

INTRODUCTION TO SOFTWARE ENGINEERING

BY
DR. PRAVEEN KANTHA

EXTREME PROGRAMMING (XP)

- Extreme programming (XP), introduced in 1996 lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software
- It is the type of Agile Development we develop software with small team where the software requirements changes rapidly.
- It provides values and principles to guide the team behavior



EXTREME PROGRAMMING (XP) VALUES

Extreme Programming (XP) is based on the five values –

- Communication
- Simplicity
- Feedback
- Courage
- Respect



Communication

- Extreme Programming emphasizes continuous and constant communication among the team members, managers and the customer.
- The Extreme Programming practices, such as unit testing, pair programming, simple designs, common metaphors, collective ownership and customer feedback focus on the value of communication.

Simplicity

Extreme Programming believes in ‘it is better to do a simple thing today and pay a little more tomorrow to change it’ than ‘to do a more complicated thing today that may never be used anyway’.




Feedback

- Every iteration commitment is taken seriously by delivering a working software.
- The software is delivered early to the customer and a feedback is taken so that necessary changes can be made if needed.

Courage

Extreme Programming provides courage to the developers in the following way –

- To focus on only what is required
 - To communicate and accept feedback
 - To tell the truth about progress and estimates
 - To adapt to changes whenever they happen
- 

Respect

Respect is a deep value, one that lies below the surface of the other four values. In Extreme Programming,

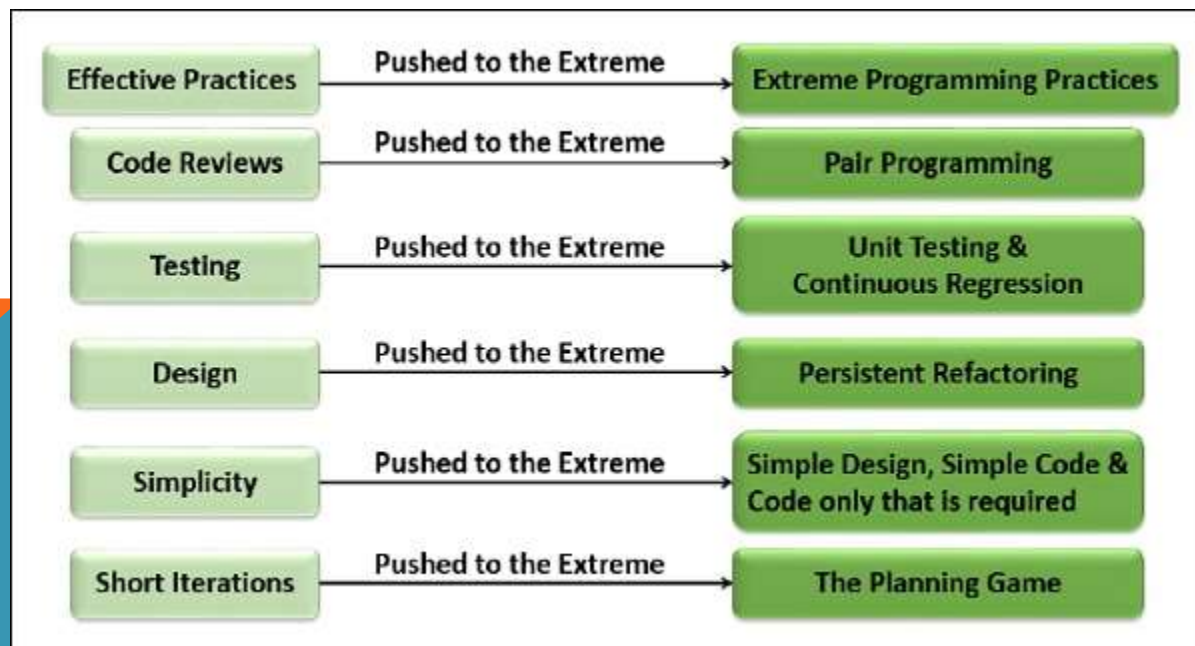
- Everyone respects each other as a valued team member.
- Everyone contributes value such as enthusiasm.
- Developers respect the expertise of the customers and vice versa.
- Management respects the right of the developers to accept the responsibility and receive authority over their own work.



Why is it called “Extreme?”

Extreme Programming takes the effective principles and practices to extreme levels.

- Code reviews are effective as the code is reviewed all the time.
- Testing is effective as there is continuous regression and testing.
- Design is effective as everybody needs to do refactoring daily.
- Integration testing is important as integrate and test several times a day.
- Short iterations are effective as the planning game for release planning and iteration planning.



XP is summed up or based on twelve Practices. . .



1

The Planning Process

- 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

Small Releases

- This practice suggests releasing the MVP quickly and further developing the product by making small and incremental updates.
- An MVP, or a minimum viable product, is the earliest version of a product that has only required features, enough to deliver the core value and verify it to early customers.
- MVP is deployed to gather feedback and see whether the product is needed by users at all.
- Small releases allow developers to frequently receive feedback, detect bugs early, and monitor how the product works in production. One of the methods of doing so is the continuous integration practice (CI) we mentioned before.

3

Metaphor

- A Metaphor is an expression, often found in literature that describes a person or object by referring to something that is considered to have similar characteristics to that person or object. For example, ‘The mind is an ocean’ and ‘the city is a jungle’ are both Metaphors.
- You can think of metaphor as the architecture of the system to be built in a way that it is easily understandable by everyone involved in the development.
- System metaphor stands for a simple design that has a set of certain qualities.
- First, a design and its structure must be understandable to new people.
- They should be able to start working on it without spending too much time examining specifications.

4

Simple Design

The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

The right design for the software at any given time is the one that –

- Runs all the tests
- Has no duplicated logic like parallel class hierarchies
- States every intention important to the developers
- Has the fewest possible classes and methods



5

Testing

- The developers continually write unit tests, which need to pass for the development to continue.
- The customers write tests to verify that the features are implemented. The tests are automated so that they become a part of the system and can be continuously run to ensure the working of the system.
- The result is a system that is capable of accepting change.



6

Refactoring

- Refactoring is the process of restructuring code, while not changing its original functionality.
- The goal of refactoring is to improve internal code by making many small changes without altering the code's external behavior.
- The goal of this technique is to continuously improve code.
- Refactoring is about removing redundancy, eliminating unnecessary functions, increasing code coherency, and at the same time decoupling elements. ***Keep your code clean and simple, so you can easily understand and modify it when required*** would be the advice of any XP team member.

Pair Programming

- This practice requires two programmers to work jointly on the same code.
- While the first developer focuses on writing, the other one reviews code, suggests improvements, and fixes mistakes along the way.
- Such teamwork results in high-quality software and faster knowledge sharing but takes about 15 percent more time. In this regard, it's more reasonable trying pair programming for long-term projects.



8

Collective Ownership

- This practice declares a whole team's responsibility for the design of a system.
- Each team member can review and update code.
- Developers that have access to code won't get into a situation in which they don't know the right place to add a new feature.
- The practice helps avoid code duplication.
- The implementation of collective code ownership encourages the team to cooperate more and feel free to bring new ideas.



9

Continuous Integration

- Developers always keep the system fully integrated.
- XP teams take iterative development to another level because they commit code multiple times a day, which is also called continuous delivery.
- XP practitioners understand the importance of communication.
- Programmers discuss which parts of the code can be re-used or shared.



10

40-Hour Week

- Extreme Programming emphasizes on the limited number of hours of work per week for every team members, based on their sustainability, to a maximum of 45 hours a week.
- If someone works for more time than that, it is considered as overtime. Overtime is allowed for at most one week.
- This practice is to ensure that every team member be fresh, creative, careful and confident.

11

On-Site Customer

Include a real, live user on the team, available full-time to answer the questions, resolve disputes and set small-scale priorities. This user may not have to spend 40 hours on this role only and can focus on other work too.

On-Site Customer – Advantages

The advantages of having an onsite customer are –

- Can give quick and knowledgeable answers to the real development questions.
- Makes sure that what is developed is what is needed.
- Functionality is prioritized correctly.




Coding Standard

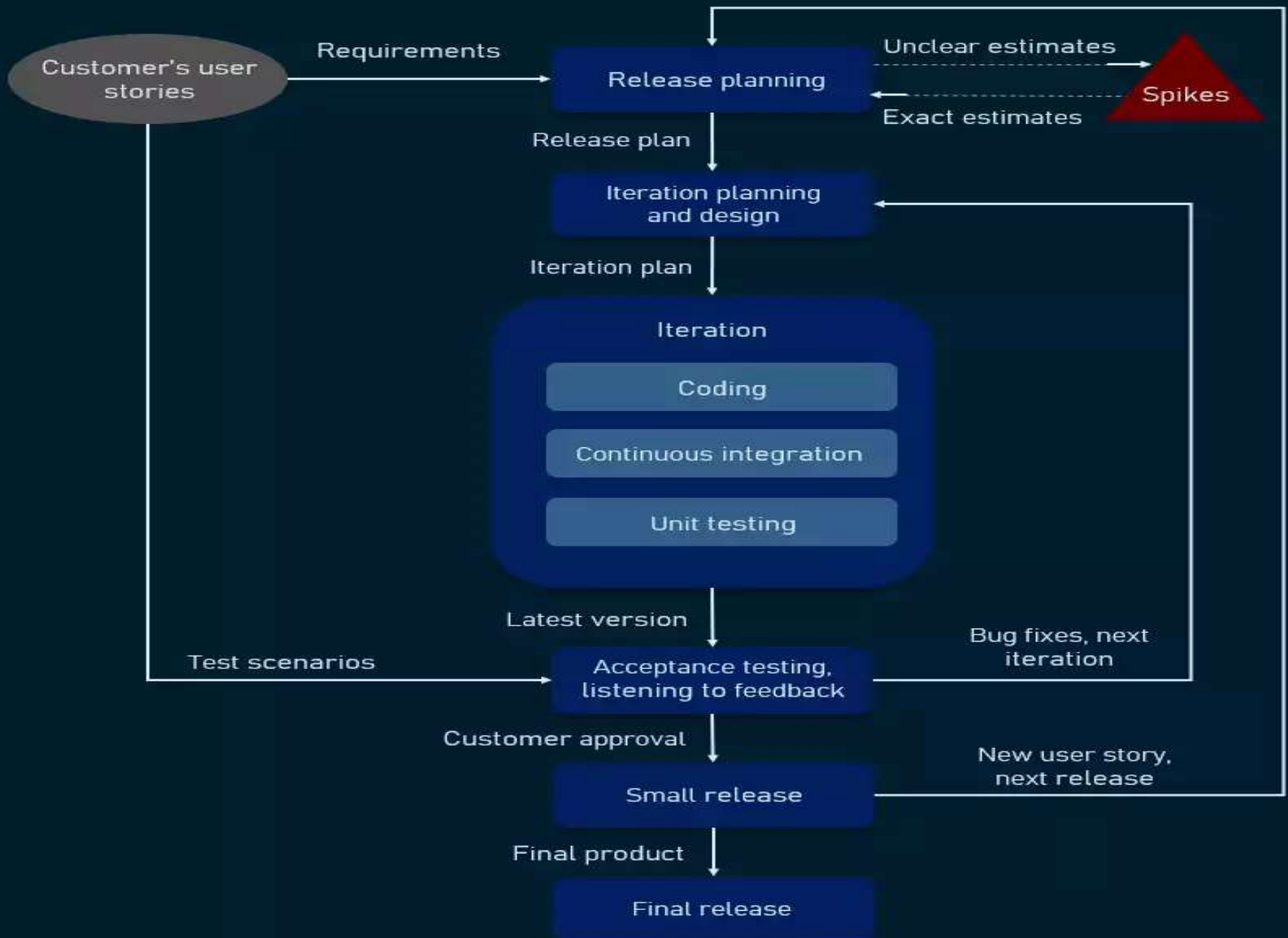
Developers write all code in accordance with the rules emphasizing-

- Communication through the code.
- The least amount of work possible.
- Consistent with the “once and only once” rule (no duplicate code).
- Voluntary adoption by the whole team.

These rules are necessary in Extreme Programming because all the developers –

- Can change from one part of the system to another part of the system.
 - Swap partners a couple of times a day.
 - Refactor each other's code constantly.
- 

EXTREME PROGRAMMING LIFECYCLE




EXTREME PROGRAMMING ADVANTAGES

- Continuous testing and refactoring practices help create **stable well-performing systems** with minimal debugging;
- Simplicity value implies creating a **clear, concise code** that is easy to read and change in the future if needed;
- **Documentation is reduced** as bulky requirements documents are substituted by user stories;
- No or **very little overtime** is practiced;
- Constant communication provides a high level of **visibility and accountability** and allows all team members to keep up with the project progress;
- Pair programming has proven to result in **higher-quality products** with fewer bugs; most research participants also reported enjoying such collaboration more and feeling more confident about their job;
- **Customer engagement ensures their satisfaction** as their participation in the development and testing process can directly influence the result, getting them exactly what they wanted.

EXTREME PROGRAMMING DISADVANTAGES

- In many instances, the customer has no clear picture of the end result, which makes it almost **unrealistic to accurately estimate scope, cost, and time**;
- **Regular meetings with customers often take a great deal of time** that could instead be spent on actual code writing;
- **Documentation can be scarce** and lack clear requirements and specifications, leading to project scope creep;
- The rapid transition from traditional methods of software development to extreme programming demands significant **cultural and structural changes**;
- **Pair programming takes more time** and doesn't always work right due to the human factor and character incompatibility;
- **XP works best with collocated teams** and customers present in person to conduct face-to-face meetings, limiting its application with distributed teams;

ADAPTIVE SOFTWARE DEVELOPMENT (ASD)

- Adaptive Software Development is one of the earliest agile methodologies.
 - It was a result of the work by Jim Highsmith and Sam Bayer on Rapid Application Development (RAD).
 - This methodology interestingly validates the fact that it is quite normal to have continuous adaptation to the software development process.
 - This validation removes the fear of the unknown and uncertainty often involved in any software development cycle.
- 

HISTORY OF ADAPTIVE SOFTWARE DEVELOPMENT (ASD)

- Adaptive Software Development in software engineering is a classic example of *necessity is the mother of invention adage*.
- In the early 1990s, when Sam Bayer and Jim Highsmith were making the most out of RAD and creating RADical Software Development, they realized that these approaches were not enough.
- It was important to have a process that encouraged collaboration inside the organization as well as with the clients.
- They realized that they needed the process that makes all the stakeholders at ease with the inherent uncertainty of software development while making room for continuous, consistent, and genuine learning during the process itself.
- It is out of these needs that Adaptive Software Development came into being.

ADAPTIVE SOFTWARE DEVELOPMENT (ASD)

- ASD is a development methodology that encourages continuous learning throughout the software development project.
- ASD strongly advocates a software development process that is fun to be in as well as natural or organic.
- Adaptive Software Development is a method to build complex software and system.



ADAPTIVE SOFTWARE DEVELOPMENT (ASD) LIFE CYCLE

- The adaptive life cycle is an evolution of the spiral life cycle that started during the mid-1980s.
- The major problem with the spiral life cycle was its reliance of predictability.
- Though RAD was less predictable and deterministic, the mindset of the practitioners of the spiral life cycle was still not changed.
- The adaptive life cycle in Adaptive Software Development tries to address this issue of changing the mindset by way of reflection and naming the phases during this process exactly as per that reflection.
- These phases of the adaptive life cycle that accept uncertainty and chaos as “normal” during the software development process are

1. Speculation
2. Collaboration,
3. Learning.

SPECULATION

- Speculation phase carefully and intentionally removes the factor of planning that often brings with it lots of unnecessary baggage and tension.
- This phase gives the teams full liberty to welcome and accept the outcomes without the fear of the unknown or uncertainty. It totally eliminates the toxic need to be right all the time, putting all the stakeholders at ease.
- In the speculation phase:
 - ✓ A project mission statement is defined
 - ✓ Creation and sharing of the general idea of the goals to be achieved take place
 - ✓ Teams adopt the tools that would assist them in adapting and changing as per the requirements during the entire cycle of the project.

SPECULATION

- Mostly, the speculation phase finds itself divided into two steps.
 - ✓ project initiation
 - ✓ adaptive planning

Project initiation

- The first step of initiation involves stuff that serves as the project's foundation.
- This includes project management information, mission statement and other essential tools and information.




SPECULATION

Adaptive planning:

In adaptive planning, different product features get assigned to the different teams as per their expertise and skills.

This requires the teams to decide:

- Time box for the project
 - Number of development cycles
 - Amount of time each cycle would take as per the unanimously decided timebox
 - A theme and objective for the cycle
 - Assignment of components for each cycle
 - A task list for the project
- 

SPECULATION

The purpose of the speculation phase is:

- To remove the unnecessary burden of planning
- To create space for innovation by making the entire process open-ended
- To keep the planning at its most minimum or essential
- To set an appropriate framework for the end product
- To allow exploration and experimentation with each new speculation phase with small iterations.



COLLABORATION

- It is in this phase that the actual development begins.
- This phase is about group emergence.
- It is about coming together of diverse experiences, knowledge, and skills.
- This forms a collaborative environment where diversity serves as the building block for creativity and innovation during the entire development cycle.



LEARNING.

- The Learning part of the Lifecycle is vital for the success of the project.
- Team has to enhance their knowledge constantly, using practices such as-
 - Technical Reviews
 - Project Retrospectives
 - Customer Focus Groups

Adaptive cycle planning
mission statement
project constraints basic
requirements time boxed
released plans

Speculation

Requirements gathering
JAD
mini-specs

Collaboration

Release
Software increment
adjustments for subsequent
cycles

Learning

components
implemented/tested focus
groups for feedback
formal technical reviews
postmortems



1. Speculation:

During this phase project is initiated and planning is conducted. The project plan uses project initiation information like project requirements, user needs, customer mission statement, etc., to define set of release cycles that the project wants.

2. Collaboration:

It is the difficult part of ASD as it needs the workers to be motivated. It collaborates communication and teamwork but emphasizes individualism as individual creativity plays a major role in creative thinking. People working together must trust each others to

- Assist without resentment,
 - Work as hard as possible,
 - Possession of skill set,
 - Communicate problems to find effective solution
- 