A Quiet Revolution in Requirements Management

John Favaro

Computer Programming, Italian Edition, May 2003

Exactly 15 years ago, an article on werewolves set the stage for a revolution in requirements management. But nobody noticed.

Fred Brooks, already famous as the author of the *Mythical Man-Month* [Brooks 1987], wrote an article in the American journal *Computer* that destroyed the hope that the software crisis could be resolved once and for all time. Comparing the fearful software failures that inhabit our profession to the fearful werewolves that inhabit our imaginations, he argued that, unlike the mythical silver bullet that can slay the werewolf, there is no silver bullet that can slay the software monster.

The reasons he gave were as devastating as they were compelling: the very *essence* of software is its complexity; its changeability; its invisibility. No magic tools would ever be invented that could change its essential nature – and therefore, software development would always be difficult.

The only solution, he argued convincingly, was to *avoid* software development – that is, to search for ways to develop *less* software. He then made two concrete suggestions to address that issue, many of which have become important paradigms today. He suggested assembling software from reusable parts – which became the component-oriented development so widely practiced today. And he suggested the use of "power tools" such as spreadsheets – which became the powerful application generators of today, ranging from environments such as Visual Basic to entire E-Commerce application frameworks. Each of these approaches contributed to reducing the volume of new software that had to be written for a new application – and thus a direct attack on the essential complexity of the software werewolf.

Everybody nodded his head in agreement after reading Brooks's article, and it became a classic in the software engineering literature. But at the time, nobody noticed that a *third* approach to slaying the software werewolf was lurking inside that article. It was lurking inside a seemingly innocuous statement:

> Much of present-day software-acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong. I would go a step further and assert that it is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.

John Favaro
Via Gamerra, 21
56123 Pisa - Italy

Tel +39 050 55 60 74
Fax +39 050 6143 1145
john@favaro.net

www.johnfavaro.com

Everybody understood at the time that Brooks was talking about the need for prototyping in order to understand the requirements of a system. But nobody (perhaps not even Brooks) at the time realized another implication of his statement: not only can we not know beforehand the *exact* requirements for a system; we cannot even know *how many* requirements there are. In particular: *maybe there are fewer of them*. And if there are fewer requirements, then we have to write less software – the third attack on the software werewolf.

In spite of development techniques inspired by Brooks such as component-based development and application generators, the software systems continued to become larger and larger. But as the 1990s progressed, suspicions were aroused. Statistics were published that confirmed those suspicions: much of the software was not being used. In their paranoia to find every possible requirement, analysts were over-specifying software systems.

A couple of years ago the Standish Group, a prominent American market research organization, published some damning statistics and reported them to an astonished audience at the XP2001 conference in Villasimius, Sardinia. In a study they had found that of all the features in software systems today

- 7% are always used
- 13% are often used
- 16% are sometimes used
- 19% are rarely used
- 45% are *never* used

Another study by DuPont found that only 25% of a system's features were really needed. The conclusion was inescapable: most software systems are too large because they implement too many requirements – too many *useless* requirements.

So began quietly a revolution in requirements management: rather than trying to discover the *maximum* set of requirements, the goal became to discover the *minimum* set of requirements. “Most software systems should be only about 25% of their actual size,” wrote Martin Fowler. “Inside every big system, there is a small system trying to get out,” noted Chet Hendrickson.

It was not easy to overcome the prevailing way of thinking. Nervous managers continued to specify system requirements in the old way: careful, conservative, making sure to “find every possible requirement” before starting development. They believed it was their only weapon to make sure that they received full value for their money. In fact, it *was* their only weapon, according to the traditional “waterfall” model of development: once the requirements phase was finished, there were no more second chances with the waterfall model.

But gradually another model of development began to be used, an *iterative* model. No longer did all requirements have to be specified and implemented beforehand. A few – perhaps the most important or the riskiest – could be specified and implemented in a so-called “iteration,” followed by a checkpoint. This checkpoint provided that second chance that managers needed – a second chance to review the requirements, to discover new ones, to discard others. Both managers and developers had the opportunity to learn from the experience of previous iterations to understand better not only the required features of the system, but also the relative importance of each

feature. As their understanding improved, they were able to rank the requirements in order of importance, making sure that the most important ones were implemented in early iterations, saving the less important ones for later iterations.

The result of this ranking was that the most important requirements of the system were implemented early, while the less important requirements were deferred to later iterations. Sometimes, when the nature of a requirement remained unclear, it was simply deferred to a later iteration in the hope that experience with the emerging system would clarify its nature. Often, a surprising thing happened: it was discovered that the requirement was not really necessary at all, and it was dropped. Experience gained with early versions of the emerging system showed both managers and developers that in fact often only the high-priority requirements were needed to construct a useful system – in other words, they had discovered the minimum set of requirements for their system.

Gradually, the relationship between managers and developers began to change. Rather than antagonists in a game of requirements specification and implementation, they became collaborators in requirements *management*, with the common goal of utilizing the available time and resources in the best possible way. Managers could be confident that they would see their most important requirements implemented quickly and have feedback on the results, including that all-important "second chance" to make changes if necessary. Developers could be confident that they would not have to waste valuable effort in implementing and testing useless features. Rather, they could concentrate on providing high quality on those features that were truly needed.

The quiet revolution in requirements management has begun to affect many areas of software project management. One important example is the structure of contracts. With the traditional way of project management, where there was "no second chance" to modify the set of requirements after development began, rigid contracts with fixed prices were the norm. This was perfectly understandable: it was a way for the manager to protect himself against the many surprises that would inevitably be revealed during development. But with the increased flexibility offered by the iterative paradigm, it became possible to create contracts with more flexibility – contracts with *optional features*. The nature of such software development contracts is more like a modern professional services contract: rather than specify an overall fixed set of features one time for the entire project, the contract specifies a fixed price and time period for a certain subset of features. The developers guarantee that the agreed features will be implemented in their entirety – that is, in a working system, not a prototype. The contract itself is periodically renegotiated over the course of the entire project for different sets of features.

Does this sound like a disadvantage for managers? On the contrary: as the project progresses, the ability of managers and developers to estimate the true cost of implementing features improves, increasing financial control over the project. Furthermore, the manager will have the option to stop the project when he determines that a sufficient set of features has been implemented. Often this will allow him to spend *less* money than he would have had to spend with a fixed price agreement.

The quiet revolution in requirements management has taken place in what has become known as the community of agile methods. The most famous of these methods is known as Extreme Programming [Beck 1999].

But this revolution is not destined to remain quiet. Before long, it will be heard around the world.

REFERENCES

[Beck 1999] Beck, K., *Extreme Programming Explained*, Addison-Wesley, 1999

[Brooks 1987] Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer Magazine*, April 1987.

[Brooks 1967] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.