Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development,[1][2][3] it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, code simplicity and clarity, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.[2][3][4] The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed continuously, i.e. the practice of pair programming.

Practices[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=22)]

*Further information:*[*Extreme programming practices*](https://en.wikipedia.org/wiki/Extreme_programming_practices)

Extreme programming has been described as having 12 practices, grouped into four areas:

**Fine-scale feedback**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=23)]

* [Pair programming](https://en.wikipedia.org/wiki/Pair_programming)[[5]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-Cworld92-5)
* [Planning game](https://en.wikipedia.org/wiki/Planning_game)
* [Test-driven development](https://en.wikipedia.org/wiki/Test-driven_development)
* [Extreme programming practices](https://en.wikipedia.org/wiki/Extreme_programming_practices)

**Continuous process**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=24)]

* [Continuous integration](https://en.wikipedia.org/wiki/Continuous_integration)
* [Refactoring](https://en.wikipedia.org/wiki/Refactoring) or design improvement[[5]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-Cworld92-5)
* [Small releases](https://en.wikipedia.org/wiki/Extreme_programming_practices#Small_releases)

**Shared understanding**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=25)]

* [Coding standards](https://en.wikipedia.org/wiki/Extreme_programming_practices#Coding_standard)
* [Collective code ownership](https://en.wikipedia.org/wiki/Extreme_programming_practices#Collective_code_ownership)[[5]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-Cworld92-5)
* [Simple design](https://en.wikipedia.org/wiki/Extreme_programming_practices#Simple_design)[[5]](https://en.wikipedia.org/wiki/Extreme_programming#cite_note-Cworld92-5)
* [System metaphor](https://en.wikipedia.org/wiki/Extreme_programming_practices#System_metaphor)

**Programmer welfare**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=26)]

* [Sustainable pace](https://en.wikipedia.org/wiki/Extreme_programming_practices#Sustainable_pace)

**Coding**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=27)]

* The customer is always available
* Code the [unit test](https://en.wikipedia.org/wiki/Unit_test) first
* Only one pair integrates code at a time
* Leave [optimization](https://en.wikipedia.org/wiki/Program_optimization) until last
* No [overtime](https://en.wikipedia.org/wiki/Overtime)

**Testing**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming&action=edit&section=28)]

* All code must have [unit tests](https://en.wikipedia.org/wiki/Unit_tests)
* All code must pass all [unit tests](https://en.wikipedia.org/wiki/Unit_tests) before it can be released.
* When a [bug](https://en.wikipedia.org/wiki/Software_bug) is found, tests are created before the bug is addressed (a bug is not an error in logic, it is a test that was not written)
* [Acceptance tests](https://en.wikipedia.org/wiki/Acceptance_tests) are run often and the results are published

Fine scale feedback[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=1)]

**Pair programming**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=2)]

[Pair programming](https://en.wikipedia.org/wiki/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).

**Planning game**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=3)]

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](https://en.wikipedia.org/wiki/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]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-2) The Planning Game approach has also been adopted by non-software projects and teams in the context of [business agility](https://en.wikipedia.org/wiki/Business_agility).[[3]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-3)

***Release planning***[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=4)]

**Exploration phase**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=5)]

This is an iterative process of gathering requirements and estimating the work impact of each of those requirements.

* Write a Story: Business has come with a problem; during a meeting, development will try to define this problem and get requirements. Based on the business problem, a story ([user story](https://en.wikipedia.org/wiki/User_story)) has to be written. This is done by business, where they point out what they want a part of the system to do. It is important that development has no influence on this story. The story is written on a user story card.
* Estimate a Story: Development estimates how long it will take to implement the work implied by the story card. Development can also create spike solutions to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem. Again, this may not influence the business requirements.
* Split a Story: Every design critical complexity has to be addressed before starting the iteration planning. If development isn't able to estimate the story, it needs to be split up and written again.

When business cannot come up with any more requirements, one proceeds to the commitment phase.

**Commitment phase**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=6)]

This phase involves the determination of costs, benefits, and schedule impact. It has four components:

* Sort by Value: Business sorts the user stories by [Business Value](https://en.wikipedia.org/wiki/Business_Value).
* Sort by Risk: Development sorts the stories by risk.
* Set Velocity: Development determines at what speed they can perform the project.
* Choose scope: The user stories that will be finished in the next release will be picked. Based on the user stories the release date is determined.

**Sort by value**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=7)]

The business side sorts the user stories by business value. They will arrange them into three piles:

* Critical: stories without which the system cannot function or has no meaning.
* Significant [Business Value](https://en.wikipedia.org/wiki/Business_Value): Non-critical user stories that have significant business value.
* Nice to have: User stories that do not have significant business value.

**Sort by risk**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=8)]

The developers sort the user stories by risk. They also categorize into three piles: low, medium and high risk user stories. The following is an example of an approach to this:

* Determine Risk Index: Give each user story an index from 0 to 2 on each of the following factors:
  + Completeness (do we know all of the story details?)
    - Complete (0)
    - Incomplete (1)
    - Unknown (2)
  + Volatility (is it likely to change?)
    - low (0)
    - medium (1)
    - high (2)
  + Complexity (how hard is it to build?)
    - simple (0)
    - standard (1)
    - complex (2)

All indexes for a user story are added, assigning the user stories a risk index of low (0–1), medium (2–4), or high (5–6).

**Steering phase**[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=9)]

Within the steering phase the programmers and business people can "steer" the process. That is to say, they can make changes. Individual user stories, or relative priorities of different user stories, might change; estimates might prove wrong. This is the chance to adjust the plan accordingly.

***Iteration planning***[[edit](https://en.wikipedia.org/w/index.php?title=Extreme_programming_practices&action=edit&section=10)]

Considering team velocity storypoints to be planned. Iteration duration can be 1 to 3 weeks.

**Exploration phase**

The exploration phase of the iteration planning is about creating tasks and estimating their implementation time.

* Translate the requirement to tasks: Place on task cards.
* Combine/Split task: If the programmer cannot estimate the task because it is too small or too big, the programmer will need to combine or split the task.
* Estimate task: Estimate the time it will take to implement the task.

**Commitment phase**

Within the commitment phase of the iteration planning programmers are assigned tasks that reference the different user stories.

* A programmer accepts a task: Each programmer picks a task for which he or she takes responsibility.
* Programmer estimates the task: Because the programmer is now responsible for the task, he or she should give the eventual estimation of the task.
* Set load factor: The load factor represents the ideal amount of hands-on development time per programmer within one iteration. For example, in a 40-hour week, with 5 hours dedicated to meetings, this would be no more than 35 hours.
* Balancing: When all programmers within the team have been assigned tasks, a comparison is made between the estimated time of the tasks and the load factor. Then the tasks are balanced out among the programmers. If a programmer is overcommitted, other programmers must take over some of his or her tasks and vice versa.

**Steering phase**

The implementation of the tasks is done during the steering phase of the iteration.

* Get a task card: The programmer gets the task card for one of the tasks to which he or she has committed.
* Find a Partner: The programmer will implement this task along with another programmer. This is further discussed in the practice [Pair Programming](https://en.wikipedia.org/wiki/Pair_Programming).
* Design the task: If needed, the programmers will design the functionality of the task.
* Implement the task using [Test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) (TDD) (see below)
* Run Functional test: Functional tests (based on the requirements in the associated user story and task card) are run.

**Test driven development**

[Unit tests](https://en.wikipedia.org/wiki/Unit_testing) 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Code_refactoring): Remove any [code smells](https://en.wikipedia.org/wiki/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]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-4).

**Whole team**

Within XP, the "customer" is not the one who pays the bill, but the one who really uses the system. XP says that the customer should be on hand at all times and available for questions. For instance, the team developing a financial administration system should include a financial administrator.

Continuous process

**Continuous integration**

The development team should always be working on the latest version of the software. Since different team members may have versions saved locally with various changes and improvements, they should try to upload their current version to the code repository every few hours, or when a significant break presents itself. [Continuous integration](https://en.wikipedia.org/wiki/Continuous_integration) will avoid delays later on in the project cycle, caused by integration problems.

**Design improvement**

Because XP doctrine advocates programming only what is needed today, and implementing it as simply as possible, at times this may result in a system that is stuck. One of the symptoms of this is the need for dual (or multiple) maintenance: functional changes start requiring changes to multiple copies of the same (or similar) code. Another symptom is that changes in one part of the code affect lots of other parts. XP doctrine says that when this occurs, the system is telling you to [refactor](https://en.wikipedia.org/wiki/Refactoring) your code by changing the architecture, making it simpler and more generic.

**Small releases**

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.

Shared understanding

**Coding standard**

[Coding standard](https://en.wikipedia.org/wiki/Coding_conventions) 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]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-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](https://en.wikipedia.org/wiki/Self-documenting) to the furthest degree possible. This reduces the need for [code comments](https://en.wikipedia.org/wiki/Comment_(computer_programming)), which can get out of sync with the code itself.[[6]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-6)

**Collective code ownership**

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]](https://en.wikipedia.org/wiki/Extreme_programming_practices#cite_note-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.

**Simple design**

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.

**System metaphor**

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.

Programmer welfare

**Sustainable pace**

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.