Chapter 5. Principles

Values are too abstract to directly guide behavior. Long documents are intended to communicate, so are daily conversations. Which is the most effective? The answer depends partly on context and partly on intellectual principles. In this case, the principle of humanity suggests conversation meets the basic human need for connection and so is the preferred form of communication, all other things being equal. Written communication is inherently more wasteful. While written communication allows you to reach a large audience, it is a one-way communication. Conversation allows for clarification, immediate feedback, brainstorming together, and other things you can't do with a document. Written communication tends to be taken as fact or rejected outright, neither of which is an invitation to increased communication.

The principles listed here are not the only possible principles to guide software development. In the development of safety-critical systems, for example, the principle of traceability is at work. At any time you should be able to trace a path from the work done back to an explicitly expressed need from the users. No work should be done for its own sake. If you work in safety-critical systems, the principle of traceability is important for gaining certification for your systems. Because it is not applicable to all software, I did not include it in this list. Other principles may guide your team's practices, but these are the principles that guide XP.

Humanity

People develop software. This simple, inescapable fact invalidates most of the available methodological advice. Often, software development doesn't meet human needs, acknowledge human frailty, and leverage human strength. Acting like software isn't written by people exacts a high cost on participants, their humanity ground away by an inhumane process that doesn't acknowledge their needs. This isn't good for business either, with the costs and disruption of high turnover and missed opportunities for creative action.

What do people need to be good developers?

- Basic safety freedom from hunger, physical harm, and threats to loved ones. Fear of job loss threatens this need.
- Accomplishment the opportunity and ability to contribute to their society.
- Belonging the ability to identify with a group from which they receive validation and accountability and contribute to its shared goals.
- Growth the opportunity to expand their skills and perspective.
- Intimacy the ability to understand and be understood deeply by others.

I chose practices for XP because they meet both business and personal needs. There are other human needs; such as rest, exercise, and socialization; that don't need to be met in the work environment. Time away from the team gives each individual more energy and perspective to bring back to the team. Limiting work hours allows time for these other human needs and enhances each person's contribution while he is with the team.

Part of the challenge of team software development is balancing the needs of the individual with the needs of the team. The team's needs may meet your own long-term individual goals, so are worth some amount of sacrifice. Always sacrificing your own needs for the team's doesn't work. If I need privacy, I am responsible for finding a way to get my need met in a way that doesn't hurt the team. The magic of great teams is that after the team members develop trust they find that they are free to be *more* themselves as a result of their work together.

While intimacy feels great, work is still work. Private details of life confuse team communication. I talked to one team in which a member, once he got comfortable on the team, talked about private details of his life every morning. No one else was comfortable hearing about his private life but they didn't know how to confront him. Eventually, the senior team member took him aside and asked him to keep private matters private.

I try to separate my life into private matters that I only discuss with my spouse, personal matters that I discuss with those who have earned my trust, and public matters that I don't mind talking about with anyone. Figuring out which is which is not a simple matter, nor is it simple to figure out who to trust. The rewards of this separation when it works well are effective communication on the job and relationships that are valuable in all aspects of my life.

Economics

Somebody has to pay for all this. Software development that doesn't acknowledge economics risks the hollow victory of a "technical success". Make sure what you are doing has business value, meets business goals, and serves business needs. For example, solving the highest priority business need first maximizes the value of the project.

Two aspects of economics that affect software development are the time value of money and the option value of systems and teams. The time value of money says that a dollar today is worth more than a dollar tomorrow. Software development is more valuable when it earns money sooner and spends money later. Incremental design explicitly defers design investment until the last responsible moment in an effort to spend money later. Pay-per-use provides a way of realizing revenue from features as soon as they are deployed.

Another source of economic value in software development is its value as options for the future. If I can redeploy my media scheduling program for a variety of scheduling-related tasks, it is much more valuable than if it can only be used for its originally intended purpose. All the practices are intended to enhance the option value of both the software and the team while keeping in mind the time value of money by not investing in speculative flexibility.

Mutual Benefit

Every activity should benefit all concerned. Mutual benefit is the most important XP principle and the most difficult to adhere to. There are always solutions to any problem that cost one person while benefitting another. When the situation is desperate, these solutions seem attractive. They are always a net loss, however, because the ill will they create tears down relationships that we need to value. The computer business is really a people business and maintaining working relationships is important.

Extensive internal documentation of software is an example of a practice that violates mutual benefit. I am supposed to slow down my development considerably so some unknown person in a potential future will have an easier time maintaining this code. I can see a possible benefit to the future person should the documentation still happen to be valid, but no benefit now.

XP solves the communication-with-the-future problem in mutually beneficial ways:

- I write automated tests that help me design and implement better today. I leave these tests for future programmers to use as well. This practice benefits me now and maintainers down the road.

- I carefully refactor to remove accidental complexity, giving me both satisfaction and fewer defects and making the code easier to understand for those who encounter it later.

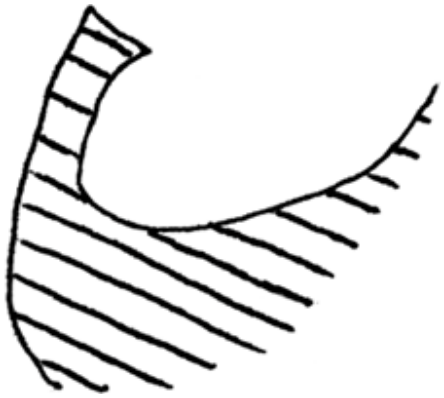
- I choose names from a coherent and explicit set of metaphors which speeds my development and makes the code clearer to new programmers.

If you want people to take your advice, you need to solve more problems than you create. Mutual benefit in XP is searching for practices that benefit me now, me later, and my customer as well. Win-win-win practices are easier to sell because they relieve some immediate pain. For example, someone wrestling with a tough defect is ready to learn test-first programming. When it benefits me now, it is easier to accept doing something to help others both now and in the future.

Self-Similarity

One day I went walking along the Sardinian coast. I saw a little tide pool, maybe two feet across, with the shape outlined in Figure 2. I looked up and noticed that the bay I was walking around, maybe a mile across, had roughly the same shape. "What a great example of the fractal nature of geology," I thought to myself. This drawing is actually a tracing of a map of the whole northwest corner of Sardinia. When nature finds a shape that works, she uses it everywhere she can.

Figure 2. Naturally occurring shape



The same principle applies to software development: try copying the structure of one solution into a new context,

even at different scales. For example, the basic rhythm of development is that you write a test that fails and then you make it work. The rhythm operates at all different scales. In a quarter, you list the themes you want to address and then you address them with stories. In a week, you list the stories you want to address, write tests expressing the stories, then make them work. In a few hours, you list the tests you know you need to write, then write a test, make it work, write another test, and make them both work until the list is done.

Self-similarity isn't the only principle at work in software development. Just because you copy a structure that works in one context doesn't mean it will work in another. It is a good place to start, though. Likewise, just because a solution is unique doesn't mean it's bad. The situation may really call for a unique solution.

In the first edition of *Extreme Programming Explained*, my advice for the weekly cycle was much more like a waterfall: write some code, then test it to make sure it works. I should have paid attention to self-similarity. Having the system-level tests before you begin implementation simplifies design, reduces stress, and improves feedback.

Improvement

In software development, "perfect" is a verb, not an adjective. There is no perfect process. There is no perfect design. There are no perfect stories. You can, however, perfect your process, your design, and your stories.

"Best is the enemy of good enough" suggests that mediocrity is preferable to waiting. This phrase misses the point of XP, which is excellence in software development through improvement. The cycle is to do the best you can today, striving for the awareness and understanding necessary to do better tomorrow. It doesn't mean waiting for perfection in order to begin.

In translating values to practices, the principle of improvement shows in practices that get an activity started right away but refine the results over time. The quarterly cycle is an expression of the possibility of improving long-term plans in the light of experience. Incremental design puts improvement to work by refining the design of the system. The actual design will never be a perfect reflection of the ideal, but you can strive daily to bring the two closer.

The history of software development technology shows us gradually eliminating wasted effort. For example, symbolic assemblers eliminated the wasteful tedium of translating machine instructions into physical bit encodings; "automatic programming" then eliminated the wasteful tedium of translating an abstract description of a program into assembly language; and so on up through automatic storage deallocation.

While our improved technology has eliminated waste, our increased rigidity and specialized social structures in development organizations are increasingly wasteful. The

key to improvement is reconciling the two, using newfound technological efficiency to enable new, more effective social relationships. Put improvement to work by not waiting for perfection. Find a starting place, get started, and improve from there.

Diversity

Software development teams where everyone is alike, while comfortable, are not effective. Teams need to bring together a variety of skills, attitudes, and perspectives to see problems and pitfalls, to think of multiple ways to solve problems, and to implement the solutions. Teams need diversity.

Conflict is the inevitable companion of diversity. Not conflict in the "we hate each other and we just can't make progress" sense, but in the "there are two ways to solve this" sense. How do you choose?

Two ideas about a design present an opportunity, not a problem. The principle of diversity suggests that the programmers should work together on the problem and both opinions should be valued.

What if the team isn't good at conflict? Every team has conflict. The question is whether they resolve it productively. Respecting others and maintaining myself smooths communication in times of stress.

Diversity is expressed in the practice of Whole Team, where you bring together on the team people with a variety of skills and perspectives. The various planning cycles encourage people with different perspectives to interact with the goal of creating the most valuable software possible in the time available.

Reflection

Good teams don't just do their work, they think about *how* they are working and *why* they are working. They analyze why they succeeded or failed. They don't try to hide their mistakes, but expose them and learn from them. No one stumbles into excellence.

The quarterly and weekly cycles include time for team reflection, as do pair programming and continuous integration. But reflection should not be limited to "official" opportunities. Conversation with a spouse or friend, vacation, and non-software-related reading and activities all provide individual opportunities to think about how and why you are working the way you are. Shared meals and coffee breaks provide an informal setting for shared reflection.

Reflection isn't a purely intellectual exercise. You can gain insight by analyzing data, but you can also learn from your gut. The "negative" emotions like fear, anger, and anxiety have long provided cues that something bad was about to happen. It takes effort to listen to what your emotions tell you about your work, but feelings tempered by the intellect are a source of insight.

Reflection can be taken too far. Software development has a long tradition of people so busy thinking *about* software development they don't have time to develop software. Reflection comes after action. Learning is action reflected. To maximize feedback, reflection in XP teams is mixed with doing.

Flow

Flow in software development is delivering a steady flow of valuable software by engaging in all the activities of development simultaneously. The practices of XP are biased towards a continuous flow of activities rather than discrete phases.

Software development has long delivered value in big chunks. "Big Bang" integration reflects this tendency. Many teams make the problem worse by tending to respond to stress by making the chunks of value bigger, from deploying software less frequently to integrating less often. Less feedback makes the problem worse, leading to a tendency for even bigger chunks. The more things are deferred, the larger the chunk, the higher the risk. In contrast, the principle of flow suggests that for improvement, deploy smaller increments of value ever more frequently.

A few trends in software development buck the concept of bigger batches. The daily build, for example, is flow-oriented. However, daily builds are a small step on the road to flow. It is not enough that the software compile and link every day; it should also function correctly every day or, better yet, several times a day.

I visited a team that used to deploy every week. It had more and more problems, until it was taking six days to deploy a week's worth of software. The team chose to deploy every two weeks. This amplified their integration and deployment problems. Any time you move away from flow, resolve to return. Resolve the problems that disrupted your flow and get back to weekly deployment as soon as you can.

Opportunity

Learn to see problems as opportunities for change. This isn't to say there are no problems in software development. However, the attitude of "survival" leads to just enough problem solving to get by. To reach excellence, problems need to turn into opportunities for learning and improvement, not just survival.

You might not know what to do about a problem. You might want more time to think about what to do. Sometimes the desire for more time is a mask worn to protect from the fear of the consequences of getting going. Sometimes, though, patience solves a problem by itself.

Turning problems into opportunities takes place across the development process. It maximizes strengths and minimizes weaknesses. Can't make accurate long-term plans? Finehave a quarterly cycle during which you refine your long-term plans. A person alone makes too many mistakes? Fineprogram in pairs. The practices are effective precisely because they address the enduring problems of people developing software together.

As you begin practicing XP, you will certainly encounter problems. Part of being extreme is consciously choosing to transform each problem into an opportunity: an opportunity for personal growth, deepening relationships, and improved software.

Redundancy

Yes, 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. The cost of the redundancy is more than paid for by the savings from not having the disaster.

For example, defects corrode trust and trust is the great waste eliminator. Defects are a critical, difficult problem. Defects are addressed in XP by many of the practices: pair programming, continuous integration, sitting together, real customer involvement, and daily deployment, for example. Even if your partner doesn't catch an error, someone else sitting across the room might or it might be caught by the next integration. Some of these practices are certainly redundant, catching some of the same defects.

You can't solve the defect problem with a single practice. It is too complex, with too many facets, and it will never be solved completely. What you hope to achieve is few enough defects to maintain trust both within the team and with the customer.

While redundancy can be wasteful, be careful not to remove redundancy that serves a valid purpose. Having a testing phase after development is complete should be redundant. However, eliminate it only when it is proven redundant in practice by not finding any defects several deployments in a row.

Failure

If you're having trouble succeeding, fail. Don't know which of three ways to implement a story? Try it all three ways. Even if they all fail, you'll certainly learn something valuable.

Isn't failure waste? No, not if it imparts knowledge. Knowledge is valuable and sometimes hard to come by. Failure may not be avoidable waste. If you knew the best way to implement the story you'd just implement it that way. Given that you don't already know the best way, what's the cheapest way to find out?

I coached a team that had several good designers, so good that each of them could come up with two or three ways of solving any given problem. They would sit for hours, talking about each of their ideas in turn. By the time they were tired of talking, they could have implemented all the alternatives twice. They didn't want to waste programming time, though, so they wasted talking time instead.

I bought the team a kitchen timer and asked them to limit design discussions to fifteen minutes. When the timer went off, two of them would go implement something. They only used the timer a couple of times, but they kept it around as a reminder to fail instead of talk.

This is not intended to excuse failure when you really knew better. When you don't know what to do though, risking failure can be the shortest, surest road to success.

Quality

Sacrificing quality is not effective as a means of control. Quality is not a control variable. Projects don't go faster by accepting lower quality. They don't go slower by demanding higher quality. Pushing quality higher often results in faster delivery; while lowering quality standards often results in later, less predictable delivery.

One of my biggest surprises since the first edition of *Extreme Programming Explained* was released has been just how far teams have been able to push quality as measured in defects, design quality, and the experience of development. Each increase in quality leads to improvements in other desirable project properties, like productivity and effectiveness, as well. There is no apparent limit to the benefits of quality, only limits in our ability to understand how to achieve higher quality.

Quality isn't a purely economic factor. People need to do work they are proud of. I remember talking to the manager of a mediocre team. He went home on the weekends and made fancy ironwork as a blacksmith. He met his need for quality; he just met it outside of work.

If you can't control projects by controlling quality, how can you control them? Time and cost are most often fixed. XP chooses scope as the primary means of planning, tracking, and steering projects. Since scope is never known precisely in advance, it makes a good lever. The weekly and quarterly cycles provide explicit points for tracking and choosing scope.

A concern for quality is no excuse for inaction. If you don't know a clean way to do a job that has to be done, do it the best way you can. If you know a clean way but it would take

too long, do the job as well as you have time for now. Resolve to finish doing it the clean way later. This often occurs during architectural evolution, where you have to live with two architectures solving the same problem while you transition from one to the other. Then the transition itself becomes a demonstration of quality: making a big change efficiently in small, safe steps.

Baby Steps

It's always tempting to make big changes in big steps. After all, there's a long way to go and a short time to get there. Momentous change taken all at once is dangerous. It is people who are being asked to change. Change is unsettling. People only change so fast.

I often ask, "What's the least you could do that is recognizably in the right direction?" Baby steps do not justify stasis or glacial change. Under the right conditions, people and teams can take many small steps so rapidly that they appear to be leaping.

Baby steps acknowledge that the overhead of small steps is much less than when a team wastefully recoils from aborted big changes. Baby steps are expressed in practices like test-first programming, which proceeds one test at a time, and continuous integration, which integrates and tests a few hours' worth of changes at a time.

Accepted Responsibility

Responsibility cannot be assigned; it can only be accepted. If someone tries to give you responsibility, only you can decide if you are responsible or if you aren't.

The practices reflect accepted responsibility by, for example, suggesting that whoever signs up to do work also estimates it. Similarly, the person responsible for implementing a story is ultimately responsible for the design, implementation, and testing of the story.

With responsibility comes authority. Misalignments distort the team's communication. When a process expert can tell me how to work, but doesn't share in that work or its consequences, authority and responsibility are misaligned. Neither of us is in an intellectual position to see or use the feedback we need to improve. There is also an emotional cost of living with misalignment.

Conclusion

You can use the principles to understand the practices better and to improvise complementary practices when you don't find one that suits your purpose. While the statement of the practices is intended to be clear and objective (for example "write a test before changing code"), understanding how to apply the practice in your context may not be obvious. The principles give you a better idea of what the practice is intended to accomplish. Also, no fixed list of situated, context-dependent practices covers all of software development. You will create new practices occasionally to fill your specific need. Understanding the principles gives you the opportunity to create practices that work in harmony with your existing practices and your overall goals.

Chapter 6. Practices

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

Practices are situation dependent. If the situation changes, you choose different practices to meet those conditions. Your values do not have to change in order to adapt to a new situation. Some new principles may be called when you change domains.

The practices are stated as absolutes. My intention is to motivate you to aim for perfection, provide you with clear goals, and give you practical ways to get there. The practices are a vector from where you are to where you can be with XP. In XP, you make progress towards this ideal state of effective development. For example, daily deployment may make no sense if you only deploy once a year. Successfully deploying more frequently is an improvement, building confidence for the next step.

Applying a practice is a choice. I think the practices make programming more effective. This is a collection of practices that work and work even better together. They have been used before. Experiment with XP using these practices as your hypotheses. For example, let's try deploying more frequently and see if that helps.

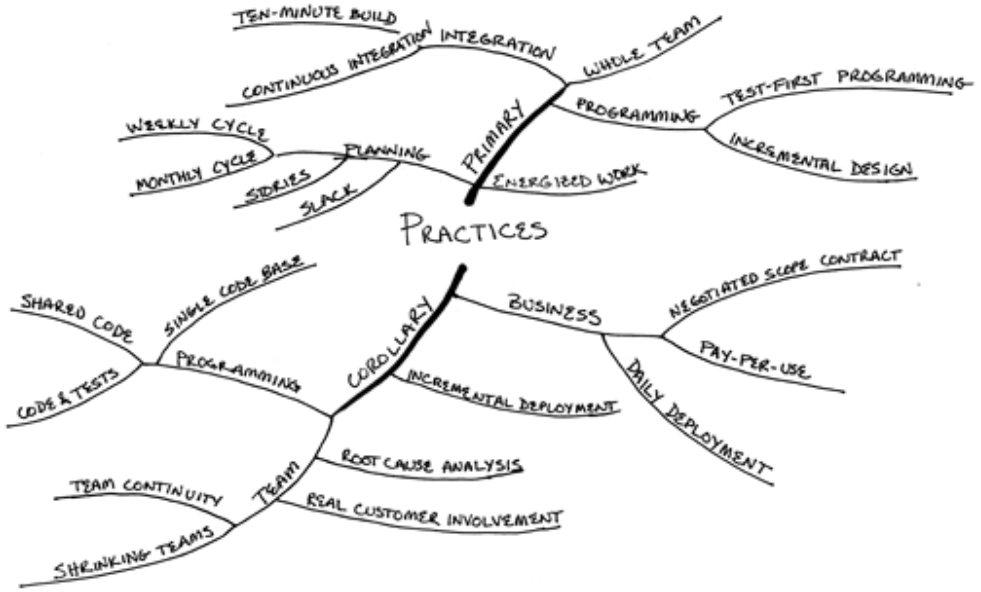
The XP practices do not represent some kind of pinnacle in the evolution of software development. They are a common way station on the road to improvement. The XP practices tend to work well together. Taken one at a time you will likely see improvement. When they begin to compound you may see dramatic improvement. The interactions between the practices amplify their effect.

I have divided the practices into two chapters: "Primary Practices," Chapter 7, and "Corollary Practices," Chapter 9. The primary practices are useful independent of what else you are doing. They each can give you immediate improvement. You can start safely with any of them. The corollary practices are likely to be difficult without first mastering the primary practices. The amplification effect of using the practices together means there is an advantage to adding practices as quickly as you can.

Figure 3 is a summary of the practices:

Figure 3. Summary of practices

[View full size image]


PRACTICES
PRIMARY
INTEGRATION
TEN-MINUTE BUILD
CONTINUOUS INTEGRATION
WHOLE TEAM
PROGRAMMING
TEST-FIRST PROGRAMMING
INCREMENTAL DESIGN
PLANNING
WEEKLY CYCLE
MONTHLY CYCLE
STORIES
SLACK
ENERGIZED WORK
COROLLARY
BUSINESS
NEGOTIATED SCOPE CONTRACT
PAY-PER-USE
DAILY DEPLOYMENT
INCREMENTAL DEPLOYMENT
PROGRAMMING
SINGLE CODE BASE
SHARED CODE
CODE & TESTS
TEAM
ROOT CAUSE ANALYSIS
REAL CUSTOMER INVOLVEMENT
TEAM CONTINUITY
SHRINKING TEAMS

Chapter 7. Primary Practices

In this chapter you'll find practices you can safely start with as you begin to apply XP to improve your software development. Which one you should use first depends completely on your environment and what you perceive as your biggest opportunity for improvement. Some people need planning because they don't know what needs to be done. Some need one of the quality-related practices because they are creating too many defects to be able to see what else is happening.

Sit Together

Develop in an open space big enough for the whole team. Meet the need for privacy and "owned" space by having small private spaces nearby or by limiting work hours so team members can get their privacy needs met elsewhere.

I was called to consult at a floundering project on the outskirts of Chicago. Why the project was floundering was a mystery, because the team consisted of the best technical talent in the company. I walked from cubicle to cubicle trying to figure out what was wrong with their computer program.

After a couple of days, it struck me: I was walking a lot. The senior people of course had corner offices, one in each corner of a floor of a substantial building. The team interacted only a few minutes each day. I suggested that they find a place to sit together. When I returned a month later, the project was humming along. The only space they could find to sit together was in the machine room. They were spending four to five hours a day in a cold, drafty, noisy room; but they were happy because they were successful.

I took two lessons from that experience. One is that no matter what the client says the problem is, it is always a people problem. Technical fixes alone are not enough. The other lesson I took was how important it is to sit together, to communicate with all our senses.

You can creep up on sitting together, if necessary. Put a comfortable chair in your cubicle to encourage conversation. Spend half a day programming in a conference room. Ask for a conference room for a one-week

trial of a more open workspace. All of these are steps towards finding a workspace that is effective for your team.

Tearing down the cubicle walls before the team is ready is counterproductive. If the team members' sense of safety is tied to having their own little space, removing that sense of safety before replacing it with the safety of shared accomplishment is likely to produce resentment and resistance. With a little encouragement, teams can shape their own space. A team that knows that physical proximity enhances communication and that has learned the value of communication will open up their own space, given the chance.

Does the practice of sitting together mean that multisite teams can't "do XP"? Chapter 21, "Purity," explores this question in more depth; but the simple answer is no, teams can be distributed and do XP. Practices are theories, predictions. "Sit Together" predicts that the more face time you have, the more humane and productive the project. If you have a multisite project and everything is going well, keep doing what you're doing. If you have problems, think about ways to sit together more, even if it means traveling.

Whole Team

Include on the team people with all the skills and perspectives necessary for the project to succeed. This is really nothing more than the old idea of cross-functional teams. The name reflects the purpose of the practice, a sense of wholeness on the team, the ready availability of all the resources necessary to succeed. Where intense interactions are necessary for the health of the project, those interacting should be primarily identified with the team and not their functions.

People need a sense of "team":

- We belong.
- We are in this together.
- We support each others' work, growth, and learning.

What constitutes a "whole team" is dynamic. If a set of skills or attitudes becomes important, bring a person with these skills on the team. If someone is no longer necessary, he can go elsewhere. For example, if your project requires many changes to a database, you will need a database administrator on the team. When the need for database changes diminishes that person no longer needs to be part of the team, at least for that function.

An issue that often arises is ideal team size. *The Tipping Point* by Malcolm Gladwell describes two discontinuities in team size: 12 and 150. Many organizations; military, religious, and business; split teams when they cross these thresholds. Twelve is the number of people who can

comfortably interact with each other in a day. With more than 150 people on a team, you can no longer recognize the faces of everyone on your team. Across both of these thresholds it is harder to maintain trust, and trust is necessary for collaboration. For larger projects, finding ways to fracture the problem so it can be solved by a team of teams allows XP to scale up.

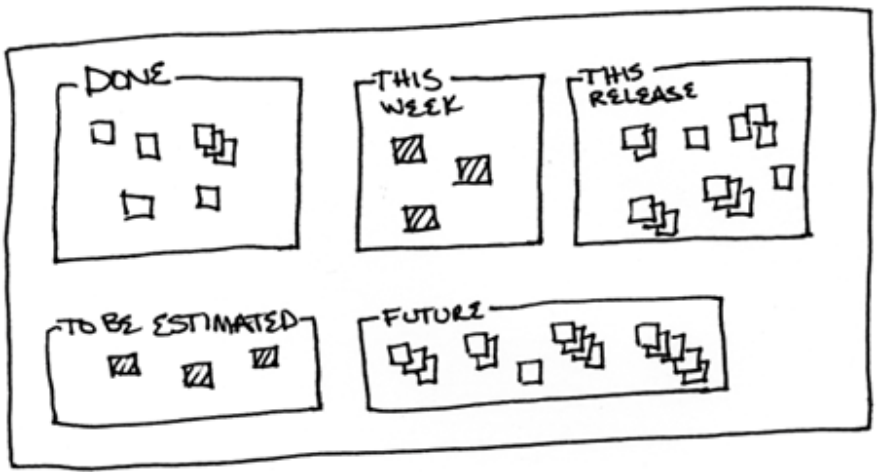
Some organizations try to have teams with fractional people: "You'll spend 40% of your time working for these customers and 60% work for those customers." In this case, so much time is wasted on task-switching that you can see immediate improvement by grouping the programmers into teams. The team responds to the customers' needs. This frees the programmers from fractured thinking. The customer receives the benefit of the expertise of the whole team as needed. People need acceptance and belonging. Identifying with this program on Mondays and Thursdays and that program on Tuesdays, Wednesdays, and Fridays, without having other programmers to identify with, destroys the sense of "team" and is counterproductive.

Informative Workspace

Make your workspace about your work. An interested observer should be able to walk into the team space and get a general idea of how the project is going in fifteen seconds. He should be able to get more information about real or potential problems by looking more closely.

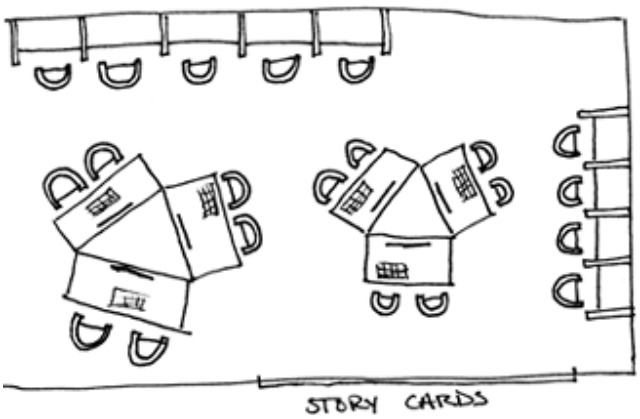
Many teams implement this practice in part by putting story cards on a wall. Sorting the cards spatially conveys information quickly. If the "Done" area isn't collecting cards, what does the team needs to improve in its planning, estimation, or execution? I'll also wonder what customers need to be involved so the slipping scope has minimal business impact. Figure 4 shows an idealized story wall with spatially sorted stories.

Figure 4. Stories on a wall



The workspace (Figure 5) also needs to provide for other human needs. Water and snacks provide comfort and encourage positive social interactions. Cleanliness and order leave minds free to think about the problems at hand. While programming happens in a public space people also need privacy, which can be provided by separate cubes or by limiting work hours.

Figure 5. A team workspace



Another implementation of the informative workspace is big, visible charts. If you have an issue that requires steady

progress, begin charting it. Once the issue is resolved, or if the chart stops getting updated, take it down. Use your space for important, active information.

Energized Work

Work only as many hours as you can be productive and only as many hours as you can sustain. Burning yourself out unproductively today and spoiling the next two days' work isn't good for you or the team.

Where does the penchant for long hours come from? I'm often asked for "scientific" evidence for the practices in XP, as if science could somehow bear the responsibility for project success or failure. Work hours are one area where I wish I could turn this argument around. Where is the scientific evidence that members of a software team produce more value in eighty hour weeks than in forty hour weeks? Software development is a game of insight, and insight comes to the prepared, rested, relaxed mind.

In my own case I think I turn to long work hours as a way of grabbing control in a situation in which I am otherwise out of control. I can't control how the whole project is going; I can't control whether the product sells; but I can always stay later. With enough caffeine and sugar, I can keep typing long past the point where I have started removing value from the project. It's easy to remove value from a software project; but when you're tired, it's hard to recognize that you're removing value.

When you're sick, respect yourself and the rest of your team by resting and getting well. Taking care of yourself is the quickest way back to energized work. You also protect the team from losing more productivity because of illness. Coming in sick doesn't show commitment to work, because when you do you aren't helping the team.

You can make incremental improvements in work hours. Stay at work the same amount of time but manage that

time better. Declare a two-hour stretch each day as Code Time. Turn off the phones and email notification, and just program for two hours. That may be enough improvement for now and may set the stage for fewer hours at work later.

Pair Programming

Write all production programs with two people sitting at one machine. Set up the machine so the partners can sit comfortably side-by-side. Move the keyboard and mouse back and forth so you are comfortable while you are typing. Pair programming is a dialog between two people simultaneously programming (and analyzing and designing and testing) and trying to program better. Pair programmers:

- Keep each other on task.
- Brainstorm refinements to the system.
- Clarify ideas.
- Take initiative when their partner is stuck, thus lowering frustration.
- Hold each other accountable to the team's practices.

Pairing doesn't mean that you can't think alone. People need both companionship and privacy. If you need to work on an idea alone, go do it. Then come back and check in with your team. You can even prototype alone and still respect pairing. However, this is not an excuse to act outside of the team. When you're done exploring, bring the resulting idea, not the code, back to the team. With a partner, you'll reimplement it quickly. The results will be more widely understood, benefitting the project as a whole.

Pair programming is tiring but satisfying. Most programmers can't pair for more than five or six hours in a day. After a week like that, they are ready for a relaxing weekend away from work. I keep a bottle of water beside me while I pair. It's good for my health and I'm eventually reminded to take a break. The breaks keep me fresh for the whole day.

Rotate pairs frequently. Some teams report good results obeying a timer that tells them to shift partners every sixty minutes (every thirty minutes when solving difficult problems). I don't think I'd like this, but I haven't tried it. I like to program with someone new every couple of hours, switching at natural breaks in development.

Pairing and Personal Space

An issue that has come up and requires comment is the close contact in pair programming. Different individuals and cultures are comfortable with different amounts of body space. Pairing with an Italian who communicates best when very close is completely different than pairing with a Dane who likes a few feet of personal space. If you aren't aware of the difference it can be acutely uncomfortable. Personal space must be respected for both parties to work well.

Personal hygiene and health are important issues when pairing. Cover your mouth when you cough. Don't come to work when you are sick. Avoid strong colognes that might affect your partner.

Working effectively together feels good. It may be a new experience in the workplace for some. When programmers aren't emotionally mature enough to separate approval from arousal, working with a person of the opposite gender can bring up sexual feelings that are not in the best interest

of the team. If these feelings arise when pairing, stop pairing with the person until you have taken responsibility for and dealt with your feelings. Even if the feelings are mutual, acting on them will hurt the team. If you want to have an intimate relationship, one of you should leave the team so you can build a personal relationship in a personal setting without confusing the team's communication with a sexual subtext. Ideally, emotions at work will be about work.

It is important to respect individual differences when pairing. In Figure 6 the man has moved closer to the woman than is comfortable for her. Neither is making his or her best technical decisions at this point, although they may be completely unaware of the source of their discomfort.

Figure 6. Personal space and pairing

If you are uncomfortable pairing with someone on the team, talk about it with someone safe; a respected team member, a manager, or someone in human resources. If you aren't comfortable, the team isn't doing as well as it could. And chances are others are uncomfortable too.

Stories

Plan using units of customer-visible functionality. "Handle five times the traffic with the same response time." "Provide a two-click way for users to dial frequently used numbers." As soon as a story is written, try to estimate the development effort necessary to implement it.

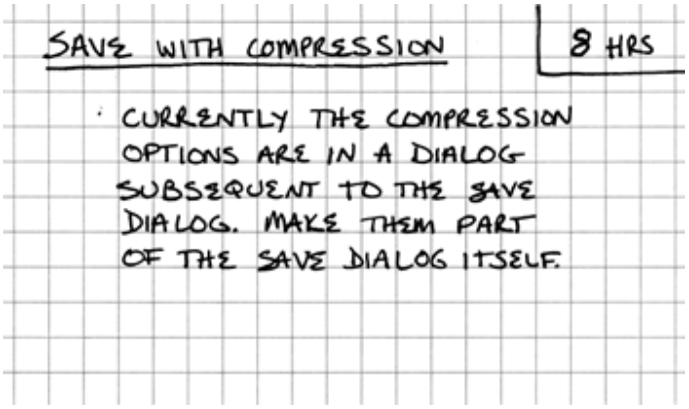
Software development has been steered wrong by the word "requirement", defined in the dictionary as "something mandatory or obligatory." The word carries a connotation of absolutism and permanence, inhibitors to embracing change. And the word "requirement" is just plain wrong. Out of one thousand pages of "requirements", if you deploy a system with the right 20% or 10% or even 5%, you will likely realize all of the business benefit envisioned for the whole system. So what were the other 80%? Not "requirements"; they weren't really mandatory or obligatory.

Early estimation is a key difference between stories and other requirements practices. Estimation gives the business and technical perspectives a chance to interact, which creates value early, when an idea has the most potential. When the team knows the cost of features it can split, combine, or extend scope based on what it knows about the features' value.

Give stories short names in addition to a short prose or graphical description. Write the stories on index cards and put them on a frequently-passed wall. Figure 7 is a sample card of a story I wish my scanner program implemented. Every attempt I've seen to computerize stories has failed to provide a fraction of the value of having real cards on a real wall. If you need to report progress to other parts of the

organization in a familiar format, translate the cards into that format periodically.

Figure 7. Sample story card



One feature of XP-style planning is that stories are estimated very early in their life. This gets everyone thinking about how to get the greatest return from the smallest investment. If someone asks me whether I want the Ferrari or the minivan, I choose the Ferrari. It will inevitably be more fun. However, as soon as someone says, "Do you want the Ferrari for $150,000 or the minivan for $25,000?" I can begin to make an informed decision. Adding new constraints like "I need to haul five children" or "It has to go 150 miles per hour" clear the picture further. There are cases where either decision makes sense. You can't make a good decision based on image alone. To choose a car wisely you need to know your constraints,

both cost and intended use. All other things being equal, appeal comes into play.

Weekly Cycle

Plan work a week at a time. Have a meeting at the beginning of every week. During this meeting:

1. Review progress to date, including how actual progress for the previous week matched expected progress.

2. Have the customers pick a week's worth of stories to implement this week.

3. Break the stories into tasks. Team members sign up for tasks and estimate them.

Start the week by writing automated tests that will run when the stories are completed. Then spend the rest of the week completing the stories and getting the tests to pass. A team proud of its work will fully implement the stories, not just do enough work to get the tests to pass. The goal is to have deployable software at the end of the week which everyone can celebrate as progress.

The week is a widely shared time scale. The nice thing about a week as opposed to two or three (as I recommended in the first edition) is that everyone is focused on Friday. The team's jobprogrammers, testers, and customers togetheris to write the tests and then get them to run in five days. If you get to Wednesday and it is clear that all the tests won't be running, that the stories won't be completed and ready to deploy, you still have time to choose the most valuable stories and complete them.

Some people like to start their week on a Tuesday or Wednesday. I was surprised when I first saw it, but it's common enough to mention. As one manager told me, "Monday's are unpleasant and planning is unpleasant, so

why put them together?" I don't think planning should be unpleasant; but in the meantime, shifting the start of the cycle makes some sense as long as it doesn't put pressure on people to work over the weekend. Working weekends is not sustainable; if the real problem is that the estimates are overly optimistic, then work on improving your estimates.

Planning is a form of necessary waste. It doesn't create much value all by itself. Work on gradually reducing the percentage of time you spend planning. Some teams start with a whole day of planning for a week, but gradually refine their planning skills until they spend an hour planning for the week.

I like to break stories into tasks that individuals take responsibility for and estimate. I think ownership of tasks goes a long way towards satisfying the human need for ownership. I've seen other styles work well, though. You can write small stories that eliminate the need for separate tasks. The cost of this approach is more work for the customer. You can also eliminate sign-up by keeping a stack of tasks. When a programmer is ready to start a new task, he takes one from the top of the stack. This eliminates the opportunity for him to choose a task he particulary cares about or is especially good at, but ensures that each programmer gets a variety of tasks. (Pairing gives programmers a chance to use specialist skills, whoever holds the task.)

The weekly heartbeat also gives you a convenient, frequent, and predictable platform for team and individual experiments. "OK, for the next week we're going to switch pair partners every hour on the hour," or "I'll juggle for five minutes every morning before I start programming."

Quarterly Cycle

Plan work a quarter at a time. Once a quarter reflect on the team, the project, its progress, and its alignment with larger goals.

During quarterly planning:

- Identify bottlenecks, especially those controlled outside the team.
- Initiate repairs.
- Plan the theme or themes for the quarter.
- Pick a quarter's worth of stories to address those themes.
- Focus on the big picture, where the project fits within the organization.

A season is another natural, widely shared timescale to use in organizing time for a project. Using a quarter as a planning horizon synchronizes nicely with other business activities that occur quarterly. Quarters are also a comfortable interval for interaction with external suppliers and customers.

The separation of "themes" from "stories" is intended to address the tendency of the team to get focused and excited about the details of what they are doing without reflecting on how this week's stories fit into the bigger

picture. Themes also fit well into larger-scale planning such as drawing marketing roadmaps.

Quarters are also a good interval for team reflection, finding gnawing-but-unconscious bottlenecks. You can also propose and evaluate long-running experiments quarterly.

Slack

In any plan, include some minor tasks that can be dropped if you get behind. You can always add more stories later and deliver more than you promised. It is important in an atmosphere of distrust and broken promises to meet your commitments. A few met commitments go a long way toward rebuilding relationships.

In Iceland, one of the winter sports is taking monstrous trucks bouncing around the backcountry. These trucks all have four-wheel-drive; but when they are out crashing around, they only use two-wheel-drive. If they get stuck in two-wheel-drive they have four-wheel-drive to get them out. If they get stuck in four-wheel-drive they're just stuck.

I remember two conversations: one with a middle manager who had one hundred people reporting to him and another with his executive manager who had three hundred people in his organization. I suggested to the middle manager that he encourage his teams to only sign up for what they were confident they could actually do. They had a long history of overcommitting and underdelivering. "Oh, I couldn't do that. If I don't agree to aggressive [i.e. unrealistic] schedules, I'll be fired." The next day, I talked to the executive. "Oh, they never come in on time. It's okay. They still deliver enough of what we need."

I had been watching first-hand the incredible waste generated by their habitual overcommitment: unmanageable defect loads, dismal morale, and antagonistic relationships. Meeting commitments, even modest ones, eliminates waste. Clear, honest communication relieves tension and improves credibility.

You can structure slack in many ways. One week in eight could be "Geek Week". Twenty percent of the weekly budget could go to programmer-chosen tasks. You may have to begin slack with yourself, telling yourself how long you actually think a task will take and giving yourself time to do it, even if the rest of the organization is not ready for honest and clear communication.

Ten-Minute Build

Automatically build the whole system and run all of the tests in ten minutes. A build that takes longer than ten minutes will be used much less often, missing the opportunity for feedback. A shorter build doesn't give you time to drink your coffee.

Physics has reassuringly concrete natural constants. At sea level on earth, the force of gravity accelerates objects at 9.8 meters per second per second. You can count on gravity. Software has few such certainties. The ten-minute build is as close as we get in software engineering. I've observed several teams that started with an automated build-and-test process never letting the process take longer than ten minutes. If it did, someone optimized it but only until it took ten minutes again.

The ten-minute build is an ideal. What do you do on your way to that ideal? The statement of the practice gives three clues: *automatically* build the *whole* system and run *all* of the tests in ten minutes. If your process isn't automated, that's the first place to start. Then you may be able to build only the part of the system you have changed. Finally, you may be able to run only tests covering the part of the system at risk because of the changes you made.

Any guess about what parts of the system *need* to be built and what parts *need* to be tested introduces the risk of error. If you are wrong, you may miss unpredictable errors with all of their social and economic costs. How ever, being able to test some of the system is much better than being able to test none at all.

Automated builds are much more valuable than builds requiring manual intervention. As the general stress level

rises, manual builds tend to be done less often and less well, resulting in more errors and more stress. Practices should lower stress. An automated build becomes a stress reliever at crunch time. "Did we make a mistake? Let's just build and see."

Continuous Integration

Integrate and test changes after no more than a couple of hours. Team programming isn't a divide and conquer problem. It is a divide, conquer, and integrate problem. The integration step is unpredictable, but can easily take more time than the original programming. The longer you wait to integrate, the more it costs and the more unpredictable the cost becomes.

The most common style of continuous integration is asynchronous. I check in my changes. Soon thereafter, the build system notices the change and starts to build and test. If there are problems; I am notified by email, text message, or (most coolly) a glowing red lava lamp.

I prefer a synchronous model in which my partner and I integrate after each pair-programming episode, no more than a couple of hours. We wait for the build to complete and the entire test suite to run with no regressions before proceeding.

Asynchronous integrations are a big improvement on daily builds (especially without automated tests), but they don't have the inherent reflection time built into the synchronous style. Waiting for the compiler and the tests is a natural time to talk about what we've just done together and how we might have done it better. Synchronous builds also create positive pressure for a short, clear feedback cycle. When I get notified of a problem half an hour after starting a new task; I waste a lot of time remembering what I was doing, fixing the problem, and then finding my place in the interrupted task.

Integrate and build a complete product. If the goal is to burn a CD, burn a CD. If the goal is to deploy a web site,

deploy a web site, even if it is to a test environment. Continuous integration should be complete enough that the eventual first deployment of the system is no big deal.

Test-First Programming

Write a failing automated test before changing any code. Test-first programming addresses many problems at once:

- Scope creep It's easy to get carried away programming and put in code "just in case." By stating explicitly and objectively what the program is supposed to do, you give yourself a focus for your coding. If you really want to put that other code in, write another test after you've made this one work.

- Coupling and cohesion If it's hard to write a test, it's a signal that you have a design problem, not a testing problem. Loosely coupled, highly cohesive code is easy to test.

- Trust It's hard to trust the author of code that doesn't work. By writing clean code that works and demonstrating your intentions with automated tests, you give your teammates a reason to trust you.

- Rhythm It's easy to get lost for hours when you are coding. When programming test-first, it's clearer what to do next: either write another test or make the broken test work. Soon this develops into a natural and efficient rhythmtest, code, refactor, test, code, refactor.

The XP community hasn't done much exploration of alternatives to tests for verifying the behavior of the system. Tools like static analysis and model checking could be used test-first style. You start with a "test" that says, for example, that there are no deadlocks in the system. After

every change, you verify again that there are no deadlocks. The static analysis tools I've seen aren't intended to be used this way. They run too slowly to be part of the minute-by-minute cycle of programming. However, this seems to be merely a matter of focus, not a fundamental limitation.

Another refinement of test-first programming is continuous testing, first reported by David Saff and Michael Ernst in "An Experimental Evaluation of Continuous Testing During Development," and also explored in Erich Gamma's and my book *Contributing to Eclipse*. In continuous testing the tests are run on every program change, much as an incremental compiler is run on every change to the source code. Test failures are reported in the same format as compiler errors. Continuous testing reduces the time to fix errors by reducing the time to discover them. The tests have to run quickly, however.

The tests you write while coding test-first have the limitation that they take a microview of the program: do these two objects work well together? As your experience grows, you'll be able to squeeze more and more reassurance into these tests. Because of their limited scope, these tests tend to run very fast. You can run thousands of them as part of the Ten-Minute Build.

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. When your understanding of the best possible design leaps forward, work gradually but persistently to bring the design back into alignment with your understanding.

I was taught exactly the opposite of this strategy in school: "Put in all the design you can before you begin implementation because you'll never get another chance." The intellectual justification for this strategy came from a Barry Boehm study of 1960's defense contracts showing that the cost of fixing defects rose exponentially over time. If the same data also hold for adding features to today's software, then the cost of large-scale design changes should rise dramatically over time. In that case, the most economical design strategy is to make big design decisions early and defer all small-scale decisions until later.

For an assumption that shaped software development orthodoxy for decades, the increasing cost of change over time received little scrutiny. This assumption may no longer be valid. Do changes also increase in cost, the same way defects do? Even assuming changes do increase in cost sometimes, are there conditions under which the cost of changes does not increase? If changes do not grow increasingly expensive, what does that imply about the best way to develop software?

XP teams work hard to create conditions under which the cost of changing the software doesn't rise catastrophically. The automated tests, the continual practice of improving

the design, and the explicit social process all contribute to keep the cost of changes low.

XP teams are confident in their ability to adapt the design to future requirements. Because of this, XP teams can meet their human need for immediate and frequent success as well as their economic need to defer investment to the last responsible moment. Some of the teams who read and applied the first edition of this book didn't get the part of the message about the last *responsible* moment. They piled story on story as quickly as possible with the least possible investment in design. Without daily attention to design, the cost of changes does skyrocket. The result is poorly designed, brittle, hard-to-change systems.

The advice to XP teams is not to minimize design investment over the short run, but to keep the design investment in proportion to the needs of the system so far. The question is not whether or not to design, the question is when to design. Incremental design suggests that the most effective time to design is in the light of experience.

If small, safe steps are *how* to design, the next question is *where* in the system to improve the design. The simple heuristic I have found helpful is to eliminate duplication. If I have the same logic in two places, I work with the design to understand how I can have only one copy. Designs without duplication tend to be easy to change. You don't find yourself in the situation where you have to change the code in several places to add one feature.

As a direction for improvement, incremental design doesn't say that designing in advance of experience is horrible. It says that design done close to when it is used is more efficient. As your expertise grows in making changes to a running system in small, safe steps, you can afford to defer more and more of the design investment. As you do so, the system will get simpler, progress will start sooner, tests will

be easier to write, and because the system is smaller there will be less to communicate with the team.

As more teams invest in daily design, they notice that the changes they are making are similar regardless of the purpose of the system. Refactoring is a discipline of design that codifies these recurring patterns of changes. These refactorings can occur at any level of scale. Few design decisions are difficult to change once made. The result is systems that can start small and grow as needed without exorbitant cost.