't go slower by demanding higher quality. Iteratively improving quality can lead to faster, more predictable delivery.
Control projects by adapting scope as necessary to meet fixed deadlines with fixed costs.

Baby Steps
What is the least you could do that is recognizably in the right direction? Take many small steps rapidly to avoid stasis or glacial change. Momentous change taken all at once is dangerous.

Accepted Responsibility
Accept responsibility for the task at hand. With your acceptance of responsibility, comes the authority to decide how best to reach resolution. Maintain alignment of responsibility and authority.

XP Primary Practices

Primary Practices

Sit Together

Sit together with your team whenever possible. The more face time you have the more humane and productive the project. Sitting together encourages communication with all your senses. Working together encourages opportunistic productive conversation and reduce unproductive scheduled meetings.
(Problems observed are always people problems. Technical fixes are not enough. Address root people problems that lead to technical problems.)

Whole Team

Include all skills on the cross-functional team that are required for the project. Identify primarily with the team instead of with your function. Identify with one team. For large projects, decompose the problem so that it can be addressed by a team of teams.

Informative Workspace

An interested observer should be able to get a general sense of how the project is going by looking around the workspace for 15 seconds. Post project artifacts that benefit the team. If a visible artifact stops getting updated or becomes irrelevant, take it down.

Energized Work

Work as much time as you can sustain productively. Creativity comes from a prepared, rested, relaxed mind. It's easy to remove value from a project, and when you're tired it's hard to recognize that you're removing value. Instead of working longer hours, manage the existing time more effectively.

Pair Programming

Write programs with two people sitting at one machine. Keep each other on task. Brainstorm refinements. Clarify ideas. Take initiative when your partner is stuck. Hold each other accountable to the team's practices. Take breaks when you need to work on an idea alone.

Stories

Plan using units of customer-visible functionality. As soon as a story is written try to estimate the development effort necessary to implement it. Estimation gives the business and technical perspectives a chance to interact. Split, combine, or extend scope based on what you know about features' estimated value and effort. Identify how to get the greatest return from the smallest investment.

Weekly Cycle

Plan work a week at a time. At the beginning of the week, review progress to date, including how actual progress matched expected progress; ask your customers -- or suitable representative of the customer -- to pick a week's worth of stories; break the stories into tasks. Team members accept responsibility for the tasks and estimate them.
Planning is a necessary form of waste. Work on gradually reducing the percentage of time you spend planning.

Quarterly Cycle

Plan work a quarter at a time. At the end of a quarter, reflect on the team, the project, its progress, and its alignment with larger goals. Identify bottlenecks, initiate repairs, plan the themes for the quarter, Pick a quarter's worth of stories to address those themes, and focus on the big picture where the project fits within the organization.

Slack

In any plan, allow for time to compensate if you get behind. Approaches to do so include planning some optional tasks that you could drop, one week in eight could be "Geek Week", 20% of the weekly budget could be used for programmer-chosen tasks. Begin slack with yourself by telling yourself how long you actually think a task will take and giving yourself time to do it.

Ten-Minute Build

Automatically build the whole system and run all of the tests in ten minutes. Any guess about what parts of the system need to be built and tested introduces the risk of error.

Continuous Integration

Integrate and test changes after no more than a couple of hours. Integration is unpredictable and can take more time than programming. The longer you wait to integrate the more it costs and the more unpredictable the cost becomes. Integrate and build a complete product. Continuous integration should be complete enough that production deployment of the system is no big deal.

Test-First Programming

Start a week by writing failing automated system tests for each story that will pass when the story is complete. Spend the rest of the week completing the stories by getting the tests to pass. As you work on a story, write an automated unit test just before each code change you will make. Test-first programming addresses scope creep, coupling and cohesion, trust, and rhythm.

Incremental Design

Invest in the design of the system every day. Strive to make the design of the system an excellent fit for the needs of the system that day.
While studies have shown that the cost of fixing defects increases over time, it is a fallacious conclusion that the cost of all changes increases over time. Maintain conditions that support your ability to change the system over time.
Align design investment with the needs of the system so far. The most effective time to design is in the light of experience. Design in advance of experience when necessary, while deferring design until the last responsible moment.
Eliminate duplication as a guide for where within the system to design.

XP Corollary Practices

Corollary Practices

Real Customer Involvement
Make people whose lives and business are affected by your system part of the team. Visionary custoemrs can be part of quarterly and weekly planning.

Incremental Deployment
When changing an existing system, gradually adapt its production behavior beginning very early in the project. Find a little piece of functionality or a limited data set you can handle right away. Deploy it.

Team Continuity
Keep effective teams together. People create value not just by what individuals know, but also by what they accomplish together. Ignoring the value of relationships and trust to simplify a scheduling problem is false economy. Mix in new members while mostly keeping teams together to get the benefits of both stable teams and consistently spread knowledge and experience.

Shrinking Teams
As a team grows in capability, keep its workload constant and gradually reduce its size. When the team has too few members, merge it with another too-small team.

Root-Cause Analysis
Every time a defect is found after deployment, eliminate the symptom and its cause. Never make the same kind of mistake again. (1) Write an automated system test that demonstrates the defect by expecting the absent desired behavior. (2) Write a unit test with the smallest possible scope that also reproduces the defect. (3) Fix the system so the unit test works. This should cause the system test to pass also. If not, return to (2). (4) Figure out why the defect was created and wasn't caught. Initiate the necessary changes to prevent this kind of defect in the future. Use Five Whys to accomplish (4).

Shared Code
Anyone on the team can improve any part of the system at any time. If something is wrong with the system and fixing it is not out of scope for what you're doing right now, fix it. Until the team develops a sense of collective responsibility no one is responsible and quality deteriorates.

Code and Tests
Maintain only the code and the tests as permanent artifacts. Generate other documents from the code and tests. Rely on social mechanisms to keep alive important history of the project.

Single Code Base
Maintain only one code stream. Develop in a temporary branch but never let it live longer than a few hours. Many rationalizations of multiple code streams are micro-optimizations that ignore macro-consequences.

Daily Deployment
Put new software into production every night. Any gap between what is on a programmer's desk and what is in production is a risk and a waste.

Negotiated Scope Contract
Write contracts for software development that fix time, costs, and quality, and call for an ongoing negotiation of the precise scope of the system. Reduce risk by signing a sequence of short contracts instead of one long one.

Pay-Per-Use
Charge for the time the system is used. Money is the ultimate feedback. Connecting money flow directly to software development provides accurate information with which to drive development.
Pay-per-release opposes the supplier's and the customer's interests. The supplier is selfishly motivated to provide many releases. The customer wants fewer releases because of the pain of upgrading.