



CMR ENGINEERING COLLEGE

KANDLAKOYA (V), MEDCHAL (M), HYDERABAD



Step Material on Modern Software Engineering



Subject: Modern Software Engineering

Year: IV Year/II Sem

Academic Year:2020-2021

INSTITUTE VISION AND MISSION

VISION

To be recognized as a premier institution in offering value based and futuristic quality technical education to meet the technological needs of the society.

MISSION

- To impart value based quality technical education through innovative teaching and learning methods
- To continuously produce employable technical graduates with advanced technical skills to meet the current and future technological needs of the society
- To prepare the graduates for higher learning with emphasis on academic and industrial research.

DEPARTMENT VISION AND MISSION

VISION

To create a high quality academic and research environment, empower graduates to attain highest levels of excellence as IT professionals and produce innovative products and leaders to meet societal needs & global challenges.

MISSION

- To provide quality education with basic competence in applied mathematics and computing
- To offer state of art IT education, imparting skills for building cutting edge & innovative IT applications.
- To promote and support student involvement in collaborative IT research and development for a quality product .

PART A

Short Question and Answers

UNIT 1

1. What is Extreme Programming(Xp)- Agile Development Definition?

Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

2. When it is Applicable?

- Dynamically changing software requirements
- Risks caused by fixed time projects using new technology
- Small, co-located extended development team
- The technology you are using allows for automated unit and functional tests

3. Explain about why agile?

Agile development is popular. All the cool kids are doing it: Google, Yahoo, Symantec, Microsoft, and the list goes on.* I know of one company that has already changed its name to Agili-something in order to ride the bandwagon. (They called me in to pitch their “agile process,” which, upon further inspection, was nothing more than outsourced offshore development, done in a different country than usual.) I fully expect the big consulting companies to start offering Certified Agile Processes and Certified Agile Consultants—for astronomical fees, of course—any day now.

4. What is Understanding Success?

The traditional idea of success is delivery on time, on budget, and according to specification. [Standish] provides 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.” Despite their popularity, there’s something wrong with these definitions. A project can be successful even if it never makes a dime. It can be challenged even if it delivers millions of dollars in revenue.

5. Explain about 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.



Figure 1-1 Types of success

people ... particularly the ones signing my paycheck. In fact, for the people funding the work, the value of the software had to exceed its cost. Success meant delivering value to the organization. These definitions aren't incompatible. 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.

6. Describe The Importance of Organizational Success.

Organizational success is often neglected by software teams in favor 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.

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.

7. How will be Organizational Success done?

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.

8. How will be Technical Success done?

Extreme Programming, the agile method I focus on in this book, is particularly adept at achieving technical successes. XP programmers work together, which helps them keep track of the nitpicky 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.

9. How will be Personal Success done?

Personal success is, well, personal. 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
- Users, stakeholders, domain experts, and product managers
- Project and product managers
- Developers
- Testers

10.Explain Agile Methods

- A method, or process, is a way of working. Whenever you do something, you're following a process. Some processes are written, as when assembling a piece of furniture; others are ad hoc and informal, as when I clean my house.
- Agile methods are processes that support the agile philosophy. Examples include Extreme Programming and Scrum.
- Agile methods consist of individual elements called practices. Practices include using version control, setting coding standards, and giving weekly demos to your stakeholders. Most of these practices have been around for years. Agile methods combine them in unique ways, accentuating those parts that support the agile philosophy, discarding the rest, and mixing in a few new ideas. The result is a lean, powerful, self-reinforcing package.

11.Describe about Don't Make Your Own Method.

Just as established agile methods combine existing practices, you might want to create your own agile method by mixing together practices from various agile methods. At first glance, this doesn't seem too hard. There are scores of good agile practices to choose from.

The Agile Manifesto consists of four key values:

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

12.Explain about The Road to Mastery?

The core thesis of this book is that mastering the art of agile development requires real-world experience using a specific, well-defined agile method. I've chosen Extreme Programming for

this purpose. It has several advantages:

Of all the agile methods I know, XP is the most complete. It places a strong emphasis on technical practices in addition to the more common teamwork and structural practices.

XP has undergone intense scrutiny. There are thousands of pages of explanations, experience reports, and critiques out there. Its capabilities and limitations are very well understood.

I have a lot of experience with XP, which allows me to share insights and practical tips that will help you apply XP more easily.

To master the art of agile development—or simply to use XP to be more successful—follow **these steps**:

1. Decide why you want to use agile development. Will it make your team and organization more successful? How?

Teams new to XP often under apply its practices.

2. Determine whether this book’s approach will work for your team.

3. Adopt as many of XP’s practices as you can. XP’s practices are self-reinforcing, so it works best when you use all of them together.

4. Follow the XP practices rigorously and consistently. If a practice doesn’t work, try following the book approach more closely. Teams new to XP often under apply its practices. Expect to take two or three months to start feeling comfortable with the practices and another two to six months for them to become second nature.

5. As you become confident that you are practicing XP correctly—again, give it several months—start experimenting with changes that aren’t “by the book.” Each time you make a change, observe what happens and make further improvements.

13.Explain The XP Lifecycle.

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

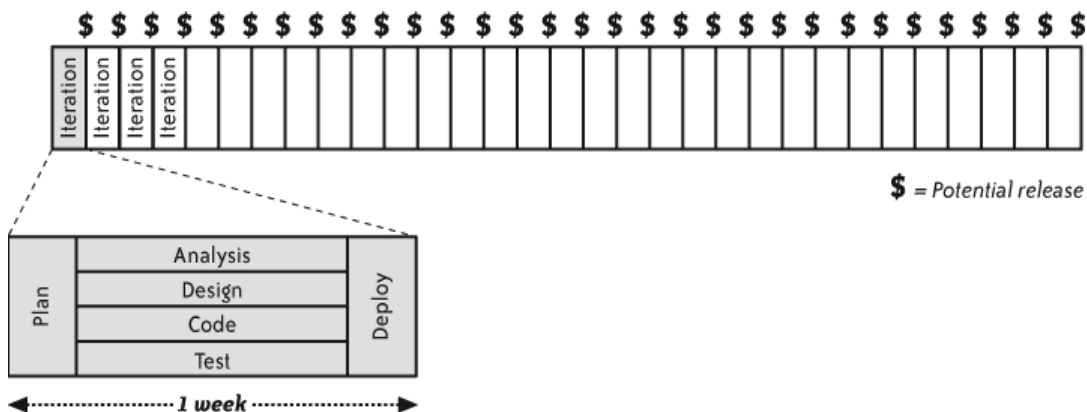
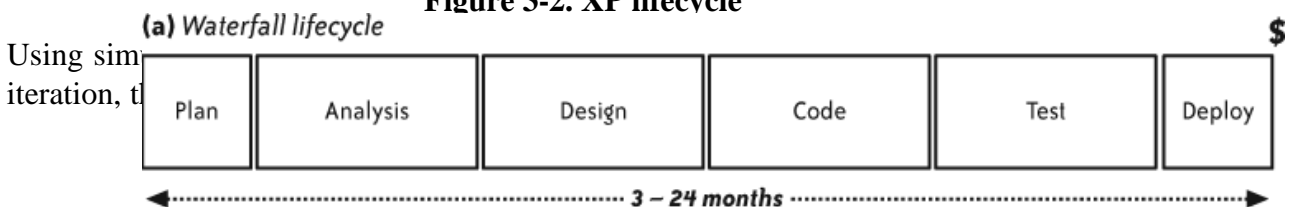


Figure 3-2. XP lifecycle



this approach doesn't necessarily mean that the team is more productive,* it does mean that the team gets feedback much more frequently. As a result, the team can easily connect successes and failures to their underlying causes.

How It Works

XP teams perform nearly every software development activity simultaneously. Analysis, design, coding, testing, and even deployment occur with rapid frequency. That's a lot to do simultaneously. 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.

14.Explain about Planning phase.

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.

In addition to the overall release plan, 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.

15.Explain about Analysis phase.

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.

16. Explain about Design and coding phase.

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 XP includes a sophisticated suite of testing practices. 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 defense 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.

17. What Deployment phase do?

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.

18. What is The XP Team?

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. XP acknowledges this reality by creating cross-functional teams composed of diverse people who can fulfill all the team's roles.

19. Describe 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.

20. Describe about 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.

WHY SO MANY CUSTOMERS?

Two customers for every three programmers seems like a lot, doesn't it? Initially I started with a much smaller ratio, but I often observed customers struggling to keep up with the programmers. Eventually I arrived at the two-to-three ratio after trying different ratios on several successful teams. I also asked other XP coaches about their experiences. The consensus was that the two-to-three ratio was about right.

21. Describe about The product manager (aka product owner).

The product manager has only one job on an XP project, but it's a doozy. That job is to maintain and promote the product vision. In practice, 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.

22. Describe about 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.

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.

Business analysts

On non agile teams, business analysts typically act as liaisons between the customers and developers, by clarifying and refining customer needs into a functional requirements specification.

Programmers

A great product vision requires solid execution. 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. To emphasize this, XP calls all developers programmers.

Designers and architects

Everybody codes on an XP team, and everybody designs. Test-driven development combines design, tests, and coding into a single, ongoing activity.

Technical specialists

In addition to the obvious titles (programmer, developer, software engineer), the XP "programmer" role includes other software development roles. The programmers could include a database designer, a security expert, or a network architect. XP programmers are generalizing specialists.

23. Explain about Testers.

Testers help XP teams produce quality results from the beginning. 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.*

WHY SO FEW TESTERS?

As with the customer ratio, I arrived at the one-to-four tester-to-programmer ratio through

trial and error. In fact, that ratio may be a little high. Successful teams I've worked with have had ratios as low as one tester for every six programmers, and some XP teams have no testers at all.

24. Explain about 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 traditional management roles.

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.

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.

25.Explain about 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.

26.Explain about XP Concepts.

As with any specialized field, XP has its own vocabulary. This vocabulary distills several important concepts into snappy descriptions. Any serious discussion of XP (and of agile in general) uses this vocabulary. Some of the most common ideas follow.

Refactoring

Every day, our code is slightly better than it was the day before. Entropy always wins. Eventually, chaos turns your beautifully imagined and well-designed code into a big mess of spaghetti.

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. Divergent

Change and Shotgun Surgery

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. Yet at some point you need to make a decision. At some point you've identified as many options as you ever will.

The Last Responsible Moment

XP views a potential change as an opportunity to exploit; it's the chance to learn something significant. This is why XP teams delay commitment until the last responsible moment.*
Coddling Nulls

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. .

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. (I recommend one-week iterations for new teams; see "Iteration Planning" 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.

27.Explain about Adopting XP

"I can see how XP would work for IT projects, but product development is different." —a product development team
"I can see how XP would work for product development, but IT projects are different." — an in-house IT development team.

Prerequisite #1: Management Support

Prerequisite #2: Team Agreement

Prerequisite #3: A Colocated Team.

Prerequisite #4: On-Site Customers

Prerequisite #5: The Right Team Size

Prerequisite #6: Use All the Practice

28. Explain about Thinking

What's wrong with this sentence?

What we really need is more keyboards cranking out code.

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

- Pair programming doubles the brainpower available during coding, and gives one person in each pair the opportunity to think about strategic, long-term issues.
- Energized work acknowledges that developers do their best, most productive work when they're energized and motivated.
- An informative workspace gives the whole team more opportunities to notice what's working well and what isn't.
- Root-cause analysis is a useful tool for identifying the underlying causes of your problems.
- Retrospectives provide a way to analyze and improve the entire development process.

29.Explain about 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?

This chapter is called Thinking, yet I included pair programming as the first practice. That's because pair programming is all about increasing your brainpower.

How to Pair

I recommend pair programming on all production code. 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.

Driving and Navigating

When you start pairing, expect to feel clumsy and fumble-fingered as you drive. You may feel that Pairing will feel natural in time. your navigator sees ideas and problems much more quickly than you do. She does—navigators have more time to think than drivers do. The situation will be reversed when you navigate. Pairing will feel natural in time.

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.

30.Explain about Energized Work?

We work at a pace that allows us to do our best, most productive work indefinitely.

I enjoy programming. I enjoy solving problems, writing good code, watching tests pass, and especially removing code while refactoring. I program in my spare time and sometimes even think about work in the shower.

How to Be Energized

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

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 Stay home when you're sick. You risk getting other people sick, too worth. This is particularly true when someone is sick ; in addition to doing poor work, she could infect other people.

31.Explain about Informative Workspace?

We are tuned in to the status of our project.

Your workspace is the cockpit of your development effort. Just as a pilot surrounds himself with information necessary to fly a plane, arrange your workspace with information necessary to steer your project: create an informative workspace.

- Subtle Cues
- Big Visible Charts
- Hand-Drawn Charts
- Process Improvement Charts

32.Explain about Root-Cause Analysis?

We prevent mistakes by fixing our process.

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

How to Find the Root Cause A classic approach to root-cause analysis is to ask “why” five times. Here’s a real-world example.

Problem: When we start working on a new task, we spend a lot of time getting the code into a working state.

Why? Because the build is often broken in source control.

Why? Because people check in code without running their tests. It’s easy to stop here and say, “Aha! We found the problem. People need to run their tests before checking in.” That is a correct answer, as running tests before check-in is part of continuous integration. But it’s also already part of the process. People know they should run the tests, they just aren’t doing it. Dig deeper.

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

When Not to Fix the Root Cause When you first start applying root-cause analysis, you’ll find many more problems than you can address simultaneously. Work on a few at a time. I like to chip away at the biggest problem while simultaneously picking off low-hanging fruit.

33.Explain about Retrospectives?

We continually improve our work habits.

No process is perfect. Your team is unique, as are the situations you encounter, and they change all the time. You must continually update your process to match your changing situations. Retrospectives are a great tool for doing so.

Types of Retrospectives

The most common retrospective, the iteration retrospective, occurs at the end of every iteration.

34.How to Conduct an Iteration Retrospective?

Anybody can facilitate an iteration retrospective if the team gets along well. An experienced, neutral facilitator is best to start with. When the retrospectives run smoothly, give other people a chance to try.

I keep the following schedule in mind as I conduct a retrospective. Don’t try to match the schedule exactly; let events follow their natural pace:

1. Norm Kerth’s Prime Directive
2. Brainstorming (30 minutes)
3. Mute Mapping (10 minutes)
4. Retrospective objective (20 minutes)

Retrospectives are a powerful tool that can actually be damaging when conducted poorly. The process I describe here skips some important safety exercises for the sake of brevity. Pay particular attention to the

contraindications before trying this practice.

Step 1: The Prime Directive

Step 2: Brainstorming

Step 3: Mute Mapping

Step 4: Retrospective Objective

After the Retrospective

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

UNIT-II

35.Explain about collaborating?

This chapter contains eight practices to help your team and its stakeholders collaborate efficiently and effectively:

- Trust is essential for the team to thrive.
- Sitting together leads to fast, accurate communication.
- Real customer involvement helps the team understand what to build.
- A ubiquitous language helps team members understand each other.
- Stand-up meetings keep team members informed.
- Coding standards provide a template for seamlessly joining the team's work together.
- Iteration demos keep the team's efforts aligned with stakeholder goals.
- Reporting helps reassure the organization that the team is working well.

36.Explain about 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” [Tuckman]. It takes the team some time to get through each of these stages. They make progress, fall back, bicker, and get along. Over time—often months—and with adequate support and a bit of luck, they get to know each other and work well together. The team jells. Productivity shoots up. They do really amazing work. Here are some strategies for generating trust in your XP team.

37.What are the Team strategies in Trust?

Team Strategy #1: Customer-Programmer Empathy

Team Strategy #2: Programmer-Tester Empathy

Team Strategy #3: Eat Together

Team Strategy #4: Team Continuity

38.What are the Organizational strategies in Trust?

Organizational Strategy #1: Show Some Hustle

Organizational Strategy #2: Deliver on Commitments

Organizational Strategy #3: Manage Problems

Organizational Strategy #4: Respect Customer Goals

Organizational Strategy #5: Promote the Team
Organizational Strategy #6: Be Honest

39.Explain about Sit Together

We communicate rapidly and accurately. If you've tried to conduct a team meeting via speakerphone, you know how much of a difference face-to-face conversations make. Compared to an in-person discussion, teleconferences are slow and stutter-filled, with uncomfortable gaps in the conversation and people talking over each other.

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.

A Better Way

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

Exploiting Great Communication

Sitting together eliminates the waste caused by waiting for an answer, which dramatically improves productivity. In a field study of six colocated teams, found that sitting together doubled productivity and cut time to market to almost one-third of the company baseline.

40.What are the Secrets of Sitting Together?

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

Making Room

Sitting together is one of those things that's easy to say and hard to do. It's not that the act itself is difficult—the real problem is finding space. A team that sits in adjacent cubicles can convert them into an adequate shared workspace, but even with cubicles, it takes time and money to hire people to rearrange the walls.

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.

41.Discuss about Sample Workspaces.

The sample workspace

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

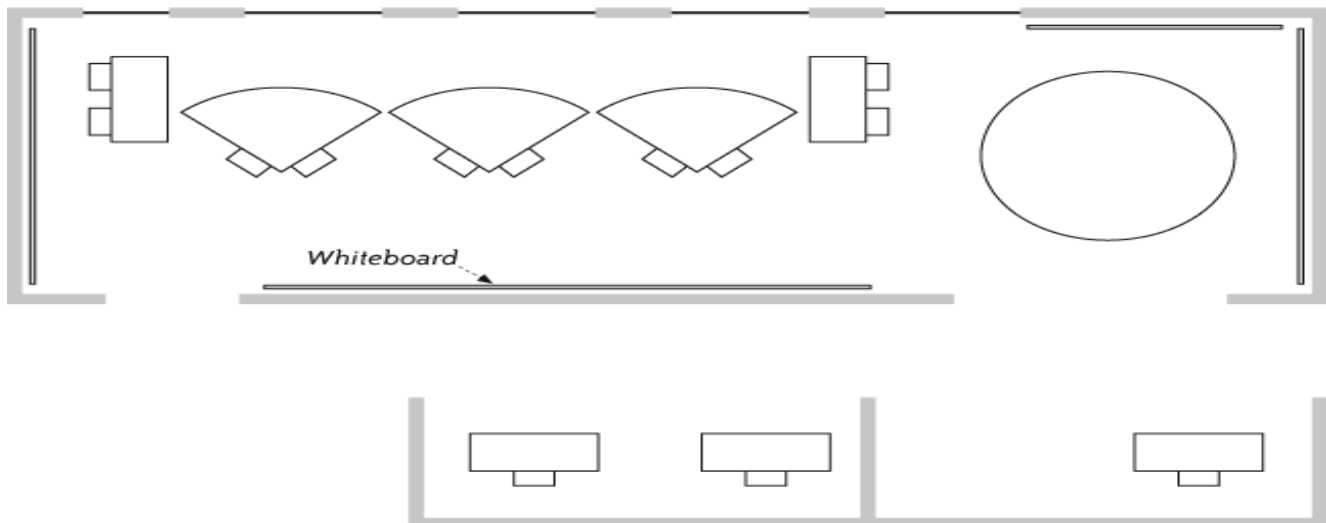
even when they weren't pairing. Programmers' cubbies were at the far end because they typically sat at the pairing stations. For privacy, people adjourned to the far end of the workspace or went to one of the small conference rooms down the hall.

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.

42. Explain about Adopting an Open Workspace ?

Some team members may resist moving to an open workspace. Common concerns include loss of individuality and privacy, implied reduction in status from losing a private office, and managers not recognizing individual contributions. Team members may also mention worries about distractions and noise, but I find that this is usually a cover for one of the other concerns.



However, forcing people to sit together in hopes that they'll come to like it is a bad idea. When I've forced team members to do so, they've invariably found a way to leave the team, even if it meant quitting the

company. Instead, talk with the team about their concerns and the trade-offs of moving to an open workspace.

43.Explain about Real Customer Involvement?

We understand the goals and frustrations of our customers and end-users. An XP team I worked with included a chemist whose previous job involved the software that the team was working to replace. She was an invaluable resource, full of insight about what did and didn't work with the old product. We were lucky to have her as one of our on-site customers—thanks to her, we created a more valuable product.

- Personal Development

- In-House Custom Development
- Outsourced Custom Development

44.Explain about Be Brief(Meetings).

The purpose of a stand-up meeting is to give everybody a rough idea of where the team is. It's not to give a complete inventory of everything happening in the project. The primary virtue of the stand-up meeting is brevity. That's why we stand: our tired feet remind us to keep the meeting short.

- A programmer
- The product manager
- A domain expert
- A programmer responds

45. Explain about Coding Standards

We embrace a joint aesthetic. Back in the days of the telegraph, as the story goes, telegraph operators could recognize each other on the basis of how they keyed their dots and dashes. Each operator had a unique style, or fist, that experts could recognize easily. Programmers have style, too. We each have our own way of producing code. We refine our style over years until we think it's the most readable, the most compact, or the most informative it can be.

Beyond Formatting I once led a team of four programmers who had widely differing approaches to formatting. When we discussed coding standards, I catalogued three different approaches to braces and tabs. Each approach had its own vigorous defender. I didn't want us to get bogged down in arguments, so I said that people could use whatever brace style they wanted.

46. How to Create a Coding Standard?

Creating a coding standard is an exercise in building consensus. It may be one of the first things that programmers do as a team. Over time, you'll amend and improve the standards. The most important thing you may learn from creating the coding standard is how to disagree constructively. To that end, I recommend applying two guidelines:

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

47. What is The best way to start your coding standard ?

The best way to start your coding standard is often to select an industry-standard style guide for your language. This will take care of formatting questions and allow you to focus on design- related questions. If you're not sure what it should encompass, starting points include:

- Development practices
- Tools, key bindings, and IDE
- File and directory layout
- Build conventions
- Error handling and assertions
- Approach to events and logging

- Design conventions (such as how to deal with null references)

48. What is the Dealing with Disagree & Adhering to the Standard?

Dealing with Disagreement It's possible to pressure a dissenter into accepting a coding standard she doesn't agree with, but it's probably not a good idea. Doing so is a good way to create resentment and discord. Instead, remember that few decisions are irrevocable in agile development; mistakes are opportunities to learn and improve. Ward Cunningham put it well:*

Adhering to the Standard People make mistakes. Pair programming helps developers catch mistakes and maintain self-discipline. It provides a way to discuss formatting and coding questions not addressed by the guidelines. It's also an excellent way to improve the standard; it's much easier to suggest an improvement when you can talk it over with someone first.

49. Explain about Iteration Demo.

Iteration Demo We keep it real. An XP team produces working software every week, starting with the very first week. Sound impossible? It's not. It's merely difficult. It takes a lot of discipline to keep that pace. Programmers need discipline to keep the code clean so they can continue to make progress. Customers need discipline to fully understand and communicate one set of features before starting another. Testers need discipline to work on software that changes daily.

The rewards for this hard work are significantly reduced risk, a lot of energy and fun, and the satisfaction of doing great work and seeing progress. The biggest challenge is keeping your momentum. The iteration demo is a powerful way to do so. First, it's a concrete demonstration of the team's progress. The team is proud to show off its work, and stakeholders are happy to see progress.

Second, the demos help the team be honest about its progress. Iteration demos are open to all stakeholders, and some companies even invite external customers to attend. It's harder to succumb to the temptation to push an iteration deadline "just one day" when stakeholders expect a demo. Finally, the demo is an opportunity to solicit regular feedback from the customers. Nothing speaks more clearly to stakeholders than working, usable software. Demonstrating your project makes it and your progress immediately visible and concrete. It gives stakeholders an opportunity to understand what they're getting and to change direction if they need to.

50. How to Conduct an Iteration Demo?

Anybody on the team can conduct the iteration demo, but I recommend that the product manager do so. He has the best understanding of the stakeholders' point of view and speaks their language. His leadership also emphasizes the role of the product manager in steering the product. Invite anybody who's interested. The whole team, key stakeholders, and the executive sponsor should attend as often as possible. Include real customers when appropriate. Other teams working nearby and people who are curious about the XP process are welcome as well. If you can't get everyone in a room, use a teleconference and desktop-sharing software.

Two Key Questions

At the end of the demo, ask your executive sponsor two key questions:*

1. Is our work to date satisfactory?
 2. May we continue?
- These questions help keep the project on

track and remind your sponsor to speak up if she's unhappy. You should be communicating well enough with your sponsor that her answers are never a surprise.

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

51. Explain about Reporting.

We inspire trust in the team's decisions. You're part of a whole team. Everybody sits together. An informative workspace clearly tracks your progress. All the information you need is at your fingertips.

Types of Reports

Progress reports

Management reports

52. Explain about Source lines of code (SLOC) and function points?

Source lines of code (SLOC) and its language- independent cousin, function points, are common approaches to measuring software size. Unfortunately, they're also used for measuring productivity. As with a fancy cell phone, however, software's size does not necessarily correlate to features or value.

Number of stories Some people think they can use the number of stories delivered each iteration as a measure of productivity. Don't do that. Stories have nothing to do with productivity. A normal-sized team will typically work on 4 to 10 stories every iteration. To achieve this goal, they combine and split stories as needed. A team doing this job well will deliver a consistent number of stories each iteration regardless of its productivity.

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

Code quality

There's no substitute for developer expertise in the area of code quality. The available code quality metrics, such as cyclomatic code complexity, all require expert interpretation. There is no single set of metrics that clearly shows design or code quality. The metrics merely recommend areas that deserve further investigation. Avoid reporting code quality metrics. They are a useful tool for developers but they're too ambiguous for reporting to stakeholders.

III UNIT

53. EXPLAIN ABOUT RELEASING.

In order to meet commitments and take advantage of opportunities, you must be able to push your software into production within minutes. This chapter contains 6 practices that give you leverage to turn your big release push into a 10-minute tap:

- "done done" ensures that completed work is ready to release.
- No bugs allows you to release your software without a separate testing phase.

- Version control allows team members to work together without stepping on each other's toes.
- A ten-minute build builds a tested release package in under 10 minutes.
- Continuous integration prevents a long, risky integration phase
- Collective code ownership allows the team to solve problems no matter where they may lie.
- Post-hoc documentation decreases the cost of documentation and increases its accuracy.

54. Explain about “Done Done”

We're done when we're production-ready. Production-Ready Software

Wouldn't it be nice if, once you finished a story, you never had to come back to it? That's the idea behind “done done.” A completed story isn't a lump of un integrated, untested code. It's ready to deploy.

I write mine on the iteration planning board:

- Tested (all unit, integration, and customer tests finished)
- Coded (all code written)
- Designed (code refactored to the team's satisfaction)
- Integrated (the story works from end to end—typically, UI to database—and fits into the rest of the software)
- Builds (the build script includes any new modules)
- Installs (the build script includes the story in the automated installer)
- Migrates (the build script updates database schema if necessary; the installer migrates data when appropriate)
- Reviewed (customers have reviewed the story and confirmed that it meets their expectations)
- Fixed (all known bugs have been fixed or scheduled as their own stories)
- Accepted (customers agree that the story is finished)

55. How to Be “Done Done”?

XP works best when you make a little progress on every aspect of your work every day, rather than reserving the last few days of your iteration for Make a little progress on every aspect of your work every day. getting stories “done done.” This is an easier way to work, once you get used to it, and it reduces the risk of finding unfinished work at the end of the iteration.

Making Time

This may seem like an impossibly large amount of work to do in just one week. It's easier to do if you work on it throughout the iteration rather than saving it up for the last day or two. The real secret, though, is to make your stories small enough that you can completely finish them all in a single week.

56. Explain about No Bugs(Bug Free Release)?

We confidently release without a dedicated testing phase. Let's cook up a bug pie. First, start with a nice, challenging language. How about C? We'll season it with a dash of assembly. Next, add extra bugs by mixing in concurrent programming. Our old friends Safety and Liveness are happy to fight each other over who provides the most bugs. They supplied the Java multithreading library with bugs for years!

57. How No Bugs Possible?

If you're on a team with a bug count in the hundreds, the idea of “no bugs” probably sounds ridiculous. I'll admit: “no bugs” is an ideal to strive for, not something your team will necessarily achieve. However, XP teams can achieve dramatically lower bug rates. [Van Schoonderwoert]'s team averaged one and a half bugs per month in a very difficult domain. In an independent analysis of a company practicing a variant of XP, QSM Associates reported an average reduction from 2,270 defects to 381 defects [Mah].

58. How to Achieve Nearly Zero Bugs ?

Many approaches to improving software quality revolve around finding and removing more defects† through traditional testing, inspection, and automated analysis. The agile approach is to generate fewer

defects. This isn't a matter of finding defects earlier; it's a question of not generating them at all.

To achieve these results, XP uses a potent cocktail of techniques:

1. Write fewer bugs by using a wide variety of technical and organizational practices.
2. Eliminate bug breeding grounds by refactoring poorly designed code.
3. Fix bugs quickly to reduce their impact, write tests to prevent them from reoccurring, then fix the associated design flaws that are likely to breed more bugs.
4. Test your process by using exploratory testing to expose systemic problems and hidden assumptions.
5. Fix your process by uncovering categories of mistakes and making those mistakes impossible.

59. Explain about Version Control

We keep all our project artifacts in a single, authoritative place.

To work as a team, you need some way to coordinate your source code, tests, and other important project artifacts. A version control system provides a central repository that helps coordinate changes to files and also provides a history of changes.

Different version control systems use different terminology. Here are the terms I use throughout this book:

- Repository
- Sandbox
- Check out:
- Update Lock
- Check in or commit Revert Tip or head
- Tag or label etc.,

60. Explain About Fast Build (Ten-Minute Build)

We eliminate build and configuration hassles. Here's an ideal to strive for. Imagine you've just hired a new programmer. On the programmer's first day, you walk him over to the shiny new computer you just added to your open workspace. "We've found that keeping everything in version control and having a really great automated build makes us a lot faster," you say. "Here, I'll show you. This computer is new, so it doesn't have any of our stuff on it yet."

61. How to Automate Your Build?

What if you could build and test your entire product—or create a deployment package—at any time, just by pushing a button? How much easier would that make your life?

Producing a build is often a frustrating and lengthy experience. This frustration can spill over to the rest of your work. "Can we release the software?" "With a few days of work." "Does the software work?" "My piece does, but I can't build everything." "Is the demo ready?" "We ran into a problem with the build—tell everyone to come back in an hour."

How to Automate

Automating your build is one of the easiest ways to improve morale and increase productivity. There are plenty of useful build tools available, depending on your platform and choice of language. If you're using Java, take a look at Ant. In .NET, NAnt and MSBuild are popular. Make is the old standby for C and C++. Perl, Python, and Ruby each have their preferred build tools as well.

62. When to Automate Your Build?

At the start of the project, in the very first iteration, set up a bare-bones build system. The goal of this first iteration is to produce the simplest possible product that exercises your entire system. That includes delivering a working—if minimal—product to stakeholders.

Automating Legacy Projects

If you want to add a build script to an existing system, I have good news and bad news. The good news is

that creating a comprehensive build script is one of the easiest ways to improve your life. The bad news is that you probably have a bunch of technical debt to pay off, so it won't happen overnight.

63. Explain About Continuous Integration?

Continuous integration is a better approach. It keeps everybody's code integrated and builds release infrastructure along with the rest of the application. The ultimate goal of continuous integration is to be able to deploy all but the last few hours of work at any time.

Why It Works

If you've ever experienced a painful multiday (or multiweek) integration, integrating every few hours probably seems foolish. Why go through that hell so often?

Actually, short cycles make integration less painful. Shorter cycles lead to smaller changes, which means there are fewer chances for your changes to overlap with someone else's. That's not to say collisions don't happen. They do. They're just not very frequent because everybody's changes are so small.

64. How to Practice Continuous Integration ?

In order to be ready to deploy all but the last few hours of work, your team needs to do **two** things:

1. Integrate your code every few hours.
2. Keep your build, tests, and other release infrastructure up-to-date.

To integrate, update your sandbox with the latest code from the repository, make sure everything builds, then commit your code back to the repository. You can integrate any time you have a successful build. With test-driven development, that should happen every few minutes. I integrate whenever I make a significant change to the code or create something I think the rest of the team will want right away.

65. How The Continuous Integration Script ?

To guarantee an always-working build, you have to solve two problems. First, you need to make sure that what works on your computer will work on anybody's computer. (How often have you heard the phrase, "But it worked on my machine!?"?) Second, you need to make sure nobody gets code that hasn't been proven to build successfully.

To update from the repository

1. Check that the integration token is available. If it isn't, another pair is checking in unproven code and you need to wait until they finish.
2. Get the latest changes from the repository. Others can get changes at the same time, but don't let anybody take the integration token until you finish.

To integrate

1. Update from the repository (follow the previous script). Resolve any integration conflicts and run the build (including tests) to prove that the update worked.
2. Get the integration token and check in your code.
3. Go over to the integration machine, get the changes, and run the build (including tests).
4. Replace the integration token.

66. How the CONTINUOUS INTEGRATION SERVERS ?

There's a lively community of open-source continuous integration servers (also called CI servers). The granddaddy of them all is CruiseControl, pioneered by ThoughtWorks employees.

Introducing Continuous Integration

The most important part of adopting continuous integration is getting people to agree to integrate frequently (every few hours) and never to break the build. Agreement is the key to adopting continuous integration because there's no way to force people not to break the build.

Dealing with Slow Builds

The most common problem facing teams practicing continuous integration is slow builds. Whenever possible, keep your build under 10 minutes. On new projects, you should be able to keep your build under 10 minutes all the time. On a legacy project, you may not achieve that goal right away. You can still practice

continuous integration, but it comes at a cost.

Multistage Integration Builds

Some teams have sophisticated tests, measuring such qualities as performance, load, or stability, that simply cannot finish in under 10 minutes. For these teams, multistage integration is a good idea. A multistage integration consists of two separate builds. The normal 10-minute build, or commit build, contains all the normal items necessary to prove that the software works: unit tests, integration tests, and a handful of end-to-end. This build runs synchronously as usual.

67. Explain about Collective Code Ownership

We are all responsible for high-quality code. There's a metric for the risk imposed by concentrating knowledge in just a few people's heads—it's called the truck number. How many people can get hit by a truck before the project suffers irreparable harm? It's a grim thought, but it addresses a real risk. What happens when a critical person goes on holiday, stays home with a sick child, takes a new job, or suddenly retires? How much time will you spend training a replacement? Collective code ownership spreads responsibility for maintaining the code to all the programmers. Collective code ownership is exactly what it sounds like: everyone shares responsibility for the quality of the code. No single person claims ownership over any part of the system, and anyone can make any necessary changes anywhere.

Making Collective Ownership Work

Collective code ownership requires letting go of a little bit of ego. Rather than taking pride in your code, take pride in your team's code. Rather than complaining when someone edits your code, enjoy how the code improves when you're not working on it. Rather than pushing your personal design vision, discuss design possibilities with the other programmers and agree on a shared solution.

68. Explain about Documentation

The word documentation is full of meaning. It can mean written instructions for end-users, or detailed specifications, or an explanation of APIs and their use. Still, these are all forms of communication—that's the commonality. Communication happens all the time in a project. Sometimes it helps you get your work done; you ask a specific question, get a specific answer, and use that to solve a specific problem. This is the purpose of work-in-progress documentation, such as requirements documents and design documents. Other communication provides business value, as with product documentation, such as user manuals and API documentation. A third type—handoff documentation—supports the long-term viability of the project by ensuring that important information is communicated to future workers.

69. How will be the Work-In-Progress Documentation ?

In XP, the whole team sits together to promote the first type of communication. Close contact with domain experts and the use of ubiquitous language create a powerful oral tradition that transmits information when necessary. There's no substitute for face-to-face communication. Even a phone call loses important nuances in conversation.

70. What is Product Documentation?

Some projects need to produce specific kinds of documentation to provide business value. Examples include user manuals, comprehensive API reference documentation, and reports. One team I worked with created code coverage metrics—not because they needed them, but because senior management wanted the report to see if XP would increase the amount of unit testing.

UNIT-4

PLANNING

71. What are the practices included in the Planning?

This approach may sound like it's out of control. It would be, except for eight practices that

allow you to control the chaos of endless possibility:

- Vision reveals
- Release Planning
- The Planning Game
- Risk Management
- Iteration Planning
- Slack allows
- Stories
- Estimating

72. Describe about VISION.

Vision

Vision. If there's a more derided word in the corporate vocabulary, I don't know what it is. This word brings to mind bland corporate-speak: "Our vision is to serve customers while maximizing stakeholder value and upholding the family values of our employees." Bleh. Content-free baloney.

Product Vision

Before a project is a project, someone in the company has an idea. Suppose it's someone in the Wizzle-Frobitz company.* "Hey!" he says, sitting bolt upright in bed. "We could frobitz the wizzles so much better if we had some software that sorted the wizzles first!"

Maybe it's not quite that dramatic. The point is, projects start out as ideas focused on results. Sell more hardware by bundling better software. Attract bigger customers by scaling more effectively. Open up a new market by offering a new service. The idea is so compelling that it gets funding, and the project begins.

73. Explain about Where Visions Come From?

Sometimes the vision for a project strikes as a single, compelling idea. One person gets a bright idea, evangelizes it, and gets approval to pursue it. This person is a *visionary*.

More often, the vision isn't so clear. There are multiple visionaries, each with their own unique idea of what the project should deliver.

Identifying the Vision

Like the children's game of telephone, every step between the visionaries and the product manager reduces the product manager's ability to accurately maintain and effectively promote the vision.

If you only have one visionary, the best approach is to have that visionary act as product manager. This reduces the possibility of any telephone-game confusion. As long as the vision is both worthwhile and achievable, the visionary's day-to-day involvement as product manager greatly improves the project's chances of delivering an impressive product.

74. How the Documenting the Vision will be done?

After you've worked with visionaries to create a cohesive vision, document it in a vision statement. It's best to do this collaboratively, as doing so will reveal areas of disagreement and confusion. Without a vision statement, it's all too easy to gloss over

disagreements and end up with an unsatisfactory product.

Once created, the vision statement will help you maintain and promote the vision. It will act as a vehicle for discussions about the vision and a touch point to remind stakeholders why the project is valuable.

Don't forget that the vision statement should be a *living* document: the product manager should review it on a regular basis and make improvements. However, as a fundamental statement of the project's purpose, it may not change much.

75. How to Create a Vision Statement?

The vision statement documents three things: *what* the project should accomplish, *why* it is valuable, and the project's *success criteria*.

The vision statement can be short. I limit mine to a single page. Remember, the vision statement is a clear and simple way of describing why the project deserves to exist. It's not a roadmap; that's the purpose of release planning.

In the first section—*what* the project should accomplish—describe the problem or opportunity that the project will address, expressed as an end result. Be specific, but not prescriptive. Leave room for the team to work out the details.

Here is a real vision statement describing “Sasquatch,” a product developed by two entrepreneurs who started a new company: Sasquatch helps teams collaborate over long distance. It enables the high- quality team dynamics that occur when teams gather around a table and use index cards to brainstorm, prioritize, and reflect.

76. How the Promoting the Vision will be occur?

After creating the vision statement, post it prominently as part of the team's informative workspace. Use the vision to evangelize the project to stakeholders and to explain the priority (or deprioritization) of specific stories.

Be sure to include the visionaries in product decisions. Invite them to release planning sessions. Make sure they see iteration demos, even if that means a private showing. Involve them in discussions with real customers. Solicit their feedback about progress, ask for their help in improving the plan, and give them opportunities to write stories. They can even be an invaluable resource in company politics, as successful visionaries are often senior and influential.

77. Explain about Releasing Planning shortly?

Imagine you've been freed from the shackles of deadlines. “Maximize our return on investment,” your boss says. “We've already talked about the vision for this project. I'm counting on you to work out the details. Create your own plans and set your own release dates just make sure we get a good return on our investment.”

78. How to Create a Release Plan?

There are two basic types of plans: *scopeboxed* plans and *timeboxed* plans.

A scopeboxed plan defines the features the team will build in advance, but the release date is uncertain. A timeboxed plan defines the release date in advance, but the specific features that release will include are uncertain.

Timeboxed plans are almost always better. They constrain the amount of work you can do and force people to make difficult but important prioritization decisions. This requires

the team to identify cheaper, more valuable alternatives to some requests. Without a timebox, your plan will include more low-value features.

79. Describe about Planning at the Last Responsible Moment.

The way to look at this is to think in terms of planning horizons. Your *planning horizon* determines how far you look into the future. Many projects try to determine every requirement for the project up front, thus using a planning horizon that extends to the end of the project.

- Define the *vision* for the entire project.
- Define the *release date* for the next two releases.
- Define the *minimum marketable features* for the current release, and start to place features that won't fit in this release into the next release.
- Define all the *stories* for the current feature and most of the current release. Place stories that don't fit into the next release.
- *Estimate and prioritize* stories for the current iteration and the following three iterations.
- Determine *detailed requirements and customer tests* for the stories in the current iteration.

80. Explain about RISK MANAGEMENT

Every project faces a set of common risks: turnover, new requirements, work disruption, and so forth. These risks act as a multiplier on your estimates, doubling or tripling the amount of time it takes to finish your work.

Because most organizations don't have this information available, I've provided some generic risk multipliers instead. These multipliers show your chances of meeting various schedules. For example, in a "Risky" approach, you have a 10 percent chance of finishing according to your estimated schedule. Doubling your estimates gives you a 50 percent chance of on-time completion, and to be virtually certain of meeting your schedule, you have to quadruple your estimates.

81. What is Project-Specific Risks?

Using the XP practices and applying risk multipliers will help contain the risks that are common to all projects. The generic risk multipliers include the normal risks of a flawed release plan, ordinary requirements growth, and employee turnover. In addition to these risks, you probably face some that are specific to your project. To manage these, create a *risk census*—that is, a list of the risks your project faces that focuses on your project's *unique* risks.

Suggest starting work on your census by brainstorming catastrophes. Gather the whole team and hand out index cards. Remind team members that during this exercise, negative thinking is not only OK, it's necessary.

82. How to Make a Release Commitment?

With your risk exposure and risk multipliers, you can predict how many story points you can finish before your release date. Start with your timeboxed release date from your release plan. Figure out how many iterations remain until your release date and subtract your risk exposure. Multiply by your velocity to determine the number of points remaining in your schedule, then divide by each risk multiplier to calculate your chances

of finishing various numbers of story points.

$$\text{risk_adjusted_points_remaining} = (\text{iterations_remaining} - \text{risk_exposure}) * \text{velocity} / \text{risk_multiplier}$$

For example, if you're using a rigorous approach, your release is 12 iterations away, your velocity is 14 points, and your risk exposure is one iteration, you would calculate the range of possibilities as:

```
points remaining = (12 - 1) * 14 = 154 points
10 percent chance: 154 / 1 = 154 points
50 percent chance: 154 / 1.4 = 110 points
90 percent chance: 154 / 1.8 = 86 points
```

In other words, when it's time to release, you're 90 percent likely to have finished 86 more points of work, 50 percent likely to have finished 110 more points, and only 10 percent likely to have finished 154 more points.

83. Explain about Iteration Planning?

Iterations are the heartbeat of an XP project. When an iteration starts, stories flow in to the team as they select the most valuable stories from the release plan. Over the course of the iteration, the team breathes those stories to life. By the end of the iteration, they've pumped out working, tested software for each story and are ready to begin the cycle again.

Iterations are an important safety mechanism. Every week, the team stops, looks at what it's accomplished, and shares those accomplishments with stakeholders. By doing so, the team coordinates its activities and communicates its progress to the rest of the organization. Most importantly, iterations counter a common risk in software projects: the tendency for work to take longer than expected.

84. Discuss about The Iteration Timebox.

Iterations allow you to avoid this surprise. Iterations are exactly one week long and have a strictly defined completion time. This is a *timebox*: work ends at a particular time regardless of how much you've finished. Although the iteration timebox doesn't *prevent* problems, it *reveals* them, which gives you the opportunity to correct the situation.

In XP, the iteration demo marks the end of the iteration. Schedule the demo at the same time every week. Most teams schedule the demo first thing in the morning, which gives them a bit of extra slack the evening before for dealing with minor problems.

85. How to Plan an Iteration?

After the iteration demo and retrospective are complete, iteration planning begins. Start by measuring the velocity of the previous iteration. Take all the stories that are "done done" and add up their original estimates. This number is the amount of story points you can reasonably expect to complete in the upcoming iteration. With your velocity in hand, you can select the stories to work on this iteration. Ask your customers to select the most

important stories from the release plan. Select stories that exactly add up to the team's velocity. You may need to split stories or include one or two less important stories to make the estimates add up perfectly.

Engineering tasks are concrete tasks for the programmers to complete. Unlike stories, engineering tasks don't need to be customer-centric. Instead, they're programmer-centric. Typical engineering tasks include:

- Update build script
- Implement domain logic
- Add database table and associated ORM objects
- Create new UI form

86. Explain about SLACK?

Imagine that the power cable for your workstation is just barely long enough to reach the wall receptacle. You can plug it in if you stretch it taut, but the slightest vibration will cause the plug to pop out of the wall and the power to go off. You'll lose everything you were working on.

I can't afford to have my computer losing power at the slightest provocation. My work's too important for that. In this situation, I would move the computer closer to the outlet so that it could handle some minor bumps. People couldn't trip over it, install an uninterruptable power supply, and invest in a continuous backup server.)

Your project plans are also too important to be disrupted by the slightest provocation. Like the power cord, they need slack.

87. How Much Slack?

The amount of slack you need doesn't depend on the number of problems you face. It depends on the *randomness* of problems. If you always experience exactly 20 hours of problems in each iteration, your velocity will automatically compensate. However, if you experience between 20 and 30 hours of problems in each iteration, your velocity will bounce up and down. You need 10 hours of slack to stabilize your velocity and to ensure that you'll meet your commitments.

88. How to Introduce Slack?

One way to introduce slack into your iterations might be to schedule no work on the last day or two of your iteration. This would give you slack, but it would be pretty wasteful. A better approach is to schedule useful, important work that isn't time-critical work you can set aside in case of an emergency. *Paying down technical debt* fits the bill perfectly.

Even the best teams in advertantly accumulate technical debt. Although you should always make your code as clean as you can, some technical debt will slip by unnoticed.

89. Explain about STORIES?

Stories may be the most misunderstood entity in all of XP. They're not requirements. They're not use cases. They're not even narratives. They're much simpler than that.

Stories are for planning. They're simple one- or two-line descriptions of work the team should produce. Alistair Cockburn calls them "promissory notes for future conversation. Everything that stakeholders want the team to produce should have a story.

90. Write about Story Cards?

Story cards also form an essential part of an informative workspace. After the planning meeting, move the cards to the release planning board—a big, six-foot whiteboard, placed prominently in the team's open workspace. You can post hundreds of cards and still see them all clearly. For each iteration, place the story cards to finish during the iteration on the iteration planning board (another big whiteboard; and move them around to indicate your status and progress. Both of these boards are clearly visible throughout the team room and constantly broadcast information to the team.

91. How to Improve Your Velocity?

The following options might allow you to improve your velocity.

- Pay down technical debt
- Improve customer involvement
- Improve customer involvement
- Support energized work
- Offload programmer duties

UNIT-5 DEVELOPING

92. Explain the Practices of Developing.

Here are nine practices that keep the code clean and allow the entire team to contribute to development:

- Incremental Requirements
- Customer Tests
- Test-Driven Development
- Refactoring
- Simple Design
- Incremental Design and Architecture
- Spike Solutions
- Performance Optimization
- Exploratory Testing

93. Write about CUSTOMER TESTS?

Customer tests are really just examples. Your programmers turn them into automated tests, which they then use to check that they've implemented the domain rules correctly. Once the tests are passing, the programmers will include them in their 10-minute build, which will inform the programmers if they ever do anything to break the tests.

To create customer tests, follow the Describe, Demonstrate, Develop processes outlined in the next section. Use this process during the iteration in which you develop the corresponding stories.

94. What is TEST-DRIVEN DEVELOPMENT?

Test-driven development, or TDD, is a rapid cycle of testing, coding, and refactoring. When adding a feature, a pair may perform dozens of these cycles, implementing and refining the software in baby steps until there is nothing left to add and nothing left to take away.

Research shows that TDD substantially reduces the incidence of defects. When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

TDD isn't perfect, of course. TDD is difficult to use on legacy codebases. Even with green field systems, it takes a few months of steady use to overcome the learning curve.

Try it anyway although TDD benefits from other XP practices, it doesn't require them. You can use it on almost any project.

95. Why TDD Works?

TDD uses an approach similar to double-entry bookkeeping. You communicate your intentions twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere. Getting code to compile isn't such a big deal anymore. Most IDEs check your syntax as you type, and some even compile every time you save. The feedback loop is so fast that errors are easy to find and fix. If something doesn't compile, there isn't much code to check. Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code.

96. How to Use TDD ?

You can start using TDD today. It's one of those things that takes moments to learn and a lifetime to master. Imagine TDD as a small, fast-spinning motor. It operates in a very short cycle that repeats over and over again.

Step 1: Think TDD uses small tests to force you to write your code

Step 2: Red bar Now write the test. Write only enough code for the current increment of behavior

Step 3: Green bar Next, write just enough production code to get the test to pass.

Step 4: Refractor With all your tests passing again, you can now refactor without worrying about breaking anything.

Step 5: Repeat when you're ready to add new behavior, start the cycle over again.

97. What is REFACTORING?

Entropy always wins. Eventually, chaos turns your beautifully imagined and well-designed code into a big mess of spaghetti. At least, that's the way it used to be, before refactoring. Refactoring is the process of changing the design of your code without changing its behavior—what it does stays the same, but how it does it changes. Refactorings are also reversible; sometimes one form is better than another for certain cases.

98. What is the PERFORMANCE OPTIMIZATION?

Our organization had a problem.* Every transaction our software processed had a three- second latency. During peak business hours, transactions piled up and with our recent surge in sales, the lag sometimes became hours. We cringed every time the phone rang; our customers were upset.

We knew what the problem was: we had recently changed our order preprocessing code. I remember thinking at the time that we might need to start caching the intermediate results of expensive database queries. I had even asked our customers to schedule a performance story. Other stories had been more important, but now performance was top priority.

99. How to Optimize?

Modern computers are complex. Reading a single line of a file from a disk requires the coordination of the CPU, the kernel, a virtual file system, a system bus, the hard drive controller, the hard drive cache, OS buffers, system memory, and scheduling pipelines. Every component exists to solve a problem, and each has certain tricks to squeeze out performance. Is the data in a cache? Which cache? How's your memory aligned? Are you reading asynchronously or are you blocking? There are so many variables it's nearly impossible to predict the general performance of any single method. The days in which a programmer could accurately predict performance by counting instruction clock cycles are long gone, yet some still approach performance with a simplistic, brute-force mindset.

100. What is the EXPLORATORY TESTING?

XP teams have no separate QA department. There's no independent group of people responsible for assessing and ensuring the quality of the final release. Instead, the whole team—customers, programmers, and testers—is responsible for the outcome. On traditional teams, the QA group is often rewarded for finding bugs. On XP teams, there's no incentive program for finding or removing bugs. The goal in XP isn't to find and remove bugs; the goal is not to write bugs in the first place. In a well-functioning team, bugs are a rarity—only a handful per month.

Does that mean testers have no place on an XP team? No! Good testers have the ability to look at software from a new perspective, to find surprises, gaps, and holes. It takes time for the team to learn which mistakes to avoid. By providing essential information about what the team overlooks, testers enable the team to improve their work habits and achieve their goal of producing zero bugs.

PART-B

Long Question and answers

1. Explain about why agile?

Agile development is popular. All the cool kids are doing it: Google, Yahoo, Symantec, Microsoft, and the list goes on.* I know of one company that has already changed its name to Agili-something in order to ride the bandwagon. (They called me in to pitch their “agile process,” which, upon further inspection, was nothing more than outsourced offshore development, done in a different country than usual.) I fully expect the big consulting companies to start offering Certified Agile Processes and Certified Agile Consultants—for astronomical fees, of course—any day now.

- a) **Understanding Success:** The traditional idea of success is delivery on time, on budget, and according to specification. [Standish] provides 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.” Despite their popularity, there's something wrong with these definitions. A project can be successful even if it never makes a dime. It can be challenged even if it delivers millions of dollars in revenue.

b) **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.



Figure 1-1 Types of success

people ... particularly the ones signing my paycheck. In fact, for the people funding the work, the value of the software had to exceed its cost. Success meant delivering value to the organization. These definitions aren't incompatible. 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.

c) The Importance of Organizational Success

Organizational success is often neglected by software teams in favor 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.

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.

▪ **Technical Success**

Extreme Programming, the agile method I focus on in this book, is particularly adept at achieving technical successes. XP programmers work together, which helps them keep track of the nitpicky 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.

▪ **Personal Success**

Personal success is, well, personal. Agile development may not satisfy all of your requirements for personal success. However, once you get used to it, you'll probably find a lot to like about it, no matter who you are:

➤ **Executives and senior management**

They will appreciate the team's focus on providing a solid return on investment and the software's longevity.

➤ **Users, stakeholders, domain experts, and product managers**

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

➤ **Project and product managers**

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

➤ **Developers**

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

➤ **Testers**

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

2) Explain about how to Be Agile?

A) Agile Methods

- A method, or process, is a way of working. Whenever you do something, you're following a process. Some processes are written, as when assembling a piece of furniture; others are ad hoc and informal, as when I clean my house.
- Agile methods are processes that support the agile philosophy. Examples include Extreme Programming and Scrum.
- Agile methods consist of individual elements called practices. Practices include using version control, setting coding standards, and giving weekly demos to your stakeholders. Most of these practices have been around for years. Agile methods combine them in unique ways, accentuating those parts that support the agile philosophy, discarding the rest, and

mixing in a few new ideas. The result is a lean, powerful, self-reinforcing package.

B) Don't Make Your Own Method

Just as established agile methods combine existing practices, you might want to create your own agile method by mixing together practices from various agile methods. At first glance, this doesn't seem too hard. There are scores of good agile practices to choose from.

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

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, the above authors
This declaration may be freely copied in any form,
but only in its entirety through this notice.

Figure 2-1. Agile values

triple- duty, solving multiple software development problems simultaneously and supporting each other in clever and surprising ways.

Every project and situation is unique, of course, so it's a good idea to have an agile method that's customized to your situation. Rather than making an agile method from scratch, start with an existing, proven method and iteratively refine it. Apply it to your situation, note where it works and doesn't, make an educated guess about how to improve, and repeat. That's what experts do.

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

c) The Road to Mastery

The core thesis of this book is that mastering the art of agile development requires real-world experience using a specific, well-defined agile method. I've chosen Extreme Programming for this purpose. It has several advantages:

- Of all the agile methods I know, XP is the most complete. It places a strong emphasis on technical practices in addition to the more common teamwork and structural practices.
- XP has undergone intense scrutiny. There are thousands of pages of explanations, experience reports, and critiques out there. Its capabilities and limitations are very well understood.
- I have a lot of experience with XP, which allows me to share insights and practical tips that will help you apply XP more easily.

To master the art of agile development—or simply to use XP to be more successful—follow **these steps:**

1. Decide why you want to use agile development. Will it make your team and organization more successful? How?

Teams new to XP often under apply its practices.

2. Determine whether this book's approach will work for your team.

3. Adopt as many of XP's practices as you can. XP's practices are self-reinforcing, so it works

best when you use all of them together.

4. Follow the XP practices rigorously and consistently. If a practice doesn't work, try following the book approach more closely. Teams new to XP often under apply its practices. Expect to take two or three months to start feeling comfortable with the practices and another two to six months for them to become second nature.

5. As you become confident that you are practicing XP correctly—again, give it several months—start experimenting with changes that aren't "by the book." Each time you make a change, observe what happens and make further improvements.

3) Explain about Understanding XP ?

- "Welcome to the team, Pat," said Kim, smiling at the recent graduate. "Let me show you around. As I said during the interview, we're an XP shop. You may find that things are a little different here than you learned in school."
- "I'm eager to get started," said Pat. "I took a software engineering course in school, and they taught us about the software development lifecycle. That made a lot of sense. There was a bit about XP, but it sounded like it was mostly about working in pairs and writing tests first. Is that right?"
- "Not exactly," said Kim. "We do use pair programming, and we do write tests first, but there's much more to XP than that. Why don't you ask me some questions? I'll explain how XP is different than what you learned."
- Pat thought for a moment. "Well, one thing I know from my course is that all development methods use the software development lifecycle: analysis, design, coding, and testing [see Figure 3-1]. Which phase are you in right now? Analysis? Design? Or is it coding or testing?"
- "Yes!" Kim grinned. She couldn't help a bit of showmanship.
- "I don't understand. Which is it?"
- "All of them .We're working on analysis, design, coding, *and* testing. Simultaneously. Oh, and we deploy the software every week, too."
- Pat looked confused. Was she pulling his leg?
- Kim laughed. "You'll see! Let me show you around.
- "This is our team room .As you can see, we all sit together in one big workspace. This helps us collaborate more effectively."
- Kim led Pat over to a big whiteboard where a man stood frowning at dozens of index cards. "Brian, I'd like you to meet Pat, our new programmer. Brian is our product manager. What are you working on right now?"

(a) Waterfall lifecycle

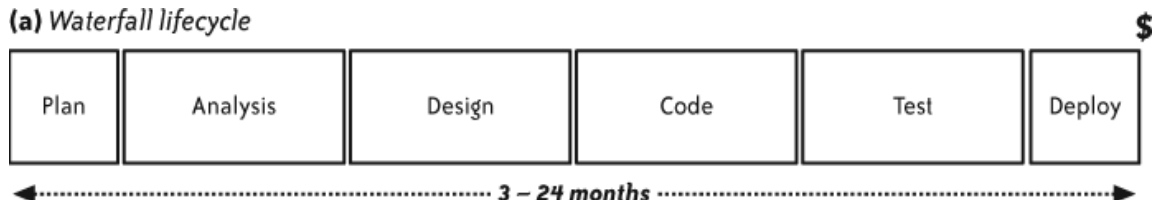


Figure 3-1. Traditional lifecycles

- “I’m working on making our stakeholders happy,” Brian shrugged, turning back to the whiteboard.
- “Don’t mind him,” Kim whispered to Pat as they walked away. “He’s under a lot of pressure right now. This whole project was his idea. It’s already saved the company two and a half million dollars, but now there’s some political stuff going on. Luckily, we programmers don’t have to worry about that. Brian and Rachel take care of it—Rachel’s our project manager.

a) 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.

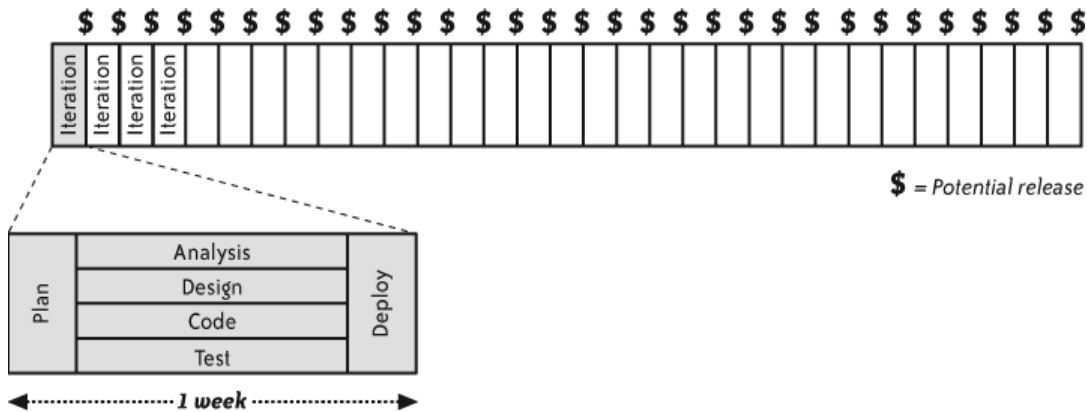


Figure 3-2. XP lifecycle

Using simultaneous phases, an XP team produces deployable software every week. In each iteration, the team analyzes, designs, codes, tests, and deploys a subset of features. Although this approach doesn’t necessarily mean that the team is more productive,* it does mean that the team gets feedback much more frequently. As a result, the team can easily connect successes and failures to their underlying causes.

How It Works

XP teams perform nearly every software development activity simultaneously. Analysis, design, coding, testing, and even deployment occur with rapid frequency. That’s a lot to do simultaneously. 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.

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.

In addition to the overall release plan, 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.

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 XP includes a sophisticated suite of testing practices. 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 defense 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.

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.

B) The XP Team

Working solo on your own project—“scratching your own itch”—can be a lot of fun. There are no questions about which features to work on, how things ought to work, if the software works correctly, or whether stakeholders are happy. All the answers are right there in one brain.

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. XP acknowledges this reality by creating cross-functional teams composed of diverse people who can fulfill all the team's roles.

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.

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.

WHY SO MANY CUSTOMERS?

Two customers for every three programmers seems like a lot, doesn't it? Initially I started with a much smaller ratio, but I often observed customers struggling to keep up with the programmers. Eventually I arrived at the two-to-three ratio after trying different ratios on several successful teams. I also asked other XP coaches about their experiences. The consensus was that the two-to-three ratio was about right.

The product manager (aka product owner)

The product manager has only one job on an XP project, but it's a doozy. That job is to maintain and promote the product vision. In practice, 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.

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.

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.

Business analysts

On non agile teams, business analysts typically act as liaisons between the customers and developers, by clarifying and refining customer needs into a functional requirements specification.

Programmers

A great product vision requires solid execution. 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. To emphasize this, XP calls all developers programmers.

Designers and architects

Everybody codes on an XP team, and everybody designs. Test-driven development combines design, tests, and coding into a single, ongoing activity.

Technical specialists

In addition to the obvious titles (programmer, developer, software engineer), the XP "programmer" role includes other software development roles. The programmers could include a database designer, a security expert, or a network architect. XP programmers are generalizing specialists.

Testers

Testers help XP teams produce quality results from the beginning. 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.*

WHY SO FEW TESTERS?

As with the customer ratio, I arrived at the one-to-four tester-to-programmer ratio through trial and error. In fact, that ratio may be a little high. Successful teams I've worked with have had ratios as low as one tester for every six programmers, and some XP teams have no testers at all.

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 traditional management roles.

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.

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.

Other Team Members

The preceding roles are a few of the most common team roles, but this list is by no means comprehensive. The absence of a role does not mean the expertise is inappropriate for an XP team; an XP team should include exactly the expertise necessary to complete the project successfully and cost-effectively. For example, one team I worked with included a technical writer and an ISO 9001 analyst.

The Project Community

Projects don't live in a vacuum ; 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.

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

Filling Roles The exact structure of your team isn't that important as long as it has all the knowledge it needs. The makeup of your team will probably depend more on your

Organization's traditions than on anything else.

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.

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.

C) XP Concepts

As with any specialized field, XP has its own vocabulary. This vocabulary distills several important concepts into snappy descriptions. Any serious discussion of XP (and of agile in general) uses this vocabulary. Some of the most common ideas follow.

Refactoring

Every day, our code is slightly better than it was the day before. Entropy always wins. Eventually, chaos turns your beautifully imagined and well-designed code into a big mess of spaghetti.

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. Divergent Change and Shotgun Surgery

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. Yet at some point you need to make a decision. At some point you've identified as many options as you ever will.

The Last Responsible Moment

XP views a potential change as an opportunity to exploit; it's the chance to learn something significant. This is why XP teams delay commitment until the last responsible moment.*
Coddling Nulls

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. .

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. (I recommend one-week iterations for new teams; see "Iteration Planning")

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.

Theory of Constraints [Goldratt 1992]'s Theory of Constraints says, in part, that every system has a single constraint that determines the overall throughput of the system. This book assumes that programmers are the constraint on your team. 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 the programmers, the work piles up, falls out of date and needs reworking, and slows the programmers further.

Mindfulness Agility—the ability to respond effectively to change—requires that everyone pay attention to the process and practices of development. This is mindfulness.

4) Explain about Adopting XP.

“I can see how XP would work for IT projects, but product development is different.” —a product development team
“I can see how XP would work for product development, but IT projects are different.” — an in-house IT development team.

A) Is XP Right for Us?

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 as described in this book, 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

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.

Prerequisite #2: Team Agreement Just as important as management support is the team's agreement to use XP. 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.

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 need to sit together in the same room.

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 on-site customers' decisions determine the value of the software.

Domain experts, and possibly interaction designers, are also important. They 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...

If you have an experienced product manager who makes high-level decisions, it's fine if your product manager is inexperienced...

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

If you can't get a product manager at all...

Although good product managers are in high demand, the absence of a product manager is a big danger sign.

If you can't get other on-site customers... Because XP doesn't have an upfront requirements phase, the work of figuring out requirements happens concurrently with software development. This compresses the overall schedule, and it means that at least one person—usually several—needs to work on requirements full-time.

Prerequisite #5: The Right Team Size

I wrote this book for teams as large as 20 people and as small as 1 person. For teams new to XP, however, I recommend 4 to 6 programmers and no more than 12 people on the team.

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 you can't do that, the XP practices are still appropriate for you, but try to find useful nonproduction code work for the programmer who isn't pairing. This will help the team consistently apply XP's technical practices and will improve code quality.

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. You may also benefit from hiring another experienced XP programmer to assist the coach in mentoring the team.

If your team is smaller than four programmers... Most of the XP practices are still appropriate for you, but you probably won't be able to pair program much. In this situation, it's best if your team members are conscientious programmers who are passionate about producing high-quality code. That passion will help them apply XP's technical practices with discipline.

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.

Prerequisite #6: Use All the Practices

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.

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 spill over into XP's planning practices

Recommendation #2: Strong Design Skills Simple, easily changed design is XP's core enabler. This means at least one person on the team—preferably a natural leader—needs to have strong design skills.

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.

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.

EXTREME SHOPPING

As your project start date draws near, you'll need supplies for the team's open workspace. The following is a good shopping list

Equipment:

- Pairing stations
- A dedicated build machine
- Noise-dampening partitions to define your team's workspace and prevent noise pollution

Software:

- A unit-testing tool such as the xUnit family
- An automated build tool such as the Ant family
- Any other software you normally use.

Supplies:

- Index cards—start with 5,000 of white and 2,000 of each color you want. Be sure to choose colors that all members of your team can distinguish (7 to 10 percent of men have a degree of color blindness).

The Challenge of Change

It's a fact of life: change makes people uncomfortable. XP is probably a big change for your team. If you previously used a rigid, document-centric process, XP will seem loose and informal.

Final Preparation

- Before starting XP, it's a good idea to discuss working agreements—that is, which practices your team will follow and how your practice of XP will differ from what I describe in this book.

SECOND ADOPTER SYNDROME

I've noticed a surprising trend among companies that adopt XP: the first team is often very successful, inspiring the organization to use XP on more projects, but then this second wave of XP projects struggles.

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.

Applying XP to an Existing Project Greenfield projects can adopt all the XP practices at once. You'll experience some bumps along the way, but you'll typically have things figured out in four to nine months.

- **The big decision**

- **Bring order to chaos**
- All the “Thinking” practices
 - All the “Collaborating” practices
 - All the “Planning” practices
- **Pay down technical debt**
- **Organize your backlog**
- **Fix important bugs**
- **Move testers forward**
- **Emerge from the darkness**

Applying XP in a Phase-Based Organization

XP assumes that you use iterations, not phases, which makes using XP in a phase-based environment difficult.

- Mandatory planning phase
- Mandatory analysis phase
- Mandatory design phase
- Mandatory coding phase
- Mandatory testing phase
- Mandatory deployment phase

Extremities: Applying Bits and Pieces of XP

What if your team doesn’t meet this book’s conditions for using XP? What then? Although you won’t be able to use all the XP practices in this book, you may be able to add some practices to your existing method. Several practices are easy to adopt and are likely to make an immediate difference:

- Iterations
- Retrospectives
- Ten-minute build
- Continuous integration
- Test-driven development
- Other practices

B) Assess Your Agility

The score of the lowest spoke identifies your risk, as follows:

- 75 points or less: immediate improvement required (red)
- 75 to 96 points: improvement necessary (yellow)
- 97, 98, or 99: improvement possible (green)
- 100: no further improvement needed

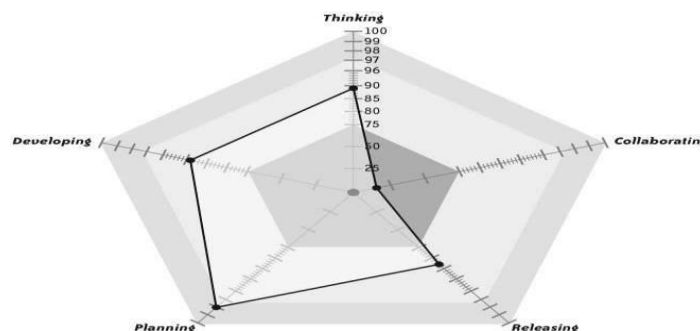


Figure 4-1. Example assessment

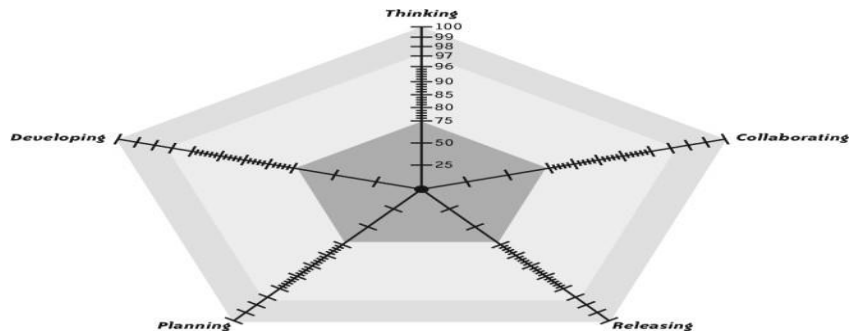


Figure 4-2. Self-assessment chart

5) Explain about Thinking?

What's wrong with this sentence?

What we really need is more keyboards cranking out code.

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

- Pair programming doubles the brainpower available during coding, and gives one person in each pair the opportunity to think about strategic, long-term issues.
- Energized work acknowledges that developers do their best, most productive work when they're energized and motivated.
- An informative workspace gives the whole team more opportunities to notice what's working well and what isn't.
- Root-cause analysis is a useful tool for identifying the underlying causes of your problems.

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

You will need multiple copies of this book (if you don't have enough copies on hand, you can make photocopies of specific practices for the purpose of this exercise), paper, and writing implements.

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: (Timebox this step to 15 minutes.) Within your pair, pick one practice from If you aren't using the practice:

- What about the practice would be easy to do? What would be hard? What sounds ridiculous or silly?
- How does it differ from your previous experiences?
- What would have to be true in order for you to use the practice exactly as written? If you are using the practice:
- What aspects of the practice do you do differently than the book says? (Observations only—no reasons.)
- If you were to follow the practice exactly as written, what would happen?
- What one experimental change could you try that would give you new insight about the practice? (Experiments to prove that the practice is inappropriate are OK.)

Step 3: (Timebox this step to 15 minutes.) Choose three pairs to lead discussions of their answers. Try to pick pairs so that, over time, everyone gets to lead equally. Timebox each presentation to five minutes.

A) 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?

This chapter is called Thinking, yet I included pair programming as the first practice. That's because pair programming is all about increasing your brainpower.

How to Pair

I recommend pair programming on all production code. 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.

Driving and Navigating

When you start pairing, expect to feel clumsy and fumble-fingered as you drive. You may feel that Pairing will feel natural in time. your navigator sees ideas and problems much more quickly than you do. She does—navigators have more time to think than drivers do. The situation will be reversed when you navigate. Pairing will feel natural in time.

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

To enjoy pair programming, good pairing stations are essential. You need plenty of room for both people to sit side by side. Typical cubicles, with a workstation located in a corner, won't work. They're uncomfortable and require one person to sit behind another, adding psychological as well as physical barriers to peer collaboration.

Challenges

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

Comfort

It bears repeating: 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.

Communication style

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

Tools and key bindings

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. Some teams even create a standard image and check it into version control. When you discuss coding standards, discuss these issues as well.

Questions

Isn't it wasteful to have two people do the work of one? In pair programming, two people aren't really doing the work of one. Although only one keyboard is in use, there's more to programming than that. As Ward Cunningham said, "If you don't think carefully, you might think that programming is just typing statements in a programming language."* In pair programming, one person is programming and the other is thinking ahead, anticipating problems, and strategizing.

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.

Contraindications

Pairing requires a comfortable work environment. Most offices and cubicles just aren't set up that way. If your workspace doesn't allow programmers to sit side by side comfortably, either change the workspace or don't pair program. Similarly, if your team doesn't sit together, pairing may not work for you. Although you can pair remotely, it's not as good as in-person.

Alternatives

Pairing is a very powerful tool. It reduces defects, improves design quality, shares knowledge amongst team members, supports self-discipline, and reduces distractions, all without sacrificing productivity. If you cannot pair program, you need alternatives.

Further Reading

Pair Programming Illuminated [Williams] discusses pair programming in depth. "The Costs and Benefits of Pair Programming" [Cockburn & Williams] reports on Laurie Williams' initial study of pair programming.

B) Energized Work

We work at a pace that allows us to do our best, most productive work indefinitely.

I enjoy programming. I enjoy solving problems, writing good code, watching tests pass, and especially removing code while refactoring. I program in my spare time and sometimes even think about work in the shower.

How to Be Energized

One of the simplest ways to be energized is to take care of yourself. Spend Go home on time every day.

time with family and friends and engage in activities that take your mind off of work.

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. Stay home when you're sick. You risk getting other people sick, too. This is particularly true when someone is sick ; in addition to doing poor work, she could infect other people.

Taking Breaks When you make more mistakes than progress, it's time to take a break. If you're like me, that's the hardest time to stop. I feel like the solution is just around the corner—even if it's been just around the corner for the last 45 minutes—and I don't want to stop until I find it. That's why it's helpful for someone else to remind me to stop. After a break or a good night's sleep, I usually see my mistake right away.

Results When your team is energized, there's a sense of excitement and camaraderie. As a group, you pay attention to detail and look for opportunities to improve your work habits. You make consistent progress every week and feel able to maintain that progress indefinitely. You value health over short-term progress and feel productive and successful.

Contraindications Energized work is not an excuse to goof off. Generate trust by putting in a fair day's work. Some organizations may make energized work difficult. If your organization uses the number of hours worked as a yardstick to judge dedication, you may be better off sacrificing energized work and working long hours. The choice between quality of life and career advancement is a personal one that only you and your family can make.

Alternatives

If your organization makes energized work difficult, mistakes are more likely. Pair programming can help tired programmers stay focused and catch each other's errors. Additional testing may be necessary to find the extra defects. If you can, add additional contingency time to your release plan for fixing them.

Further Reading

Peopleware [De Marco & Lister 1999] is a classic work on programmer motivation and productivity. It should be at the top of every software development manager's reading list.

C) Informative Workspace

We are tuned in to the status of our project.

Your workspace is the cockpit of your development effort. Just as a pilot surrounds himself with information necessary to fly a plane, arrange your workspace with information necessary to steer your project: create an informative workspace.

Subtle Cues

The essence of an informative workspace is information. One simple source of information is the feel of the room. A healthy project is energized. There's a buzz in the air—not tension, but activity. People converse, work together, and make the occasional joke. It's not rushed or hurried, but it's clearly productive.

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.

Hand-Drawn Charts

Avoid the reflexive temptation to computerize your charts. The benefits of the informative workspace Don't rush to computerize. stem from the information being constantly visible from everywhere in the room. It's difficult and expensive for computerized charts to meet that criterion; you'd have to install plasma screens or projectors everywhere.

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

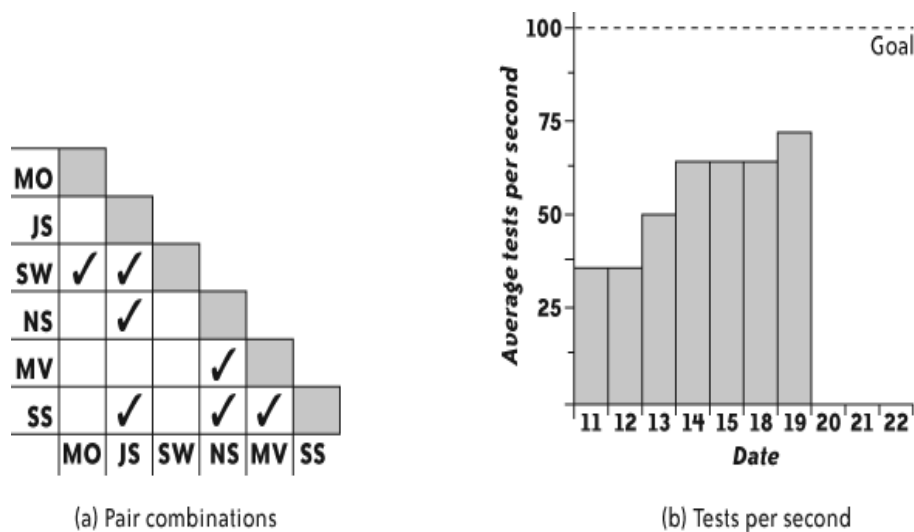


Figure 3: Sample process improvement charts

Gaming

Although having too many process improvement charts can reduce their impact, a bigger problem occurs when the team has too much interest in a chart, that is, in improving a number on a chart. They often start gaming the process. Gaming occurs when people try to improve a number at the expense of overall progress.

Questions 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.

Results

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

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

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

Further Reading Agile Software Development [Cockburn] has an interesting section called "Convection Currents of Information" that describes information as heat and big visible charts as "information radiators."

D) Root-Cause Analysis

We prevent mistakes by fixing our process.

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

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

Problem: When we start working on a new task, we spend a lot of time getting the code into a working state.

Why? Because the build is often broken in source control.

Why? Because people check in code without running their tests. It's easy to stop here and say, "Aha! We found the problem. People need to run their tests before checking in." That is a correct answer, as running tests before check-in is part of continuous integration. But it's also already part of the process. People know they should run the tests, they just aren't doing it. Dig deeper.

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

When Not to Fix the Root Cause When you first start applying root-cause analysis, you'll find many more problems than you can address simultaneously. Work on a few at a time. I like to chip away at the biggest problem while simultaneously picking off low-hanging fruit.

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.

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.

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.

E) Retrospectives

We continually improve our work habits.

No process is perfect. Your team is unique, as are the situations you encounter, and they change all the time. You must continually update your process to match your changing situations. Retrospectives are a great tool for doing so.

Types of Retrospectives

The most common retrospective, the iteration retrospective, occurs at the end of every iteration.

How to Conduct an Iteration Retrospective

Anybody can facilitate an iteration retrospective if the team gets along well. An experienced, neutral facilitator is best to start with. When the retrospectives run smoothly, give other people a chance to try.

I keep the following schedule in mind as I conduct a retrospective. Don't try to match the schedule exactly; let events follow their natural pace:

1. Norm Kerth's Prime Directive
2. Brainstorming (30 minutes)
3. Mute Mapping (10 minutes)
4. Retrospective objective (20 minutes)

Retrospectives are a powerful tool that can actually be damaging when conducted poorly. The process I describe here skips some important safety exercises for the sake of brevity. Pay particular attention to the contraindications before trying this practice.

Step 1: The Prime Directive: In his essay, "The Effective Post-Fire Critique," New York City Fire Department Chief Frank Montagna writes:
Firefighters, as all humans, make mistakes.

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

Step 3: Mute Mapping Mute mapping is a variant of affinity mapping in which no one speaks. It's a great way to categorize a lot of ideas quickly. You need plenty of space for this. Invite everyone to stand up, go over to the whiteboard, and slide cards around. There are three rules:

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

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.

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.

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.

Results

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

Contraindications

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

Alternatives

There are many ways to conduct retrospectives.

Further Reading

Project Retrospectives [Kerth] is the definitive resource for project retrospective.

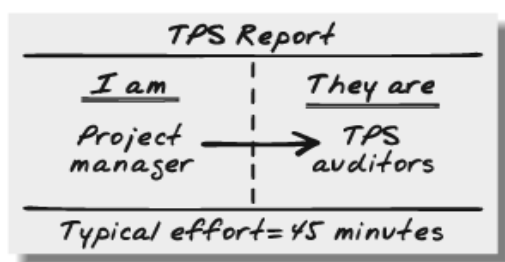
UNIT II COLLABORATING

LONG QUESTION AND ANSWERS

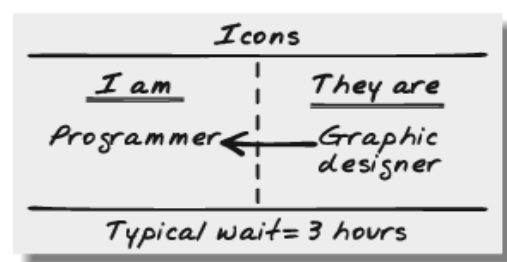
1) EXPLAIN ABOUT COLLABORATING

This chapter contains eight practices to help your team and its stakeholders collaborate efficiently and effectively:

- Trust is essential for the team to thrive.
- Sitting together leads to fast, accurate communication.
- Real customer involvement helps the team understand what to build.
- A ubiquitous language helps team members understand each other.
- Stand-up meetings keep team members informed.
- Coding standards provide a template for seamlessly joining the team's work together.
- Iteration demos keep the team's efforts aligned with stakeholder goals.
- Reporting helps reassure the organization that the team is working well.



(a) Providing information



(b) Requesting information

FIGURE: Sample Cards

2) Explain about 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” [Tuckman]. It takes the team some time to get through each of these stages. They make progress, fall back, bicker, and get along. Over time—often months—and with adequate support and a bit of luck, they get to know each other and work well together. The team jells. Productivity shoots up. They do really amazing work. Here are some strategies for generating trust in your XP team.

Team Strategy #1: Customer-Programmer Empathy Many organizations I’ve worked with have struggled with an “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.

Team Strategy #2: Programmer-Tester Empathy I've also seen "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.

Team Strategy #3: Eat Together

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

Team Strategy #4: Team Continuity

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

Impressions

I know somebody who worked in a company with two project teams. One used XP, met its commitments, and delivered regularly. The team next door struggled: it fell behind schedule and didn't have any working software to show. Yet when the company downsized, it let the XP team members go rather than the other team!

Organizational Strategy #1: Show Some Hustle

"Things may come to those who wait, but only the things left by those who hustle." —Abraham Lincoln†

Organizational Strategy #2: Deliver on Commitments

If your stakeholders have worked with software teams before, they probably have plenty of war wounds from slipped schedules, unfixed defects, and wasted money. In addition, they probably don't know much about software development. That puts them in the uncomfortable position of relying on your work, having had poor results before, and being unable to tell if your work is any better.

Organizational Strategy #3: Manage Problems

Did I say, "All you have to do?" Silly me. It's not that easy. Some problems are too big to absorb no matter how much slack you have. If this is the case, get together as a whole team as soon as possible and replan. You may need to remove an entire story or you might be able to reduce the scope of some stories.

Organizational Strategy #4: Respect Customer Goals

When XP teams first form, it usually takes individual members a while to think of themselves as part of a single team. In the beginning, programmers, testers, and customers often see themselves as separate groups.

Organizational Strategy #5: Promote the Team

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

Organizational Strategy #6: Be Honest

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

THE CHALLENGE OF TRUTH-TELLING

The most challenging project I've ever coached had a tight deadline. (Don't they all?) Our end-customer was a critical customer: a large institution that represented the majority of our income. If we didn't satisfy them, we risked losing a huge chunk of vital business.

Questions: Our team seems to be stuck in the "Storming" stage of team development. How can we advance?

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. The atmosphere in the team room is busy but relaxed, and you genuinely enjoy working with your teammates

Contraindications: Compensation practices can make teamwork difficult. An XP team produces results through group effort. If your organization relies on individual task assignment for personnel evaluation, teamwork may suffer.

Alternatives : Trust is vital for agile projects—perhaps for any project. I'm not sure it's possible to work on an agile project without it.

Further Reading :The Wisdom of Teams [Katzenbach & Smith], which organizational development consultant Diana Larsen describes as "the best book about teams extant."

3) Explain about Sit Together.

We communicate rapidly and accurately. If you've tried to conduct a team meeting via speakerphone, you know how much of a difference face-to-face conversations make. Compared to an in-person discussion, teleconferences are slow and stutter-filled, with uncomfortable gaps in the conversation and people talking over each other.

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.

A Better Way

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

Exploiting Great Communication

Sitting together eliminates the waste caused by waiting for an answer, which dramatically improves productivity. In a field study of six colocated teams, found that sitting together doubled productivity and cut time to market to almost one-third of the company baseline.

Secrets of Sitting Together

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

Making Room

Sitting together is one of those things that's easy to say and hard to do. It's not that the act itself is difficult—the real problem is finding space. A team that sits in adjacent cubicles can convert them into an adequate shared workspace, but even with cubicles, it takes time and money to hire people to rearrange the walls.

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.

Sample Workspaces:

The sample workspace

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.

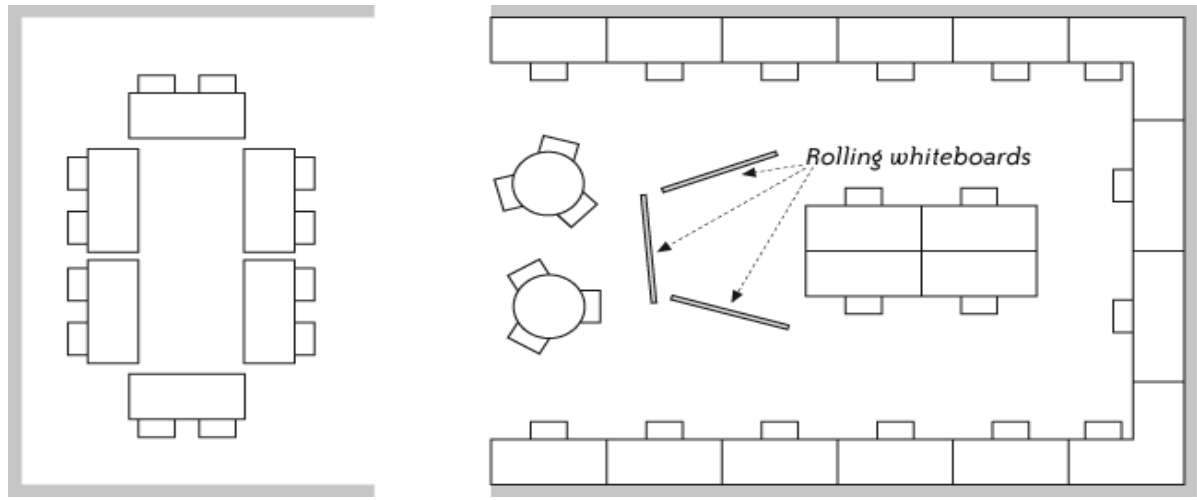


Figure 6-2. A sample workspace

paired with programmers frequently, and there were also extra cubbies for bringing people into the team temporarily.

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.

Adopting an Open Workspace :

Some team members may resist moving to an open workspace. Common concerns include loss of individuality and privacy, implied reduction in status from losing a private office, and managers not recognizing individual contributions. Team members may also mention worries about distractions and noise, but I find that this is usually a cover for one of the other concerns.

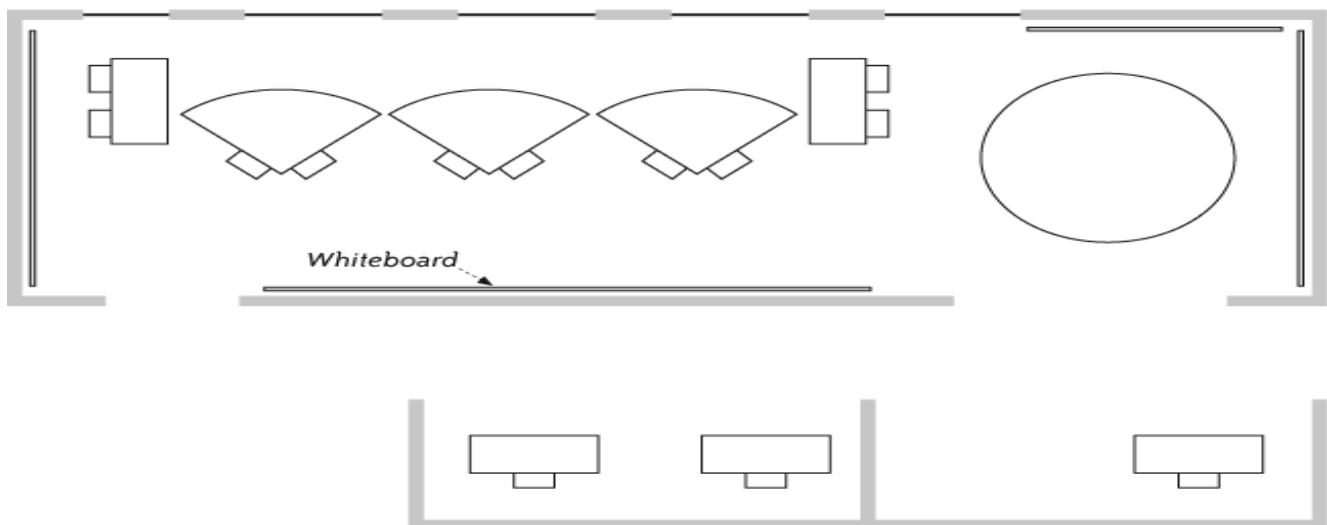


Figure 6-3. A small workspace

However, forcing people to sit together in hopes that they'll come to like it is a bad idea. When I've forced team members to do so, they've invariably found a way to leave the team, even if it meant quitting the

company. Instead, talk with the team about their concerns and the trade-offs of moving to an open workspace.

Questions: How can I concentrate with all that background noise? A team that's working together in a shared workspace produces a busy hum of activity. This can be distracting at first, but most people get used to it in time.

Results: When your team sits together, communication is much more effective. You stop guessing at answers and ask more questions. You overhear other people's conversations and contribute answers you may not expect. Team members spontaneously form cross-functional groups to solve problems. There's a sense of camaraderie and mutual respect.

Contraindications: The hardest part about sitting together is finding room for the open workspace. Cubicles, even adjacent cubicles, won't provide the benefits that an open workspace does. Start working on this problem now as it can take months to resolve.

Further Reading: Agile Software Development [Cockburn] has an excellent chapter on communication. Chapter 3, "Communicating, Cooperating Teams," discusses information radiators, communication quality, and many other concepts related to sitting together.

4) Explain about Real Customer Involvement.

We understand the goals and frustrations of our customers and end-users. An XP team I worked with included a chemist whose previous job involved the software that the team was working to replace. She was an invaluable resource, full of insight about what did and didn't work with the old product. We were lucky to have her as one of our on-site customers—thanks to her, we created a more valuable product.

Personal Development:

In personal development, the development team is its own customer. They're developing the software for their own use. As a result, there's no need to involve external customers—the team is the real customer.

In-House Custom Development:

In-house custom development occurs when your organization asks your team to build something for the organization's own use. This is classic IT development. It may include writing software to streamline operations, automation for the company's factories, or producing reports for accounting.

Outsourced Custom Development:

Outsourced custom development is similar to in-house development, but you may not have the connections that an in-house team does. As a result, you may not be able to recruit real customers to act as the team's on-site customers. Still, you should try. One way to recruit real customers is to move your team to your customer's offices rather than asking them to join you at yours.

Vertical-Market Software:

Unlike custom development, vertical-market software is developed for many organizations. Like custom development, however, it's built for a particular industry and it's often customized for each customer.

Horizontal-Market Software:

Horizontal-market software is the visible tip of the software development iceberg: software that's intended to be used across a wide range of industries. The rows of shrink wrapped software boxes at your local electronics store are a good example of horizontal-market software. So are many web sites.

Questions: Who should we use as on-site customers when we can't include real customers on the team? Your organization should supply a product manager and domain experts.

Results: When you include real customers, you improve your knowledge about how they use the software in practice. You have a better understanding of their goals and frustrations, and you use that knowledge to revise what you produce. You increase your chances of delivering a truly useful and thus successful product.

Contraindications: One danger of involving real customers is that they won't necessarily reflect the needs of all your customers. Be careful that they don't steer you toward creating software that's only useful for them. Your project should remain based on a compelling vision. Customer desires inform the vision and may even change it, but ultimately the product manager holds final responsibility for product direction.

Alternatives: Real customer involvement is helpful but not crucial. Sometimes the best software comes from people who have a strong vision and pursue it vigorously. The resulting software tends to be either completely new or a strong rethinking of existing products.

Ubiquitous Language: We understand each other. Try describing the business logic in your current system to a nonprogrammer domain expert. Are you able to explain how the system works in terms the domain expert understands? Can you avoid programmer jargon, such as the names of design patterns or coding styles? Is your domain expert able to identify potential problems in your business logic?

The Domain Expertise Conundrum: One of the challenges of professional software development is that programmers aren't necessarily experts in the areas for which they write software. For example, I've helped write software that controls factory robots, directs complex financial transactions, and analyzes data from scientific instruments. When I started on these projects, I knew nothing about those things.

5) Explain about Be Brief(Meetings)

The purpose of a stand-up meeting is to give everybody a rough idea of where the team is. It's not to give a complete inventory of everything happening in the project. The primary virtue of the stand-up meeting is brevity. That's why we stand: our tired feet remind us to keep the meeting short.

- A programmer

- The product manager
- A domain expert
- A programmer responds

Questions Can people outside the team attend the stand-up? Yes; I ask that outsiders stand outside the circle and not speak unless they have something brief and relevant to add.

Results

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

Contraindications Don't let the daily stand-up stifle communication. Some teams find themselves waiting for the stand-up rather than going over and talking to someone. Communicate issues as soon as they come up, when they need to. If you find this happening, eliminating the stand-up for a little while may actually improve communication.

Alternatives If you can't conduct a daily stand-up meeting, you need to stay in touch in some other way. If your team sits together, the resulting natural communication may actually be sufficient. Watch for unpleasant surprises that more communication can prevent.

Further Reading "It's Not Just Standing Up: Patterns for Daily Stand-up Meetings" [Yip], at <http://www.martinfowler.com/articles/itsNotJustStandingUp.html>, is a nice collection of patterns for stand-up meetings.

6) Explain about Coding Standards.

We embrace a joint aesthetic. Back in the days of the telegraph, as the story goes, telegraph operators could recognize each other on the basis of how they keyed their dots and dashes. Each operator had a unique style, or fist, that experts could recognize easily. Programmers have style, too. We each have our own way of producing code. We refine our style over years until we think it's the most readable, the most compact, or the most informative it can be.

Beyond Formatting I once led a team of four programmers who had widely differing approaches to formatting. When we discussed coding standards, I catalogued three different approaches to braces and tabs. Each approach had its own vigorous defender. I didn't want us to get bogged down in arguments, so I said that people could use whatever brace style they wanted.

How to Create a Coding Standard Creating a coding standard is an exercise in building consensus. It may be one of the first things that programmers do as a team. Over time, you'll amend and improve the standards. The most important thing you may learn from creating the coding standard is how to disagree constructively. To that end, I recommend applying two guidelines:

3. Create the minimal set of standards you can live with.

4. Focus on consistency and consensus over perfection.

The best way to start your coding standard is often to select an industry-standard style guide for your language. This will take care of formatting questions and allow you to focus on design-related questions. If you're not sure what it should encompass, starting points include:

- Development practices
- Tools, keybindings, and IDE
- File and directory layout
- Build conventions
- Error handling and assertions
- Approach to events and logging
- Design conventions (such as how to deal with null references)

Dealing with Disagreement It's possible to pressure a dissenter into accepting a coding standard she doesn't agree with, but it's probably not a good idea. Doing so is a good way to create resentment and discord. Instead, remember that few decisions are irrevocable in agile development; mistakes are opportunities to learn and improve. Ward Cunningham put it well:*

Adhering to the Standard People make mistakes. Pair programming helps developers catch mistakes and maintain self-discipline. It provides a way to discuss formatting and coding questions not addressed by the guidelines. It's also an excellent way to improve the standard; it's much easier to suggest an improvement when you can talk it over with someone first.

Questions We have legacy code that doesn't fit our standard. Should we fix it?

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

Contraindications Don't allow coding standards to become a divisive issue for your team.

Alternatives : Some teams work together so well that they don't need a written coding standard; their coding standard is implicit.

7) Explain about Iteration Demo.

Iteration Demo We keep it real. An XP team produces working software every week, starting with the very first week. Sound impossible? It's not. It's merely difficult. It takes a lot of discipline to keep that pace. Programmers need discipline to keep the code clean so they can continue to make progress. Customers need discipline to fully understand and communicate one set of features before starting another. Testers need discipline to work on software that changes daily.

The rewards for this hard work are significantly reduced risk, a lot of energy and fun, and the satisfaction of doing great work and seeing progress. The biggest challenge is keeping your momentum. The iteration

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

Second, the demos help the team be honest about its progress. Iteration demos are open to all stakeholders, and some companies even invite external customers to attend. It's harder to succumb to the temptation to push an iteration deadline "just one day" when stakeholders expect a demo. Finally, the demo is an opportunity to solicit regular feedback from the customers. Nothing speaks more clearly to stakeholders than working, usable software. Demonstrating your project makes it and your progress immediately visible and concrete. It gives stakeholders an opportunity to understand what they're getting and to change direction if they need to.

How to Conduct an Iteration Demo

Anybody on the team can conduct the iteration demo, but I recommend that the product manager do so. He has the best understanding of the stakeholders' point of view and speaks their language. His leadership also emphasizes the role of the product manager in steering the product. Invite anybody who's interested. The whole team, key stakeholders, and the executive sponsor should attend as often as possible. Include real customers when appropriate. Other teams working nearby and people who are curious about the XP process are welcome as well. If you can't get everyone in a room, use a teleconference and desktop-sharing software.

Two Key Questions

At the end of the demo, ask your executive sponsor two key questions:*

1. Is our work to date satisfactory?
 2. May we continue?
- These questions help keep the project on track and remind your sponsor to speak up if she's unhappy. You should be communicating well enough with your sponsor that her answers are never a surprise.

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

Questions : What do we do if the stakeholders keep interrupting and asking questions during the demo?

Results When you conduct a weekly iteration demo and demo release, you instill trust in stakeholders, and the team is confident in its ability to deliver. You share problems forthrightly, which allows you to manage them and helps prevent them from ballooning out of control.

Contraindications Because the iteration demo is highly visible, you may be tempted to fake a demo. You might show a user interface that doesn't have any logic behind it, or purposefully avoid showing an action that has a significant defect.

Alternatives The iteration demo is a clear indication of your ability to deliver: either you have the ability to demonstrate new features every week, or you don't. Your executive sponsor either gives you permission to continue, or he doesn't. I'm not aware of any alternatives that provide such valuable feedback.

8) Explain about Reporting.

We inspire trust in the team's decisions. You're part of a whole team. Everybody sits together. An informative workspace clearly tracks your progress. All the information you need is at your fingertips. Why do you need reports?

Types of Reports

Progress reports are exactly that: reports on the progress of the team, such as an iteration demo or a release plan. Although progress reports seem to exist so that stakeholders can monitor and correct the team's direction, that's not their purpose. Instead, good progress reports allow stakeholders to trust the team's decisions.

Management reports are for upper management. They provide high-level information that allows management to analyze trends and set goals. It's not information you can pick up by casually lingering in an open workspace for an hour or two every month; it includes trends in throughput or defect rates.

Progress Reports to Provide XP teams have a pronounced advantage when it comes to reporting progress: they make observable progress every week, which removes the need for guesswork. Furthermore, XP teams create several progress reports as a normal byproduct of their work. Useful and free? There's little not to like about these four reports.

Vision statement Your on-site customers should create and update a vision statement that describes what you're doing, why you're doing it, and how you'll know if you're successful. This provides important context for other reports. Post it prominently and reference it in conversation.

Weekly demo Nothing is as powerful at demonstrating progress as working software. Invite stakeholders to the weekly iteration demo. They probably won't attend every week, but the mere fact that you hold weekly demos will build confidence in your work.

Release and iteration plans

The release and iteration planning boards already posted in your workspace provide great detail about progress. (See Figure 8-4, a release planning board, and Figure 8-9, an iteration planning board.) Invite stakeholders to look at them any time they want detailed status information. For off-site stakeholders, consider using a webcam or regularly posted digital photos to broadcast the plans.

Burn-up chart A

burn-up chart is an excellent way to get a bird's-eye view of the project. It shows progress and predicts a completion date. Most teams produce a burn-up chart when they update their release plan.

Progress Reports to Consider

If your stakeholders want more information, consider providing one or more of the following reports. Avoid providing them by default; each takes time that you could spend on development instead.

Roadmap

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

Status email

A weekly status email can supplement the iteration demo. I like to include a list of the stories completed for each iteration and their value. I also include our current range of probable completion scope and dates, and I explain any changes from the previous report.

Management Reports to Consider

Whereas progress reports demonstrate that the team will meet its goals, management reports demonstrate that the team is working well. As with progress reports, report only what you must.

Productivity

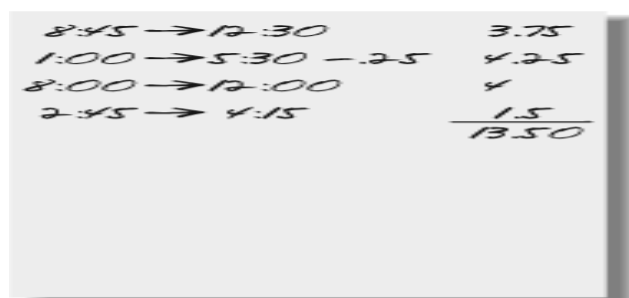
Software development productivity is notoriously difficult to measure [Fowler 2003]. It sounds simple—productivity is the amount of production over time—but in software, we don't have an objective way to measure production. What's the size of a feature?

Throughput

Throughput is the number of features the team can develop in a particular amount of time. To avoid difficult questions such as "What's a feature?," measure the amount of time between the moment the team agrees to develop some idea and the moment that idea is in production and available for general use. The less time, the better.

Defects

Anyone can produce software quickly if it doesn't have to work. Consider counterbalancing your throughput report with defect counts. One of the biggest challenges of counting defects is figuring out the difference between a defect and intentional behavior. Decide who will arbitrate these discussions early so you can avoid arguments.



8:45 → 12:30	3.75
1:00 → 5:30 - .25	4.25
8:00 → 12:00	4
2:45 → 4:15	1.5
	<hr/>
	13.50

Figure : Time tracking example

Time usage

The project manager for collating into these categories:

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

Reports to Avoid :Some reports, although common, don't provide useful information.

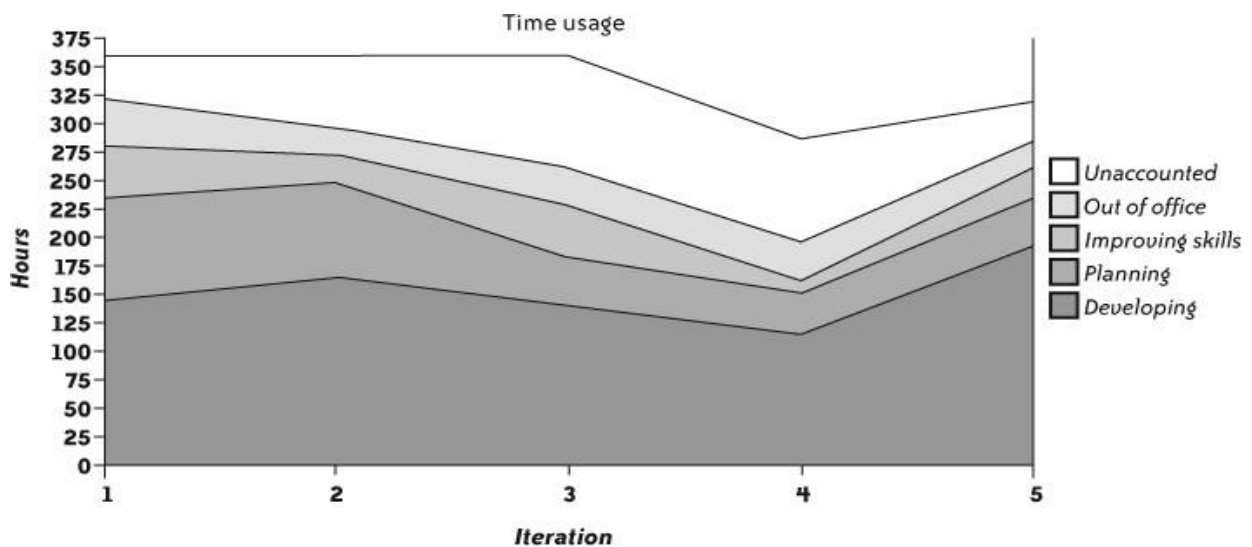


Figure : Time usage report

Source lines of code (SLOC) and function points Source lines of code (SLOC) and its language-independent cousin, function points, are common approaches to measuring software size. Unfortunately, they're also used for measuring productivity. As with a fancy cell phone, however, software's size does not necessarily correlate to features or value.

Number of stories Some people think they can use the number of stories delivered each iteration as a measure of productivity. Don't do that. Stories have nothing to do with productivity. A normal-sized team will typically work on 4 to 10 stories every iteration. To achieve this goal, they combine and split stories as needed. A team doing this job well will deliver a consistent number of stories each iteration regardless of its productivity.

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

Code quality

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

clearly shows design or code quality. The metrics merely recommend areas that deserve further investigation. Avoid reporting code quality metrics. They are a useful tool for developers but they're too ambiguous for reporting to stakeholders.

Questions

What do you mean, "Progress reports are for stakeholder trust"? Shouldn't we also report when we need help with something?

Results

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

Contraindications

Time spent on reports is time not spent developing. Technically speaking, reports are wasteful because they don't contribute to development progress. As a result, I prefer to produce as few reports as possible.

Alternatives

Frequent communication can sometimes take the place of formal reporting. If this option is available to you, it's a better option.

Further Reading

Why Does Software Cost So Much? [DeMarco 1995] contains an essay titled "Mad About Measurement" that discusses challenges of measuring performance, and includes a brief investigation into the results of reporting cyclomatic code complexity.

III UNIT RELEASING

1) EXPLAIN ABOUT RELEASING.

What is the value of code? Agile developers value "working software over comprehensive documentation."* Does that mean a requirements document has no value? Does it mean unfinished code has no value? Like a rock at the top of a hill, code has potential—potential energy for the rock and potential value for the code. It takes a push to realize that potential. The rock has to be pushed onto a slope in order to gain kinetic energy; the software has to be pushed into production in order to gain value.

In order to meet commitments and take advantage of opportunities, you must be able to push your software into production within minutes. This chapter contains 6 practices that give you leverage to turn your big release push into a 10-minute tap:

- "done done" ensures that completed work is ready to release.
- No bugs allows you to release your software without a separate testing phase.
- Version control allows team members to work together without stepping on each other's toes.
- A ten-minute build builds a tested release package in under 10 minutes.
- Continuous integration prevents a long, risky integration phase
- Collective code ownership allows the team to solve problems no matter where they may lie.
- Post-hoc documentation decreases the cost of documentation and increases its accuracy.

Figure . A sample card

Writing Implements For Everyone:(Realsing MINI-ÉTUDE)

Step 1.Start by forming 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. (Timebox this step to 10 minutes.) Within pairs, consider all the activities that have to happen between the time someone has an idea and when you can release it to real users or customers. Count an iteration as one activity, and group together any activities that take less than a day. Consider time spent waiting as an activity, too. If you can't think of anything new, pick an existing card and skip to Step 3.

Step 3. (Time box this step to 10 minutes.) Discuss things that your team can do to reduce the time required for this activity or to eliminate it entirely. Choose just one idea and write it on a green card.

Step 4. (Time box this step to 15 minutes.) As a team, discuss your cards and place them on the table or whiteboard in a value stream map. Place activities (red cards) that must happen first before activities that can happen afterward. If you're using a whiteboard, draw arrows between the cards to make the flow of work more clear. Place green cards underneath red cards.

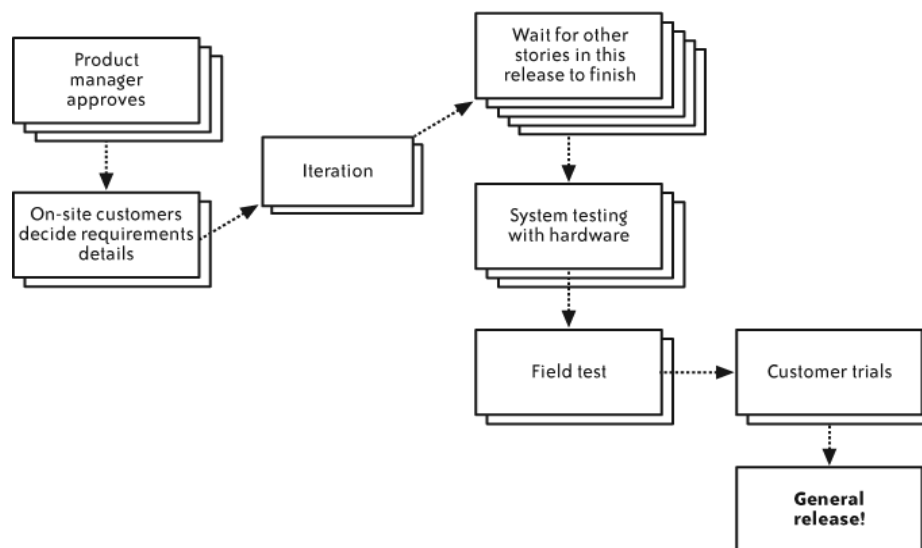


Figure . A sample value stream map

- At which step does work pileup?
- Which results surprise you?
- Who is the constraint in the overall system? How can you improve the performance of the overall system?

- Are there green cards with ideas you can adopt now?

2. Explain about “Done Done”

We’re done when we’re production-ready. Production-Ready Software

Wouldn’t it be nice if, once you finished a story, you never had to come back to it? That’s the idea behind “done done.” A completed story isn’t a lump of un integrated, untested code. It’s ready to deploy.

I write mine on the iteration planning board:

- Tested (all unit, integration, and customer tests finished)
- Coded (all code written)
- Designed (code refactored to the team’s satisfaction)
- Integrated (the story works from end to end—typically, UI to database—and fits into the rest of the software)
- Builds (the build script includes any new modules)
- Installs (the build script includes the story in the automated installer)
- Migrates (the build script updates database schema if necessary; the installer migrates data when appropriate)
- Reviewed (customers have reviewed the story and confirmed that it meets their expectations)
- Fixed (all known bugs have been fixed or scheduled as their own stories)
- Accepted (customers agree that the story is finished)

How to Be “Done Done”

XP works best when you make a little progress on every aspect of your work every day, rather than reserving the last few days of your iteration for Make a little progress on every aspect of your work every day. getting stories “done done.” This is an easier way to work, once you get used to it, and it reduces the risk of finding unfinished work at the end of the iteration.

Making Time

This may seem like an impossibly large amount of work to do in just one week. It’s easier to do if you work on it throughout the iteration rather than saving it up for the last day or two. The real secret, though, is to make your stories small enough that you can completely finish them all in a single week.

Questions How does testers’ work fit into “done done”?

Results When your stories are “done done,” you avoid unexpected batches of work and spread wrap-up and polish work throughout the iteration. Customers and testers have a steady workload through the entire iteration. The final customer acceptance demonstration takes only a few minutes. At the end of each iteration, your software is ready to demonstrate to stakeholders with the scheduled stories working to their satisfaction.

Contraindications This practice may seem advanced. It’s not, but it does require self-discipline. To be “done done” every week, you must also work in iterations and use small, customer-centric stories.

Alternatives This practice is the cornerstone of XP planning. If you aren’t “done done” at every iteration, your velocity will be unreliable. won’t be able to ship at any time. This will disrupt your release planning and prevent you from meeting your commitments, which will in turn damage stakeholder trust.

3. Explain about No Bugs(Bug Free Release)

We confidently release without a dedicated testing phase. Let’s cook up a bug pie. First, start with a nice,

challenging language. How about C? We'll season it with a dash of assembly. Next, add extra bugs by mixing in concurrent programming. Our old friends Safety and Liveness are happy to fight each other over who provides the most bugs. They supplied the Java multithreading library with bugs for years!

How Is This Possible?

If you're on a team with a bug count in the hundreds, the idea of "no bugs" probably sounds ridiculous. I'll admit: "no bugs" is an ideal to strive for, not something your team will necessarily achieve. However, XP teams can achieve dramatically lower bug rates. [Van Schoenderwoert]'s team averaged one and a half bugs per month in a very difficult domain. In an independent analysis of a company practicing a variant of XP, QSM Associates reported an average reduction from 2,270 defects to 381 defects [Mah].

How to Achieve Nearly Zero Bugs

Many approaches to improving software quality revolve around finding and removing more defects through traditional testing, inspection, and automated analysis. The agile approach is to generate fewer defects. This isn't a matter of finding defects earlier; it's a question of not generating them at all.

To achieve these results, XP uses a potent cocktail of techniques:

1. Write fewer bugs by using a wide variety of technical and organizational practices.
2. Eliminate bug breeding grounds by refactoring poorly designed code.
3. Fix bugs quickly to reduce their impact, write tests to prevent them from reoccurring, then fix the associated design flaws that are likely to breed more bugs.
4. Test your process by using exploratory testing to expose systemic problems and hidden assumptions.
5. Fix your process by uncovering categories of mistakes and making those mistakes impossible.

Ingredient #1: Write Fewer Bugs Don't worry—I'm not going to wave my hands and say, "Too many bugs? No problem! Just write fewer bugs!" To stop writing bugs, you have to take a rigorous, thoughtful approach to software development.

Ingredient #2: Eliminate Bug Breeding Grounds _Writing fewer bugs is an important first step to reducing the number of defects your team generates. If you accomplish that-much, you're well ahead of most teams. Don't stop now, though. You can generate even fewer defects.

Ingredient #3: Fix Bugs Now- Programmers know that the longer you wait to fix a bug, the more it costs to fix [McConnell 1996] (p. 75). In addition, unfixed bugs probably indicate further problems. Each bug is the Allies Simple result of a flaw in your system that's likely to breed more mistakes. Fix it now and you'll improve both quality and productivity.

Ingredient #4: Test Your Process These practices will dramatically cut down the number of bugs in your system. However, they only prevent bugs you expect. Before you can write a test to prevent a problem, you have to realize the problem can occur.

Ingredient #5: Fix Your Process Some bugs occur because we're human. (D'oh!) More often, bugs indicate an unknown flaw in our approach.

Questions:

If novices can do this, what's stopping management from firing everybody and hiring novices?

How do we prevent security defects and other challenging bugs?

If our guideline is to fix bugs as soon as we find them, won't we have the unconscious temptation to overlook bugs?

Results:

When you produce nearly zero bugs, you are confident in the quality of your software. You're comfortable releasing your software to production without further testing at the end of any iteration. Stakeholders, customers, and users rarely encounter unpleasant surprises, and you spend your time producing great software instead of fighting fires.

Contraindications:

"No Bugs" depends on the support and structure of all of XP. To achieve these results, you need to practice nearly all of the XP practices rigorously.

Alternatives:

You can also reduce bugs by using more and higher quality testing (including inspection or automated analysis) to find and fix a higher percentage of bugs. However, testers will need some time to review and test your code, which will prevent you from being "done done" and ready to ship at the end of each iteration.

Further Reading:

"Embedded Agile Project by the Numbers with Newbies" [Van Schooenderwoert] expands on the embedded C project described in the introduction.

4. Explain about Version Control

We keep all our project artifacts in a single, authoritative place.

To work as a team, you need some way to coordinate your source code, tests, and other important project artifacts. A version control system provides a central repository that helps coordinate changes to files and also provides a history of changes.

Different version control systems use different terminology.

Here are the terms I use throughout this book:

- **Repository** The repository is the master storage for all your files and their history. It's typically stored on the version control server. Each standalone project should have its own repository.
- **Sandbox:** Also known as a working copy, a sandbox is what team members work out of on their local development machines. (Don't ever put a sandbox on a shared drive. If other people want to develop, they can make their own sandbox.) The sandbox contains a copy of all the files in the repository from a particular point in time.
- **Check out:** To create a sandbox, check out a copy of the repository. In some version control systems, this term means "update and lock."
- **Update:** Update your sandbox to get the latest changes from the repository. You can also update to a particular point in the past.
- **Lock:** A lock prevents anybody from editing a file but you.
- **Check in or commit:** Check in the files in your sandbox to save them into the repository.
- **Revert:** Revert your sandbox to throw away your changes and return to the point of your last update. This is handy when you've broken your local build and can't figure out how to get it working again. Sometimes reverting is faster than debugging, especially if you have checked in recently.

- **Tip or head:**The tip or head of the repository contains the latest changes that have been checked in. When you update your sandbox, you get the files at the tip. (This changes somewhat when you use branches.)
- **Tag or label:**A tag or label marks a particular time in the history of the repository, allowing you to easily access it again.
- **Roll back:** Roll back a check-in to remove it from the tip of the repository. The mechanism for doing so varies depending on the version control system you use. **Branch::**A branch occurs when you split the repository into distinct “alternate histories,” a process known as branching. All the files exist in each branch, and you can edit files in one branch independently of all other branches.
- **Merge** A merge is the process of combining multiple changes and resolving any conflicts. If two programmers change a file separately and both check it in, the second programmer will need to merge in the first person’s changes.

Concurrent Editing: If multiple developers modify the same file without using version control, they’re likely to accidentally overwrite each other’s changes. To avoid this pain, some developers turn to a locking model of version control: when they work on a file, they lock it to prevent anyone else from making changes. The files in their sandboxes are read-only until locked. If you have to check out a file in order to work on it, then you’re using a locking model.

Time Travel

One of the most powerful uses of a version control system is the ability to go back in time. You can update your sandbox with all the files from a particular point in the past.

Whole Project

It should be obvious that you should store your source code in version control. It’s less obvious that you should store everything else in there, too. Although most version control systems allow you to go back in time, it doesn’t do you any good unless you can build the exact version you had at that time.

Customers and Version Control

Customer data should go in the repository, too.

Keep It Clean

One of the most important ideas in XP is that you keep the code clean and ready to ship. It starts with your sandbox. Although you have to break the build in your sandbox in order to make progress, confine it to your sandbox. Never check in code that breaks the build. This allows anybody to update at any time without worrying about breaking their build—and that, in turn, allows everyone to work smoothly and share changes easily.

Single Codebase

One of the most devastating mistakes a team can make is to duplicate their codebase. It’s easy to do. First, a customer innocently requests a customized version of your software. To deliver this version quickly, it seems simple to duplicate the codebase, make the changes, and ship it. Yet that copy and paste customization doubles the number of lines of code that you need to maintain.

Appropriate Uses of Branches

Branches work best when they are short-lived or when you use them for small numbers of changes. If you support old versions of your software, a branch for each version is the best place to put bug fixes for those versions.

Questions

- Which version control system should I use?
- Should we really keep all our tools and libraries in version control?

- How can we store our database in version control?
- How much of our core platform should we include in version control?
- With so many things in version control, how can I update as quickly as I need to?

Results

With good version control practices, you are easily able to coordinate changes with other members of the team. You easily reproduce old versions of your software when you need to. Long after your project has finished, your organization can recover your code and rebuild it when they need to.

Contraindications

You should always use some form of version control, even on small one-person projects. Version control will act as a backup and protect you when you make sweeping changes. Concurrent editing, on the other hand, can be dangerous if an automatic merge fails and goes undetected. Be sure you have a decent build if you allow concurrent edits. Concurrent editing is also safer and easier if you practice continuous integration and have good tests.

Alternatives

There is no practical alternative to version control.

You may choose to use file locking rather than concurrent editing. Unfortunately, this approach makes refactoring and collective code ownership very difficult, if not impossible. You can alleviate this somewhat by keeping a list of proposed refactorings and scheduling them, but the added overhead is likely to discourage people from suggesting significant refactorings.

Further Reading

[Mason] is a good introduction to the nuts and bolts of version control that specifically focuses on Subversion.

[Sink], at http://www.ericssink.com/scm/source_control.html, is a helpful introduction to version control for programmers with a Microsoft background.

[Berczuk & Appleton] goes into much more detail about the ways in which to use version control.

5. Explain About Fast Build

Ten-Minute Build

We eliminate build and configuration hassles. Here's an ideal to strive for. Imagine you've just hired a new programmer. On the programmer's first day, you walk him over to the shiny new computer you just added to your open workspace. "We've found that keeping everything in version control and having a really great automated build makes us a lot faster," you say. "Here, I'll show you. This computer is new, so it doesn't have any of our stuff on it yet."

Automate Your Build What if you could build and test your entire product—or create a deployment package—at any time, just by pushing a button? How much easier would that make your life?

Producing a build is often a frustrating and lengthy experience. This frustration can spill over to the rest of your work. "Can we release the software?" "With a few days of work." "Does the software work?" "My piece does, but I can't build everything." "Is the demo ready?" "We ran into a problem with the build—tell everyone to come back in an hour."

How to Automate

Automating your build is one of the easiest ways to improve morale and increase productivity. There are

plenty of useful build tools available, depending on your platform and choice of language. If you're using Java, take a look at Ant. In .NET, NAnt and MSBuild are popular. Make is the old standby for C and C++. Perl, Python, and Ruby each have their preferred build tools as well.

When to Automate

At the start of the project, in the very first iteration, set up a bare-bones build system. The goal of this first iteration is to produce the simplest possible product that exercises your entire system. That includes delivering a working—if minimal—product to stakeholders.

Automating Legacy Projects

If you want to add a build script to an existing system, I have good news and bad news. The good news is that creating a comprehensive build script is one of the easiest ways to improve your life. The bad news is that you probably have a bunch of technical debt to pay off, so it won't happen overnight.

Ten Minutes or Less

A great build script puts your team way ahead of most software teams. After you get over the rush of being able to build the whole system at any time you want, you'll probably notice something new: the build is slow. With continuous integration, you integrate every few hours. Each integration involves two builds: one on your machine and one on the integration machine. You need to wait for both builds to finish before continuing because you can never let the build break in XP. If the build breaks, you have to roll back your changes. A 10-minute build leads to a 20-minute integration cycle. That's a pretty long delay. I prefer a 10- or 15-minute integration cycle.

Questions

- Who's responsible for maintaining the build script?
- Should we really keep all our tools and libraries in version control?
- We use an IDE with an integrated build system. How can we automate our build process?
- We have different target and development environments. How do we make this build work?
- How can we build our entire product when we rely on third-party software and hardware?
- How often should we build from scratch?

Results

With a good automated build, you can build a release any time you want. When somebody new joins the team, or when you need to wipe a workstation and start fresh, it's a simple matter of downloading the latest code from the repository and running the build.

When your build is fast and well-automated, you build and test the whole system more frequently. You catch bugs earlier and, as a result, spend less time debugging. You integrate your software frequently without relying on complex background build systems, which reduces integration problems.

Contraindications

Every project should have a good automated build. Even if you have a system that's difficult to build, you can start chipping away at the problem today.

Some projects are too large for the 10-minute rule to be effective. Before you assume this is true for your project, take a close look at your build procedures. You can often reduce the build time much more than you realize.

Alternatives

If the project truly is too large to build in 10 minutes, it's probably under development by multiple teams or subteams. Consider splitting the project into independent pieces that you can build and test separately.

If you can't build your system in less than 10 minutes (yet), establish a maximum acceptable

threshold and stick to it. Drawing this line helps identify a point beyond which you will not allow more technical debt to accumulate.

6. Explain About Continuous Integration

Continuous integration is a better approach. It keeps everybody's code integrated and builds release infrastructure along with the rest of the application. The ultimate goal of continuous integration is to be able to deploy all but the last few hours of work at any time.

Why It Works

If you've ever experienced a painful multiday (or multiweek) integration, integrating every few hours probably seems foolish. Why go through that hell so often?

Actually, short cycles make integration less painful. Shorter cycles lead to smaller changes, which means there are fewer chances for your changes to overlap with someone else's. That's not to say collisions don't happen. They do. They're just not very frequent because everybody's changes are so small.

How to Practice Continuous Integration

In order to be ready to deploy all but the last few hours of work, your team needs to do **two** things:

1. Integrate your code every few hours.
2. Keep your build, tests, and other release infrastructure up-to-date.

To integrate, update your sandbox with the latest code from the repository, make sure everything builds, then commit your code back to the repository. You can integrate any time you have a successful build. With test-driven development, that should happen every few minutes. I integrate whenever I make a significant change to the code or create something I think the rest of the team will want right away.

Never Break the Build

When was the last time you spent hours chasing down a bug in your code, only to find that it was a problem with your computer's configuration or in somebody else's code? Conversely, when was the last time you spent hours blaming your computer's configuration (or somebody else's code), only to find that the problem was in code you just wrote? On typical projects, when we integrate, we don't have confidence in the quality of our code or in the quality of the code in the repository. The scope of possible errors is wide; if anything goes wrong, we're not sure where to look.

The Continuous Integration Script

To guarantee an always-working build, you have to solve two problems. First, you need to make sure that what works on your computer will work on anybody's computer. (How often have you heard the phrase, "But it worked on my machine!"?) Second, you need to make sure nobody gets code that hasn't been proven to build successfully.

To update from the repository

1. Check that the integration token is available. If it isn't, another pair is checking in unproven code and you need to wait until they finish.
2. Get the latest changes from the repository. Others can get changes at the same time, but don't let anybody take the integration token until you finish.

To integrate

1. Update from the repository (follow the previous script). Resolve any integration conflicts and run the build (including tests) to prove that the update worked.
2. Get the integration token and check in your code.
3. Go over to the integration machine, get the changes, and run the build (including tests).
4. Replace the integration token.

CONTINUOUS INTEGRATION SERVERS There's a lively community of open-source continuous integration servers (also called CI servers). The granddaddy of them all is CruiseControl, pioneered by

ThoughtWorks employees.

Introducing Continuous Integration

The most important part of adopting continuous integration is getting people to agree to integrate frequently (every few hours) and never to break the build. Agreement is the key to adopting continuous integration because there's no way to force people not to break the build.

Dealing with Slow Builds

The most common problem facing teams practicing continuous integration is slow builds. Whenever possible, keep your build under 10 minutes. On new projects, you should be able to keep your build under 10 minutes all the time. On a legacy project, you may not achieve that goal right away. You can still practice continuous integration, but it comes at a cost.

Multistage Integration Builds

Some teams have sophisticated tests, measuring such qualities as performance, load, or stability, that simply cannot finish in under 10 minutes. For these teams, multistage integration is a good idea. A multistage integration consists of two separate builds. The normal 10-minute build, or commit build, contains all the normal items necessary to prove that the software works: unit tests, integration tests, and a handful of end-to-end. This build runs synchronously as usual.

Questions

I know we're supposed to integrate at least every four hours, but what if our current story or task takes longer than that?

What should we do while we're waiting for the integration build to complete?

Isn't asynchronous integration more efficient than synchronous integration?

Are you saying that asynchronous integration will never work?

Our version control system doesn't allow us to roll back quickly. What should we do?

Why do we need an integration machine? Can't we just integrate locally and check in?

Results

When you integrate continuously, releases are a painless event. Your team experiences fewer integration conflicts and confusing integration bugs. The on-site customers see progress in the form of working code as the iteration progresses.

Contraindications

Don't try to force continuous integration on a group that hasn't agreed to it. This practice takes everyone's willful cooperation. Using continuous integration without a version control system and a 10-minute build is *painful*. Synchronous integration becomes frustrating if the build is longer than 10 minutes and too wasteful if the build is very slow. My threshold is 20 minutes. The best solution is to speed up the build.

Integration tokens don't work at all for very large teams; people spend too much time waiting to integrate. Use private branches in your version control system instead. Check your code into a private branch, build the branch on an integration machine—you can have several—then promote the branch to the mainline if the build succeeds.

Alternatives

If you can't perform synchronous continuous integration, try using a CI server and asynchronous integration. This will likely lead to more problems than synchronous integration, but it's the best of the alternatives.

If you don't have an automated build, you won't be able to practice asynchronous integration. Delaying integration is a very high-risk activity. Instead, create an automated build as soon as possible, and start practicing one of the forms of continuous integration.

Some teams perform a daily build and smoke test. Continuous integration is a more advanced version of the same practice; if you have a daily build and smoke test, you can migrate to continuous integration. Start with asynchronous integration and steadily improve your build and tests until you can use synchronous integration.

7. Explain about Collective Code Ownership

We are all responsible for high-quality code. There's a metric for the risk imposed by concentrating knowledge in just a few people's heads—it's called the truck number. How many people can get hit by a truck before the project suffers irreparable harm? It's a grim thought, but it addresses a real risk. What happens when a critical person goes on holiday, stays home with a sick child, takes a new job, or suddenly retires? How much time will you spend training a replacement? Collective code ownership spreads responsibility for maintaining the code to all the programmers. Collective code ownership is exactly what it sounds like: everyone shares responsibility for the quality of the code. No single person claims ownership over any part of the system, and anyone can make any necessary changes anywhere.

Making Collective Ownership Work

Collective code ownership requires letting go of a little bit of ego. Rather than taking pride in your code, take pride in your team's code. Rather than complaining when someone edits your code, enjoy how the code improves when you're not working on it. Rather than pushing your personal design vision, discuss design possibilities with the other programmers and agree on a shared solution.

Working with Unfamiliar Code

If you're working on a project that has knowledge silos—in other words, little pockets of code that only one or two people understand—then collective code ownership might seem daunting. How can you take ownership of code that you don't understand? To begin, take advantage of pair programming. When somebody picks a task involving code you don't understand, volunteer to pair with him. When you work on a task, ask the local expert to pair with you. Similarly, if you need to work on some unfamiliar code, take advantage of your shared workspace to ask a question or two.

Hidden Benefits

"Of course nobody can understand it... it's job security!" —Old programmer joke It's not easy to let a great piece of code out of your hands. It can be difficult to subsume the desire to take credit for a particularly clever or elegant solution, but it's necessary so your team can take advantage of all the benefits of collaboration. It's also good for you as a programmer. Why? The whole codebase is yours—not just to modify, but to support and improve. You get to expand your skills. Even if you're an absolute database guru, you don't have to write only database code throughout the project.

Questions

We have a really good UI designer/database programmer/scalability guru. Why not take advantage of those skills and specialties?

How can everyone learn the entire codebase?

Doesn't collective ownership increase the possibility of merge conflicts?

We have some pretty junior programmers, and I don't trust them with my code. What should we do?

Results

When you practice collective code ownership, you constantly make minor improvements to all parts of the codebase, and you find that the code you've written improves without

your help. When a team member leaves or takes a vacation, the rest of the team continues to be productive.

Contraindications

Don't use collective code ownership as an excuse for *no* code ownership. Managers have a saying: "Shared responsibility is no responsibility at all." Don't let that happen to your code. Collective code ownership doesn't mean someone else is responsible for the code; it means *you* are responsible for the code—all of it. Collective code ownership requires good communication. Without it, the team cannot maintain a shared vision, and code quality will suffer. Several XP practices help provide this communication: a team that includes experienced designers, sitting together, and pair programming.

Alternatives

A typical alternative to collective code ownership is *strong code ownership*, in which each module has a specific owner and only that person may make changes. A variant is *weak code ownership*, in which one person owns a module but others can make changes as long as they coordinate with the owner. Neither approach, however, shares knowledge nor does enables refactoring as well as collective ownership.

8. Explain about Documentation

We communicate necessary information effectively.

The word documentation is full of meaning. It can mean written instructions for end-users, or detailed specifications, or an explanation of APIs and their use. Still, these are all forms of communication—that's the commonality. Communication happens all the time in a project. Sometimes it helps you get your work done; you ask a specific question, get a specific answer, and use that to solve a specific problem. This is the purpose of work-in-progress documentation, such as requirements documents and design documents. Other communication provides business value, as with product documentation, such as user manuals and API documentation. A third type—handoff documentation—supports the long-term viability of the project by ensuring that important information is communicated to future workers.

Work-In-Progress Documentation

In XP, the whole team sits together to promote the first type of communication. Close contact with domain experts and the use of ubiquitous language create a powerful oral tradition that transmits information when necessary. There's no substitute for face-to-face communication. Even a phone call loses important nuances in conversation.

Product Documentation

Some projects need to produce specific kinds of documentation to provide business value. Examples include user manuals, comprehensive API reference documentation, and reports. One team I worked with created code coverage metrics—not because they needed them, but because senior management wanted the report to see if XP would increase the amount of unit testing.

Handoff Documentation

If you're setting the code aside or preparing to hand off the project to another team (perhaps as part of final delivery), create a small set of documents recording big decisions and information. Your goal is to summarize the most important information you've learned while creating the software the kind of information necessary to sustain and maintain the project.

Questions

Isn't it a risk to reduce the amount of documentation?

Results

When you communicate in the appropriate ways, you spread necessary information effectively. You reduce the amount of overhead in communication. You mitigate risk by presenting only necessary information.

Contraindications

Alistair Cockburn describes a variant of Extreme Programming called “Pretty Adventuresome Programming”:^{*}

A PrettyAdventuresomeProgrammer says:

“Wow! That ExtremeProgramming stuff is neat! We almost do it, too! Let’s see...

“Extreme Programming requires:

- You do pair programming.
- You deliver an increment every three[†] weeks.
- You have a user on the team full time.
- You have regression unit tests that pass 100% of the time.

Alternatives

If you think of documents as *communication mechanisms* rather than simply printed paper, you’ll see that there are a wide variety of alternatives for documentation. Different media have different strengths.

UNIT-4 PLANNING

Long Question and answers

1. Explain the concept the VISION?

Vision

Vision. If there’s a more derided word in the corporate vocabulary, I don’t know what it is. This word brings to mind bland corporate-speak: “Our vision is to serve customers while maximizing stakeholder value and upholding the family values of our employees.” Bleh. Content-free baloney.

Product Vision

Before a project is a project, someone in the company has an idea. Suppose it’s someone in the Wizzle-Frobitz company.^{*} “Hey!” he says, sitting bolt upright in bed. “We could frobitz the wizzles so much better if we had some software that sorted the wizzles first!”

Maybe it’s not quite that dramatic. The point is, projects start out as ideas focused on results. Sell more hardware by bundling better software. Attract bigger customers by scaling more effectively. Open up a new market by offering a new service. The idea is so compelling that it gets funding, and the project begins.

Where Visions Come From

Sometimes the vision for a project strikes as a single, compelling idea. One person gets a bright idea, evangelizes it, and gets approval to pursue it. This person is a *visionary*.

More often, the vision isn’t so clear. There are multiple visionaries, each with their own unique idea of what the project should deliver.

Either way, the project needs a single vision. Someone must unify, communicate, and promote the vision to the team and to stakeholders. That someone is the product

manager.

Identifying the Vision

Like the children's game of telephone, every step between the visionaries and the product manager reduces the product manager's ability to accurately maintain and effectively promote the vision.

If you only have one visionary, the best approach is to have that visionary act as product manager. This reduces the possibility of any telephone-game confusion. As long as the vision is both worthwhile and achievable, the

visionary's day-to-day involvement as product manager greatly improves the project's chances of delivering an impressive product.

If the visionary isn't available to participate fully, as is often the case, someone else must be the product manager. Frequently, a project will have multiple visionaries. This is particularly common in custom software development. If this is the case on your project, you need to help the visionaries combine their ideas into a single, cohesive vision.

Documenting the Vision

After you've worked with visionaries to create a cohesive vision, document it in a vision statement. It's best to do this collaboratively, as doing so will reveal areas of disagreement and confusion. Without a vision statement, it's all too easy to gloss over disagreements and end up with an unsatisfactory product.

Once created, the vision statement will help you maintain and promote the vision. It will act as a vehicle for discussions about the vision and a touchpoint to remind stakeholders why the project is valuable.

Don't forget that the vision statement should be a *living* document: the product manager should review it on a regular basis and make improvements. However, as a fundamental statement of the project's purpose, it may not change much.

How to Create a Vision Statement

The vision statement documents three things: *what* the project should accomplish, *why* it is valuable, and the project's *success criteria*.

The vision statement can be short. I limit mine to a single page. Remember, the vision statement is a clear and simple way of describing why the project deserves to exist. It's not a roadmap; that's the purpose of release planning.

In the first section—*what* the project should accomplish—describe the problem or opportunity that the project will address, expressed as an end result. Be specific, but not prescriptive. Leave room for the team to work out the details.

Here is a real vision statement describing "Sasquatch," a product developed by two entrepreneurs who started a new company:

Sasquatch helps teams collaborate over long distance. It enables the high- quality team dynamics that occur when teams gather around a table and use index cards to brainstorm, prioritize, and reflect.

Sasquatch's focus is *collaboration* and *simplicity*. It is not a project management tool, a tracking tool, or a retrospectives tool. Instead, it is a free-form sandbox that can fulfill any of these purposes. Sasquatch assumes that participants are well-meaning and create their own rules. It does not incorporate mechanisms to enforce particular behaviors among participants.

Sasquatch exudes *quality*. Customers find it a joy to use, although they would be hard-pressed to say why. It is full of small touches that make the experience more enjoyable.

Collaboration, simplicity, and quality take precedence over breadth of features. Sasquatch development focuses on polishing existing features to a high gloss before adding new ones.

In the second section, describe *why* the project is valuable:

Sasquatch is valuable to our customers because long-distance collaboration is so difficult without it. Even with desktop sharing tools, one person becomes the bottleneck for all discussion. Teams used to the power of gathering around a table chafe at these restrictions. Sasquatch gives everyone an opportunity to participate. It makes long-distance collaboration effective and even enjoyable.

Sasquatch is valuable to us because it gives us an opportunity to create an entrepreneurial product. Sasquatch's success will allow us to form our own product company and make a good living doing work that we love.

In the final section, describe the project's *success criteria*: how you will know that the project has succeeded and when you will decide. Choose concrete, clear, and unambiguous targets:

We will deploy Sasquatch in multiple stages with increasing measures of success at each stage.

In the first stage, we will demonstrate a proof-of-concept at a major Agile event. We will be successful if we attract a positive response from agile experts.

In the second stage, we will make a Sasquatch beta available for free use.

We will be successful if at least 10 teams use it on a regular basis for real- world projects within 6 months.

In the third stage, we will convert Sasquatch to a pay service. We will be successful if it grosses at least \$1,000 in the first three months.

In the fourth stage, we will rely on Sasquatch for our income. We will be successful if it meets our minimum monthly income requirements within one year of accepting payment.

Promoting the Vision

After creating the vision statement, post it prominently as part of the team's informative workspace. Use the vision to evangelize the project to stakeholders and to explain the priority (or deprioritization) of specific stories.

Be sure to include the visionaries in product decisions. Invite them to release planning sessions. Make sure they see iteration demos, even if that means a private showing. Involve them in discussions with real customers. Solicit their feedback about progress, ask for their help in improving the plan, and give them opportunities to write stories. They can even be an invaluable resource in company politics, as successful visionaries are often senior and influential.

Including your visionaries may be difficult, but make the effort; distance between the team and its visionaries decreases the team's understanding of the product it's building. While the vision statement is necessary and valuable, a visionary's personal passion and excitement for the product communicates far more clearly. If the team interacts with the visionary frequently, they'll understand the product's purpose better and they'll come up with more ideas for increasing value and decreasing cost.

2. Describe about the RELEASE PLANNING in Planning.

Imagine you've been freed from the shackles of deadlines. "Maximize our return on investment," your boss says. "We've already talked about the vision for this project. I'm counting on you to work out the details. Create your own plans and set your own release dates just make sure we get a good return on our investment."

How to Release Frequently

Releasing frequently doesn't mean setting aggressive deadlines. In fact, aggressive deadlines *extend* schedules rather than reducing them. Instead, release more often by including less in each release. Minimum marketable features are an excellent tool for doing so.

A minimum marketable feature, or MMF, is the smallest set of functionality that provides value to your market, whether that market is internal users (as with custom software) or external customers (as with commercial software). MMFs provide value in many ways, such as competitive differentiation, revenue generation, and cost savings.

As you create your release plan, think in terms of stakeholder value. Sometimes it's helpful to think of stories and how they make up a single MMF. Other times, you may think of MMFs that you can later decompose into stories. Don't forget the *minimum* part of minimum marketable feature try to make each feature as small as possible.

Once you have minimal features, group them into possible releases. This is a brainstorming exercise, not your final plan, so try a variety of groupings. Think of ways to minimize the number of features needed in each release.

The most difficult part of this exercise is figuring out how to make small releases. It's one thing for a *feature* to be marketable, and another for a whole *release* to be marketable. This is particularly difficult when you're launching a new product. To succeed, focus on what sets your product apart, not the features it needs to match the competition.

An Example

Imagine you're the product manager for a team that's creating a new word processor. The market for word processors is quite mature, so it might seem impossible to create a small first release. There's so much to do just to *match* the competition, let alone to provide something new and compelling. You need basic formatting, spellchecking, grammar checking, tables, images; printing... the list goes on forever.

Approaching a word processor project in this way is daunting to the point where it may seem like a worthless effort. Rather than trying to match the competition, focus on the features that make your word processor unique. Release those features first they probably have the most value.

Suppose that the competitive differentiation for your word processor is its

powerful collaboration capabilities and web-based hosting. The first release might have four features: basic formatting, printing, web-based hosting, and collaboration. You could post this first release as a technical preview to start generating buzz. Later releases could improve on the base features and justify charging a fee: tables, images, and lists in one release, spellchecking and grammar checking in another, and so on.

How to Create a Release Plan

There are two basic types of plans: *scopeboxed* plans and *timeboxed* plans.

A scopeboxed plan defines the features the team will build in advance, but the release date is uncertain. A timeboxed plan defines the release date in advance, but the specific features that release will include are uncertain.

Timeboxed plans are almost always better. They constrain the amount of work you can do and force people to make difficult but important prioritization decisions. This requires the team to identify cheaper, more valuable alternatives to some requests. Without a timebox, your plan will include more low-value features.

To create your timeboxed plan, first choose your release dates. I like to schedule releases at regular intervals, such as once per month and no more than three months apart.

This final list of stories is your release plan. Post it prominently (I use a magnetic whiteboard and refer to it during iteration planning. Every week, consider what you've learned from stakeholders and discuss how you can use that information to improve your plan.

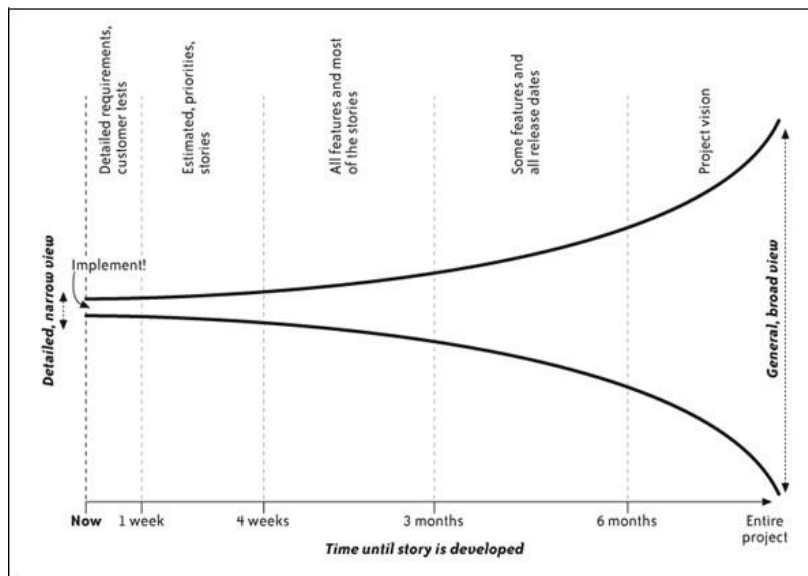
Planning at the Last Responsible Moment

It takes a lot of time and effort to brainstorm stories, estimate them, and prioritize them. If you're adapting your plan as you go, some of that effort will be wasted. To reduce waste, plan at the last responsible moment. The *last responsible moment* is the last moment at which you can responsibly make a decision. In practice, this means that the further away a particular event is, the less detail your release plan needs to contain.

Another way to look at this is to think in terms of planning horizons. Your *planning horizon* determines how far you look into the future. Many projects try to determine every requirement for the project up front, thus using a planning horizon that extends to the end of the project.

To plan at the last responsible moment, use a tiered *set* of planning horizons. Use long planning horizons for general plans and short planning horizons for specific, detailed plans, as shown in figure.

- Define the *vision* for the entire project.
- Define the *release date* for the next two releases.
- Define the *minimum marketable features* for the current release, and start to place features that won't fit in this release into the next release.
- Define all the *stories* for the current feature and most of the current release. Place stories that don't fit into the next release.
- *Estimate and prioritize* stories for the current iteration and the following three iterations.
- Determine *detailed requirements and customer tests* for the stories in the current iteration.



3. Write Briefly about RISK MANAGEMENT?

The following statement is *nearly* true:

Our team delivers a predictable amount of work every iteration. Because we had a velocity of 14 story points last week, we'll deliver 14 story points this week, and next week, and the next. By combining our velocity with

our release plan, we can commit to a specific release schedule!

Good XP teams do achieve a stable velocity. Unfortunately, velocity only reflects the issues the team *normally* faces. Life always has some additional curve balls to throw. Team members get sick and take vacations; hard drives crash, and although the backups worked, the restore doesn't; stakeholders suddenly realize that the software you've been showing them for the last two months needs some major tweaks before it's ready to use.

Despite these uncertainties, your stakeholders need schedule commitments that they can rely upon. *Risk management* allows you to make and meet these commitments.

A Generic Risk-Management Plan

Every project faces a set of common risks: turnover, new requirements, work disruption, and so forth. These risks act as a multiplier on your estimates, doubling or tripling the amount of time it takes to finish your work.

How much of a multiplier do these risks entail? It depends on your organization. In a perfect world, your organization would have a database of the type shown in .It would show the chance of completing before various risk multipliers.

Because most organizations don't have this information available, I've provided some generic risk multipliers instead. These multipliers show your chances of meeting various schedules. For example, in a "Risky" approach, you have a 10 percent chance of finishing according to your estimated schedule. Doubling your estimates gives you a 50 percent chance of on-time completion, and to be virtually certain of meeting your schedule, you have to quadruple your estimates.

Project-Specific Risks

Using the XP practices and applying risk multipliers will help contain the risks that are common to all projects. The generic risk multipliers include the normal risks of a flawed release plan, ordinary requirements growth, and employee turnover. In addition to these risks, you probably face some that are specific to your project. To manage these, create a *risk census*— that is, a list of the risks your project faces that focuses on your project's *unique* risks.

Suggest starting work on your census by brainstorming catastrophes. Gather the whole team and hand out index cards. Remind team members that during this exercise, negative thinking is not only OK, it's necessary.

Ask them to consider ways in which the project could fail. Write several questions on the board:

1. What about the project keeps you up at night?
2. Imagine it's a year after the project's disastrous failure and you're being interviewed about what went wrong. What happened?
3. Imagine your best dreams for the project, then write down the opposite.
4. How could the project fail without anyone being at fault?
5. How could the project fail if it were the stakeholders' faults? The customers' faults? Testers? Programmers? Management? Your fault? Etc.
6. How could the project succeed but leave one specific stakeholder unsatisfied or angry?

Write your answers on the cards, then read them aloud to inspire further thoughts. Some people may be more comfortable speaking out if a neutral facilitator reads the cards anonymously.

Once you have your list of catastrophes, brainstorm scenarios that could lead to those catastrophes. From those scenarios, imagine possible root causes. These root causes are your risks: the causes of scenarios that will lead to catastrophic results.

For example, if you're creating an online application, one catastrophe might be "extended downtime." A scenario leading to that catastrophe would be "excessively high demand," and root causes include "denial of service attack" and "more popular than expected."

After you've finished brainstorming risks, let the rest of the team return to their iteration while you consider the risks within a smaller group. (Include a cross-section of the team.) For each risk, determine:

- Estimated probability—I prefer "high," "medium," and "low."
- Specific impact to project if it occurs—dollars lost, days delayed, and project cancellation are common possibilities.

You may be able to discard some risks as unimportant immediately. I ignore unlikely risks with low impact and all risks with negligible impact. Your generic risk multiplier accounts for those already.

For the risks you decide to handle, determine transition indicators, mitigation and contingency activities, and your risk exposure:

- **Transition indicators** tell you when the risk will come true. It's human nature to downplay upcoming risks, so choose indicators that are objective rather than subjective. For example, if your risk is "unexpected popularity causes extended downtime," then your transition indicator might be "server utilization trend shows upcoming utilization over 80 percent."
- **Mitigation activities** reduce the impact of the risk. Mitigation happens in advance, regardless of whether the risk comes to pass. Create stories for them and add them to your release plan. To continue the example, possible stories include "support horizontal scalability" and "prepare load balancer."
- **Contingency** activities also reduce the impact of the risk, but they are only necessary if the risk occurs. They often depend on mitigation activities that you perform in advance. For example, "purchase more bandwidth from ISP," "install load balancer," and "purchase and prepare additional frontend servers."
- **Risk exposure** reflects how much time or money you should set aside to contain the risk. To calculate this, first estimate the numerical probability of the risk and then multiply that by the impact. When considering your impact, remember that you will have already paid for mitigation activities, but contingency activities are part of the impact. For example, you might believe that downtime due to popularity is 35 percent likely, and the impact is three days of additional programmer time and \$20,000 for bandwidth, colocation fees, and new equipment. Your total risk exposure is \$7,000 and one day. Some risks have a 100 percent chance of occurring. These are no longer risks—they are reality. Update your release plan to deal with them.

How to Make a Release Commitment

With your risk exposure and risk multipliers, you can predict how many story points you can finish before your release date. Start with your timeboxed release date from your release plan. Figure out how many iterations remain until your release date and subtract your risk exposure. Multiply by your velocity to determine the number of points remaining in your schedule, then divide by each risk multiplier to calculate your chances of finishing various numbers of story points.

$$\text{risk_adjusted_points_remaining} = (\text{iterations_remaining} - \text{risk_exposure}) * \text{velocity} / \text{risk_multiplier}$$

For example, if you're using a rigorous approach, your release is 12

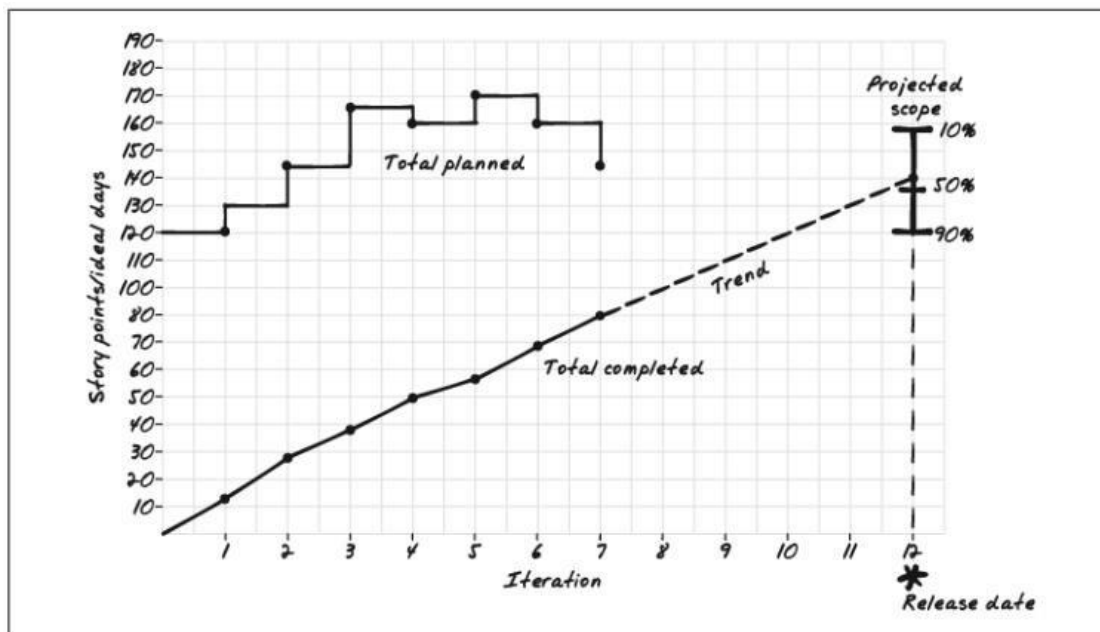
iterations away, your velocity is 14 points, and your risk exposure is one iteration, you would calculate the range of possibilities as:

points remaining = $(12 - 1) * 14 = 154$ points
10 percent chance: $154 / 1 = 154$ points
50 percent chance: $154 / 1.4 = 110$ points
90 percent chance: $154 / 1.8 = 86$ points

In other words, when it's time to release, you're 90 percent likely to have finished 86 more points of work, 50 percent likely to have finished 110 more points, and only 10 percent likely to have finished 154 more points.

You can show this visually with a *burn-up chart*, shown in .Every week, note how many story points you have completed, how many total points exist in your next release (completed plus remaining stories), and your range of risk-adjusted points remaining. Plot them on the burn-up chart as shown in the figure.

Use your burn-up chart and release plan to provide stakeholders with a list of features you're committing to deliver on the release date. I commit to delivering features that are 90 percent likely to be finished, and I describe features between 50 and 90 percent likely as *stretch goals*. I don't mention features that we're less than 50 percent likely to complete.



A burn-up chart

4. Explain about Iteration Planning.

Iterations are the heartbeat of an XP project. When an iteration starts, stories flow in to the team as they select the most valuable stories from the release plan. Over the course of the iteration, the team breathes those stories to life. By the end of the iteration, they've pumped out working, tested software for each story and are ready to begin the cycle again.

Iterations are an important safety mechanism. Every week, the team stops, looks at what it's accomplished, and shares those accomplishments with stakeholders. By doing so, the team coordinates its activities and communicates its progress to the rest of the organization. Most importantly, iterations counter a common risk in software projects: the tendency for work to take longer than expected.

The Iteration Timebox

Iterations allow you to avoid this surprise. Iterations are exactly one week long and have a strictly defined completion time. This is a *timebox*: work ends at a particular time regardless of how much you've finished. Although the iteration timebox doesn't *prevent* problems, it *reveals* them, which gives you the opportunity to correct the situation.

In XP, the iteration demo marks the end of the iteration. Schedule the demo at the same time every week. Most teams schedule the demo first thing in the morning, which gives them a bit of extra slack the evening before for dealing with minor problems.

The Iteration Schedule

Iterations follow a consistent, unchanging schedule:

- Demonstrate previous iteration (up to half an hour)
- Hold retrospective on previous iteration (one hour)
- Plan iteration (half an hour to four hours)
- Commit to delivering stories (five minutes)
- Develop stories (remainder of iteration)
- Prepare release (less than 10 minutes)

How to Plan an Iteration

After the iteration demo and retrospective are complete, iteration planning begins. Start by measuring the velocity of the previous iteration. Take all the stories that are “done done” and add up their original estimates. This number is the amount of story points you can reasonably expect to complete in the upcoming iteration. With your velocity in hand, you can select the stories to work on this iteration. Ask your customers to select the most important stories from the release plan. Select stories that exactly add up to the team’s velocity. You may need to split stories or include one or two less important stories to make the estimates add up perfectly.

Engineering tasks are concrete tasks for the programmers to complete. Unlike stories, engineering tasks don’t need to be customer-centric. Instead, they’re programmer-centric. Typical engineering tasks include:

- Update build script
- Implement domain logic
- Add database table and associated ORM objects
- Create new UI form

Brainstorm the tasks you need in order to finish all the iteration’s stories. Some tasks will be specific to a single story; others will be useful for multiple stories. Focus only on tasks that are necessary for completing the iteration’s stories. Don’t worry about all-hands meetings, vacations, or other interruptions.

After the Planning Session

After you finish planning the iteration, work begins. Decide how you’ll deliver on your commitment. In practice, this usually means that programmers volunteer to work on a task and ask for someone to pair with them. As pairs finish their tasks, they break apart. Individuals pick up new tasks from the board and form new pairs.

Other team members each have their duties as well. so other team members may not have a task planning board like the programmers do. Instead, the customers and testers keep an eye on the programmers’ progress and organize their work so it’s ready when the programmers need it. This maximizes the productivity of the whole team.

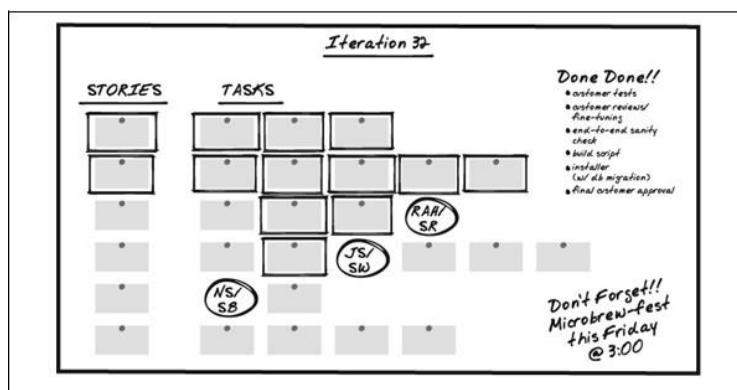
At the end of the iteration, release your completed software to stakeholders. With a good 10-minute build, this shouldn't require any more than a button press and a few minutes' wait. The following morning, start a new iteration by demonstrating what you completed the night before.

Dealing with Long Planning Sessions

Iteration planning should take anywhere from half an hour to four hours. Most of that time should be for discussion of engineering tasks. For established XP teams, assuming they start their iteration demo first thing in the morning, planning typically ends by lunchtime.

New teams often have difficulty finishing planning so quickly. This is normal during the first several iterations. It will take a little while for you to learn your problem space, typical approaches to design problems, and how best to work together.

If iteration planning still takes a long time after the first month or so, look for ways to speed it up. One common problem is spending too much time doing release planning during the iteration planning meeting. Most release planning should happen during the previous iteration, primarily among customers, while the programmers work on stories. Picking stories for the iteration plan should be a simple matter of taking stories from the front of the release plan. It should only take a few minutes because you won't estimate or discuss priorities.



An iteration planning board

Long planning sessions also result from spending a lot of time trying to break down the stories into engineering tasks. This may be a result of doing too much design work. Although iteration planning *is* a design activity, it's a very high-level one. Most of the real design work will happen during the iteration as pairs work on specific tasks. If you spend much time discussing possible design details, ask yourself whether you really need to solve these problems in order to come up with good engineering tasks.

If you find that team members don't understand the existing design, or if you have long discussions about how it works, you may lack shared design understanding. Remedy this problem with collective code ownership and more pair programming.

If you find yourselves speculating about possible design choices, your problem may be a result of trying to make your design too general. Remember to keep the design simple. Focus on the requirements that you have today. Trust pairs doing test-driven development to make good decisions on their own.

Design speculation can also occur when you don't understand the requirements well. Take advantage of the on-site customer in your meeting. Ask him to explain the details of each story and why the system needs to behave in a particular way.

5. Explain about the SLACK.

Imagine that the power cable for your workstation is just barely long enough to reach the wall receptacle. You can plug it in if you stretch it taut, but the slightest vibration will cause the plug to pop out of the wall and the power to go off. You'll lose everything you were working on.

I can't afford to have my computer losing power at the slightest provocation. My work's too important for that. In this situation, I would move the computer closer to the outlet so that it could handle some minor bumps. People couldn't trip over it, install an uninterruptable power supply, and invest in a continuous backup server.)

Your project plans are also too important to be disrupted by the slightest provocation. Like the power cord, they need slack.

How Much Slack?

The amount of slack you need doesn't depend on the number of problems you face. It depends on the *randomness* of problems. If you always experience exactly 20 hours of problems in each iteration, your velocity will automatically compensate. However, if you experience between 20 and 30 hours of problems in each iteration, your velocity will bounce up and down. You need 10 hours of slack to stabilize your velocity and to ensure that you'll meet your commitments.

These numbers are just for illustration. Instead of measuring the number of hours you spend on problems, take advantage of velocity's feedback loop. If your velocity bounces around a lot, stop signing up for more stories than your velocity allows. This will cause your velocity to settle at a lower number that incorporates enough slack for your team. On the other hand, if your velocity is rock solid, try reducing slack by committing to a small extra story next iteration.

How to Introduce Slack

One way to introduce slack into your iterations might be to schedule no work on the last day or two of your iteration. This would give you slack, but it would be pretty wasteful. A better approach is to schedule useful, important work that isn't time-critical work you can set aside in case of an emergency. *Paying down technical debt* fits the bill perfectly.

Even the best teams inadvertently accumulate technical debt. Although you should always make your code as clean as you can, some technical debt will slip by unnoticed.

Rather than doing the bare minimum necessary to keep your head above water, be generous in refactoring and cleaning up technical debt in existing code. Every iteration, look for opportunities to make existing code better. Make this part of your everyday work. Every time I find myself scratching my head over a variable or method name, I change it. If I see some code that's no longer in use, I delete it.

When Your Iteration Commitment Is at Risk

Research time and paying down technical debt are important tasks that enhance your programmers' skills and allow them to deliver more quickly. Paying down technical debt, in particular, should be part of every iteration. However, if your iteration commitments are at risk, it's OK to set these two

tasks aside *temporarily* in order to meet those commitments.

Use refactoring as a shock absorber

Before starting a big refactoring to pay down technical debt, consider the iteration plan and think about the kinds of problems the team has encountered. If the iteration is going smoothly, go ahead and refactor. If you've encountered problems or you're a little behind schedule, shrug your shoulders and work on meeting your iteration commitment instead. You'll have another opportunity to fix the problem later. By varying the amount of time you spend paying down technical debt, you can ensure that most iterations come in exactly on time.

Reducing the Need for Slack

In my experience, there are two big sources of randomness on XP teams: customer unavailability and technical debt. Both lead to an unpredictable environment, make estimating difficult, and require you to have more slack in order to meet your commitments.

If programmers have to wait for customer answers as they work, you can reduce the need for slack by making customers more available to answer programmer questions. They may not like that—customers are often surprised by the amount of time XP needs from them—but if you explain that it will help improve velocity, they may be more interested in helping.

On the other hand, if programmers often encounter unexpected *technical* delays, such as surprising design problems, difficulty integrating, or unavailability of a key staging environment, then your need for slack is due to too much technical debt. Fortunately, using your slack to pay down technical debt will automatically reduce the amount of slack you need in the future.

6. Write about the STORIES concept in planning?

Stories may be the most misunderstood entity in all of XP. They're not requirements. They're not use cases. They're not even narratives. They're much simpler than that.

Stories are for planning. They're simple one- or two-line descriptions of work the team should produce. Alistair Cockburn calls them "promissory notes for future conversation. Everything that stakeholders want the team to produce should have a story, for example:

- "Warehouse inventory report"
- "Full-screen demo option for job fair"
- "TPS report for upcoming investor dog-and-pony show"
- "Customizable corporate branding on user login screen"

This isn't enough detail for the team to implement and release working software, nor is that the intent of stories. A story is a placeholder for a detailed discussion about requirements. Customers are responsible for having the requirements details available when the rest of the team needs them.

Although stories are short, they still have two important characteristics:

1. Stories represent *customer value* and are written in the customers' terminology. (The best stories are actually written by customers.) They describe an end-result that the customer values, not implementation details.

2. Stories have clear *completion criteria*. Customers can describe an objective test that would allow programmers to tell when they've successfully implemented the story.

The following examples are *not* stories:

- “Automate integration build” does not represent customer value.
- “Deploy to staging server outside the firewall” describes implementation details rather than an end-result, and it doesn't use customer terminology. “Provide demo that customer review board can use” would be better.
- “Improve performance” has no clear completion criteria. Similarly, “Make it fast enough for my boss to be happy” lacks *objective* completion criteria. “Searches complete within one second” is better.

Story Cards

This isn't the result of some strange Ludditian urge on the part of XP's creators; it's a deliberate choice based on the strengths of the medium. You see, physical cards have one feature that no conglomeration of pixels has: they're tactile. You can pick them up and move them around. This gives them power.

Story cards also form an essential part of an informative workspace. After the planning meeting, move the cards to the release planning board—a big, six-foot whiteboard, placed prominently in the team's open workspace. You can post hundreds of cards and still see them all clearly. For each iteration, place the story cards to finish during the iteration on the iteration planning board (another big whiteboard; and move them around to indicate your status and progress. Both of these boards are clearly visible throughout the team room and constantly broadcast information to the team.

Index cards also help you be responsive to stakeholders. When you talk with a stakeholder and she has a suggestion, invite her to write it down on a blank index card. I always carry cards with me for precisely this purpose. Afterward, take the stakeholder and her card to the product manager. They can walk over to the release planning board and discuss the story's place in the overall vision.

Physical index cards enable these ways of working in a very easy and natural way that's surprisingly difficult to replicate with computerized tools. Although you can use software, index cards just work better: they're easier to set up and manipulate, make it easier to see trends and the big picture, and allow you to change your process with no configuration or programming.

Most people are skeptical about the value of index cards at first, so if you feel that way, you're not alone. The best way to evaluate the value of physical story cards is to try them for a few months, then decide whether to keep them.

Splitting and Combining Stories

Although stories can start at any size, it is difficult to estimate stories that are too large or too small. Split large stories; combine small ones.

The right size for a story depends on your velocity. You should be able to complete 4 to 10 stories in each iteration. Split and combine stories to reach this goal. For example, a team with a velocity of 10 days per iteration might split stories with estimates of more than 2 days and combine stories that are less than half a day.

Combining stories is easy. Take several similar stories, staple their cards together, and write your new estimate on the front.

Splitting stories is more difficult because it tempts you away from vertical stripes and releasable stories. It's easiest to just create a new story for each step in the previous story. Unfortunately, this approach leads to story clumps. Instead, consider the *essence* of the story. Peel away all the other issues and write them as new stories.

Agile Estimating and Planning. He summarizes various options for splitting stories:

- Split large stories along the boundaries of the data supported by the story.
- Split large stories based on the operations that are performed within the story.
- Split large stories into separate CRUD (Create, Read, Update, Delete) operations.
- Consider removing cross-cutting concerns (such as security, logging, error handling, and so on) and creating two versions of the story: one with and one without support for the cross-cutting concern.
- Consider splitting a large story by separating the functional and nonfunctional (performance, stability, scalability, and so forth) aspects into separate stories.
- Separate a large story into smaller stories if the smaller stories have different priorities.

Special Stories

Most stories will add new capabilities to your software, but any action that requires the team's time and is not a part of their normal work needs a story.

Documentation stories

XP teams need very little documentation to do their work but you may need the team to produce documentation for other reasons. Create documentation stories just like any other: make them customer-centric and make sure you can identify specific completion criteria. An example of a documentation story is "Produce user manual."

"Nonfunctional" stories

Performance, scalability, and stability—so-called *nonfunctional* requirements should be scheduled with stories, too. Be sure that these stories have precise completion criteria.

Bug stories

Bug stories can be difficult to estimate. Often, the biggest timesink in

debugging is figuring out what's wrong, and you usually can't estimate how long that will take. Instead, provide a timeboxed estimate: "We'll spend up to a day investigating this bug."

Spike stories

Sometimes programmers won't be able to estimate a story because they don't know enough about the technology required to implement the story. In this case, create a story to research that technology. An example of a research story is "Figure out how to estimate 'Send HTML' story." Programmers will often use a *spike solution* so these sorts of stories are typically called *spike stories*.

Word these stories in terms of the goal, not the research that needs to be done. When programmers work on a research story, they only need to do enough work to make their estimate for the real story. They shouldn't try to figure out all the details or solve the entire problem.

7. Write briefly about Estimating?

Other than spike stories, you normally don't need to schedule time for the programmers to estimate stories you can just ask them for an estimate at any time. It's part of the overhead of the iteration, as are support requests and other unscheduled interruptions. If your programmers feel that estimating is too much of an interruption, try putting new story cards in a pile for the programmers to estimate when it's convenient.

Meetings

Like estimating, most meetings are part of the normal overhead of the iteration. If you have an unusual time commitment, such as training or an off-site day, you can reserve time for it with a story.

Architecture, design, refactoring, and technical infrastructure

Don't create stories for technical details. Technical tasks are part of the cost of implementing stories and should be part of the estimates. Use incremental design and architecture to break large technical requirements into small pieces that you can implement incrementally.

Velocity

Although estimates are almost never accurate, they are *consistently* inaccurate. While the estimate accuracy of *individual* estimates is all over the map one estimate might be half the actual time, another might be 20percent more than the actual time the estimates *are* consistent in aggregate.

This is where velocity comes in. Your *velocity* is the number of story points you can complete in an iteration. It's a simple yet surprisingly sophisticated tool. It uses a feedback loop, which means every iteration's velocity reflects what the team actually achieved in the previous iteration.

Velocity and the Iteration Timebox

Velocity relies upon a strict iteration timebox. To make velocity work, *never* count stories that aren't "done done" at the end of the iteration. *Never* allow the iteration deadline to slip, not even by a few hours.

How to Make Consistent Estimates

There's a secret to estimating: experts automatically make consistent estimates. Velocity automatically applies the appropriate amount of padding for short-term estimates and risk management adds padding for long-term estimates.

There are two corollaries to this secret.

First, if you're an expert but you don't trust your ability to make estimates, relax. You automatically make good estimates. Just imagine the work you're going to do and pick the first number that comes into your head. It won't be right, but it will be consistent with your other estimates. That's sufficient.

Second, if you're not an expert, the way to make good estimates is to become an expert. This isn't as hard as it sounds. An expert is just a beginner with lots of experience. To become an expert, make a lot of estimates with relatively short timescales and pay attention to the results. In other words, follow the XP practices.

All the programmers should participate in estimating. At least one customer should be present to answer questions. Once the programmers have all the information they need, one programmer will suggest an estimate. Allow this to happen naturally. The person who is most comfortable will speak first. Often this is the person who has the most expertise.

How to Estimate Stories

Estimate stories in story points. To begin, think of a story point as an ideal day. When you start estimating a story, imagine the engineering tasks you will need to implement it. Ask your on-site customers about their expectations, and focus on those things that would affect your estimate. If you encounter something that seems expensive, provide the customers with less costly alternatives.

How to Estimate Iteration Tasks

During iteration planning, programmers create engineering tasks that allow them to deliver the stories planned for the iteration. Each engineering task is a concrete, technical task such as "update build script" or "implement domain logic."

How to Improve Your Velocity

The following options might allow you to improve your velocity.

Pay down technical debt

The most common technical problem is excessive technical debt. This has a greater impact on team productivity than any other factor does. Make code quality a priority and your velocity will improve dramatically. However, this isn't a quick fix. Teams with excessive technical debt often have months or even years of cleanup ahead of them. Rather than stopping work to pay down technical debt, fix it incrementally. Iteration slack is the best way to do so, although you may not see a noticeable improvement for several months.

Improve customer involvement

If your customers aren't available to answer questions when programmers need them, programmers have to either wait or make guesses about the answers. Both of these hurt velocity. To improve your velocity, make sure that a customer is always available to answer programmer questions.

Improve customer involvement

If your customers aren't available to answer questions when programmers need them,

programmers have to either wait or make guesses about the answers. Both of these hurt velocity.

Support energized work

Tired, burned-out programmers make costly mistakes and don't put forth their full effort. If your organization has been putting pressure on the team, or if programmers have worked a lot of extra hours, shield the programmers from organizational pressure and consider instituting a no-overtime policy.

Offload programmer duties

If programmers are the constraint for your team—as this book assumes—then hand any work that other people can do to other people. Find ways to excuse programmers from unnecessary meetings, shield them from interruptions, and have somebody else take care of organizational bureaucracy such as time sheets and expense reports. You could even hire an administrative assistant for the team to handle all non-project-related matters.

UNIT-5 DEVELOPING

Long Question and Answers

1. Write about INCREMENTAL REQUIREMENTS.

A team using an up-front requirements phase keeps their requirements in a requirements document. An XP team doesn't have a requirements phase and story cards aren't miniature requirements documents, so where do requirements come from?

The Living Requirements Document

In XP, the on-site customers sit with the team. They're expected to have all the information about requirements at their fingertips. When somebody needs to know something about the requirements for the project, she asks one of the on-site customers rather than looking in a document.

Face-to-face communication is much more effective than written communication, as discussed, and it allows XP to save time by eliminating a long requirements analysis phase. However, requirements work is still necessary. The on-site customers need to understand the requirements for the software before they can explain it.

The key to successful requirements analysis in XP is *expert customers*. Involve real customers, an experienced product manager, and experts in your problem domain. Many of the requirements for your software will be intuitively obvious to the right customers.

Work Incrementally

Work on requirements *incrementally*, in parallel with the rest of the team's work. This makes your work easier and ensures that the rest of the team won't have to wait to get started. Your work will typically parallel your release-planning horizons. Vision, features, and stories

Start by clarifying your project vision, then identify features and stories as described in These initial ideas will guide the rest of your requirements work.

Rough expectations

Figure out what a story means to you and how you'll know it's finished slightly before you ask programmers to estimate it. As they estimate, programmers will ask questions about your expectations; try to anticipate those questions and have answers ready. A rough sketch of the visible aspects of the story might help.

Mock-ups, customer tests, and completion criteria

Figure out the details for each story just before programmers start implementing it. Create rough mock-ups that show what you expect the work to look like when it's done. Prepare customer tests that provide examples of tricky domain concepts, and describe what "done done" means for each story.

Customer review

While stories are under development, before they're "done done," review each story to make sure it works as you expected. You don't need to exhaustively test the application—you can rely on the programmers to test their work—but you should check those areas in which programmers might think differently than you do. These areas include terminology, screen layout, and interactions between screen elements.

Some of your findings will reveal errors due to miscommunication or misunderstanding. Others, while meeting your requirements, won't work as well in practice as you had hoped. In either case, the solution is the same: talk with the programmers about making changes.

You can even pair with programmers as they work on the fixes. Many changes will be minor, and the programmers will usually be able to fix them as part of their iteration slack. If there are major changes, however, the programmers may not have time to fix them in the current iteration.

Create story cards for these changes. Before scheduling such a story into your release plan, consider whether the value of the change is worth its cost. Over time, programmers will learn about your expectations for the application. Expect the number of issues you discover to decline each iteration.

2. Explain briefly about CUSTOMER TESTS?

We implement tricky domain concepts correctly. Customers have specialized expertise, or domain knowledge, that programmers don't have. Some areas of the application— what programmers call domain rules—require this expertise. You need to make sure that the programmers understand the domain rules well enough to code them properly in the application.

Customer tests help customers communicate their expertise. Don't worry; this isn't as complicated as it sounds.

Customer tests are really just examples. Your programmers turn them into automated tests, which they then use to check that they've implemented the domain rules correctly. Once the tests are passing, the programmers will include them in their 10-minute build, which will inform the programmers if they ever do anything to break the tests.

To create customer tests, follow the Describe, Demonstrate, Develop processes outlined in the next

section. Use this process during the iteration in which you develop the corresponding stories.

Describe

At the beginning of the iteration, look at your stories and decide whether there are any

aspects that programmers might misunderstand. You don't need to provide examples for everything. Customer tests are for communication, not for proving that the software works. **Demonstrate** After a brief discussion of the rules, provide concrete examples that illustrate the scenario. Tables are often the most natural way to describe this information, but you don't need to worry about formatting. Just get the examples on the whiteboard. The scenario might continue like this: Customer (continued): As an example, this invoice hasn't been sent to customers, so an Account Rep can delete it.

Sent	User	OK to delete
N	Account Rep	Y

In fact, anybody can delete it—CSRs, managers, and admins.

Sent	User	OK to delete
N	CSR	Y
N	Manager	Y
N	Admin	Y

But once it's sent, only managers and admins can delete it, and even then it's audited.

Sent	User	OK to delete
Y	Account Rep	N
Y	CSR	N
Y	Manager	Audited
Y	Admin	Audited

Also, it's not a simple case of whether something has been sent or not. "Sent" actually means one of several conditions. If you've done anything that could have resulted in a customer seeing the invoice, we consider it "Sent." Now only a manager or admin can delete it

Sent	User	OK to delete
Printed	Account Rep	N
Exported	Account Rep	N
Posted to Web	Account Rep	N
Emailed	Account Rep	N

One particularly effective way to work is to elaborate on a theme. Start by discussing the most basic case and providing a few examples. Next, describe a special case or additional

detail and provide a few more examples. Continue in this way, working from simplest to most complicated, until you have described all aspects of the rule.

Focus on Business

Rules One of the most common mistakes in creating customer tests is describing what happens in the user interface rather than providing examples of business rules. For example, to show that an account rep must not delete a mailed invoice, you might make the mistake of writing this:

1. Log in as an account rep
2. Create new invoice
3. Enter data
4. Save invoice
5. Email invoice to customer
6. Check if invoice can be deleted (should be “no”)

What happened to the core idea? It’s too hard to see. Compare that to the previous approach:

Sent	User	OK to delete
Emailed	Account Rep	N

Good examples focus on the *essence* of your rules. Rather than imagining how those rules might work in the application, just think about what the rules are. If you weren’t creating an application at all, how would you describe those rules to a colleague? Talk about *things* rather than *actions*. Sometimes it helps to think in terms of a template: “When (*scenario X*), then (*scenario Y*).”

It takes a bit of practice to think this way, but the results are worth it. The tests become more compact, easier to maintain, and (when implemented correctly) faster to run.

Automating the Examples

Programmers may use any tool they like to turn the customers’ examples into automated tests. Ward Cunningham’s Fit (Framework for Integrated Test),* is specifically designed for this purpose. It allows you to use HTML to mix descriptions and tables, just as in my invoice auditing example, then runs the tables through programmer-created fixtures to execute the tests.

Fit is a great tool for customer tests because it allows customers to review, run, and even expand on their own tests. Although programmers have to write the fixtures, customers can easily add to or modify existing tables to check an idea. Testers can also modify the tests as an aid to exploratory testing. Because the tests are written in HTML, they can use any HTML editor to modify the tests, including Microsoft Word.

Programmers, don’t make Fit too complicated. It’s a deceptively simple tool. Your fixtures should work like unit tests, focusing on just a few domain objects. For example, the invoice auditing example would use a custom Column Fixture. Each column in the table corresponds to a variable or method in the fixture. The code is almost trivial

```

public class InvoiceAuditingFixture : ColumnFixture
{
    public InvoiceStatus Sent;
    public UserRole User;

    public Permission OkayToDelete() {
        InvoiceAuditer auditor = new InvoiceAuditer(User, InvoiceStatus)
        return auditor.DeletePermission;
    }
}

```

3. Explain about TEST-DRIVEN DEVELOPMENT?

Test-driven development, or TDD, is a rapid cycle of testing, coding, and refactoring. When adding a feature, a pair may perform dozens of these cycles, implementing and refining the software in baby steps until there is nothing left to add and nothing left to take away.

Research shows that TDD substantially reduces the incidence of defects. When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

TDD isn't perfect, of course. TDD is difficult to use on legacy codebases. Even with green field systems, it takes a few months of steady use to overcome the learning curve.

Try it anyway although TDD benefits from other XP practices, it doesn't require them. You can use it on almost any project.

Why TDD Works

Back in the days of punch cards, programmers laboriously hand-checked their code to make sure it would compile. A compile error could lead to failed batch jobs and intense debugging sessions to look for the misplaced character. Getting code to compile isn't such a big deal anymore. Most IDEs check your syntax as you type, and some even compile every time you save.

The feedback loop is so fast that errors are easy to find and fix. If something doesn't compile, there isn't much code to check.

Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code. Every few minutes—as often as every 20 or 30 seconds—TDD verifies that the code does what you think it should do. If something goes wrong, there are only a few lines of code to check. Mistakes are easy to find and fix.

TDD uses an approach similar to double-entry bookkeeping. You communicate your intentions twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere. Getting code to compile isn't such a big deal anymore. Most IDEs check your syntax as you type, and some even compile every time you save. The feedback loop is so fast that errors are easy to find and fix. If something doesn't compile, there isn't much code to check. Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code.

Every few minutes—as often as every 20 or 30 seconds—TDD verifies that the code does what you think it should do. If something goes wrong, there are only a few lines of code to check. Mistakes are

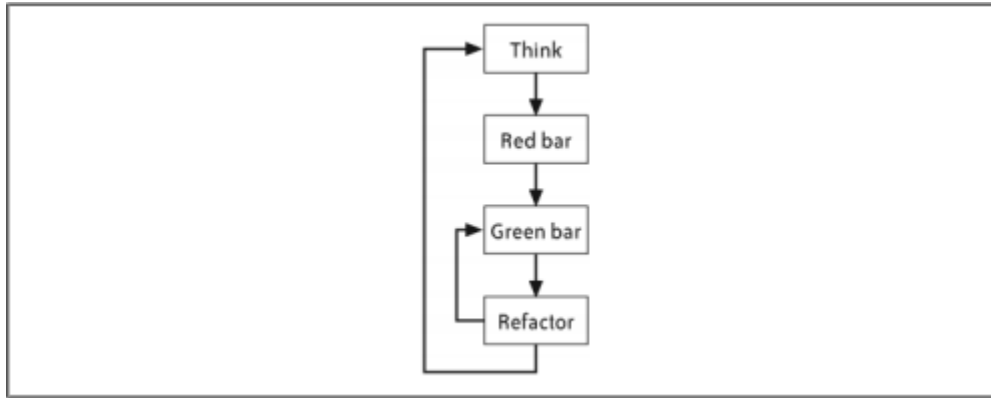
easy to find and fix. TDD uses an approach similar to double-entry bookkeeping. You communicate your intentions twice, stating the same idea in different ways: first with a test, then with production code. When they match, it's likely they were both coded correctly. If they don't, there's a mistake somewhere.

In TDD, the tests are written from the perspective of a class' public interface. They focus on the class' behavior, not its implementation. Programmers write each test before the corresponding production code. This focuses their attention on creating interfaces that are easy to use rather than easy to implement, which improves the design of the interface.

After TDD is finished, the tests remain. They're checked in with the rest of the code, and they act as living documentation of the code. More importantly, programmers run all the tests with (nearly) every build, ensuring that code continues to work as originally intended. If someone accidentally changes the code's behavior—for example, with a misguided refactoring—the tests fail, signalling the mistake.

How to Use TDD

You can start using TDD today. It's one of those things that takes moments to learn and a lifetime to master. Imagine TDD as a small, fast-spinning motor. It operates in a very short cycle that repeats over and over again. Every few minutes, this cycle ratchets your code forward a notch, providing code that— although it may not be finished—has been tested, designed, coded, and is ready to check in.



The TDD cycle

Step 1:

Think TDD uses small tests to force you to write your code—you only write enough code to make the tests pass. The XP saying is, “Don’t write any production code unless you have a failing test.” Your first step, therefore, is to engage in a rather odd thought process. Imagine what behavior you want your code to have, then think of a small increment that will require fewer than five lines of code. Next, think of a test—also a few lines of code—that will fail unless that behavior is present. In other words, think of a test that will force you to add the next few lines of production code. This is the hardest part of TDD because the concept of tests driving your code seems backward, and because it can be difficult to think in small increments. Pair programming helps. While the driver tries to make the current test pass, the navigator should stay a few steps ahead, thinking of tests that will drive the code to the next increment.

Step 2: Red bar Now write the test. Write only enough code for the current increment of behavior—typically fewer than five lines of code. If it takes more, that’s OK, just try for a smaller increment next time. Code in terms of the class’ behavior and its public interface, not how you think you will implement the internals of the class. Respect encapsulation.

In the first few tests, this often means you write your test to use method and class names that don’t exist yet. This is intentional—it forces you to design your class’ interface from the perspective of a user of the class, not as its implementer.

After the test is coded, run your entire suite of tests and watch the new test fail. In most TDD testing tools, this will result in a red progress bar. This is your first opportunity to compare your intent with what’s actually happening. If the test doesn’t fail, or if it fails in a different way than you expected, something is wrong. Perhaps your test is broken, or it doesn’t test what you thought it did.

Troubleshoot the problem; you should always be able to predict what’s happening with the code. After the test is coded, run your entire suite of tests and watch the new test fail. In most TDD testing tools, this will result in a red progressbar.

This is your first opportunity to compare your intent with what's actually happening. If the test doesn't fail, or if it fails in a different way than you expected, something is wrong. Perhaps your test is broken, or it doesn't test what you thought it did. Troubleshoot the problem; you should always be able to predict what's happening with the code.

Step 3: Green bar Next, write just enough production code to get the test to pass. Again, you should usually need less than five lines of code. Don't worry about design purity or conceptual elegance—just do what you need to do to make the test pass. Sometimes you can just hardcode the answer. This is OK because you'll be refactoring in a moment. Run your tests again, and watch all the tests pass. This will result in a green progress bar. This is your second opportunity to compare your intent with reality. If the test fails, get back to known-good code as quickly as you can. Often, you or your pairing partner can see the problem by taking a second look at the code you just wrote. If you can't see the problem, consider erasing the new code and trying again. Sometimes it's best to delete the new test (it's only a few lines of code, after all) and start the cycle over with a smaller increment.

Step 4: Refactor With all your tests passing again, you can now refactor without worrying about breaking anything. Review the code and look for possible improvements. Ask your navigator if he's made any notes. For each problem you see, refactor the code to fix it. Work in a series of very small refactorings—a minute or two each, certainly not longer than five minutes—and run the tests after each one. They should always pass. As before, if the test doesn't pass and the answer isn't immediately obvious, undo the refactoring and get back to known-good code. Refactor as many times as you like. Make your design as good as you can, but limit it to the code's existing behavior. Don't anticipate future needs, and certainly don't add new behavior.

Step 5: Repeat When you're ready to add new behavior, start the cycle over again. Each time you finish the TDD cycle, you add a tiny bit of well-tested, well-designed code. The key to success with TDD is small increments. Typically, you'll run through several cycles very quickly, then spend more time on refactoring for a cycle or two, then speed up again. With practice, you can finish more than 20 cycles in an hour. Don't focus too much on how fast you go, though. That might tempt you to skip refactoring and design, which are too important to skip. Instead, take very small steps, run the tests frequently, and minimize the time you spend with a red bar.

Testing Tools

To use TDD, you need a testing framework. The most popular are the open source xUnit tools, such as JUnit (for Java) and NUnit (for .NET). Although these tools have different authors, they typically share the basic philosophy of Kent Beck's pioneering SUnit.

If your platform doesn't have an xUnit tool, you can build your own. Although the existing tools often provide GUIs and other fancy features, none of that is necessary

Unit Tests

Unit tests focus just on the class or method at hand. They run entirely in memory, which makes them very fast. Depending on your platform, your testing tool should be able to run at least 100 unit tests per second.

1. It talks to a database
2. It communicates across a network
3. It touches the file system
4. You have to do special things to your environment (such as editing configuration files) to run it

End-to-End Tests

In a perfect world, the unit tests and focused integration tests mesh perfectly to give you total confidence in your tests and code. You should be able to make any changes you want without fear, comfortable in the knowledge that if you make a mistake, your tests will catch them. How can you be sure your unit tests and integration tests mesh perfectly? One way is to write end-to-end tests. End-to-end tests exercise large swaths of the system, starting at (or just behind) the user interface, passing through the business layer, touching the database, and returning.

Acceptance tests and functional tests are common examples of end-to-end tests. Some people also call them integration tests, although I reserve that term for focused integration tests. End-to-end tests can give you more confidence in your code, but they suffer from many problems. They're difficult to create because they require error-prone and labor-intensive setup and teardown procedures. They're brittle and tend to break whenever any part of the system or its setup data changes. They're very slow—they run in seconds or even minutes per test, rather than multiple tests per second.

They provide a false sense of security, by exercising so many branches in the code that it's difficult to say which parts of the code are actually covered. Instead of end-to-end tests, use exploratory testing to check the effectiveness of your unit and integration tests. When your exploratory tests find a problem, use that information to improve your approach to unit and integration testing, rather than introducing end-to-end tests.

4. Explain about REFACTORING?

Entropy always wins. Eventually, chaos turns your beautifully imagined and well-designed code into a big mess of spaghetti. At least, that's the way it used to be, before refactoring. Refactoring is the process of changing the design of your code without changing its behavior—what it does stays the same, but how it does it changes. Refactorings are also reversible; sometimes one form is better than another for certain cases. Just as you can

change the expression $x^2 - 1$ to $(x + 1)(x - 1)$ and back, you can change the design of your code—and once you can do that, you can keep entropy at bay.

Reflective Design

Refactoring enables an approach to design I call reflective design. In addition to creating a design and coding it, you can now analyze the design of existing code and improve it. One of the best ways to identify improvements is with code smells: condensed nuggets of wisdom that help you identify common problems in design.

A code smell doesn't necessarily mean there's a problem with the design. It's like a funky smell in the kitchen: it could indicate that it's time to take out the garbage, or it could just mean that Uncle Gabriel is cooking with a particularly pungent cheese. Either way, when you smell something funny, take a closer look.

Divergent Change and Shotgun Surgery These two smells help you identify cohesion problems in your code. Divergent Change occurs when unrelated changes affect the same class. It's an indication that your class involves too many concepts.

Split it, and give each concept its own home. Shotgun Surgery is just the opposite: it occurs when you have to modify multiple classes to support changes to a single idea. This indicates that the concept is represented in many places throughout your code. Find or create a single home for the idea.

Primitive Obsession and Data Clumps

Some implementations represent high-level design concepts with primitive types. For example, a decimal might represent dollars. This is the Primitive Obsession code smell. Fix it by encapsulating the concept in a class.

Data Clumps are similar. They occur when several primitives represent a concept as a group. For example, several strings might represent an address. Rather than being encapsulated in a single class, however, the data just clumps together. When you see batches of variables consistently passed around together, you're probably facing a Data Clump. As with Primitive Obsession, encapsulate the concept in a single class.

Data Class and Wannabee Static Class

One of the most common mistakes I see in object-oriented design is when programmers put their data and code in separate classes. This often leads to duplicate data-manipulation code. When you have a class that's little more than instance variables combined with accessors and mutators (getters and setters), you have a Data Class. Similarly, when you have a class that contains methods but no meaningful per-object state, you have a

Wannabee Static Class.

Coddling Nulls

Null references pose a particular challenge to programmers: they're occasionally useful, but most they often indicate invalid states or other error conditions. Programmers are usually unsure what to do when they receive a null reference; their methods often check for null references and then return null themselves.

Coddling Nulls like this leads to complex methods and error-prone software. Errors suppressed with null cascade deeper into the application, causing unpredictable failures later in the execution of the software. Sometimes the null makes it into the database, leading to recurring application failures.

Instead of Coddling Nulls, adopt a fail fast strategy Don't allow null as a parameter to any method, constructor, or property unless it has explicitly defined semantics. Throw exceptions to signify errors rather than returning null. Make sure that any unexpected null reference will cause the code to throw an exception, either as a result of being dereferenced or by hitting an explicit assertion.

How to Refactor

Reflective design helps you understand what to change; refactoring gives you the ability to make those changes. When you refactor, proceed in a series of small transformations. (Confusingly, each type of transformation is also called a refactoring.) Each refactoring is like making a turn on a Rubik's cube. To achieve anything significant, you have to string together several individual refactorings, just as you have to string together several turns to solve the cube.

The fact that refactoring is a sequence of small transformations sometimes gets lost on people new to refactoring. Refactoring isn't rewriting. You don't just change the design of your code; to refactor well, you need to make that change in a series of controlled steps. Each step should only take a few moments, and your tests should pass after each one.

There are a wide variety of individual refactorings. Refactoring is the classic work on the subject. It contains an in-depth catalog of refactoring, and is well worth studying—I learned more about good code from reading that book than from any other.

You don't need to memorize all the individual refactorings. Instead, try to learn the mindset behind the refactorings. Work from the book in the beginning. Over time, you'll learn how to refactor intuitively, creating each refactoring as you need it.

Refactoring in Action

Any significant design change requires a sequence of refactorings. Learning how to change your design through a series of small refactorings is a valuable skill. Once you've mastered it, you can make dramatic design changes without breaking anything. You can even do this in pieces, by fixing part of the design one day and the rest of it another day.

To illustrate this point, I'll show each step in the simple refactoring from my TDD example. Note how small each step is. Working in small steps enables you to remain in control of the code, prevents confusion, and allows you to work very quickly. The purpose of this example was to create an HTTP query string parser. At this point, I had a working, bare-bones parse.

Here are the tests:

```
public class QueryStringTest extends TestCase {

    public void testOneNameValuePair() {
        QueryString query = new QueryString("name=value");
        assertEquals(1, query.count());
        assertEquals("value", query.valueFor("name"));
    }

    public void testMultipleNameValuePairs() {
        QueryString query = new QueryString("name1=value1&name2=value2&name3=value3");
        assertEquals(3, query.count());
        assertEquals("value1", query.valueFor("name1"));
        assertEquals("value2", query.valueFor("name2"));
        assertEquals("value3", query.valueFor("name3"));
    }

    public void testNoNameValuePairs() {
        QueryString query = new QueryString("");
        assertEquals(0, query.count());
    }

    public void testNull() {
        try {
            QueryString query = new QueryString(null);
            fail("Should throw exception");
        }
        catch (NullPointerException e) {
            // expected
        }
    }
}
```

The code worked—it passed all the tests—but it was ugly. Both the `count()` and `valueFor()` methods had duplicate parsing code. I wanted to eliminate this duplication and put parsing in just one place.

```

public class QueryString {
    private String _query;

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        _query = queryString;
    }

    public int count() {
        if ("".equals(_query)) return 0;
        String[] pairs = _query.split("&");
        return pairs.length;
    }

    public String valueFor(String name) {
        String[] pairs = _query.split("&");

        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            if (nameAndValue[0].equals(name)) return nameAndValue[1];
        }
        throw new RuntimeException(name + " not found");
    }
}

```

To eliminate the duplication, I needed a single method that could do all the parsing. The other methods would work with the results of the parse rather than doing any parsing of their own. I decided that this parser, called from the constructor, would parse the data into a `HashMap`.

Although I could have done this in one giant step by moving the body of `valueFor()` into a `parseQueryString()` method and then hacking until the tests passed again, I knew from hardwon experience that it was faster to proceed in small steps.

My first step was to introduce `HashMap()` into `valueFor()`. This would make `valueFor()` look just like the `parseQueryString()` method I needed. Once it did, I could extract out `parseQueryString()` easily

```

public String valueFor(String name) {
    HashMap<String, String> map = new HashMap<String, String>();

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        map.put(nameAndValue[0], nameAndValue[1]);
    }
    return map.get(name);
}

```

After making this refactoring, I ran the tests. They passed.

Now I could extract the parsing logic into its own method. I used my IDE's built-in Extract Method refactoring to do so.

```

public String valueFor(String name) {
    HashMap<String, String> map = parseQueryString();
    return map.get(name);
}

private HashMap<String, String> parseQueryString() {
    HashMap<String, String> map = new HashMap<String, String>();

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        map.put(nameAndValue[0], nameAndValue[1]);
    }
    return map;
}

```

The tests passed again, of course. With such small steps, I'd be surprised if they didn't. That's the point: by taking small steps, I remain in complete control of my changes, which reduces surprises. I now had a `parseQueryString()` method, but it was only available to `valueFor()`.

My next step was to make it available to all methods. To do so, I created a `_map` instance variable and had `parseQueryString()` use it.

```

public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

    ...

    public String valueFor(String name) {
        HashMap<String, String> map = parseQueryString();
        return map.get(name);
    }

    private HashMap<String, String> parseQueryString() {
        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
        return _map;
    }
}

```

This is a trickier refactoring than it seems. Whenever you switch from a local variable to an instance variable, the order of operations can get confused. That's why I continued to have `parseQueryString()` return `_map`, even though it was now available as an instance variable. I wanted to make sure this first step passed its tests before proceeding to my next step, which was to get rid of the unnecessary return.

```

public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

    ...

    public String valueFor(String name) {
        parseQueryString();
        return _map.get(name);
    }

    private void parseQueryString() {
        String[] pairs = _query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}

```

The tests passed again.

Because `parseQueryString()` now stood entirely on its own, its only relationship to `valueFor()` was that it had to be called before `valueFor()`'s return statement. I was finally ready to achieve my goal of calling `parseQueryString()` from the constructor.

```

public class QueryString {
    private String _query;
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        _query = queryString;
        parseQueryString();
    }

    ...

    public String valueFor(String name) {
        return _map.get(name);
    }

    ...

}

```

This seemed like a simple refactoring. After all, I moved only one line of code. Yet when I ran my tests, they failed. My parse method didn't work with an empty string—a degenerate case that I hadn't yet implemented in `valueFor()`. It wasn't a problem as long as only `valueFor()` ever called `parseQueryString()`, but it showed up now that I called `parseQueryString()` in the constructor.

The problem was easy to fix with a guard clause.

```
private void parseQueryString() {
    if ("".equals(_query)) return;

    String[] pairs = _query.split("&");
    for (String pair : pairs) {
        String[] nameAndValue = pair.split("=");
        _map.put(nameAndValue[0], nameAndValue[1]);
    }
}
```

At this point, I was nearly done. The duplicate parsing in the `count()` method caused all of this mess, and I was ready to refactor it to use the `_map` variable rather than do its own parsing. It went from:

```
public int count() {
    if ("".equals(_query)) return 0;
    String[] pairs = _query.split("&");
    return pairs.length;
}
```

to:

```
public int count() {
    return _map.size();
}
```

I love it when I can delete code.

I reviewed the code and saw just one loose end remaining: the `_query` instance variable that stored the unparsed query string. I no longer needed it anywhere but `parseQueryString()`, so I demoted it from an instance variable to a `parseQueryString()` parameter.

```
public class QueryString {
    private HashMap<String, String> _map = new HashMap<String, String>();

    public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
        parseQueryString(queryString);
    }

    public int count() {
        return _map.size();
    }

    public String valueFor(String name) {
        return _map.get(name);
    }

    private void parseQueryString(String query) {
        if ("".equals(query)) return;

        String[] pairs = query.split("&");
        for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
        }
    }
}
```

When you compare the initial code to this code, there's little in common. Yet this change took place as a series of small, careful refactorings. Although it took me a long time to describe the steps, each individual refactoring took a matter of seconds in practice. The

whole series occurred in a few minutes.

5. Explain about Incremental Design and Architecture.

XP makes challenging demands of its programmers: every week, programmers should finish 4 to 10 customer-valued stories. Every week, customers may revise the current plan and introduce entirely new stories—with no advance notice. This regimen starts with the first week of the project.

In other words, as a programmer you must be able to produce customer value, from scratch, in a single week. No advance preparation is possible. You can't set aside several weeks for building a domain model or persistence framework; your customers need you to deliver completed stories.

Fortunately, XP provides a solution for this dilemma: incremental design (also called evolutionary design) allows you to build technical infrastructure (such as domain models and persistence frameworks) incrementally, in small pieces, as you deliver stories.

How It Works

Incremental design applies the concepts introduced in test-driven development to all levels of design. Like test-driven development, programmers work in small steps, proving each before moving to the next. This takes place in three parts: start by creating the simplest design that could possibly work, incrementally add to it as the needs of the software evolve, and continuously improve the design by reflecting on its strengths and weaknesses. To be specific, when you first create a design element—whether it's a new method, a new class, or a new architecture—be completely specific. Create a simple design that solves only the problem you face at the moment, no matter how easy it may seem to solve more general problems.

This is difficult! Experienced programmers think in abstractions. In fact, the ability to think in abstractions is often a sign of a good programmer. Coding for one specific scenario will seem strange, even unprofessional.

Do it anyway. The abstractions will come. Waiting to make them will enable you to create designs that are simpler and more powerful.

The second time you work with a design element, modify the design to make it more general but only general enough to solve the two problems it needs to solve. Next, review the design and make improvements. Simplify and clarify the code.

The third time you work with a design element, generalize it further but again, just enough to solve the three problems at hand. A small tweak to the design is usually enough. It will be pretty general at this point. Again, review the design, simplify, and clarify.

Continue this pattern. By the fourth or fifth time you work with a design element be it a method, a class, or something bigger you'll typically find that its abstraction is perfect for

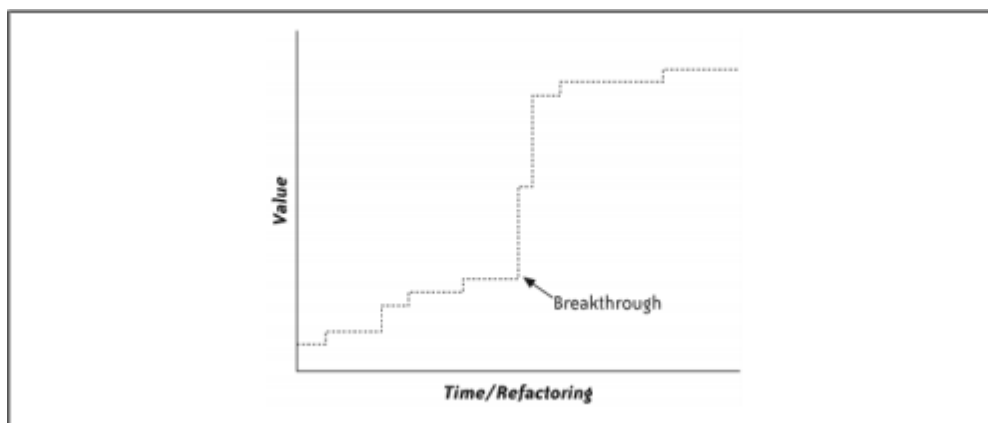
your needs. Best of all, because you allowed practical needs to drive your design, it will be simple yet powerful.

Continuous Design

Incremental design initially creates every design element method, class, namespace, or even architecture to solve a specific problem. Additional customer requests guide the incremental evolution of the design. This requires continuous attention to the design, albeit at different timescales. Methods evolve in minutes; architectures evolve over months. No matter what level of design you're looking at, the design tends to improve in bursts.

Typically, you'll implement code into the existing design for several cycles, making minor changes as you go. Then something will give you an idea for a new design approach, which will require a series of refactorings to support it. [Evans] calls this a breakthrough.

Breakthroughs happen at all levels of the design, from methods to architectures. Breakthroughs are the result of important insights and lead to substantial improvements to the design. (If they don't, they're not worth implementing.)



Breakthrough

Incrementally Designing Methods

You've seen this level of incremental design before: it's test-driven development. While the driver implements, the navigator think about the design. She looks for overly complex code and missing elements, which she writes on her note card. She thinks about which features the code should support next, what design changes might be necessary, and which tests may guide the code in the proper direction. During the refactoring step of TDD, both members of the pair look at the code, discuss opportunities for improvements, and review the navigator's notes.

Method refactoring happen every few minutes. Breakthroughs may happen several times

per hour and can take 10 minutes or more to complete.

Incrementally Designing Classes

When TDD is performed well, the design of individual classes and methods is beautiful: they're simple, elegant, and easy to use. This isn't enough. Without attention to the interaction between classes, the overall system design will be muddy and confusing. During TDD, the navigator should also consider the wider scope.

When you see a problem, jot it down on your card. During one of the refactoring steps of TDD usually, when you're not in the middle of something else—bring up the issue, discuss solutions with your partner, and refactor. If you think your design change will significantly affect other members of the team, take a quick break to discuss it around a whiteboard.

Class-level refactoring happen several times per day. Depending on your design, breakthroughs may happen a few times per week and can take several hours to complete. Use your iteration slack to complete breakthrough refactoring. In some cases, you won't have time to finish all the refactoring you identify. That's OK; as long as the design is better at the end of the week than it was at the beginning, you're doing enough.

Incrementally Designing Architecture

Large programs use overarching organizational structures called architecture. For example, many programs segregate user interface classes, business logic classes, and persistence classes into their own namespaces; this is a classic three-layer architecture. Other designs have the application pass the flow of control from one machine to the next in an n-tier architecture.

These architectures are implemented through the use of recurring patterns.

They aren't design patterns in the formal Gang of Four* sense. Instead, they're standard conventions specific to your codebase. For example, in a three-layer architecture, every business logic class will probably be part of a "business logic" namespace, may inherit from a generic "business object" base class, and probably interfaces with its persistence layer counterpart in a standard way.

These recurring patterns embody your application architecture. Although they lead to consistent code, they're also a form of duplication, which makes changes to your architecture more difficult. Fortunately, you can also design architectures incrementally. As with other types of continuous design, use TDD and pair programming as your primary vehicle.

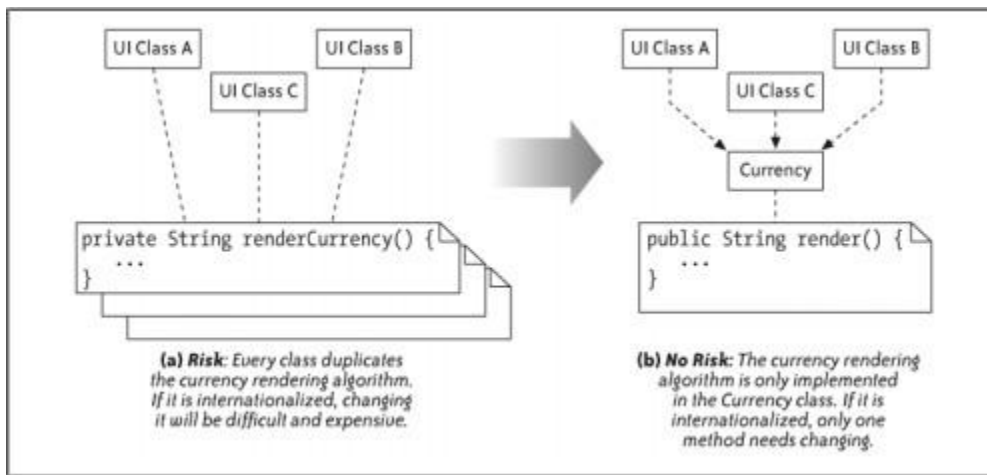
While your software grows, be conservative in introducing new architectural patterns: introduce just what you need to for the amount of code you have and the features you support at the moment.

Before introducing a new pattern, ask yourself if the duplication is really necessary. Perhaps there's some language feature you can use that will reduce your need to rely on the pattern. In my experience, breakthroughs in architecture happen every few months. (This estimate will vary widely depending on your team members and code quality.) Refactoring to support the breakthrough can take several weeks or longer because of the amount of duplication involved. Although changes to your architecture may be tedious, they usually aren't difficult once you've identified the new architectural pattern. Start by trying out the new pattern in just one part of your design. Let it sit for a while—a week or two—to make sure the change works well in practice.

Once you're sure it does, bring the rest of the system into compliance with the new structure. Refactor each class you touch as you perform your everyday work, and use some of your slack in each iteration to fix other classes. Keep delivering stories while you refactor. Although you could take a break from new development to refactor, that would disenfranchise your customers. Balance technical excellence with delivering value. Neither can take precedence over the other. This may lead to inconsistencies within the code during the changeover, but fortunately, that's mostly an aesthetic problem—more annoying than problematic.

Risk-Driven Architecture

Architecture may seem too essential not to design up-front. Some problems do seem too expensive to solve incrementally, but I've found that nearly everything is easy to change if you eliminate duplication and embrace simplicity. Common thought is that distributed processing, persistence, internationalization, security, and transaction structure are so complex that you must consider them from the start of your project. I disagree; I've dealt with all of them incrementally. Of course, no design is perfect. Even with simple design, some of your code will contain duplication, and some will be too complex. There's always more refactoring to do than time to do it. That's where risk-driven architecture comes in. Although I've emphasized designing for the present, it's OK to think about future problems. Just don't implement any solutions to stories that you haven't yet scheduled.



Use risk to drive refactoring

To apply risk-driven architecture, consider what it is about your design that concerns you and eliminate duplication around those concepts. For example, if your internationalization concern is that you always format numbers, dates, and other variables in the local style, look for ways to reduce duplication in your variable formatting. One way to do so is to make sure every concept has its own class (as described in “Once and Only Once” earlier in this chapter), then condense all formatting around each concept into a single method within each class, as shown in Figure.

If there’s still a lot of duplication, the Strategy pattern would allow you to condense the formatting code even further. Limit your efforts to improving your existing design. Don’t actually implement support for localized formats until your customers ask for them. Once you’ve eliminated duplication around a concept—for example, once there’s only one method in your entire system that renders numbers as strings—changing its implementation will be just as easy later as it is now.

6. Write about SPIKE SOLUTIONS.

You’ve probably noticed by now that XP values concrete data over speculation. Whenever you’re faced with a question, don’t speculate about the answer—conduct an experiment! Figure out how you can use real data to make progress.

That’s what spike solutions are for, too.

About Spikes

A spike solution, or spike, is a technical investigation. It’s a small experiment to research the answer to a problem. For example, a programmer might not know whether Java throws

an exception on arithmetic overflow. A quick 10-minute spike will answer the question:

```
public class ArithmeticOverflowSpike {
    public static void main(String[] args) {
        try {
            int a = Integer.MAX_VALUE + 1;
            System.out.println("No exception: a = " + a);
        }

        catch (Throwable e) {
            System.out.println("Exception: " + e);
        }
    }
}

No exception: a = -2147483648
```

Performing the Experiment

The best way to implement a spike is usually to create a small program or test that demonstrates the feature in question. You can read as many books and tutorials as you like, but it's my experience that nothing helps me understand a problem more than writing working code. It's important to work from a practical point of view, not just a theoretical one. Writing code, however, often takes longer than reading a tutorial. Reduce that time by writing small, standalone programs.

You can ignore the complexities necessary to write production code—just focus on getting something working. Run from the command line or your test framework. Hardcode values. Ignore user input, unless absolutely necessary.

I often end up with programs a few dozen lines long that run almost everything from main(). Of course, this approach means you can't reuse this code in your actual production codebase, as you didn't develop it with all your normal discipline and care. That's fine. It's an experiment. When you finish, throw it away, check it in as documentation, or share it with your colleagues, but don't treat it as anything other than an experiment.

Scheduling Spikes

Most spikes are performed on the spur of the moment. You see a need to clarify a small technical issue, and you write a quick spike to do so. If the spike takes more than a few minutes, your iteration slack absorbs the cost. If you anticipate the need for a spike when estimating a story, include the time in your story estimate.

7. Write about PERFORMANCE OPTIMIZATION.

Our organization had a problem.* Every transaction our software processed had a three- second latency. During peak business hours, transactions piled up and with our recent surge in sales, the lag sometimes became hours. We cringed every time the phone rang; our customers were upset.

We knew what the problem was: we had recently changed our order preprocessing code. I remember thinking at the time that we might need to start caching the intermediate results of expensive database queries. I had even asked our customers to schedule a performance story. Other stories had been more important, but now performance was top priority.

I checked out the latest code and built it. All tests passed, as usual. Carlann suggested that we create an end-to-end performance test to demonstrate the problem. We created a test that placed 100 simultaneous orders, then ran it under our profiler. The numbers confirmed my fears: the average transaction took around 3.2 seconds, with a standard deviation too small to be significant. The program spent nearly all that time within a single method: `verify_order_id()`.

We started our investigation there. I was pretty sure a cache was the right answer, but the profiler pointed to another possibility. The method retrieved a list of active order IDs on every invocation, regardless of the validity of the provided ID. Carlann suggested discounting obviously flawed IDs before testing potentially valid ones, so I made the change and we reran the tests.

All passed. Unfortunately, that had no effect on the profile. We rolled back the change. Next, we agreed to implement the cache. Carlann coded a naïve cache. I started to worry about cache coherency, so I added a test to see what happened when a cached order went active. The test failed. Carlann fixed the bug in the cache code, and all tests passed again.

How to Optimize

Modern computers are complex. Reading a single line of a file from a disk requires the coordination of the CPU, the kernel, a virtual file system, a system bus, the hard drive controller, the hard drive cache, OS buffers, system memory, and scheduling pipelines. Every component exists to solve a problem, and each has certain tricks to squeeze out performance. Is the data in a cache? Which cache? How's your memory aligned? Are you reading asynchronously or are you blocking? There are so many variables it's nearly impossible to predict the general performance of any single method. The days in which a programmer could accurately predict performance by counting instruction clock cycles are long gone, yet some still approach performance with a simplistic, brute-force mindset.

They make random guesses about performance based on 20 -line test programs, flail around while writing code the first time, leave a twisty mess in the real program, and then take a long lunch. Sometimes that even works. More often, it leaves a complex mess that doesn't benefit overall performance. It can actually make your code slower.

A holistic approach is the only accurate way to optimize such complex systems. Measure the performance of the entire system, make an educated guess about what to change, then remeasure. If the performance gets better, keep the change. If it doesn't, discard it. Once your performance test passes, stop—you're done.

Look for any missed refactoring opportunities and run your test suite one more time. Then integrate. Usually, your performance test will be an end-to-end test. Although I avoid end- to-end tests in other situations (because they're slow and fragile—see “Test-Driven Development” earlier in this chapter), they are often the only accurate way to reproduce real-world performance conditions. You

may be able to use your existing testing tool, such as xUnit, to write your performance tests.

Sometimes you get better results from a specialized tool. Either way, encode your performance criteria into the test. Have it return a single, unambiguous pass/fail result as well as performance statistics. If the test doesn't pass, use the test as input to your profiler. Use the profiler to find the bottlenecks, and focus your efforts on reducing them.

Although optimizations often make code more complex, keep your code as clean and simple as possible. If you're adding new code, such as a cache, use test-driven development to create that code. If you're removing or refactoring code, you may not need any new tests, but be sure to run your test suite after each change.

When to Optimize

Optimization has two major drawbacks: it often leads to complex, buggy code, and it takes time away from delivering features. Neither is in your customer's interests. Optimize only when it serves a real, measurable need. That doesn't mean you should write stupid code. It means your priority should be code that's clean and elegant.

Once a story is done, if you're still concerned about performance, run a test. If performance is a problem, fix it—but let your customer make the business decision about how important that fix is. XP has an excellent mechanism for prioritizing customer needs: the combination of user stories and release planning. In other words, schedule performance optimization just like any other customer-valued work: with a story.

Of course, customers aren't always aware of the need for performance stories, especially not ones with highly technical requirements. If you have a concern about potential performance problems in part of the system, explain your concern in terms of business tradeoffs and risks. They still might not agree. That's OK—they're the experts on business value and priorities.

It's their responsibility, and their decision. Similarly, you have a responsibility to maintain an efficient development environment. If your tests start to take too long, go ahead and optimize until you meet a concrete goal, such as five or ten minutes. Keep in mind that the most common cause of a slow build is too much emphasis on end-to-end tests, not slow

code.

8. Describe in detailed about EXPLORATORY TESTING.

XP teams have no separate QA department. There's no independent group of people responsible for assessing and ensuring the quality of the final release. Instead, the whole team—customers, programmers, and testers—is responsible for the outcome. On traditional teams, the QA group is often rewarded for finding bugs. On XP teams, there's no incentive program for finding or removing bugs. The goal in XP isn't to find and remove bugs; the goal is not to write bugs in the first place. In a well-functioning team, bugs are a rarity—only a handful per month.

Does that mean testers have no place on an XP team? No! Good testers have the ability to look at software from a new perspective, to find surprises, gaps, and holes. It takes time for the team to learn which mistakes to avoid. By providing essential information about what the team overlooks, testers enable the team to improve their work habits and achieve their goal of producing zero bugs.

One particularly effective way of finding surprises, gaps, and holes is exploratory testing: a style of testing in which you learn about the software while simultaneously designing and executing tests, using feedback from the previous test to inform the next. Exploratory testing enables you to discover emergent behavior, unexpected side effects, holes in the implementation (and thus in the automated test coverage), and risks related to quality attributes that cross story boundaries such as security, performance, and reliability.

It's the perfect complement to XP's raft of automated testing techniques. Exploratory testing predates XP. [Kaner] coined the term in the book *Testing Computer Software*, although the practice of exploratory testing certainly preceded the book, probably by decades. Since the book came out, people such as Cem Kaner, James and Jonathan Bach, James Lyndsay, Jonathan Kohl, and Elisabeth Hendrickson have extended the concept of exploratory testing into a discipline.

About Exploratory Testing

Philosophically, exploratory testing is similar to test-driven development and incremental design: rather than designing a huge suite of tests up-front, you design a single test in your head, execute it against the software, and see what happens. The result of each test leads you to design the next, allowing you to pursue directions that you wouldn't have anticipated if you had attempted to design all the tests up-front. Each test is a little experiment that investigates the capabilities and limitations of the emerging software.

Exploratory testing can be done manually or with the assistance of automation. Its defining characteristic is not how we drive the software but rather the tight feedback loop between test design, test execution, and results interpretation.

Exploratory testing works best when the software is ready to be explored— that is, when stories are “done done.” You don’t have to test stories that were finished in the current iteration. It’s nice to have that sort of rapid feedback, but some stories won’t be “done done” until the last day of the iteration. That’s OK—remember, you’re not using exploratory testing to guarantee quality; you’re using it provide information about how the team’s process guarantees quality.

Tool #1: Charters

Some people claim that exploratory testing is simply haphazard poking at the software under test. This isn’t true, any more than the idea that American explorers Lewis and Clark mapped the Northwest by haphazardly tromping about in the woods. Before they headed into the wilderness, Lewis and Clark knew where they were headed and why they were going. President Thomas Jefferson had given them a charter:*

The Object of your mission is to explore the Missouri river & such principal stream of it as by [its] course and communication with the waters of the Pacific ocean, whether the Columbia, Oregon, Colorado or any other river may offer the most direct & practicable water communication across this continent for the purpose of commerce.

Similarly, before beginning an exploratory session, a tester should have some idea of what to explore in the system and what kinds of things to look for. This charter helps keep the session focused.

Tool #2: Observation

Automated tests only verify the behavior that programmers write them to verify, but humans are capable of noticing subtle clues that suggest all is not right. Exploratory testers are continuously alert for anything out of the ordinary. This may be an editable form field that should be read-only, a hard drive that spun up when the software should not be doing any disk access, or a value in a report that looks out of place.

Tool #3: Notetaking

While exploring, it’s all too easy to forget where you’ve been and where you’re going. Exploratory testers keep a notepad beside them as they explore and periodically take notes on the actions they take. You can also use screen recorders such as Camtasia to keep track of what you do. After all, it’s quite frustrating to discover a bug only to find that you have

no idea what you did to cause the software to react that way. Notes and recordings tell you what you were doing not just at the time you encountered surprising behavior but also in the minutes or hours before.

Tool #4: Heuristics

A heuristic is a guide: a technique that aids in your explorations. Boundary testing is an example of a test heuristic. Experienced testers use a variety of heuristics, gleaned from an understanding of how software systems work as well as from experience and intuition about what causes software to break. You can improve your exploratory efforts by creating and maintaining a catalog of heuristics that are worth remembering when exploring your software. Of course, that same catalog can also be a welcome reference for the programmers as they implement the software.

R16

Code No: 138DK

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B. Tech IV Year II Semester Examinations, September - 2020 MODERN

SOFTWARE ENGINEERING

(Common to CSE, IT)

Time: 2 Hours

Max. Marks: 75

Answer any Five Questions All
Questions Carry Equal Marks

1. Appraise the importance of organizational success. [15]
- 2.a) How to hold a daily standup meeting?
b) What is iteration demo? How does it help the team? [7+8]
- 3.a) How to prevent security defects and other challenging bugs?
b) Code goes through four levels of completion. Discuss these levels. [8+7]
- 4.a) Discuss ways to introduce slack into iterations. Explain
b) combining stories with illustration. [7+8]
5. Explain the life cycle of test driven development with suitable example. [15]
6. XP teams are self organized. Support this statement. [15]
- 7.a) Contrast in-house custom development with outsourced custom development.
b) How to deal with disagreement regarding coding standards in the team? [8+7]
8. Is asynchronous integration more efficient than synchronous integration? Substantiate your answer. [15]

---ooOoo---

Code No: 138DK**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD****B. Tech IV Year II Semester Examinations, December - 2020****MODERN SOFTWARE ENGINEERING****(Common to CSE, IT)****Time: 2 Hours****Max. Marks: 75****Answer any Five Questions All
Questions Carry Equal Marks**

- - -

1. State and explain the need of Agile methodology process mode. [15]
2. Discuss about Practicing XP and Root cause analysis for extreme programming. [15]
3. Explain about coding standards and collective ownership in software process development model. [15]
4. Discuss the role of collaboration in software development, during iteration demo and reporting process. [15]
5. Discuss the significance of collective ownership. Explain the importance of Real customer involvement. [15]
6. Explain about Risk management and estimation. [15]
7. Discuss about spike solutions and Performance optimization. [15]
8. Explain about Incremental design and architecture. [15]

---ooOoo---