Why Agile?

- Will agile development help us be more successful?
- When you can answer that question, you'll know whether agile development is right for you.

Benefits

- the ability to release software more frequently.

Understanding Success

- The traditional idea of success is **delivery on time, on budget, and according to specification**.

Some classic definitions:

- Successful : “Completed on time, on budget, with all features and functions as originally specified.”
- Challenged: “Completed and operational but over budget, over the time estimate, [with] fewer features and functions than originally specified.”
- Impaired: “Cancelled at some point during the development cycle.”

- Projects that were found to meet all of the traditional criteria for success—time, budget and specifications—may still be failures in the end because they **fail to appeal to the intended users or because they ultimately fail to add much value to the business**.

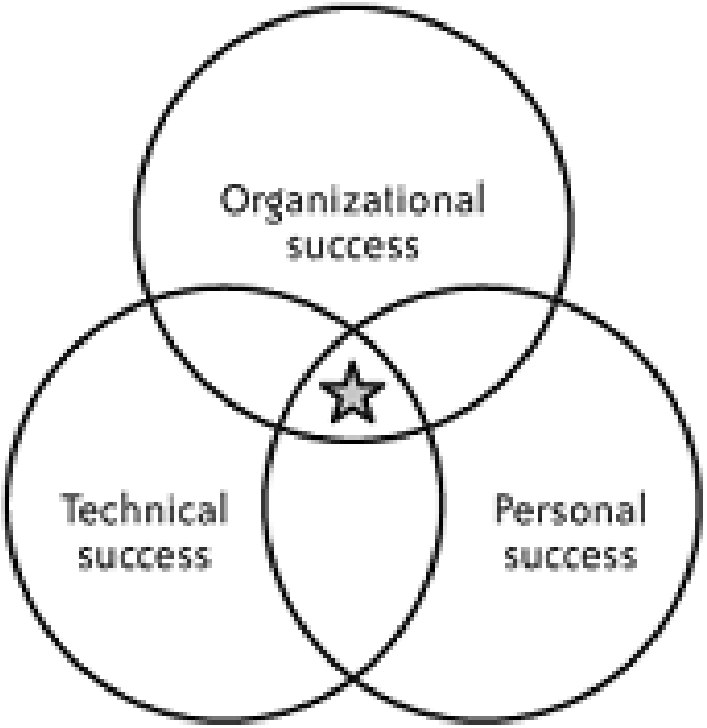
Similarly, projects considered failures according to traditional IT metrics may wind up being successes because despite cost, time or specification problems, **the system is loved by its target audience or provides unexpected value.**

For understanding

(Is anything)

Beyond Deadlines

- There has to be more to success than meeting deadlines... but what?
- When I was a kid, I was happy just to play around. I loved the challenge of programming.
- When I got a program to work, it was a major victory. Back then, even a program that didn't work was a success of some sort, as long as I had fun writing it. My definition of success centered on personal rewards



- 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

The Importance of Organizational Success

- Organizational success is often neglected by software teams in favour of the more easily achieved technical and personal successes. Rest assured, however, that even if you're not taking responsibility for organizational success, the broader organization is judging your team at this level.

- Senior management and executives aren't likely to care if your software is elegant, maintainable, or even beloved by its users; they care about results. That's their return on investment in your project.
- If you don't achieve this sort of success, they'll take steps to ensure that you do. (Complete Working Product)

- When managers are unhappy with your team’s results, the swords come out. Costs are the most obvious target. There are two easy ways to cut them: set aggressive deadlines to reduce development time, or ship the work to a country with a lower cost of labour. Or both.
- These are clumsy techniques. Aggressive deadlines end up increasing schedules rather than reducing them.

WHAT DO ORGANIZATIONS VALUE?

- Although some projects' value comes directly from sales, there's more to organizational value than revenue. Projects provide value in many ways, and you can't always measure that value in dollars and cents.
- Aside from revenue and cost savings, sources of value include:

- Competitive differentiation
- Brand projection
- Enhanced customer loyalty
- Satisfying regulatory requirements
- Original research
- Strategic information

Enter Agility

- Will agile development help you be more successful?

It might.

Agile development focuses on achieving personal, technical, and organizational successes.

If you're having trouble with any of these areas, agile development might help.

Organizational Success

- Agile methods achieve organizational successes by focusing on ***delivering value and decreasing costs*.** This directly translates to increased return on investment.
- Agile methods also set expectations early in the project, so if your project won't be an organizational success, you'll ***find out early enough to cancel it*** before your organization has spent much money.
- Specifically, agile teams increase value by **including business experts** and by *focusing development* ***efforts on the core value*** that the project provides for the organization.
- Agile projects release their ***most valuable features first*** *and release* ***new versions frequently***, which dramatically increases value.

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.

How to Be Agile

- Agile development is a philosophy. It's a way of thinking about software development.
- The canonical description of this way of thinking is the Agile Manifesto, a collection of 4 values (Figure 2-1) and 12 principles (Figure 2-2).

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

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.

Figure 2-2. Agile principles

Understanding XP

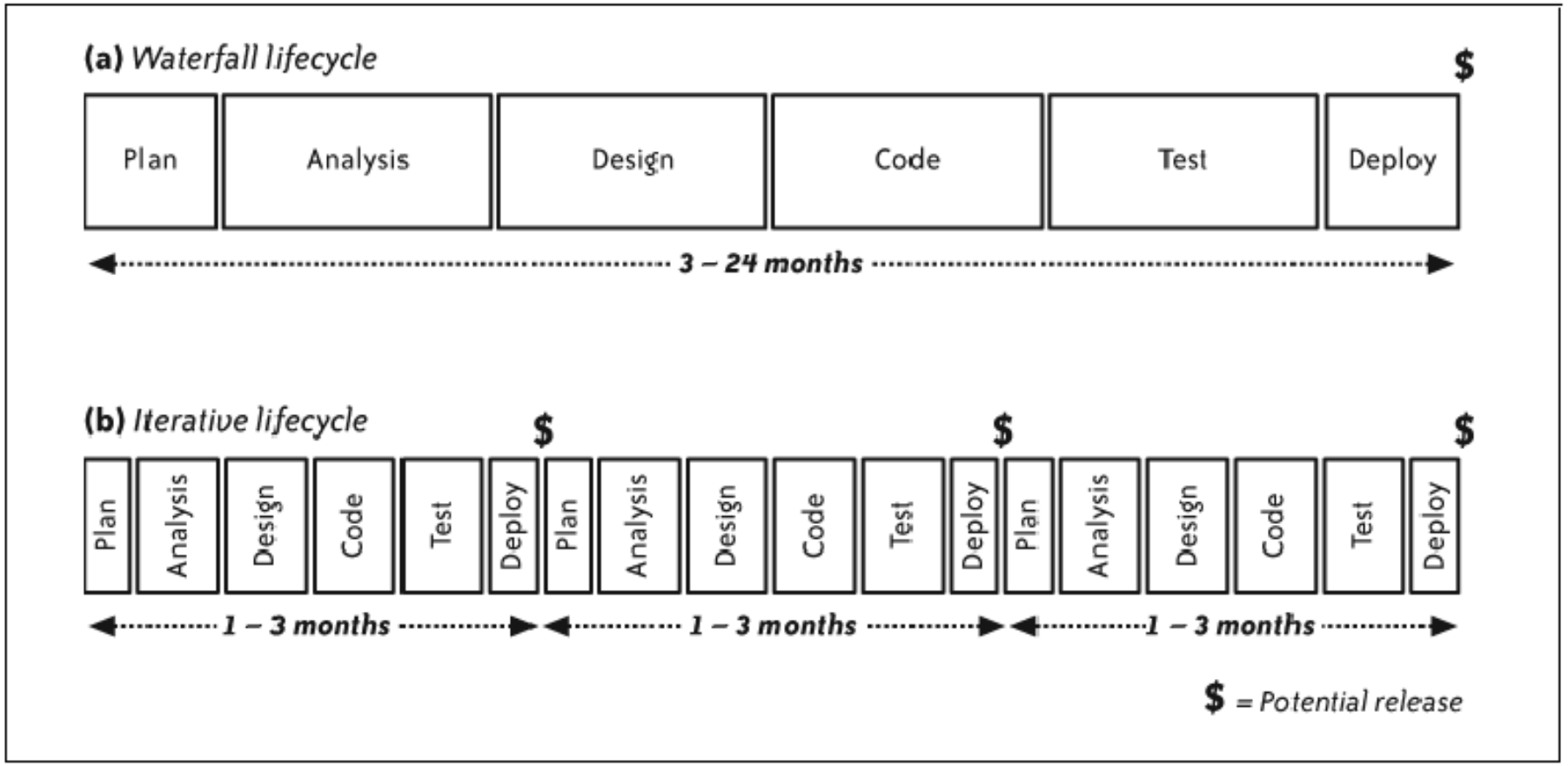


Figure 3-1. Traditional lifecycles

The XP Lifecycle

One of the most astonishing premises of XP is that you can eliminate requirements, design, and testing phases as well as the formal documents that go with them.

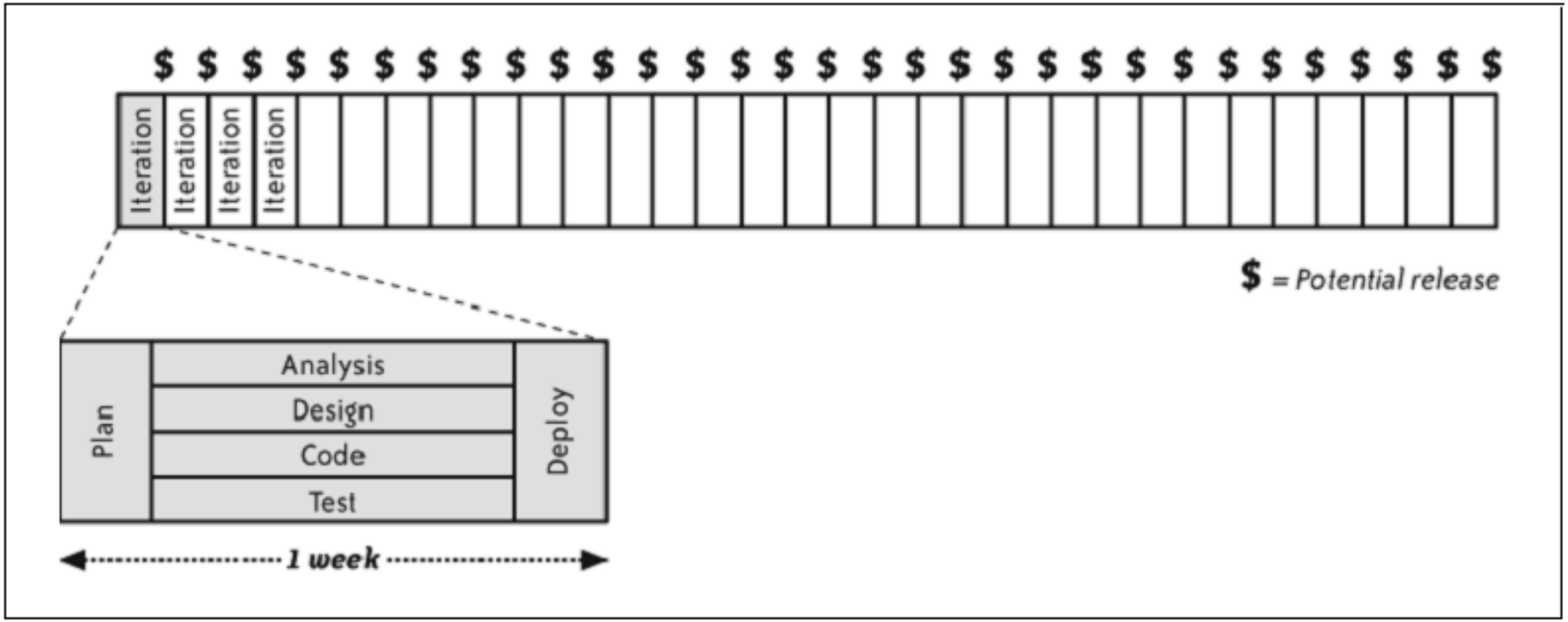


Figure 3-2. XP lifecycle

- That’s true. Software projects do need more requirements, design, and testing—which is why XP teams work on these activities every day. **Yes, every day.**
- XP emphasizes face-to-face collaboration. This is so effective in eliminating communication delays and misunderstandings.
- This allows them to work on all activities every day—with simultaneous phases—as shown in Figure 3-2.
- Using simultaneous phases, an XP team produces **deployable software every week**. In each iteration, the team analyses, designs, codes, tests, and deploys a subset of features.
- The team gets feedback much more frequently. As a result, the team can easily connect successes and failures to their underlying causes.

- The amount of unproven work is very small, which allows the team to correct some mistakes on the fly, as when *coding reveals a design flaw*, or when a *customer review reveals* that a user *interface layout is confusing* or ugly.
- The *tight feedback loop* also allows XP teams to *refine their plans* quickly. It's much easier for a customer to refine a feature idea if she can request it and start to explore a working prototype within a few days. The same principle applies for tests, design, and team policy.
- If you find a design defect during coding or testing, you can use that knowledge as you continue to analyze requirements and design the system in subsequent iterations.

How It Works:

- XP teams perform nearly every software development activity simultaneously. Analysis, design, coding, testing, and even deployment occur with rapid frequency.
- XP does it by working in iterations: week-long increments of work. Every week, the team does a bit of release planning, a bit of design, a bit of coding, a bit of testing, and so forth.
- They work on stories: very small features, or parts of features, that have customer value. Every week, the team commits to delivering four to ten stories. Throughout the week, they work on all phases of development for each story. At the end of the week, they deploy their software for internal review. (In some cases, they deploy it to actual customers.)
- The following sections show how traditional phase-based activities correspond to an XP iteration.

Planning:

Every XP team includes several business experts—the on-site customers—who are responsible for making business decisions.

The on-site customers **point the project in the right direction by clarifying the project vision, creating stories, constructing a release plan, and managing risks.**

Programmers provide estimates and suggestions, which are blended with customer priorities in a **process called the planning game**. Together, the team strives to create small, frequent releases that maximize value.

The planning effort is most **intense during the first few weeks** of the project. During the remainder of the project, **customers continue to review and improve the vision and the release plan** to account for new opportunities and unexpected events.

The **team creates a detailed plan for the upcoming week** at the beginning of each iteration. The team touches base **every day** in a brief stand-up meeting, and its informative workspace keeps **everyone informed about the project status**

Analysis:

Rather than using an upfront analysis phase to define requirements, on-site **customers sit with the team full-time**. On-site customers may or may not be real customers depending on the type of project, but **they are the people best qualified to determine what the software should do**.

On-site customers are responsible for figuring out the requirements for the software. To do so, they use their own knowledge as customers combined with traditional requirements-gathering techniques. When programmers need information, they simply ask.

Customers are responsible for organizing their work so they are ready when programmers ask for information. They figure out the general requirements for a story before the programmers estimate it and the detailed requirements before the programmers implement it.

.

Some requirements are tricky or difficult to understand. Customers formalize these requirements, with the assistance of testers, by creating customer tests: Customers and testers create the customer tests for a story around the same time that programmers implement the story. To assist in communication, programmers use a ubiquitous language in their design and code

For the UI, customers work with the team to create sketches of the application screens. In some cases, customers work alongside programmers as they use a UI builder to create a screen. Some teams include an interaction designer who's responsible for the application's UI.

Design and coding

XP **uses incremental design and architecture** to continuously create and improve the design in small steps. This work is driven by test-driven development(TDD), an activity that inextricably weaves together testing, coding, design, and architecture.

To support this process, **programmers work in pairs**, which **increases** the amount of **brainpower** brought to bear on each task and ensures that one person in each pair always has time to think about larger design issues.

Programmers are also responsible for managing their development environment. They **use a version control system** for configuration management and maintain their own automated build. Programmers integrate their code every few hours and ensure that every integration is technically capable of deployment.

To support this effort, **programmers also maintain coding standards and share ownership of the code. The team shares a joint aesthetic for the code, and everyone is expected to fix problems in the code regardless of who wrote it.**

Testing

Each member of the team—programmers, customers, and testers—makes his own contribution to software quality. **Well-functioning XP teams produce only a handful of bugs per month** in completed work.

Programmers provide the first line of defence with test-driven development.

TDD produces automated unit and integration tests. In some cases, programmers may also create end-to-end tests. These tests help ensure that the software does what the programmers intended.

Likewise, customer tests help ensure that the programmers' intent matches customers' expectations.

Customers review work in progress to ensure that the UI works the way they expect. They also produce examples for programmers to automate that provide examples of tricky business rules.

Finally, testers help the team understand whether their efforts are in fact **producing high quality code**. When the testers find a bug, the team conducts root-cause analysis and considers how to improve their process to prevent similar bugs from occurring in the future.

Testers also **explore** the software's **non-functional characteristics**, such as performance and stability. Customers then use this information to decide whether to create additional stories. When bugs are found, programmers create automated tests to show that the bugs have been resolved. This suite is sufficient to prevent regressions. Every time programmers integrate (once every few hours), they run the entire suite of regression tests to check if anything has broken.

The team also supports their **quality efforts through pair programming, energized work, and iteration slack**. These practices enhance the brainpower that each team member has available for creating high-quality software.

Deployment

XP teams keep their software ready to deploy at the end of any iteration. They **deploy the software** to **internal stakeholders** every week in preparation for the weekly iteration demo.

Deployment to real customers is scheduled according to business needs.

As long as the team is active, **it maintains the software** it has released. In other cases, a separate support team may take over. Similarly, when the project ends, the team may hand off maintenance duties to another team.

In this case, the team creates documentation and provides training as necessary during its last few weeks.

XP PRACTICES BY PHASE

- XP uses iterations rather than phases; teams perform every one of these activities each week. Most are performed every day

Table 3-1. XP Practices by phase

XP Practices	Planning	Analysis	Design & Coding	Testing	Deploym
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	•	•	•	•	•
Releasing					
"Done Done"			•		•
No Bugs			•	•	
Version Control			•		
Ten-Minute Build			•		•

XP Practices	Planning	Analysis	Design & Coding	Testing	Deployment
Continuous Integration			•		•
Collective Code Ownership			•		
Documentation					•
Planning					
Vision	•	•			
Release Planning	•	•			
The Planning Game	•	•			
Risk Management	•				
Iteration Planning	•		•		
Slack	•		•		
Stories	•	•			
Estimating	•				
Developing					
Incremental Requirements		•	•		
Customer Tests		•		•	
Test-Driven Development			•	•	
Refactoring			•		
Simple Design			•		
Incremental Design and Architecture			•		
Spike Solutions			•		
Performance Optimization			•		
Exploratory Testing				•	

Team software development is different. The same information is spread out among many members of the team. Different people know:

- How to design and program the software (programmers, designers, and architects)
- Why the software is important (product manager)
- The rules the software should follow (domain experts)
- How the software should behave (interaction designers)
- How the user interface should look (graphic designers)
- Where defects are likely to hide (testers)
- How to interact with the rest of the company (project manager)
- Where to improve work habits (coach)
- All of this knowledge is necessary for success.

The Whole Team

- XP teams sit together in an open workspace. At the beginning of each iteration, the team meets for a series of activities: an iteration demo, a retrospective, and iteration planning.
- These **typically take two to four hours in total**. The **team also meets for daily stand-up meetings**, which usually **take five to ten minutes** each.
- Other than these scheduled activities**, everyone on the team plans his own work. (**That doesn't mean everybody works independently; they just aren't on an explicit schedule.) Team members work out the details of each meeting when they need to.
- Sometimes it's as informal as somebody standing up and announcing across the shared workspace that he would like to discuss an issue. This self-organization is a hallmark of agile teams.

On-Site Customers

On-site customers—often just called customers—are **responsible for defining the software the team builds**. The rest of the team can and should contribute suggestions and ideas, but the **customers are ultimately responsible for determining what stakeholders find valuable.**

Customers' **most important activity is release planning**. Customers need to evangelize the project's vision; identify features and stories; determine how to group features into small, frequent releases; manage risks; and create an achievable plan by coordinating with programmers and playing the planning game.(set the plan)

On-site customers may or may not be real customers, depending on the type of project. Regardless, customers are responsible for refining their plans by soliciting (ask or collect) feedback from real customers and other stakeholders. One of the venues for this feedback is the weekly iteration demo, which customers lead.

On-Site Customers

- In addition to planning, customers are responsible for providing programmers with requirements details upon request.
- XP uses requirements documents only as memory aids for customers. Customers themselves act as living requirements documents, researching information in time for programmer use and providing it as needed. Customers also help communicate requirements by **creating mock-ups, reviewing work in progress, and creating detailed customer tests that clarify complex business rules**. The entire team must sit together for this communication to take place effectively.
- Typically, product managers, domain experts, interaction designers, and business analysts play the role of the on-site customer. One of the most difficult aspects of creating a cross-functional team is finding people qualified and willing to be on-site customers.

The product manager (aka product owner)

- The product manager has only one job on an XP project, but it's a doozy(outstanding). That job is to maintain and promote the product vision, this means documenting the vision, sharing it with stakeholders, incorporating feedback, generating features and stories, setting priorities for release planning, providing direction for the team's on-site customers, reviewing work in progress, leading iteration demos, involving real customers, and dealing with organizational politics.
- In addition to vision, she must have the authority to make difficult trade-off decisions about what goes into the product and what stays out. She must have the political savvy(ability to make good judgement) to align diverse stakeholder interests, consolidate them into the product vision, and effectively say "no" to wishes that can't be accommodated.
- Make sure your product manager is committed to the project full-time. Once a team is running smoothly, the product manager might start cutting back on his participation. He should still participate in every retrospective, every iteration demo, and most release planning sessions. Her role may be nothing more than consolidating the ideas of the committee into a single vision.

Domain experts (aka subject matter experts)

- Most software operates in a particular industry, such as finance, that has its own specialized rules for doing business. To succeed in that industry, the software must implement those rules faithfully and exactly. These rules are domain rules, and knowledge of these rules is domain knowledge.
- Most programmers have gaps in their domain knowledge, even if they've worked in an industry for years. In many cases, the industry itself doesn't clearly define all its rules. The basics may be clear, but there are **nitpicky details** where domain rules are implicit or even contradictory.
- The team's domain experts are **responsible for figuring out these details and having the answers at their fingertips**. Domain experts, **also known as subject matter experts, are experts in their field**. Examples include financial analysts and PhD chemists.
- Domain experts spend most of their time with the team, figuring out the details of upcoming stories and standing ready to answer questions when programmers ask. For complex rules, they create customer tests (often with the help of testers) NOTE: On small teams, product managers often double as domain experts.

Interaction designers

- The user interface is the public face of the product. For many users, the UI is the product. They judge the product's quality solely on their perception of the UI.
- Interaction designers help define the product UI. Their job focuses on understanding users, their needs, and how they will interact with the product. They perform such tasks as interviewing users, creating user personas, reviewing paper prototypes with users, and observing usage of actual software.
- You may not have a professional interaction designer on staff. Some companies fill this role with a graphic designer, the product manager, or a programmer.
- Interaction designers divide their time between working with the team and working with users.

- They contribute to release planning by advising the team on user needs and priorities. During each iteration, they help the team create mock-ups of UI elements for that iteration's stories.

- As each story approaches completion**, they review the look and feel of the UI and confirm that it works as expected**.
- Rather than spending time researching users and defining behaviours before development begins, **interaction designers must iteratively refine their models concurrently with iterative refinement of the program itself.**
- XP produces working software every week, which provides a rich grist for the interaction designer's mill. Designers have the opportunity to take real software to users, observe their usage patterns, and use that feedback to effect changes as soon as one week later

Business analysts

- On non-agile teams, business analysts typically act as liaisons(communicator) between the customers and developers, by clarifying and refining customer needs into a functional requirements specification.
- On an XP team, business analysts augment(increase) a team that already contains a product manager and domain experts. The **analyst continues to clarify and refine customer needs**. Analysts **help customers think of details they might otherwise forget** and help programmers express technical trade-offs in business terms.

Programmers:

The bulk of the XP team consists **of software developers** in a variety of specialties. Each of these **developers contributes directly to creating working code**. So to emphasize this, XP calls all developers programmers.

If the **customers' job is to maximize the value of the product, then the programmers' job is to minimize its cost**. Programmers are responsible for finding the most effective way of delivering the stories in the plan

- **Programmers spend** most of their **time pair programming**. Using test-driven development, they write tests, implement code, refactor, and incrementally design and architect the application.
- They **pay careful attention to design quality**, and they're keenly aware of technical depth and its impact on development time and future maintenance costs.
- **Programmers also ensure** that the customers may choose to **release** the software **at the end of any iteration**. With the help of the whole team, the **programmers strive to produce no bugs** in completed software. They maintain a ten-minute build that can build a complete release package at any time. They **use version control** and practice continuous integration, keeping all but the last few hours' work integrated and passing its tests.
- This work is a joint effort of all the programmers. At the beginning of the project, the programmers establish coding standards that allow them to collectively share responsibility for the code. **Programmers** have the right and the responsibility to **fix any problem** they see, no matter which part of the application it touches.
- **Programmers rely on customers for information** about the software to be built. Rather than guessing when they have a question, they ask one of the on-site customers.
- Finally, **programmers help ensure the long-term maintainability** of the product by providing documentation at appropriate times.

- **Designers and architects**

Expert designers and architects are still necessary. They contribute by **guiding the team's incremental design** and architecture efforts and by helping team members see ways of **simplifying complex designs**. They **act as peers—that is, as programmers**—rather than teachers, guiding rather than dictating.

- **Technical specialists**:

The programmers could include a **database designer, a security expert, or a network architect.** XP programmers are generalizing specialists. Although each person has his own area of expertise, everybody is expected to work on any part of the system that needs attention.

Testers:

- Testers apply their critical thinking skills to help customers consider all possibilities when envisioning the product. They **help customers identify holes in the requirements and assist in customer testing**. Testers also act as technical investigators for the team. They use exploratory testing to help the team identify whether it is successfully preventing bugs from reaching finished code. Testers **also provide information about the software's non-functional characteristics, such as performance, scalability, and stability**, by using both exploratory testing and long-running automated tests.
- However, testers don't exhaustively test the software for bugs. Rather than relying on testers to find bugs for programmers to fix, **the team should produce nearly bug-free code on their own.**
- **When testers find bugs, they help the rest of the team figure out what went wrong** so that the team as a whole can prevent those kinds of bugs from occurring in the future.
- These responsibilities require creative thinking, flexibility, and experience defining test plans. Because **XP automates repetitive testing rather than performing manual regression testing**, testers who are used to self-directed work are the best fit.
- Some XP teams don't include dedicated testers. If you don't have testers on your team, programmers and customers should share this role.

Coaches

- XP teams self-organize, which means each member of the team figures out how he can best help the team move forward at any given moment.
- XP teams eschew(deliberately avoid) traditional management roles. Instead, XP leaders lead by example, helping the team reach its potential rather than creating jobs and assigning tasks. To emphasize this difference, **XP leaders are called coaches**. **Over time, as the team gains experience and self-organizes, explicit leadership becomes less necessary and leadership roles dynamically switch from person to person as situations dictate**.
- **A coach's work is subtle(difficult to analyse or describe); it enables the team to succeed. Coaches help the team start their process by arranging for a shared workspace and making sure that the team includes the right people. They help set up conditions for energized work, and they assist the team in creating an informative workspace.**
- One of the most important things the **coaches can do is to help the team interact with the rest of the organization**. They **help the team generate organizational trust and goodwill, and they often take responsibility for any reporting needed.**
- **Coaches also help the team members maintain their self-discipline, helping them remain in control of challenging practices such as risk management, test-driven development, slack, and incremental design and architecture.**

The programmer-coach

- Every team needs a programmer-coach to help the other programmers with XP's technical practices. **Programmer-coaches are often senior developers and may have titles such as "technical lead" or "architect."** They can even be functional managers. While some programmer-coaches make good all-around coaches, others require the assistance of a project manager.
- Programmer-coaches also **act as normal programmers and participate fully in software development.**

The project manager

- Project managers **help the team work with the rest of the organization**. They are usually **good at coaching non-programming practices**. Some functional managers fit into this role as well.
- However, most project managers lack the technical expertise to coach XP's programming practices, which necessitates the assistance of a programmer-coach.
- Project managers may also **double as customers**.
- NOTE: Include a programmer-coach and consider including a project manager

Other Team Members

An XP team should include exactly the expertise necessary to complete the project successfully and cost-effectively.

The Project Community

- Projects don't live in a vaccuum; every team has an ecosystem surrounding it. This ecosystem extends beyond the team to the project community, which includes everyone who affects or is affected by the project.
- Keep this community in mind as you begin your XP project, as everybody within it can have an impact on your success
- Two members of your project community that you may forget to consider are your organization's Human Resources and Facilities departments. Human Resources often handles performance reviews and compensation. Their mechanisms may not be compatible with XP's team-based effort. Similarly, in order to use XP, you'll need the help of Facilities to create an open workspace

Stakeholders

- Stakeholders form a large subset of your project community. Not only are they affected by your project, they have an active interest in its success.
- **Stakeholders may include end users, purchasers, managers, and executives**. Although they don't participate in day-to-day development, do invite them to attend each iteration demo.
- The on-site customers particularly the **product manager—are responsible for understanding the needs of your stakeholders**, deciding which needs are most important, and knowing how to best meet those needs.

The executive sponsor

- The executive sponsor is particularly important: he holds the purse strings for your project.
- Take extra care to identify your executive sponsor and understand what he wants from your project. He's your ultimate customer. Be sure to provide him with regular demos and confirm that the project is proceeding according to his expectations.

XP PRACTICES BY ROLE

- The following table shows the practices you should learn to practice XP. You can always learn more,
- of course! In particular, if you're in a leadership role (or would like to be), you should study all the practices.
- Table 3-6. XP Practices by role

XP Practices	On-Site Customers	Programmers	Testers	Coaches
Thinking				
Pair Programming		•		
Energized Work	•	•	•	•
Informative Workspace	•	•	•	•
Root-Cause Analysis	•	•	•	•
Retrospectives	•	•	•	•
Collaborating				
Trust	•	•	•	•

• = You will be involved with this practice. Studying it will be helpful, but not necessary.

• = You should study this practice carefully.

XP Practices	On-Site Customers	Programmers	Testers	Coaches
Sit Together	-	-	-	•
Real Customer Involvement	-			•
Ubiquitous Language		•		
Stand-Up Meetings	-	-	-	•
Coding Standards		-		•
Iteration Demo	•	•	-	-
Reporting	-	-	-	•
Releasing				
"Done Done"	•	•	•	•
No Bugs	-	•	•	•
Version Control	-	•	-	
Ten-Minute Build		•		
Continuous Integration		•		
Collective Code Ownership		•		
Documentation	-	-		•
Planning				
Vision	•			•
Release Planning	•			•
The Planning Game	-	-		•
Risk Management	-	-	-	•
Iteration Planning	-	•		•
Slack		-		•
Stories	-	-	-	•
Estimating		•		
Developing				
Incremental Requirements	•		•	•
Customer Tests	•	•	•	
Test-Driven Development		•	-	
Refactoring		•		
Simple Design		•		
Incremental Design and Architecture		•		
Spike Solutions		•		
Performance Optimization	-	•		

Filling Roles

The **makeup of your team will probably depend more on your organization's traditions** than on anything else.

In other words**, if project managers and testers are typical for your organization, include them. If they're not, you don't necessarily need to hire them**. You don't have to have one person for each role— **some people can fill multiple roles**.

Just keep in mind that someone has to perform those duties even if no one has a specific job title saying so.

At a minimum, however, one person clearly designated as "product manager" (who may do other customer-y things) and one person clearly defined as "programmer-coach" (who also does programmer-y things).

Product managers are usually domain experts and can often fill the project manager's shoes, too.

One of the customers may be able to play the role of interaction designer, possibly with the help of a UI programmer.

On the programming side, many programmers are generalists and understand a variety of technologies. In the absence of testers, both programmers and customers should pick up the slack.

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.

XP Concepts

Refactoring

There are multiple ways of expressing the same concept in source code. Some are better than others. Refactoring is the process of changing the structure of code—rephrasing it—without changing its meaning or behaviour. It's used to improve code quality, to fight off software's unavoidable entropy, and to ease adding new features

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.**

The Last Responsible Moment

XP teams delay commitment until the last responsible moment.

By delaying decisions until this crucial point, you increase the accuracy of your decisions, decrease your workload, and decrease the impact of changes. Why?

A delay gives you time to increase the amount of information you have when you make a decision, which increases the likelihood it is a correct decision.

That, in turn, decreases your workload by reducing the amount of rework that results from incorrect decisions.

Changes are easier because they are less likely to invalidate decisions or incur additional rework.

Stories

Stories represent self-contained, individual elements of the project. They tend to correspond to individual features and typically represent one or two days of work.

Stories are customer-centric, describing the results in terms of business results.

They're not implementation details, nor are they full requirements specifications. They are traditionally just an index card's worth of information used for scheduling purposes.

Iterations

An iteration is the full cycle of design-code-verify-release practiced by XP teams.

It’s a timebox that is usually one to three weeks long.

Each iteration begins with the customer selecting which stories the team will implement during the iteration, and it ends with the team producing software that the customer can install and use.

The beginning of each iteration represents a point at which the customer can change the direction of the project.

Smaller iterations allow more frequent adjustment.

Fixed-size iterations provide a well-timed rhythm of development.

Though it may seem that small and frequent iterations contain a lot of planning overhead, the amount of planning tends to be proportional to the length of the iteration

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.

Theory of Constraints

Every system has a single constraint that determines the overall throughput of the system. Regardless of how much work testers and customers do, many software teams can only complete their projects as quickly as the programmers can program them.

If the rest of the team outpaces(ask them to work more faster) the programmers, the work piles up, falls out of date and needs reworking, and slows the programmers further.

Therefore, the programmers set the pace, and their estimates are used for planning.

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

Adopting XP

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.

Adopting XP

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.

Adopting XP

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.

If your team isn't colocated(sharing facility or location)...

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

Adopting XP

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.

Adopting XP

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.

Adopting XP

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.

Adopting XP

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.

Applying XP to a Brand-New Project (Recommended)

When starting a brand-new XP project, expect the first three or four weeks to be pretty chaotic as everyone gets up to speed.

During the first month, on-site customers will be working out the release plan, programmers will be establishing their technical infrastructure, and everyone will be learning how to work together.

Plan your first iteration: Normally, this involves selecting stories from the release plan, but you won't have a release plan yet. Instead, think of one feature that will definitely be part of your first release.

These basic stories will give you ideas for more stories that will add missing details. Brainstorm 10 to 20 in the first planning session and have the programmers estimate them. These should keep the programmers busy for several iterations. Try to choose stories that the programmers already understand well; this will reduce the amount of time customers need to spend answering programmer questions so they can focus on creating the release plan.

Iteration planning is a little more difficult during the first iteration because you haven't established a velocity yet. During the iteration, work on just one or two stories at a time and check your progress every day.

After you've finished planning, programmers should start establishing their technical infrastructure. After the first few days, the fundamentals should be well-established and the project should be large enough for people to work on separate parts without unduly interfering with each other. At this point, you can break into pairs and work

While the programmers are working on stories, customers and testers should work on the vision and release plan. First, work with stakeholders to create the product vision.

Finalizing the vision can take a few weeks, so while that's in progress, brainstorm the stories for your first feature. Start thinking about other features you want to include, and pick a date for your first release.

The feeling of chaos will subside as the team works in a steady, predictable rhythm.

Applying XP to an Existing Project:

If you have a legacy project—you can achieve the same results, but it will take more time. In this case, adopt XP incrementally.

The big decision

Other than change itself, the biggest challenge in applying XP to an existing project is not writing tests, refactoring, or cleaning up your bug database. The biggest challenge is setting aside enough time to **pay down technical debt.**

To improve productivity and reduce bug production, not only do you need to stop incurring new technical debt, you need to set aside extra slack(both time and money spent).

Start by introducing XP's structural practices. Move the team, including customers and testers, into a shared workspace, start pair-programming, conduct iteration planning and retrospectives, and so forth. Apply:

All the “Thinking” practices

- All the “Collaborating” practices
- All the “Planning” practices
- Version control, collective code ownership, and “done done” and customer reviews

Unit 2: Thinking

- 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.
- Do you want somebody to watch over your shoulder all day? Do you want to waste half your time sitting in sullen silence watching somebody else code?
- Of course not. Nobody does—especially not people who pair program.
- 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. Most programmers I know who tried pairing for a month find that they prefer it to programming alone.

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

Pairing Stations:

Need plenty of room for both people to sit side by side.

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. I like to plug in two keyboards and mice so each person can have a set.

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.

How can I convince my team or organization to try pair programming?

Ask permission to try it as an experiment. Set aside a month in which everyone pairs on all production code. Be sure to keep going for the entire month, as pair programming may be difficult and uncomfortable for the first few weeks.

Don’t just ask permission of management; be sure your fellow team members are interested in trying pairing as well. The only programmers I know who tried pairing for a month and didn’t like it are the ones who were forced to do it against their will.

Do we really have to pair program all the time?

This is a decision that your whole team should make together. Before you decide, try pairing on all production code (everything you need to maintain) for a month. You may enjoy it more than you expect.

Regardless of your rule, you will still produce code that you don’t need to maintain.

Some production tasks are so repetitive that they don’t require the extra brainpower a pair provides. Use the navigator’s extra time to think about design improvements and consider discussing it with your whole team.

How can I concentrate with someone talking to me?

When you navigate, you shouldn’t have too much trouble staying several steps ahead of your driver. If you do have trouble, ask your driver to think out loud so you can understand her thought process. As driver, you may sometimes find that you’re having trouble concentrating. Let your navigator know—she may have a suggestion that will help you get through the roadblock. At other times, you may just need a few moments of silence to think through the problem.

If you have trouble concentrating, try taking smaller steps.

Rely on your navigator to keep track of what you still need to do (tell him if you have an idea; he'll write it down) and focus only on the few lines of code needed to make the next test pass.

What if we have an odd number of programmers?

A programmer flying solo can do productive tasks that don't involve production code. She **can research new technologies, or learn more about a technology the team is using**. She can pair with a customer or tester to **review recent changes, polish the application, or do exploratory testing**. She can be the team's batman

Alternatively, a solo programmer may wish to spend some time **reviewing the overall design** —either to improve his own understanding, or to come up with ideas for improving problem areas.

If a large refactoring is partially complete, the team may wish to authorize a conscientious programmer to **finish** those **refactorings**.

There are only two (or three) of us. Should we still pair all the time?

Use your own judgment about when to pair and when you need time to yourself. If you feel fine but your partner is getting cranky, don't escalate; just say you'retired and need a break.

We get engrossed in our work and forget to switch pairs. How can we encourage more frequent pair switching?

One approach is to remember that you can switch when you feel stuck or frustrated. In fact, that is a perfect time to switch partners and get a fresh perspective.

How can we pair remotely?

You can use a phone headset and a desktop sharing tool such as VNC or NetMeeting to pair remotely. I have heard of teams who use individual workstations with shared screensessions and VoIP applications.

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. You enjoy the intense focus and the camaraderie of working with your teammates.

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

Contraindications

Pairing requires a comfortable work environment. If your workspace doesn't allow programmers to sit side by side comfortably, either change the workspace or don't pair program.

Alternatives

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,

Energized Work: XP's practice of energized work recognizes that, although 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.

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.

It may sound silly, but 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. Mid-afternoon lows are also common. Cereal, milk, vegetables, and energy snacks are a good choice. Donuts and junk food, while popular, contribute to the mid-afternoon crash.

The project manager can also help team members do fulfilling work by **pushing back unnecessary meetings and conference calls**. Providing an informative workspace and appropriate reporting can eliminate the need for status meetings. 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.

Taking Breaks

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

Sometimes a snack or walk around the building is good enough. For programmers, switching pairs can help. If it's already the end of the day, though, going home is a good idea. You can usually tell when somebody needs a break. Angry concentration, cursing at the computer, and abrupt movements are all signs. In a highly collaborative environment, going dark—not talking—can also be a sign that someone needs a break.

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.

What if I'm not ready to check in my code and it's time to go home?

If you're practicing test-driven development and continuous integration, your code should be ready to check in every few minutes. If you're struggling with a problem and can't check in, go home anyway. Often the answer will be obvious in the morning.

If you work overtime one week (whatever "overtime" means in your situation), don't work overtime again the next week.

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. Sometimes, that brief zone-out will result in an aha moment of discovery.

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.

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.

Ex: The iteration and release planning boards (just for understanding refer these two figures)

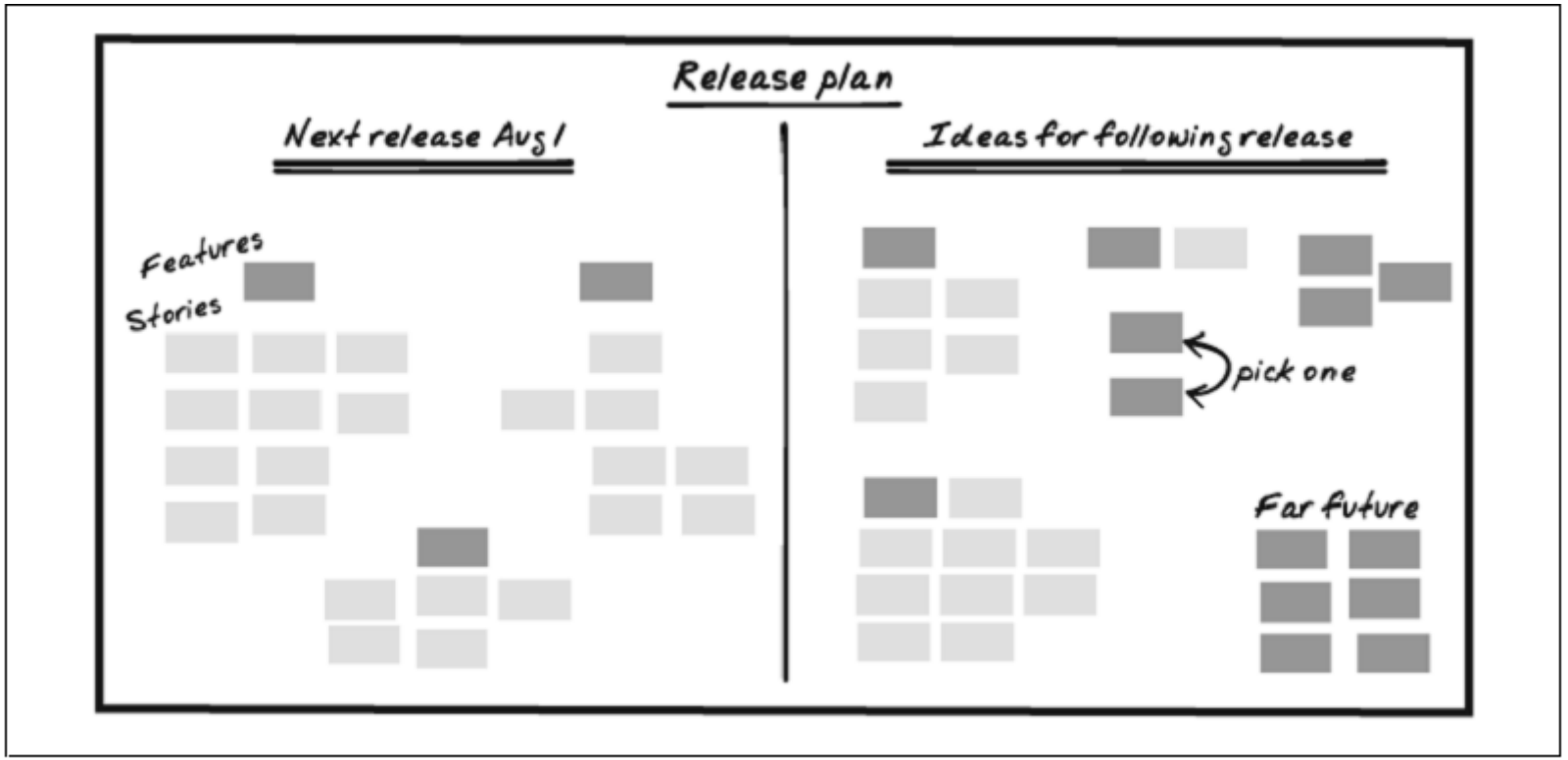


Figure 8-4. A release planning board

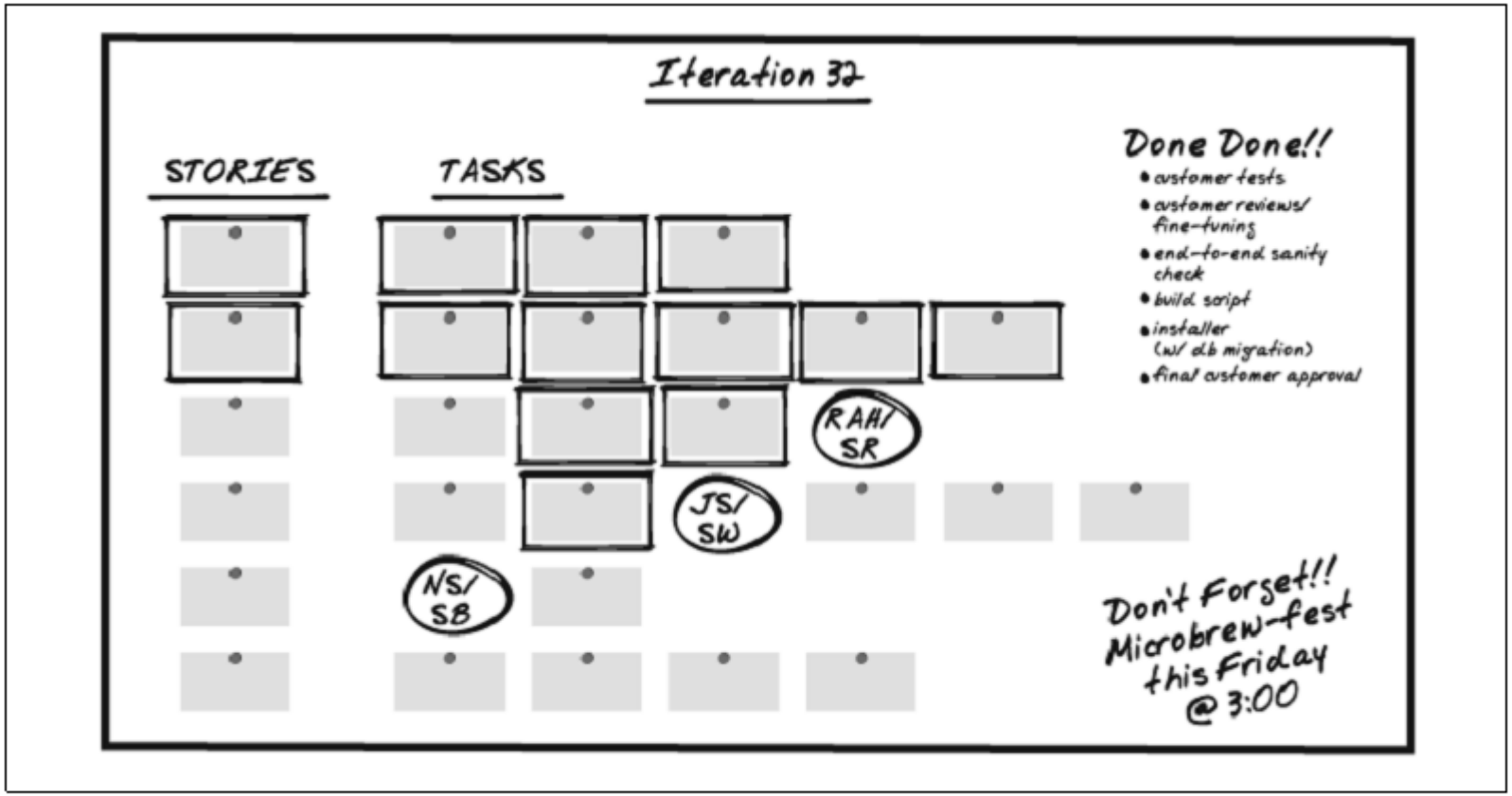


Figure 8-9. An iteration planning board

Another useful status chart is a team calendar, which shows important dates, iteration numbers, and when team members will be out of the office (along with contact information, if appropriate).

Hand-Drawn Charts 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. Often, the issues come up during a retrospective. 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

Questions (pg 90)

1) *We need to share status with people who can't or won't visit the team workspace regularly.*

How do we do that without computerized charts?

A digital camera can effectively capture a whiteboard or other chart. You can even point a
webcam at a chart and webcast it. Get creative

2) Our charts are constantly out of date. How can I get team members to keep them up-to-date?

The first question to ask is, "Did the team really agree to this chart?" An informative workspace is for the team's benefit, so if team members aren't keeping a chart up-to-date, they may not think that it's beneficial. It's possible that the team is passively-aggressively ignoring the chart rather than telling you that they don't want it.

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. OR have a weekly status meeting,

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.

We know what our problems are. Why do we need to bother with root-cause analysis?
If you already understand the underlying causes of your problems, and you're making progress on fixing them, then you have already conducted root-cause analysis. However, it's easy to get stuck on a particular solution. Asking "why" five times may give you new insight.

How do we avoid blaming individuals?
If your root cause points to an individual, ask "why" again.
Why did that person do what she did?
How was it possible for her to make that mistake?
Keep digging until you learn how to prevent that mistake in the future.
Keep in mind that lectures and punitive approaches are usually ineffective. It's better to make it difficult for people to make mistakes than to expect them always to do the right thing.

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.

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

Although few organizations actively discourage root-cause analysis, you may find that it is socially unacceptable. I

f 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

You must continually update **your process to match your changing situations**. Retrospectives are a great tool for doing so.

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.

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.

Questions

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.

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.

The retrospective recipe described here assumes that your team gets along fairly well. If your team doesn't get along well enough to use this recipe, refer to [Derby & Larsen]for more options and consider bringing in an outside facilitator.

If only one or two team members are disruptive, and attempts to work the problem through with them are ineffective, you may be better off removing them from the team. Their antisocial influence probably extends beyond the retrospective, hurting teamwork and productivity.

Alternatives

There are many ways to conduct retrospectives. See [Derby & Larsen]for ideas.

I'm not aware of any other techniques that allow you to improve your process and team cohesiveness as well as retrospectives do. 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, and neither approach fosters team cohesiveness.

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.

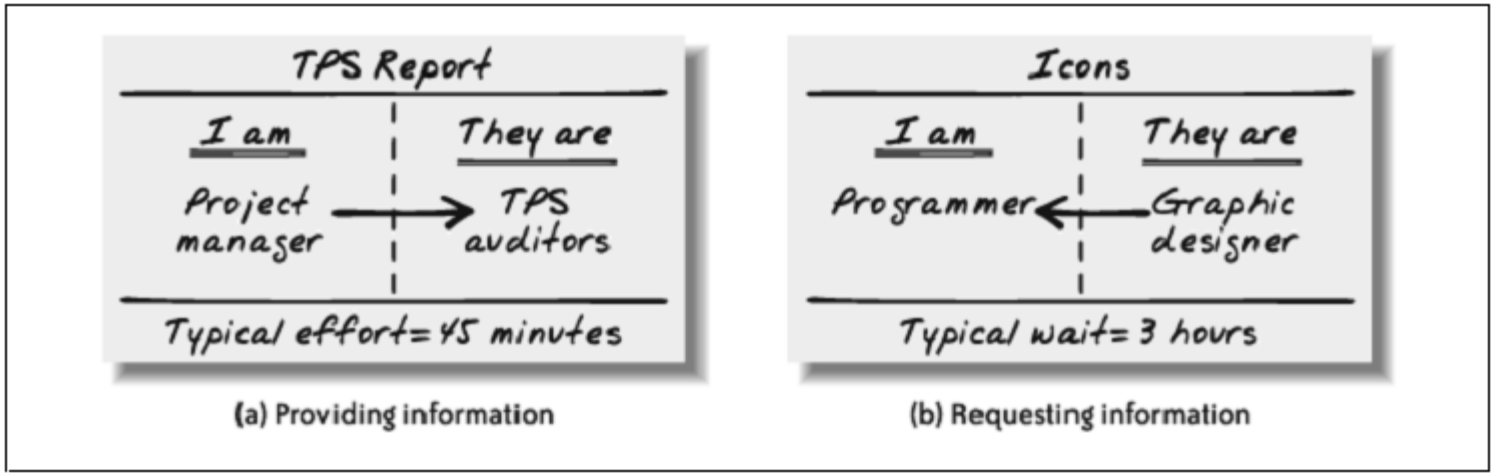


Figure 6-1. Sample cards

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?

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*.

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.

Sit Together

Accommodating Poor Communication

As the distance between people grows, the effectiveness of their communication decreases. Misunderstandings occur and delays creep in. People start guessing to avoid the hassle of waiting for answers. Mistakes appear

Secrets of Sitting Together

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.

A domain expert is often a good backup for a traveling product manager.

Designing Your Workspace

Your team will produce a buzz of conversation in its workspace. Because they'll be working together, this buzz won't be too distracting for team members. For people outside the team, however, it can be very distracting. Make sure there's good sound insulation between your team and the rest of the organization.

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.

An open workspace doesn't leave much room for privacy, and pair programming stations aren't very personal. This loss of individuality can make people uncomfortable. 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.

Some teams include a projector in their workspace as it allows the team to collaborate on a problem without moving to a conference room.

XP workspace :The center of an XP workspace is typically a set of pairing stations.

A hollow triangle, square, or oval setup works well. Provide a few more pairing stations than there are programming pairs. This allows testers and customers to pair as well (either with each other or with programmers),and it provides programmers with space to work solo when they need to.

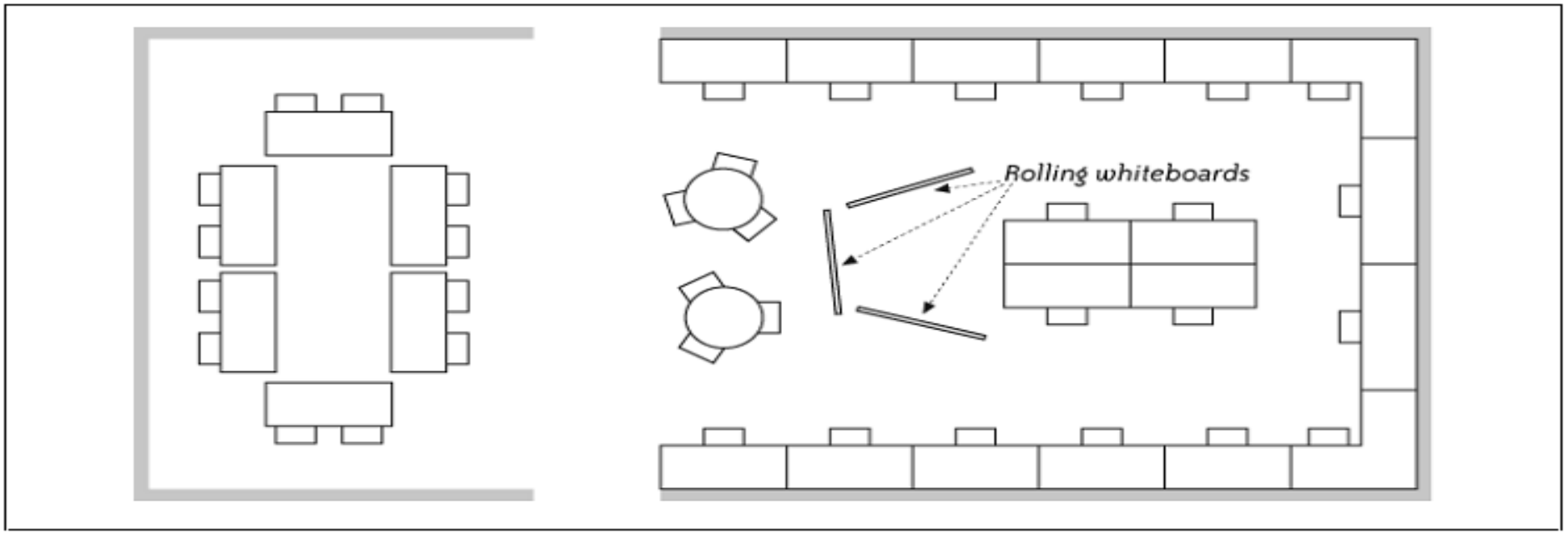


Figure 6-2. A sample workspace

Sample Workspaces

The sample workspace in Figure 6-2 was designed for a team of 13. They had six programmers, six pairing stations, and a series of cubbies for personal effects. Nonprogrammers worked close to the pairing stations so they could be part of the conversation even when they weren't pairing. Programmers' cubbies were at the far end because they typically sat at the pairing stations. For privacy, people adjourned to the far end of the workspace or went to one of the small conference rooms down the hall.

In addition to the pairing stations, everybody had a laptop for personal work and email. The pairing stations all used a group login so any team member could work at them.

This workspace was good, but not perfect. It didn't have nearly enough wall space for charts and whiteboards and nonprogrammers didn't have enough desk space. On the plus side, there was plenty of room to accommodate people at the pairing stations

A small workspace

The small workspace in Figure 6-3 was created by an up-and-coming startup when they moved into new offices. They were still pretty small so they couldn't create a fancy workspace. They had a team of seven: six programmers and a product manager.

This team arranged its five pairing stations along a long wall. They had a table on the side for meetings, and charts and whiteboards on dividers surrounded them. The programmers had a pod of half-cubicles on the other side for personal effects, and there were small conference rooms close by for privacy.

This was a great workspace with one serious problem: the product manager wasn't in earshot and didn't participate in team discussion. The team couldn't get ready answers to its questions and often struggled with requirements.

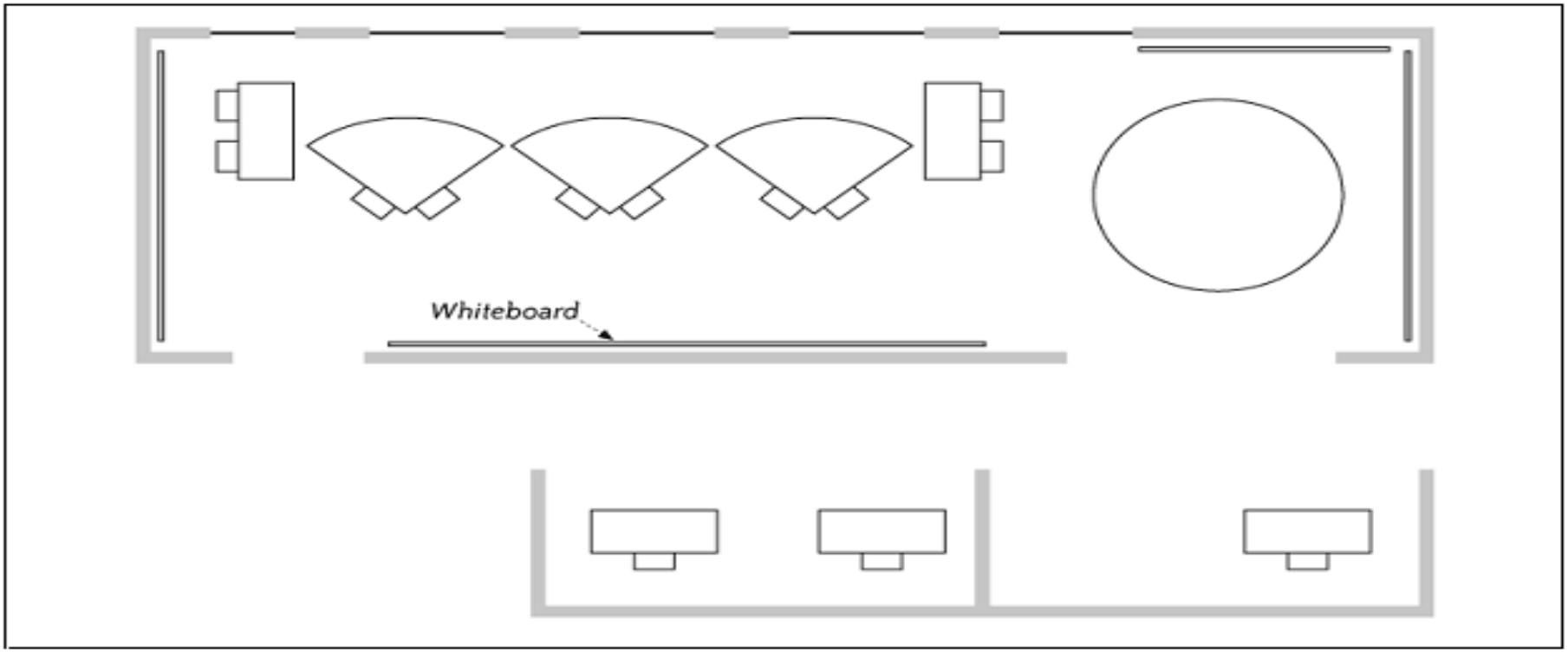


Figure 6-3. A small workspace

Results

When your team sits together, communication is much more effective. You stop guessing at answers and ask more questions.

Real Customer Involvement:

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.

Personal Development: