8: Extreme Programming

Origin and Values

XP is a set of practices – these weren't new, but the best practices at the time. They were then taken to the extreme for XP.

Originally there were 12 practices, and it has since been expanded to 24 – we'll only look at the 12.

Values

Extreme Programming is based on 5 values:

1. Communication
2. Simplicity
3. Feedback
4. Courage
5. Respect

(Don't need to remember these, but need to be able to understand/explain them.)

Communication

Different people have different knowledge and experience – communication is needed to exchange these.

Simplicity

Focus on the simplest solution that works – aim to eliminate wasted effort on solutions that are overly complex, and not needed right now.

Feedback

Requirements will change – the sooner you know about it, the better. Get feedback early and often.

4. Courage

Speak unpleasant truths to foster communication and trust.

Discard failing solutions – don't fall into the sunk cost fallacy.

5. Respect

All members of a team must case about each other and the project.

Each person's contributions and independence must be respected.

The Customer & the Developer

The customer decides:

- the scope – what the system must do
- the priority – what is most important
- composition of releases
-

The developers decide:

- estimated time to add a feature
- technical consequences – explains the consequences of technical choices (but the customer makes the decision)
- the process – how the team will work
- detailed schedule – within a given iteration

(This is similar to the product owner and dev team in Scrum).

Practices

1. Planning Game

Plan for the upcoming iteration, based on user stories provided by the customer.

The dev team reviews user stories:

- What is the time estimation / effort? [...]

2. Small Releases

Release a working and tested software product as early as possible.

The first release is the minimum viable product. The customer uses the product as soon as possible and gives feedback.

3. System Metaphor

A story that everyone (customers, programmers, managers) can tell about how the system works.

Example metaphors:

- Desktop for GUIs
- Shopping Cart in online shopping

4. Simple Design

The system design should be as simple as possible at any moment in time.

Always implement things when you actually need them – additional code (e.g. for things you may never need) is just pollution. You can't anticipate feature changes.

5. Test-first Development

Testing is done first, and done throughout the process.

Write unit tests first, then write the code to pass those tests. This ensures the tests are written, and improves the testability of the system.

It also focuses the developer on what needs to be done – requirements as tests.

All tests should be automated.

6. Refactoring

Improving the design of existing code.

Need this once changes have happened. Refactoring must make no changes to the software's behaviour – can use unit tests to check this.

Done to keep the code as simple as possible, or to make it better/more efficient.

This works very well with test-first development and simple design.

7. Pair Programming

Have 2 people work on production code – they take turns typing and watching.

The Driver codes, and the Navigator asks questions and reviews the code continuously.

This can catch errors very quickly, but can be exhausting. You can rotate pairs to mix things up.

8. Continuous Integration

New code is integrated into the main system no later than a few hours after it's written, triggering a build from scratch.

Unit tests already exist, thanks to test-first development. These must all pass, otherwise the changes are rolled back and fixed.

CI has become a key practice independent from XP.

9. Collective Ownership

All developers own all the code – everybody is responsible.

This facilitates refactoring, and helps mitigate the effect of team members leaving.

If anybody sees an opportunity to improve the code, they can/should go for it.

10. On-site Customer

The customer of the software sits with the team to answer questions.

This isn't always feasible, but can avoid working on assumptions, because you can always ask the customer.

11. 40-hour Workweeks

Work at a sustainable pace. Prevent regular overtime – it's not sustainable.

40's meant to be a guideline, not an exact number.

12. Open Workspace

Everybody works in an open workspace, usually with small private spaces around a central area.

The rationale is that this encourages regular conversations, and enables serendipitous communication – people can overhear useful things.

The room needs to be big enough, and lots of wall space should be allowed for white-boards for design discussions.

In Practice

Most teams don't adopt all of these practices – one study found that no team used more than 8 of the practices.