Software Architecture in Industry: Misuse and Non-Use

Alessandro Maccari
Nokia Research Center
PO Box 407
FIN-00045 Nokia Group, Finland
alessandro.maccari@nokia.com

Titos Saridakis
Nokia Research Center
PO Box 407
FIN-00045 Nokia Group, Finland
titos.saridakis@nokia.com

Abstract: Requirements engineering, object-oriented design, component-based technology, software architecture: trendy terms that bring a lot of excitement to the research community but have little impact on the industrial software development practice. Why does the industry remain practically untouched by the benefits that a thorough system design promises to deliver, and the sort- and long-term risks that its absence reserves? This paper examines a number of reasons responsible for this research-industry gap, assesses their importance and proposes solutions to bypass the identified obstacles.

1. Introduction

Academic research has long acclaimed the benefits of rigorous design in software development. As an integral part of the design, software architecture has emerged in the early 90s and promised to provide a better understanding of the software properties and to deliver systems easier to understand, maintain and upgrade. In some (not to say many) cases however, industrial practice insistently ignores the promised benefits and produces software without relying on software architecture. In other cases, the employment of software architecture is so superficial that makes its use look like a drudgery rather than the safe way to robust development of software. The research-industry gap regarding software architecture seems unjustifiably bigger than the *ipso facto* theory-practice distance.

Based on our aggregated experience coming from our participation in the design of new Nokia products and the architectural assessment of existing ones, as much as from our involvement in European projects in the areas of software re-engineering and O-O design, we have identified a number of reasons for the aforementioned reality. Part of these reasons are technical obstacles that make for the research-industry gap and part of them are excuses that lack technical substance, yet they present the most resistant obstacles. This paper presents the most prominent reasons for the apparent indifference of industry to software architecture benefits, assesses their importance and suggest ways to overcome them in bridging the research-industry gap.

2. Underestimate Software Architecture

Despite the interest of the research community around software architecture, industry has underestimated its importance. Sometimes, software architecture is considered as an issue of purely academic interest, which in practice provides for complete and too general solutions while neglecting the specificity of a particular product. In other cases, conceiving the software architecture of a system is considered as a time of no progress, where the development team is thinking about the product instead of doing the actual work, *i.e.* the coding.

Diagnosis. Identifying the existence of this problem is rather easy. Managers tend to consider that designing the software architecture is a waste of time since it produces no code, and it provides a good excuse for engineers to stay inactive. On their side, engineers consider software architecture as a set of management imposed constraints to their creativity. An unerring way to verify the existence of this problem is to discuss the position "coding captures and tests the design". Reactions like "yeah, sure!" reveal the severity of the situation.

Importance. Underestimating its importance is the most malicious obstacle in bringing software architecture in quotidian industry practice. It is impossible to convince industrial developers to use a new way of developing products when they do not believe in the alien trend recommended by academia. Unfortunately, real-life experience (with most salient evidence the infamous NORTEL case) has shown that without the in-depth and clear understanding of a product's software architecture, the product's evolution leads to a disastrous dead-end.

Solution. The only solution to this problem is education. Personnel engaged in the development of software products should be taught the value of software architecture in order to apply it in product development. The learning process may happen in an organized way (*e.g.* through seminars) or the hard way (*i.e.* suffering the consequences of lacking architectural design). In any case one thing is for sure: if managers and engineers don't understand the need for software architecture, they will not be convinced to use it just because academia suggests it with enthusiasm.

3. Lack of Know-How

Another problem with software architecture is understanding its need but not knowing how to use it right. This is partly due to the fact that the difference between problem and solution domains is not clearly understood [4], which leads developers to confuse architectural design with implementation details. There are also managers and engineers who have been designing and building software long before the emerge of software architecture. These *ex officio* experts in software development often fall into the pitfall of "not knowing what they don't know" [7] about software architecture. Both these facts result in the misuse of software architecture, which can eventually produce the same effects as its non-use.

Diagnosis. The lack of know-how in using software architecture can be detected in the documentation of the product design. Describing the architecture of different product parts using different notations, graphical conventions and expression styles is a safe indication for the lack of rigorous design effort, which in turn reveals *ad hoc* analysis and design efforts. Another indication is the lack of sync between design documents and code documentation. Adding features or modifying the product at the code level without updating accordingly the architectural description of the system degrades the utility of the design documents in product maintenance and evolution.

Importance. This problem is potentially as important as the previous one. Of course, having some use of software architecture in product development is better than having none, but continuous misuse eventually deteriorates the utility of software architecture to the point that matches its complete lack. In addition, due to the law of inertia in software development, the managers and engineers boldly resist any forces that try to change the way they do their work. All in all, the impact of this problem in software product is less severe than the previous one, yet tackling it is more challenging.

Solution. Dealing with the misuse of software architecture requires re-education of the development team. Our experience has shown that one efficient way is to have the designers of a product participating in its architectural assessment. The process is tough especially due to the fact that designers and engineers tend to defend their way of dealing with software architecture. However, it helps designers identify the points of software architecture misuse and how it should have been done properly. In addition, the results of the assessment can be useful for refreshing the overall condition of the software product. One thing to avoid at all costs, is to employ methodologists to point out misuse cases in the product. They tend to criticize any misuse of the design method or the development process, without caring explaining what the practical benefits of their proper use would be.

4. Re-architecting Legacy Systems

Software architecture non-use or misuse is especially evident in legacy systems. Such systems have entered the market some decades ago, and they have been designed and implemented using technologies of that time. Since their appearance, they have evolved to meet the new market needs. But their evolution has taken place in terms of patches adding new functionalities on request, without being followed by the analysis of the modification impact on the overall system architecture. As a result, new modifications become more and more difficult to manage and the whole product turns into a fragile, inflexible construction with mysterious intrinsic properties. In such systems, architecture mining, re-architecting and re-engineering activities needed to give new life to the system meet significant resistance due to the number of changes they require on the out-in-the-market product.

Diagnosis. A software system whose last revision dates before the 90s is a good candidate for architectural misuse or non-use. Independently of the revision date, legacy systems that resist re-architecting activities usually have insufficient design documentation, and the existing one regards more their latest features. In addition, whenever architecture mining has been completed but the re-architecting impact on the product was considered unsustainable, the design documents contain only the structural information about the software architecture, without explaining the rationale of many design decisions.

Importance. This case of software architecture misuse is founded on the cost of doing the architecture mining and the fear of the impact that it will produce on the legacy system. However, the more complex the product is, the more dangerous is to continue evolving it in a patch-like manner, and the higher the cost will be for re-architecting some later version. Continuing the evolution of a legacy system without taking the time to mine its architecture and use the result to understand its properties encompasses the same risks as the nonuse of software architecture in building a new system. In a sense, this case can be seen as another expression of software architecture underestimate.

Solution. Dealing with the need for introducing software architecture to legacy systems if fairly different from the previous two cases. Here, the maintenance team does not need to be convinced about the benefits of the proper use of software architecture. More likely it need to be convinced to invested the admittedly high cost in man-effort for carrying out the architecture mining and possibly the re-architecting tasks. In some cases, it suffices to perform a risk analysis that puts in evidence the high probability of a system collapse if its evolution continues without gaining the deeper understanding of its properties. An important factor is to have highly placed managers and marketing people participating in the risk analysis. These people usually judge the situation of a legacy system from its market success, ignoring that its maintenance is under crisis. If they get convinced to reduce their demands for new product releases until the re-architecting task is completed, then half of the problem is already solved.

5. Missing Architectural Requirements

Another problem leading to the misuse of software architecture is the lack of strict system requirements [5]. Without strict system requirements, software architecture ends up being a set of loose guidelines for the product construction, describing in a vague way the properties of the product that are not well defined in the first place. This weak description of the product behavior leaves many of the system properties to be determined by implementation decisions, and not the vice versa as it should be. The reasons for this situation can be any of the cases discussed in the three previous sections, the lack of time discussed in the next section, or simply the fact that some behavior aspects of the system, especially those related to its quality properties, are difficult to capture and express in the requirements specification.

Diagnosis. Good candidates for systems that lack or have weak architectural requirements are products that appeared as the short-term evolution of prototypes and concept

demonstrators. Such systems lack by default the basic characteristics of a robust construction [1], and most importantly the thorough design phase which consists of the requirements specification and engineering and the design of the software architecture of the system. Another indication of potential lack of strict requirements is the poor documentation of the system design. If there are no hard requirements on the expected system behavior then there is no need for analyzing the architectural solution that leads to a construction which adheres to the system specification.

Importance. The consequences of this case is equally severe to those of underestimating the usefulness of software architecture. Without a set of strict system requirements, there is no need for a rigorous software architecture, and without the latter there are no guarantees for the robustness of the system development or the behavior it will exhibit. The system properties are governed by implementation decisions, which brings the product almost to the same situation as a legacy system discussed in the previous section.

Solution. In the case where the lack of architectural requirements is due to the short interval from the prototype to the ready-to-market product, the basis of the problem is not at all technical and the cure is simple: allow the indispensable time for a rigorous design, where system requirements are strictly defined and the software architecture is properly designed to reflect the solutions given to them. However, if the lack of architectural requirements are due to the fact that the quality properties of the system (*e.g.* availability, reliability, scalability, security, timeliness, *etc*) are difficult to be captured or transformed in architectural constraints, then the problem is much deeper. The truth is that academia does not have a variety of options to recommend for dealing with quality system properties at the software architecture level. Some recent results (*e.g.* see [3] and [6]) promise solutions in this direction, but they are not yet mature enough to enter the industry. Hence, the designers and architects have to struggle with the available means to capture the architectural impact of the system requirements concerning quality properties.

6. Managing by Deadlines

Another factor that plays an important role in the research-industry gap discussed in this paper is the time allocated to system design, which includes the analysis, specification and engineering of system requirements and the conception and design of its software architecture. There is a consensus (*e.g.* see [1] and [4]) about the percentage of the software lifecycle that should be allocated to its design, where roughly 40% is design, 30% is coding/testing and 30% is maintenance. But that is theoretically speaking, because in reality things seem to happen according to the following scenario: deadline for the ready-to-market product delivery is in X days and engineers need ¾ X days to produce the code and some time to test it. In such conditions, where the hard deadline for product delivery is the strictest requirement in the software development, the system design stages tend to be dropped regarded as a luxury under tight time constraints. Although this problem results in software architecture non-use, it is fundamentally different from the previous cases that produce the same effect. In this case, managers and engineers may well understand the significance of software architecture and they may also have a set of detailed system requirements. But they have to deliver code in a fixed deadline and the software architecture of the product is not code.

Diagnosis. This is a problem quite easy to identify. Any software system for which both the requirements and the delivery date are fixed in advance, will have to adapt the duration of the different phases of its lifecycle. And in real life it is always the design phase that pays the adaptation price. Potentially, all products that attack a competitive and time critical domain experience this problem, since they have to provide at least as many features as their competitors (*i.e.* minimum system requirements are fixed) and to be out in the market before their competitors (*i.e.* maximum delay for product delivery is fixed).

Importance. There is no technical problem behind this case of software architecture non-use, which is clearly a management issue. The development of a product for which both

requirements and delivery date are fixed is like an equation for which both variables have been assigned values independently and the only way to make the equation to hold is to adjust its constant, which corresponds to the duration of the software lifecycle phases. Such situations lead with mathematical precision to "mission impossible" projects [8] which is the nightmare of every manager and engineer.

Solution. The safest way to deal with this problem is prevention. Let the delivery date of the product be determined by estimation of the development duration given by the analysis of the system requirements. Otherwise, if the product delivery is scheduled for a fixed deadline, negotiate the requirements so that the resulting development duration fits in the time left to that deadline. Finally, it might be possible to find a compromise between a set of exigent requirements and a product delivery deadline, which would allow the different phases of the software lifecycle to be followed properly. At first sight it sounds exorbitant to suggest trimming or delaying a product as a solution for software architecture non-use. On a second thought however, such an approach would be much more practical than today's reality: products delayed by half a year or more, that fall short from the initial expectations and have to be replaced soon after their market appearance by the next generation which is much more ergonomic, robust, secure, economic, reliable, scalable, *etc.*

7. Conclusions

Despite the widely recognized benefits of rigorous system design and architecting, industrial practice does not always apply the academia recommendations in software development. This produces a domino effect that starts with a number of cases of misuse or even non-use of software architecture, which has severe consequences in the comprehensibility of the product behavior, resulting in turn in significant compromises in the product's maintenance and evolution. This paper has presented a summary of a number of cases of software architecture misuse or non-use, assessed their importance and proposed solutions for dealing with the analyzed cases. The bottom line in every case was that employing software architecture properly in the system development is not as difficult as it may sound; it suffice to inform the development team about the necessity of architecting a system before coding it and about the disastrous consequences of not doing it, and to train the team on how to perform the design and architecting activities properly. To wind up, here is a good piece of advice for the development and the evolution of any system: take the time to design, it will pay back when maintaining and upgrading the system [2].

8. References

[1] F. P. Brooks Jr. The Mythical Man-Month: Essays on Software Engineering. 20th Anniversary Edition, Addison-Wesley, 1995.

[2] A. Cockburn. Surviving Object-Oriented Projects. Addison Wesley, 1998.

[3] V. Issarny, T. Saridakis, A. Zarras. Multi-View Description of Software Architectures. In Proceedings of the 3rd International Software Architecture Workshop, pages 81-84, November 1998, Orlando, FL, USA.

[4] M. Jackson. Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press Books, 1995.

[5] A. Maccari. The Challenges of Requirements Engineering in Mobile Telephones Industry. In the proceedings of the 1st International Workshop on the Requirements Engineering Process (REP99), September 1999, Florence, Italy.

[6] T. Saridakis, V. Issarny. Developing Dependable Systems Using Software Architecture. In Proceedings of the 1st Working IFIP Conference on Software Architecture, February 1999, San Antonio, TX, USA.

[7] B. F. Webster. Pitfalls of Object-Oriented Development. M&T Books, 1995.

[8] E. Youdron. Death March. Prentice Hall, 1997.