**Agile sw development**

**Agile software development** is an approach to [software development](https://en.wikipedia.org/wiki/Software_development) under which requirements and solutions evolve through the collaborative effort of [self-organizing](https://en.wikipedia.org/wiki/Self-organizing_communities) and [cross-functional](https://en.wikipedia.org/wiki/Cross-functional_team) teams and their [customer(s)](https://en.wikipedia.org/wiki/Customer)/[end user(s)](https://en.wikipedia.org/wiki/End_user). It advocates adaptive planning, evolutionary development, early delivery, and [continual improvement](https://en.wikipedia.org/wiki/Continual_improvement_process), and it encourages rapid and flexible response to change.

The term *agile* (sometimes written *Agile*) was popularized, in this context, by the [*Manifesto for Agile Software Development*](https://en.wikipedia.org/wiki/Agile_software_development#The_Agile_Manifesto). The values and principles espoused in this manifesto were derived from and underpin a broad range of [software development frameworks](https://en.wikipedia.org/wiki/Software_development_process), including [Scrum](https://en.wikipedia.org/wiki/Scrum_(software_development)) and [Kanban](https://en.wikipedia.org/wiki/Kanban_(development)).

There is significant anecdotal evidence that adopting agile practices and values improves the agility of software professionals, teams and organizations; however, some empirical studies have found no scientific evidence.

**The Manifesto of Agile sw development**

**Agile software development values**

Based on their combined experience of developing software and helping others do that, the seventeen signatories to the manifesto proclaimed that they value:

* ***Individuals and Interactions*** *over processes and tools*
* ***Working Software*** *over comprehensive documentation*
* ***Customer Collaboration*** *over contract negotiation*
* ***Responding to Change*** *over following a plan*

That is to say, the items on the left are valued more than the items on the right.

As [Scott Ambler](https://en.wikipedia.org/wiki/Scott_Ambler) elucidated:

* Tools and processes are important, but it is more important to have competent people working together effectively.
* Good documentation is useful in helping people to understand how the software is built and how to use it, but the main point of development is to create software, not documentation.
* A contract is important but is no substitute for working closely with customers to discover what they need.
* A project plan is important, but it must not be too rigid to accommodate changes in technology or the environment, stakeholders' priorities, and people's understanding of the problem and its solution.

The Agile movement is not anti-methodology, in fact many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment.

**Overview**

### Iterative, incremental and evolutionary

Most agile development methods break product development work into small increments that minimize the amount of up-front planning and design. Iterations, or sprints, are short time frames ([timeboxes](https://en.wikipedia.org/wiki/Timeboxing)) that typically last from one to four weeks. Each iteration involves a [cross-functional team](https://en.wikipedia.org/wiki/Cross-functional_team) working in all functions: [planning](https://en.wikipedia.org/wiki/Project_planning), [analysis](https://en.wikipedia.org/wiki/Requirements_analysis), [design](https://en.wikipedia.org/wiki/Software_design), [coding](https://en.wikipedia.org/wiki/Computer_programming), [unit testing](https://en.wikipedia.org/wiki/Unit_testing), and [acceptance testing](https://en.wikipedia.org/wiki/Acceptance_testing). At the end of the iteration a working product is demonstrated to stakeholders. This minimizes overall risk and allows the product to adapt to changes quickly. An iteration might not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal [bugs](https://en.wikipedia.org/wiki/Software_bug)) at the end of each iteration. Multiple iterations might be required to release a product or new features. Working software is the primary measure of progress.

### Efficient and face-to-face communication

The principle of [co-location](https://en.wikipedia.org/wiki/Colocation_(business)) is that co-workers on the same team should be situated together to better establish the identity as a team and to improve communication. This enables [face-to-face interaction](https://en.wikipedia.org/wiki/Face-to-face_interaction), ideally in front of a whiteboard, that reduces the cycle time typically taken when questions and answers are mediated through phone, persistent chat, wiki, or email.

No matter which development method is followed, every team should include a [customer representative](https://en.wikipedia.org/wiki/Customer_representative) ("Product Owner" in [Scrum](https://en.wikipedia.org/wiki/Scrum_(software_development))). This person is agreed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer questions throughout the iteration. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the [return on investment](https://en.wikipedia.org/wiki/Rate_of_return) (ROI) and ensuring alignment with customer needs and company goals.

In agile software development, an **information radiator** is a (normally large) physical display located prominently near the development team, where passers-by can see it. It presents an up-to-date summary of the product development status. A [build light indicator](https://en.wikipedia.org/wiki/Build_light_indicator) may also be used to inform a team about the current status of their product development.

### Very short feedback loop and adaptation cycle

A common characteristic in agile software development is the [daily stand-up](https://en.wikipedia.org/wiki/Stand-up_meeting) (also known as the *daily scrum)*. In a brief session, team members report to each other what they did the previous day toward their team's iteration goal, what they intend to do today toward the goal, and any roadblocks or impediments they can see to the goal.

### Quality focus

Specific tools and techniques, such as [continuous integration](https://en.wikipedia.org/wiki/Continuous_integration), automated [unit testing](https://en.wikipedia.org/wiki/Unit_testing), [pair programming](https://en.wikipedia.org/wiki/Pair_programming), [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development), [design patterns](https://en.wikipedia.org/wiki/Software_design_pattern), [behavior-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development), [domain-driven design](https://en.wikipedia.org/wiki/Domain-driven_design), [code refactoring](https://en.wikipedia.org/wiki/Code_refactoring) and other techniques are often used to improve quality and enhance product development agility. This is predicated on designing and building quality in from the beginning and being able to demonstrate software for customers at any point, or at least at the end of every iteration.

**Philosophy**

Compared to traditional software engineering, agile software development mainly targets complex systems and product development with dynamic, non-deterministic and non-linear characteristics. Accurate estimates, stable plans, and predictions are often hard to get in early stages, and confidence in them is likely to be low. Agile practitioners will seek to reduce the *leap-of-faith* that is needed before any evidence of value can be obtained. Requirements and design are held to be emergent. Big up-front specifications would probably cause a lot of waste in such cases, i.e., are not economically sound. These basic arguments and previous industry experiences, learned from years of successes and failures, have helped shape agile development's favor of adaptive, iterative and evolutionary development.

### Adaptive vs. predictive

Development methods exist on a continuum from *adaptive* to *predictive*. Agile software development methods lie on the *adaptive* side of this continuum. One key of adaptive development methods is a [*rolling wave*](https://en.wikipedia.org/wiki/Rolling_Wave_planning) approach to schedule planning, which identifies milestones but leaves flexibility in the path to reach them, and also allows for the milestones themselves to change.

*Adaptive* methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team has difficulty describing exactly what will happen in the future. The further away a date is, the more vague an adaptive method is about what will happen on that date. An adaptive team cannot report exactly what tasks they will do next week, but only which features they plan for next month. When asked about a release six months from now, an adaptive team might be able to report only the mission statement for the release, or a statement of expected value vs. cost.

*Predictive* methods, in contrast, focus on analysing and planning the future in detail and cater for known risks. In the extremes, a predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive methods rely on effective early phase analysis and if this goes very wrong, the project may have difficulty changing direction. Predictive teams often institute a [change control board](https://en.wikipedia.org/wiki/Change_control_board) to ensure they consider only the most valuable changes.

### Agile vs. waterfall

One of the differences between agile software development methods and waterfall is the approach to quality and testing. In the [waterfall model](https://en.wikipedia.org/wiki/Waterfall_model), there is always a separate *testing phase* after a *build phase*; however, in agile software development testing is completed in the same iteration as programming.

Another difference is that traditional "waterfall" software development moves a project through various Software Development Lifecycle (SDLC) phases. One phase is completed in its entirety before moving on to the next phase.

Because testing is done in every iteration—which develops a small piece of the software—users can frequently use those new pieces of software and validate the value. After the users know the real value of the updated piece of software, they can make better decisions about the software's future. Having a value retrospective and software re-planning session in each iteration—[Scrum](https://en.wikipedia.org/wiki/Scrum_(software_development)) typically has iterations of just two weeks—helps the team continuously adapt its plans so as to maximize the value it delivers. This follows a pattern similar to the [PDCA](https://en.wikipedia.org/wiki/PDCA) cycle, as the work is *planned*, *done*, *checked* (in the review and retrospective), and any changes agreed are *acted* upon.

This iterative approach supports a *product* rather than a *project* mindset. This provides greater flexibility throughout the development process; whereas on projects the requirements are defined and locked down from the very beginning, making it difficult to change them later. Iterative product development allows the software to evolve in response to changes in business environment or market requirements.

Because of the short iteration style of agile software development, it also has strong connections with the [lean startup](https://en.wikipedia.org/wiki/Lean_startup) concept.

### Code vs. documentation

In a letter to [*IEEE Computer*](https://en.wikipedia.org/wiki/Computer_(magazine)), Steven Rakitin expressed cynicism about agile software development, calling it "yet another attempt to undermine the discipline of software engineering" and translating "working software over comprehensive documentation" as "we want to spend all our time coding. Remember, real programmers don't write documentation."

This is disputed by proponents of agile software development, who state that developers should write documentation if that is the best way to achieve the relevant goals, but that there are often better ways to achieve those goals than writing static documentation. [Scott Ambler](https://en.wikipedia.org/wiki/Scott_Ambler) states that documentation should be "just barely good enough" (JBGE), that too much or comprehensive documentation would usually cause waste, and developers rarely trust detailed documentation because it's usually out of sync with code, while too little documentation may also cause problems for maintenance, communication, learning and knowledge sharing. [Alistair Cockburn](https://en.wikipedia.org/wiki/Alistair_Cockburn) wrote of the *Crystal Clear* method:

Crystal considers development a series of co-operative games, and intends that the documentation is enough to help the next win at the next game. The work products for Crystal include use cases, risk list, iteration plan, core domain models, and design notes to inform on choices...however there are no templates for these documents and descriptions are necessarily vague, but the objective is clear, **just enough documentation** for the next game. I always tend to characterize this to my team as: what would you want to know if you joined the team tomorrow.

— Alistair Cockburn

**XP**

**Extreme programming** (**XP**) is a [software development methodology](https://en.wikipedia.org/wiki/Software_development_methodology) which is intended to improve software quality and responsiveness to changing customer requirements. As a type of [agile software development](https://en.wikipedia.org/wiki/Agile_software_development), 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](https://en.wikipedia.org/wiki/Pair_programming) or doing extensive [code review](https://en.wikipedia.org/wiki/Code_review), [unit testing](https://en.wikipedia.org/wiki/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. 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](https://en.wikipedia.org/wiki/Code_review) are considered a beneficial practice; taken to the extreme, code can be reviewed *continuously*, i.e. the practice of [pair programming](https://en.wikipedia.org/wiki/Pair_programming).

**History**

Extreme programming was created by [Kent Beck](https://en.wikipedia.org/wiki/Kent_Beck) during his work on the [Chrysler Comprehensive Compensation System](https://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System) (C3) payroll project. Beck became the C3 [project leader](https://en.wikipedia.org/wiki/Project_management) in March 1996 and began to refine the development methodology used in the project and wrote a book on the methodology (in October 1999, *Extreme Programming Explained* was published). [Chrysler](https://en.wikipedia.org/wiki/Chrysler) cancelled the C3 project in February 2000, after seven years, when the company was acquired by [Daimler-Benz](https://en.wikipedia.org/wiki/Daimler-Benz).

Many extreme programming practices have been around for some time; the methodology takes "[best practices](https://en.wikipedia.org/wiki/Best_practices)" to extreme levels. For example, the "practice of test-first development, planning and writing tests before each micro-increment" was used as early as NASA's [Project Mercury](https://en.wikipedia.org/wiki/Project_Mercury), in the early 1960s. To shorten the total development time, some formal test documents (such as for [acceptance testing](https://en.wikipedia.org/wiki/Acceptance_testing)) have been developed in parallel (or shortly before) the software is ready for testing. A NASA independent test group can write the test procedures, based on formal requirements and logical limits, before the software has been written and integrated with the hardware. In XP, this concept is taken to the extreme level by writing automated tests (perhaps inside of software modules) which validate the operation of even small sections of software coding, rather than only testing the larger features.

### Origins

Software development in the 1990s was shaped by two major influences: internally, [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) replaced [procedural programming](https://en.wikipedia.org/wiki/Procedural_programming) as the programming paradigm favored by some in the industry; externally, the rise of the Internet and the [dot-com boom](https://en.wikipedia.org/wiki/Dot-com_boom) emphasized speed-to-market and company growth as competitive business factors. Rapidly changing requirements demanded shorter [product life-cycles](https://en.wikipedia.org/wiki/Product_life_cycle_management), and were often incompatible with traditional methods of software development.

The [Chrysler Comprehensive Compensation System](https://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System) (C3) was started in order to determine the best way to use object technologies, using the payroll systems at Chrysler as the object of research, with [Smalltalk](https://en.wikipedia.org/wiki/Smalltalk) as the language and [GemStone](https://en.wikipedia.org/wiki/Gemstone_Database_Management_System) as the [data access layer](https://en.wikipedia.org/wiki/Data_access_layer). They brought in [Kent Beck](https://en.wikipedia.org/wiki/Kent_Beck), a prominent Smalltalk practitioner, to do [performance tuning](https://en.wikipedia.org/wiki/Performance_tuning) on the system, but his role expanded as he noted several problems they were having with their development process. He took this opportunity to propose and implement some changes in their practices based on his work with his frequent collaborator, [Ward Cunningham](https://en.wikipedia.org/wiki/Ward_Cunningham). Beck describes the early conception of the methods:

The first time I was asked to lead a team, I asked them to do a little bit of the things I thought were sensible, like testing and reviews. The second time there was a lot more on the line. I thought, "Damn the torpedoes, at least this will make a good article," [and] asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else.

Beck invited [Ron Jeffries](https://en.wikipedia.org/wiki/Ron_Jeffries) to the project to help develop and refine these methods. Jeffries thereafter acted as a coach to instill the practices as habits in the C3 team.

Information about the principles and practices behind XP was disseminated to the wider world through discussions on the original [wiki](https://en.wikipedia.org/wiki/Wiki), Cunningham's [WikiWikiWeb](https://en.wikipedia.org/wiki/WikiWikiWeb). Various contributors discussed and expanded upon the ideas, and some spin-off methodologies resulted (see [agile software development](https://en.wikipedia.org/wiki/Agile_software_development)). Also, XP concepts have been explained, for several years, using a [hypertext](https://en.wikipedia.org/wiki/Hypertext) system map on the XP website at [http://www.extremeprogramming.org](http://www.extremeprogramming.org/) circa 1999.

Beck edited a series of books on XP, beginning with his own *Extreme Programming Explained* (1999, [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-201-61641-6](https://en.wikipedia.org/wiki/Special:BookSources/0-201-61641-6)), spreading his ideas to a much larger audience. Authors in the series went through various aspects attending XP and its practices. The series included a book that was critical of the practices.

### Current state

XP generated significant interest among software communities in the late 1990s and early 2000s, seeing adoption in a number of environments radically different from its origins.

The high discipline required by the original practices often went by the wayside, causing some of these practices, such as those thought too rigid, to be deprecated or reduced, or even left unfinished, on individual sites. For example, the practice of end-of-day [integration tests](https://en.wikipedia.org/wiki/Integration_test) for a particular project could be changed to an end-of-week schedule, or simply reduced to mutually agreed dates. Such a more relaxed schedule could avoid people feeling rushed to generate artificial stubs just to pass the end-of-day testing. A less-rigid schedule allows, instead, for some complex features to be more fully developed over a several-day period.

Meanwhile, other agile development practices have not stood still, and XP is still evolving, assimilating more lessons from experiences in the field, to use other practices. In the second edition of *Extreme Programming Explained* (November 2004), five years after the first edition, Beck added more values and practices and differentiated between primary and corollary practices.

The Theory of Sustainable Software Development explains why extreme programming teams can thrive in spite of team disruptions.