1. **Types of success along with diagram.**

- Figure 1-1. Types of success

- All three types of success are important (see Figure 1-1).
- Without personal success, you'll have trouble motivating yourself and employees.
- Without technical success, your source code will eventually collapse under its own weight.

- Without organizational success, your team may find that they're no longer wanted in the company

Organizational Success

- When business needs change or when new information is discovered, *agile teams change direction to match*. In fact, an **experienced agile team will actually seek out unexpected opportunities to improve its plans**.
- Agile teams ***decrease costs*** as well. They do this partly by *technical excellence*; the best agile projects generate only a **few bugs per month.**
- They *also eliminate waste by cancelling bad projects early* and **replacing expensive development practices with simpler ones.**
- **Agile teams *communicate quickly and accurately,*** and they make **progress even when key individuals are unavailable**. They ***regularly review their process and continually improve their code,*** **making the software easier to maintain and enhance over time.**

Technical Success

- Extreme Programming, the agile method is particularly adept at achieving technical successes. XP programmers work together, which helps them *keep track of the nit-picky* details necessary for great work and ensures that at least *two people review every piece of code*.
- *Programmers continuously integrate their code*, which enables the team to release the software whenever it makes business sense.
- The whole team *focuses on finishing each feature* completely before starting the next, which prevents unexpected delays before release and allows the team to change direction at will.
- In addition to the structure of development, Extreme Programming includes advanced technical practices that lead to technical excellence. The most well-known practice is *test driven development*, which helps programmers write code that does exactly what they think it will.
- XP teams also *create simple, ever-evolving designs* that are easy to modify when plans change.

Personal Success

- Agile development may not satisfy all of your requirements for personal success. However, once you get used to it, you'll probably find a lot to like about it, no matter who you are:
- **Executives and senior management**

  They will appreciate the team's focus on providing a solid return on investment and the software's longevity.
- **Users, stakeholders, domain experts, and product managers**

  They will appreciate their ability to influence the *direction of software development, the team's focus on delivering useful and valuable software, and increased delivery frequency*.
- **Project and product managers**

  They will appreciate their ability to *change direction as business needs change*, *the team's ability to make and meet commitments, and improved stakeholder satisfaction*.
- **Developers**

  They will appreciate their *improved quality of life resulting from increased technical quality, greater influence over estimates and schedules, and team autonomy*.
- **Testers**

  They will appreciate their integration as first-class members of the team, their ability to influence quality at all stages of the project, and more challenging, less repetitious work.

2. Agile values and principles

Figure 2-1. Agile values

Principles behind the Agile Manifesto

We follow these principles:

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

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

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

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

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

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

Working software is the primary measure of progress.

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

Continuous attention to technical excellence and good design enhances agility.

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

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

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

3. **Traditional vs XP Lifecycle**
4. **XP Whole Team**
5. **Team Size**

Team Size

- The guidelines in this book assume teams with 4 to 10 programmers (5 to 20 total team members). For new teams, four to six programmers is a good starting point.
- Applying the staffing guidelines to a team of 6 programmers produces a team that also includes 4 customers, 1 tester, and a project manager, for a total team size of 12 people. Twelve people turns out to be a natural limit for team collaboration.
- XP teams can be as small as one experienced programmer and one product manager, but full XP might be overkill for such a small team. The smallest team I would use with full XP consists of five people: four programmers (one acting as coach)and one product manager (who also acts as project manager, domain expert, and tester).
- A team of this size might find that the product manager is overburdened; if so, the programmers will have to pitch in. Adding adomain expert or tester will help.
- On the other end of the spectrum, starting with 10 programmers produces a 20-person team that includes 6 customers, 3 testers, and a project manager.
- You can create even larger XP teams, but they require special practices that are out of the scope of this book.

- Before you scale your team to more than 12 people, however, remember that large teams incur extra communication and process overhead, and thus reduce individual productivity. The combined overhead might even reduce overall productivity.
- If possible, hire more experienced, more productive team members rather than scaling to a large team.
- A 20-person team is advanced XP. Avoid creating a team of this size until your organization has had extended success with a smaller team. If you're working with a team of this size, continuous review, adjustment, and an experienced coach are critical.

Full-Time Team Members

- All the team members should sit with the team full-time and give the project their complete attention. This particularly applies to customers, who are often surprised by the level of involvement XP requires of them.
- Some organizations like to assign people to multiple projects simultaneously.
- You can bring someone in to consult on a problem temporarily. However, while she works with the team, she should be fully engaged and available.

6. Technical Debt, Time-boxing, velocity,

TECHNICAL DEBT

Imagine a customer rushing down the hallway to your desk. “It’s a bug!” she cries, out of breath. “We have to fix it now.” You can think of two solutions: the right way and the fast way. You just know she’ll watch over your shoulder until you fix it. So you choose the fast way, ignoring the little itchy feeling that you’re making the code a bit messier. Technical debt is the total amount of less-than-perfect design and implementation decisions in your project. This includes quick and dirty hacks intended just to get something working right now! and design decisions that may no longer apply due to business changes. Technical debt can even come from development practices such as an unwieldy build process or incomplete test coverage. These dark corners of poor formatting, unintelligible control flow, and insufficient testing breed bugs. The bill for this debt often comes in the form of higher maintenance costs. There may not be a single lump sum to pay, but simple tasks that ought to take minutes may stretch into hours or afternoons. Left unchecked, technical debt grows to overwhelm (have a strong emotional effect) software projects. Software costs millions of dollars to develop, and even small projects cost hundreds of thousands. It’s foolish to throw away that investment and rewrite the software, but it happens all the time. Why? Unchecked technical debt makes the software more expensive to modify than to re-implement. What a waste. XP takes a fanatical approach to technical debt. The key to managing it is to be constantly vigilant. Avoid shortcuts, use simple design, refactor relentlessly(continuosly) . in short, apply XP’s development practices

TIMEBOXING

Some activities invariably stretch to fill the available time. There’s always a bit more polish you can put on a program or a bit more design you can discuss in a meeting Recognizing the point at which you have enough information is not

easy. If you use timeboxing, you set aside a specific block of time for your research or discussion and stop when your time is up, regardless of your progress. This is both difficult and valuable. It's difficult to stop working on a problem when the solution may be seconds away. Timeboxing meetings, for example, can reduce wasted discussion

VELOCITY

In well-designed systems, programmer estimates of effort tend to be consistent but not accurate. Programmers also experience interruptions that prevent effort estimates from corresponding to calendar time. Velocity is a simple way of mapping estimates to the calendar. It's the total of the estimates for the stories finished in an iteration. In general, the team should be able to achieve the same velocity in every iteration. This allows the team to make iteration commitments and predict release dates. The units measured are deliberately vague(indefinite); velocity is a technique for converting effort estimates to calendar time and has no relation to productivity.

7. For Adopting XP Prerequisites and Recommendations

Adopting XP

Similarly, if you want to practice XP, do everything you can to meet the following prerequisites and recommendations. This is a lot more effective than working around limitations.

Prerequisite #1: Management Support

It's very difficult to use XP in the face of opposition from management. Active support is best.

To practice XP you will need the following:

- A common workspace with pairing stations
- Team members solely allocated to the XP project
- A product manager, on-site customers, and integrated testers

You will often need management's help to get the previous three items. In addition, the more management provides the following things, the better:

- Team authority over the entire development process, including builds, database schema, and version control
- Compensation and review practices that are compatible with team-based effort
- Acceptance of new ways of demonstrating progress and showing
- Patience with lowered productivity while the team learns

If management isn't supportive...
If you want management to support your adoption of XP, they need to believe in its benefits.
Think about what the decision-makers care about.
What does an organizational success mean to your management?
What does a personal success mean?
How will adopting XP help them achieve those successes?
What are the risks of trying XP, how will you mitigate those risks, and what makes XP worth the risks?
Talk in terms of your managers' ideas of success, not your own success.
If you have a trusted manager you can turn to, ask for her help and advice. If not, talk to your mentor
If management refuses your overtures, then XP probably isn't appropriate for your team.

Prerequisite #2: Team Agreement
If team members don't want to use XP, it's not likely to work.
XP assumes good faith on the part of team members
—there's no way to force the process on somebody who's resisting it
If people resist...
It's never a good idea to force someone to practice XP against his will. In the best case, he'll find some way to leave the team, quitting if necessary. In the worst case, he'll remain on the team and silently sabotage(destroy) your efforts.
Reluctant sceptics (ask questions or doubt) are OK. If somebody says, "I don't want to practice XP, but I see that the rest of you do, so I'll give it a fair chance for a few months," that's fine. She may end up liking it. If not, after a few months have gone by, you'll have a better idea of what you can do to meet the whole team's needs.

Prerequisite #3: A Colocated Team
XP relies on fast, high-bandwidth communication for many of its practices. In order to achieve that communication, your team members needs to sit together in the same room.
<u>If your team isn't colocated(sharing facility or location)...</u>
Colocation makes a big difference in team effectiveness. Don't assume that your team can't sit together; be sure that bringing the team together is your first option.
That said, it's OK if one or two noncentral team members are off-site some of the time. You'll be surprised, though, at how much more difficult it is to interact with them.
Talk with your mentor about how to best deal with the problem.
If a lot of people are off-site, if a central figure is often absent, or if your team is split across multiple locations, you need help beyond this book

Prerequisite #4: On-Site Customers

On-site customers are critical to the success of an XP team. They, led by the product manager, determine which features the team will develop. In other words, **their decisions determine the value of the software**.

Of all the on-site customers, the product manager is likely the most important. She makes the final determination of value.

A good product manager will choose features that provide value to your organization.

Domain experts, and possibly interaction designers take the place of an upfront requirements phase, sitting with the team to plan upcoming features and answering questions about what the software needs to do.

If your product manager is too busy to be on-site you may be able to ask a business analyst or one of the other on-site customers to act as a proxy.

If your product manager is inexperienced...

This may be OK as long as she has a more experienced colleague she turns to for advice

XP requires that somebody with business expertise take responsibility for determining and prioritizing features.

Prerequisite #5: The Right Team Size

For teams new to XP recommend size is 4 to 6 programmers and no more than 12 people on the team. I also recommend having an even number of programmers so that everyone can pair program .

Teams with fewer than four programmers are less likely to have the intellectual diversity they need. They'll also have trouble using pair programming, an important support mechanism in XP.

Large teams face coordination challenges. Although experienced teams can handle those challenges smoothly, a new XP team will struggle.

If you don't have even pairs...

The easiest solution to this problem is to add or drop one programmer so you have even pairs.

If your team is larger than seven programmers...

The coordination challenges of a large team can make learning XP more difficult. Consider hiring an experienced XP coach to lead the team through the transition.

If your team is smaller than four programmers...

you probably won't be able to pair program much. In this situation, it's best if your team members are conscientious(serious) programmers who are passionate about producing high-quality code.

If you have many developers working solo...

Some organizations—particularly IT organizations—have a lot of small projects rather than one big project. They structure their work to assign one programmer to each project.

Although this approach has the advantage of connecting programmers directly with projects, it has several disadvantages. It's high-risk: every project is the responsibility of one programmer, so that any programmer who leaves orphans a project. Her replacement may have to learn it from first principles.

Code quality can also be a challenge. Projects don't benefit from peer review, so the code is often idiosyncratic(individual).

Junior programmers, lacking the guidance of their more senior peers, create convoluted(complex or difficult to follow), kludgey systems and have few opportunities to learn better approaches.

Combining your programmers into a single team has some drawbacks. The biggest is likely to be a perceived lack of responsiveness. Although projects will be finished more quickly, customers will no longer have a dedicated programmer to talk to about the status of their projects. The team will only work on one project at a time, so other customers may feel they are being ignored.

Prerequisite #6: Use All the Practices

You may be tempted to ignore or remove some XP practices, particularly ones that make team members uncomfortable. Be careful of this.

Nearly every practice directly contributes to the production of valuable software. For example, pair programming supports collective code ownership, which is necessary for
refactoring. Refactoring allows incremental design and architecture. Incremental design and
architecture enables customer-driven planning and frequent releases, which are the key to XP's ability to increase value and deliver successful software.

If practices don't fit...

You may think that some XP practices aren't appropriate for your organization. That may be true, but it's possible you just feel uncomfortable or unfamiliar with a practice. Are you sure the practice won't work, or do you just not want to do it?

: XP will work much better if you give all the practices a fair chance rather than picking and choosing the ones you like. If you're sure a practice won't work, you need to replace it. For example, in order to achieve the benefits of collective code ownership without pair programming, you must provide another way for people to share knowledge about the codebase.

Recommendation #1: A Brand-New Codebase

Easily changed code is vital to XP. If your code is cumbersome to change, you'll have difficulty with XP's technical practices, and that difficulty will spread over into XP's planning practices.
XP teams put a lot of effort into keeping their code clean and easy to change. If you have a brand-new codebase, this is easy to do. If you have to work with existing code, you can still practice XP, but it will be more difficult. Even well-maintained code is unlikely to have the simple design and suite of automated unit tests that XP requires.

Recommendation #2: Strong Design Skills

At least one person on the team —preferably a natural leader—needs to have strong design skills.
It's hard to tell if somebody has strong design skills unless you have strong design skills yourself.

Recommendation #1: A Brand-New Codebase

Easily changed code is vital to XP. If your code is cumbersome to change, you'll have difficulty with XP's technical practices, and that difficulty will spread over into XP's planning practices.
XP teams put a lot of effort into keeping their code clean and easy to change. If you have a brand-new codebase, this is easy to do. If you have to work with existing code, you can still practice XP, but it will be more difficult. Even well-maintained code is unlikely to have the simple design and suite of automated unit tests that XP requires.

Recommendation #2: Strong Design Skills

At least one person on the team —preferably a natural leader—needs to have strong design skills.
It's hard to tell if somebody has strong design skills unless you have strong design skills yourself.

Recommendation #3: A Language That's Easy to Refactor

XP relies on refactoring to continuously improve existing designs, so any language that makes refactoring difficult will make XP difficult. Of the currently popular languages, object-oriented and dynamic languages with garbage collection are the easiest to refactor. C and C++, for example, are more difficult to refactor.

Recommendation #4: An Experienced Programmer-Coach

Some people are natural leaders. They're decisive, but appreciate others' views; competent, but respectful of others' abilities. Team members respect and trust them. You can recognize a leader by her influence—regardless of her title, people turn to a leader for advice.

XP relies on self-organizing teams. This kind of team doesn't have a predefined hierarchy; instead, the team decides for itself who is in charge of what. These roles are usually informal. In fact, in a mature XP team, there is no one leader. Team members seamlessly defer leadership responsibilities from one person to the next, moment to moment, depending on the task at hand and the expertise of those involved.

When your team first forms, though, it won't work together so easily. Somebody will need to help the team remember to follow the XP practices consistently and rigorously.

Your coach also needs to be an experienced programmer so she can help the team with XP's technical practices.

Recommendation #4: An Experienced Programmer-Coach

If your leaders are inexperienced, you may want to try pair coaching. Pick one person who's a good leader and one person who has a lot of experience. Make sure they get along well. Ask the two coaches to work together to help the team remember to practice XP consistently and rigorously.

If you're assigned a poor coach...

Your organization may assign somebody to be coach who isn't a good leader. In this case, if the assigned coach recognizes the problem, pair coaching may work for you.

Recommendation #5: A Friendly and Cohesive Team

XP requires that everybody work together to meet team goals. There's no provision for someone to work in isolation, so it's best if team members enjoy working together.

If your team doesn't get along...

XP requires people to work together. Combined with the pressure of weekly deliveries, this can help team members learn to trust and respect each other. However, it's possible for a team to implode(collapse) from the pressure. Try including a team member who is level-headed and has a calming influence.

8. Why Pair and how to Pair, Pairing Tips, Challenges,

Why Pair?

- Pair programming is all about increasing your brainpower.

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.
- 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. It's a different kind of flow than normal because you're working with a partner. To start with, you'll discover that 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.
- Further, you'll find yourself paying more attention to the conversation with your programming partner than to surrounding noise; pairing really is a lot of fun. For the most part, you'll be collaborating with smart, like-minded people. Plus, if your wrists get sore from typing, you can hand off the keyboard to your partner and continue to be productive.

How to Pair

Many teams who pair frequently, but not exclusively, discover that they find more defects in solo code. 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. Never assign partners: pairs are fluid, forming naturally and shifting throughout the day. Over time, pair with everyone on the team. This will **improve team cohesion and spread design skills and knowledge throughout the team**.

When you need a fresh perspective, switch partners. I usually switch when I'm feeling frustrated or stuck. Have one person stay on the task and bring the new partner up to speed. Often, even explaining the problem to someone new will help you resolve it.

It's a good idea to **switch partners several times per day even** if you don't feel stuck. This will help keep everyone informed and moving quickly. I switch whenever I finish a task.
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.

Take small, frequent design steps—test-driven development works best—and talk about your assumptions, short-term goals, general direction, and any relevant history of the feature or project. If you're confused about something, ask questions. The discussion may enlighten your partner as much as it does you.

Expect to feel tired at the end of the day. Pairs typically feel that they have worked harder and accomplished more together than when working alone.

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 write? How does this code fit into the rest of the system? Is there duplication you need to remove? Can the code be more clear? Can the overall design be better?

PAIRING TIPS

- Pair on everything you'll need to maintain.
- Allow pairs to form fluidly rather than assigning partners.
- Switch partners when you need a fresh perspective.
- Avoid pairing with the same person for more than a day at a time.
- Sit comfortably, side by side.
- Produce code through conversation. Collaborate, don't critique.
- Switch driver and navigator roles frequently

Challenges

Pairing can be uncomfortable at first, as it may require you to collaborate more than you're used to.

1)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**. Clear debris off the desk and make sure there's room for your legs, feet, and knees.

Some people (like me) need a lot of **personal space**. Others like to get up close and personal. 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 flavours** such as coffee, garlic, onions, and spicy foods can lead to foul breath.

2) Mismatched skills: Some time 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. Ex: give chance to junior to learn inner workimg of ibrary.

3) Communication style:

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.

Try transforming declarations (such as “This method is too long”) into questions or suggestions (“Could we make this method shorter?” or “Should we extract this code block into a new method?”). Adopt an attitude of collaborative problem solving.

4) Tools and keybindings

Even if you don’t fall victim to the endless vi versus emacs editor war, you may find your coworkers’ tool preferences annoying. **Try to standardize on a particular toolset**. When you discuss coding standards, discuss issues if any.

Questions

Isn’t it wasteful to have two people do the work of one?

In pair programming, one person is programming and the other is thinking ahead, anticipating problems, and strategizing.

9. Explanation of Root-Cause Analysis: How to find and fix these, When Not to Fix the Root Cause

Root-Cause Analysis

When you hear about a serious mistake on my project, once natural reaction is to get angry or frustrated. You want to blame someone for screwing up.

Unfortunately, this response ignores the reality of Murphy's Law.

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, we have to try to remember Norm Kerth's Prime Directive: everybody is doing the best job they can given their abilities and knowledge. Rather than blaming people, I 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. (Build is based upon Source code)

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.

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

How to Fix the Root Cause

Root-cause analysis is a technique you can use for every problem you encounter, from the trivial to the significant.
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.
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.

Questions

Who should participate in root-cause analysis?
I usually conduct root-cause analysis in the privacy of my own thoughts, then share my conclusions and reasoning with others. Include whomever is necessary to fix the root cause.
When should we conduct root-cause analysis?
You can use root-cause analysis any time you notice a problem—when you find a bug, when you notice a mistake, as you're navigating, and in retrospectives. It need only take a few seconds. Keep your brain turned on and use root-cause analysis all the time.

10. Types of Retrospectives

Types of Retrospectives

1. The most common retrospective, *the iteration retrospective*, occurs at the end of every iteration.
2. Conduct longer, more **intensive retrospectives at crucial milestones.**
3. **Release retrospectives, project retrospectives, and surprise retrospectives**(conducted when an unexpected event changes your situation) give you a chance to reflect more deeply on your experiences and condense key lessons to share with the rest of the organization.

They work best when conducted by neutral third parties, so consider bringing in an experienced retrospective facilitator. Larger organizations may have such facilitators on staff (start by asking the HR department), or you can bring in an outside consultant.

11. How to Conduct an Iteration Retrospective?

How to Conduct an Iteration Retrospective

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.*

I timebox my retrospectives to exactly one hour. Your first few retrospectives will probably run long. Give it an extra half-hour, but *don't be shy about politely wrapping up and moving to the next step*. The whole team will get better with practice, and the next retrospective is only a week away.

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)

After you've acclimated to this format, change it. The retrospective is a great venue for trying new ideas.

Retrospectives are a powerful tool that can actually be damaging when conducted poorly. The process steps are as bellow

Step 1: The Prime Directive:

Everyone makes mistakes, even when lives are on the line. The retrospective is an opportunity to learn and improve. The team should never use the retrospective to place blame or attack individuals.

As facilitator, it's your job to nip (remove) destructive behaviour in the bud. To this end, start each retrospective by repeating Norm Kerth's Prime Directive. 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."

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(doubt) 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.

Step 3: 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:

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, when activity dies down, check the time and either wait for more ideas or move on.

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. Don't try to come up with a perfect name, and don't move cards between categories. Help the group move quickly through this step. The names aren't that important, and trying for perfection can easily drag this step out. Finally, after you have circled and named all the categories, vote on which categories should be improved during the next iteration.

I like to hand out little magnetic dots to represent votes; stickers also work well.

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. If not, don't spend too much time; flip a coin or something.

Discard the cards from the other categories. If someone wants to take a card to work on individually, that's fine, but not necessary.

Remember, you'll do another retrospective next week. Important issues will recur.

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. Half a dozen should suffice. 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.

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. Pick just one—it will help you focus. The retrospective objective is the goal that the whole team will work toward during the next iteration. Figure out how to keep track of the objective and who should work out the details.

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.
The thing I dislike about iteration retrospectives is that they often don't lead to specific changes.

12. Questions that could arise during Iteration Retrospective

What if management isn't committed to making things better? Although some ideas may require the assistance of others, if those people can't or won't help, refocus your ideas to what you can do. The retrospective is an opportunity for you to decide, as a team, how to improve your own process, not the processes of others. Your project manager may be able to help convey your needs to management and other groups.

Despite my best efforts as facilitator, our retrospectives always degenerate into blaming and arguing. What can I do? This is a tough situation, and there may not be anything you can do. If there are just one or two people who like to place blame, try talking to them alone beforehand. Describe what you see happening and your concern that it's disrupting the retrospective. Rather than adopting a parental attitude, ask for their help in solving the problem and be open to their concerns.

If a few people constantly argue with each other, talk to them together. Explain that you're concerned their arguing is making other people uncomfortable. Again, ask for their help. If the problem is widespread across the group, the same approach— talking about it—applies. This time, hold the discussion as part of the retrospective, or even in place of it. Share what you've observed, and ask the group for their observations and ideas about how to solve the problem If all else fails, you may need to stop holding retrospectives for a while. Consider bringing an organizational development (OD) expert to facilitate your next retrospective.

We come up with good retrospective objectives, but then nothing happens. What are we doing wrong? Your ideas may be too big. Remember, you only have one week, and you have to do your other work, too. Try making plans that are smaller scale—perhaps a few hours of work—and follow up every day. Finally, it's possible that the team doesn't feel like they truly have a voice in the retrospective. Take an honest look at the way you conduct it. Are you leading the team by the nose rather than facilitating? Consider having someone else facilitate the next retrospective.

Some people won't speak up in the retrospective. How can I encourage them to participate? It's possible they're just shy. It's not necessary for everyone to participate all the time. Waiting for a verbal response to the Prime Directive can help break the ice. On the other hand, they may have something they want to say but don't feel safe doing it. You can also try talking with them individually outside of the retrospective.

One group of people (such as testers) always gets outvoted in the retrospective. How can we meet their needs, too? Over time, every major issue will get its fair share of attention. One team in my experience had a few testers that felt their issue was being ignored. A month later, after the team had addressed another issue, the testers' concern was on the top of everyone's list. If time doesn't solve the problem—and be patient to start—you can use weighted dot voting, in which some people get more dot votes than others. If you can do this without recrimination, it may be a good way to level the playing field.

Another option is for one group to pick a different retrospective objective to focus on in addition to the general retrospective objective.

Our retrospectives always take too long. How can we go faster? It's OK to be decisive about wrapping things up and moving on. There's always next week. If the group is taking a long time brainstorming ideas or mute mapping, you might say something like, "OK, we're running out of time. Take two minutes to write down your final thoughts (or make final changes) and then we'll move on."

The retrospective takes so much time. Can we do it less often? It depends on how much your process needs improvement. An established team may not need as many iteration retrospectives as a new team. I would continue to conduct retrospectives at least every other week. If you feel that your

retrospective isn't accomplishing much, perhaps the real problem is that you need a change of pace. Try a different approach

13. COLLABORATING" MINI-ÉTUDE

- The purpose of this étude is to explore the flow of information in your project.
- Conduct this étude for a timeboxed half-hour every day for as long as it is useful. Expect to feel rushed by the deadline at first. If the étude becomes stale, discuss how you can change it to make it interesting again.
- You will need white, red, yellow, and green index cards; an empty table or magnetic whiteboard for your information flow map; and writing implements for everyone.
- Step 1: Start by forming pairs. Try for heterogeneous pairs—have a programmer work with a customer, a customer work with a tester, and so forth, rather than pairing by job description. Work with a new partner every day.
- Step 2: Within your pair, discuss the kinds of information that you need in order to do your job, or that other people need from you in order to do their job.

For information you needed, think of the calendar time needed from the moment you realized you needed the information to the moment you received it. For information you provided, think of the total effort that you and other team members spent preparing and providing the information.

think of the typical time required for this piece of information. If the typical time required is less than 10 minutes, take a green index card. If it's less than a day, take a yellow index card. If it's a day or longer, take a red index card. Write down the type of information involved, the group that you get it from (or give it to), the role you play, and the time required, as shown in Figure 6-1.

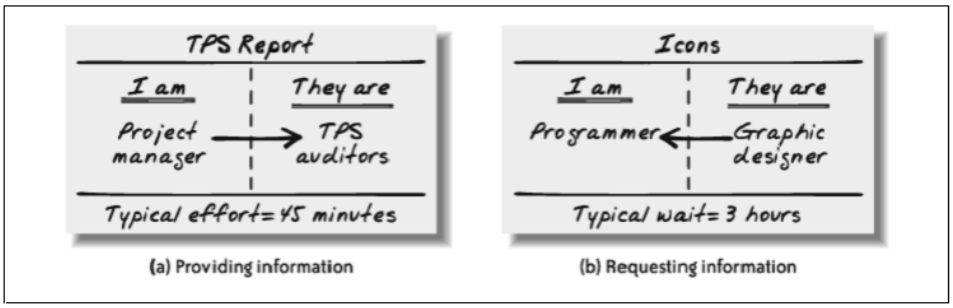

Figure 6-1. Sample cards

Step 3: Within your pair, discuss things that your team can do to reduce or eliminate the time required to get or provide this information. Pick one and write it on a white card.

Step 4 : As a team, discuss all the cards generated by all the pairs. Consider these questions:
- Where are the bottlenecks in receiving information?
- Which latencies are most painful in your current process?
- Which flow is most important to optimize in your next iteration?
- What is the root cause of those latencies?

14. Trust and some strategies for generating trust in your XP team

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”

The team must take joint responsibility for their work. Team members need to think of the rest of the team as “us,” not “them.”

You need to trust that you’ll be treated with respect when you ask for help or disagree with someone. It takes trust to believe that the team will deliver a success. Trust doesn’t magically appear—you have to work at it.

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

Team Strategy **#1: Customer-Programmer Empathy**

“us versus them” attitude between customers and programmers. 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, remember that customers have corporate masters that demand results. Bonuses, career advancement, and even job security depend on successful delivery, and the demands aren’t always reasonable. Customers must deliver results anyway. Customers, remember that ignoring or overriding programmers’ professional recommendations about timelines often leads to serious personal consequences for programmers.

Each group gets to see that the others are working just as hard.

Team Strategy #2: Programmer-Tester Empathy : "us versus them" attitudes between programmers and testers, although it isn't quite as prevalent as customer-programmer discord. When it occurs, programmers tend not to show respect for the testers' abilities, and testers see their mission as shooting down the programmers' work.

Empathy and respect are the keys to better relations.

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.

Testers, focus on the team's joint goal: releasing a great product. When you find a mistake, it's not an occasion for celebration or gloating. 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

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.

You can avoid this waste by keeping productive teams together. Most organizations think of people as the basic "resource" in the company.

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*.

15. Organizational Strategy

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.

#2: Deliver on Commitments: 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.

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.

#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.

When you've identified a problem, let the stakeholders know about it. They'll appreciate your professionalism even if they don't like the problem. Addressing a problem successfully can build trust like nothing else.

#4: Respect Customer Goals

If the customers are unhappy, those feelings transmit directly back to stakeholders.

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.

If customers want something that may take a long time or involves tremendous technical risks, suggest alternate approaches to reach the same underlying goal for less cost.

#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.

Being open about what you're doing will also help people appreciate your team. Other people in the company are likely to be curious, and a little wary, about your strange new approach to software development. That curiosity can easily turn to resentment if the team seems insular or stuck-up. You can be open in many ways. Consider holding brown-bag lunch sessions describing the process, public code-fests in which you demonstrate your code and XP technical practices, or an "XP open-house day" in which you invite people to see what you're doing and even participate for a little while. If you like flair, you can even wear buttons or hats around the office that say "Ask me about XP."

#6: Be Honest

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

These are minor frauds, yes. You may even think that “fraud” is too strong a word—but all of these behaviors give stakeholders the impression that you’ve done more than you actually have.

There’s a practical reason not to do these things: stakeholders will expect you to complete the remaining features just as quickly, when in fact you haven’t even finished the first set. You’ll build up a backlog of work that looks done but isn’t. At some point, you’ll have to finish that backlog, and the resulting schedule slip will produce confusion, disappointment, and even anger.

Results

When you have a team that works well together, you cooperate to meet your goals and solve your problems. You collectively decide priorities and collaboratively allocate tasks.