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.
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 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.

### 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.

### 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.