1. Planning:

- Determine features to be included in next release

- Business priorities and technical estimates

The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The planning process is divided into two parts:

• Release Planning: This is focused on determining what requirements are included in which near-term releases, and when they should be delivered. The customers and developers are both part of this. Release Planning consists of three phases:

• Exploration Phase: In this phase the customer will provide a shortlist of high-value requirements for the system. These will be written down on user story cards.

• Commitment Phase: Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release.

• Steering Phase: In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed.

• Iteration Planning: This plans the activities and tasks of the developers. In this process the customer is not involved. Iteration Planning also consists of three phases:

• Exploration Phase: Within this phase the requirement will be translated to different tasks. The tasks are recorded on task cards.

• Commitment Phase: The tasks will be assigned to the programmers and the time it takes to complete will be estimated.

• Steering Phase: The tasks are performed and the end result is matched with the original user story.

The purpose of the Planning Game is to guide the product into delivery. Instead of predicting the exact dates of when deliverables will be needed and produced, which is difficult to do, it aims to "steer the project" into delivery using a straightforward approach.[2] The Planning Game approach has also been adopted by non-software projects and teams in the context of business agility

2. Short release:

The delivery of the software is done via frequent releases of live functionality creating concrete value. The small releases help the customer to gain confidence in the progress of the project. This helps maintain the concept of the whole team as the customer can now come up with his suggestions on the project based on real experience.

- Get a working system quickly

- Release new versions in short cycles (2 to 4 weeks)

- Base new detail plans on customer feedback

3. Metaphor:

- Use a metaphor instead of an architecture

The system metaphor is a story that everyone - customers, programmers, and managers - can tell about how the system works. It's a naming concept for classes and methods that should make it easy for a team member to guess the functionality of a particular class/method, from its name only. For example a library system may create loan\_records(class) for borrowers(class), and if the item were to become overdue it may perform a make\_overdue operation on a catalogue(class). For each class or operation the functionality is obvious to the entire team.

4. Simple design

- Eliminate unnecessary complexity as soon as possible

- Design can be changed in later versions

Programmers should take a "simple is best" approach to software design. Whenever a new piece of code is written, the author should ask themselves 'is there a simpler way to introduce the same functionality?'. If the answer is yes, the simpler course should be chosen. Refactoring should also be used, to make complex code simpler

5. Test-driven development

- Continuous and automated

- Write tests before writing code

Unit tests are automated tests that test the functionality of pieces of the code (e.g. classes, methods). Within XP, unit tests are written before the eventual code is coded. This approach is intended to stimulate the programmer to think about conditions in which his or her code could fail. XP says that the programmer is finished with a certain piece of code when he or she cannot come up with any further conditions under which the code may fail.

Test driven development proceeds by quickly cycling through the following steps, with each step taking minutes at most, preferably much less. Since each user story will usually require one to two days of work, a very large number of such cycles will be necessary per story.

• Write unit test: The programmers write a minimal test that should fail because the functionality hasn't been fully implemented in the production code.

• Watch the new test fail: The programmers verify the test does indeed fail. While it may seem like a waste of time, this step is critical because it verifies that your belief about the state of the production code is correct. If the test does not fail, the programmers should determine whether there is a bug in the test code, or that the production code does support the functionality described by the new test.

• Write code: The programmers write just enough production code so the new test will pass.

• Run test: The unit tests are executed to verify that the new production code passes the new test, and that no other tests are failing.

• Refactor: Remove any code smells from both the production and test code.

For a more intense version of the above process, see Uncle Bob's Three Rules of TDD[4].

6. Design improvement (Refactoring)

- Remove duplication, improve communication, and simplify or add needed flexibility

7. Pair programming

- Two programmers working at same machine

Pair programming means that all code is produced by two people programming on one task on one workstation. One programmer has control over the workstation and is thinking mostly about the coding in detail. The other programmer is more focused on the big picture, and is continually reviewing the code that is being produced by the first programmer. Programmers trade roles after minute to hour periods.

The pairs are not fixed; programmers switch partners frequently, so that everyone knows what everyone is doing, and everybody remains familiar with the whole system, even the parts outside their skill set. This way, pair programming also can enhance team-wide communication. (This also goes hand-in-hand with the concept of Collective Ownership).

8. Collective ownership

- Anyone can change any code at any time

Collective code ownership (also known as "team code ownership" and "shared code") means that everyone is responsible for all the code; therefore, everybody is allowed to change any part of the code. Collective code ownership is not only an organizational policy but also a feeling. "Developers feel team code ownership more when they understand the system context, have contributed to the code in question, perceive code quality as high, believe the product will satisfy the user needs, and perceive high team cohesion."[7] Pair programming, especially overlapping pair rotation, contributes to this practice: by working in different pairs, programmers better understand the system context and contribute to more areas of the code base.

Collective code ownership may accelerate development because a developer who spots an error can fix it immediately, which can reduce bugs overall. However, programmers may also introduce bugs when changing code that they do not understand well. Sufficiently well-defined unit tests should mitigate this problem: if unforeseen dependencies create errors, then when unit tests are run, they will show failures.

9. Continuous integration

- Integrate and test every time a task is completed (many times a day)

10. Sustainable pace

- 40 hours a week reasonable, need relax

- Never overtime two consecutive weeks,

The concept is that programmers or software developers should not work more than 40 hour weeks, and if there is overtime one week, that the next week should not include more overtime. Since the development cycles are short cycles of continuous integration, and full development (release) cycles are more frequent, the projects in XP do not follow the typical crunch time that other projects require (requiring overtime).

Also, included in this concept is that people perform best and most creatively if they are rested.

A key enabler to achieve sustainable pace is frequent code-merge and always executable & test covered high quality code. The constant refactoring way of working enforces team members with fresh and alert minds. The intense collaborative way of working within the team drives a need to recharge over weekends.

Well-tested, continuously integrated, frequently deployed code and environments also minimize the frequency of unexpected production problems and outages, and the associated after-hours nights and weekends work that is required.

11. On-site customer

- Include real customer on the team

12. Coding standards

- Everyone needs to use the same rules

Coding standard is an agreed upon set of rules that the entire development team agree to adhere to throughout the project. The standard specifies a consistent style and format for source code, within the chosen programming language, as well as various programming constructs and patterns that should be avoided in order to reduce the probability of defects.[5] The coding standard may be a standard conventions specified by the language vendor (e.g. The Code Conventions for the Java Programming Language, recommended by Sun), or custom defined by the development team.

Extreme Programming backers advocate code that is self-documenting to the furthest degree possible. This reduces the need for code comments, which can get out of sync with the code itself.[6]

Trả lời: 12 điều kiện trong XP

1. Lập kế hoạch

- Xác định các tính năng sẽ được bao gồm trong phiên bản tiếp theo

- Ưu tiên kinh doanh và dự toán kỹ thuật

2. Phát hành ngắn

- Nhận hệ thống làm việc nhanh chóng

- Phát hành phiên bản mới trong chu kỳ ngắn (2 đến 4 tuần)

- Cơ sở kế hoạch chi tiết mới về phản hồi của khách hàng

3. Ẩn dụ

- Sử dụng một phép ẩn dụ thay vì kiến ​​trúc

4. Thiết kế đơn giản

- Loại bỏ sự phức tạp không cần thiết càng sớm càng tốt

- Thiết kế có thể được thay đổi trong các phiên bản sau

5. Phát triển dựa trên thử nghiệm

- Liên tục và tự động

- Viết bài kiểm tra trước khi viết mã

6. Cải tiến thiết kế (Tái cấu trúc)

- Loại bỏ trùng lặp, cải thiện giao tiếp và đơn giản hóa hoặc thêm tính linh hoạt cần thiết

7. Lập trình cặp

- Hai lập trình viên làm việc cùng một máy

8. Sở hữu tập thể

- Bất cứ ai cũng có thể thay đổi bất kỳ mã nào bất cứ lúc nào

9. Tích hợp liên tục

- Tích hợp và kiểm tra mỗi khi hoàn thành một nhiệm vụ (nhiều lần trong ngày)

10. Tốc độ bền vững

- 40 giờ một tuần hợp lý

- Không bao giờ làm thêm hai tuần liên tiếp

11. Khách hàng tại chỗ

- Bao gồm khách hàng thực sự trong nhóm

12. Tiêu chuẩn mã hóa

- Mọi người cần sử dụng các quy tắc giống nhau