Robert C. Martin: Clean Agile. Back to Basics

Book Summary

Patrick Bucher

2021-08-04

Contents

1 Introduction to Agile

The *Agile Manifesto* originated from the gathering of 17 software experts in early 2001 as a reaction to heavy-weight processes like *Waterfall*. Since then, Agile was adopted widely and has been extended in various ways—unfortunately, not always in the spirit of the original idea.

1.1 History of Agile

The basic idea of Agile—working with small, intermediate goals and measuring the process—might be as old as civilization. Agile practices might also have been used in the early days

of software development. However, the idea of *Scientific Management*, which is based on Taylorism, with its top-down approach and heavy planning, was prevalent in many industries at that time, conflicting with the *Pre-Agile* practice then prevalent in software development.

Scientific Management was suitable for projects with a high cost of change, a well-defined problem definition, and extremely specific goals. Pre-Agile practices, on the other hand, were a good fit for projects with a low cost of change, an only partially defined problem, and goals only being informally specified.

Unfortunately, there was no discussion at that time on which approach was the better fit for software projects. Instead, the Waterfall model—initially used as a straw man to be proven unsuitable by Winston Royce in his 1970 paper *Managing the Development of Large Software Systems*—was widely adopted in the industry. And Waterfall, with its emphasize on analysis, planning, and closely sticking to a plan, was a descendant of Scientific Management, not of Pre-Agile practices.

Waterfall dominated the industry since the 1970s for almost 30 years. Its subsequent phases of analysis, design, and implementation looked promising to developers working in endless "code and fix" cycles, lacking even the discipline of Pre-Agile.

What looked good on paper—and produced promising looking results in the analysis and design phases—often terribly failed in the implementation phase. Those problems, however, have been attributed to bad execution, and the Waterfall approach itself wasn't criticized. Instead, it became so dominant that new developments in the software industry like structured or object-oriented *programming* were soon followed by the disciplines of structured and object-oriented *analysis* and *design*—perfectly fitting the Waterfall mindset.

Some proponents of those ideas, however, started to challenge the Waterfall model in the mid-1990s, for example Grady Booch with his method of object-oriented design (OOD), the *Design Pattern* movement, and the authors of the *Scrum* paper. Kent Beck's *Extreme Programming* (XP) and *Test-Driven Development* (TDD) approaches of the late 1990s clearly moved away from Waterfall towards an Agile approach. Martin Fowler's take on *Refactoring* emphasizing continuous improvement certainly is a bad fit for Waterfall.

1.2 The Agile Manifesto

17 proponents of various agile ideas—Kent Beck, Robert C. Martin, Ward Cunningham (XP), Ken Schwaber, Mike Beedle, Jeff Sutherland (Scrum), Andrew Hunt, David Thomas ("Pragmatic Programmers"), among others—met in Snowbird, Utah in early 2001 to come up with a manifesto capturing the common essence of all those lightweight ideas. After two days, broad consensus was reached:

> We are uncovering better ways of developing software by doing it and helping others do it.
>
> - **Individuals and interactions** over processes and tools

- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

This *Agile Manifesto* was published after the gathering on agilemanifesto.org, where it still can be signed. The 12 Principles were written as a collaborative effort within the two weeks that followed the conference. This document explains and directs the four values stated in the manifesto; it shows, that those values have actual consequences.

1.3 Agile Overview

Many software projects are managed using approaches based on faith and motivational techniques. As a result, such projects are chronically late, despite developers working overtime.

All projects are constrained by a trade-off called the *Iron Cross*: good, fast, cheap, done—pick three! Good project managers understand this trade-off and strive for results that are done good enough within an acceptable time frame and budget, which provide the crucial features.

Agile produces data that helps managers taking good decisions. The *velocity* shows the amount of points a development team finishes within an iteration. A *burn-down chart* shows the points remaining until the next milestone. The latter not necessarily shrinks at the rate of the velocity, because requirements and their estimations can change. Still, the burn-down chart's slope can be used to predict a likely date when the milestone is going to be reached. Agile is a feedback-driven approach. Even though the Agile Manifesto doesn't mention velocity or burn-down charts, collecting such data and taking decisions based on it is crucial. Make that data public, transparent, and obvious.

A project's end date is usually given and not negotiable, often for good business reasons. The requirements, however, often change, because customers only have a rough goal, but don't know the detailed steps how to reach it.

1.4 A Waterfall Project

In the Waterfall days, a project was often split up into three pases of equal length: analysis, design, and implementation. In the analysis phase, requirements are gathered and planning is done. In the design phase, a solution is sketched and the planning is refined. Neither phase has hard and tangible goals; they are done when the end date of the phase is reached.

The implementation phase, however, needs to produce working software—a goal that is hard and tangible, and whose attainment is easy to judge. Schedule slips are only detected in this phase, and stakeholder only become aware of such issues when the project should be done almost finished.

Such projects often end in a *Death March*: a hardly working solution is produced after a lot of overtime work, despite deadlines being moved forward repeatedly. The "solution" for the next

project usually is to do even more analysis and design—more of what didn't work in the first place (*Runaway Process Inflation*).

1.5 The Agile Way

Like Waterfall, an Agile projects starts with analysis—but analysis never ends. The time is divided in iterations or sprints of typically one or two weeks. *Iteration Zero* is used to write the initial stories, to estimate them, to set up the development environment, to draft a tentative design, and to come up with a rough plan. Analysis, design, and implementation take place in every iteration.

After the first iteration is completed, usually fewer stories have been finished than originally estimated. This is not a failure, but provides a first measurement that can be used to adjust the original plan. After a couple of iterations, a realistic average velocity and an estimation of the project's release date can be calcualted. This might be disappointing, but realistic. Hope is replaced by real data early in the process.

Project management dealing with the Iron Cross—good, fast, cheap, done: pick three!—can now do the following adjustments:

- *Schedule*: The end date is usually not negotiable, and if it is, delays usually cost the business significantly.
- *Staff*: *"Adding manpower to a late project makes it later."* (Brooke's Law) If more staff is added to a project, productivity first plummets, and only increases over time. Staff can be added in the long run, if one can afford it.
- *Quality*: Lowering the quality might give the impression of going faster in the short run, but slows down the project in the long run, because more defects are introduced. *"The only way to go fast, is to go well."*
- *Scope*: If there's no other way, stakeholders can often be convinced to limit their demands to features that are absolutely needed.

Reducing the scope is often the only sensible choice. Make sure at the beginning of every sprint to only implement features that are really needed by the stakeholders. You might waste precious time on "nice to have" features otherwise.

1.6 Circle of Life

Extreme Programming (XP), as described in Kent Beck's _Extreme Programming Explained", captures the essence of Agile. The practices of XP are organized in the *Circle of Life*, which consists of three rings.

The outer ring contains the *business-facing* practices, which are quite similar to the Scrum process:

- **Planning Game**: breaking down a project into features, stories, and tasks

- **Small Releases**: delivering small, but regular increments
- **Acceptance Tests**: providing unambiguous completion criteria (definition of "done")
- **Whole Team**: working together in different functions (programmers, testers, management)

The middle ring contains the *team-facing* practices:

- **Sustainable Pace**: making progress while preventing burnout of the developing team
- **Collective Ownership**: sharing knowledge on the project to prevent silos
- **Continuous Integration**: closing the feedback loop frequently and keeping the team's focus
- **Metaphor**: working with a common vocabulary and language

The inner ring contains *technical* practices:

- **Pairing**: sharing knowledge, reviewing, collaborating
- **Simple Design**: preventing wasted efforts
- **Refactoring**: refining and improving all work products continuously
- **Test-Driven Development**: maintaining quality when going quickly

These practices closely match the values of the Agile Manifesto:

- **Individuals and interactions** over processes and tools
  - Whole Team (business-facing)
  - Metaphor (team-facing)
  - Collective Ownership (team-facing)
  - Pairing (technical)
- **Working software** over comprehensive documentation
  - Acceptance Tests (business-facing)
  - Test-Driven Development (technical)
  - Simple Design (technical)
  - Refactoring (technical)
  - Continuous Integration (technical)
- **Customer collaboration** over contract negotiation
  - Planning Game (business-facing)
  - Small Releases (business-facing)
  - Acceptance Tests (business-facing)
  - Metaphor (team)
- **Responding to change** over following a plan
  - Planning Game (business-facing)
  - Small Releases (business-facing)
  - Acceptance Tests (business-facing)

  - Sustainable Pace (team-facing)
  - Refactoring (technical)
  - Test-Driven Development (technical)

To sum up:

> Agile is a small discipline that helps small software teams manage small projects.
> Big projects are made from small projects.

2 The Reasons for Agile

Many developers adopting Agile for the promise of speed and quality end up disappointed as these results do not show up immediately. However, the more important reasons for adopting Agile are *professionalism* and *reasonable customer expectations*.

2.1 Professionalism

In Agile, high commitment to discipline is more important than ceremony. Disciplined, professional behaviour becomes more important as software itself becomes more important. Computers are almost everywhere nowadays, and so is software. Little gets accomplished without software.

Software is written by programmers—and bad software can kill people. Therefore, programmers will be blames as people are getting killed due to erroneous software. The disciplines of Agile development are a first step towards professionalism—which might save people's life in the long run.

2.2 Reasonable Customer Expectations

Managers, customers, and users have reasonable expectations of software and its programmers. The goal of Agile development is to meet t hose expectations, which is not an easy task:

- **Do not ship bad software**: A system should not require from a user to think like a programmer. People spend good money on software—and should get high quality with few defects in return.
- **Continuous technical readiness**: Programmers often fail to ship useful software in time, because they work on too many features at the same time, instead of working only on the most important features first. Agile demands that a system must be technically deployable at the end of every iteration. The code is clean, and the tests all pass. Deploying or not—this is no longer a technical but a business decision.

- **Stable Productivity**: Progress usually is fast at the beginning of a project, but slows down as messy code accumulates. Adding people to a project only helps in the long run—but not at all if those new programmers are trained by those programmers that created the mess in the first place. As this downward spiral continues, progress comes to a halt. Developers demand to start again from scratch. A new code base is built—with the old, messy code base as the sole reliable source for requirements. The old system is maintained and further developed by one half of the team, and the other half lacks behind working on the new system; trying to hit a moving target. Big redesigns often fail, few are ever deployed to customers.
- **Inexpensive Adoptability**: Software ("soft"), as opposed to hardware ("hard") is supposed to be easy to change. Often seen as a nuisance by some developers, changing requirements are the reason why the discipline of software engineering exists. A good software system is easy to change.
- **Continuous Improvement**: Software should become better as time goes. Design, architecture, code structure, efficiency, and throughput of a system should improve and not detoriate over time.
- **Fearless Competence**: Developers are often afraid of modifying bad code, and therefor, bad code isn't improved. ("You touch it, you break it. You break it, you own it.") Test-Driven Development is helpful to overcome this fear by allowing for an automated quality assessment after every change to the code.
- **No QA Findings**: Bugs should not be discovered by QA, but avoided or eliminated by the development team in the first place. If the QA finds bugs, the developers must not only fix those, but also improve their process.
- **Test Automation**: Manual tests are expensive and, thus, will be reduced or skipped if the project's budget is cut. If development is late, QA has too little time to test. Parts of the system remain untested. Machines are better at performing repetetive tasks like manual testing than humans (except for exploratory testing). It is a waste of time and money to let humans perform manual tests; it's also immoral.
- **Cover for each other**: Developers must help each other; they must act as a team. IF somebody fails or gets sick, the other team members must help out. Every developer must ensure that others can cover for him or her by documenting the code, sharing knowledge, and helping others reciprocally.
- **Honest Estimates**: Developers must be honest with their estimates based on their level of knowledge. Under uncertainty, ranges ("5 to 15 days") rather than exact estimates ("10 days") should be provided. Tasks can't always be estimated exactly, but in relation to other tasks ("this takes twice as long as that").
- **Saying "No"**: If no feasible solution for a problem can be found, the developer must say so. This can be inconvenient, but could also save bigger trouble down the road.
- **Continuous Learning**: Developers must keep up with an ever and fast changing industry by learning all the time. It's great if a company provides training, byt the responsibility for learning remains with the developer.
- **Mentoring**: Existing team members must teach new team members. Both sides learn in the process, because teaching is a great way of learning.

2.3 The Bill of Rights

Agile is supposed to heal the divide between business and development. Both sides—customers and developers—have complementary rights.

Customers have the right to …

- … an overall plan: what can be accomplished when at what cost?
- … get the most out of every iteration.
- … see progress in terms of passing tests they define.
- … change their minds and priorities.
- … be informed on schedule and estimate changes.
- … cancel at any time and remain with a working system nonetheless.

Developers have the right to …

- … know what is needed, and what the priorities are.
- … produce high-quality work.
- … ask for and receive help.
- … update their estimates.
- … accept responsibilities rather than having them assigned.

Agile is not a process, it is a set of *rights, expectations, and disciplines* that form the basis for an ethical profession.