Agile is the label given to a growing number of methodologies which are based on the premise that if you hire competent developers, presumably they know how to write code.

Any problems your developers encounter, therefore, aren’t coding issues but organizational and communications ones, and those are what the agile approaches attempt to address.
## Definition

- **Agile approaches 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, while we believe following a plan as well as responding to change, we value flexible process allowing change at any time even more.”

## Core Principles - 1

- **Assume Simplicity.**
  - As you develop you should assume that the simplest solution is the best solution.
  - Don't overbuild your software, or in the case of modeling don't depict additional features in your models that you don't need today.
  - Have the courage that you don't need to over-model your system today, that you can model based on your existing requirements today and refactor your system in the future when your requirements evolve.
  - Keep your models as simple as possible.
Core Principles - 2

Embrace Change.

- Requirements evolve over time. People's understanding of the requirements change over time.
- Project stakeholders can change as your project moves forward, new people are added and existing ones can leave.
- Project stakeholders can change their viewpoints as well, potentially changing the goals and success criteria for your effort.
- The implication is that your project's environment changes as your efforts progress, and that as a result your approach to development must reflect this reality.

Core Principles - 3

Enabling The Next Effort Is Your Secondary Goal.

- Your project can still be considered a failure even when your team delivers a working system to your users; part of fulfilling the needs of your project stakeholders is to ensure that your system is robust enough so that it can be extended over time.
- When you are playing the software development game your secondary goal is to setup to play the next game.
- Your next effort may be the development of the next major release of your system or it may simply be the operations and support of the current version you are building.
Core Principles – 3 cont.

**Enabling The Next Effort Is Your Secondary Goal.**

- To enable it you will not only want to develop quality software but also create just enough documentation and supporting materials so that the people playing the next game can be effective.
- Factors that you need to consider include whether members of your existing team will be involved with the next effort, the nature of the next effort itself, and the importance of the next effort to your organization.
- In short, when you are working on your system you need to keep an eye on the future.

Core Principles - 4

**Incremental Change.**

- An important concept to understand with respect to modeling is that you don't need to get it right the first time, in fact, it is very unlikely that you could do so even if you tried.
- Furthermore, you do not need to capture every single detail in your models, you just need to get it good enough at the time.
- Instead of futilely trying to develop an all encompassing model at the start, you instead can put a stake in the ground by developing a small model, or perhaps a high-level model, and evolve it over time (or simply discard it when you no longer need it) in an incremental manner.
Core Principles – 4 cont

Maximize Stakeholder Investment.

- Your project stakeholders are investing resources -- time, money, facilities, and so on -- to have software developed that meets their needs.
- Stakeholders deserve to invest their resources the best way possible and not to have resources frittered away by your team.
- Furthermore, they deserve to have the final say in how those resources are invested or not invested.
- If it was your resources, would you want it any other way?

Core Principles – 4 cont.1

Model With A Purpose.

- Many developers worry about whether their artifacts -- such as models, source code, or documents -- are detailed enough or if they are too detailed, or similarly if they are sufficiently accurate.
- What they're not doing is stepping back and asking why they're creating the artifact in the first place and who they are creating it for.
- With respect to modeling, perhaps you need to understand an aspect of your software better, perhaps you need to communicate your approach to senior management to justify your project, or perhaps you need to create documentation that describes your system to the people who will be operating and/or maintaining/evolving it over time.
- If you cannot identify why and for whom you are creating a model then why are you bothering to work on it all?
Core Principles – 4 cont.2

Model With A Purpose. Continued:

• If you cannot identify why and for whom you are creating a model then why are you bothering to work on it all?
• Your first step is to identify a valid purpose for creating a model and the audience for that model, then based on that purpose and audience develop it to the point where it is both sufficiently accurate and sufficiently detailed.
• Once a model has fulfilled its goals you're finished with it for now and should move on to something else, such as writing some code to show that the model works.
• This principle also applies to a change to an existing model: if you are making a change, perhaps applying a known pattern, then you should have a valid reason to make that change (perhaps to support a new requirement or to refactor your work to something cleaner).
• An important implication of this principle is that you need to know your audience, even when that audience is yourself.

Core Principles – 5

Multiple Models.

• You potentially need to use multiple models to develop software because each model describes a single aspect of your software.
• "What models are potentially required to build modern-day business applications?" Considering the complexity of modern day software, you need to have a wide range of techniques in your intellectual modeling toolkit to be effective.
• An important point is that you don't need to develop all of these models for any given system, but that depending on the exact nature of the software you are developing you will require at least a subset of the models.
• Different systems, different subsets. Just like every fixit job at home doesn't require you to use every tool available to you in your toolbox, over time the variety of jobs you perform will require you to use each tool at some point.
• Just like you use some tools more than others, you will use some types of models more than others.
Core Principles – 6

**Quality Work.**

- Nobody likes sloppy work. The people doing the work don't like it because it's something they can't be proud of, the people coming along later to refactor the work (for whatever reason) don't like it because it's harder to understand and to update, and the end users won't like the work because it's likely fragile and/or doesn't meet their expectations.

**Rapid Feedback.**

- The time between an action and the feedback on that action is critical.
- By working with other people on a model, particularly when you are working with a shared modeling technology (such as a whiteboard, CRC cards, or essential modeling materials such as sticky notes) you are obtaining near-instant feedback on your ideas. Working closely with your customer, to understand the requirements, to analyze those requirements, or to develop a user interface that meets their needs, provides opportunities for rapid feedback.

---

Core Principles – 6

**Software Is Your Primary Goal ("cutting code").**

- The goal of software development is to produce software that meets the needs of your project stakeholders in an effective manner.
- The primary goal is **not** to produce extraneous documentation, extraneous management artifacts, or even models.
- Any activity that does not directly contribute to this goal should be questioned and avoided if it cannot be justified in this light.

**Travel Light.**

- Every artifact that you create, and then decide to keep, will need to be maintained over time.
- If you decide to keep seven models, then whenever a change occurs (a new/updated requirement, a new approach is taken by your team, a new technology is adopted, ...) you will need to consider the impact of that change on all seven models and then act accordingly.
Core Principles – 6

Travel Light. cont

- If you decide to keep only three models then you clearly have less work to perform to support the same change, making you more agile because you are traveling lighter.
- Similarly, the more complex/detailed your models are, the more likely it is that any given change will be harder to accomplish (the individual model is "heavier" and is therefore more of a burden to maintain).
- Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner.
- Never underestimate the seriousness of this trade-off.

Core Principles – 6

Travel Light. cont

- Someone trekking across the desert will benefit from a map, a hat, good boots, and a canteen of water they likely won't make it if they burden themselves with hundreds of gallons of water, a pack full of every piece of survival gear imaginable, and a collection of books about the desert.
- Similarly, a development team that decides to develop and maintain a detailed requirements document, a detailed collection of analysis models, a detailed collection of architectural models, and a detailed collection of design models will quickly discover they are spending the majority of their time updating documents instead of writing source code.
Core Principles – 6

Travel Light. cont

- However, someone building a highway across the desert will benefit not only from a map, a hat, good boots, and a canteen of water but from a substantial set of tools and provisions including a solid plan.

Agile Technologies (Modeling, Programming…)

Although the various agile approaches are different, they have some things in common: they are intended to produce software that can be changed quickly, and all specify short iterations and maximize the amount of time spent face to face. They also focus on team morale.

The agile approaches include extreme programming (XP) as all of them are lightweight methodologies. Lightweight methodologies dispense with much of the software development process overhead (“ceremony”) that bogs down developers, such as lengthy requirements definitions and extensive documentation.

XP, however, differs from agile approaches by being much more prescriptive, even dogmatic, some might say. XP revolves around 12 practices identified by Kent Beck "Extreme Programming Explained: Embrace Change" (Addison Wesley Longman Inc., 1999).
Extreme Programming (XP)

The three things to think about before making a decision about adopting an agile programming methodology:

» Beware the engineering metaphor. Building software is nothing like building bridges. When you're spanning a river, design represents perhaps 15% of the cost; construction accounts for the rest. When you're building a software application, in contrast, all the effort is in design; the actual coding (i.e., construction) is so cheap as to be essentially free.

» Creative processes, including application design, aren't easily planned, and predictability may be an impossible target. Developers and customers thus need to be ready to change their methods and goals as a project continues.

» Agile programming is people-centered, not process-oriented. If you treat your developers as fungible, interchangeable programming units instead of the creative and talented individuals they are, the good people will leave.

XP

There are four dimensions along which one can improve any software project.

• You need to improve communication.
• You need to seek simplicity.
• You need to get feedback on how well you are doing.
• And you need to always proceed with courage.

Communication, Simplicity, Feedback, and Courage are the four values sought out by XP programmers.
Extreme Programming - XP

Extreme Programming (XP) is one of a growing group of agile software development methodologies.

XP uses integrated teams of programmers, customers, and managers to develop high-quality software at high speed.

In particular, XP builds on treating people as the first-order influence on software development.

Working software is the primary measure of progress.

Extreme Programming is viable because of advances in our understanding of the problems we are trying to solve, and because the tools available let us change the previously exponential cost-of-change curve.

XP keeps the cost of change low, so that it is not much more expensive to implement a feature later than it is to implement it early, and then leverages this cost-of-change environment to produce software faster.
1. **Communication.** An XP team thrives on shared understanding of the problem and the software, and the most efficient and effective method of achieving shared understanding is **face-to-face communication**. Anything that obstructs efficient communication needs to be removed.

2. **Simplicity.** Simplicity is the art of maximizing the amount of work not done. "Simple, clear purpose and principles give rise to complex, intelligent behavior. Complex rules and regulations give rise to simple, stupid behavior."

3. **Feedback.** Often project teams and their customers don’t realize they’re in trouble until a short time before delivery. XP teams get frequent feedback - week to week by delivering working software, but also minute to minute through testing tools and any other mechanism they can implement.

4. **Courage.** Successful software teams need to operate on the edge of chaos - they need to go as fast as they possibly can without losing control. This means that sometimes they fail. If people are scared to fail then they’ll go too slowly.
XP promotes twelve practices. If you examine any of these practices in isolation you’ll see flaws, but one of the strengths of XP is that the practices combine in a mutually supporting way, each practice covering flaws in the others. Each of the XP practices is simple to describe, but requires skill to master. Together the practices lead to complex, emergent behavior. Each practice has a place in keeping the cost of change low, and leveraging the low cost of change. The XP practices are:

- **Coding Standards**
- **Simple Design**
- **Testing**
- **Pair Programming**
- **Collective Code Ownership**
- **Refactoring**
- **Continuous Integration**
- **Small Releases**
- **On-site Customer**
- **The Planning Game**
- **40-hour week**
- **Metaphor**

1. **Coding Standards.** Coding is a team activity. Over time different people will work on different pieces of the code, and disparities in coding style and conventions make the code harder to work with. To be effective we need it to look like all the code on the team was written by the same person, and we need coding standards to accomplish this.

2. **Simple Design.** Keeping the cost of change low means keeping the system as simple as possible. It also means not spending time and effort implementing features that may or may not be needed later. XP projects do the simplest thing possible, confident that it can be changed later at little added cost.

3. **Testing.** Every requirement is reflected in an acceptance test. Acceptance tests are owned by the customer. Programmers write test cases before they write code, using the test to focus on what they are trying to achieve, and to specify interfaces (test-first-design). All test cases are automated and run regularly. Programmers can make changes aggressively because the tests will catch mistakes.
4. **Pair Programming.** All production development is done by two people sharing one machine. Contrary to your intuition, this is much more efficient than having two people program separately. Pairs rotate frequently, so knowledge and experience ripples through the team. Code is reviewed continuously.

5. **Collective Code Ownership.** Anyone on the team has the authority to change any of the code, provided they do it with a partner, comply with coding standards, and ensure that all the tests work when they’ve finished. This removes the bottlenecks and architectural distortions that can occur with individual code ownership.

6. **Refactoring.** Refactoring is the technique of improving the design of existing code without changing functionality. Refactoring is feasible because an XP team has automated tests to catch mistakes. Refactoring lets us change code to reflect our continually improving understanding of the problem, and to extend our simple designs over time.

---

7. **Continuous Integration.** XP teams work in small steps, and integrate their code several times a day. This means that integration problems are discovered soon after they are created, and it’s fairly easy to rectify them. Continuous integration avoids lengthy but incompatible developments, and helps ensure that everyone is working on the latest version of the system all the time.

8. **Small Releases.** Releases should be as small as possible while delivering enough business value to make them worthwhile (even if the value is just measured in confidence). XP teams can perform releases in cycles of a few weeks because they are working in small steps, they have tests to catch regression, and they are integrating continuously. Some XP projects perform a release every day.
9. **On-site Customer.** No written requirement is complete and unambiguous. Programmers always need access to a customer for clarification, no matter how much effort went into the original specification. An XP team keeps it simple by skipping a lot of the effort put into the specification, and having a customer available to the programmers all of the time. XP programmers don’t guess the details of a feature, they ask the customer instead.

10. **The Planning Game.** XP codifies the regular negotiations over functionality that occur in a software project and turn them into a game. The Planning Game is run every iteration (every few weeks) to determine what functionality will go into the next (small) release. Programmers make technical decisions (estimates) and customers make business decisions (selecting the functionality).

11. **40-hour Week.** Software development is a creative exercise, and no one can be creative if they’re exhausted. Restricting the number of hours in a work week keeps people fresh, reduce staff turnover, and improves the quality of the finished product. Other XP practices help the team to deliver a competitive quantity of software (however you measure it) in a regular week.

12. **Metaphor.** The system metaphor gives the team a consistent vocabulary for discussing their problems and their solutions. It could be "this system is like an assembly line", or it could mean talking about the system in terms of business objects.
Extreme and Agile Programming

Although XP and Agile Approaches in various forms have been around for several years, corporate IT is only just beginning to consider it.

The chief obstacle is that some practices prescribed by XP contradict long-established IT policy. For example, XP specifies pair programming, in which two programmers sit side by side coding at a single workstation.

Pair programming seems blatantly inefficient, but a series of studies has confirmed that the approach results in fewer code defects, which ultimately speeds final delivery.

"Pair programming is not as productive initially, but the design happens on the fly and the quality is outstanding. And since we have fewer defects, we're not spending time doing as much bug fixing. So the cost issue is moot."

Extreme and Agile Programming

XP also requires the customer for whom the software is being written to take an active, ongoing role in the development process - to the extent that the customer is asked to write a test to prove a requested function before it's actually coded.

In XP, customers write desired functions on index cards (one function per card).

The developers estimate how long it will take to code that function.

Based on the estimates, the customer decides which functions to tackle first.
Extreme and Agile Programming

Then the customer writes a test, and the developers write code to pass the test. This requires a serious commitment on the part of the customer, but in return, the customer gets exactly what she asked for.

XP's approach to requirements alone saves tremendous effort. Quote: "We're talking about a bunch of index cards vs. 100-plus-page requirements documents".

XP isn't perfect, however, and some customers have looked at other agile approaches. "XP doesn't address deployment," says one manager. So now he is looking for a method, "which has some life-cycle project management".

Extreme Programming - Experiences

- Extreme Programming (XP) is actually a deliberate and disciplined approach to software development. About six years old, it has already been proven at cost conscious companies like Bayerische Landesbank, Credit Swiss Life, DaimlerChrysler, First Union National Bank, Ford Motor Company and UBS.

- XP is successful because it stresses customer satisfaction. The methodology is designed to deliver the software your customer needs when it is needed. XP empowers your developers to confidently respond to changing customer requirements, even late in the life cycle.

- This methodology also emphasizes team work. Managers, customers, and developers are all part of a team dedicated to delivering quality software. XP implements a simple, yet effective way to enable groupware style development.
Observations

- XP implements a simple, yet effective way to enable groupware style development.
- XP improves a software project in four essential ways: communication, simplicity, feedback, and courage.
- XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology.
- XP is different. It is a lot like a jigsaw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This is a significant departure from traditional software development methods.

Cost Aspects

- Software which is engineered to be simple and elegant is no more valuable than software that is complex and hard to maintain. Can this really be true? Extreme Programming (XP) is based on the idea that this is not in fact true.
- A typical project will spend about twenty times as much on people than on hardware. That means a project spending 2 million dollars on programmers per year will spend about 100 thousand dollars on computer equipment each year.
- Let's say that we are smart programmers and we find a way to save 20% of the hardware costs by some very clever programming tricks. It will make the source code harder to understand and maintain, but we are saving 20% or 20 thousand dollars per year, a big savings. Now what if instead we wrote our programs such that they were easy to understand and extend. We could expect to save no less than 10% of our people costs. That would come to 200 thousand dollars, a much bigger savings.
Issues

- Another important issue to customers are bugs. XP emphasizes not just testing, but testing well.
- Tests are automated and provide a safety net for programmers and customers alike.
- Tests are created before the code is written, while the code is written, and after the code is written.
- As bugs are found new tests are added. A safety net of tight mesh is created.
- Bugs don’t get through twice, and this is something the customers will notice.

Issues

- Another aspect the customers will notice is the attitude XP programmers have towards changing requirements.
- XP enables us to embrace change.
- Too often a customer will see a real opportunity for making a system useful after it has been delivered.
- XP short cuts this by getting customer feedback early while there is still time to change functionality or improve user acceptance.
Issues

- Much of what went into XP was a re-evaluation of the way software was created.
- The quality of the source code is much more important than one might realize.
- Just because our customers can't see our source code doesn't mean we shouldn't put the effort into creating something we can be proud of.