

What Is Agile Development?

From the SPC website, <https://www.software.org/MembersOnly/agile/default.asp?PageId=0>

Agile development is an approach to software engineering and project management that addresses current challenges in software development. It characterizes general principles and techniques that encompass specific, name-brand methods such as Scrum, Feature-Driven Development, and [XP](#). Agile projects attack complex problems, where product features and design robustness must be balanced against cost and time-to-market. Further, agile projects usually operate in dynamic markets that are not well understood and impossible to predict. Developers and customers embark on an agile project as a learning exploration; they must find their way to the best solution, which may continue to shift over time.

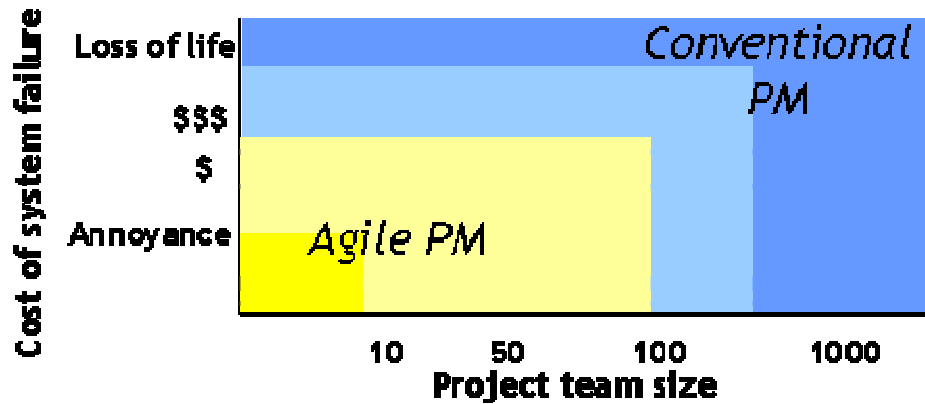
What Problem Is It Trying To Solve?

Many software development projects today are challenged by:

- Customer/user demands to rapidly release a value-added system
- Lack of specificity (or clarity or detail) in requirements
- Rapid, market-driven (or otherwise) evolution of customer needs
- Rapid evolution in the underlying implementation technologies
- Limited knowledge about the underlying technologies or off-the-shelf components to be used to implement the system

Conventional project management, such as the waterfall and spiral method derivations, are some of the most common forms of “deterministic” project management used in software development. The philosophies of these project management methods control projects by formulating a complete plan, giving commands to execute the plan, and then monitoring and ensuring its faithful execution. These practices assume the system to be developed is largely understood; this knowledge is the basis for credible planning (including preplanning for contingencies), especially on large, *complicated* projects. Management focus is on coordinating the many diverse parts that make a large project complicated.

Agile projects attack *complex* problems. The primary challenge of such projects is their inherent unpredictability. Although the general direction of the project is known, no amount of up-front analysis can determine the requirements or the design with confidence. The project team and the customer have to learn what is possible and where their best opportunities lie through actual development.



Where Agile Best Fits

Projects often have both complex and complicated aspects, but usually one characteristic predominates. The above figure compares the suitability of both conventional and agile project management in relationship to criticality (cost of a system failure) and to the total project team size. The current conventional wisdom suggests that projects with higher criticality and larger team size require greater development predictability. Agile development is best suited for projects of small to medium size that are of non-life critical requirements. However, this view may arise from industry's relatively limited experience with agile techniques, especially on larger and more critical projects. The debate among proponents of agile and conventional techniques continues.

Agile Solution

Philosophy

Agile development reflects a distinct philosophy and demands that practitioners accept the assumptions underlying that philosophy. These assumptions define software projects as complex:

- Many aspects of a software project are unknown at its outset. The system providing greatest value to the customer is unclear. The time it will take to produce that system can only be guessed. You can only speculate what problems must be solved to deliver the system.
- You are reasonably confident of the direction the project should take. However, your understanding of the system requirements will change significantly before delivery because your knowledge is incomplete and the customer's needs continue to change. Unanticipated problems will occur as you develop the system.
- Given the above, your best strategy is to rely on the project team to discover issues and opportunities and adapt to them. To succeed, a team must have a rapport among its members and project stakeholders. It must focus on customer value, and learn rapidly and continuously.

Evolutionary Delivery

Agile development embraces rapid evolutionary delivery of software applications, despite unclear and/or evolving requirements and technologies. Under conditions of ongoing discovery and change, an agile development project does not expect to start with a detailed requirements specification, nor does it attempt to plot and follow a detailed plan. Instead, an agile project performs a series of very quick (1 to 13 weeks equivalent to 1% to 5% of the project budget) but complete development cycles. Each evolutionary delivery cycle focuses on a small increment of the system and produces an executable release that is demonstrated to a user/customer.

These evolutionary cycles promote learning (many chances to learn from and correct mistakes) and help the project converge on a solution that best balances all the project's goals and constraints. In particular, these short development/delivery cycles enable:

- Improved assessments of project progress (based on actual working code).
- Credible estimates and schedules. A relatively small piece of work (the next increment) is easier to plan and manage. Over multiple cycles, real measures of progress yield better estimates for each cycle and the project as a whole.
- High-quality releases, from more thorough testing of small increments. Multiple cycles of regression testing increase confidence in the overall product.
- More robust design. Over multiple additions and modifications, the design enables typical changes, responding to the same sort of upgrade demands as it would in “maintenance mode” after being delivered.

More broadly, other agile development techniques employ the framework provided by evolutionary delivery cycles to:

- Build strong relationships both within the team and with project stakeholders
- Understand stakeholder needs and explore new implementation technologies
- Learn and exploit the most important engineering tradeoffs
- Guide the project to earliest delivery of successive operational releases of the application, reflecting the highest-value cost/time/quality/feature balance

Tom Gilb espoused fundamental concepts of agile development in the early 1980s, but the approach has been recently revitalized with a fresh infusion of ideas and techniques from Highsmith, XP, Cockburn, and others.

Agile Life Cycle

Agile development emphasizes
“Responding to change over following a plan”

The agile life-cycle figure below depicts the five phases of agile development. Click on a phase to view details.



In addition, there are a number of supporting practices and techniques that apply in many contexts; they include:

- **Colocation**
Colocation maximizes the potential for fast, efficient team interactions. Key elements are team members working in close proximity in facilities arranged to minimize the time and energy required to conduct ad hoc face-to-face discussion. Some approaches advocate adjoining office space, bullpen offices, and combining common workspaces with private offices. Key challenges include pushback from team members demanding more privacy as well as the furniture police demanding conformance to corporate plans.
- **Diagram of effects**
Diagrams of effects provide a way to represent and reason about complex, non-linear problems. Teams can use them to help reveal effective solutions to those problems. Key elements include identifying the effects that you want to change, determining what impacts those effects positively or negatively, identifying feedback cycles among the effects, and finding appropriate leverage points. The biggest challenge is that it takes practice and experience to use the technique effectively.

- **Facilitated meetings**
Facilitated meetings promote team alignment on specific issues, decisions, or approaches. Use facilitated meetings to plan cycles (e.g., [CRC card-based design sessions](#), [customer focus group reviews](#), and [retrospectives](#)). Key elements include an agenda, the facilitator, the meeting participants, and their results. The main challenges are finding or creating effective facilitators, and getting team participation.
- **Issue tracking**
Issue tracking is a valuable mechanism for retaining project history, recording rationales and ensuring that the project stays focused on customer priorities. Key elements include recording and prioritizing issues (i.e., features, deliverables, risks, and defects). Tracking these items to closure gives customers and project managers confidence in the project. Project culture should determine the specific mechanisms used (e.g., whiteboards, spreadsheets, databases, or purpose-built tools).
- **Spike solutions**
When faced with a serious unknown, such as major design issues or generating credible effort estimates for unfamiliar territory, you can establish a task to explore the issue as a spike solution. The key element is creating a simple program or model to explore solutions. This program or model is not put into the production code, no matter how tempting. Challenges include the aggregate cost of too many spike solutions and successfully resisting the urge to leave those spikes in the product.

Macro Plan Phase

The purpose of the macro plan phase is to establish a project basis or starting point, including:

- Stakeholder objectives and constraints
- Measurable success criteria
- Project unity statement

When the project is committed:

- Endorse the project's value proposition
- Conceptualize the overall system technical approach
- Estimate costs
- Establish the project budget
- Establish the project schedule
- Risk management

Practices employed in the Macro Plan phase include:

- **Feature list**
The feature list establishes the project scope. Its key elements are statements of project requirements in terms of features to be provided. Customers provide the prioritization of features within the list. You can describe features as actor/goal pairs or user stories. You can assign the features speculatively to release cycles.
- **Release scheduling**
Release scheduling establishes a target cycle length based on the project culture. Some projects require a series of development cycles going into a single release cycle. Some practitioners recommend between 2 and 5% of schedule and budget for a cycle. Others recommend between 1 and 13 weeks. The greatest challenge to release scheduling is finding an appropriate cycle for a new situation.
- **Risk management**
Use risk management techniques to avoid or reduce the impact or likelihood of risks to the project's success. The key elements are first identifying those risks, their likelihood, and their potential impact, and then developing mitigation strategies to reduce the probabilities and impacts. The key challenges are "knowing what you don't know" and prioritizing the risks that you know about.
- **Technology exploration**
You can jumpstart progress on tricky issues or concerns with a technology exploration activity. The key elements are using prototyping or conceptualization techniques to compare design alternatives or to explore new technologies or components. The key challenges are avoiding sinking too much effort (10% of the project or more) into the activity and canceling a project when the risks are

too high.

- **Unity statement**

Have the team develop a unity statement, ideally in a [facilitated meeting](#). A unity statement is an expression of team alignment on goals and direction. The key elements are a project vision, methodology, and team consensus on these issues. The key challenges are establishing the direction of the project without presuming specifics of the destination and building consensus in the face of cynicism and divergent personal agendas.

Cycle Plan Phase

The purpose of the cycle plan phase is to launch a new cycle including:

- Revisiting the Macro Plan to adjust assumptions, goals, cycle length, and risk management strategies
- Establishing a scope for the current cycle
- Defining, and possibly, assigning tasks for the current cycle

Practices employed in the Cycle Plan phase include:

- **Bottleneck management**

Bottleneck management is making the best possible use of overtaxed specialists. The key elements in bottleneck management are identifying those overtaxed specialists (the bottlenecks) and modifying your methodology to reduce their load. Possible approaches are increasing the precision or stability of their input artifacts, accepting less finished work from them, or assigning non-specialists to assist them. Challenges include Brooks' law (adding resources to an overdue project makes it later [[Brooks 1982](#)]) and creating new bottlenecks when trying to remove existing ones.

- **Feature selection**

Selecting the features to be addressed in a cycle sets the scope of the cycle. Feature selection starts with a facilitated meeting in which the customer selects features from a new feature list or from the project issue backlog. Then the team, as a whole, estimates the effort required to create the feature or resolve the issue. The challenge for feature selection is, of course, choosing the right ones. Some approaches attack the features with the most customer value; others prioritize based them on risk.

- **Goal-directed use cases**

Goal-directed use cases elaborate feature requirements in a way that keeps the customer involved. Key elements of goal-directed use cases include focusing on externally visible interactions between users and the system, maintaining a user point of view, describing the feature as a narrative of system usage, and covering the alternative courses as branches from the main success scenario. The

challenges are abstracting away from user interface detail, treating the system as a black box, and knowing how much detail is enough for a given situation.

- **Project velocity**

Project velocity refers to matching tasks to the cycle length. Velocity is the ratio of actual effort to estimated effort. Estimates are created in "perfect engineering days." Actual effort is measured in calendar days from task start to task finish. At the end of a cycle, velocity is computed, folded in with velocity from previous cycles, and used to help refine estimates for the next cycle. Challenges associated with project velocity include guessing what it should be for the first project cycle ([XP](#) proponents recommend starting with 2.5 to 3) and resisting the urge to trim estimates.

- **Self-tasking**

Self tasking addresses identifying and assigning tasks for the cycle. The team builds a task list together, members volunteer for tasks and then estimate the effort required for their tasks. If the cycle gets too large in terms of the overall time required, the team must renegotiate the scope. Alternative approaches have technical leads defining tasks and teams prioritizing tasks before assigning them. The challenges are breaking larger tasks into cycle-friendly chunks and ordering tasks based on dependencies among them.

Do Phase

The purpose of the Do phase is to accomplish the work scoped out in the Cycle Plan phase. Depending on the cycle, this may involve

- Designing and or coding selected features
- Elaborating requirements as needed
- Maintaining team synchronization
- Maintaining product quality
- Developing tests to enhance confidence in the code
- Maintaining customer awareness of progress

Practices employed in the Do phase include:

- **Automated testing**

Automated testing provides continuous feedback on progress and quality. Various levels of test, including unit, feature, and acceptance tests, are an expression of your detailed requirements. Key elements of automated testing include constructing a test framework in parallel with software feature development, regression testing every code submission, and designing software to maximize the ability to perform automated testing. Difficulties include [GUI](#) and hardware-in-the-loop testing, and choosing appropriate tests.

- Coding standards**
 Coding standards ensure the uniformity of code within a project; the idea is that uniformity enhances understandability. Key elements of coding standards include naming and formatting conventions, consistent use of specific language features, required commentary, and consistent use of the file system. The challenge with coding standards is enforcement.
- Collective code ownership**
 Collective code ownership ensures that team members are empowered to maintain code quality. Any team member can change any code with appropriate configuration management. The challenges are communicating about and coordinating rapid code changes. Some approaches advocate the designation of feature or class owners in the team so that the corresponding units in the product have a design authority.
- Continuous team integration**
 Continuous team integration means keeping the team together in terms of currency to avoid inconsistency in designs caused by knowledge gaps or divergent personal agendas. Key elements include frequent baselining of code under development and matching communication media to project information. The team should decide how often to integrate and what triggers integration. Possibilities include daily, on task completion, or on an artifact state change.
- CRC cards**
 The use of [CRC](#) cards enables a lightweight, "good enough" design practice to ensure that sufficient information is available to coders. CRC cards is a team-based design approach, using cards to represent classes, responsibility and collaboration.
- Customer on site**
 Keeping the customer tightly integrated with the project ensures that the team gets timely help on elaborating and interpreting requirements, and that the customer is enrolled in project progress. Key elements include locating a customer representative (or surrogate in cases where a customer is unattainable) with the project team, getting the customer to answer questions or resolve issues on demand, and involving the customer in acceptance test development. Challenges include getting a customer to provide a representative and balancing verbal versus recorded communications.
- Inspections**
 Inspections are an efficient defect detection mechanism. An inspection is a facilitated meeting of authors and reviewers in which the reviewers step through the artifacts being inspected, someone records issues, and the authors resolve the issues after the meeting. Challenges include resisting the urge to solve problems in the inspection meeting and balancing the cost of meetings with the

benefits obtained.

- **Monitoring and adjusting**

Monitoring progress and adjusting the cycle scope to match progress are required to ensure that cycles stay manageable. Key elements include tracking effort days expended on task, comparing expended effort with estimates, and renegotiating cycle scope with the customer, as needed. The challenge is recognizing the significance of a deviation between planned and actual effort. Insignificant deviation causes wasted adjustment efforts. Waiting until a deviation becomes a disaster wastes effort when the tasks involved impact dependent tasks.

- **OOASIS design**

Using [OOASIS](#) for design generates flexible designs that minimize the cost of downstream changes and enable you to maintain and communicate a close integration between the software design and the problem. Key elements include modeling the problem in terms of actors, agents, device interfaces, and persistent entity classes and performing the realize, allocate, elaborate, and define concurrency activities.

- **Pair programming**

Pair programming is the practice of teaming pairs of programmers who work side-by-side using one computer and sharing its keyboard. This practice contributes to good design, low defect rates via continuous inspection, and peer-reinforced standards. The challenges are an initial productivity cost to ramp up and overcoming programmer resistance based on concerns for privacy, independence, interference, and skill differences.

- **Refactoring**

Refactoring is the process of revising existing code to improve its design. Key elements include identifying poor structures, ensuring that tests verify behavior of code to be modified, changing the code to preserve correct behavior while improving its design, and verifying the revisions with test. Challenges are recognizing when refactoring is advantageous and knowing where and when to stop refactoring. Possible triggers include the end of a cycle and before the addition of a new feature.

- **Strict software configuration management**

Configuration management is a control on the chaos that can result from rapid and/or concurrent updates to artifacts. For code, [SCM](#) requires that you maintain a common baseline and use a tool that supports branching, cycle tagging, and mainline integrity, and enforces quality constraints such as requiring that submitted code pass all unit tests. The biggest challenge in SCM is merging changes produced in parallel by different developers. Tool support in this area is variable.

- **Test first**
The test-first approach is intended to focus development efforts on quality within essential functionality. In this approach, you write code only in response to test failure; development is driven by the need to get tests passed. This practice avoids big-bang development of gold-plated components and encourages developers to focus on small incremental improvements just adequate to pass tests. The challenge is getting developers to accept the idea.
- **Unit testing**
The purpose of unit testing is to catch defects as early as possible and prevent them from slipping into later stages where they will cost more to fix. Key elements are that each module or class has a test that verifies only that module or class, dependent modules are stubbed out, anything that can fail is tested, and a specific test must be added to the test suite for every defect found. The challenge with unit testing is creating effective tests.
- **Variability analysis**
Variability analysis is a technique for identifying likely changes to requirements or design constraints so that you can respond by building in flexibility. Key elements include identifying volatilities, enumerating possible changes, and bounding the changes. The challenges are determining the most likely changes and balancing the cost of design flexibility against the likelihood of change.

The Learn Phase

The purpose of the Learn phase is to provide a checkpoint at which you

- Validate the results of the previous cycle
- Identify process or other needed improvements
- Review the project objectives

Practices employed in the Learn phase include:

- **Customer focus group reviews**
The purpose of customer focus group reviews is to validate delivered items and generate any needed customer direction. Key elements include a facilitated meeting at which the team observes customers learning details about the latest delivery, possibly through a demonstration, test drive, or actual production use. Challenges include getting feedback from customers without defensiveness.
- **Retrospectives**
A retrospective is a chance to examine the just-completed cycle and identify any needed improvements. Key elements include a facilitated meeting with the entire project team that follows soon after a related customer focus group review. In this meeting, the team discusses both problems and successes, revisits the project objectives, agrees to make a

needed change to a single aspect of their methodology (one per cycle) in response, and documents its retrospective results.

Deliver Phase

The Deliver phase of the agile lifecycle delivers a working system for review and, ideally, puts that system into production use. The agile development focus on delivering working versions of the system in short, regular cycles yields these advantages:

- Brings early [ROI](#) to the customer.
A project is nothing more than a black pit of cost until at least some version of the system has been put into productive use. Rapidly delivering a system with just the most valuable functionality, then repeatedly updating that system with the most valuable remaining functionality gives customers the maximum payback on their investment as quickly as possible.
- Enables valuable feedback.
By far the most reliable feedback on a system is that received as a result of the customer's use of it. By far the most useful feedback to a project is timely feedback, reliable information to the development team concerning what is good and bad about the system while there is still time for the team to do something about it. Frequently, customers do not know what they need or want (and often, no one else can figure it out) until they see the system in operation. Frequently, delivering working versions of the growing system enables the customer and the development team to correct problems, make adjustments, or even redirect the project for the remaining schedule and budget.
- Provides credible measure of progress.
One of the [principles underlying the Agile Manifesto](#) states: "Working software is the primary measure of progress." Various attributes of intermediate project artifacts (lines of code, "goodness" or "completeness" of a design or requirements specification) are only faint indications of project progress compared to the reality of a working system and the customer's perception of it. Although a working system is an imperfect indicator of how much work remains to be done, it is at least a true representation of what you have accomplished so far.
- Mitigates risk.
Most project risks -- including technical risks such as requirements validity, design feasibility, and code correctness, and management risks such as schedule and budget -- can only be laid to rest after some version of a working system is delivered. Moreover, the development team has the opportunity to practice deploying the system into the production environment. On non-agile projects, the project team has no opportunity to face the many difficulties and uncertainties of system deployment, with all its attendant risks, *until the very end of the project* when the consequences of failure are the greatest.
- Establishes project rhythm; builds confidence.
Subtle, but as important as any of the previous advantages, the regular delivery and

deployment of an enhanced system establishes a habit of success. Visible success creates its own momentum. The customer is motivated to become further engaged in the project and to continue the project; the team is motivated to build on its previous successes. The customer becomes confident in the project team, and the project team becomes increasingly self-confident. The rhythm of regular development and delivery cycles provides a consistent background for the team to settle into successful work patterns.

The details of how system delivery and deployment is accomplished differ in each project with the type of system and the nature of the production environment. There are no agile delivery practices per se.

However, there are fallback strategies to use when the cost of deployment into the customer's production environment is high enough to inhibit frequent deployment of new system versions. Frequent delivery of working system versions can proceed at an optimal rate even when deployment of every delivery is prohibited by cost or other difficulties. Admittedly, some of the benefits are compromised; the customer cannot receive value from an undeployed system. But the feedback the project needs in the [Learn phase](#) of the agile life cycle can still be supported by:

- Limited deployment of the delivered system
- Setting up an emulated environment for the customers to test drive the system
- Demonstrating the system to customers in a test environment

Agile Resources

This page contains links to resources providing more information on agile development practices and related topics, arranged in these sections:

- [General overviews of agile development](#)
- ["Name Brand" Agile Methods](#)
- [Agile Practices](#)
- [Agile Experience](#)
- [Miscellaneous Topics](#)

General Overviews

General overviews of agile development:

Web - Agile Alliance (www.agilealliance.com).

Paper - "Extreme Programming" ([Highsmith 2000b](#)).

Book - *Agile Software Development* ([Cockburn 2002](#)).

Course - [Managing Agile Software Development](#) (Software Productivity Consortium)

Survey of agile methods:

Agile Software Development Ecosystems ([Highsmith 2002a](#)).

"Name Brand" Agile Methods

[Adaptive Software Development](#) - Jim Highsmith's agile method written about in [Highsmith 2000a](#).

[Crystal](#) - Alistair Cockburn's (pronounced "Coburn") set of agile methods, described briefly in [Cockburn 2002](#).

[Dynamic Systems Development Method](#) - An agile method supported by a consortium dedicated to it. Unfortunately, you need to join in order to look at the detailed method description. You can read an overview of it in [Highsmith 2002a](#).

[Evolutionary Development](#) (Evo) - Tom Gilb's approach predates the term "agile development" but it is the original agile approach to system development. Gilb will also let you download a draft guidebook for the approach from his [download center](#).

[XP \(extremeprogramming.org, extremeprogramming.com\)](http://extremeprogramming.org) - Currently the most popular agile method, as well as the most talked about. It is an extremely disciplined method focusing on the engineering activities. There are several books written about XP, but the best is [Jeffries, Anderson, and Hendrickson 2001](#).

[Feature-Driven Development](#) - An agile method closer to traditional development approaches. More planning and design is done up front. Each feature is planned to the month of its release, but no more detail than that, leaving room for agility. It has been used successfully in a 250-person project. You can read about it in detail in [Palmer and Felsing 2002](#).

[Lean Development](#) - Applies principles of [Lean Thinking](#) to software development.

[Scrum](#) - Mostly a management method that is completely agile. Its main focus is exchanging status information every day in a daily standup meeting.

Agile Practices

Following is a list of sources with more detailed information on selected agile practices.

Practice	Sources
Macro Plan Phase	
Feature list	Jeffries, Anderson, and Hendrickson 2001
Release scheduling	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/planninggame.html
Risk management	www.dacs.dtic.mil/databases/url/key.hts?keycode=270
Unity statement	Highsmith 2000a More broadly on team building see: Robbins and Finley 2000 , DeMarco and Lister1987 and McCarthy and McCarthy 2002
Cycle Plan Phase	
Bottleneck management	Cockburn 2002 , pages 129-134
Feature selection	Jeffries, Anderson, and Hendrickson 2001
Goal-directed use cases	Cockburn 2000 OOASIS Capturing System Behavioral Requirements
Project velocity	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/velocity.html
Self-tasking	Jeffries, Anderson, and Hendrickson 2001

Do Phase	
Automated testing	Jeffries, Anderson, and Hendrickson 2001 www.xprogramming.com/software.htm www.junit.org/index.htm Consortium's Test Automation Framework Acceptance testing: Jeffries, Anderson, and Hendrickson 2001 and www.extremeprogramming.org/rules/functionaltests.html
Coding standards	C , C++ , Java
Collective code ownership	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/collective.html
Continuous team integration	Continuous integration of code: Jeffries, Anderson, and Hendrickson 2001 and http://www.extremeprogramming.org/rules/integrateoften.html Daily standup: www.extremeprogramming.org/rules/standupmeeting.html and www.controlchaos.com/rules.htm
CRC cards	<i>The CRC Card Book</i> by David Bellin, Susan Suchman Simone; Addison-Wesley Object Technology Series, 1997. ootips.org/crc-cards.html
Customer on site	www.extremeprogramming.org/rules/customer.html
OOASIS design	OOASIS Define System Software Architecture
Inspections	<i>Software Inspection</i> by Tom Gilb and Dorothy Graham; Addison-Wesley Longman Ltd., 1993. www.result-planning.com/Pages/2ndLevel/gilbdownload.html
Pair programming	Jeffries, Anderson, and Hendrickson 2001 www.pairprogramming.com
Refactoring	Fowler 1999 www.refactoring.com
Sustainable pace	Jeffries, Anderson, and Hendrickson 2001
Strict SCM	www.faqs.org/faqs/sw-config-mgmt/faq
Test first	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/testfirst.html
Unit testing	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/unittests.html
Variability analysis	OOASIS Capturing System Behavioral Requirements: Variations section Consortium's Product Lines Overview

Learn Phase	
Customer focus group review	Highsmith 2000a
Retrospectives	Kerth 2001 www.retrospectives.com
Supporting	
Diagram of effects	Weinberg 1992
Facilitated meetings	Kaner 1996
Spike solution	Jeffries, Anderson, and Hendrickson 2001 www.extremeprogramming.org/rules/spike.html

Agile Experience

Don Reifer in ([Reifer 2002](#)) provides the results of a survey of 31 organizations (in 14 companies) using agile methods in a variety of markets and development environments. The results show improvements in productivity (15-23%), cost (5-7%), and especially time-to-market (25-50%) over industry norms with no loss in quality (defined in terms of defect rate).

MacCormack's study ([MacCormack 2001](#)) shows a strong correlation between high product quality (defined here as "a combination of reliability, technical performance (such as speed) and breadth of functionality" relative to "other products that targeted similar customer needs at the time the product was launched") and agile-like methods such as evolutionary delivery, rapid feedback to the development team via continuous integration and automated testing, and "major investments in the design of the product architecture (interpreted as denoting attention to a modular and scalable architecture capable of incorporating significant changes in the late stages of the project).

Teasley's study ([Teasley et al. 2000](#)) focused on just the effect of "radical collocation" (entire team works in one open room), one of the practices of XP and other agile approaches. The study followed 6 pilot projects at a major automobile company. Statistics are given for the subsequent production projects after the company committed to collocation (as a result of the pilots' success). Despite initial developer concerns about loss of privacy, quiet, and concentration, productivity doubled and time-to-market dropped to a third of previous projects at the company (and similarly compared to the industry standard). In production projects, productivity nearly *quadrupled*. Moreover, all the stakeholders, including the developers preferred collocation at the end of the study.

Other available experience reports are largely anecdotal. The report on Symantec's (a mid-sized, multi-site development company) adoption of XP ([Morales 2002](#)) is particularly interesting.

Miscellaneous Topics

Different Development Lifecycles

["Another Look at Incremental and Iterative Development"](#) by [Pascal Van Cauwenberghe](#) in the Winter 2002 issue of the [Methods and Tools](#) online magazine contains an excellent discussion of the history, characteristics, and rationale of the various styles of development lifecycles (waterfall, spiral, iterative, incremental, agile).

Compliance with Process Standards

There has been a great deal of controversy and some number of papers concerning whether agile development is compatible or in conflict with the [CMM](#) and [CMMI](#). Although we have heard claims that XP and agile approaches have passed CMM level 3 assessments, much appears to depend on the interpretation of the assessors. So far, we have not seen a detailed analysis of compliance issues that we can recommend. ["Extreme Programming and the Capability Maturity Model" by Ron Jeffries](#) (January 2000) is a high-level discussion of how the levels of the CMM match up with the practices of XP.

Agile compliance with [ISO9000](#):

- ["Quality Management and the Agile Approach"](#) (May 2002) from the [Synop Website](#). Again, a high-level discussion mapping quality principles from ISO9000 to agile principles.
- ["Combining Extreme Programming with ISO 9000"](#) ([Nawrocki et al. 2002](#))
The authors propose specific changes in XP to enable compliance with ISO 9000.

From <https://www.software.org/MembersOnly/agile/default.asp?PageID=3> 10/29/03

Glossary and Acronyms

Acronym	Term	Definition
	bottleneck management	The practice of shifting the work load between specialists and other developers in each delivery cycle by adjusting the precision and/or stability of the inputs given to the specialists or expected from them.
CMM	Capability Maturity Model	An industry standard describing key goals and practices for consistent quality in developed software. Consists of five maturity levels.
CMMI	Capability Maturity Model Integration	A combination of many different maturity models into one integrated model for management and engineering of systems.
CRC cards	Class, Responsibility, and Collaboration cards	An agile design technique that records each class name and, optionally, its responsibilities and collaborating classes on a separate index card. Further, the development team uses the cards as a prop in design sessions to explore how classes should interact.
	diagram of effects	A systems analysis technique employing a graphical notation to explore and record how different aspects of an environment have positive and negative influences on each other.
XP	eXtreme Programming	Currently, the most widely known and practiced of the various name-brand agile processes.
	goal-directed use cases	An interpretation of the general use case technique where each use case corresponds to a goal of a system stakeholder. A use case is a step-wise narrative of inputs and outputs between the system and external entities (called <i>actors</i>).
GUI	graphical user interface	A user interface oriented to mouse use (as opposed to a "command line" interface).
ISO 9000		The set of standards from the International Organization for Standardization primarily concerned with creating and sustaining a quality management system.
OO	object-oriented	Using objects, entities that are a combination of function and data, as the key demarcation of a system.
OOASIS	Object-Oriented Approach to Software-Intensive Systems	The Consortium's current methodology for object-oriented software/systems development.

	pair programming	A technique of having programmers work in pairs while coding and unit testing.
	project velocity	Ratio of actual calendar days to estimated "perfect engineering days" in completing tasks.
QA	Quality Assurance	The function of quality assurance and/or an organization chartered to perform that function.
	refactoring	Specifically, a set of techniques for simplifying and improving the structure of object-oriented code in a systematic, step-wise fashion. More generally, the activity of simplifying and improving the structure of any sort of code.
	retrospectives	The practice of holding frequent team meetings (usually, at the end of every development cycle) to reinforce those practices that have been working well and modify other practices to improve future cycles.
ROI	Return on Investment	The ratio of return to investment made. Usually stated in percentages.
SCM	Software Change Management	As used with regard to agile development, this term refers to the procedures (usually tool supported) by which updates are made to a base of controlled code in a disciplined fashion.
	spike solutions	Rapid development of a throw-away prototype to clarify a requirement, explore a potential design approach, or investigate specific properties of interest in an existing component or technology.
TAF	Test Automated Framework	The Consortium technology supporting requirements analysis and verification, including automated test generation, execution, and results analysis.
	test first	The practice of writing the automated unit tests (and testing framework) before writing the code those tests are intended to verify.
	unity statement	The practice of creating and maintaining a written statement of project vision (including goals and constraints) and process via facilitated team meetings.
	variability analysis	The practice of explicitly recording alternative requirements or design constraints that have a good probability of becoming actual.

Bibliography

The following is a list of books and papers that have guided the Consortium's views on Agile Software Development.

Ambler 2002a	Agile Modeling. John Wiley & Sons. <i>This book covers how, when and why to model in an agile project. Offers lots of practical advice and real-world examples. A must read for the technical lead of an agile project.</i>
Ambler 2002b	“Duking it Out.” Software Development, July 2002. <i>An interesting read discussing points made by Barry Boehm about agile and what agile proponents thought about his comments.</i>
Armour 2000	"The Case for A New Business Model." Communications of the ACM, 43(8), pages 19-22, August 2000. <i>Armour argues that software development is more like knowledge acquisition than producing a product.</i>
Armour 2001	"Zeppelins and Jet Planes: A Metaphor for Modern Software Projects." Communications of the ACM, 44(10), pages 13-15, October 2001. <i>In this essay Armour asserts that traditional project management assumes long-range planning will be predictable, like aiming cannons at Zeppelins, whereas modern projects demand that we give up any expectations for that level of control.</i>
Beck 2000	Extreme Programming Explained. Addison-Wesley. <i>This is the classic text on XP. Gives a good foundation on XP with a good bit of justification.</i>
Brooks 1982	The Mythical Man-Month. Addison-Wesley. <i>A classic text that tells of the reasons why adding people to a late project makes it later.</i>
Brown, et. al. 1998	AntiPatterns. John Wiley & Sons. <i>A good list of common mistakes made in software projects. It also has recommendations for each antipattern.</i>
Cohn and Ford 2002	“ Introducing an Agile Process to an Organization. ” <i>This is a good introduction into the problems and solutions of introducing an agile process into an organization for the first time.</i>

Conway 1968	"How Do Committees Invent?" Datamation, April 1968, pages 28-31. <i>A seminal article explaining why the structure of a system design is invariably congruent with the organizational structure of the design team, and the unfortunate consequences of this. Further, it insightfully foreshadows Brooks' (Brooks 1982) mythical man-month observations.</i>
Cockburn 1998	Surviving Object-Oriented Projects: A Manager's Guide. Addison-Wesley. <i>The best book out on managing OO projects. Cockburn cuts through most of the hype and gives you sound, practical advice.</i>
Cockburn 2000	Writing Effective Use Cases. Addison-Wesley. <i>This book was read in draft form and was very influential in the Capture System Requirements task of OOASIS. The best treatment of use cases we have seen.</i>
Cockburn 2002	Agile Software Development. Addison-Wesley. <i>This is a great introduction to agile software development.</i>
DeMarco and Lister 1987	Peopleware: Productive Projects and Teams. Dorset House Publishing Company. <i>This classic book was one of the first to emphasize the human aspect of software development. Most of this book is still applicable today.</i>
Drucker 2001	The Essential Drucker. Harper Collins Publishing. <i>This is a collection of classics from the guru of management. Gives a good foundation for any manager and how the job fits into the big picture (delivering customer value). He also explains how management is a social function.</i>
Elssamadisy and Schalliol 2002	"Recognizing and Responding to Bad Smells in Extreme Programming." http://www.thoughtworks.com/library . <i>Covers how to react to problems with XP. Has some good general guidance even if you are following another agile method.</i>
Fowler 1999	Refactoring. Addison-Wesley <i>A great book that gives practical advice on how to improve the design of existing code.</i>
Gilb 1985	"Evolutionary Delivery versus the Waterfall Model." ACM Sigsoft Software Engineering Notes, Vol. 10, No. 3, pp. 49-62. <i>This is the first article on Evolutionary Delivery (later named EVO). Gilb was preaching short delivery cycles long before XP. The article is a short read and hits the high points, arguing the problems with the waterfall approach versus the benefits of short delivery cycles.</i>
Gilb 1988	Principles of Software Engineering Management. Addison-Wesley. <i>This book covers EVO (Evolutionary Delivery) in more depth than (Gilb 1985).</i>

Highsmith 2000a	Adaptive Software Development. Dorset House Publishing. <i>This book has some interesting ideas in it. However, it drags on in spots and is generally hard to follow. We can only recommend this book if you have read many other agile books and are looking for more insight on select topics.</i>
Highsmith 2000b	"Extreme Programming." Agile Project Management Advisory Service. http://www.cutter.com/project/ead0002.html . <i>One of the best introductory articles on XP you can find.</i>
Highsmith 2002a	Agile Software Development Ecosystems. Addison-Wesley. <i>Covers the main players in the agile movement along with the most popular agile methods. Has a good introduction that clearly spells out where agile fits in the software business.</i>
Highsmith 2002b	"Does Agility Work?" Software Development, June 2002. <i>This is a good article discussing real examples of agile development in practice and what the practitioners thought about the experience.</i>
Jeffries, Anderson, and Hendrickson 2001	Extreme Programming Installed. Addison-Wesley. <i>This is the best overall book on XP. Gives practical guidelines as well as justification for the practices.</i>
Kaner 1996	Facilitator's Guide to Participatory Decision Making. New Society Publishers. <i>This is a great book for any current facilitator as well as anyone who needs to learn the art of facilitation.</i>
Kerth 2001	Project Retrospectives. Dorset House Publishing. <i>This is a good book on how to conduct a retrospective. The book focuses mostly on post-project, 3-day retrospectives, but it can be applied on smaller ones.</i>
MacCormack 2001	"Product-Development Practices That Work: How Internet Companies Build Software," MIT Sloan Management Review, Winter 2001. <i>MacCormack's study of a variety of products (including web-based and not web-based, commercial and consumer) show strong correlation between evolutionary delivery and product quality.</i>
Maguire 1994	Debugging the Development Process. Microsoft Press. <i>This a great book about how to fix a broken development process, which is pertinent to most projects. Many of his recommendations fit nicely into agile development, but this book was published before the movement began.</i>

<p>McCarthy and McCarthy 2002</p>	<p>Software For Your Head. Addison-Wesley. <i>A groundbreaking book that claims to have solved how to create successful, aligned teams. Gives you a protocol, the CORE, for all team interactions, developed over years of running BootCamp workshops. The CORE protocol is a little radical, but you can gain significant insight from the introduction alone. The book is an exposition of the authors' primary mantra: "team = product." Even if you don't plan on following the CORE, this book is filled with good reasons to change the way your team interacts. For example, the book advocates that team decisions should always be unanimous. Your first reaction is that this would never work, but you would be surprised at how compelling an argument the McCarthy's put forth in favor of this approach.</i></p>
<p>Morales 2002</p>	<p>"Embracing Change." Software Development, January 2002. <i>A good example of how XP can be adapted in a real-world situation, including dealing with an independent Quality Assurance organization and other pressures of an organization used to traditional processes for delivering high quality.</i></p>
<p>Nawrocki, Jasiński, Walter, and Wojciechowski 2001</p>	<p>"Combining Extreme Programming with ISO 9000", 1st EurAsian Conference on Advances in Information and Communication Technology, October 2002, <i>The authors propose specific changes in XP to enable compliance with ISO 9000.</i></p>
<p>Newkirk and Martin 2001</p>	<p>Extreme Programming in Practice. Addison-Wesley. <i>Details an example of XP on a small program. A fun read, but light on applicable information. Read the other books on XP and only get this if you just can't get enough reading about XP.</i></p>
<p>Palmer and Felsing 2002</p>	<p>A Practical Guide to Feature-Driven Development. Prentice Hall PTR. <i>This is the best book about the agile process known as Feature-Driven Development. It gives you guidance as well as real-world examples.</i></p>
<p>Reifer 2002</p>	<p>"How Good Are Agile Methods?" IEEE Software, July/August 2000. <i>Survey of 14 firms employing agile methods shows promising results in productivity, cost, and time-to-market.</i></p>
<p>Robbins and Finley 2000</p>	<p>The New Why Teams Don't Work: What Goes Wrong and How to Make It Right. Berrett-Koehler Publishers. <i>Good general text on teams. Doesn't advocate anything radical and has several good points. We especially like the chapter "Toxic Teaming Atmosphere".</i></p>

<p>Senge 1990</p>	<p>The Fifth Discipline. Currency Doubleday. <i>This is a classic on the learning organization. The five disciplines (team learning, personal mastery, mental models, shared vision, and systems thinking) all resonate with the philosophies of agile development.</i></p>
<p>Teasley, Covi, Krishnan, and Olson 2000</p>	<p>"How Does Radical Collocation Help a Team Succeed?" In <i>Proceedings of CSCW 2000</i> (pp. 339-346), ACM Press. Also available to subscribers of the ACM Digital Library. <i>A study showing an increase in productivity of from 2-4 times in multiple development teams at a large automobile company, attributed largely to the agile practice of team collocation.</i></p>
<p>Weinberg 1992</p>	<p>Quality Software Management: Systems Thinking. Dorset House Publishing. <i>This is a good introduction into systems thinking and how it can and should be applied to software project management.</i></p>
<p>Weinberg 1993</p>	<p>Quality Software Management: First-Order Measurement. Dorset House Publishing. <i>This is a refreshing look at measurement and communication. It explains how the viewpoint of the observer biases the measurement. It provides practical advice as well as an in depth look at the Satir interaction model of communication.</i></p>
<p>Williams et. al. 2000</p>	<p>"Strengthening the Case for Pair Programming." IEEE Software, July/August 2000. <i>This article tries to make a case for pair programming's benefits versus its costs.</i></p>