Best of Study

Agile Technologies

Module-3

Practicing XP: Thinking: Pair Programming, Energized Work, Informative Workspace, Root Cause Analysis, Retrospectives,

Collaborating: Trust, Sit Together, Real Customer Involvement, Ubiquitous Language, Stand-Up Meetings, Coding Standards, Iteration Demo, Reporting,

Best of Study

Thinking

XP doesn't require experts. It does require a habit of mindfulness. The following contains five practices to help mindful developers excel:

• **Pair programming** doubles the brainpower available during coding, and gives one person in each pair the opportunity to think about strategic, long-term issues.

• **Energized work** acknowledges that developers do their best, most productive work when they're energized and motivated.

• **An informative workspace** gives the whole team more opportunities to notice what's working well and what isn't.

• **Root-cause analysis** is a useful tool for identifying the underlying causes of your problems.

• **Retrospectives** provide a way to analyze and improve the entire development process.

Pair Programming

We help each other succeed.

Pair programming is one of the first things people notice about XP. Two people working at the same keyboard? It's weird. It's also extremely powerful and, once you get used to it, tons of fun.
.

Why Pair?

When you pair, one person codes—the driver. The other person is the navigator, whose job is to think. As navigator, sometimes you think about what the driver is typing.

Sometimes you think about what tasks to work on next and sometimes you think about how your work best fits into the overall design.

This arrangement leaves the driver free to work on the tactical challenges of creating rigorous, syntactically correct code without worrying about the big picture, and it gives the navigator the opportunity to consider strategic issues without being distracted by the details of coding.

Together, the driver and navigator create higher-quality work more quickly than either could produce on their own.

Best of Study

Pairing also reinforces good programming habits. XP's reliance on continuous testing and design refinement takes a lot of self-discipline.

When pairing, you'll have positive peer pressure to perform these difficult but crucial tasks. You'll spread coding knowledge and tips throughout the team.

You'll also spend more time in flow—that highly productive state in which you're totally focused on the code.

Your office mates are far less likely to interrupt you when you're working with someone. When they do, one person will handle the interruption while the other continues his train of thought.

How to Pair

A good rule of thumb is to pair on anything that you need to maintain, which includes tests and the build script.

When you start working on a task, ask another programmer to work with you. If another programmer asks for help, make yourself available.

When you need a fresh perspective, switch partners. Have one person stay on the task and bring the new partner up to speed.

Switch whenever a task is finished. If you're working on a big task, switch within four hours.

When you sit down to pair together, make sure you're physically comfortable. Position your chairs side by side, allowing for each other's personal space, and make sure the monitor is clearly visible. When you're driving, place the keyboard directly in front of you. Keep an eye out for this one—for some reason, people pairing tend to contort themselves to reach the keyboard and mouse rather than moving them closer.

Driving and Navigating

Navigators have more time to think than drivers do. The situation will be reversed when you navigate.

When navigating, expect to feel like you want to step in and take the keyboard away from your partner. Relax; your driver will often communicate an idea with both words and code.

He'll make typos and little mistakes—give him time to correct them himself. Use your extra brainpower to think about the greater picture. What other tests do you need to

Best of Study

write? How does this code fit into the rest of the system? Is there duplication you need to remove? Can the code be clearer? Can the overall design be better?
As navigator, help your driver be more productive. Think about what's going to happen next and be prepared with suggestions.

Rather than interrupting the driver when I think of an issue, write your ideas on the index card and wait for a break in the action to bring them up. At the end of the pairing session,

Pairing Stations

To enjoy pair programming, good pairing stations are essential. You need plenty of room for both people to sit side by side.

The best ones are just simple folding tables found at any good office supply store. They should be six feet long, so that two people can sit comfortably side by side, and at least four feet deep. Each table needs a high-powered development workstation.

Plug in two keyboards and mice so each person can have a set. Splurge on large monitors so that both people can see clearly.

Challenges

Pairing can be uncomfortable at first, as it may require you to collaborate more than you're used to. These feelings are natural and typically go away after a month or two, but you have to face some challenges.

Comfort

Pairing is no fun if you're uncomfortable. When you sit down to pair, adjust your position and equipment so you can sit comfortably. Make sure there's room for your legs, feet, and knees.

When you start to pair, discuss your personal space needs and ask about your partner's.

Similarly, while it goes without saying that personal hygiene is critical, remember that strong flavors such as coffee, garlic, onions, and spicy foods can lead to foul breath.

Mismatched skills

Pairing is a collaboration between peers, but sometimes a senior developer will pair with a junior developer. Rather than treating these occasions as student/teacher situations, restore the peer balance by creating opportunities for both participants to learn.

Communication style

Best of Study

New drivers sometimes have difficulty involving their partners; they can take over the keyboard and shut down communication. To practice communicating and switching roles while pairing, consider ping-pong pairing. In this exercise, one person writes a test. The other person makes it pass and writes a new test. Then the first person makes it pass and repeats the process by writing another test.

Different people have different thresholds, so pay attention to how your partner receives your comments.

Results

When you pair program well, you find yourself focusing intently on the code and on your work with your partner. You experience fewer interruptions and distractions. When interrupted, one person deals with the problem while the other continues working. Afterward, you slide back into the flow of work immediately. At the end of the day, you feel tired yet satisfied.

The team as a whole enjoys higher quality code. Technical debt decreases. Knowledge travels quickly through the team, raising everyone's level of competence.

Contraindications

Pairing requires a comfortable work environment. Most offices and cubicles just aren't set up that way. If your workspace doesn't allow programmers to sit side by side comfortably, either change the workspace or don't pair program.

Similarly, if your team doesn't sit together, pairing may not work for you. Although you can pair remotely, it's not as good as in-person.

Alternatives

If you cannot pair program, you need alternatives. Formal code inspections can reduce defects, improve quality, and support self-discipline.

If you're going to use inspections in place of pairing, add some sort of support mechanism to help them take place.

Inspections alone are unlikely to share knowledge as thoroughly as collective code ownership requires. If you cannot pair program, consider avoiding collective ownership, at least at first.

If you'd still like to have collective code ownership, you need an alternative mechanism for sharing knowledge about the state of the codebase.

Best of Study

Energized Work

XP's practice of energized work recognizes that professionals can do good work under difficult circumstances, they do their best, most productive work when they're energized and motivated.

How to Be Energized

One of the simplest ways to be energized is to take care of yourself. Go home on time every day. Spend time with family and friends and engage in activities that take your mind off of work. Eat healthy foods, exercise, and get plenty of sleep.

While at work, give it your full attention. Turn off interruptions such as email and instant messaging. Silence your phones. Ask your project manager to shield you from unnecessary meetings and organizational politics.

This isn't easy. Energized work requires a supportive workplace and home life. It's also a personal choice; there's no way to force someone to be energized.

Supporting Energized Work

One of my favorite techniques as a coach is to remind people to go home on time. Tired people make mistakes and take shortcuts. The resulting errors can end up costing more than the work is worth. This is particularly true when someone is sick; in addition to doing poor work, she could infect other people.

Pair programming is another way to encourage energized work. It encourages focus like no other practice I know. After a full day of pairing, you'll be tired but satisfied. It's particularly useful when you're not at your best: pairing with someone who's alert can help you stay focused.

Having healthy food available in the workplace is another good way to support energized work. Breakfast really is the most important meal of the day

The nature of the work also makes a difference.

Creating and communicating this vision is the product manager's responsibility.

Nothing destroys morale faster than being held accountable for an unachievable goal.

The planning game addresses this issue by combining customer value with developer estimates to create achievable plans.

Best of Study

Speaking of plans, every organization has some amount of politics. Sometimes, politics lead to healthy negotiation and compromising. Other times, they lead to unreasonable demands and blaming. The project manager should deal with these politics, letting the team know what's important and shielding them from what isn't.

The project manager can also help team members do fulfilling work by pushing back unnecessary meetings and conference calls.

In an environment with a lot of external distractions, consider setting aside core hours each day—maybe just an hour or two to start—during which everyone agrees not to interrupt the team.

Finally, jelled teams have a lot of energy. They're a lot of fun, too. You can recognize a jelled team by how much its members enjoy spending time together. They go to lunch together, share in-jokes, and may even socialize outside of work.

Taking Breaks

When you make more mistakes than progress, it's time to take a break.

After a break or a good night's sleep, usually we see mistake right away. Sometimes a snack or walk around the building is good enough.

You can usually tell when somebody needs a break. Angry concentration, cursing at the computer, and abrupt movements are all signs.

Suggesting a break requires a certain amount of delicacy. If someone respects you as a leader, then you might be able to just tell him to stop working. Otherwise, get him away from the problem for a minute so he can clear his head.

Try asking him to help you for a moment, or to take a short walk with you to discuss some issue you're facing.

Results

When your team is energized, there's a sense of excitement

You make consistent progress every week and feel able to maintain that progress indefinitely.

You value health over short-term progress and feel productive and successful.

Contraindications

Energized work is not an excuse to goof off. Generate trust by putting in a fair day's work.

Best of Study

Some organizations may make energized work difficult. If your organization uses the number of hours worked as a yardstick to judge dedication, you may be better off sacrificing energized work and working long hours. The choice between quality of life and career advancement is a personal one that only you and your family can make.

Alternatives

If your organization makes energized work difficult, mistakes are more likely.

Pair programming can help tired programmers stay focused and catch each other's errors. Additional testing may be necessary to find the extra defects.

The extreme form of this sort of organization is the death march organization, which requires employees to work extensive overtime week after week. Sadly, "Death march projects are the norm, not the exception".

Informative Workspace

Just as a pilot surrounds himself with information necessary to fly a plane, arrange your workspace with information necessary to steer your project: create an informative workspace.

An informative workspace broadcasts information into the room. When people take a break, they will sometimes wander over and stare at the information surrounding them.

An informative workspace also allows people to sense the state of the project just by walking into the room. It conveys status information without interrupting team members and helps improve stakeholder trust.

Subtle Cues

The essence of an informative workspace is information. One simple source of information is the feel of the room.

A healthy project is energized. People work together, and make the occasional joke. It's not rushed or hurried, but it's clearly productive. When a pair needs help, other pairs notice, lend their assistance, and then return to their tasks. When a pair completes something well, everyone celebrates for a moment.

An unhealthy project is quiet and tense. Team members don't talk much, if at all.

People live by the clock, punching in and punching out—or worse, watching to see who is the first one to dare to leave.

Best of Study

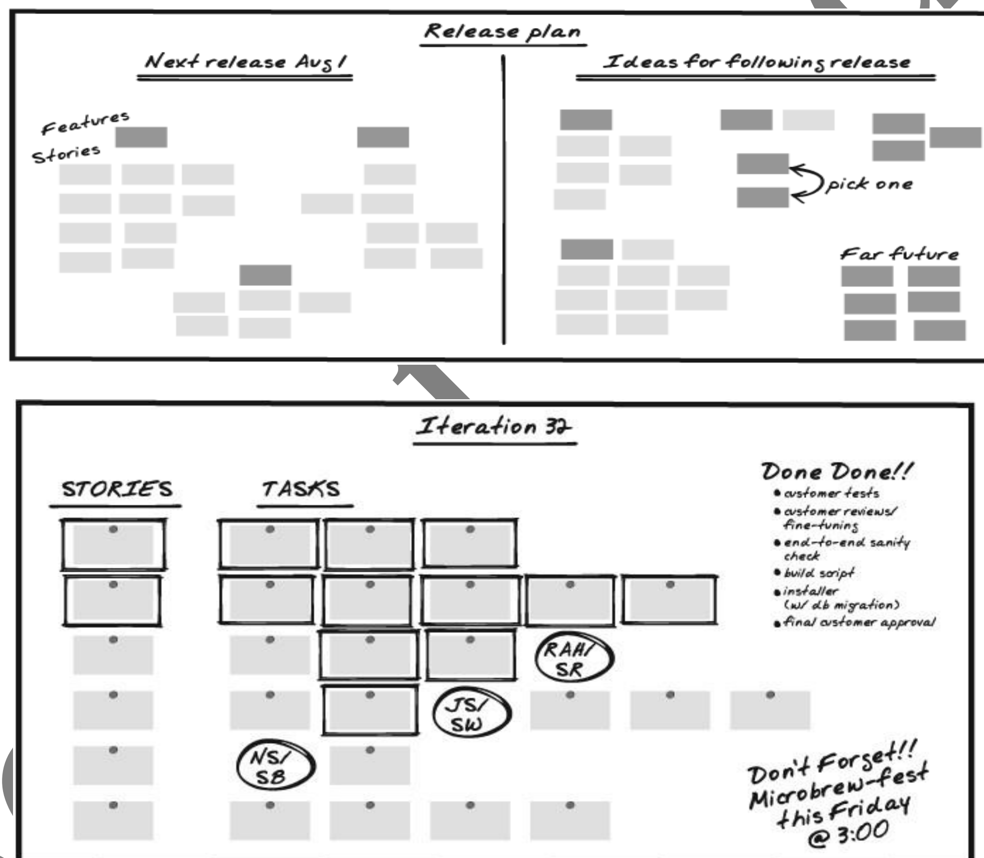
An informative workspace also provides ways for people to communicate. This usually means plenty of whiteboards around the walls and stacks of index cards. A collaborative design sketch on a whiteboard can often communicate an idea far more quickly and effectively than a half-hour PowerPoint presentation.

Big Visible Charts

An essential aspect of an informative workspace is the big visible chart. The goal of a big visible chart is to display information so simply and unambiguously that it communicates even from across the room.

The iteration and release planning boards are ubiquitous examples of such a chart.

For examples, see the release planning board shown in below Figure 1 and the iteration planning board shown in Figure 2.



Hand-Drawn Charts

Avoid the reflexive temptation to computerize your charts. The benefits of the informative workspace stem from the information being constantly visible from everywhere in the room. It's difficult and expensive for computerized charts to meet that criterion; you'd have to install plasma screens or projectors everywhere.

Even if you can afford big screens everywhere, you will constantly change the types of charts you display. This is easier with flip charts and whiteboards than with computers, as creating or modifying a chart is as simple as drawing with pen and paper.

Process Improvement Charts

One type of big visible chart measures specific issues that the team wants to improve.

Unlike the planning boards or team calendar, which stay posted, post these charts only as long as necessary.

Create process improvement charts as a team decision, and maintain them as a team responsibility. When you agree to create a chart, agree to keep it up-to-date. For some charts, this means taking 30 seconds to mark the board when the status changes.

Each team member should update his own status. Some charts involve collecting some information at the end of the day. For these, collectively choose someone to update the chart.

XP teams have successfully used charts to help improve:

- Amount of pairing, by tracking the percentage of time spent pairing versus the percentage of time spent flying solo

- Pair switching, by tracking how many of the possible pairing combinations actually paired during each iteration

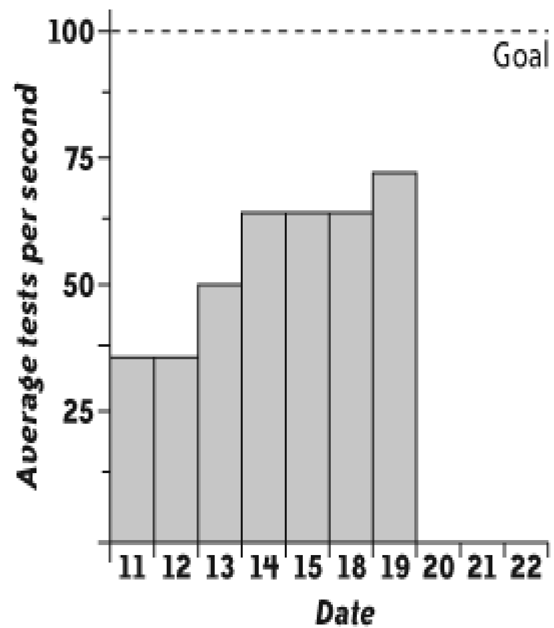
- Build performance, by tracking the number of tests executed per second

- Support responsiveness, by tracking the age of the oldest support request

Best of Study

	MO	JS	SW	NS	MV	SS
MO						
JS						
SW	✓	✓				
NS		✓				
MV				✓		
SS		✓		✓	✓	

(a) Pair combinations



(b) Tests per second

Results

When you have an informative workspace, you have up-to-the-minute information about all the important issues your team is facing. You know exactly how far you've come and how far you have to go in your current plan, you know whether the team is progressing well or having difficulty, and you know how well you're solving problems.

Contraindications

If your team doesn't sit together in a shared workspace, you probably won't be able to create an effective informative workspace.

Alternatives

If your team doesn't sit together, but has adjacent cubicles or offices, you might be able to achieve some of the benefits of an informative workspace by posting information in the halls or a common area. Teams that are more widely distributed may use electronic tools supplemented with daily stand-up meetings.

Root-Cause Analysis

We prevent mistakes by fixing our process. When I hear about a serious mistake on my project, my natural reaction is to get angry or frustrated. I want to blame someone for screwing up.

Best of Stu

Unfortunately, this response ignores the reality. If something can go wrong, it will. People are, well, people. Everybody makes mistakes. I certainly do. Aggressively laying blame might cause people to hide their mistakes, or to try to pin them on others, but this dysfunctional behaviour won't actually prevent mistakes.

Instead of getting angry, try to remember: everybody is doing the best job they can given their abilities and knowledge.

Rather than blaming people, blame the process. What is it about the way we work that allowed this mistake to happen? How can we change the way we work so that it's harder for something to go wrong? This is root-cause analysis.

How to Find the Root Cause

A classic approach to root-cause analysis is to ask "why" five times. Here's a real-world example.

Problem: When we start working on a new task, we spend a lot of time getting the code into a working state.
Why? Because the build is often broken in source control.
Why? Because people check in code without running their tests.
It's easy to stop here and say, "Aha! We found the problem. People need to run their tests before checking in." That is a correct answer, as running tests before check-in is part of continuous integration. But it's also already part of the process. People know they should run the tests, they just aren't doing it.

Dig deeper. Why don't they run tests before checking in? Because sometimes the tests take longer to run than people have available.

Why do the tests take so long? Because tests spend a lot of time in database setup and teardown.

Why? Because our design makes it difficult to test business logic without touching the database.

Asking "why" five times reveals a much more interesting answer than "people aren't running tests." It helps you move away from blaming team members and toward identifying an underlying, fixable problem. In this example, the solution is clear, if not easy: the design needs improvement.

How to Fix the Root Cause

Root-cause analysis is a technique you can use for every problem you encounter.

Best of Study

You can ask yourself "why" at any time. You can even fix some problems just by improving your own work habits.

More often, however, fixing root causes requires other people to cooperate. If your team has control over the root cause, gather the team members, share your thoughts, and ask for their help in solving the problem.

If the root cause is outside the team's control entirely, then solving the problem may be difficult or impossible. For example, if your problem is "not enough pairing" and you identify the root cause as "we need more comfortable desks," your team may need the help of Facilities to fix it.

In this case, solving the problem is a matter of coordinating with the larger organization. Your project manager should be able to help. In the meantime, consider alternate solutions that are within your control.

When Not to Fix the Root Cause

When you first start applying root-cause analysis, you'll find many more problems than you can address simultaneously. Work on a few at a time. I like to chip away at the biggest problem while simultaneously picking off low-hanging fruit.

Over time, work will go more smoothly. Mistakes will become less severe and less frequent.

Eventually—it can take months or years—mistakes will be notably rare. At this point, you may face the temptation to over apply root-cause analysis. Beware of thinking that you can prevent all possible mistakes. Fixing a root cause may add overhead to the process. Before changing the process, ask yourself whether the problem is common enough to warrant the overhead.

Results

When root-cause analysis is an instinctive reaction, your team values fixing problems rather than placing blame. Your first reaction to a problem is to ask how it could have possibly happened. Rather than feeling threatened by problems and trying to hide them, you raise them publicly and work to solve them.

Contraindications

The primary danger of root-cause analysis is that, ultimately, every problem has a cause outside of your control.

Don't use this as an excuse not to take action. If a root cause is beyond your control, work with someone (such as your project manager) who has experience coordinating with other groups.

Best of Study

In the meantime, solve the intermediate problems. Focus on what is in your control.

If your efforts are called "disruptive" or a "waste of time," you may be
better off avoiding root-cause analysis.

Alternatives

You can always perform root-cause analysis in the privacy of your thoughts. You'll probably find that a lot of causes are beyond your control. Try to channel your frustration into energy for fixing processes that you can influence.

Retrospectives

Types of Retrospectives

The most common retrospective, the iteration retrospective, occurs at the end of every iteration. In addition to **iteration retrospectives** we have **release retrospectives**, **project retrospectives**, and **surprise retrospectives** (conducted when an unexpected event changes your situation).

How to Conduct an Iteration Retrospective

Anybody can facilitate an iteration retrospective if the team gets along well. An experienced, neutral facilitator is best to start with.

Everyone on the team should participate in each retrospective. In order to give participants a chance to speak their minds openly, non-team members should not attend.

We time-box retrospectives to exactly one hour.

Try to keep the following schedule in mind as we conduct a retrospective. Don't try to match the schedule exactly; let events follow their natural pace:
1. Norm Kerth's Prime Directive
2. Brainstorming (30 minutes)
3. Mute Mapping (10 minutes)
4. Retrospective objective (20 minutes)

Step 1: The Prime Directive

The team should never use the retrospective to place blame or attack individuals.

Best of Study

Norm Kerth's Prime Directive. I write it at the top of the whiteboard:

Regardless of what we discover today, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand.

I ask each attendee in turn if he agrees to the Prime Directive and wait for a verbal "yes." If not, I ask if he can set aside his scepticism just for this one meeting. If an attendee still won't agree, I won't conduct the retrospective.

Step 2: Brainstorming

If everyone agrees to the Prime Directive, hand out index cards and pencils, then write the following headings on the whiteboard:

- Enjoyable
- Frustrating
- Puzzling
- Same
- More
- Less

Ask the group to reflect on the events of the iteration and brainstorm ideas that fall into these categories. Think of events that were enjoyable, frustrating, and puzzling, and consider what you'd like to see increase, decrease, and remain the same. Write each idea on a separate index card. As facilitator, you can write down your ideas, too—just be careful not to dominate the discussion.
If people are reluctant to say what they really think, try reading the cards anonymously. Ask people to read out their cards as they finish each one, then hand them in. Stick the cards up on the board under their headings.

If people have trouble getting started, describe what happened during the iteration. ("Wednesday, we had our planning session. .. ") This approach takes longer, but it might be a good way to jump-start things when you first start doing retrospectives.

As people read their cards, others will come up with new ideas. The conversation will feed on itself.

Step 3: Mute Mapping

Mute mapping is a variant of affinity mapping in which no one speaks. It's a great way to categorize a lot of ideas quickly.

You need plenty of space for this. Invite everyone to stand up, go over to the whiteboard, and slide cards around. There are three rules:

Best of Study

1. Put related cards close together.
2. Put unrelated cards far apart.
3. No talking.

If two people disagree on where to place a card, they have to work out a compromise without talking.

This exercise should take about 10 minutes, depending on the size of the team. As before.

Once mute mapping is complete, there should be clear groups of cards on the whiteboard. Ask everyone to sit down, then take a marker and draw a circle around each group.

Don’t try to identify the groups yet; just draw the circles

Each circle represents a category. You can have as many as you need. Once you have circled the categories, read a sampling of cards from each circle and ask the team to name the category

Finally, after you have circled and named all the categories, vote on which categories should be improved during the next iteration

Give each person five votes. Participants can put all their votes on one category if they wish, or spread their votes amongst several categories.

Step 4: Retrospective Objective

After the voting ends, one category should be the clear winner.

Discard the cards from the other categories.

Now that the team has picked a category to focus on, it’s time to come up with options for improving it. This is a good time to apply your **root-cause analysis skills**. Read the cards in the category again, then brainstorm some ideas.

Don’t be too detailed when coming up with ideas for improvement. A general direction is good enough. For example, if “pairing” is the issue, then “switching pairs more often” could be one suggestion, “ping-pong pairing” could be another, and switching at specific times” could be a third.

Best of Study

When you have several ideas, ask the group which one they think is best. If there isn't a clear consensus, vote.

This final vote is your retrospective objective.

After the Retrospective

The retrospective serves two purposes: sharing ideas gives the team a chance to grow closer and coming up with a specific solution gives the team a chance to improve.

Results

When your team conducts retrospectives well, your ability to develop and deliver software steadily improves. The whole team grows closer and more cohesive, and each group has more respect for the issues other groups face. You are honest and open about your successes and failures and are more comfortable with change.

Contraindications

The biggest danger in a retrospective is that it will become a venue for acrimony rather than for constructive problem solving. A skilled facilitator can help prevent this, but you probably don't have such a facilitator on hand. Be very cautious about conducting retrospectives if some team members tend to lash out, attack, or blame others.

Alternatives

There are many ways to conduct retrospectives.

Some organizations define organization-wide processes. Others assign responsibility for the process to a project manager, technical lead, or architect. Although these approaches might lead to a good initial process, they don't usually lead to continuous process improvement.

Collaborating

Trust

We work together effectively and without fear.

When a group of people comes together to work as a team, they go through a series of group dynamics known as "**Forming, Storming, Norming, and Performing**"

It takes the team some time to get through each of these stages.

The team jells. Productivity shoots up. They do really amazing work.

Best of Study

What does it take to achieve this level of productivity? The team must take joint responsibility for their work.

When one member of a team encounters a question that she cannot answer, she doesn't hesitate to ask someone who does know the answer.

Trust is essential for the team to perform this well. You need to trust that taking time to help others won't make you look unproductive. You need to trust that you'll be treated with respect when you ask for help or disagree with someone.

Here are some strategies for generating trust in your XP team.

Team Strategy #1: Customer-Programmer Empathy

Customers often feel that programmers don't care enough about their needs and deadlines, some of which, if missed, could cost them their jobs.

Programmers often feel forced into commitments they can't meet, hurting their health and relationships.

Programmers react by inflating estimates and focusing on technical toys at the expense of necessary features; customers react by ignoring programmer estimates and applying schedule pressure.

"Death march teams are the norm, not the exception... [These teams] are often found working 13- to 14-hour days, six days a week...

Sitting together is the most effective way I know to build empathy. Each group gets to see that the others are working just as hard. Retrospectives also help, if your team can avoid placing blame. Programmers can help by being respectful of customer goals, and customers can help by being respectful of programmer estimates and technical recommendations. All of this is easier with energized work.

Team Strategy #2: Programmer-Tester Empathy

When Programmers tend not to show respect for the testers' abilities, and testers see their mission as shooting down the programmers' work.

Programmers, remember that testing takes skill and careful work, just as programming does.

Take advantage of testers' abilities to find mistakes you would never consider, and thank them for helping prevent embarrassing problems from reaching stakeholders and users.

Best of Study

Testers, focus on the team's joint goal: releasing a great product. When you find a mistake, it's not an occasion for celebration. Remember, too, that everybody makes mistakes, and mistakes aren't a sign of incompetence or laziness.

Team Strategy #3: Eat Together

Another good way to improve team cohesiveness is to eat together. Something about sharing meals breaks down barriers and fosters team cohesiveness. Try providing a free meal once per week. If you have the meal brought into the office, set a table and serve the food family-style to prevent people from taking the food back to their desks. If you go to a restaurant, ask for a single long table rather than separate tables.

Team Strategy #4: Team Continuity

After a project ends, the team typically breaks up. All the wonderful trust and cohesiveness that the team has formed is lost. The next project starts with a brand-new team, and they have to struggle through the four phases of team formation all over again.

Rather than assigning people to projects, assign a team to a project. Have people join teams and stick together for multiple projects.

Some teams will be more effective than others. Take advantage of this by using the most effective teams as a training ground for other teams. Rotate junior members into those teams so they can learn from the best, and rotate experienced team members out to lead teams of their own. If you do this gradually, the team culture and trust will remain intact.

Organizational Strategy #1: Show Some Hustle

In the case of a software team, hustle is energized, productive work. It's the sense that the team is putting in a fair day's work for a fair day's pay. Energized work, an informative workspace, appropriate reporting, and iteration demos all help convey this feeling of productivity.

For Example: Several years ago, I hired a small local moving company to move my belongings from one apartment to another. When the movers arrived, I was surprised to see them hustle—they moved quickly from the van to the apartment and back. This was particularly unexpected because I was paying them by the hour. There was no advantage for them to move so quickly. Those movers impressed me. I felt that they were dedicated to meeting my needs and respecting my pocketbook. If I still lived in that city and needed to move again, I would hire them in an instant. They earned my goodwill—and my trust.

Organizational Strategy #2: Deliver on Commitments

Best of Study

If your stakeholders have worked with software teams before, they probably have plenty of war wounds from slipped schedules, unfixed defects, and wasted money.

In addition, they probably don't know much about software development. That puts them in the uncomfortable position of relying on your work, having had poor results before, and being unable to tell if your work is any better.

Meanwhile, your team consumes thousands of dollars every week in salary and support. How do stakeholders know that you're spending their money wisely? How do they know that the team is even competent?

Stakeholders may not know how to evaluate your process, but they can evaluate results. Two kinds of results speak particularly clearly to them: working software and delivering on commitments.

Fortunately, XP teams demonstrate both of these results every week. You make a commitment to deliver working software when you build your iteration and release plans.

Organizational Strategy #3: Manage Problems

When you encounter a problem, start by letting the whole team know about it. Bring it up by the next stand-up meeting at the very latest. This gives the entire team a chance to help solve the problem.

If the setback is relatively small, you might be able to absorb it into the iteration by using some of your iteration slack. Some problems are too big to absorb no matter how much slack you have. If this is the case, get together as a whole team as soon as possible and replan. You may need to remove an entire story or you might be able to reduce the scope of some stories.

When you've identified a problem, let the stakeholders know about it. But bring bigger problems to stakeholders' attentions right away. The product manager is probably the best person to decide who to talk to and when.

The sooner your stakeholders know about a problem the more time they have to work around it. I include an analysis of the possible solutions as well as their technical costs.

Organizational Strategy #4: Respect Customer Goals

When starting a new XP project, programmers should make an extra effort to welcome the customers. One particularly effective way to do so is to treat customer goals with respect

Another way for programmers to take customer goals seriously is to come up with creative alternatives for meeting those goals. If customers want something that may

Best of Study

take a long time or involves tremendous technical risks, suggest alternate approaches to reach the same underlying goal for less cost.

As programmers and customers have these conversations, barriers will be broken and trust will develop.

Organizational Strategy #5: Promote the Team

You can also promote the team more directly. One team posted pictures and charts on the outer wall of the workspace that showed what they were working on and how it was progressing. Another invited anyone and everyone in the company to attend its iteration demos.

Organizational Strategy #6: Be Honest

In your enthusiasm to demonstrate progress, be careful not to step over the line. Borderline behaviour includes glossing over known defects in an iteration demo, taking credit for stories that are not 100 percent complete, and extending the iteration for a few days in order to finish everything in the plan.

Sit Together

We communicate rapidly and accurately. If you've tried to conduct a team meeting via Speaker phone, you know how much of a difference face-to-face conversations make.

Compared to an in-person discussion, teleconferences are slow and stutter-filled, with uncomfortable gaps in the conversation and people talking over each other.

What you may not have realized is how much this affects your work. Imagine you're a programmer on a non agile team and you need to clarify something in your requirements document in order to finish an algorithm. You fire off an email to your domain expert, Mary, then take a break to stretch your legs and get some coffee.
When you get back, Mary still hasn't responded, so you check out a few technical blogs youve been meaning to read. Half an hour later, your inbox chimes. Mary has responded. Uh-oh... it looks like Mary misunderstood your message and answered the wrong question.
You send another query, but you really can't afford to wait any longer. You take your best guess at the answer—after all, you've been working at this company for a long time, and you know most of the answers—and get back to work.

A day later, after exchanging a few more emails, you've hashed out the correct answer with Mary. It wasn't exactly what you thought, but you were pretty close. You go back and fix your code.

Accommodating Poor Communication

Best of Study

As the distance between people grows, the effectiveness of their communication decreases.

People start guessing to avoid the hassle of waiting for answers. Mistakes appear.

To combat this problem, most development methods attempt to reduce the need for direct communication. It's a sensible response. If questions lead to delays and errors, reduce the need to ask questions!

The primary tools teams use to reduce reliance on direct communication are development phases and work-in-progress documents.

For example, in the requirements phase, business analysts talk to customers and then produce a requirements document. Later, if a programmer has a question, he doesn't need to talk to an expert; he can simply look up the answer in the document.

It's a sensible idea, but it has flaws. it's impossible to anticipate all possible questions. Also, adding up-front documentation phases stretches out the development process.

A Better Way

In XP, the whole team—including experts in business, design, programming, and testing —sits together in a open workspace. When you have a question, you need only turn your head and ask. You get an instant response, and if something isn't clear, you can discuss it at the whiteboard.

Consider the previous story from this new perspective. You're a programmer and you need some information from your domain expert, Mary, in order to code an algorithm.
This time, rather than sending an email, you turn your head. "Mary, can you clarify something for me?"
After some more discussion, the answer is clear. You're a little surprised: Mary's answer was completely different than you expected. It's good that you talked it over. Now, back to work!

Exploiting Great Communication

Sitting together eliminates the waste caused by waiting for an answer, which dramatically improves productivity.

Programmers on XP teams spend a far greater percentage of their time programming. I attribute that to the increased communication effectiveness of sitting together.

Best of Study

Rather than sitting in hour-long meetings, conversations last only as long as needed and involve only the people necessary.

Imagine a team that sits together. Team members are concentrating on their work and talking quietly with their partners. Then somebody mentions something about managing database connections, and another programmer perks up. "Oh, Tom and I refactored the database connection pool last week. You don't need to manage the connections manually anymore."

When team members are comfortable speaking up like this, it happens often and saves time and money every time. There's another hidden benefit to sitting together: it helps teams jell and breaks down us-versus-them attitudes between groups.

Secrets of Sitting Together

To get the most out of sitting together, be sure you have a complete team. It's important that people be physically present to answer questions. If someone must be absent often —product managers tend to fall into this category—make sure that someone else on the team can answer the same questions.

Similarly, sit close enough to each other that you can have a quick discussion without getting up from your desk or shouting.

Available instant help doesn't do any good if you don't ask for it. Many organizations discourage interruptions

There's no sense in banging your head against a wall when the person with the answer is right across the room. To support this attitude, many teams have a rule: "We must always help when asked."

Making Room

Sitting together is one of those things that's easy to say and hard to do. It's not that the act itself is difficult—the real problem is finding space.

A team that sits in adjacent cubicles can convert them into an adequate shared workspace, but even with cubicles, it takes time and money to hire people to rearrange the walls.

A big conference room is a good alternative.

Designing Your Workspace

Best of Study

Programmers should all sit next to each other because they collaborate moment-to-moment.

Testers should be nearby so programmers can overhear them talk about issues.

Domain experts and interaction designers don't need to be quite so close, but should be close enough to answer questions without shouting.

The product manager and project manager are most likely to have conversations that would distract the team. They should sit close enough to be part of the buzz but not so close that their conversations are distracting.

Be sure that everyone has a space they can call their own. You also need an additional enclosed room with a door, or cubes away from the open workspace, so people can have privacy for personal phone calls and individual meetings.

Results

When your team sits together, communication is much more effective. You stop guessing at answers and ask more questions. You overhear other people's conversations and contribute answers you may not expect.

Real Customer Involvement

We understand the goals and frustrations of our customers and end-users.

In an XP team, on-site customers are responsible for choosing and prioritizing features. The value of the project is in their hands. This is a big responsibility—as an on-site customer, how do you know which features to choose?

Some of that knowledge comes from your expertise in the problem domain and with previous versions of the software.

In-House Custom Development

In-house custom development occurs when your organization asks your team to build something for the organization's own use.

In this environment, the team has multiple customers to serve: the executive sponsor who pays for the software and the end-users who use the software. Their goals may not be in alignment

Despite this challenge, in-house custom development makes it easy to involve real customers because they're easily accessible. The best approach is to bring your customers onto the team —to turn your real customers into on-site customers.

Best of Study

Outsourced Custom Development

Outsourced custom development is similar to in-house development, but you may not have the connections that an in-house team does.

If you can't bring real customers onto the team, make an extra effort to involve them.

Meet in person with your real customers for the first week or two of the project so you can discuss the project vision and initial release plan.

Try to meet face-to-face at least once per month to discuss plans. If you are so far apart that monthly meetings aren't feasible, meet at least once per release.

Ubiquitous Language

The Domain Expertise Conundrum

One of the challenges of professional software development is that programmers aren't necessarily experts in the areas for which they write software.

It's a conundrum. The people who are experts in the problem domain—the domain experts— are rarely qualified to write software. The people who are qualified to write software—the programmers—don't always understand the problem domain.

Overcoming this challenge is, fundamentally, an issue of communication.

The challenge is communicating that information clearly and accurately.

Two Languages

Imagine for a moment that you're driving to a job interview. You forgot your map, so you're getting directions from a friend on your cell phone (hands free, of course!):

The problem in this scenario is that you and your friend are speaking two different languages. You're talking about what you see on the road and your friend is talking about what he sees on his map. You need to translate between the two, and that adds delay and error.

A similar problem occurs between programmers and domain experts. Programmers program in the language of technology: classes, methods, algorithms, and databases. Domain experts talk in the language of their domain: financial models, chip fabrication plants, and the like.

Best of Study

You could try to translate between the two languages, but it will add delays and errors.

Instead, pick just one language for the whole team to use—a ubiquitous language.

How to Speak the Same Language

Programmers should speak the language of their domain experts, not the other way around.

Ubiquitous Language in Code

As a programmer, you might have trouble speaking the language of your domain experts.

When you're working on a tough problem, it's difficult to make the mental translation from the language of code to the language of the domain.

A better approach is to design your code to use the language of the domain. You can name your classes, methods, and variables anything. Why not use the terms that your domain experts use?

This is more than learning the domain to write the software; this is reflecting in code how the users of the software think and speak about their work.

Refining the Ubiquitous Language

The ubiquitous language informs programmers, but the programmers need for rigorous formalization also informs the rest of the team. I often see situations in which programmers ask a question—inspired by a coding problem—that in turn causes domain experts to question some of their assumptions.

Your ubiquitous language, therefore, is a living language. It's only as good as its ability to reflect reality. As you learn new things, improve the language as well. There are three caveats about doing this, however.

First, ensure that the whole team—especially the domain experts—understands and agrees with the changes you're proposing. This will probably require a conversation to resolve any conflicts. Embrace that!

Second, check that the changes clarify your understanding of the business requirements. It may seem clearer to make a change, but the language must still reflect what the users need to accomplish with the software.

Third, update the design of the software with the change. The model and the ubiquitous language must always stay in sync.

Results

Best of Study

When you share a common language between customers and programmers, you reduce the risk of miscommunication. When you use this common language within the design and implementation of the software, you produce code that's easier to understand and modify.

Stand-Up Meetings

XP projects have a more effective mechanism: informative workspaces and the daily stand-up meeting.

How to Hold a Daily Stand-Up Meeting

A stand-up meeting is very simple. At a pre-set time every day, the whole team stands in a circle. One at a time, each person briefly describes new information that the team should know.

Some teams use a formal variant of the stand-up called the Daily Scrum.

In the Daily Scrum, participants specifically answer three questions:

1. What did I do yesterday?
2. What will I do today?
3. What problems are preventing me from making progress?

One problem with stand-up meetings is that they interrupt the day. This is a particular problem for morning stand-ups; because team members know the meeting will interrupt their work, they sometimes wait for the stand-up to end before starting to work.

You can reduce this problem by moving the stand-up to later in the day, such as just before lunch.

Be Brief

The purpose of a stand-up meeting is to give everybody a rough idea of where the team is. It's not to give a complete inventory of everything happening in the project.

That's why we stand: our tired feet remind us to keep the meeting short. Each person usually only needs to say a few sentences about her status. Thirty seconds per person is usually enough. More detailed discussions should take place in smaller meetings with only the people involved.

Brevity is a tough art to master. To practice, try writing your statement on an index card in advance, then read from the card during the stand-up.
Another approach is to time-box the stand-up. Set a timer for 5 or 10 minutes, depending on the size of the team. When the timer goes off, the meeting is over, even if there are some people who haven't spoken yet. At first, you'll find that the meeting is cut

Best of Study

off prematurely, but the feedback should help people learn to speak more briefly after a week or two.

Results

When you conduct daily stand-up meetings, the whole team is aware of issues and challenges that other team members face, and it takes action to remove them. Everyone knows the project's current status and what the other team members are working on.

Coding Standards

XP suggests creating a coding standard: guidelines to which all developers agree to adhere when programming.

Beyond Formatting

We agreed on how we should and shouldn't handle exceptions, what to do about debugging code, and when and where to log events.

These standards helped us far more than a consistent formatting style would have because each one had a concrete benefit. Perhaps that's why we were able to agree on them when we couldn't agree on formatting styles.

How to Create a Coding Standard

It may be one of the first things that programmers do as a team. Over time, you'll amend and improve the standards.

Applying two guidelines:

1. Create the minimal set of standards you can live with.
2. Focus on consistency and consensus over perfection.

Hold your first discussion of coding standards during the first iteration.

The best way to start your coding standard is often to select an industry-standard style guide for your language.

Starting points include:

• Development practices

• Tools, key bindings, and IDE

• File and directory layout

• Build conventions

Best of Study

- Error handling and assertions
- Approach to events and logging
- Design conventions (such as how to deal with null references)

Limit your initial discussion to just one hour. Write down what you agree on. If you disagree about something, move on. You can come back to it later.

Consider bringing in a professional facilitator to redirect the discussion to your team goals when the things get heated.

Plan to hold another one-hour coding standard meeting a few days later, and another one a few weeks after that. The long break will allow you to learn to work together and to try out your ideas in practice. If there's still disagreement, experiment with one approach or the other, then revisit the issue.

No matter what standards you choose, someone will be probably unhappy with some guideline.

Adhering to the Standard

People make mistakes. Pair programming helps developers catch mistakes and maintain self-discipline. It provides a way to discuss formatting and coding questions not addressed by the guidelines.

Collective code ownership also helps people adhere to the standard, because many different people will edit the same piece of code. Code tends to settle on the standard as a result.

Start by talking with your colleague alone to see if there's a disagreement. Take an attitude of collaborative problem solving

If the objector agrees with the standard but isn't applying it, it's possible that the standard isn't appropriate in every situation.
During this discussion, you may learn that the objector doesn't understand the standard. By this time, you should be in a good situation to discuss the standard and what it means.

Results

When you agree on coding standards and conventions, you improve the maintainability and readability of your code. You can take up different tasks in different subsystems with greater ease. Pair programming moves much more smoothly and you look for ways to improve the express ability and robustness of your code as you write it.

Best of Study

Iteration Demo

An XP team produces working software every week, starting with the very first week.

Programmers need discipline to keep the code clean so they can continue to make progress.

Customers need discipline to fully understand and communicate one set of features before starting another.

Testers need discipline to work on software that changes daily. The rewards for this hard work are significantly reduced risk, a lot of energy and fun, and the satisfaction of doing great work and seeing progress.

The iteration demo is a powerful way to do so. First, it's a concrete demonstration of the team's progress. The team is proud to show off its work, and stakeholders are happy to see progress.

Second, the demos help the team be honest about its progress. Iteration demos are open to all stakeholders, and some companies even invite external customers to attend.

Finally, the demo is an opportunity to solicit regular feedback from the customers.

How to Conduct an Iteration Demo

Anybody on the team can conduct the iteration demo, but I recommend that the product manager do so. He has the best understanding of the stakeholders′ point of view and speaks their language.

The whole team, key stakeholders, and the executive sponsor should attend as often as possible. Include real customers when appropriate.

The entire demo should take about 10 minutes.

If it runs long, I look for ways to bring it to a close before it reaches half an hour.

Both the product manager and the demo should be available for further discussion and exploration after the meeting.

Once everyone is together, briefly describe the features scheduled for the iteration and their value to the project. If the plan changed during the middle of the iteration, explain what happened

After your introduction, go through the list of stories one at a time. Read the story, add any necessary explanation, and demonstrate that the story is finished. Use customer tests to demonstrate stories without a user interface.

Best of Study

Once the demo is complete, tell stakeholders how they can run the software themselves. Make an installer available on the network, or provide a server for stakeholder use, or something similar.

Two Key Questions

At the end of the demo, ask your executive sponsor two key questions:*
1. Is our work to date satisfactory?
2. May we continue?
These questions help keep the project on track and remind your sponsor to speak up if she's unhappy.
Sometimes, she may answer "no" to the first question, or she may answer "yes" but be clearly reluctant. These are early indicators that something is going wrong. After the demo, talk with your sponsor and find out what she's unhappy about. Take immediate action to correct the problem.

In rare cases, the executive sponsor will answer "no" to the second question. You should never hear this answer—it indicates a serious breakdown in communication.

Try to find out what went wrong, and include your sponsor in the project retrospective, if possible.

Weekly Deployment Is Essential

The iteration demo isn't just a dog and pony show; it's a way to prove that you're making real progress every iteration. Always provide an actual release that stakeholders can try for themselves after the demo. Even if they are not interested in trying a demo release, create it anyway; with a good automated build, it takes only a moment. If you can't create a release, your project may be in trouble.

One of the biggest schedule risks in software is the hidden time between "we're done" and "we've shipped." Teams often have to spend several extra weeks (or months) after the end of the planned schedule to make their product buildable and shippable. Releasing a usable demo every iteration mitigates this risk.

The weekly rhythm of iteration demos and stakeholder releases is an excellent way to keep the code releasable.

Results

When you conduct a weekly iteration demo and demo release, you instil trust in stakeholders, and the team is confident in its ability to deliver.

Best of Study

Reporting

All the information you need is at your fingertips. Why do you need reports?

The people who aren't on your team, particularly upper management and stakeholders, do. They have a big investment in you and the project, and they want to know how well it's working.

Types of Reports

Progress reports are exactly that: reports on the progress of the team, such as an iteration demo or a release plan.

Management reports are for upper management. They provide high-level information that allows management to analyze trends and set goals.

Progress Reports to Provide

Vision statement

Your on-site customers should create and update a vision statement that describes what you're doing, why you're doing it, and how you'll know if you're successful.

Weekly demo

Nothing is as powerful at demonstrating progress as working software. Invite stakeholders to the weekly iteration demo

Release and iteration plans

The release and iteration planning boards already posted in your workspace provide great detail about progress. Invite stakeholders to look at them any time they want detailed status information.

Burn-up chart

A burn-up chart is an excellent way to get a bird's-eye view of the project .
It shows progress and predicts a completion date.

Roadmap

Best of Study

Some stakeholders may want more detail than the vision statement provides, but not the overwhelming detail of the release and iteration plans. For these stakeholders, consider maintaining a document that summarizes planned releases and the significant features in each one.

Status email

A weekly status email can supplement the iteration demo. I like to include a list of the stories completed for each iteration and their value.

Management Reports to Consider

Whereas progress reports demonstrate that the team will meet its goals, management reports demonstrate that the team is working well.

Productivity

Instead of trying to measure features, measure the team's impact on the business. Create an objective measure of value, such as return on investment. You can base it on revenue, cost savings, or some other valuable result.

This productivity metric reflects that fact. To score well on this metric, you should have a team that includes on-site customers. These customers will figure out what customers or users want and show key stakeholders how to sell or use the software. By doing so, they will help turn technically excellent software into truly valuable software.

Throughput

Throughput is the number of features the team can develop in a particular amount of time.

Defects

Anyone can produce software quickly if it doesn't have to work. Consider counterbalancing your throughput report with defect counts.

Time usage

If the project is under time pressure—and projects usually are—stakeholders may want to
know that the team is using its time wisely. Often, when the team mentions its velocity, stakeholders question it. "Why does it take 6 programmers a week to finish 12 days of work? Shouldn't they finish 30 days of work in that time?"

Best of Study

To keep the burden low programmers, write their times on the back of each iteration task card and hand them in to the project manager for collating into these categories:

• Unaccounted and nonprojected work (time spent on other projects, administration,company-wide meetings, etc.)
• Out of office (vacation and sick days)
• Improving skills (training, research time, etc.)
• Planning (time spent in planning activities, including the retrospective and iteration demo)
• Developing (time spent testing, coding, refactoring, and designing)

Reports to Avoid

Source lines of code (SLOC) and function points Source lines of code (SLOC) and its language-independent cousin, function points, are common approaches to measuring software size. Unfortunately, they're also used for measuring productivity.

Number of stories

Some people think they can use the number of stories delivered each iteration as a measure of productivity. Don't do that. Stories have nothing to do with productivity.

Velocity

If a team estimates its stories in advance, an improvement in velocity may result from an improvement in productivity. Unfortunately, there's no way to differentiate between productivity changes and inconsistent estimates.

Code quality

There's no substitute for developer expertise in the area of code quality. The available code quality metrics, such as cyclomatic code complexity, all require expert interpretation. There is no single set of metrics that clearly shows design or code quality.

Results

Appropriate reporting will help stakeholders trust that your team is doing good work. Over time, the need for reports will decrease, and you will be able to report less information less frequently.

Best of Study