The Empowered Programmer

John Favaro

Computer Programming, Italian Edition, June 2003

In the old days, it was different.

In the old days, the programmer was important.

In 1970, the first class graduated from Yale University in the United States with a degree in the newly established discipline of "computer science." Shortly afterwards, the first degrees in "informatics" were awarded in Europe. The graduates with these degrees were recognized as highly trained professionals with the skills and training necessary to work in a complex and demanding field. They had studied the theory and implementation of complex mathematical algorithms. They had training in the techniques of design, analysis, and abstraction. And they were programmers.

The respect for the programmer reached an apex in 1972, when Edgser Dijkstra received the computing profession's highest honor, the Turing Award. He entitled his acceptance speech simply "The Humble Programmer," [Dijkstra 1972] honoring those who worked with those marvelous machines whose influence as tools for mathematical calculations had already been profound, but

> " … will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind."

With this statement, Dijkstra acknowledged the central importance of the programmer in confronting one of the greatest technical challenges of our age.

But the winds of change were already blowing. The first step that took the programmer's role away from central importance came as the concept of the software *lifecycle* was developed. The first, and still best known model of the software lifecycle, was presented in 1970 and became known as the waterfall model [Royce 1970]. This model did two things:

- It defined a software development *process*, broken into a sequential series of phases, in which programming (which became known as the "coding phase") took place only after two *earlier* phases of requirements and design.

- It also made an indirect statement about the relative importance of each of the phases, immortalized in a diagram presented to software engineering classes around the world (Figure 1).

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

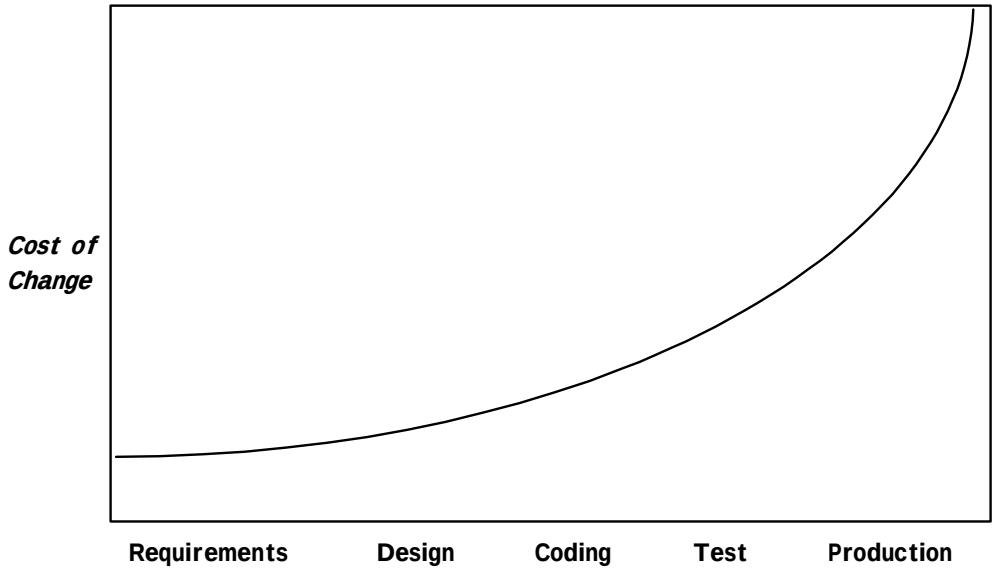

Figure 1: Exponentially Rising Cost of Change in Waterfall Lifecycle [Beck 1999]

This diagram embodied the assertion that the cost of changing software grew exponentially over the life cycle. In particular, it was orders of magnitude more expensive to make a change during programming (the "coding phase") than during the earlier phases of design and requirements analysis.

The effect of this diagram was as might be expected: suddenly those earlier phases took over the central importance. Suddenly, the Requirements Analyst became a Very Important Person in the software process, and an entire discipline of requirements analysis grew up around him. The Designer (or, even more eloquently the "Architect") became another Very Important Person, and an entire discipline of software architecture grew up around him.

And conversely, the programmer was suddenly much less important. Now that all the real effort was concentrated in the earlier requirements and design phases, the programmer became a mere assistant, transforming the results of the requirements analyst and architect in a mechanical fashion into code. No special skill was needed, because after all, the requirements analyst and the design architect had already done the important work.

The spread of the personal computer only added to the perception of the programmer as a low-skilled worker, as the computer became perceived in the public mind as a simple machine, because it had become simple to use. After all, if it was simple to *use*, then it must be simple to *program*, went the reasoning.

By the 1980s, a programmer in Germany was no longer classified as a professional (like lawyers and engineers) for tax purposes, but as a kind of "merchant" (like a shoe salesman) and was obliged to pay the same extra tax (the *Gewerbesteuer*) that a merchant is obliged to pay. (This is also true in Italy today.)

The final blow to the programmer's prestige came in the form of the software process management movement that began in the 1980s. The focus shifted dramatically away from the individual and toward the organization. Suddenly entire organizations were

being judged on their overall maturity against such standards as the CMM model of the Software Engineering Institute in the United States [Paulk 1991]. Large software development infrastructures were created, including separate quality assurance divisions and special units for monitoring, measuring, and documenting every aspect of the organization's process. In this environment, the anonymity of the individual programmer was now complete. The heavy, organizational approach to software development persisted well into the 1990s. Interestingly, the success rates of software projects did not seem to improve with all of these efforts. The same rates of project failure and customer dissatisfaction seemed to prevail. [DeMarco and Boehm, 2002]

But around the mid-1990s, the winds of change began to blow again. In spite of the emphasis on managerial issues over the previous decade, a great deal of progress had also been made in technical areas. In particular, the rise of the object-oriented approach had created a kind of renaissance in software construction techniques. Innovative and highly skilled researchers and programmers had introduced new paradigms such as *architectural styles* and *design patterns.*

Most importantly, an alternative approach to life cycle management emerged. In a desire to distance itself from the rigid approaches that had developed in the years before, the new approach began calling itself "agile."

The agile movement embraced many ideas that were the antithesis of what had been promoted in the heavier, process-oriented days. The phases were no longer clear-cut and sequential, but interleaved. Systems were smaller; teams were smaller; documentation was reduced. And most of all, the individual programmer came back onto center stage, as DeMarco noted:

> Part of our 20-year-long obsession with process is that we have tried to invest at the organizational level instead of the individual level. We've spent big bucks teaching the *organization* how to build systems. If agile means anything to me, it means investing heavily in *individual* skill-building rather than organizational rule sets. Most of the companies I visit that don't have enough "superbly trained people" today don't have them because they haven't even tried. [DeMarco and Boehm, 2002]

The agile movement featured small teams of programmers who exploited the technical advances of the past decade (many of which they were the authors), to create a highly sophisticated technical environment in which working code was released continuously. To achieve this they introduced a new set of techniques and practices, including:

- **Test-first programming** – a new emphasis on the total integration of coding and testing. No longer was there a testing "phase" subsequent to programming. On the contrary, tests were rigorously created before code was written.

- **Fully automated tests** – tests were run continuously and automatically, using new tools (such as JUnit) that they created for this purpose.

- **Continuous refactoring** – as code was created, it was continuously modified through a sophisticated, ever-growing set of manipulations that improved its quality and robustness.

- **Continuous integration** – the software was built and integrated daily, so that there was always a working version of the system, from its earliest to its final releases.

- **Emergent Design** – rather than attempting to create a complete up-front design (from the inevitably poorly defined initial requirements), the design of the system was grown along with the system itself, reacting to feedback and improved information from implementation cycles. Such a practice, inconceivable only a decade earlier, had become possible with the sophisticated techniques of design patterns, architectural styles, and the like.

All of these practices led to an impressive claim by agile movement: the claim that the cost of change curve had been modified (Figure 2).

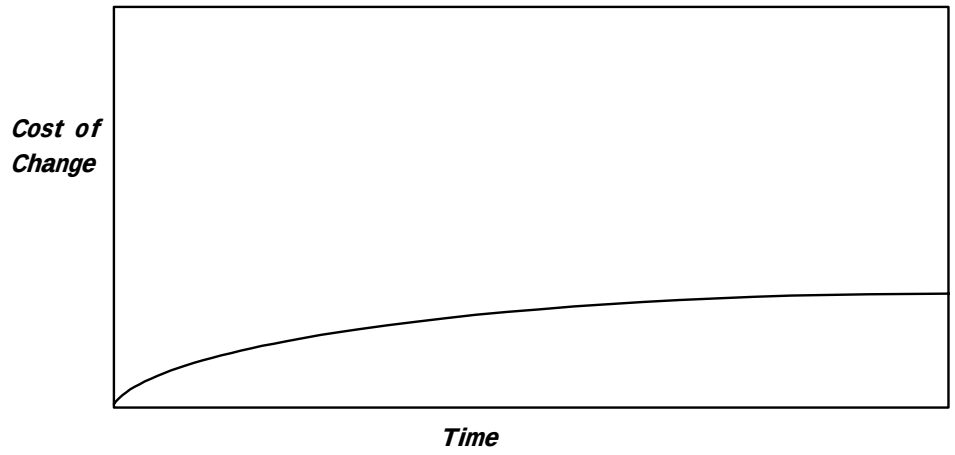

Figure 2: Flattened Cost of Change [Beck 1999]

No longer was it true that the cost of changing a program needed to rise exponentially over time. It was now *flattened*. As Beck said [Beck 1999]:

> The software development community has spent enormous resources in recent decades trying to reduce the cost of change—better languages, better database technology, better programming practices, better environments and tools, new notations … It is *the* technical premise of [the agile movement].

Their great technical skills allowed programmers in the agile movement to make promises to the customer such as the following [Jeffries 2001]:

> Customer Development Right #1: […] You may cancel at any time and be left with a useful working system reflecting investment to date.

Never would such a promise have been possible with the heavy, organizationally focused processes. A customer wishing to cancel a contract risked finding himself with only a stack of requirements and design documents, without a single line of code written.

In the agile movement, the motto is, “The code contains all the answers.” Although it is a simplification, this is a reflection of the return to the most fundamental facts of software development: the ultimate purpose of a software development is the

production of a working system. The working system embodies the results of all requirements, design, and coding activities and is the only totally reliable source of information about the software. This is not to say that requirements and design do not have their place in software development (indeed, they have a very important role to play in the agile movement). But it does say that software engineering has finally come full circle, back to its roots, where the newly empowered programmer has returned to his rightful, central place.

REFERENCES

[Beck 1999] Beck, K., *Extreme Programming Explained*, Addison-Wesley Longman, 2000

[Jeffries 2001] Jeffries, R., A. Anderson, and Hendrickson, C., *Extreme Programming Installed*, Addison-Wesley Longman, 2001

[DeMarco and Boehm, 2002] DeMarco, T. and B. Boehm, "The Agile Methods Fray," *IEEE Computer*, June 2002.

[Dijkstra 1972] Dijkstra, E. W., "The Humble Programmer," *Communications of the ACM 15* (1972), 10: 859-866.

[Royce 1970] Royce, Winston W., *Managing the Development of Large Software Systems*, IEEE Press, 1970

[Paulk 1991] Paulk, M., Curtis, B., and Chrissis, M., "Capability Maturity Model for Software." Technical Report CMU/SEI-91-TR-24, Software Engineering Institute, Pittsburgh, PA, 1991.