Architecture-Centric Evolution

Zdun, Uwe; Avgeriou, Paris

Published in:
EPRINTS-BOOK-TITLE

Architecture-Centric Evolution

Uwe Zdun[1] and Paris Avgeriou[2]

[1] Department of Information Systems,
Vienna University of Economics, Austria
zdun@acm.org
[2] Department of Mathematics and Computing Science,
University of Groningen, the Netherlands
paris@cs.rug.nl

Abstract Despite the general acceptance of software architecture as a pivotal player in software engineering, software evolution techniques have been traditionally concentrated on the code level. The state-of-the-practice is comprised of refactoring and re-engineering techniques that focus on code artefacts. However, recent advances have shifted the focus of evolution from the code level towards higher levels of abstraction and particularly the architectural level. The grounds behind this trend is that architecture captures the architectural knowledge (and particularly the design decisions and their rationale) for the whole system. Architecture can thus facilitate making new design decisions during evolution cycles, having full knowledge of past decisions. Furthermore the revision of non-functional requirements and especially cross-cutting issues can only be managed efficiently at an architectural level. The Workshop on Architecture-Centric Evolution (ACE 2005) attempted to explore the evolution of software systems based on their architecture. The workshop delved into this field, by presenting the latest research advances and by facilitating discussions between experts.

1 Introduction

Industry and academia have reached consensus that investing on architecture in the early phases of the lifecycle is of paramount importance to object-oriented software systems. Moreover, there is an undoubted tendency to create an engineering discipline on the field of software architecture if we consider the published textbooks, the international conferences devoted to it, and recognition of architecting software systems as a professional practice. Evidently, there have been advances in the field, especially concerning design and evaluation methods, as well as reusable architectural artefacts such as architectural patterns and frameworks. And there is growing consensus nowadays about certain aspects of the task of software architecture description, such as the satisfaction of stakeholders' concerns through multiple views, and the use of UML for modelling architecture. Software architecture has become a key issue in the object-oriented community, as architecture is praised for facilitating effective communication between stakeholders, early analysis of the system, support of qualities and successful evolution of the system.

However, the evolution of software systems largely takes place at the code level. For example, a substantial part of industrial practice of software evolution concerns

storing code artefacts in configuration management systems and applying refactoring techniques on them. This hinders the development team from having an overview of the "big picture" and grasping the significant design decisions that can only appear at the architectural level. As a result, the new design decisions that are taken during evolution may compromise or even contradict fundamental principles of the system's architecture. Moreover, the most substantial properties of the system are its non-functional requirements, the so-called "quality attributes", and the evolution of such properties can only be tackled at the level of architecture. To make matters worse, the more complex and sizeable the systems get, the more severe the problem gets. In a product line for example, where domain-specific variation and evolution of the various products is required, software evolution is too complex to be dealt with at the code level, due to the higher level of interdependency between the various software assets in a product-line. In essence, software architecture is the best means for facilitating the synchronization of the system requirements and its implementation during the evolution cycles.

The theme of architecture-centric evolution is complex and multi-faceted, both in its core and in its relevance to other advances of software engineering. In overall, it involves at least the following topics:

- Software engineering processes and methods for architecture-centric evolution
- Theoretical aspects of architecture-centric evolution, e.g. causes of architectural changes
- Configuration management techniques for architectural artefacts
- Modelling Architectures to support evolution of software systems (e.g. through ADLs or UML)
- Synchronizing requirements, architecture and code during evolution
- Architecture-centric evolution in the context of Model-Driven Engineering
- Tools that support and enforce architecture-centric evolution
- Architecture-centric Evolution of Aspects in the Aspect-Oriented paradigm
- Evolution of quality attributes through architectural evaluation
- Software architecture patterns that support evolution
- Evolution of legacy software through its architecture
- Evolution in product lines and system families

The rest of this workshop report is organized as follows: Section 2 presents the theme of the keynote speech, which discussed the metaphors from buildings and biology with respect to architecture and evolution. Section 3 outlines the contents of the papers that were presented in the workshop, as well as some of the discussion they raised. Section 4 describes the findings of the dialogue triggered by the previous sessions and the conclusions reached by the participants. Finally Section 5 concludes with a brief synopsis of the state-of-the-art and future trends.

2 Architecture and Evolution / Building and Biology: A Tale of Two Metaphors

In his keynote, Brian Foote[1] highlighted the importance of metaphors in the field of computer science, their usefulness in some situations, and their weaknesses in others.

[1] More information on Brian Foote can be found on http://www.laputan.org/

During the talk, emphasis was given on comparing the architectural metaphor to other metaphors inspired from biology, like evolution – the topic of the workshop.

Foote initially presented different metaphors that come from the early days of computer science and are still in use today, such as understanding software development as a special branch of mathematics or physics. He argued that these metaphors are strongly influenced by an inferiority complex of early programmers, comparing themselves to people working in these established sciences. In the following years, other metaphors arose, such as the "software engineer" or the "analyst", which also primarily describe a stronger organizational position than a "simple" programmer. The architecture metaphor can also be seen as a part of this tradition.

Foote argued that none of these metaphors really holds, and gave the example of the waterfall model, a common metaphor for the software development process. This model was rarely used in reality – the systems that were finally implemented were quite different to the upfront designs and analyses dictated by a waterfall process. Foote claimed that in the waterfall era, engineers, analysts, and managers only rarely looked at the code because working with the code was seen as lower-class work. Change came with concepts like Design Patterns [6] and Extreme Programming (XP) [2], which raised the attention for the code and low-level design concepts, and resulted generally to the empowerment of the programmer in the software development process. In this sense, Foote proposed that "Craftsmen and Tradesmen" might be better metaphors for programmers than the ones named before.

Next, Foote compared the ideal metaphor of architecting a system with what happens in practice: many systems actually look rather like a Big Ball of Mud [5] than a neatly architected structure. He then asked the question: why is the gap between what we preach and what we practice so large? He gave further evidence by illustrating the problems of masterplans – both in building architecture and software architecture – leading to boomtowns, ghost towns, urban sprawl, etc. The pioneer of patterns in urban architecture, Christopher Alexander, instead sees a city as an organically growing entity, and argues that you cannot impose structure via a masterplan on a city and expect this to work. Similarly, in software engineering, Brooks [3] claims that software is so entangled that it is not possible to impose a structure on it. Foote draws the conclusion that a software system should rather be seen as a body with sub-systems such as the nervous, lymphatic or digestive sub-system, because for both kinds of systems it is hard to apply strict architectural planning (e.g. layering, decomposition) in a mathematical sense. Foote sees the success of glue languages and reusable components as an evidence for the usefulness of this metaphor. He called systems built following the gluing approach "Big Buckets of Glue".

Architectural evolution should thus follow a similar path as biological evolution: systems should grow incrementally and "learn", designs should facilitate change, frameworks should be used, and monocultures should be avoided. In other words, the designs should emerge without a designer, following a process of piecemeal growth [1]. Designs do not have to follow a "glamorous" or clean approach; in biology the genome also makes use of what it can, it uses a lot of duplication, etc. Foote claims that XP has implicitly rediscovered many of these principles. He gave the examples of cut-

and-paste code – which might be a useful technique for a wide range of applications: from initially developing frameworks and trying things out, to the MIR space station!

In conclusion, we need to address the biological nature of our software architectures better: our tools are at the moment too primitive and our metaphors are out-of-date. Refactoring techniques need to take place in richer and higher level descriptions, than just programs. Architecture-Centric Evolution must be two-way: the descriptions must be richer than the actual code (descriptions at the meta-level) and round-trip engineering must be better supported (e.g. to visualize the architecture). Useful approaches for these goals, in turn, can only be found by an evolutionary approach.

3 Issues in Architecture-Centric Evolution

The paper reviewing process was rigorous, assigning at least three program committee members per submitted paper. After the selection process, six papers were accepted, of which five papers were presented in the workshop. The presentations provided a range of interesting research topics, which lead to lively discussions. The essence of each paper as well as the key points of the raised discussions are summarized in the sub-sections below. The heading of each sub-sections is the title of the corresponding paper.

3.1 Architecture-Centric Evolution in Software Product Lines

Hassan Gomaa introduced his notion of evolution based on Model-Driven Engineering, product line architectures and UML 2. The approach follows the model-driven paradigm, and introduces a process model of several steps. The process model, Gomaa claims, is evolutionary, because there are feedback loops between all steps, and there is a continuous evolution of the product line (the artefacts in the product line reuse library) alongside the creation of applications that derive from the product line architecture. Gomaa uses a kernel-first approach, meaning that first a stable product line kernel is designed, and feature-based evolution is applied to this kernel (like artefacts being tagged with optional features). The result is a map of feature/class-dependencies, which can be mapped onto distributed UML 2 components. Different architectural patterns are applied during this process: layered structures (Layers, Kernel), client/server structures (Client/Server, Client/Broker/Server, Client/Agent/Server), and control structures (Centralized, Distributed, Hierarchical Control).

The difference between Gomaa's approach and some of the issues raised in Foote's keynote, raised some interesting discussion points between the workshop participants:

- How does the team size affect the architecture evolution process? On the one hand, Gomaa's approach is clearly aimed at a large team size of at least 10-100 developers and on programming-in-the-large. On the other hand numerous other architecture evolution approaches (e.g. those based on agile approaches) are rather assuming smaller team sizes. The size of the development team is often an implicit assumption in both heavy-weight and light-weight processes. Furthermore, another significant issue in evolution processes is the expected skills/competencies/experience of the developers, which is also left implicit in most cases.

– Can layered structures (and similarly hierarchical structures) really serve as a primary structure for a software system? Or do we have a tendency to use hierarchical structuring, but it is not always appropriate? The question was raised because of Foote's claim that many real systems rather resemble a Big Ball of Mud, rather than a neatly architected structure. This was enforced by the fact that Gomaa showed an example of a layered structure during his presentation, in which layers where bypassed (i.e. it was not a pure layered architecture). In many cases it is hard or even inappropriate to structure systems according to a strict layered/hierarchical structure (see the discussion of masterplans above).
– Does Gomaa's approach allow for round-trip engineering? As a model-driven approach it allows to have code entities synchronized with the models. But it does have difficulties dealing with code changes to entities generated from models.

3.2 Towards a Formal Foundation for Dynamic Evolutionary Systems

Andreas Rausch focused on dynamic (i.e. runtime) system evolution of active components. He claims that in the context of dynamic system evolution, formal techniques for modelling system configurations are required in order to be able to check their correctness. Furthermore, Rausch proposes to map these formal descriptions to description techniques, executable at runtime. Application areas, envisioned by Rausch, are ambient systems, e.g. a PDA which is dynamically introduced into the system of a car. The running toy example of his talk assigns jobs to robots and automatically detects conflicts in job assignments, according to the composition constraints given in UML/OCL. In his motivating example, Rausch shows that these description techniques do not clearly show what a class requires and thus changes can have unwanted side-effects.

To solve this problem, Rausch proposed that components should declare both their required and provided interfaces, and he mapped the UML/OCL examples to the formal component model, introduced in his paper. Unfortunately, it is not possible to prove at runtime that there are no conflicts (a theorem prover would be needed). Instead, Rausch proposed a solution based on constructive composition validation, which is essentially based on a "replay" of a (similar) proof done by a human being before.

Rausch's talk also raised a number of discussion points. A major issue that was brought up concerned whether it is really necessary to have more explicit component dependencies or would a component manager (having all necessary information) solve the issue of conflict detection. Another related discussion topic, concerned the degree of transparency (black-box vs. white-box) components should have in order to support evolution. In some cases, we might require to look inside a component (e.g. when the component has side-effects in its constraints) in order to safely support evolution.

3.3 A Systematic Approach for the Evolution of Reusable Software Components

Fernando Castor Filho presented the next paper co-authored by Ana Elisa C. Lobo, Paulo Asterio C. Guerra, Fernando Castor Filho, and Cecilia Mary Fischer Rubira. The paper is focused on software configuration management for the so-called "eternal" systems, e.g. enterprise systems that are of a large size and have a long lifespan.

The approach introduces a component version model, based on the component definition by Szyperski (see [8]), which uses a component meta-model similar to UML 2. An interesting issue in the version model is that interfaces are not versioned. Filho explained this design decision with practical project experiences in which generating new interfaces was less difficult than providing version changes of interfaces. Other questions that arose were: Why are ports needed in the meta-model? Why is the meta-model slightly different to the UML 2 meta-model? Why are ports and interfaces both connected to Abstract Component? The discussion showed that there are many ways to design a component meta-model, and its still unclear which meta-models are best suited for supporting architectural evolution.

A second element in the approach concerned evolution rules, which were applied to measure the substitution impact of the three change operations: modification, addition, or subtraction. A table connects the attribute or relationship to be changed and the change operations. After applying the rules (perhaps multiple times), a substitution impact that is either low, medium, or high can be inferred from the table. The question was then raised, whether the three change operations (modification, addition, or subtraction) are enough because simple type changes were modelled by a subtraction of the changed attribute followed by an addition. A richer evolution model might be helpful here, but of course it would result in increased complexity.

Finally, the authors propose a versioning scheme with a meaning defined on basis of the calculated substitution impact: $major$ implies changes with a high substitution impact, $minor$ if a medium substitution impact occurs, and $update$ by changes with low substitution impact.

3.4 Architecture-Centric Software Migration for the Evolution of Web-based Systems

Martti Jeenicke presented his paper on software migration in which experiences from two language migration projects are discussed: in both cases, a Web-based System was migrated from the PHP scripting language to Java. The motivation behind the two migrations stems from the poor quality that the two web-based systems suffered from. Users of these systems demanded higher quality and quicker version updates, yet a complete rewrite was not possible due to the code size and the limited development resources. The goal in both projects was thus a gradual replacement which leverages architecture-centric evolution. There was a short discussion between the workshop participants that highlighted the differences of different scripting languages and Web framework architectures: the quality problems in these specific PHP applications can hardly be generalized for scripting applications in general.

Next, two gradual language migration strategies were presented: horizontal migration (layer by layer) and vertical migration (functionality by functionality). The horizontal migration seemed infeasible because this would have required a lot of effort for integrating the languages, so instead vertical migration was used. The question was raised, whether this strategy can be generalized. It turned out that the individual PHP components communicated only over the database and no session data was held in PHP. Thus a Java component could take over seamlessly. This strategy might be impossible or difficult for other (more well-structured) Web applications. Nonetheless, the approach

is applicable as a piecemeal migration strategy for many simple Web applications, in which it is possible that the old system components can evolve, while the new system components are developed.

3.5 Use case maps as an extension of UML for system integration and verification

Czeslaw Jedrzejek gave the final presentation of the workshop. He presented a paper authored by Arkadiusz Rys, Czeslaw Jedrzejek and Andrzej Figaj. The paper discusses a model-driven architecture approach, that supports integration and verification, in a large research project (350 people, 150 developers) with a lot of feature inter-dependencies and constant change. The domain of the project is telecommunications, which is well-understood, but it also involved many research ideas from multiple other domains (such as mobility, IPv6, pervasive computing, user profiles, etc.). The author presented an initial architecture with a huge number of components, which required integration and verification. The author explained that the general architecture originated from earlier architectures (e.g. from the UMTS area), and then grew uncontrolled because there was no general evolution plan for the different research sub-projects.

To integrate this system, object models, sequence diagrams, and state charts were developed. Unfortunately, the sequence diagrams and state charts were on a much too detailed level leading to two problems: first, an overall overview was impossible, and second many changes meant that the sequence diagrams needed to be completely redrawn, which meant a lot of overhead.

The proposed solution to these problems is to base the architecture development on views, similar to the 4+1 model of Kruchten [7], and then to apply scenario-based integration and testing. The author proposes to start with use case maps, build coarse-grained sequence diagrams next, and refine them with use case maps, which are more convenient as a general overview and for making changes. The authors' goal is to formalize the model later on. A discussion followed that involved whether use case maps are really needed here, or if coarse-grained sequence diagrams could be enough to do the job. This discussion showed that the level of detail in models used for architecture evolution is crucial: too detailed models tend to impose complexity and unnecessary overheads, whereas too general models might mean that we miss important aspects. An optimal trade-off needs to be achieved, and it is mostly dependent of the specific project at hand.

4 Discussion and Outcomes

The last part of the workshop was a discussion session, in which we summarized the participants' insights into architecture-centric evolution gained from the presentations and discussions, and discussed the different concerns that arose.

Firstly, the participants agreed that if the architectural metaphor is indeed understood as a static, top-down approach, including a masterplan produced by the architect, the terms architecture and evolution seem to be oxymorons or at least contradictory. In

many practical approaches, such as those taken in agile software development methods, this is not the case. Also, Christopher Alexander's notion of piecemeal growth [1], which has gained much attention and for which we find many analogies in software architecture research and practice approaches, describes an evolutionary approach to building architecture: Alexander's views on architecture strongly refute the idea of an architectural masterplan.

Furthermore, there are many important factors and assumptions to be made, before an architecture-centric evolution approach can be described or understood, and depending on these factors and assumptions, the "optimal" architecture-centric evolution approach may substantially vary. For instance, the workshop participants identified the team size, the project size, and the individual skills of the team members (are they good architects?) as strong influences. Large-scale evolution is a "different game", especially because of the many influences apart from technical problems (e.g. social and political issues) and the sheer complexity of the projects. The participants agreed that in all discussed projects, either small scale or large scale, a lasting architecture emerges late (another reason to refute masterplanning). This is also a reason why it is a good tactic for large-scale projects to strongly base the initial architecture on how it has been done before – by questioning the existing domain experts.

It is also important at which binding time the architecture evolution takes place (e.g. compile time, load time, runtime). The later the binding time, the more difficult it is to make assumptions and to rely on the outcomes of the architecture evolution.

The means to express the architecture-centric evolution also strongly influence the chosen solutions. In the workshop, programming language techniques (e.g. reflection), models (like use case maps and UML 2.0 models), model-driven software development, product lines, formal models, and tools (like Eclipse) were presented as means to express the architecture-centric evolution – all with quite different properties. Also, additional means like verification, testing, validation, etc. of the architecture-centric evolution need to be considered.

In addition, we can learn from how evolution takes place in nature and try to adapt these techniques in software system evolution. We should not make a masterplan for the architecture and its evolution. We should not evolve the architecture in x directions at the same time, but only one at a time. We should try out how different potential evolution schemes would turn out and adopt the successful ones ("survival of the fittest"). In any case we need the right tools to tackle the problem of architecture-centric evolution efficiently.

Finally, a central lesson learned from the workshop discussions is that *consistency* is central: all architectural artefacts (code, documentation, design documents) should at any time reflect the same information. This gets even more important for larger projects, and is important for product lines, round-trip engineering, and model-driven approaches. If maintained independently, they will quickly get inconsistent, especially when rigid processes (like the waterfall model) are used. Thus some means should be provided to ensure consistency (examples are automatically documenting from the code base or generating from a model).

5 Epilogue

In light of the outcomes of this workshop, the main question is not whether it is useful to base the evolution of software systems on their architecture, but how to do it. The workshop raised awareness about the invalidity of current metaphors since they not only lead to false practices, but they also hinder the discovery of more effective ones. It also indicated a slow but steady trend: to think of software systems not as calculated static structures but as living organisms. This paradigm has also been proposed by Clements et al. [4], where the architecture of physiological systems was considered more apt than building architecture. In this sense, a software system should be able to evolve as a living organism: through natural selection, which means to facilitate change and to select the best of alternative options, iteratively and incrementally.

References

1. C. Alexander. *The Timeless Way of Building*. Oxford Univ. Press, 1979.
2. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
3. F. P. Brooks. No silver bullet, essence and accidents of software engineering. *Computer Magazine*, April 1987.
4. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
5. B. Foote and J. W. Yoder. Big ball of mud. In *Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97), Technical Report # WUCS-97-34 (PLoP '97/EuroPLoP '97)*, Monticello, Illinois, September 1997.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
7. P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.
8. C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.

Appendix: Acknowledgement

We extend our thanks to all those who have participated in the organization of this workshop, particularly to the program committee, which is comprised of:

– Bosch Jan, Nokia Research, Finland
– Goedicke Michael, University of Essen, Germany
– Guelfi Nicolas, University of Luxembourg, Luxembourg
– Heckel Reiko, University of Leicester, UK
– Kniesel Guenter, University of Bonn, Germany
– Koschke Rainer, University of Bremen, Germany
– Laemmel Ralf, Microsoft Corporation, USA
– Medvidovic Nenad, University of Southern California, USA
– Oberleitner Joe, Technical University of Vienna, Austria
– Riehle Dirk, independent consultant, Germany
– Tandler Peter, Fraunhofer IPSI, Germany
– Wermelinger Michel, Open University, UK