Software Evolution as the Key to Productivity*

Oscar Nierstrasz

University of Bern, Switzerland
oscar.nierstrasz@acm.org
www.iam.unibe.ch/~oscar

Abstract. Despite the existence of a seemingly continuous stream of new technologies and methods, software productivity remains universally unimpressive. We argue that, as long as industry remains focused on short-term goals, and maintains a technology-centric view of software development, no progress will be made. A clear symptom of this problem is the fact that the metaphors we apply to software development are largely obsolete. Instead of thinking about software as we do about bridges, buildings or hardware components, we should encourage a view of software as a living and evolving entity that is developed and maintained by *people*. We begin with some assertions that are intended as food for thought. We continue by reviewing what we consider to be some of the key difficulties with software development today. We conclude with a few recommendations for research into software practices that take evolution into account.

1 Food for Thought ...

- Software "engineering" is only a metaphor.
- Software "architecture" is only a metaphor.
- Software "components" are only a metaphor.
- The most cost-intensive phase of any successful software project is "maintenance".
- What is termed "maintenance" consists in practice of continuous development and software evolution.
- Poor software quality is the greatest impediment to software evolution.
- Formal methods have a strictly limited impact on software quality in general.
- Constant refactoring is a prerequisite for effective software evolution.
- Aggressive testing is a prerequisite for refactoring.
- Standardized architectures and interfaces are a prerequisite to effective component reuse.
- A good software architect must be a good abstract thinker.

* In *Radical Innovations of Software and Systems Engineering in the Future*, A. Knapp, M. Wirsing and S. Balsamo (Eds.), LNCS, vol. 2941, Springer-Verlag, 2004, pp. 274-282.

- Comparatively few programmers are good abstract thinkers.
- Software reuse can only have a limited impact on a small portion of the software lifecycle.
- Software reuse does not come for free.
- Object-oriented programming offers a means to model complex domains.
- It is hard to develop models that can be easily adapted over time.
- Objects are not components.
- Object-oriented designs expose a class hierarchy, not a run-time architecture.
- Raising the level of abstraction is the only way to produce more in the same amount of time.
- Scripting languages are the most effective rapid application development tools known today.
- Scripting languages are good for "programming in the small", not "programming in the large".
- Yesterday's large programs are today's small programs.
- Different people are motivated by very different things.
- Hardly anybody is motivated by money above all else.
- Technologists are often motivated by technology.
- Technology has only minimal impact on the success of a software project.
- Effective communication is the single greatest factor contributing to the success of software projects.
- Short and frequent iterations are a prerequisite for effective communication.
- You can lead a horse to water but you can't make it drink.

2 Software Evolution and Productivity

Despite the appearance of innumerable new software development techniques, tools and methods over the past couple of decades, there is a general perception that software productivity has not actually improved as a result of these innovations.

Why is this?

First of all, let us consider what we mean by *software productivity*. From an Engineering perspective, we usually consider that $productivity = units \div effort$ [10]. Units of software can be notoriously difficult to measure, but, without belaboring the point, let us argue that

$$\text{productivity} = \frac{\text{functionality} \oplus \text{quality}}{\text{effort}}$$

where $\oplus$ is some kind of "addition" over functionality and quality. The desired functionality and quality are specified as Software Requirements, and the product is manifested in terms of Software Artefacts. Requirements being the input to our development process, productivity should just depend on the quality of the Requirements specifications, the methods and tools we use, and our own programming skill.

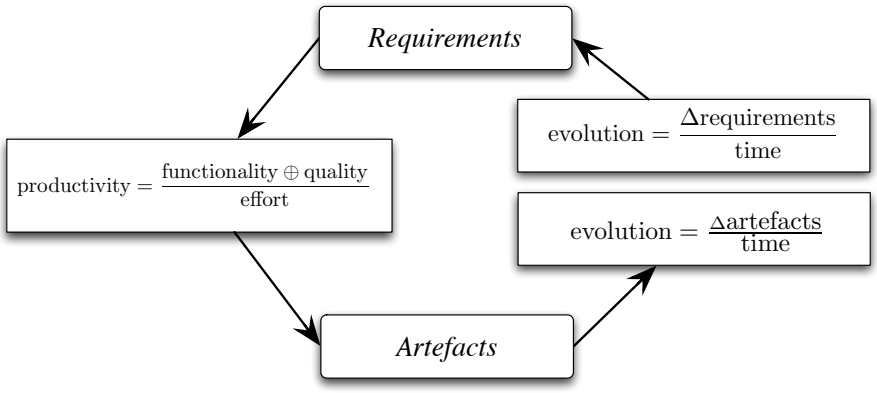

Fig. 1. Software evolution and productivity

At this point, however, we must not forget that "Software Engineering" is just a metaphor, in particular, one that says that "software is like a physical product". Object-Oriented Programming, Component-Based Software Development (CBSD), and Software Architecture are other popular metaphors that suggest different ways of thinking about software. At the same time, however, metaphors can be dangerous if one forgets that they are *just* metaphors.

Software Engineering itself may well be one of these dangerous metaphors. There are many ways in which software is *not* like a typical Engineering product. For example, software is not subject to physical constraints. Many software application domains are also highly unstable due to the high rate of innovation. The most striking of these differences, however, are perhaps those highlighted by the following laws of software evolution [13]:

- *The Law of Continuing Change:* A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- *The Law of Increasing Complexity:* As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

This tells us that the Requirements are *not* the only input to our development process, but that legacy Artefacts also constitute an important input. Furthermore, as the Artefacts evolve, Requirements will also evolve in a never-ending cycle (see figure 1), and, as complexity increases, quality will degrade and productivity will decrease.

Oddly, most software development methods seem to assume that a new application is being developed rather than that some existing software base is being extended or modified, whereas in practice the latter is almost always the case. Most of the real problems with software development, in fact, have to do with

software evolution: How can we construct software systems that can be gracefully adapted to changing requirements over time? If we consider most of the other perceived problems (poor quality, lack of reuse, and so on), they are largely subsumed by the problem of software evolution.

If we consider the typical lifecycle of a *successful* software product, we quickly see that most of the costs are associated with its life *after* deployment [3, 14]. Furthermore, what is often misnamed "maintenance" actually consists mainly of addition of new functionality, *i.e.*, continuous development, or simply *software evolution* [3, 14]. This suggests that the impact of evolution of productivity cannot be merely incidental or occasional, but *fundamental* to software productivity. Attempts to improve productivity that do not take this into account are therefore doomed to failure!

If software evolution is really the key issue in developing successful software systems, why is it almost universally ignored in proposals of new methods and techniques for software development?

2.1 What's Wrong with OOP?

In the 1980s, object-oriented programming was widely considered to offer solutions to a wide range of software woes. In the 1990s, by the time that OOP became mainstream, it was clear that it was not a silver bullet. Worse, the learning curve with OOP is much steeper than with conventional procedural programming [15], reuse with OOP is much harder to achieve than with simple libraries of procedures, and all the problems with legacy applications recur with OOP except that they have an object-oriented flavour [9].

Implicit Architecture. Whereas procedural source code reflects procedural design quite well, object-oriented code does not normally reflect the run-time OO architecture. That means that it is typically much harder to read and understand a well-designed object-oriented program than it is to understand a well-designed procedural one because the source code exposes a class hierarchy, not the set of objects that provide the run-time behaviour. In order to understand the run-time behaviour, one needs to know which objects will be instantiated and how they relate to one another. Due to polymorphism, however, this information can be very hard to extract. This steep learning curve can make it hard to understand and evolve an object-oriented application.

Implicit Reuse Contracts. Although OOP offers very expressive mechanisms for software reuse, these mechanisms can be hard to understand and use correctly. An object-oriented *framework* consists of a class hierarchy that must be subclassed and extended to instantiate an application. Frameworks make use of many common idioms and design patterns, and the rules for correctly subclassing the framework classes typically entail *reuse contracts* [20] that may only be implicit in the code. Not only are OO frameworks hard to develop, but they entail a steep learning curve to use them.

Missing Sockets and Plugs. OO frameworks tend to be based on "white box reuse", *i.e.*, requiring knowledge of implementation details. Black box (component-based) reuse is more attractive since it makes the reuse contracts explicit as plugs (plug 'n play). Current OO analysis and design methods, however, encourage designers and developers to model domain objects in a way that leads to rich interfaces that are *not* plug compatible. This means that OOA/OOD as it is practiced today conflicts with the principles of CBSD.

Refactoring. Although it is well-established that the quality of OO software depends on a culture of refactoring and reengineering [9], it is still hardly standard practice.

Although OOP has demonstrated some benefits for productivity through reuse of libraries and frameworks, object-oriented methods alone do not especially address software evolution, so their impact on productivity is *necessarily* limited.

2.2 Are Components the Solution?

Although "object" is not yet a four-letter word, "component" seems to be the current buzz. But components, like objects, have also been around since the sixties [16].

Szyperski defines a software component as "a unit of independent deployment, a unit of third-party composition, [that] has no persistent state" [21]. Clearly this definition can fit many different kinds of software entity. Whether we call it a "component" or a library or a framework or an application generator or a 4G environment or a programming language does not really matter. In each case, the key idea is to factor out everything that is known and stable and put it into a box, thereby *raising the level of abstraction*. In each case, components may be composed by means of a graphical or textual specification that plugs together compatible parts.

The idea of building applications by "simply plugging together components" is very attractive, and certainly has some merit. But what is often forgotten is that components do not come from thin air. The cost of developing "reusable" components is significantly higher than that of building isolated applications [3, 15]. Repositories of existing software elements help no more than do catalogues of the contents of junkyards (unless you are a junkyard artist). CBSD can only be successful when the process of developing components proceeds in parallel with the process of developing applications based on components, *and* the cost of developing components is amortized by the gains in productivity in developing and maintaining the resulting applications [11, 17].

So what is the added value of CBSD? Certainly more rapid development is a gain in productivity since components will only have to be developed once. But the cost of developing and maintaining components can be higher than that of developing applications that are not component-based. Certainly higher software quality can be achieved by reusing components that have been thoroughly tested across a range of applications. But this presupposes a significant investment in

ensuring the quality of the components, and presupposes a robust infrastructure for composing them. Only such a compositional infrastructure can enable "independent deployment".

If the biggest cost in the lifecycle of applications is evolution, we should ask ourselves if CBSD can help reduce maintenance costs. A component-based application clearly separates what is stable from what is not. Increased productivity during software evolution means that new functionality can be more easily integrated, and existing functionality can be more easily adapted. However this is not achieved by components alone. We know and understand components, but what is hard to manage is *flexibility*.

So, although CBSD addresses software evolution to a greater degree than does OOP, mainly by facilitating certain kinds of change, it fails to address the hardest problems. The metaphor of "component-based software development" puts the word "component" in the pole position, but this is misleading. The real added value comes from how components are *composed*, so perhaps we should start to think more about *composition-based software development*.

2.3 What about Formal Methods? Testing? ...

Software quality pays off during *deployment* and *evolution*. Correctness, reliability, efficiency, usability, maintainability, portability, and other software qualities have an impact either on the cost of deployment or the cost of development and evolution of a piece of software. Costs, on the other hand, are entailed while *achieving* and in *maintaining* that quality. When is it worth paying for that quality? Whenever a piece of software is expected to live beyond an iteration or two of its lifecycle, the investment in its quality can be amortized over its future lifetime.

Formal methods clearly have their place in software development. However, many critical aspects of software quality are inherently impossible to formalize (for example, whether the user requirements have been adequately captured)[4]. Aside from certain well-understood domains, the cost of formally specifying requirements is not only exorbitant, but the cost of *proving* the correctness of an implementation may be unacceptably high. Furthermore, although there are some well-documented counter-examples [12], such as the application of model-checking tools, formal specifications and proofs typically do not scale well to large systems, and are rarely robust in the face of evolutionary changes. This suggests that formal methods most likely have their place in ensuring the robustness and correctness of individual, functional software components, but not necessarily assemblages of components, or non-functional aspects of software systems.

Testing has the clear disadvantage that it can never be used to demonstrate the *absence* of software defects [7]. Despite its obvious shortcomings, however, aggressive testing *during development and evolution* can have a dramatic improvement in software quality *and* software productivity [2]. Furthermore, tests, particularly automated unit tests, can be highly robust in the face of change. The investment in the development of test cases therefore rapidly pays itself off. Considering that software needs to be tested and debugged in any case, the

investment in developing reusable test cases can be paid off even in a single iteration of the software lifecycle. It is therefore astonishing that industry largely continues to consider it to be a completely separate activity from development, and many developers consider it to be an unnecessary luxury.

There are innumerable other techniques and methods that may or may not have an impact on software productivity. Code reviews, coding standards, CASE tools, CRC cards, and so on, affect software quality in various ways, but each may or may not have a positive effect on productivity *in the long run*. We suggest that any technique should be evaluated in the context of software evolution over a large number of iterations. A technique that is not cost-effective over time and robust in the face of changes cannot help software productivity in the long run, or can do so only in a very limited context.

2.4 Peopleware and Agile Processes

Methods, tools and processes all have their place, but it is important to recall that (i) productivity of software developers varies enormously, independently of tools or techniques applied, and (ii) the most important factor contributing to the success of a software project is typically the motivation of the team. Models and standards like CMM [18] and ISO 9000 [19], can be useful to assess the maturity of the process within an organization, but this need not bear any relation to the *productivity* of its development teams. Teams with immature processes may be highly productive, and organizations with mature processes may be moribund. Optimizing processes with negative and positive feedback are surely a Good Thing, but this is not necessarily the key to higher productivity.

In the Silver Anniversary edition of *The Psychology of Computer Programming*, Weinberg notes:

> In the first edition to this book, I predicted, "... attention to the subject of personality should make substantial contributions to increased programmer performance — whether that attention is paid by a psychological researcher, a manager, or the programmer himself" (p. 158). Even though I knew next to nothing about personality at the time, this turned out to be one of my most successful predictions. [22]

Since that time, numerous authors have pointed out that technology has only a limited effect on software productivity [3, 6, 8, 11]. Not only can there be a huge difference in productivity between individual developers (variously claimed to be 10:1 or even 50:1), but factors such as team communication and motivation have repeatedly been shown to far outweigh any technological factor in the success of projects. Alistair Cockburn, for example, reflecting on 20 years of experience with various projects notes that:

- Almost any methodology can be made to work on some project.
- Any methodology can manage to fail on some project.
- Heavy processes can be successful.

– Light processes are more often successful, and more importantly, the people on those projects credit the success to the lightness of the methodology. [5]

DeMarco and Lister give many reasons for these phenomena in their book *Peopleware*, which can be summed up as:

> The major problems of our work are not so much *technological* as *sociological* in nature. [8]

If we can accept the idea that there can be no purely *technological* silver bullet [3], then we must conclude that our only hope is to pay more attention to sociological issues in the software process. The "Manifesto for Agile Software Development" adopts this point of view by proposing that we should value:

> *Individuals and interactions* over processes and tools
> *Working software* over comprehensive documentation
> *Customer collaboration* over contract negotiation
> *Responding to change* over following a plan [1]

Although there appears to be no claim in this manifesto that agile processes will improve productivity in the long run, it is clear that, *as a metaphor*, agile software development gives change a central role, and downplays purely technological tactics. This, we feel, is an important step in the right direction.

3 Research

We have argued that *software evolution* is the most important factor to influence productivity in any software development project. This leads us to the following observations:

1. *Software evolution* is unavoidable, both before and after deployment. Ignoring its influence will kill productivity.
2. *Raising the level of abstraction* is the only way to produce more in the same amount of time.
3. *Agile processes* take into account that software is a living thing, whose life source comes from the interactions of the people who use it and develop it.

We conclude that further research is urgently needed in the following areas:

– Refactoring, reengineering and round-trip engineering,
– Migrations towards component frameworks and software product lines,
– Testing strategies to support continuous development,
– Agile processes,
– Composition languages and infrastructures,
– Architecture-driven software development.

Weinberg's *Psychology of Computer Programming* was written over 30 years ago, but we have been slow to pick up on its lessons. Cockburn puts his case well:

> His characterizations and recommendations, based upon project interviews in the 1960's, are still accurate and significant 30 years later. That validates the stability and importance of these sorts of issues. It is about time we studied these issues as a core to the field of Software Engineering and stopped rediscovering their importance every 30 years. [5]

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004).

Thanks to Roel Wuyts. Markus Gaelli and the anonymous reviewers for various suggestions and corrections.

References

1. Manifesto for agile software development. http://agilemanifesto.org.
2. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
3. Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 1975.
4. Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.
5. Alistair Cockburn. Characterizing people as non-linear, first-order components in software development. In *4th International Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, FL, 1999.
6. Alistair Cockburn. *Agile Software Development*. Addison Wesley, 2002.
7. Ole-Johan Dahl, Edsgar W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.
8. Tom DeMarco and Timothy Lister. *Peopleware, Productive Projects and Teams*. Dorset House, 2nd edition, 1999.
9. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
10. Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.
11. Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison Wesley, Reading, Mass., 1995.
12. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.
13. M. M. Lehman and L. Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.
14. Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison Wesley, 1980.
15. Tom Love. *Object Lessons – Lessons Learned in Object-Oriented Development Projects*. SIGS Books, New York, 1993.
16. M. Douglas McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–150. NATO Science Committee, January 1969.

17. Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995.
18. Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis, editors. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1994.
19. Charles H. Schmauch. *ISO 9000 for Software Developers*. ASQC Quality Press, 1995.
20. Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.
21. Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.
22. Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver anniversary edition edition, 1998.