11

The Ugly, the Hype and the Good: an assessment of the agile approach

We have now studied the core principles, roles, practices and artifacts that make up the agile canon. It is time to assess the agile contribution: which of the ideas should be kept at bay, which ones do not really matter, and which ones truly help.

For the sections in this chapter, it is appropriate to reverse the order of the book's title (and to use not three but four categories, distinguishing the merely good from the brilliant). The flaws of agile methods are real enough, but the approach would not warrant our attention if it did not also include genuine advances, so it is important to end with these pearls.

11.1 The bad and the ugly

We start with the worst in the agile approach: ideas that damage the software process.

11.1.1 Deprecation of upfront tasks

The prize undisputedly goes to the deprecation of "upfront" activities, in particular upfront requirements and upfront design.

Agile criticism of "Big Upfront Anything" includes some perceptive comments. It is true that one cannot fully comprehend requirements before the development of the system; that requirements will change; that the architecture will have to be improved as implementation proceeds. Those observations express some of the fundamental difficulties of software engineering, and the futility of trying to define everything at the beginning.

There is, however, no argument for shunning the normal engineering practice — the practice, in fact, of any rational endeavor — of studying a problem before attempting to solve it, and of defining the architecture of the solution before embarking on the details. The alternative proposed by agile methods is an ad hoc approach: identify some functionality, build it, assess and correct the result, repeat. It is no substitute for serious requirements and design.

Iterative development is great. Trying out ideas on a small scale before you make final decisions is great. Treating requirements as a living, changeable product is great. Reassessing design decisions on the basis of results is great. Insisting on regular deliveries (once the basic structure is in place) is great. Refactoring is listed at the end of this chapter as one of the significant contributions of agile methods. None of these ideas justifies forsaking the initial tasks of analysis and design.

B. Meyer, *Agile!*, DOI 10.1007/978-3-319-05155-0_11,

In other cases we can see the pros and cons of agile ideas. Here there is no place for equivocating: neglecting these upfront steps, as agile authors advocate, is guaranteed to harm your development.

11.1.2 User stories as a basis for requirements

As previous chapters have discussed on several occasions, user stories play a useful role as ways to check the completeness of requirements, but to use them as the basic form of requirements means forsaking abstraction. In addition, they ignore the critical Jackson-Zave distinction between the machine being built and the domain that constrains it.

← *"The domain and the machine", 3.2.5, page 36.*

The resulting systems are narrowly geared to the specific user stories that have been identified; they often do not apply to other uses; and they are hard to adapt to more general requirements.

User stories are no substitute for a system requirements effort aimed at defining the key abstractions and the associated operations (the *domain model*) and clearly separating machine and domain properties.

← *Pages 120-121.*

11.1.3 Feature-based development and ignorance of dependencies

A core idea of agile methods is that you can treat software development as a sequence of implementations of individual features, selected at each step on the basis of their business value. It would be great if such an approach were applicable, but it exists only in a land of make-believe. Difficult projects do not lend themselves to this scheme: they require foundational work (building core architectural elements, such as a communication or persistence layer) which extend across features; and the features interact through dependencies, causing complexity of the "multiplicative" kind.

← *"Additive and multiplicative complexity: the lasagne and the linguine", page 63.*

11.1.4 Rejection of dependency tracking tools

The potential complexity of feature interactions requires a careful analysis of task dependencies; projects can skip this analysis only at their own risk. The advice to stay away from Gantt charts and dependency-management tools is not only naïve but detrimental. Such tools are not a panacea for project management but have proved their value over several decades of use. They can help agile projects just as well; dogmatic rejection of useful tools is a self-inflicted wound.

11.1.5 Rejection of traditional manager tasks

The self-organizing teams promoted by agile methods, with no manager having the traditional duty of assigning tasks, are the best solution for a few teams, and are inappropriate for many others. The picture of the manager as an incompetent control freak is a caricature. Many software projects have been brought to completion, and many projects on the brink of failure have been rescued, through the talents of a strong manager. Imposing a single management scheme on everyone is arrogant.

> Suggestions that management can exert its influence through "subtle control" make things worse. Developers are entitled to demand that any control to which they are subjected be explicit, not devious.

11.1.6 Rejection of upfront generalization

Agilist rightly note that the primary responsibility of a project is to deliver working software to its customers, and that too much early concern for extendibility (ease of change) and reusability (applicability to future projects) can hinder that goal, especially since it is not always clear initially in what direction the software will be extended and which parts will need reuse. But these observations are not a reason to reject the concept of generalization altogether. We have seen that such an attitude directly contradicts the professed agile principle of "welcoming change". Good software developers do not wait for change to happen: they plan for it by designing flexible architectures and solving more than the problem of the moment.

← *"Accept change", 4.4.5, page 68.*

11.1.7 Embedded customer

The XP idea of a customer representative embedded in the development team does not work well in practice, for reasons explained in an earlier discussion. The Scrum notion of a product owner, however, figures below in the list of excellent ideas.

← *"Onsite customer", 6.5, page 96.*

11.1.8 Coach as a separate role

The Scrum idea of a dedicated Scrum Master is good for Scrum, but not appropriate for most projects. Good development requires not just talkers but doers.

← *"Separating roles", 5.7, page 86.*

11.1.9 Test-driven development

Test-first development, and the requirement of associating a test with every piece of functionality, appear in the lists of good and excellent ideas below. So does refactoring.

Test-driven development is another matter. A software process defined as the repeated execution of the basic steps of TDD — write a test, fix the code to pass the test, refactor if needed — cannot be taken seriously. With such an approach one is limited to tunnel vision, focused on the latest test. An effective process requires a high-level perspective, considering the entire system.

← *"The TDD method of software development", 7.5.1, page 113.*

While test-driven development is extensively discussed in the literature, industry has made its choice: it is not broadly practicing this technique. (On the other hand, many companies have adopted user stories. One may only hope that they will realize that replacing requirements by user stories is the same as replacing specifications by tests.)

11.1.10 Deprecation of documents

Agile criticism of document-heavy processes that produce little real customer benefit is right on target for some segments of the industry — although in some cases, such as mission-critical systems, little can be done about the situation since the documents are legally required by certifying agencies. (And not just out of bureaucratic inertia. Even the most enthusiastic agilist might feel, when flying to the next agile conference, that it was not such a bad idea after all — not total "*waste*"— to assess the plane's software against a whole pile of certification standards.)

Outside of specific industries with high regulatory requirements, a strong case exists for lightening up the document infrastructure. It is true, as agilists emphasize, that "design" in software is not as remote from production (implementation) in other engineering fields. Modern programming languages help, because they make it possible to include some of the traditional design in the code itself. (Some of my own work has addressed this issue.) None of these observations, however, can justify the deprecation of upfront plans and documents. Software engineering is engineering, or should be, and sorely needs the benefits of a careful predictive approach, as well as the supporting documents.

← "Is design separate from implementation?", 3.3.1, page 37.

11.2 THE HYPED

The next category includes ideas that may have value but are unlikely to make a significant difference in matters that count: productivity of the software process and quality of software products. Under this heading we may include:

- **Pair programming**, hyped beyond reason. As a practice to be applied occasionally, pair programming is a useful addition to the programming team's bag of tricks. But there is no credible evidence that it provides major improvements to the programming process or that it is better than classical techniques such as code reviews, and no reason to impose it as the sole mode of development.
- **Open-space working arrangements**. There is no single formula for the layout of a working environment. What we do know is that it is essential to provide simple, obvious opportunities for informal communication. Beyond that, many office setups are possible which will not endanger a team's success. (A related point appears, however, under the "good" ideas of the next section: avoiding distributed development.)
- **Self-organizing teams**. A few teams are competent and experienced enough to manage themselves, like a conductor-less orchestra. Most are not. Each situation calls for its own organizational solutions and there is no reason to impose a single scheme on the entire industry.
- **Working at a sustainable pace**. All great advice; death marches are not a good management practice. But advice can only be wishful here; these matters are determined by economic and organizational pressures more than by good intentions. They are not specific to the programming world: like a company that is responding to a Request For Proposals, a researcher who is facing a conference submission deadline will work through the night to meet it. The most software methodologists can do is to argue that such practices should remain the exception.
- **Producing minimal functionality**. It is always a good habit to question whether proposed features are really needed. But usually they get introduced for a reason: some important customer wants them. It is easy to rail against bloat or heap scorn on monster software (Microsoft Word and Adobe Acrobat are common targets), but try to remove any functionality and brace for the screams of the outraged users.

- **Planning game, planning poker.** These are interesting techniques to help estimate in advance the cost and time of development activities, but they cannot be a substitute for more scientific approaches. In particular, they are open to the danger of intimidation by the crowd; the voice of the expert risks being smothered by the chorus of novices.
- **Members and observers.** In project meetings, the views of the people most seriously involved matter most. This trivial observation does not deserve the amount of attention that the agile canon devotes to the distinction between "pigs" and "chickens".
- **Collective code ownership.** The policy governing who is permitted to change various parts of the code is a delicate decision for each project; it depends on the nature of the team and many other considerations. It is pointless to prescribe a universal solution.
- **Cross-functional teams.** It is a good idea to encourage developers to gain broad competence and to avoid dividing the projects into narrow kingdoms of expertise each under the control of one person. Beyond this general advice, there is little a method can change here to the obvious observation that special areas require special skills. If one of your developers is a database expert and another is a concurrency expert, you will not ask the first, if you have a choice, to resolve a tricky deadlock issue, or the second to optimize queries. This observation is another reason why the agile scheduling policy of picking the highest-business-value task in the pipeline is simplistic and potentially harmful.

11.3 THE GOOD

Promoting **refactoring** is an important contribution of the agile approach, particularly of XP. Good programmers have always known that it is not sufficient to get something that works, but that they should take a second look at the design and improve it if needed. Refactoring has given a name to this activity, made it respectable, and provided a catalog of fundamental refactoring patterns. As a substitute for upfront design it is terrible advice, belonging to the "ugly" part of agile. But as a practice that accompanies careful initial design it is of benefit to all software development.

Short daily meetings focused on simple verbal reports to progress — the "three questions" — are an excellent idea. It need not be practiced in a dogmatic way, since distributed projects and companies with flexible work schedules must adapt the basic scheme, but is one of the practices that undeniably help software development, and deserves to be adopted even more widely than it already is.

Agile methods rightly insist on the importance of **team communication** ("osmotic" in Crystal terminology) to the success of projects. One of the consequences is to recommend co-located projects, whenever possible, over distributed development

The practice of identifying and removing **impediments**, in particular as a focus of progress meetings, is a powerful agile insight.

In a similar vein, Lean's identification of sources of **waste** in software development and insistence on removing them provides an excellent discipline for software projects.

11.4 THE BRILLIANT

Fortunately, in our review of agile ideas we have encountered a number of effective and truly inspiring principles and practices.

Short iterations are perhaps the most visible influence of agile ideas, an influence that has already spread throughout the industry. Few competent teams today satisfy themselves with six-month objectives. The industry has understood that constant feedback is essential, with a checkpoint every few weeks.

The related practice of **continuous integration** and the associated **regression test suite** artifact, while not agile inventions, have been popularized by XP and are major factors in the success of modern projects. The industry, or at least every competently managed project, has turned away from older "big bang" practices, and will never go back.

The **closed-window rule**, which prohibits everyone regardless of status from adding functionality during an iteration, is one of the most insightful and effective agile ideas.

← *"The closed-window rule", page 90.*

Time-boxing every iteration — not accepting any delays, even if some functionality has not been implemented — is an excellent discipline, forcing team members and customer representatives to plan carefully and realistically, and bringing stability to the project. (We have seen that it should only apply to iterations, not to an entire project, for which the customer dictates delivery dates.)

← *"The either-what-or-when fallacy", page 146.*

Scrum introduced the beneficial notion of a clearly defined **product owner** who represents the goals of the customer organization and has decision power over what goes into the product and what does not.

The emphasis on **delivering working software** is another important contribution. We have seen that it can be detrimental if understood as excluding requirements, infrastructure and other upfront work. But once a project has established a sound basis, the requirement to maintain a running version imposes a productive discipline on the team.

← *"Dual Development", page 74.*

The notion of **velocity** and the associated artifact of **task boards** to provide visible, constantly updated evidence of progress or lack thereof are practical, directly useful techniques that can help every project.

Associating a test with every piece of functionality is a fundamental rule which contributes significantly to the solidity of a software project and of the resulting product.

The ideas listed as good or brilliant are relatively few, but they are both important and beneficial; they deserve careful study and immediate application. They justify the journey, arduous at times, that we took through the land of agile methods. Once disentangled from the questionable part of the agile credo, they will leave a durable mark on the practice of software engineering, and find their place, along with earlier ideas such as structured programming, formal methods, object-oriented software construction and design patterns, in the history of major advances in the field.