UNIT - I: Introduction

Introduction to agile, understanding XP - The XP lifecycle, The XP team, XP concepts. Thinking - pair programming, root-cause analysis.

Introduction to Agile

- Agile is an iterative and incremental approach to project management and software development.
- It emphasizes flexibility, collaboration, and continuous improvement.
- Agile methods prioritize customer satisfaction and early delivery of working software.
- The Agile Manifesto was written in 2001 by a group of software development experts.
- On February 17th 2001, software developers (nearly 17 members) published the manifesto of agile software development to define approach.
- "We are uncovering better ways of developing software by doing it and helping others do it."

It outlines four key values that agile teams should prioritize:

1. Individuals and interactions over processes and tools.
2. Working Software over Comprehensive Documentation
3. Customer Collaboration over Contract Negotiation
4. Responding to Change over Following a Plan

There are some limitations of the existing process models, such as

- Schedule slips
- Business misunderstanding
- High defect rate
- Project failure
- Requirements changes and so on.

To overcome the above problems, the agile process model was introduced.

Agile Principles

Principle 1: Our Highest Priority is to **Satisfy the Customer** through **Early** and **Continuous Delivery** of Valuable Software

- Number of practices has significant impact upon quality of final system:
- Strong **correlation** between **quality** and **early delivery of a partially functioning system.**
- The less functional the initial delivery, the higher the quality of the final delivery.
- Another strong **correlation** exists between **final quality** and **frequently deliveries of increasing functionality**.

- The more frequent the deliveries, the higher the final quality.

Agile processes deliver early and often.

- Elementary system **first** followed by systems of **increasing functionality** every few weeks.
- Customers may use these systems in production, or
- May choose to review existing functionality and report on changes to be made.
- Regardless, they must provide meaningful **feedback**.

Principle 2: Welcome Changing Requirements, even late in Development. Agile Processes harness change for the Customer's Competitive Advantage.

Agile teams work to keep the **software structure** flexible, so requirement change impact is minimal. **Principles of object oriented** design help us to maintain this kind of flexibility.

Principle 3: Deliver Working Software Frequently
(From a couple of weeks to a couple of months with a preference to the shorter time scale)

- We deliver working software.
  - **Deliver early and often**.
  - **Be not content** with delivering bundles of **documents, or plans**.
  - Don't count those as true deliverables.
- The **goal** of delivering software that satisfies the customer's needs.

Principle 4: Business People and Developers Must **Work Together Daily** throughout the Project.

For agile projects, there must be **significant** and **frequent interaction** between the

- customers,
- developers, and
- Stakeholders.

An agile project must be **continuously guided**.

Principle 5: Build Projects around Motivated Individuals. (Give them the environment and support they need, and trust them to get the job done.)

An agile project has people the most important factor of success.

- All other factors, process, environment, management, etc., are considered to be second order effects, and are subject to change if they are having an adverse effect upon the people.

Example:

If the office environment is an obstacle to the team, **change the office environment**.

If certain process steps are obstacles to the team, **change the process steps.**

Principle 6: The Most Efficient and Effective Method of Conveying Information to and within a Development Team is **face-to-face Communications**.

- In agile projects, developers ***talk*** to each other.
  - The primary **mode of communication is conversation**.
  - Documents may be created, but there is no attempt to capture all project information in writing.

Principle 7: Working Software is the Primary Measure of Progress

Agile projects measure their progress by measuring the amount of **working software**.

- Progress **not measured** by phase we are in, **or**
- by the volume of produced documentation **or**
- By the amount of code they have created.

- **Agile teams are 30% done when 30% of the necessary functionality is working.**

Principle 8: Agile Processes promote sustainable development: The sponsors, developers, and users should be able to **maintain a constant speed** indefinitely.

An agile project is not run like a 50 yard dash; it is run like a marathon.

- The team does not take off at full speed and try to maintain that speed for the duration.
- Rather they run at a fast, but **sustainable**, speed.

Principle 9: Continuous Attention to Technical Excellence and Good Design enhances Agility.

High quality is the key to high speed.

- The way to go fast is to **keep the software as clean and robust as possible.**
- Thus, all agile team-members are **committed** to producing only the **highest quality code** they can.

Principle 10: **Simplicity**

Agile teams take the simplest path that is consistent with their goals.

- **They don't anticipate tomorrow's problems and try to defend against them today.**

Principle 11: **the Best Architectures, Requirements, and Designs emerge from Self-Organizing Teams**

An agile team is a self organizing team.

- Responsibilities are **communicated** to the team as a whole, and the **team determines** the best way to fulfill them.

Agile team members work together on all project aspects.

- Each is allowed input into the whole.
- No single team member is responsible for the architecture, or the requirements, or the tests, etc.
- The team shares those responsibilities and each team member has influence over them.

Principle 12: At regular Intervals, the **Team reflects** on how to become **more** effective, **then tunes and adjusts** its **behavior** accordingly.

An agile team continually adjusts its organization, rules, conventions, relationships, etc.

An agile team knows that its environment is continuously changing, and knows that they must change with that environment to remain agile.

The professional goal of every software engineer, and every development team, is to deliver the highest possible value to our employers and customers.

Why agile

Agile development is important because it helps to ensure that development teams complete projects on time and within budget. It also helps to improve communication between the development team and the product owner.

Additionally, agile development methodology can help reduce the risks associated with complex projects. It allows for development teams to make changes quickly and easily without affecting the overall project timeline.

What Are the Benefits of Agile Development Methodology?

There are many benefits of agile development methodology, some of which include:

- **Increased flexibility**: Agile development is more flexible than other project management methodologies. Development teams can make changes on the fly more easily.
- **Improved communication**: Agile development helps to improve communication between the development team and the product owner. Because of this, there is a greater focus on collaboration and feedback.
- **Reduced risks**: Agile development can help to reduce the risks associated with complex projects. By breaking down complex projects into smaller sprints, project managers can dissect them and achieve shareholder demands.

- **Increased customer satisfaction**: Agile development environments often lead to increased customer satisfaction. This is because the customer is involved in the development process and provides feedback at each stage of the project.

The Agile methodology is a project management approach that involves breaking the project into phases and emphasizes continuous collaboration and improvement. Teams follow a cycle of planning, executing, and evaluating.

Agile Methods

Several agile manifesto s/w development methods have come into practice since the inception.

- The most popular methods are
  - **Extreme programming (XP)**
  - Scrum
  - Dynamic systems development method (DSDM)
  - Adaptive software development (ASD)
  - Crystal
  - Feature-driven development (FDD)
  - Test- driven development (TDD)
  - Pair programming
  - Refactoring
  - Agile modeling
  - Internet speed development and so on.

Extreme Programming (XP)

This is a typical agile development framework, developed by Kent Beck, and can be adapted to development companies of various dimensions.

Extreme Programming ("XP") methodology is based around the idea of discovering "the simplest thing that will work" without putting too much weight on the long-term product view.

It is a methodology that emphasizes values such as Communication, Simplicity, Feedback, Courage and Respect, and prioritizes customer satisfaction over everything else. This methodology encourages trust by motivating developers to accept changes in customer requirements, even if they arrive during the latter stages of the development cycle.

Teamwork is extremely important in XP, since, when there is a problem, it is solved by the whole team of managers, developers or customers, bringing them together to promote conversation and engagement and break down barriers to communication. They all become essential pieces of the same puzzle, creating a fertile environment for high productivity and efficiency within teams. In Extreme

Programming, the software is tested from day one, collecting feedback to improve development. XP promotes activities such as pair programming, and with a strong testing component, it's an excellent engineering methodology.

- It improves software development through 4 principles
  - Simplicity
  - Feedback
  - Respect and
  - courage

Advantages:

- The simplicity of the written code is an advantage since it allows for improvement at any given time;
- The whole process and the XP development cycle is visible, creating goals for developers along with relatively rapid results;
- Software development is more agile than when using other methodologies, due to constant testing;
- Promotes a highly energising way of working;
- XP also contributes to uplifting and maintaining team talent.

Disadvantages:

- The extreme focus on code can lead to less importance being paid to design, meaning that it has to get extra attention later;
- This framework may not work at its best if all the team members are not situated in the same geographical area;
- In XP projects, a registry of possible errors is not always maintained, and this lack of monitoring can lead to similar bugs in the future.

Scrum

Scrum is, undoubtedly, the most used of the many frameworks underpinning Agile methodology. Scrum is characterized by cycles or stages of development, known as sprints, and by the maximization of development time for a software product towards a goal, the Product Goal. This Product Goal is a larger value objective, in which sprints bring the scrum team product a step closer.

It is usually used in the management of the development of software products but can be used successfully in a business-related context.

Every day starts with a small 15-minute meeting, the daily Scrum, which takes the role of synchronising activities and finding the best way to plan out the working day, allowing for a check on sprint "health" and product progress.

Advantages:

- Team motivation is good because programmers want to meet the deadline for every sprint;
- Transparency allows the project to be followed by all the members in a team or even throughout the organization;
- A simple "definition of done" is used for validating requirements
- Focus on quality is a constant with the scrum method, resulting in fewer mistakes;
- The dynamics of this method allow developers to reorganize priorities, ensuring that sprints that have not yet been completed get more attention;
- Good sprint planning is prioritized, so that the whole scrum team understands the "why, what and how" of allocated tasks.

Disadvantages:

- The segmentation of the project and the search for the agility of development can sometimes lead the team to lose track of the project as a whole, focusing on a single part;
- Every developer role may not be well defined, resulting in some confusion amongst team members.

Understanding XP - The XP Lifecycle

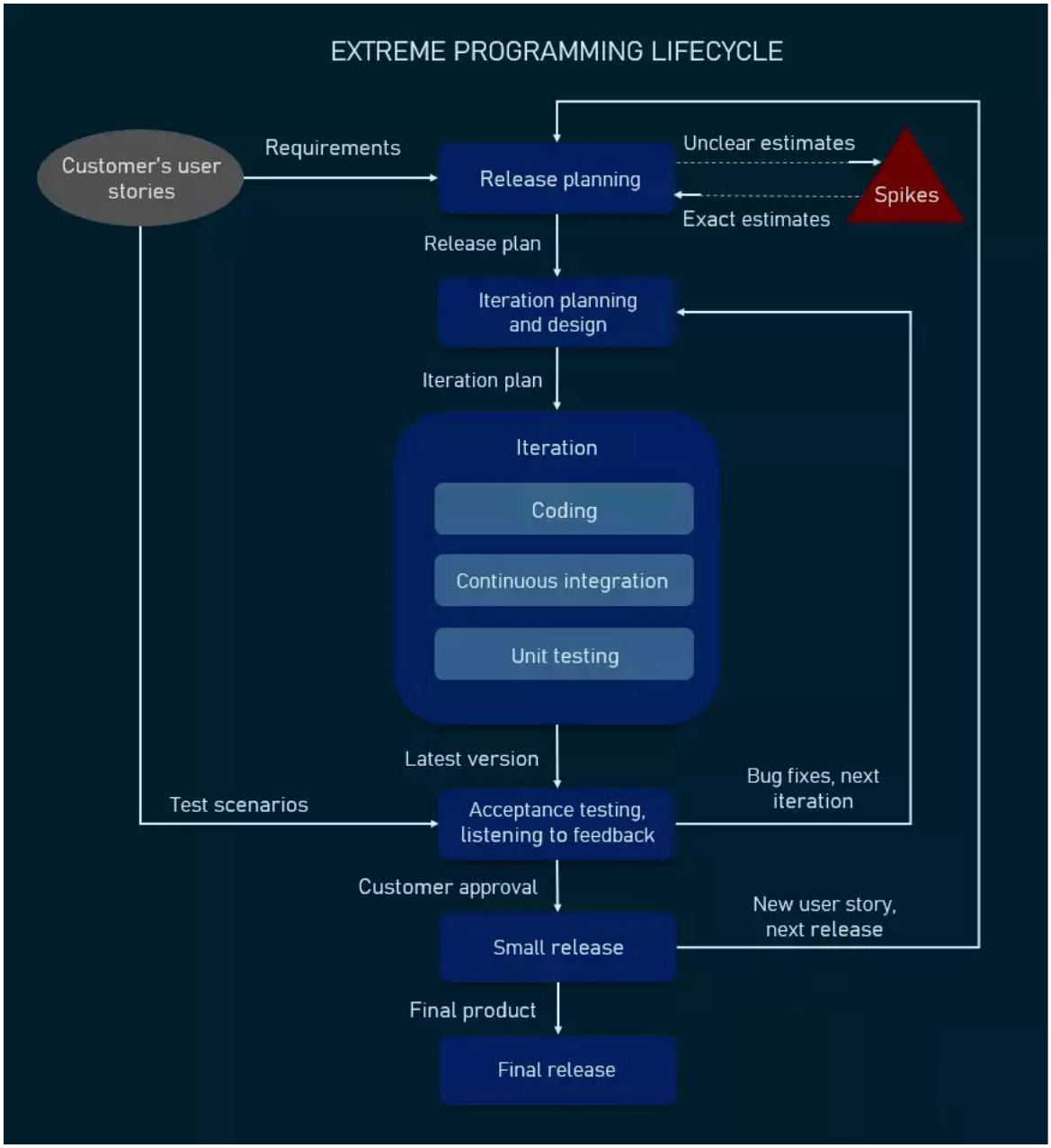
XP is a set of engineering practices. Developers have to go beyond their capabilities while performing these practices. That's where the "extreme" in the framework's title comes from. To get a better understanding of these practices, we'll start with describing XP's lifecycle and the roles engaged in the process.

The process and roles of extreme programming

The XP framework normally involves 5 phases or stages of the development process that iterate continuously:

1. **Planning**, the first stage is when the customer meets the development team and presents the requirements in the form of user stories to describe the desired result. The team then estimates the stories and creates a release plan broken down into iterations needed to cover the required functionality part after part. If one or more of the stories can't be estimated, so-called *spikes* can be introduced which means that further research is needed.
2. **Designing** is actually a part of the planning process, but can be set apart to emphasize its importance. It's related to one of the main XP values that we'll discuss below — simplicity. A good design brings logic and structure to the system and allows to avoid unnecessary complexities and redundancies.

3. **Coding** is the phase during which the actual code is created by implementing specific XP practices such as coding standards, pair programming, continuous integration, and collective code ownership (the entire list is described below).
4. **Testing** is the core of extreme programming. It is the regular activity that involves both unit tests (automated testing to determine if the developed feature works properly) and acceptance tests (customer testing to verify that the overall system is created according to the initial requirements).
5. **Listening** is all about constant communication and feedback. The customers and project managers are involved to describe the business logic and value that is expected.



The XP team

Extreme programming puts people in the center of the system, emphasizing the value and importance of such social skills as communication, cooperation, responsiveness, and feedback. So, these roles are commonly associated with XP:

Roles in Extreme Programming

The Roles that have been found effective in Extreme Programming are –

- **Developer (also called Programmer by some teams)**
- **Customer**
- **Manager (also called tracker)**
- **Coach**

Developer

The role of developer is the most important one in Extreme Programming. To be a developer in Extreme Programming, you need to be familiar with the following −

- Developer Rights
    - You have the right to know what is needed, with clear declarations of priority.
    - You have the right to produce quality work at all times.
    - You have the right to ask for and receive help from peers, superiors, and customers.
    - You have the right to make and update your own estimates.
    - You have the right to accept your responsibilities instead of having them assigned to you.
- Major responsibilities that you will be accountable for −
    - Estimate stories
    - Define tasks from stories
    - Estimate tasks
    - Write unit tests
    - Write code to pass the written unit tests
    - Perform unit testing
    - Refactor

    - Integrate continuously
- Developer Skills
    - Pair Programming
    - Communication – that is necessary and to the sufficient detail
    - Always use the metaphor to use the right names
    - Code only what is required.
    - Maintain simplicity
- Collective Ownership
    - If someone changes the code that you wrote, in whatever part of the system, you have to trust the changes and learn. In case the changes are wrong-headed, you are responsible for making things better, but be careful not to blame anyone.
    - Be prepared to acknowledge your fears. Remember that you are part of a team and courage is required for the success of Extreme Programming.
- Wear different hats as a developer in the team such as −
    - Programmer.
    - Architect and designer.
    - Interface architect/UI designer.
    - Database designer and DBA.
    - Operations and network designer.
- Sometimes one of the developers has to wear the hat of a tester.
    - Help the customer choose and write functional tests.
    - Run the functional tests regularly.
    - Report the test results.
    - Ensure the testing tools run well.

Customer

In Extreme Programming, the customer role is as crucial as the developer role as it is the customer who should know what to program, while the developer should know how to program.

This triggers the necessity of certain skills for the customer role −

- Writing required stories to the necessary and sufficient detail.

- Developing an attitude for the success of the project.
- Influencing a project without being able to control it.
- Making decisions that are required time to time on the extent of functionality that is required.
- Willing to change decisions as the product evolves.
- Writing functional tests.

In Extreme Programming, the customer is required to be in constant communication with the team and speak as a single voice to the team. This is required since −

- Customer could be multiple stakeholders.
- Customer could be a community.
- Customer is not always the PRINCIPAL (proxies).
- Customer can be a team with the following potential members −
    - Product Managers
    - Marketing, Sales
    - Business Analysts
    - End Users, their Managers
    - Business/System Operations

The Major Responsibilities of the Customer are −

- Write user stories
- Write functional tests
- Set priorities on the stories
- Explain stories
- Decide questions about the stories

The customer is accountable for these responsibilities.

Manager

In Extreme Programming, the major responsibilities of the manager are −

- Define the rules of planning game.
- Familiarize the team and the customer on the rules of the planning game.
- Monitor the planning game, fix any deviations, modify the rules if and when required.
- Schedule and conduct release planning and iteration planning meetings.

- Participate with the team while they estimate in order to provide feedback on how reality conformed to their prior estimates, both at team level and individual level that eventually help them to come up with better estimates next time.
- Ensure that the team is working towards the next release as they progress through the iterations with committed schedule and committed functionality.
- Track functional testing defects.
- Track the actual time spent by each team member.
- Adapt an ability to get all the required information without disturbing the team's work. The Manager is Accountable for these Responsibilities.

Coach

Extreme Programming is the responsibility of everyone in the team. However, if the team is new to Extreme Programming, the role of a coach is crucial.

The responsibilities of the coach are −

- Understand, in depth, the application of Extreme Programming to the project.
- Identify the Extreme Programming practices that help in case of any problem.
- Remain calm even when everyone else is panicking.
- Observe the team silently and intervene only when a significant problem is foreseen and make the team also see the problem.
- See to that the team is self-dependent.
- Be ready to help.

Extreme programming consists of small teams of 2 to 12 people.

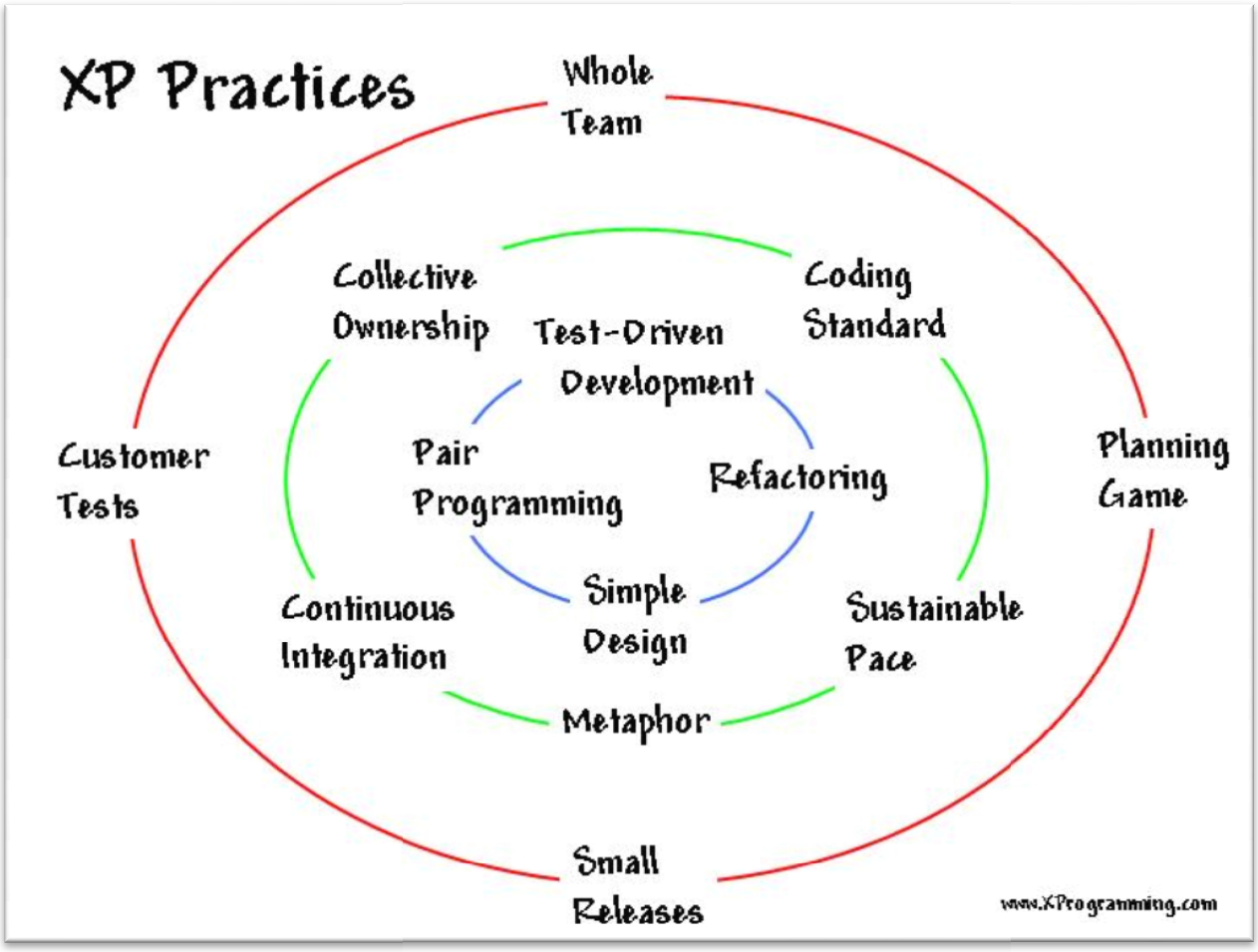
XP concepts

Whole Team

All the contributors to an XP project sit together, members of one team. This team must include a business representative. The "Customer" – who provides the requirements, sets the priorities, and steers the project. Its best if the Customer or one of her aides is a real end user who knows the domain and what is needed. The team will of course have programmers. The team may include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach, who helps the team keep on track, and facilitates the process. There may be a manager, providing resources, handling external communication, coordinating activities. None of these roles is necessarily the exclusive property of just one individual: Everyone on an XP team contributes in any way that they can. The best teams have no specialists, only general contributors with special skills.

Planning Game

XP planning addresses two key questions in software development: predicting what will be accomplished by the due date, and determining what to do next. The emphasis is on steering the project – which is quite straightforward – rather than on exact prediction of what will be needed and how long it will take – which is quite difficult. There are two key planning steps in XP, addressing these two questions:



Release Planning is a practice where the Customer presents the desired features to the programmers, and the programmers estimate their difficulty. With the cost estimates in hand, and with knowledge of the importance of the features, the Customer lays out a plan for the project. Initial release plans are necessarily imprecise: neither the priorities nor the estimates are truly solid, and until the team begins to work, we won't know just how fast they will go. Even the first release plan is accurate enough for decision making, however, and XP teams revise the release plan regularly.

Iteration Planning is the practice whereby the team is given direction every couple of weeks. XP teams build software in two-week "iterations", delivering running useful software at the end of each iteration. During Iteration Planning, the Customer presents the features desired for the next two weeks. The programmers break them down into tasks, and estimate their cost (at a finer level of detail than in

Release Planning). Based on the amount of work accomplished in the previous iteration, the team signs up for what will be undertaken in the current iteration.

These planning steps are very simple, yet they provide very good information and excellent steering control in the hands of the Customer. Every couple of weeks, the amount of progress is entirely visible. There is no “ninety percent done” in XP: a feature story was completed, or it was not. This focus on visibility results in a nice little paradox: on the one hand, with so much visibility, the Customer is in a position to cancel the project if progress is not sufficient. On the other hand, progress is so visible, and the ability to decide what will be done next is so complete, that XP projects tend to deliver more of what is needed, with less pressure and stress.

Customer Tests

As part of presenting each desired feature, the XP Customer defines one or more automated acceptance tests to show that the feature is working. The team builds these tests and uses them to prove to themselves, and to the customer, that the feature is implemented correctly. Automation is important because in the press of time, manual tests are skipped. That’s like turning off your lights when the night gets darkest.

The best XP teams treat their customer tests the same way they do programmer tests: once the test runs, the team keeps it running correctly thereafter. This means that the system only improves, always notching forward, never backsliding.

Small Releases

XP teams practice small releases in two important ways:

First, the team releases running, tested software, delivering business value chosen by the Customer, every iteration. The Customer can use this software for any purpose, whether evaluation or even release to end users (highly recommended). The most important aspect is that the software is visible, and given to the customer, at the end of every iteration. This keeps everything open and tangible.

Second, XP teams release to their end users frequently as well. XP Web projects release as often as daily, in house projects monthly or more frequently. Even shrink-wrapped products are shipped as often as quarterly.

It may seem impossible to create good versions this often, but XP teams all over are doing it all the time. See Continuous Integration for more on this, and note that these frequent releases are kept reliable by XP’s obsession with testing, as described here in Customer Tests and Test-Driven Development.

Simple Design

XP teams build software to a simple but always adequate design. They start simple, and through programmer testing and design improvement, they keep it that way. An XP team keeps the design exactly suited for the current functionality of the system. There is no wasted motion, and the software is always ready for what’s next.

Design in XP is not a one-time thing, or an up-front thing, it is an all-the-time thing. There are design steps in release planning and iteration planning, plus teams engage in quick design sessions and design revisions through refactoring, through the course of the entire project. In an incremental, iterative

process like Extreme Programming, good design is essential. That's why there is so much focus on design throughout the course of the entire development.

Pair Programming

All production software in XP is built by two programmers, sitting side by side, at the same machine. This practice ensures that all production code is reviewed by at least one other programmer, and results in better design, better testing, and better code.

It may seem inefficient to have two programmers doing "one programmer's job", but the reverse is true. Research into pair programming shows that pairing produces better code in about the same time as programmers working singly. That's right: two heads really are better than one!

Some programmers object to pair programming without ever trying it. It does take some practice to do well, and you need to do it well for a few weeks to see the results. Ninety percent of programmers who learn pair programming prefer it, so we highly recommend it to all teams.

Pairing, in addition to providing better code and tests, also serves to communicate knowledge throughout the team. As pairs switch, everyone gets the benefits of everyone's specialized knowledge. Programmers learn, their skills improve, they become more valuable to the team and to the company. Pairing, even on its own outside of XP, is a big win for everyone.

Test-Driven Development

Extreme Programming is obsessed with feedback, and in software development, good feedback requires good testing. Top XP teams practice "test-driven development", working in very short cycles of adding a test, then making it work. Almost effortlessly, teams produce code with nearly 100 percent test coverage, which is a great step forward in most shops. (If your programmers are already doing even more sophisticated testing, more power to you. Keep it up, it can only help!)

It isn't enough to write tests: you have to run them. Here, too, Extreme Programming is extreme. These "programmer tests", or "unit tests" are all collected together, and every time any programmer releases any code to the repository (and pairs typically release twice a day or more), every single one of the programmer tests must run correctly. One hundred percent, all the time! This means that programmers get immediate feedback on how they're doing. Additionally, these tests provide invaluable support as the software design is improved.

Design Improvement (Refactoring)

Extreme Programming focuses on delivering business value in every iteration. To accomplish this over the course of the whole project, the software must be well-designed. The alternative would be to slow down and ultimately get stuck. So XP uses a process of continuous design improvement called *Refactoring*

The refactoring process focuses on removal of duplication (a sure sign of poor design), and on increasing the "cohesion" of the code, while lowering the "coupling". High cohesion and low coupling have been recognized as the hallmarks of well-designed code for at least thirty years. The result is that XP teams start with a good, simple design, and always have a good, simple design for the software. This lets them sustain their development speed, and in fact generally increase speed as the project goes forward.

Refactoring is, of course, strongly supported by comprehensive testing to be sure that as the design evolves, nothing is broken. Thus the customer tests and programmer tests are a critical enabling factor. The XP practices support each other: they are stronger together than separately.

Continuous Integration

Extreme Programming teams keep the system fully integrated at all times. We say that daily builds are for wimps: XP teams build multiple times per day.

The benefit of this practice can be seen by thinking back on projects you may have heard about (or even been a part of) where the build process was weekly or less frequently, and usually led to "integration hell", where everything broke and no one knew why.

Infrequent integration leads to serious problems on a software project. First of all, although integration is critical to shipping good working code, the team is not practiced at it, and often it is delegated to people who are not familiar with the whole system. Second, infrequently integrated code is often – I would say usually – buggy code. Problems creep in at integration time that are not detected by any of the testing that takes place on an unintegrated system. Third, weak integration process leads to long code freezes. Code freezes mean that you have long time periods when the programmers could be working on important shippable features, but that those features must be held back. This weakens your position in the market, or with your end users.

Collective Code Ownership

On an Extreme Programming project, any pair of programmers can improve any code at any time. This means that all code gets the benefit of many people's attention, which increases code quality and reduces defects. There is another important benefit as well: when code is owned by individuals, required features are often put in the wrong place, as one programmer discovers that he needs a feature somewhere in code that he does not own. The owner is too busy to do it, so the programmer puts the feature in his own code, where it does not belong. This leads to ugly, hard-to-maintain code, full of duplication and with low (bad) cohesion.

Collective ownership could be a problem if people worked blindly on code they did not understand. XP avoids these problems through two key techniques: the programmer tests catch mistakes, and pair programming means that the best way to work on unfamiliar code is to pair with the expert. In addition to ensuring good modifications when needed, this practice spreads knowledge throughout the team.

Coding Standard

XP teams follow a common coding standard, so that all the code in the system looks as if it was written by a single – very competent – individual. The specifics of the standard are not important: what is important is that all the code looks familiar, in support of collective ownership.

Metaphor

Extreme Programming teams develop a common vision of how the program works, which we call the "metaphor". At its best, the metaphor is a simple evocative description of how the program works, such as "this program works like a hive of bees, going out for pollen and bringing it back to the hive" as a description for an agent-based information retrieval system.

Sometimes a sufficiently poetic metaphor does not arise. In any case, with or without vivid imagery, XP teams use a common system of names to be sure that everyone understands how the system works and where to look to find the functionality you're looking for, or to find the right place to put the functionality you're about to add.

Sustainable

Extreme Programming teams are in it for the long term. They work hard, and at a pace that can be sustained indefinitely. This means that they work overtime when it is effective, and that they normally work in such a way as to maximize productivity week in and week out. It's pretty well understood these days that death march projects are neither productive nor produce quality software. XP teams are in it to win, not to die.

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

Thinking

One of the key advantages of agile thinking is that it brings potential problems to the surface quickly. This helps reduce any wasted effort later in the process. Eliminating unnecessary steps and red tape means teams work faster. And this gives them a quicker sense of accomplishment, which is invaluable for team morale.

And consider the advantages for the business. Because your team is more efficient, it is able to deliver more projects. And this means the organization is able to bring in customers. Instead of spending time on things that don't deliver value, you can focus all of your efforts on the most important tasks at hand. Because you're getting ongoing customer feedback, you can be more confident that you're working on projects that will ultimately lead to market success.

Pair Programming

Pair programming is a development technique in which two programmers work together at single workstation. Person who writes code is called a driver and a person who observes and navigates each line of the code is called navigator. They may switch their role frequently. Sometimes pair programming is also known as pairing. Pairing Variations : There are three pairing variations –

Newbie-newbie pairing can sometimes give a great result. Because it is better than one solo newbie. But generally, this pair is rarely practiced.

Expert–newbie pairing gives significant results. In this pairing, a newbie can learn many things from expert, and expert gets a chance to share his knowledge with newbie.

Expert–expert pairing is a good choice for higher productivity as both of them would be expert, so they can work very efficiently.

Advantages of Pair Programming:

Two brains are always better than one – If driver encounters a problem with code, there will be two of them who'll solve problem. When driver is writing code, navigator can think about a solution to problem.

Detection of coding mistakes becomes easier – Navigator is observing each and every line of code written by driver, so mistakes or error can be detected easily.

Mutual learning – Both of them can share their knowledge with each other and can learn many new things together.

Team develops better communication skills – Both of them share knowledge and work together for many hours a day and constantly share information with each other so this can help in developing better communication skills, especially when one of members is a newbie and other is an expert.

Disadvantages of Pair Programming:

Team Fit – High-intensity communication of pair programming is not a good fit for every developer. Sometimes, drivers are supposed to speak loud as they write code. Some people may not agree on idea of sitting, literally shoulder-to-shoulder, with a colleague for eight hours a day. Some experienced developers are more productive in solo rather than in pair programming.

Newbie-newbie pairing problem – Newbie–newbie pairing can produce results better than two newbie working independently, although this practice is generally avoided because it is harder for newbie to develop good habits without a proper role model.

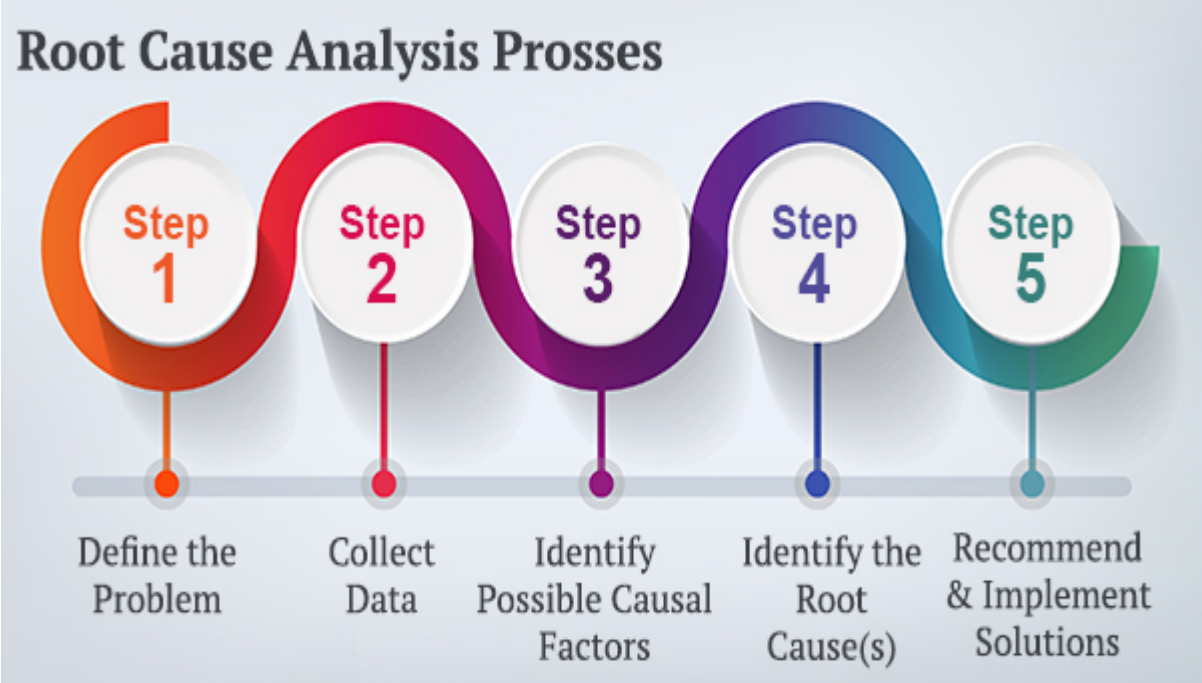
Root-cause analysis

Root Cause Analysis (RCA) is a systematic process used in various fields to identify the underlying causes of problems or issues. In the context of Extreme Programming (XP), which is an agile software development methodology, RCA can be applied to identify the root causes of defects, errors, or issues that arise during the software development process. RCA helps teams understand why problems occurred and what can be done to prevent them in the future. Here's how Root Cause Analysis can be integrated into the XP methodology:

Problem Identification: In XP, defects or issues are identified during various stages of development, such as coding, testing, or customer feedback. When an issue arises, the team acknowledges it and logs it as a potential problem that requires further investigation.

Containment and Temporary Fix: The immediate focus in XP is to ensure that the issue does not propagate further and disrupt the ongoing development process. A temporary fix or workaround may be implemented to keep the development process moving smoothly.

Issue Analysis: Once the immediate impact is contained, the team can start the RCA process. This involves investigating the issue thoroughly to understand the symptoms, impacts, and potential causes.



Data Collection: Gather relevant data, facts, and information related to the issue. This might include logs, code snippets, test results, and any other documentation that can shed light on what went wrong.

Cause Identification: Analyze the collected data to identify potential root causes of the issue. This might involve tracing the issue back through the development process to identify where and how it originated.

Cause Verification: Test the identified root causes to ensure they are indeed responsible for the issue. This might involve creating tests or simulations to validate the hypotheses.

Solution and Prevention: Once the root cause is confirmed, the XP team can work on implementing a permanent solution. In XP's spirit of embracing change, this could involve updating the development process, introducing new practices, or modifying existing ones to prevent similar issues in the future.

Feedback Loop: One of the key principles of XP is continuous improvement. After implementing the solution, it's important to monitor the effectiveness of the changes and gather feedback. If the issue is successfully prevented from recurring, the team can consider the RCA process complete. If not, they might need to revisit the analysis and solution.

Knowledge Sharing: RCA findings and the implemented solutions should be shared with the entire team to ensure that everyone learns from the experience. This helps in building a culture of shared learning and continuous improvement.

Remember that RCA is an iterative process, and in the context of XP, it aligns well with the methodology's focus on collaboration, communication, and adaptability. By systematically addressing the root causes of issues, XP teams can enhance their development process and deliver higher quality software.

UNIT - II: Planning

Product vision, release planning, the planning game, iteration planning, slack, stories, estimating.

2. Planning

Planning is a fundamental aspect of project management and is crucial for the successful execution of any endeavor. Whether you're talking about software development, business initiatives, events, or personal projects, effective planning helps you set goals, allocate resources, define tasks, and ensure that everything progresses smoothly. Here are some general steps and considerations for effective planning:

Define Your Goals and Objectives: Start by clearly understanding what you want to achieve. Define your goals, objectives, and desired outcomes. This provides a clear direction for your planning efforts.

Scope the Project: Determine the scope of the project. What needs to be accomplished? What are the boundaries and limitations? This will help prevent scope creep and keep the project focused.

Identify Tasks and Activities: Break down the project into smaller, manageable tasks and activities. This will make the project more manageable and allow you to assign responsibilities effectively.

Estimate Resources: Estimate the resources required for each task. This includes factors like time, budget, manpower, equipment, and materials. Make sure your estimates are realistic and consider potential risks.

Set Priorities: Determine the order in which tasks need to be completed. Identify critical tasks that need to be done early to avoid bottlenecks later on.

Create a Timeline: Develop a timeline that outlines when each task or activity should be started and completed. This timeline will help you track progress and identify any potential delays.

Allocate Resources: Assign resources to specific tasks and activities based on their expertise, availability, and capacity. Avoid overloading individuals or teams with too many tasks.

Risk Assessment and Mitigation: Identify potential risks that could impact your project. Develop strategies to mitigate these risks and have contingency plans in place.

Communication Plan: Define how you will communicate with team members, stakeholders, and any other relevant parties. Effective communication is crucial for keeping everyone informed and aligned.

Monitor and Adjust: Regularly monitor the progress of your project. If you encounter delays or issues, adjust your plan as needed. Adaptability is key to successful planning.

Collaboration: Involve relevant stakeholders in the planning process. Their insights and input can provide a more comprehensive view of the project's requirements and potential challenges.

Documentation: Keep thorough records of your planning process, decisions, and changes. Documentation helps maintain clarity and transparency throughout the project's lifecycle.

Review and Evaluation: Once the project is complete, conduct a post-project review to evaluate what went well, what could be improved, and what lessons were learned. This feedback can inform future planning efforts.

Remember that planning is not a one-size-fits-all process; it can vary based on the type of project, its complexity, and the resources available. Flexibility and adaptability are essential traits for any planning process, as unexpected changes and challenges can arise.

2.1 Product Vision

We know why our work is important and how we'll be successful.

Our vision is to serve customers while maximizing stakeholder value and upholding the family values of our employees."

Where Visions Come From

When talking about project vision statements, the best vision statement template to follow is this – (Action) a (deliverable) that (criteria). An example of this could be – Create (action) a furniture catalog (deliverable) that informs and attracts new clients (criteria).

"Take to market a copier that is small, inexpensive, and reliable enough for personal use on a secretary's desk."

"Design an on boarding program that quickly transforms new employees into valuable long-term contributors."

"Prepare a prioritized list of low-cost engineering recommendations that guides the organization to more energy-efficient operations."

Identifying the Vision

Four Steps to Defining Your Product Vision with Agile Management

Step 1: Developing the agile product objective. ...

Step 2: Creating a draft agile vision statement. ...

Step 3: Validating and revising the agile vision statement. ...

Step 4: Finalizing your agile vision statement.

The first stage in an agile project is defining your product vision. The product vision statement is an quick summary to communicate how your product supports the company's or organization's strategies. The vision statement must articulate the goals for the product.

The product owner is responsible for knowing about the product, its goals, and its requirements throughout the project and takes responsibility for creating the vision statement, although other people may have input.

The vision statement becomes a guiding light, the "what we are trying to achieve" statement that the development team, scrum master, and stakeholders refer to throughout the project.

Anyone involved with the project, from the development team to the CEO, should be able to understand the product vision statement.

Step 1: Developing the agile product objective

To write your vision statement, you must understand and be able to communicate the product's objective. You need to identify:

Key product goals: How will the product benefit the company creating it? The goals may include benefits for a specific department within your company as well as the company as a whole. What specific company strategies does the product support?

Customer: Who will use the product? This may be more than one entity.

Need: Why does the customer need the product? What features are critical to the customer?

Competition: How does the product compare with similar products?

Primary differentiation: What makes this product different from the status quo, or the competition, or both?

Step 2: Creating a draft agile vision statement

After you have a good grasp of the product's objective, create a first draft of your vision statement. In creating your vision statement, you help convey your product's quality, maintenance needs, and durability.

Step 3: Validating and revising the agile vision statement

After you draft your vision statement, review it against a quality checklist:

Is this vision statement clear, focused, and written for an internal audience?

Does the statement provide a compelling description of how the product meets customer needs?

Does the vision describe the best possible outcome?

Is the business objective specific enough that the goal is achievable?

Does the statement deliver value consistent with corporate strategies and goals?

Is the project vision statement compelling?

These yes-or-no questions help you determine whether your vision statement is thorough. If any answers are no, revise the vision statement.

When all answers are yes, move on to reviewing the statement with others, including:

Project stakeholders: The stakeholders will be able to identify that the vision statement includes everything the product should accomplish.

Your development team: Because the team will create the product, it must understand what the product needs to accomplish.

Scrum master: A strong understanding of the product helps the scrum master remove roadblocks and ensure that the development team is on the right path later in the project.

Agile mentor: Share the vision statement with your agile mentor, if you have one. The agile mentor is independent of the organization and can provide an external perspective, qualities that can make for a great objective voice.

Discover whether others think the vision statement is clear and delivers the message you want to convey. Review and revise the vision statement until the project stakeholders, the development team, and the scrum master fully understand the statement.

Step 4: Finalizing your agile vision statement

Make sure your development team, scrum master, and stakeholders have the final copy of the vision statement. You can even put a copy on the wall in the scrum team's work area, where everyone can see it every day. You refer to the vision statement throughout the life of the project.

If your project is more than a year long, you may want to revisit the vision statement to make sure the product reflects the marketplace and supports any changes in the company's needs.

2.2 Release planning

We plan for success.

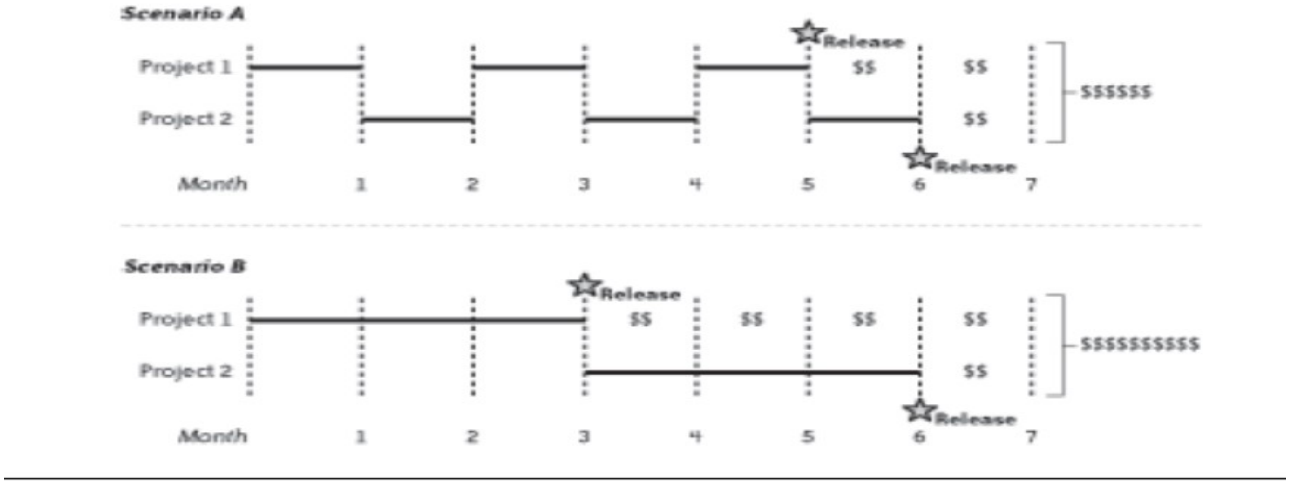
“Maximize our return on investment,” your boss (Project Manager, Customers) says-“We’ve already talked about the vision for this project. I’m counting on you to work out the details. Create your own plans and set your own release dates—just make sure we get a good return on our investment.”

2.2.1 One Project at a Time

First, work on only one project at a time. Many teams work on several projects simultaneously, which is a mistake.

Task-switching has a substantial cost Working on one project at a time allows you to release each project as you complete it, which increases the total value of your work.

Consider a team that has two projects. In this simplified example, each project has equal value; when complete, each project will yield $$ in value every month. Each project takes three months to complete.



1. Effects of multitasking on value

In Scenario A (see Figure 8-1), the team works on both projects simultaneously. To avoid task switching penalties, they switch between projects every month. They finish Project 1 after five months and Project 2 after six. At the end of the seventh month, the team has earned $$$$$$.

In Scenario B, the team works on just one project at a time. They release Project 1 at the end of the third month. It starts making money while they work on Project 2, which they complete after the sixth month, as before. Although the team's productivity didn't change—the projects still took six months—they earned more money from Project 1. By the end of the seventh month, they earned $$$$$$$$$$. That's nearly twice as much value with no additional effort.

2.2.2 Release Early, Release Often

Releasing early is an even better idea when you're working on a single project. If you group your most valuable features together and release them first, you can achieve startling improvements in value.

Consider another example team. This team has just one project. In Scenario A (see Figure 8-2), they build and release it after six months. The project is worth $$$$$ per month, so at the end of the seventh month, they've earned $$$$$.

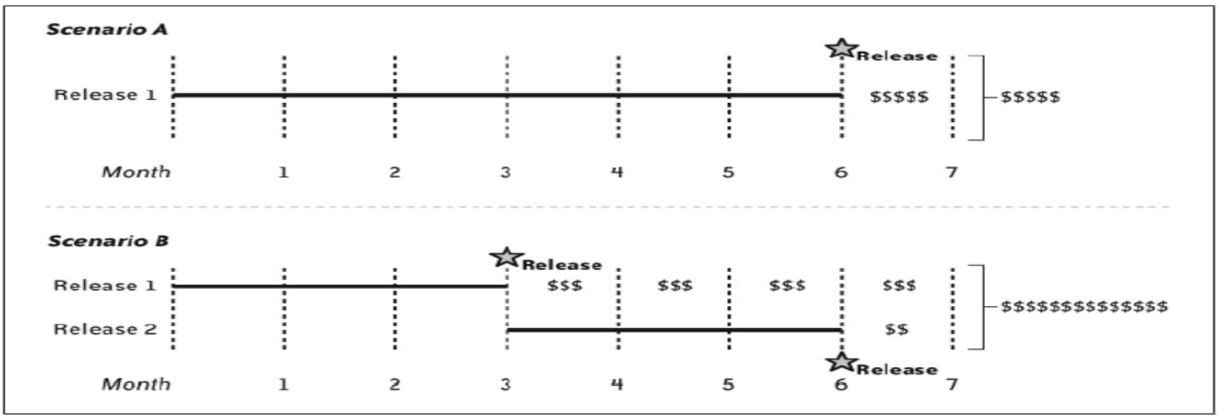


Figure 8-2. Effect of frequent releases on value

In Scenario B, the team groups the most valuable features together, works on them first, and releases them after three months. The first release starts making $$$ per month. They then work on the remaining features and release them at the end of the sixth month. As before, their productivity hasn't changed. All that's changed is their release plan. Yet due to the income from the first release, the team has made $$$$$$$$$$$$ by the end of the end of the seventh month—nearly triple that of Scenario A with its single release.

These scenarios are necessarily simplified. Software by Numbers has a more sophisticated example that uses real numbers and calculates value over the entire life of the product (see Table 8-1). In their example, the authors convert a five-year project with two end-of-project releases (Scenario A) into five yearly releases ordered by value (Scenario B). As before, the team's productivity remains the same.

Table 8-1. Realistic example of frequent releases

	Scenario A	Scenario B
Total Cost	$4.3 million	$4.712 million
Revenue	$5.6 million	$7.8 million
Investment	$2.76 million	$1.64 million
Payback	$1.288 million	$3.088 million
Net Present Value @ 10%	$194,000	$1.594 million
Internal Rate of Return	12.8%	36.3%

2.2.3 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.

2.2.4 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 overwhelming 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.

If this seems foolish, consider Writely, the online word processing application. It doesn't have

the breadth of features that Microsoft Word does, and it probably won't for many years. Instead, it focuses on what sets it apart: collaboration, remote document editing, secure online storage, and ease of use.

According to venture capitalist Peter Rip, the developers released the first alpha of Writely two

weeks after they decided to create it. How much is releasing early worth? Ask Google. Ten months later, they bought Writely, even though Writely still didn't come close to Microsoft Word's feature set. Writely is now known as Google Docs.

2.3 The Planning Game

Our plans take advantage of both business and technology expertise.

You may know when and what to release, but how do you actually construct your release plan? That's where the planning game comes in.

In economics, a game is something in which "players select actions and the payoffs depend on the actions of all players." The study of these games "deals with strategies for maximizing gains and minimizing losses and widely applied in the solution of various decision making problems.''

That describes the planning game perfectly. It is a structured approach to creating the best possible plan given the information available.

The planning game is most notable for the way it maximizes the amount of information contributed to the plan. It is strikingly effective. Although it has limitations, if you work within them, I know of no better way to plan.

2.3.1 How to Play

XP assumes that customers have the most information about value: what best serves the organization.

Programmers have the most information about costs: what it will take to implement and maintain those features. To be successful, the team needs to maximize value while minimizing costs. A successful plan needs to take into account information from both groups, as every decision to do something is also a decision not to do something else.

Accordingly, the planning game requires the participation of both customers and programmers. (Testers may assist, but they do not have an explicit role in the planning game.) It's a cooperative game; the team as a whole wins or loses not individual players.

Because programmers have the most information about costs–they are most qualified to say how long it will take to implement a story–they estimate.

Because customers have the most information about value–they're most qualified to say what is important–they prioritize.

Play the planning game must include following:

1. Anyone creates a story or selects an unplanned story.

2. Programmers estimate the story.

3. Customers place the story into the plan in order of its relative priority.

4. The steps are repeated until all stories have been estimated and placed into the plan.

During the planning game, programmers and customers may ask each other questions about estimates and priorities, but each group has final say over its area of expertise.

The result of the planning game is a plan: a single list of stories in priority order. Even if two stories are of equivalent priority, one must come before the other. If you're not sure which to put first, pick one at random.

2.4 Iteration Planning

We stop at predetermined, unchangeable time intervals and compare reality to plan.

Iterations are the heartbeat of an XP project. When 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.

2.4.1 The Iteration Timebox

Key characteristics of the Iteration Timebox in XP include:

Fixed Duration: The Iteration Timebox has a predetermined length, typically ranging from one to three weeks. This duration remains constant throughout the project to establish a predictable rhythm.

Regular Cycles: XP promotes a regular cycle of development iterations, ensuring that progress is made incrementally and consistently over time. This approach helps manage expectations and allows for frequent adaptations to changing requirements.

Focused Work: During each Iteration Timebox, the team selects a set of user stories or tasks to work on. These items are broken down into smaller units of work that can be completed within the timebox.

Collaborative Effort: XP emphasizes collaboration between team members. Cross-functional teams work together to design, develop, test, and deliver the features planned for the iteration.

Deliverable Outcome: At the end of each Iteration Timebox, the team delivers a potentially shippable product increment. This means that the code is tested and integrated, and it could be released to users if necessary.

Feedback and Adaptation: Frequent iterations provide opportunities for receiving feedback from stakeholders and end-users. This feedback informs the team's decisions for the next iteration, enabling them to adapt and refine their work based on real-world input.

Fixed Team Size: The team size is kept relatively small and consistent throughout the project. This promotes better communication, collaboration, and efficient decision-making.

Timeboxed Meetings: Various XP practices include timeboxed meetings such as the Planning Game (for selecting work for the iteration), the Stand-up Meeting (for daily coordination), and the Iteration Review (for demonstrating the completed work to stakeholders).

The Iteration Timebox in XP serves several purposes, including managing scope, maintaining a predictable development rhythm, gathering frequent feedback, and ensuring that the project stays adaptable to changing requirements. It also helps in reducing the risks associated with long development cycles and enables the team to make informed decisions based on real progress and feedback.

2.4.2 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)

Many teams start their iterations on Monday morning and end Friday evening, but I prefer iterations that start on Wednesday morning. This allows people to leave early on Friday or take Monday off without missing important events. It also allows the team to conduct releases before the weekend.

2.4.3 How to Plan an Iteration:

An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks.

Each iteration is 1 to 3 weeks long. User stories are chosen for this iteration by the customer from the release plan in order of the most valuable to the customer first.

Failed acceptance tests to be fixed are also selected. The customer selects user stories with estimates that total up to the project velocity from the last iteration.

The user stories and failed tests are broken down into the programming tasks that will support them. Tasks are written down on index cards like user stories. While user stories are in the customer's language, tasks are in the developer's language. Duplicate tasks can be removed. These task cards will be the detailed plan for the iteration.

Developers sign up to do the tasks and then estimate how long their own tasks will take to complete. It is important for the developer who accepts a task to also be the one who estimates how long it will take to finish. People are not interchangeable and the person who is going to do the task must estimate how long it will take.

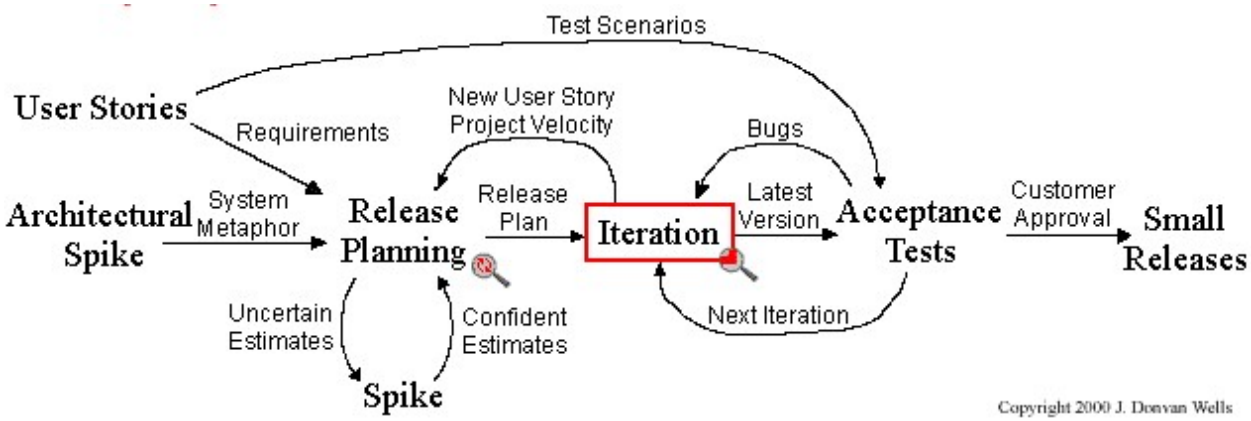
Each task should be estimated as 1, 2, or 3 (add 1/2 if you need to) ideal programming days in duration.

Ideal programming days are how long it would take you to complete the task if there were no distractions.

Tasks which are shorter than 1 day can be grouped together. Tasks which are longer than 3 days should be broken down farther.

Now the project velocity is used again to determine if the iteration is over booked or not. Total up the time estimates in ideal programming days of the tasks, this must not exceed the project velocity from the previous iteration. If the iteration has too much then the customer must choose user stories to be put off until a later iteration (snow plowing).

If the iteration has too little then another story can be accepted. The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate.



It is often alarming to see user stories being snow plowed. Don't panic. Remember the importance of unit testing and refactoring. A debt in either of these areas will slow you down. Avoid adding any functionality before it is scheduled. Just add what you need for today. Adding anything extra will slow you down.

Don't be tempted into changing your task and story estimates. The planning process relies on the cold reality of consistent estimates, fudging them to be a little lower creates more problems. Keep an eye on your project velocity and snow plowing. You may need to re-estimate all the stories and re-negotiate the release plan every three to five iterations, this is normal.

So long as you always implement the most valuable stories first you will always be doing as much as possible for your customers and management. An iterative development style can add agility to your development process. Try just in time planning by not planning specific programming tasks farther ahead than the current iteration.

2.5 Slack

2.5.1 How much Slack?

In project management and scheduling, "slack" refers to the amount of time a task or activity can be delayed without causing a delay in the project's overall timeline. Slack is often represented in a project schedule using a network diagram or Gantt chart. There are two types of slack:

Free Slack (Float): This is the amount of time a task can be delayed without delaying the start of the next dependent task.

Total Slack: This is the amount of time a task can be delayed without delaying the project's final completion date.

The amount of slack in a project depends on various factors such as task dependencies, resource availability, and project constraints. It's important for project managers to carefully manage and monitor slack to ensure that the project stays on schedule.

2.5.2. How to Introduce Slack (the messaging platform):

If you are looking to introduce Slack, the messaging and collaboration platform, to your team or organization, here are some steps to consider:

a. Assess Your Needs: First, assess your team's communication and collaboration needs. Identify the pain points and areas where Slack could be beneficial.

b. Choose a Plan: Visit the Slack website and choose a suitable plan for your team. Slack offers free and paid plans with varying features. Consider your team size and requirements when selecting a plan.

c. Create a Workspace: Sign up for Slack and create a workspace for your team. You'll need to choose a workspace name and invite team members to join.

d. Set up Channels: Channels are where conversations happen in Slack. Create channels for different teams, projects, or topics to keep discussions organized.

e. Invite Team Members: Invite team members to join your Slack workspace. You can send them email invitations, and they can set up their Slack accounts.

f. Customize Settings: Customize Slack's settings to fit your team's needs. This includes setting notification preferences, integrating with other apps, and configuring security settings.

g. Training and On boarding: Provide training and on boarding sessions for your team to help them get familiar with Slack's features and best practices.

h. Encourage Adoption: Encourage your team to use Slack for communication, file sharing, and collaboration. Highlight the benefits and how it can streamline workflows.

i. Monitor and Improve: Continuously monitor how Slack is being used and gather feedback from your team. Make adjustments to improve its effectiveness in your organization.

Slack can be a powerful tool for improving team communication and collaboration, but its successful introduction depends on proper planning and adoption strategies.

2.6 Stories

We plan our work in small, customer-centric pieces.

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.

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 (Test Procedure Specification) report for upcoming investor dog-and-pony show"
- "Customizable corporate branding on user login screen"

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.

2.6.1 Story Cards

Write stories on index cards.

In Extreme Programming (XP), "Story Cards" or "Index Cards" are a physical or digital tool used to represent user stories, which are a fundamental component of XP's agile software development methodology. These story cards serve as a way to capture and organize requirements and priorities for the development team. Here's how story cards are typically used in XP:

User Stories: User stories are brief, informal descriptions of a feature or functionality from the user's perspective. They are written in plain language and follow the "As a [type of user], I want [an action] so that [benefit/value]" format. Each user story represents a small piece of functionality or a specific requirement.

Writing User Stories: In XP, the team collaboratively writes user stories during the project's planning phase. These stories are written on physical index cards or digitally within a tool.

Physical Index Cards: Many XP teams prefer physical index cards because they are tangible and can be easily manipulated. Each card typically contains a single user story.

Information on a Story Card: A typical story card includes the user story itself, any acceptance criteria that define the story's completeness, and a unique identifier. The identifier helps track the story's progress and can be used in discussions and planning sessions.

Prioritization: Once user stories are written, they are prioritized based on their importance and value to the project or the customer. The team collectively decides on the order in which stories should be implemented.

Iteration Planning: In XP, development occurs in short iterations (usually 1-2 weeks). During iteration planning meetings, the team selects a set of high-priority user stories to work on in the upcoming iteration. These selected stories are often moved to a separate area on a physical board or marked digitally to indicate that they are in progress.

Progress Tracking: As the team works on user stories, they update the cards to reflect their status. For example, they might mark a card as "In Progress," "Testing," or "Done" as the work progresses. This provides a visual representation of the project's status.

Discussion and Collaboration: Story cards facilitate discussions and collaboration among team members. They can be easily moved around on a physical board or updated in digital tools during daily stand-up meetings or other team gatherings.

Completion and Validation: Once a user story is completed and meets its acceptance criteria, it is considered "Done." The team can then demonstrate the functionality to stakeholders for validation.

Continuous Improvement: XP encourages continuous improvement, so the team regularly reflects on its progress and the effectiveness of its processes, including the use of story cards. Adjustments are made as needed to optimize the development process.

Story cards are a simple yet effective way to manage and prioritize work in XP. They promote collaboration, transparency, and flexibility, allowing the team to adapt to changing requirements and deliver value to the customer incrementally. Whether physical or digital, story cards are a valuable tool for agile development teams practicing Extreme Programming.

2.6.2 Customer-Centricity

Stories need to be customer-centric. Write them from the on-site customers' point of view, and make sure they provide something that customers care about.

Customer centricity is a mindset that helps organizations make decisions that are based on a deep understanding of its effect on customers and end-users that motivates the following behaviors:

Focusing on the customer – aligning and focusing the organization on specific, targeted user segments.

Understanding the customer's needs – moving beyond merely listening to customers who ask for features and investing the time to identify the customer's fundamental and ongoing needs.

Thinking and feeling like the customer – striving to see the world from their customer's point of view.

Building whole product solutions – designing a complete solution for the user's needs, and ensuring that the initial and long-term customer experience is continually evolving toward the ideal solution.

Knowing customer lifetime value – Moving beyond a transactional mentality and focusing on creating longer-term relationships based on a clear understanding of how the customer gains value.

2.6.3 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.

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.

2.6.4 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.

Nonfunctional stories

Performance, scalability, and stability called nonfunctional requirements —should be scheduled with stories.

Bug stories

Ideally, your team will fix bugs as soon as they find them, before declaring a story "done done." Nobody's perfect, though, and you will miss some bugs. Schedule these bugs with story cards, such as

“Fix multiple-user editing bug.” Schedule them as soon as possible to keep your code clean and reduce your need for bug-tracking software.

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 to research the technology, so these sorts of stories are typically called spike stories.

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.

2.7 Estimating

We provide reliable estimates.

Programmers often consider estimating to be a black art—one of the most difficult things they must do. Many programmers find that they consistently estimate too low. To counter this problem, they pad their estimates, but sometimes even these rough guesses are too low.

2.7.1 What Works in Estimating

One reason estimating is so difficult is that programmers can rarely predict how they will spend their time. A task that requires eight hours of uninterrupted concentration can take two or three days if the programmer must deal with constant interruptions. It can take even longer if the programmer works on another task at the same time.

2.7.2 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 20 percent more than the actual time—the estimates are consistent in aggregate.

Additionally, although each iteration experiences a different set of interruptions, the amount of time required for the interruptions also tends to be consistent from iteration to iteration.

As a result, you can reliably convert estimates to calendar time if you aggregate all the stories in iteration. A single scaling factor is all you need. This is where velocity comes in.

Your velocity is the number of story points you can complete in 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.

UNIT - III

Collaborating

Trust, real customer involvement, ubiquitous language, stand-up meetings, iteration demo, reporting.

Collaborating

- Effective teamwork is an essential ingredient for success in Extreme Programming (XP) and other Agile software development environments.
- In these workplaces, teamwork strengthens feedback loops and fosters collaboration across functions and boundaries.
- Collaborative communication skills are a critical contribution to the software development process.
- Top-quality teamwork requires a commitment to common goals, effective working relationships, and full participation in planning and decision making; all of which are improved by people who can communicate collaboratively interpersonally and as part of a group.
- Development and IT professional who are drawn to Extreme Programming work best in an environment where ideas and information are shared, where the spirit of team ness is strong, and where team members work together to accomplish common goals ñ an environment of collaborative communication.
- When XP, or another Agile methodology, is introduced on a project, team members face the challenge of learning and applying the values, principles and practices.
- The values and principles highlight the importance of collaboration and the practices provide some structure for increasing effective communication, but eventually the team discovers the limits of its member's skills as communicators and needs to institute communication practices as well.

Three collaborative communication skills that are important for teams to use well are: active listening; seeking, giving and receiving interpersonal feedback; and group decision-making.

Eight practices to help your team and its stakeholders collaborate efficiently and effectively:

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

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”.

Team members must rely on each other for help. When one member of a team encounters a question that she cannot answer, she doesn’t hesitate to ask someone who does know the answer. Sometimes these quick questions turn into longer pairing sessions.

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

The organization needs to trust the team, too. XP is strange and different at first. It doesn’t provide the normal indicators of progress that managers are accustomed to seeing. It takes trust to believe that the team will deliver a success.

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

- Team Strategy #1: Customer-Programmer Empathy
- Team Strategy #2: Programmer-Tester Empathy
- Team Strategy #3: Eat Together
- Team Strategy #4: Team Continuity

Team Strategy #1: Customer-Programmer Empathy

Customer-Programmer Empathy focuses on building a strong connection between developers and customers to enhance project success. Programmers should actively listen to customer needs, understanding both technical requirements and business context. Frequent collaboration through regular meetings ensures alignment and minimizes misunderstandings. Agile methodologies, such as iterative feedback loops, can help programmers adjust quickly to customer changes. Empathy-driven communication enables developers to anticipate customer challenges, thus creating more user-friendly solutions. Fostering mutual respect and clear expectations between both parties can lead to higher-quality software and better user satisfaction.

Team Strategy #2: Programmer-Tester Empathy

Programmer-Tester Empathy aims to create a strong collaborative relationship between programmers and testers for better software quality. Developers and testers must understand each other’s perspectives and challenges. Programmers should recognize the value testers bring in identifying bugs and ensuring quality, while testers should appreciate the complexity of the code development process.

Effective communication between both teams can prevent misunderstandings and reduce friction. Regular interaction, such as joint discussions on potential issues or test scenarios, ensures everyone is aligned. Empathy in this relationship leads to more proactive bug identification, fewer misunderstandings, and smoother project delivery.

Team Strategy #3: Eat Together

Eat Together encourages team bonding through shared meals, fostering better relationships and collaboration. When team members, including developers, testers, managers, and other stakeholders, share a meal, it creates an informal environment where they can connect on a personal level.

This strategy breaks down hierarchical barriers and encourages open communication. People are more likely to share ideas, discuss challenges, and collaborate more effectively when they feel comfortable around each other. Eating together helps build trust and camaraderie, which can lead to improved teamwork, increased morale, and a more cohesive work environment.

Team Strategy #4: Team Continuity

Team Continuity emphasizes maintaining stable and consistent teams over time to improve efficiency and collaboration. When team members work together on multiple projects or over extended periods, they develop a strong understanding of each other's strengths, weaknesses, and communication styles.

This continuity fosters deeper trust, reduces the learning curve for new projects, and enhances problem-solving as team members build on their collective experience. Consistent teams also adapt more easily to challenges and changes, leading to smoother workflows and higher-quality output. Overall, team continuity leads to increased productivity, better project outcomes, and long-term success.

Organization Strategy

Organization Strategy focuses on fostering collaboration and flexibility within the team. It encourages small, cross-functional teams with close communication between developers, customers, and testers. Continuous feedback, pair programming, and frequent releases ensure quick adjustments to changing requirements. XP promotes a flat hierarchy, empowering team members to take ownership of their work. The goal is to create a sustainable pace and adaptive environment that delivers high-quality software aligned with customer needs.

Organizational Strategy #1: Show Some Hustle

Show Some Hustle emphasizes fostering a culture of urgency, proactiveness, and initiative within the organization. Teams are encouraged to move quickly, taking ownership of their tasks and striving for high efficiency without compromising quality. This strategy promotes a "get-it-done" attitude, where employees actively seek solutions to problems rather than waiting for direction.

Clear goals, tight deadlines, and accountability help drive this hustle, ensuring everyone stays focused on delivering results. Quick decision-making and streamlined processes reduce delays. However, balance is key—showing hustle should not lead to burnout but should create momentum and energy within the team. Ultimately, this strategy fosters a sense of motivation, increases productivity, and drives faster innovation across the organization.

Organizational Strategy #2: Deliver on Commitments

Deliver on Commitments focuses on building trust by consistently meeting deadlines and fulfilling promises. Teams must prioritize clear communication, realistic goal-setting, and accountability to ensure

commitments are achievable. By delivering as promised, organizations enhance their credibility with clients and stakeholders. This strategy fosters a culture of reliability and encourages long-term partnerships. Meeting commitments consistently also improves team morale and drives continuous improvement.

Organizational Strategy #3: Manage Problems emphasizes proactive problem-solving and effective issue resolution. Teams should identify potential issues early and address them promptly to minimize disruption. This strategy involves clear communication channels, structured problem-solving processes, and a focus on root causes rather than symptoms. By managing problems efficiently, organizations maintain productivity and prevent minor issues from escalating. Encouraging a culture of transparency and continuous improvement helps teams handle challenges effectively and adapt quickly.

Organizational Strategy #4: Respect Customer Goals centers on aligning organizational efforts with the objectives and needs of the customer. Teams should actively listen to customer feedback, understand their priorities, and integrate their goals into project planning. This strategy promotes transparency, ensures that deliverables meet customer expectations, and builds stronger relationships. By respecting and prioritizing customer goals, organizations enhance satisfaction and loyalty, ultimately driving business success.

Organizational Strategy #5: Promote the Team focuses on recognizing and celebrating the collective achievements and contributions of the team. This strategy involves highlighting team successes, providing opportunities for professional growth, and fostering a positive work environment. By promoting the team, organizations boost morale, encourage collaboration, and enhance motivation. Recognizing individual and group efforts helps build a strong, cohesive team that is aligned with organizational goals.

Organizational Strategy #6: Be Honest emphasizes the importance of transparency and integrity in all interactions. Teams should communicate openly about challenges, successes, and failures, fostering trust and accountability. This strategy involves providing honest feedback, acknowledging mistakes, and addressing issues directly. Being honest helps build strong relationships, improves decision-making, and creates a culture of respect and reliability. Ultimately, it supports a more effective and ethical work environment.

Real Customer Involvement

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

Real Customer Involvement ensures that customers actively participate throughout the project lifecycle, from planning to delivery. This approach involves regular feedback sessions, collaborative reviews, and iterative testing to align the product with customer needs. By engaging customers continuously, teams can adapt to changes quickly and address concerns promptly. This involvement enhances the relevance and quality of the final product, builds stronger customer relationships, and ensures that deliverables meet or exceed expectations.

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.

Personal Development focuses on the continuous growth and improvement of an individual's skills, knowledge, and abilities. It involves setting and pursuing personal and professional goals, seeking feedback, and engaging in learning opportunities such as courses, workshops, or self-study. Personal development also includes cultivating soft skills like communication, time management, and emotional intelligence. By investing in personal growth, individuals enhance their career prospects, boost confidence, and improve overall well-being. It supports adaptability and resilience in a rapidly changing work environment.

Ubiquitous Language

We understand each other.

Ubiquitous Language refers to a shared, common vocabulary used by both technical and non-technical team members within a project or organization. This concept, integral to Domain-Driven Design (DDD), ensures that all stakeholders—developers, business analysts, and customers—use the same terms to describe the domain and its concepts. By establishing a consistent language, teams minimize misunderstandings and miscommunications, align their efforts, and improve clarity. It fosters better collaboration and ensures that the software solution accurately reflects the domain's needs and requirements.

The Domain Expertise Conundrum

The Domain Expertise Conundrum refers to the challenge of bridging the gap between technical experts and domain experts within a project. In many cases, developers may lack in-depth knowledge of the business domain they are working on, while domain experts may not fully understand technical constraints or possibilities. This discrepancy can lead to misaligned goals, ineffective solutions, and communication barriers.

To address this conundrum, it's crucial to foster collaboration between domain experts and developers. Techniques like incorporating domain experts in the development process, using domain-driven design principles, and promoting ongoing knowledge sharing can help bridge the gap. Creating a common understanding and shared language between both groups ensures that the software accurately addresses business needs and leverages technical capabilities effectively.

Ubiquitous Language in Code

Ubiquitous Language in Code involves incorporating the shared vocabulary and concepts from the business domain directly into the software's codebase. This practice ensures that the code reflects the domain's terminology and logic, making it more understandable and maintainable for both developers and non-technical stakeholders. By using domain-specific names for classes, methods, and variables, teams align their technical implementation with business requirements.

This approach helps bridge the gap between domain experts and developers, as the code becomes a more accurate representation of the business domain. It improves communication, reduces misunderstandings, and enhances the overall clarity and coherence of the software, making it easier to adapt and evolve as business needs change.

Stand-Up Meetings

Stand-Up Meetings are brief, daily team meetings designed to facilitate quick updates and coordination. Typically lasting 10-15 minutes, these meetings involve team members standing to encourage brevity and focus. Participants share what they accomplished since the last meeting, what they plan to work on next, and any obstacles they are facing.

The primary benefits of stand-up meetings include improved communication, increased team alignment, and quicker identification of issues or blockers. They promote a collaborative environment and help teams stay on track with their goals. Regular stand-ups also contribute to maintaining a steady pace of progress and fostering a sense of accountability among team members.

How to Hold a Daily Stand-Up Meeting

To hold an effective daily stand-up meeting, follow these steps:

1. Schedule Consistently: Set a regular time each day to hold the stand-up meeting. Consistency helps team members integrate it into their routines and ensures everyone is present.
2. Keep It Short: Limit the meeting to 10-15 minutes to maintain focus and efficiency. Standing up encourages brevity and keeps the meeting fast-paced.
3. Follow a Structured Format:
    - Yesterday's Progress: Each team member briefly reports on what they accomplished since the last meeting.
    - Today's Plan: Team members outline what they will work on next.
    - Blockers: Discuss any obstacles or issues that are preventing progress.
4. Encourage Participation: Ensure that all team members have a chance to speak. This fosters a sense of involvement and helps uncover potential problems early.
5. Stay Focused: Keep the discussion on track and avoid delving into detailed problem-solving or off-topic conversations. Address detailed issues outside the stand-up in separate discussions.
6. Use Visual Aids: If helpful, use a task board or project management tool to visually track progress and highlight blockers.
7. Encourage Transparency: Promote honesty and openness about progress and challenges to foster trust and collaborative problem-solving.
8. Be Flexible: Adapt the meeting format as needed based on the team's evolving needs and feedback to ensure it remains effective and relevant.

Iteration Demo

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

In XP (Extreme Programming), **Iteration Demos** are an essential practice where the team demonstrates the working software produced during an iteration. Here's how these demos typically work:

1. **Frequency**: An XP team delivers a functional version of the software at the end of each iteration, which usually spans one to two weeks. This frequent delivery ensures that stakeholders regularly see progress.
2. **Content**: The demo includes the new features or enhancements completed during the iteration. It showcases the working software to stakeholders, including customers and team members.
3. **Feedback**: Stakeholders provide feedback on the demoed features. This feedback is crucial for making adjustments and refining the product according to user needs and expectations.
4. **Collaboration**: The demo serves as a collaborative session where developers and customers discuss the software's functionality, address any issues, and plan for upcoming iterations.
5. **Transparency**: Regular demos foster transparency in the development process, helping to manage expectations and ensuring that the team remains aligned with customer requirements.

By producing working software every week, XP teams maintain a high level of responsiveness to change and continuously validate their progress against customer needs.

How to Conduct an Iteration Demo

To conduct an effective iteration demo in XP (Extreme Programming), follow these steps:

1. Prepare in Advance:
    - Select Content: Choose the features and functionality developed during the iteration that will be demonstrated.
    - Setup Environment: Ensure that the demo environment is ready and all necessary hardware and software are functioning correctly.
2. Schedule the Demo:
    - Timing: Set a specific date and time for the demo that works for both the development team and stakeholders.

- Duration: Keep the demo concise, typically 30-60 minutes, to maintain focus and engagement.

3. Present the Demo:
    - Introduction: Briefly outline the goals and scope of the iteration, including any changes or enhancements made.
    - Demonstrate Features: Walk through the new features and functionality, showing how they work and how they address user needs.
    - Highlight Improvements: Emphasize any significant improvements or additions compared to previous iterations.
4. Engage Stakeholders:
    - Invite Feedback: Encourage stakeholders to ask questions, provide feedback, and discuss any issues they observe.
    - Document Comments: Take note of all feedback and concerns raised during the demo for future reference and action.
5. Discuss Next Steps:
    - Action Items: Identify any action items or changes needed based on feedback and plan for addressing them in the next iteration.
    - Future Goals: Outline the focus and objectives for the upcoming iteration, ensuring alignment with stakeholder expectations.
6. Reflect and Improve:
    - Team Review: After the demo, hold a brief team review to discuss what went well and what could be improved for future demos.
    - Continuous Improvement: Use insights from the demo to refine the process and enhance the effectiveness of future iterations.

By following these steps, you can ensure that iteration demos are productive, informative, and aligned with stakeholder expectations, leading to continuous improvement and successful project outcomes.

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?

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

Types of Reports

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

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

The first set of reports are a normal byproduct of the whole team's work.

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

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.

Weekly demo

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

Release and iteration plans

The release and iteration planning boards already posted in your workspace provide great detail about progress.

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

Only produce reports that are strictly necessary. 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.

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 . It sounds simple—productivity is the amount of production over time—but in software, we don't have an objective way to measure production.

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.

When you count the defects is just as important as how you count them. On an XP team, finding and fixing defects is a normal part of the process. To avoid overcounting defects, wait until you have marked a story as "**done done"** and completed its iteration before marking something as a defect.

Time usage

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

Reports to Avoid

Source lines of code (SLOC) and function points

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

Velocity

If a team estimates its stories in advance, an improvement in velocity may result from an improvement in productivity.

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

UNIT - IV

Developing

Incremental requirements, customer tests, test driven development, incremental design and architecture, performance optimization.

Developing:

Software development requires the cooperation of everyone on the team. Programmers are often called “developers,” but in reality everyone on the team is part of the development effort.

- When you share the work, customers identify the next requirements while programmers work on the current ones. Testers help the team figure out how to stop introducing bugs.
- Programmers spread the cost of technical infrastructure over the entire life of the project.
- The best way I know to reduce the cost of writing software is to improve the internal quality of its code and design.
- I’ve never seen high quality on a well-managed project fail to repay its investment. It always reduces the cost of development in the short term as well as in the long term.

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

- Incremental Requirements allows the team to get started while customers work out requirements details.
- Customer Tests help communicate tricky domain rules.
- Test-Driven Development allows programmers to be confident that their code does what they think it should.
- Refactoring enables programmers to improve code quality without changing its behavior.

Simple Design allows the design to change to support any feature request, no matter how surprising.

- Incremental Design and Architecture allows programmers to work on features in parallel with technical infrastructure.
- Spike Solutions use controlled experiments to provide information.
- Performance Optimization uses hard data to drive optimization efforts.
- Exploratory Testing enables testers to identify gaps in the team's thought processes.

Incremental Requirements:

- We define requirements in parallel with other work.
- 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 [Cockburn] discusses, 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.
- Some requirements will necessitate even expert customers to consider a variety of options or do some research before making a decision.
- Customers, you can and should include other team members in your discussions if it helps clarify your options.
- For example, you may wish to include a programmer in your discussion of user interface options so you can strike a balance between an impressive UI and low implementation cost.

- Write down any requirements you might forget. These notes are primarily for your use as customers so you can easily answer questions in the future and to remind you of the decisions you made.
- They don't need to be detailed or formal requirements documents; keep them simple and short. When creating screen mock-ups, for example, I often prefer to create a sketch on a
- White board and take a digital photo. I can create and photograph a whiteboard sketch in a fraction of the time it takes me to make a mock-up using an electronic tool.

Work Incrementally

Work on requirements incrementally, in parallel with the rest of the team's work.

It 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, discussed in "Release Planning".

Vision, features, and stories

Start by clarifying your project vision, then identify features and stories as described in "Release Planning".

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 work as you expected.

- You don't need to exhaustively test the application—you can rely 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.

Customer Tests

- 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.
- Customer tests are a communication tool that helps identify potential misunderstandings between programmers and customers at the beginning of an iteration. They are not intended to prove that software works.
- 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.
- 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.

- If you're not sure what the programmers might misunderstand, ask. Be careful, though; when business experts and programmers first sit down to create customer unit tests, both groups are often surprised by the extent of existing misunderstandings.
- Once you've identified potential misunderstandings, gather the team at a whiteboard and summarize the story in question. Briefly describe how the story should work and the rules you're going to provide examples for. It's OK to take questions, but don't get stuck on this step.
- For example, a discussion of invoice deletion might go like this:

Customer: One of the stories in this iteration is to add support for deleting invoices. In addition to the screen mock-ups we've given you, we thought some customer tests would be appropriate. Deleting invoices isn't as simple as it appears because we have to maintain an audit trail.

- The basic rule is that it's OK to delete invoices that haven't been sent to customers—because presumably that kind of invoice was a mistake. Once an invoice has been sent to a customer, it can only be deleted by a manager. Even then, we have to save a copy for auditing purposes.
- Programmer: When an invoice hasn't been sent and gets deleted, is it audited?
- Customer: No—in that case, it's just deleted.

Demonstrate

- Tables are often the most natural way to describe this information, but you don't need to worry about formatting.
- 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 admin

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,

-

It's tempting to create generic examples, such as "this invoice hasn't been sent to customers, so anybody can delete it", but those get confusing quickly and programmers can't automate them. Provide specifics. "This invoice hasn't been sent to customers, so an account rep can delete it." This will require you to create more examples—that's a good thing.

- 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.
- The purpose here is to communicate, not to exhaustively test the application. You only need enough examples to show the differences in the rules.

Develop

- When you've covered enough ground, document your discussion so the programmers can start working on implementing your rules.
- This is also a good time to evaluate whether the examples are in a format that works well for automated testing. If not, discuss alternatives with the programmers. The conversation might go like this:
- Programmer: OK, I think we understand what's going on here. We'd like to change your third set of examples, though—the ones where you say "Y" for "Sent." Our invoices don't have a "Sent" property. We'll calculate that from the other properties you mentioned. Is it OK if we use "Emailed" instead?
- Customer: Yeah, that's fine. Anything that sends it works for that example.

Test Driven Development (TDD)

Test Driven Development (TDD) is a software development methodology that emphasizes writing tests before writing the actual code. It ensures that code is always tested and functional, reducing bugs and improving code quality. In TDD, developers write small, focused tests that define the desired functionality, then write the minimum code necessary to pass these tests, and finally, refactor the code .

This cyclic process helps in creating reliable, maintainable, and efficient software. By following TDD, teams can enhance code reliability, accelerate development cycles, and maintain high standards of software quality.

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.

When used properly, it also helps improve your design, documents your public interfaces, and guards against future mistakes.

Process of Test Driven Development (TDD)

It is the process in which test cases are written before the code that validates those cases. It depends on the repetition of a concise development cycle. Test-driven Development is a technique in which automated Unit tests are used to drive the design and free decoupling of dependencies.

The following sequence of steps is generally followed:

- Run all the test cases and make sure that the new test case fails.
- Red – Create a test case and make it fail, Run the test cases
- Green – Make the test case pass by any means.
- Refactor – Change the code to remove duplicate/redundancy. Refactor code – This is done to remove duplication of code.
- Repeat the above-mentioned steps again and again

Advantages of Test Driven Development (TDD)

- Unit test provides constant feedback about the functions.
- Quality of design increases which further helps in proper maintenance.
- Test driven development act as a safety net against the bugs.
- TDD ensures that your application actually meets requirements defined for it.
- TDD have very short development lifecycle.

- Disadvantages of Test Driven Development (TDD)
- Increased Code Volume: Using TDD means writing extra code for tests cases , which can make the overall codebase larger and more Unstructured.
- False Security from Tests: Passing tests will make the developers think the code is safer only for assuming purpose.
- Maintenance Overheads: Keeping a lot of tests up-to-date can be difficult to maintain the information and its also time-consuming process.
- Time-Consuming Test Processes: Writing and maintaining the tests can take a long time.
- Testing Environment Set-Up: TDD needs to be a proper testing environment in which it will make effort to set up and maintain the codes and data.

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.

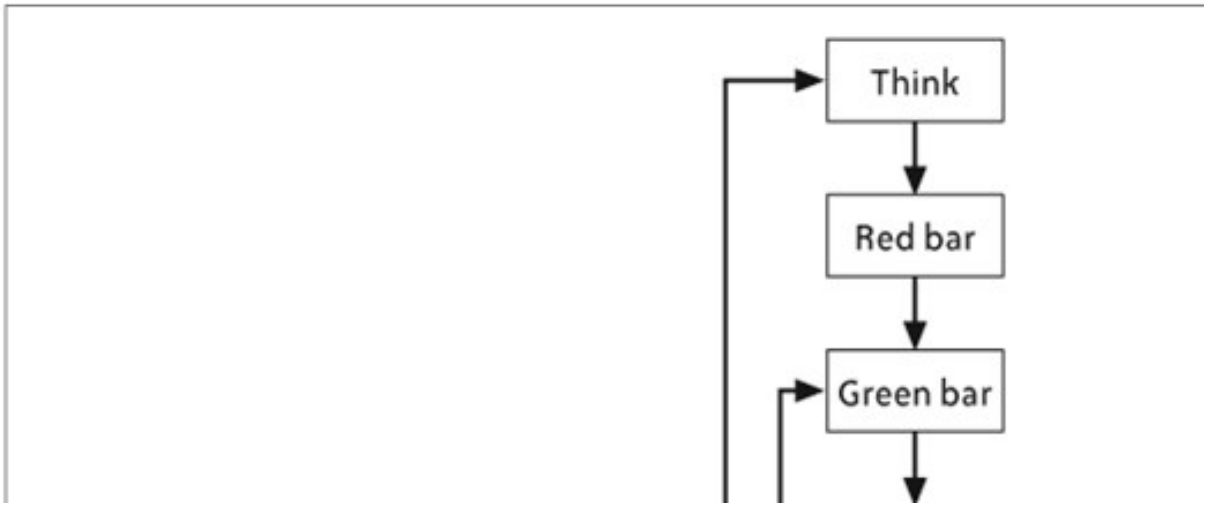
Test-driven development applies the same principle to programmer intent. Just as modern compilers provide more feedback on the syntax of your code, TDD cranks up the feedback on the execution of your code. Every few minutes—as often as every 20 or 30 seconds

- 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.
- 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.

How to Use TDD

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.

To use TDD, follow the "red, green, refactor" cycle illustrated in Figure With experience, unless you're doing a lot of refactoring, each cycle will take fewer than five minutes. Repeat the cycle until your work is finished. You can stop and integrate whenever all your tests pass, which should be every few minutes.



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

- 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.
- 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.
- 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

- Write just enough production code to get the test to pass. Again, you should usually need less than five lines of code.
- 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.
- 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

- 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.

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.

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

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.

Continue this pattern. By the fourth or fifth time you work with a design element—be it a method, a class, or something bigger.

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.

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 b reakthrough (see Figure). 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.

Incrementally Designing Methods

You've seen this level of incremental design before: it's test-driven development. While the driver implements, the navigator thinks about the design. 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 refactorings 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.

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.

Don’t let design discussions turn into long, drawn-out disagreements. Follow the 10-minute rule : if you disagree on a design direction for 10 minutes, try one and see how it works in practice.

Class-level refactorings 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 refactorings

Avoid creating TODO comments or story/task cards for postponed refactorings. If the problem is common enough for you or others to notice it again, it will get fixed eventually.

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 .
- These architectures are implemented through the use of recurring patterns. They aren’t design patterns in the formal Gang of Four* sense
- 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.
- 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.
- Before introducing a new pattern, ask yourself if the duplication is really necessary.
- In my experience, breakthroughs in architecture happen every few months.
- 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.
- Keep delivering stories while you refactor. Balance technical excellence with delivering value.
- Introducing architectural patterns incrementally helps reduce the need for multi iteration refactorings. It’s easier to expand an architecture than it is to simplify one that’s too ambitious.

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.

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.

For example, what if you know that internationalizing your code is expensive and only going to get more expensive? Your power lies in your ability to choose which refactorings to work on.

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.

Performance Optimization

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 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

All passed. Unfortunately, that had no effect on the profile. We rolled back the change. Next, we agreed to implement the cache. The test failed. fixed the bug in the cache code, and all tests passed again.

Unfortunately, all that effort was a waste. The performance was actually slightly worse than before.

They found a method that was trying to making an unnecessary network connection on each transaction. Fixing it lopped three full seconds off each transaction. They had fixed the problem in less than half an hour.

Cache coherency requires that the data in the cache change when the data in the underlying data store changes and vice versa. It's easy to get wrong.

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 predict performance by counting instructions are long gone.
- 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.
- Measure the performance of the entire system, make an educated guess about what to change, then remeasure.

- Usually, your performance test will be an end-to-end test. Although I avoid end-to-end tests in other situations earlier in 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.
- Use a profiler to guide your optimization efforts.

When to Optimize

- Performance optimizations must serve the customer's needs.
- 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.
- XP has an excellent mechanism for prioritizing customer needs: the combination of user stories and release planning.
- 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.
- 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.