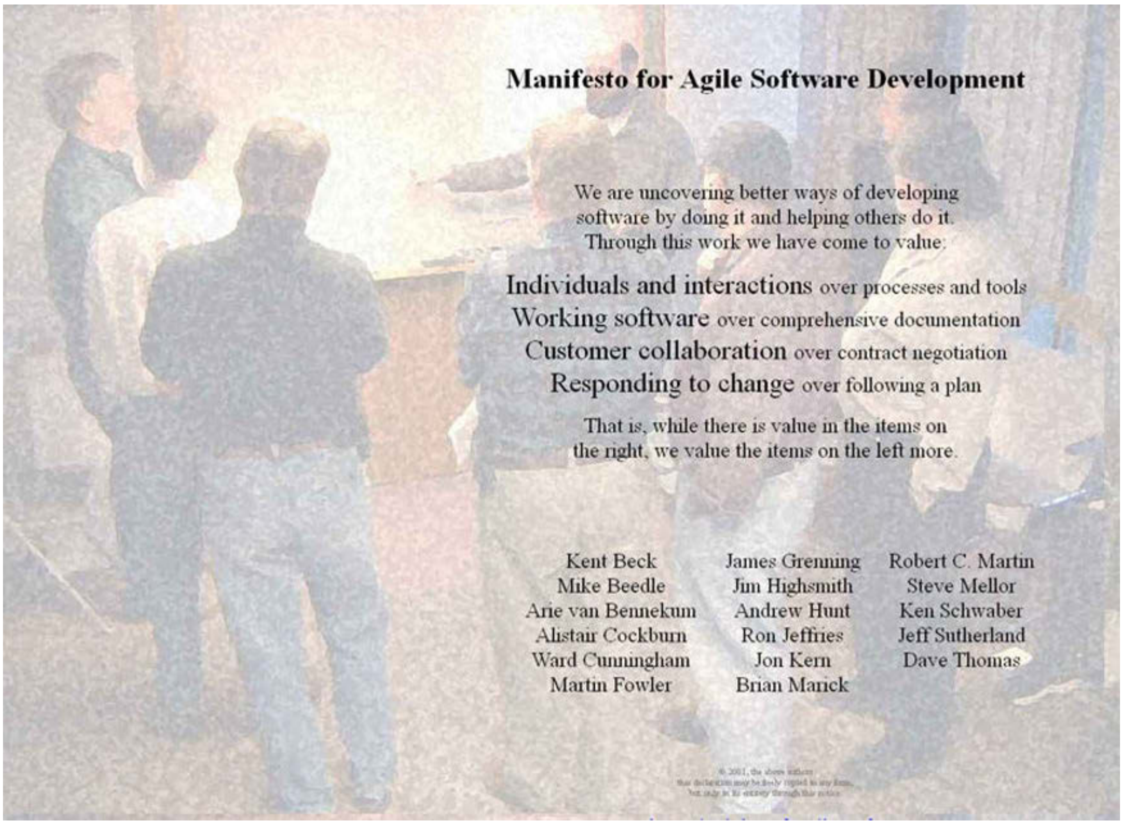
1 Overview

Agile ideas date back to the development of Extreme Programming in the 1990s, but reached fame with the appearance in 2001 of the “Agile Manifesto”: [Agile 2001]



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to 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 there is value in the items on
the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

The sight of a half-dozen middle-aged, jeans-clad, potbellied gentlemen turning their generous behinds to us appears to have provided the decisive sex appeal. Personally, had I wanted to convey the suggestion of agility, I might have turned to something like the cover photograph of this book — which only demonstrates how out of tune I am with the times, since the above picture was successful beyond anyone’s dreams. Agile ideas have

B. Meyer, *Agile!*, DOI 10.1007/978-3-319-05155-0_1,
© Springer International Publishing Switzerland 2014

become the buzz of the industry, the darling of the technical press, the kind of argument companies use in the fierce competition to attract the best programmers: *Come to us! Our development is agile!*

Rather than a single development method, "agile" denotes a compendium of ideas, which a number of full-fledged methods — particularly Extreme Programming (XP), Scrum, Lean Software and Crystal — apply in various subsets and combinations; many people also use some of the ideas without embracing a complete method. In this chapter we get into the mood, if not the details, of agile methods, by reviewing their core characteristics:

The abbreviation XP comes from the alternative capitalization "eXtreme Programming".

- *Values*: general assumptions framing the agile view of the world (1.1).
- *Principles*: core agile rules, organizational and technical (1.2).
- *Roles*: responsibilities and privileges of the various actors in an agile process (1.3).
- *Practices*: specific activities practiced by agile teams (1.4).
- *Artifacts*: tools, both virtual and material, that support the practices (1.5).

The principles follow from the values; the practices, roles and artifacts follow from the principles. The last section (1.6) provides a first assessment of the approach.

This chapter serves as a concentrate of the rest of the book, surveying the key ideas concisely. Except for the last part, it is descriptive, presenting the agile ideas neutrally. For brevity, it does not cite (with one exception, below on this page) from the agile sources describing the techniques summarized here; the following chapters include numerous citations from agile texts, where the authors explain their rationale in detail.

1.1 Values

Reading the Agile Manifesto on the previous page is enough to show that "Agile" is not just a collection of software techniques but a movement, an ideology, a cause. Going even further, one of the creators of Scrum declares that "*Agile is an emotion*". To describe the fundamental underlying assumptions, agile proponents like to use the term "values". Before looking at specific principles, practices, roles and artifacts, we must get a feel for the agile philosophy, in the form of five general tenets:

[Sutherland 2009], at 5:59.

Agile values

1 Redefined roles for developers, managers and customers.
2 No "Big Upfront" steps.
3 Iterative development.
4 Limited, negotiated functionality.
5 Focus on quality, understood as achieved through testing.

The first tenet affects a fundamental feature of project development: the **role of developers and managers**. Agile methods redefine and limit the manager's job by transferring many of the duties to the *team* as a whole, including one of the most important responsibilities: selecting tasks to be performed and assigning them to developers. It is possible to give a sociological interpretation of the agile movement as a "*revolt of the cubicles*": the rejection of rigid, top-down, Dilbert's-boss-like techniques for managing software projects. Programmers in the trenches — the cubicles — often resent these attempts as ignorant of the specific nature of software development. The Dilbert types know that documents and diagrams do not make a system: code does. Agile methods are, in part, the rehabilitation of code.

The redefinition of roles also affects *customers*, who in the agile world are not passive recipients of the software but active participants. Most methods advocate including a customer representative in the development team itself.

The second tenet is the **rejection of "Big Upfront Anything"**, a term used derogatorily for standard software engineering techniques involving extensive planning at the beginning of a project; the principal examples are *requirements*, to define the goals of the system, and *design*, to define its architecture. In the agile view:

- Requirements cannot be captured at the beginning of a project, because users do not know what they want. Even if one managed to write a requirements document, it would be useless because requirements will change through the project.
- Building a design upfront is a waste of time because we do not know what will work and what will not.

Instead of a requirements document, agile methods recommend constant interaction with the customer — hence the benefit of a customer representative in the team — to get both insights into the problem and feedback on what has been produced so far. Instead of design, the recommendation is to build the system iteratively, devising at each step the "simplest solution that can possibly work" (an Extreme Programming slogan) for the task at hand; then, if the solution turns out to be imperfect, improving its design through a process known as *refactoring*.

Agile development, as a consequence, is **iterative, time-boxed development**. The agile alternative to a requirements document is, at the beginning of each iteration, a prioritized list of functions from which the team will select for implementation the function that has the highest *Return on Investment* (ROI). In the absence of big upfront tasks, this choice will be made in successive steps, called "sprints" in Scrum, each taking a fixed time — a few weeks — hence "time-boxed". The development thus proceeds by iterative addition of functionality.

By addition, that is, of **limited, negotiated functionality**. The agile literature laments the effort that traditional projects devote to building program features that hardly anyone will use. It advocates limiting features to the most important ones, as measured by their business value: their ROI. The "Lean Software" school draws on comparisons with other industries (notably car manufacturing) to treat unused functionality as the software equivalent of "waste" in an industrial production process, and "waste minimization" as a core concern. "Kanban", influenced by processes developed for Toyota, seeks to minimize "work in progress".

The "negotiation" occurs at the step of choosing the functionality for each iteration. Just as it is impossible, in the agile view, to determine full requirements in advance, it is unrealistic to commit to both *functionality* and *delivery time*. With time-boxed development, any tradeoffs ("do you want it all or do you want it next month?") will tend to be resolved in favor of the second criterion: if not all the functions planned for an iteration can be delivered by the deadline, it is the functionality that goes; the deadline stays. The missed functionality will either be reassigned to a subsequent phase or — if further analysis deems its ROI insufficient — dropped. This process of planning and adjusting requires constant negotiation with the customer.

The final tenet is the **focus on quality**, which in the agile view essentially means continuous testing (rather than other approaches to quality, in particular those based on design techniques, formal programming methodology, or whatever smacks of "Big Upfront"). The agile approach has little patience with what it sees as the lackadaisical attention to quality in traditional development; it especially dislikes the practice of continuing to develop functionality even when the code already developed does not pass all the tests. One of its contributions is to emphasize the role of a project's *regression test suite*: the set of tests that must pass, including all tests that at some point did *not* pass and hence revealed faults that were then fixed. Regression testing has been known and applied for a long time, but agile methods have given this task a central place in the development process.

1.2 PRINCIPLES

The rest of this book considers that the following eight principles (three of them with sub-principles) constitute the core of the agile canon.

Agile principles

Organizational

1 Put the customer at the center.
2 Let the team self-organize.
3 Work at a sustainable pace.
4 Develop minimal software:
  4.1 Produce minimal functionality.
  4.2 Produce only the product requested.
  4.3 Develop only code and tests.
5 Accept change.

Technical

6 Develop iteratively:
   6.1 Produce frequent working iterations.
   6.2 Freeze requirements during iterations.
7 Treat tests as a key resource:
   7.1 Do not start any new development until all tests pass.
   7.2 Test first.
8 Express requirements through scenarios.

These principles follow from the five general "values" of the previous section, turning them into actual prescriptions.

They are not the principles — twelve of them — listed in the Agile Manifesto. Those official principles, discussed in a later chapter, are less appropriate for analysis: they are redundant and combine ideas at different levels, ranging from generous but hardly earth-shattering intentions ("*Build projects around motivated individuals*" — who would disagree?) to specific rules such as releasing software at specific intervals of two weeks to two months, which are practices rather than principles. They also omit key ideas such as the primacy of tests. The eight presented here provide a better overview.

→ *"The official principles", page 50.*

1.2.1 Organizational principles

Five principles guide agile project organization and management.

Agile development is **customer-centric**. The goal of software development is to deliver the best Return On Investment to the customer; as part of the redefinition of roles, customer representatives should be involved throughout the project.

Agile teams are **self-organizing**, deciding on their own tasks. A corollary of this empowerment of the team is, as noted, a severe curtailment of the manager's responsibilities.

Agile projects work at a **sustainable pace** by refusing so-called "death marches", periods of intense pressure forcing a team to work exceptionally hard in preparation for an upcoming deadline. "Sustainability" requires that programmers work reasonable hours, preserving evenings and week-ends. The sociological undercurrent mentioned above — agile methods as empowerment of programmers and consultants against managers — is again apparent here.

Agile development is **minimalistic** in three ways: building only the essential functions (*minimal functionality*); building only what is requested, excluding extra work to prepare for future reuse and extension (*minimal product*); and building only two kinds of software, programs and tests, at the exclusion of anything that will not be delivered to the customer and hence is considered waste (*minimal artifacts*).

Agile development **accepts change**. In software projects, full requirements cannot be determined at the beginning; needs emerge as the project develops, and evolve as customers and others try intermediate releases. Such change is considered a normal part of the development process.

1.2.2 Technical principles

Agile development implies an **iterative development process**, consisting of successive iterations. Each is fairly short — a few weeks — and produces a *working release* of the software, even a very partial one, which customer representatives can try out to provide reactions that will fuel the next iteration. Scrum introduced the important rule that *functionality is frozen during iterations*: if an idea for a new function arises during development, it is postponed to the preparation of the next iteration.

→ *"Freeze requirements during iterations", page 71.*

The **primacy of tests** embodies the approach's focus on quality. This principle has two consequences, both significant enough to be considered sub-principles on their own:

- **No new development may start until all current tests pass**. This rule reflects a strict approach to quality and a refusal to compromise on bug-fixing.
- **Test First**. This principle, introduced in connection with Extreme Programming, prescribes that no code may be written unless there is already a test for it. It makes tests the first part of the replacement for requirements and specifications in agile development. The *test-driven development* practice, introduced in a later section, takes the idea even further.

→ *"Technical practices", page 9.*

The last principle gives us the second part of the replacement for requirements: **use scenarios to define functionality**. A scenario is a description of a particular interaction of a user with the system, for example (if we are building mobile phone software) a phone conversation from the time the caller dials the number to the time the two parties get disconnected. "Scenario" is not a common agile term, but covers variants such as *use cases* and *user stories* which differ by their level of granularity (a use case is a complete interaction, a user story an application of a smaller unit of functionality). Scenarios are obtained from customers and indicate the fundamental properties of the system's functionality as seen from the user perspective. Collecting scenarios, usually in the form of user stories, is the principal agile technique for requirements; it differs from traditional requirements elicitation in two fundamental ways:

- A scenario is just one example; unlike requirements, it cannot lay claim to completeness. A set of scenarios, however large, cannot come even close to achieving this goal, in the same way that no number of tests of a program can replace a specification.

- In agile development, requirements are not collected at the beginning of the project but throughout, as development progresses. Note, however, that this difference is not as absolute as the agile literature suggests when it blasts “waterfall approaches”: while the traditional software engineering view presents requirements as a specific lifecycle step, coming early in the process, it does not rule out — except in the imagination of agile authors — a scheme in which the requirements are constantly updated in the rest of the lifecycle.

Chapter 4 discusses the organizational and technical agile principles in detail.

1.3 Roles

Agile methods define roles for the various actors of a software project.

Key agile roles

1 Team
2 Product owner.
3 Scrum Master.
4 Customer.

The first and most important role is the **team**: a self-organizing group of developers and others (such as customer representatives), responsible for the ongoing assignment of development tasks to individual members.

Scrum

Scrum has gone the furthest among agile methods in defining new roles that take over some of the traditional manager responsibilities. The definition of the properties of the product under development is the responsibility of a **product owner**; it includes the right to change these properties, but not while a sprint (a development iteration) is in progress. For the manager’s job as coach, mentor, guru and method enforcer, Scrum defines a special role of **Scrum Master**, who cannot also be the project owner.

Common to all agile methods is the emphasis on involving **customers**. Defining “customer” as an explicit project role is part of the agile rejection of up-front requirements and general distrust of documents — “*valuing customer interaction over contract negotiation*”, as the Manifesto puts it. Instead of couching the requirements on paper, the project involves customers directly. Extreme Programming, at least in its early versions, prescribed the embedding of “a customer” in the team, as a full-fledged project member; this practice, although simple to state, raises problems that we will analyze. Even when one does not go that far, every agile project reserves an important role for customers.

Chapter 5 discusses these and other agile roles in detail.

1.4 PRACTICES

To achieve the principles presented above, agile methods promote a set of practices. Here are the principal ones, again with more coming up in the chapter on the topic:

→ Chapter 6 discusses agile practices.

Key agile practices

Organizational

1 Daily meeting.
2 Planning game, planning poker.
3 Continuous integration.
4 Retrospective.
5 Shared code ownership.

Technical

6 Test-driven development.
7 Refactoring.
8 Pair programming.
9 Simplest solution that can possibly work.
10 Coding standards.

1.4.1 Organizational practices

Scrum

All agile methods advocate frequent face-to-face contact, but Scrum specifically includes a requirement for a **daily meeting**, held at the beginning of every working day and known as the "daily Scrum". The meeting must be kept short: 15 minutes is the standard. This goal is reachable, with a typical group of a dozen or two people, because the scope of the meeting is strictly limited to every member of the team answering three questions: "What did I do in the previous working day?", "What do I plan to do today?", and "What impediments am I facing?". Anything else, such as *resolving* non-trivial impediments, must occur outside of the meeting. The daily meeting — which is only applicable in its basic form to a team located in a single place — helps teams remain cohesive, know what everyone is doing, and spot problems early.

Any software development project faces the issue of planning, in particular of estimating delivery times and functionality. Agile methods propose the "**planning game**" (Extreme Programming) and the "**planning poker**" (Scrum). Both are group estimation techniques which ask the participants to come with initial estimates independently, then examine each other's estimates and iterate until a consensus is reached.

More convincing is the concept of **continuous integration**. A decade or two ago, it was not uncommon for software projects to split into sub-developments and only try to put them together ("integrate") at intervals of months or more. This is a terrible approach: when attempting integration, projects often discover that the subsystems have made incompatible assumptions and have to undergo substantial rewriting. Modern development practice calls for frequent integration, at intervals not exceeding a few weeks. Agile methods apply this principle too and some of them actually advocate integrating several times a *day*.

Another agile practice is the **retrospective**, in which a team having finished a development iteration takes time off further development to reflect on the experience and the lessons learned, with the goal of improving its development process.

In many groups, the various units of the software are each "owned" by a particular developer, not in any legal sense but in the sense that this person ultimately decides what may and may not change in the unit. This practice is, for example, common at Microsoft. Agile methods instead advocate **shared code ownership**, where all of the team is responsible for all of the code. The goal is to avoid undue dependence on individuals, to emphasize that all team members have a personal stake in the product, and to avoid territorial battles when a change or new development straddles several parts of the system.

1.4.2 Technical practices

Test-driven development turns the "Test first" principle into a specific practice. Applied iteratively, this practice consists of: writing a test corresponding to a new functionality; running the program, which should not pass the test since the functionality is new; fixing the program; running the program again, and continuing to fix it until it does pass the test (and all other tests, to prevent any regression); examining the code and performing refactoring, as discussed next, to make sure the design remains consistent. This sequence of steps, applied from the start (when the program is empty and hence will fail any non-trivial test) and repeated from then on, is the central form of software development in Extreme Programming.

XP

Refactoring is the process of critically examining a design or implementation and applying any transformations that may be needed to improve its consistency. Catalogs of standard refactoring transformations exist; they include such typical examples, in object-oriented programming, as moving a feature of a class (field or method) up or down the inheritance hierarchy, to another class where it fits better conceptually. Refactoring is particularly necessary in connection with test-driven development: a process consisting solely of adding a code element for every new test would yield programs with a messy, ad-hoc structure; refactoring is necessary to maintain a clean design. Just as scenarios and tests are the agile replacement for Big Upfront Requirements, refactoring is the agile answer to Big Upfront Design.

E.g. [Fowler 1999].

Pair programming has been particularly promoted by Extreme Programming. In this practice, code is systematically developed by two people sharing a workstation, one controlling the keyboard and mouse and explaining his thought patterns as he types, the other commenting, criticizing and making new suggestions. The pilot-and-navigator metaphor is often used to explain that process. The goal is to catch possible mistakes at the source: since the "pilot" is forced to explain his thinking aloud, he will often realize right away that something is wrong, and otherwise the navigator will catch it when trying to understand. Extreme Programming presents pair programming as the only mode of development, to be applied systematically and universally. It figures less prominently in other agile methods.

Extreme Programming also popularized the practice of **the simplest solution that can possibly work**. An application of the minimalistic principles described earlier, in particular "produce only the product requested", it shuns any work that is intended to make the solution more extendible or more reusable, as software engineering principles would normally recommend, in particular the principles of object-oriented development. In the agile view such work is illusory anyway, because we may not need reuse, and we do not know ahead of time in which direction the software may have to be extended.

Finally, agile methods promote the use of **coding standards**: defined style rules that a team should apply to all the code it produces.

1.5 ARTIFACTS

The application of agile methods relies on a number of supporting tools; some of them are conceptual, such as the notion of a user story, and others material, such as a story card used to write such a story.

Key agile artifacts

Virtual

1 Use case, user story.
2 Burndown chart.

Material

3 Story card.
4 Story board.
5 Open room.

1.5.1 Virtual artifacts

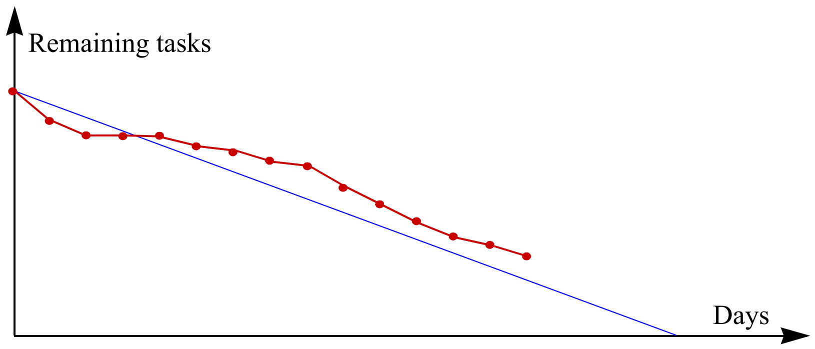
Use cases and particularly **user stories** are scenarios that represent user interactions with a system. Use cases were popularized, pre-agile, by a book due to Ivar Jacobson; user *[Jacobson 1992]* stories have emerged as part of the agile movement. The difference is in granularity; a use case covers a full run through the system, going for example from browsing for a product on an E-commerce site to completing the order; a user story is a smaller unit of functionality expected by users, such as

> *"As a customer, I want to see a list of my recent orders, so that I can track my purchases with a particular company."*

Scrum

The **burndown chart** is a record of a project's *velocity*: how fast it processes — "burns down" — the items in its task list. The chart plots against time (in the example, up to a certain day in a given iteration) the number of unimplemented tasks:

Burndown chart (red)



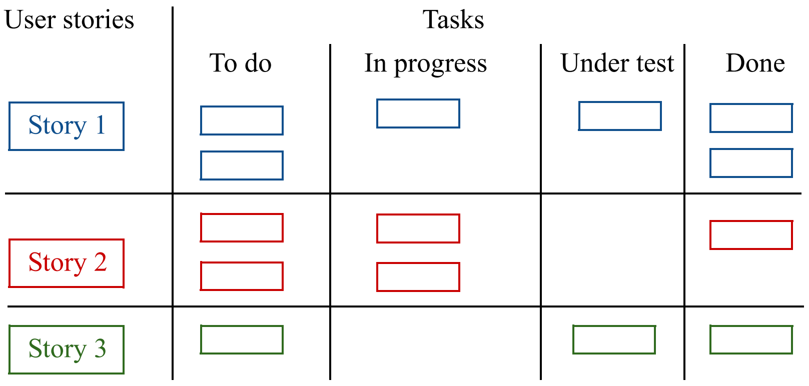
If the task list is fixed for the iteration and no completed task is re-opened, the curve will be non-increasing. The velocity is the number of tasks discharged; the blue line is the constant-velocity line. Where the burndown chart is below that line, the project is progressing faster than expected; above the line, it is progressing more slowly. Maintaining the burndown chart is a way to make the team aware of its progress and alert it when it is not discharging tasks fast enough.

1.5.2 Material artifacts

The remaining artifacts are, in their default form, material objects; for all of them, however, various companies and open-source projects offer software tools providing partial support or full replacement.

The **story card** is a paper-card (agile advocates even prescribe the size: 3 by 5 inches, presumably to be adapted to the local format when working under metric climes) used to write down a user story. Story cards are meant to be pinned to a **story board**, a large board which can host many of them; the team then moves them around the board to group them into categories.

The story board is often refined into a **task board**, which complements the burndown chart to show the progress of the project:



Task Board

Post-it notes on the board represent individual tasks. As work gets done, the team moves them towards the right.

Another recommendation of agile methods addresses the physical layout of the offices in which programmers work: rather than closed offices, it should be set up as an **open room** to favor constant interaction between team members.

1.6 A FIRST ASSESSMENT

We have not gone into enough detail for a full-fledged analysis of the pros and cons of the agile approach (the good, the hype and the ugly); it will come in the final chapter. But we can take a first cut.

Remember that this section only provides a general view, and that the comprehensive assessment of agile methods is the one that comes after the study of agile methods.

→ Chapter 11: The Ugly, the Hype and the Good: an assessment of the agile approach.

Samuel Johnson allegedly responded thus to an aspiring author:

Your work, Sir, is both new and good, but what is new is not good and what is good is not new.

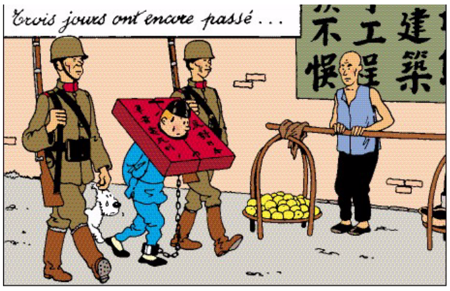
This statement (although apparently apocryphal!) provides us with a useful grid to evaluate agile ideas in four categories, resulting from two possibilities each for newness and goodness. In each category we will consider only a few examples.

See www.samuel-johnson.com/apocryph.html.

1.6.1 Not new and not good

The agile approach to requirements is based on user stories: units of functionality corresponding to interactions of users with the system. User stories, like use cases, are a valuable tool for *validating* requirements, to check that the identified functionality covers the most common scenarios. As a tool for defining requirements, they are inadequate because they only document *examples* of system execution. The task of requirements is to go beyond these individual examples, which can only cover a fraction of the possibilities available, and identify the more general functions of the system. If you forgo this step of generalization and abstraction, you get systems that do a few things — the user stories — and little else.

When using software systems, for example web applications, have you ever felt like Tintin the day he was being marched in a straightjacket? As soon as you dare to depart from the exact scenario that the designers, in their supreme wisdom, have planned for you, nothing works any more. This kind of system is the direct result of requirements based on the sole analysis of use cases or user stories.



© Hergé/Moulinsart 2014.

Good requirements shoot for more abstract specifications, subsuming many different scenarios and supporting the development of flexible, extendible applications.

1.6.2 New and not good

Pair programming was introduced by the XP. To characterize it as "not good" is a bit strong since pair programming can be an effective technique if applied with reason. XP's insistence that it should be the absolute rule, however, makes little sense conceptually, as it neglects the role of programmer personality (some excellent developers like to concentrate alone and will resent having to be paired), and practically, as studies show pair programming to be no superior to other classical techniques such as code reviews.

Code reviews are also known as inspections.

To a certain extent pair programming can be dismissed as folklore, since many projects that try it stop after a while. Worse consequences of agile methods come from the injunction to develop minimal software, stated earlier as principle 4. Its component rules 4.2 (produce only the product requested) and 4.3 (develop only code and tests) may appeal to inexperienced project managers as a way to combat programmer perfectionism and deliver results quickly, focusing on the essential. But from a software engineering perspective they are not good advice, since they discourage efforts that have proved to be among the most fruitful practices of software engineering: generalizing code for ease of extension and reuse, and developing tools to automate repetitive processes. In Lean terminology, the results of such efforts are "waste" since they are not delivered to the customer; in reality, when applied appropriately, they are the key to the continuous improvement of a company's software process and the professionalization of software practice.

← *"Agile principles", page 4.*

Worse yet is the rejection of upfront requirements. The basic observation is correct: requirements will change, and are hard anyway to capture at the beginning. In no way, however, does it imply the dramatic conclusion that upfront requirements are useless! What it does imply is that requirements should be subject to change, like all other artifacts of the software process. This point has been made by much of the software engineering literature and remains as valid as ever. Unfortunately, many projects in recent years have followed the simplistic agile advice of skipping the systematic requirements phase, replacing it by attempts to evolve the system iteratively with the help of occasional customer interactions. The results are often (predictably) disappointing; projects get delayed because requirements end up being collected anyway, but too late in the lifecyle, when some functionality has already been built; some it will have to be discarded.

The agile advice here is irresponsible and serious software projects should ignore it. The sound practice is to start collecting requirements at the beginning, produce a provisional version prior to engaging in design, and treat the requirements as a living product that undergoes constant adaptation throughout the project.

1.6.3 Not new but good

There is a charmingly adolescent quality to the agile literature: I am sooooo unique! Nobody before me understood what life is about! My folks are sooooo, like, 20-th century!

In reality, despite the scathing attacks on traditional software engineering — the irreparable insult, akin to shouting “liberal!” at a Republican candidate, is “waterfall!” — a number of the productive ideas of agile methods have long been advocated in the standard software engineering literature. We will see examples through the rest of the book; here are two.

The first is iterative development. The industry understood in the nineteen-eighties that the old model of diverging for months and then trying to bring all the pieces back together was a recipe for disaster. A 1995 book by Cusumano and Selby — New York Times best-seller, no less — publicized Microsoft’s “daily build”, a practice which as the name indicates requires the project to produce a working version every day. Open-source projects, which have flourished for decades, have a practice of releasing early and often. The advent of the Web intensified this trend: Google tools and many other cloud-based applications undergo frequent updates, often without any officially advertised release process. The agile literature has helped anchor the idea of frequent releases into the mindset of the software industry, but agile methods did not invent it.

[Cusumano 1995]

Another example is the recognition that change plays an important role in software. The better part of the software engineering literature has long emphasized this point. Object technology, which has taken the software world by storm, is successful largely because it supports change better than previous software construction methods. Agile methods may enhance software change through organizational practices, but they make no technical contribution in this area; in fact, as we will see, some of the agile precepts work *against* making software easy to change. The agile approach is not entitled to its blanket contempt of earlier methods of improving extendibility.

→ *“Accept change”, 4.4.5, page 68.*

1.6.4 New and good!

If at this point you feel ready to throw away the agile bath water, extreme and lean babies included, do not remove the tub stopper just yet. You would be missing some surprisingly good stuff.

The first major contribution is **team empowerment**. Giving a central place to the team and insisting that it can handle many traditional management responsibilities is a plus for any software project staffed by competent people.

Some of the management practices of agile methods, which may seem simple-minded at first, can actually make a considerable contribution to project success. One of the most significant is Scrum's **daily meeting**; reinforcing programmer interaction, and requiring everyone to describe every morning what he just did, what he will do next, and what impediments he faces, is a brilliant idea, the kind of egg-of-Columbus insight ("*I could have thought of this myself*" — maybe, but you didn't!) that makes a real difference, at least when it can be applied, that is to say, when the whole team is in one place rather than distributed.

A particularly interesting idea is the **freezing of requirements during iterations**. While demonstrating that — whatever the Agile Manifesto says — change is not *always* welcome even in agile development, this principle brings stability to the software process, without seriously hampering the emergence of change requests: they are not rejected, just delayed, and typically not for long since agile iterations are short.

The **time-boxed iteration** is also a productive practice, particularly through its influence on the planning process, since it discourages unrealistic promises.

On the technical side, a major achievement of agile methods has been to establish the **practical importance of tests** and specifically of the regression test suite. The regression testing idea itself is old, but agile methods taught us that the regression suite is a key asset of the project, that many activities should be organized around it (whether or not the project applies test-driven development), and that it is futile to move on to new functionality as long as important tests do not pass. Here we have the agile school at its best, advocating professionalism and quality.

A similar observation applies to several of the ideas listed earlier as "Good but not new". Even if the agile movement does not deserve the credit for inventing these concepts, which previous authors had energetically advocated, it has succeeded in conveying them effectively to the software industry, a significant achievement in itself. The two principal examples are:

- **Short iterations**. While the more competent companies have relied on iterative development for a long time, it is partly thanks to agile ideas that this practice has become so widely accepted.
- **The central role of code**. Once again this is not new but the agile movement has been instrumental in reminding everyone that our primary product is code, not diagrams or documents.

In emphasizing and popularizing these principles, the agile movement places itself in the best tradition of software engineering — of the very compendium of wisdom, accumulated over several decades, that it so haughtily deprecates. When the dust has settled and the field has matured, this is how we will remember the self-proclaimed agile revolution: as an incremental step, which — aside from indulging in some lunacies that were not destined to last long — improved our understanding of existing concepts and introduced a precious few new insights.