AGILE SOFTWARE DEVELOPMENT

Digital Notes

UNIT – II

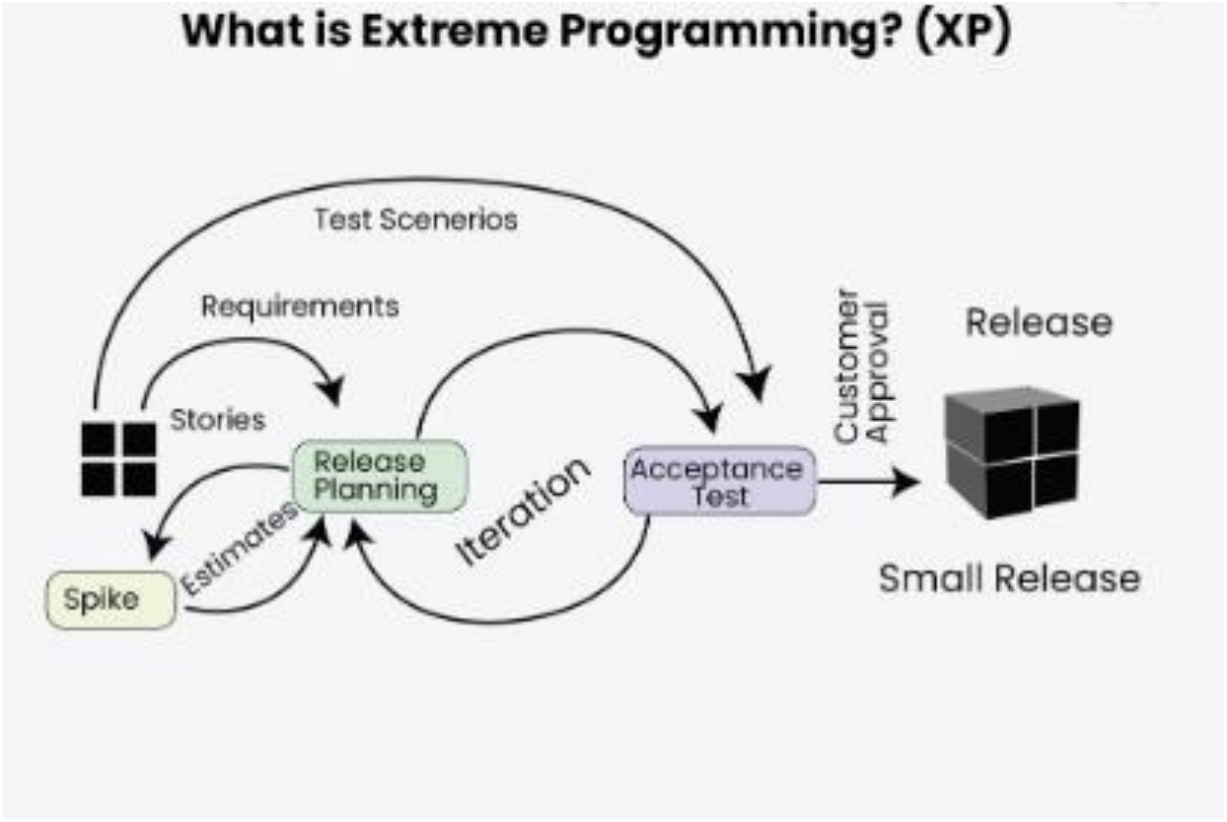
EXTREME PROGRAMMING

Extreme Programming

- Introduction
- core XP values
- The twelve XP practices
- About extreme programming
- Planning XP projects
- Test first coding
- making pair programming work

Introduction - Extreme Programming

- Originally designed as a way of supporting small development teams working within uncertain and changing requirements.
- XP was designed as an approach based on software engineering principles, but focused on the timely delivery of software that meets users' requirements (rather than on the sometimes over bearing processes that surround the development of software).
- An important aspect of XP is the empowerment of the actual developers
- XP also places great emphasis on the **software development team and teamwork**.
- The team, in turn, incorporates management, technical personnel and end users all cooperating towards the common good.
- It takes as one of its aims that **teams communicate and constantly** pay attention to all the details necessary to make sure that the software being **developed matches the user requirements**, to help to produce quality software.



- Extreme Programming (XP) is an Agile software development methodology that focuses on delivering high-quality software through frequent and continuous feedback, collaboration, and adaptation. XP emphasizes a close working relationship between the development team, the customer, and stakeholders, with an emphasis on rapid, iterative development and deployment.

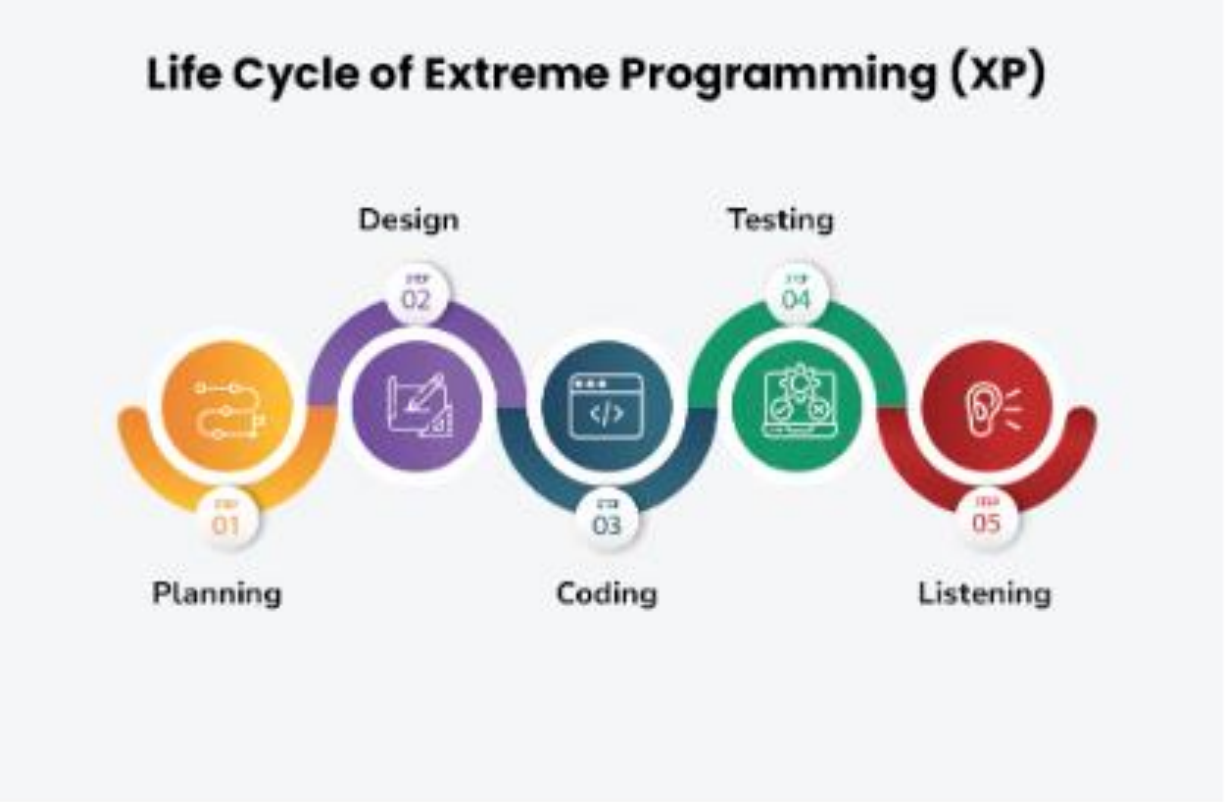
- Agile development approaches evolved in the 1990s as a reaction to documentation and bureaucracy-based processes, particularly the waterfall approach. Agile approaches are based on some common principles, some of which are:

    - Working software is the key measure of progress in a project.
    - For progress in a project, therefore software should be developed and delivered rapidly in small increments.
    - Even late changes in the requirements should be entertained.
    - Face-to-face communication is preferred over documentation.
    - Continuous feedback and involvement of customers are necessary for developing good-quality software.

- A simple design that involves and improves with time is a better approach than doing an elaborate design up front for handling all possible scenarios.

- The delivery dates are decided by empowered teams of talented individuals.

- Extreme programming is one of the most popular and well-known approaches in the family of agile methods. an XP project starts with user stories which are short descriptions of what scenarios the customers and users would like the system to support. Each story is written on a separate card, so they can be flexibly grouped.



Life Cycle of Extreme Programming (XP)

- **Planning:** The first stage of Extreme Programming is planning. During this phase, clients define their needs in concise descriptions known as user stories. The team calculates the effort required for each story and schedules releases according to priority and effort.

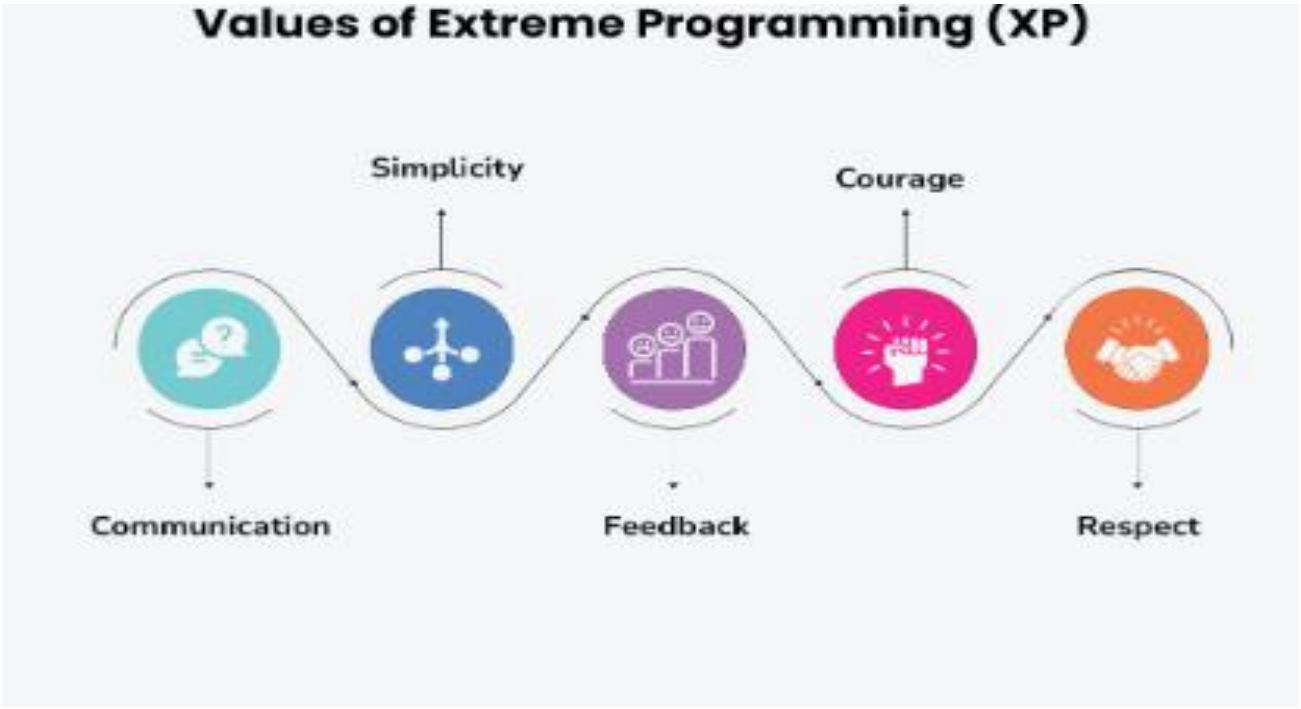
- **Design:** The team creates only the essential design needed for current user stories, using a common analogy or story to help everyone understand the overall system architecture and keep the design straightforward and clear.

- **Coding:** Extreme Programming (XP) promotes pair programming i.e**.** wo developers work together at one workstation, enhancing code quality and knowledge sharing. They write tests before coding to ensure functionality from the start (TDD), and frequently integrate their code into a shared repository with automated tests to catch issues early.

- **Testing**: Extreme Programming (XP) gives more importance to testing that consist of both unit tests and acceptance test. Unit tests, which are automated, check if specific features work correctly. Acceptance tests, conducted by customers, ensure that the overall system meets initial requirements. This continuous testing ensures the software's quality and alignment with customer needs.

- **Listening**: In the listening phase regular feedback from customers to ensure the product meets their needs and to adapt to any changes.

Core XP Values



- **Communication:** The essence of communication is for information and ideas to be exchanged amongst development team members so that everyone has an understanding of the system requirements and goals. Extreme Programming (XP) supports this by allowing open and frequent communication between members of a team.

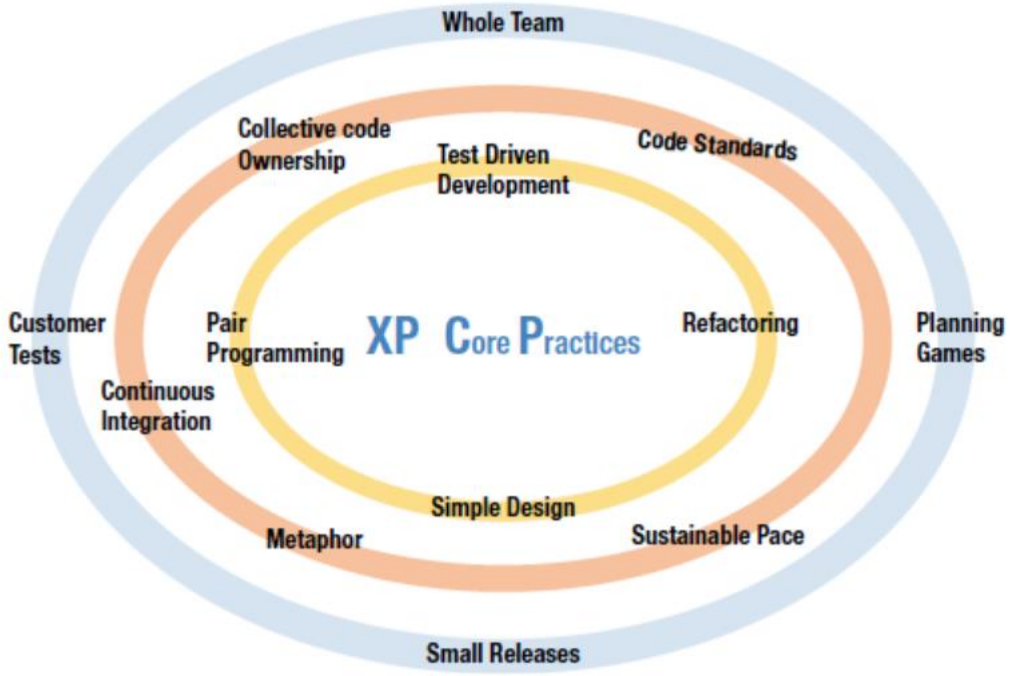
- **Simplicity:** Keeping things as simple as possible helps reduce complexity and makes it easier to understand and maintain the code.

- **Feedback:** Feedback loops which are constant are among testing as well as customer involvements which helps in detecting problems earlier during development.

- **Courage:** Team members are encouraged to take risks, speak up about problems, and adapt to change without fear of repercussions.

- **Respect**: Every member's input or opinion is appreciated which promotes a collective way of working among people who are supportive within a certain group.

The Twelve XP practices



- **Planning Game**

1. In XP the main planning process is called Planning Game. There are 2 levels of plans in XP; level one is release planning and level 2 is iteration planning.

2. In both levels, there are 3 steps

    - exploration
    - commitment
    - steering

3. The first phase is releasing planning. The team and stakeholders/customers collaboratively decide what are the requirements and features that can be delivered into production and when. This is done based on priorities, capacity, estimations, and risk factors of the team to deliver.

4. In Iteration planning the team will pick up the most valuable items from the list and break them down into tasks then estimates and a commitment to delivering at the end of the iteration.

- **Simple Design**

1. In the XP, the team will not do complex or big architecture and designs upfront; instead, the team will start with a simple design and let it emerge and evolve over a period of iterations. The code is frequently refactored so that it will be maintainable and free of technical debt. Simple designs make the 'definition of done' easier.

2. XP teams conduct a small test or proof-of-concept workout called spike. The outcome of the spike helps the team to apprehend the validity of the hypothesis, gauge the complexity of the solution, and feel assured to estimate and build something primarily based on the test.

- **Test-Driven Development (TDD)**

1. In XP Developers write the unit test cases before starting the coding. The team automates the unit tests and it helps during the build and integration stage. The main benefit of TDD is that programmers have to only write the code that passes the tests.

2. Basics steps of TDD

    - Write the unit test case first
    - Write a minimal amount of code to pass the test.
    - Refactor it by adding the needed feature and functionality, While continuously making sure the tests pass.

- **Code Standard**

1. Organizations want their programmers to hold to some well-described and standard style of coding called coding standards. It is a guideline for the development team as in XP. Since there are multiple programming pairs at play coding standards are very useful to make consistency in code, style, naming conversion, exception handling, and use of parameters.

2. These standards must be defined and agreed upon before the team starts the coding.

3. It will make the code simple to understand and help detect the problem or issues quickly and also increase the efficiency of the software.

- **Refactoring**

1. Refactoring as the word suggests, is restructuring or reconstructing existing things. In XP over a period of time the team produces lots of working code increases the complexity and contributes to technical debt. To avoid this, we should consider the below points,

    - Ensure code or functions are not duplicated.
    - Ensure all variables in scope, are defined and used.
    - No long functions or methods
    - Removing unnecessary stuff and variables
    - Proper use of access modifiers etc.

By refactoring, the programmers look to improve the overall code quality and make it more readable without altering its behavior.

- **Pair Programming**

1. This is my favorite and most used practice. Pair programming consists of two programmers operating on the same code and unit tase cases, on the same system (one display and one keyboard). One programmer plays the pilot role focuses on clean code, and compiles and runs. The second one plays the role of a navigator focuses on the big picture and reviews code for improvement or refactoring.

2. Every hour or given a period of time this pair is allowed to switch roles so that the pilot will play the role of navigator and vice versa.

3. The pairs of pilots and navigators are also not fixed and they are frequently swapped, the main benefit of that over a period of time is that everyone gets to know about the code and functionality of the whole system.

- **Collective Code Ownership**

1. By following pair programming practices the XP team always takes collective ownership of code. Success or failure is a collective effort and there is no blame game. There is no one key player here, so if there is a bug or issue then any developer can be called to fix it.

- **Continuous Integration**

2. CI is Continuous Integration. In XP, Developers do pair programming on local versions of the code. There is a need to integrate changes made every few hours or on a daily basis so after every code compilation and build we have to integrate it so that all the tests are executed automatically for the entire project.

3. If the tests fail, they are fixed then and there, so that any chance of defect propagation and further problems are avoided with minimum downtime.

4. The team can use CI tools like Jenkins, shippable, Integrity, Azure DevOps Pipelines, etc.

- **Small Release**

1. A cross-functional team in XP releases Minimum Viable Product (MVP) frequently. Small releases also help to break down complex modules into small chunks of code. This helps the developer team as well as the on-site customer to demonstrate the product and focus only on the least amount of work that has the highest priority.

- **System Metaphor**

1. This is majorly connected with the user story, the story must be simple enough to be easily understood by users and developers and to relate it with code.

2. It could be a naming conversion practice used in design and code to have a shared understanding between teams. For example, Order_Food() is easily explained -- this will be used to order food.

3. It is easy for the team to relate to the functionality of the particular component by just looking at its name.

- **Onsite Customer**

1. This is a similar role to a Product Owner in Scrum. The onsite customer plays a major role here and is responsible for crafting the vision, defining user stories and acceptance criteria, the definition of done, and release planning.

2. They are the experts who know the domain or product and know how to generate a return on investment (ROI) by delivering the minimum viable product (MVP).

3. If the onsite customer role is not full-time, the role can be filled with product managers, product owners, UI-UX designers, and business analysts who are called proxies.

4. The word “on-site” implies that the customers or their proxies sit together with the rest of the team to ensure that communication flows freely.

- Sustainable Pace

1. This is a people-centric practice. In XP practices like TDD, continuous integration and refactoring of code help to proactively improve the quality and stability of the working software.

2. XP maintains a sustainable pace by introducing downtime during the iteration. The team is not doing actual development at this time but acts as a buffer to deal with uncertainties and issues. Teams can use the slack time to pay down technical debt by refactoring code or doing research to keep up the pace.

About Extreme Programming

- When you look at the twelve practices that define Extreme Programming or XP, you will probably notice or at least comment on, the fact that there is very little new here.

- Many of the practises are, or have been incorporated into, described by, or presented as, parts of various existing software engineering methodologies

- So **what is so different, so extreme, about Extreme Programming?** The answer lies in a number of places:

1. Extreme Programming is very lightweight – it really only focuses on the programming of a software system.

2. Extreme Programming takes the best practices to their ultimate conclusions. We can say that if we picture each of the practices as a knob on a control board and turned each knob up to maximum and looked at what happened. What happened was XP!

- If we consider the second point in more detail, the concept goes as follows:

1. If code reviews are good, then review code all the time (pair programming).

2. If testing is good, everybody will test all the time (unit testing), even customers (acceptance testing).

3. If designing is good, then make it part of what everyone does every day (refactoring).

4. If simplicity is good, then always strive for the simplest effective solution (i.e., the simplest solution that works).

5. If architecture is important, then ensure that everyone is involved in creating and refining the architecture all the time (system metaphor).

6. If integration testing is good, then integration and testing should be an ongoing (daily or even hourly) thing (continuous integration).

7. If short iterations are good, then make them as short as possible, i.e., hours, or days not weeks and months (the Planning Game).

- As you can see from this, each principle has been taken to the Extreme and this is what makes Extreme Programming Extreme!

Planning XP projects

- Planning XP projects – the very idea! It is a common misconception that XP projects do not need to be planned.

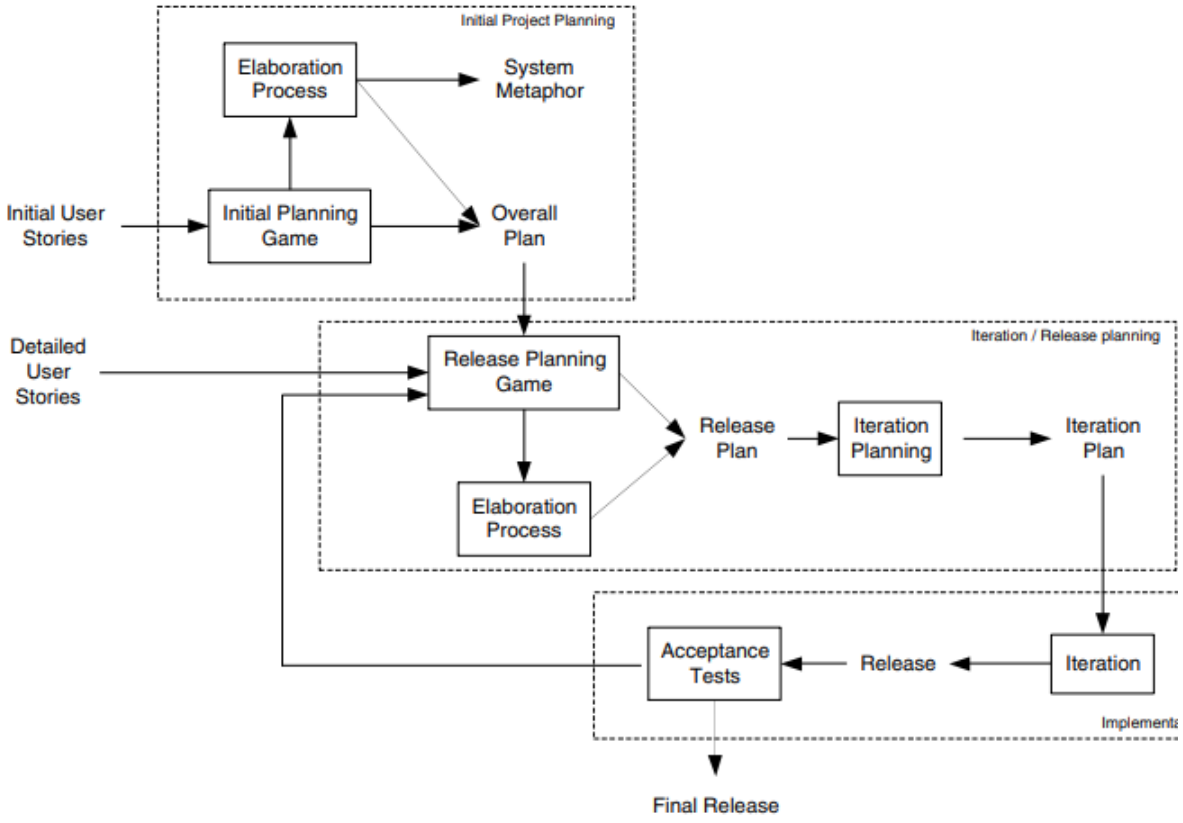
- This may be because; they may remain un-convinced because the nature of project planning changes. Rather than planning out the whole project in great and fine-grained detail, the overall project plan is left rather vague and high level and detailed plans are only created on an iteration-by-iteration or release-by-release basis.

- From personal experience, I can vouch that not only does this approach work very well but that it actually involves far more planning, which is more accurate reflecting what is actually happening on the project and is reviewed more often than the traditional approach

- So what is the aim of the game/planning workshop? It is to decide on the scope and priorities of the project and of the releases. It is also to estimate the cost of various features required by the software and to schedule those features into releases.

- The XP project lifecycle is presented in Figure below. This diagram illustrates the various planning stages and implementation stages within a typical XP project.

- For example, it starts with an initial project planning process during which the overall plan of the project is roughly sketched out. This is followed by one release planning process where the contents of a release are planned and the tasks performed in the iteration to implement the release are also planned. The release is then implemented and the results of this process fed back into the planning for the next iteration.



Test first coding

- When you create your tests first, before the code, you will find it much easier and faster to create your code. The combined time it takes to create a unit test and create some code to make it pass is about the same as just coding it up straight away. But, if you already have the unit tests you don't need to create them after the code saving you some time now and lots later.
- Creating a unit test helps a developer to really consider what needs to be done.
- Requirements are nailed down firmly by tests.
- You also have immediate feedback while you work. It is often not clear when a developer has finished all the necessary functionality. If we create our unit tests first then we know when we are done; the unit tests all run.
- With respect to test-first coding, modelling may seem at best superfluous and at worst contradictory. This is because, in test-first coding, you essentially follow this cycle:

  1. Write a test.
  2. Write the code to be tested.
  3. Run the test/get the code to work.
  4. If the test has passed, then return to step 1 until finished.

- Another point at which Agile Modelling may be relevant is once a test has been written and you need to consider how to implement the business code.

Making pair programming work

- The idea behind pair programming is of course very simple, essentially it comes done to “two heads are better than one” most of the time. In addition, in pair programming all code is always reviewed by at least one other person who is focused on what the code needs to do.
- In pair programming, one developer takes hands on control of the keyboard and mouse, while the other monitors what the “driver” is doing (often referred to as the “navigator”). This requires active participation from both sides. But this is not how most programmers have been trained to work. Most programmers expect to be the “driver,” they are not attuned to being the “navigator

1. Engage in a dialogue. The “driver” should try and explain what they are doing. The “navigator” should ask questions in order to understand what is being done. This is not as simple as it sounds. For example, the next time you are in your car and driving, try telling an imaginary passenger what you are doing, what you are considering and what concerns you have about the road ahead (it helps if the passenger is imaginary so that none of your friends think you are mad!).
2. Listen to each other. If one member of the pair is doing all the talking then it is probably isn’t working. The one who isn’t saying much might not be clear on what is being done. Try swapping roles, or have a break, or take time out to review where you are.
3. Take frequent breaks. Pair programming is intensive. It is intensive in terms of the little grey cells and also in terms of inter-personal communications and dynamics. Have regular breaks, talk about other things, see how others are getting on, catch up on the news, etc. Don’t take a break and discuss the code!
4. Make pair programming practical. Providing enough space to allow a pair to work together comfortably isn’t essential but it can help. This may only go as far as having desks that are big enough for two developers to sit next to each other, or it may involve special double size workstation environments.
5. Allowing non-pair time. There are some situations where allowing a pair to work alone can be beneficial. This may be a little controversial for some in the XP community, as they will argue that it is always between to work in pairs.
6. However, in the following situations, flying solo may be an alternative (but it should be the rare exception to pair programming):
    - Exploring completing alternative solutions,
    - Following multiple competing lines of investigation during debugging

    (But not bug fixing).
7. Change partners often. Pairs should not be permanent. Instead, pairs should change as and when required. The typical point at which to change is at the start of a new task. However, if a task ranges over a number of areas, then a “driver” developer may pair with several “navigators” in order to benefit from their different areas of expertise.