Submissions for the

4th ECOOP Workshop on
Object-Oriented Architectural Evolution

Tom Mens

Programming Technology Lab
Departement Computerwetenschappen
Vrije Universiteit Brussel
Brussels, Belgium

Galal Hassan Galal

School of Informatics and Multimedia Technology
University of North London
London, United Kingdom

Co-located with the
European Conference on Object-Oriented Programming
University Eötvös Loránd
Budapest, Hungary

June 2001

This workshop is an official activity of the
Scientific Research Network on "Foundations of Software Evolution",
which is financed by the Fund for Scientific Research - Flanders (Belgium)

Table of contents

OPEN QUESTIONS

Based on the main results of previous years, a number of important questions were compiled that deserve further attention. Workshop participants were expected to answer parts of these questions before the workshop.

The questions are subdivided into 5 categories:

1 Domain analysis

- *What is the precise relationship between domain modelling and architectural design/modelling?*
- *How can domain analysis be used to derive a better (i.e. more stable) software architecture?*
- *Can we predict certain types of architectural evolution based on a given domain analysis? Which ones? How?*

2 The use of multiple architectural views

- *Should there be a predefined set of architectural views, or do the kinds of views depend on the problem domain?*
- *Is there a relationship between the different architectural views? Should we allow for explicit constraints between the views? How? Why (not)?*
- *Is there a correspondence between the architectural views and the architectural styles that can be used in those views?*

3 Layered approach

- *How should the different layers be related? Should we put explicit constraints between them? How?*
- *Should there be a limited set of layers depending on the architectural view taken, or can there be an unlimited number of layers?*
- *How can layering ease the transition from a software architecture to the (object-oriented) software implementation?*
- *(How) can other architectural styles than a layered one be used to (i) facilitate evolution; (ii) ease the transition to the software implementation?*

4 Impact of multi-layered view approach on architectural evolution

- *How can views be used to guide/constrain/facilitate changes to the architecture and the implementation?*
- *Does it make sense to distinguish inter-view, intra-view, inter-layer and intra-layer evolution? What is the meaning of this?*
- *Can a multi-layered-view approach be beneficial for checking or enforcing the conformance of a software implementation to its architecture? Does it become simpler to synchronise an architecture and its corresponding implementation?*

5 Applicability of existing techniques

- *Where do existing evolution approaches like reverse engineering, architectural recovery, restructuring, refactoring, architectural reconfiguration fit in? Can they be used in the above approach? How can they benefit from the ideas introduced above?*
- *Can object-oriented techniques such as design patterns, frameworks and inheritance be used to facilitate evolution, or to ease the transition from a software architecture to a software implementation?*
- *How can one determine whether (part of) a given software architecture is stable?*

SUBMISSION BY DEMEYER

Serge Demeyer
University of Antwerp (Belgium)
Department of Mathematics and Computer Science
Universiteitsplein 1 B-2610 WILRIJK
Phone: +32 (0)3 820 24 14 Fax: +32 (0)3 820 24 21
e-mail: Serge.Demeyer@uia.ua.ac.be
url: http://win-www.uia.ac.be/u/sdemey/

1 The issue "multiple architectural views"

1.1 Position

First of all I want to challenge the working assumption that there should be multiple architectural views. Not that I necessarily disagree with this assumption - on the contrary ! However, I sincerely believe that that the number of architectural views should be very small; say three at maximum.

The main reason why I take this position is that for me a system architecture is something else than "course grained design". If a system evolves, the design will evolve as well. However, there must be at least one thing that remains the same, otherwise we wouldn't be talking about the same system anymore. And to me, one of those things that remains the same is precisely the system architecture.

From this description, it is clear that the number of architectural views should indeed be very small. Mainly because if we have too many views, it is likely that one of them will have to change as the system evolves. This follows from the fact that systems are constructed with a team. The safest way to ensure that all team members respect the system architecture is make sure that it is easy to remember. Hence my pragmatic definition of a software architecture:

A software architecture is something that fits on a single page.

1.2 Implications

Adapting this position on architecture has some serious implications:

There is no such thing as architectural drift (erosion).

Since the system architecture is so intimately entwined with the system it follows that an architecture cannot change without completely mutating the existing system. This does not mean that an architecture cannot change; it just means that if one changes the architecture one is actually talking about another system.

As a metaphor, consider evolution in biological systems. At a certain point in the evolution we start talking about the Cro-Magnon man instead of the Neanderthal man. This change of name was caused by significant changes in bone structure, corresponding with what we call an *architectural change*.

Architectural recovery is not a viable research topic

I myself am heavily involved with reverse engineering and have written several papers on (see [Duca99b], [Deme99c]), techniques (see [Deme98o], [Deme00a]) as well as methodology (see [Deme99n], [Deme00n]).

With this background, I would love to write the ultimate paper on architectural recovery. However I consider this impossible because the software architecture is "in the eye of the beholder". As such, one cannot recover an architecture from the source code because it is not there - it is present in the mind of the development team only. Worse, even if it would be possible to recover aspects of the architecture there is no way to set-up a proper scientific experiment to prove that the recovered artifact indeed corresponds with an architectural view. Note that this discussion is related to the intangible nature of a software architecture [Born99n] section 2.1 - point 4 "Difficult to Trace".

2 The issue "layered architectures"

2.1 Position

Layered architectures are desirable, but they are not the only good way to arrange "elements that share the same likelihood of change".

The idea of a layered software architecture, is to divide all system elements in layers, where each layer represents those elements of a system that exhibit the same robustness characteristics, i.e. change at the same rate [Gala98n]. With normal evolution, changes would be limited to a single layer, causing ripple effects to at most one other layer. This limited ripple effect is of course a very desirable feature, because it eases the natural evolution of a system.

2.2 Counterexample

I am aware of at least one counterexample where numerous designers spend the best of their talents to devise a layered architecture ([Camp87a], [Hala90a]). Unfortunately, when actual systems were constructed it turned out that the neat arrangement of layers never worked out in practice because ripple effects always caused changes in it least two other layers. Finally, they abandoned the single dimension layering and came up with a two dimensional arrangement, resembling the Danish flag. Note that the latter resemblance gave name to the architecture; its called the "Flag model" [Oste96a].

3 Conclusion

I challenge two assumptions raised in this workshops call for papers:

Software architecture requires multiple views.

This statement is challenged in the sense that I accept multiple views, but assert that the number of views should be very small (maximum 3).

Layers are the best way to arrange "elements that share the same likelihood of change"

This statement is challenged with a counter example of hypermedia systems.

4 References

[Duca99b] Stéphane Ducasse, Matthias Rieger and Serge Demeyer, "A Language Independent Approach for Detecting Duplicated Code," Proceedings ICSM'99 (International Conference on Software Maintenance), Hongji Yang and Lee White (Ed.), IEEE, September, 1999, pp. 109-118.

[Deme99c] Serge Demeyer, Stéphane Ducasse and Michele Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization," WCRE'99 Proceedings (6th Working Conference on Reverse Engineering), Francoise Balmas, Mike Blaha and Spencer Rugaber (Ed.), IEEE, October, 1999.

[Deme98o] Serge Demeyer, "Analysis of Overriden Methods to Infer Hot Spots," Object-Oriented Technology (ECOOP'98 Workshop Reader), Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, 1998.

[Deme00a] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, "Finding Refactorings via Change Metrics," Proceedings OOPSLA 2000 (Object-Oriented Programming, Systems, Languages, and Applications), ACM Press, October 2000.

[Deme99n] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, "A Pattern Language for Reverse Engineering," Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999, Paul Dyson (Ed.), UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany, July, 1999.

[Deme00n] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, "A Pattern Language for Reverse Engineering," Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing, 2000, Andreas Rüping (Ed.), UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany, July, 2000.

[Born99n] Isabelle Borne, Serge Demeyer and Galal Hassan Galal, "Object-Oriented Architectural Evolution," in ," Object-Oriented Technology (ECOOP'99 Workshop Reader), Ana Moreira and Serge Demeyer (Ed.), LNCS 1743, Springer-Verlag, 1998.

[Gala98n] Galal Hassan Galal, "A Note on Object-Oriented Software Architecting," Object-Oriented Technology (ECOOP'98 Workshop Reader), Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, 1998.

[Camp87a] Brad Campbell and Joseph M. Goodman, "HAM: A General Purpose Hypertext Abstract Machine", in Proceedings HT'87 (Hypertext'87), ACM Press, 1987. Republished in Communications of the ACM, Vol. 31(7), July 1988.

[Hala90a] Frank Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," in Proceedings of the 1990 NIST Hypertext Standardisation Workshop. Republished in Communications of the ACM, Vol. 37(2), February 1994.

[Oste96a] Kasper Østerbye and Uffe Kock Wiil, "The Flag Taxonomy of Open Hypermedia Systems," Proceedings HT'96 (Hypertext 1996), ACM Press, 1996.

SUBMISSION BY NOWACK AND BENDIX

Palle Nowack, Ph.D.
Assistant Professor
Aalborg University & University of Southern Denmark
E-mail: palle@nowack.dk
WWW: www.nowack.dk

Lars Bendix
Aalborg University, Denmark.

We have worked for several years in Software Architecture (SWA) and Configuration Management (CM) respectively. Lately we have become interested in exploring the connections between SWA and CM. It is obvious that there should be some connections as in CM they deal with system models, which could be considered a code view of the software architecture. Furthermore, in the CM world there are established techniques for handling evolution or change to code, techniques that might be applicable to handling evolution of software architectures. From this position, we try to provide clarifications and partial answers to four of the main questions.

1 The use of multiple architectural views

We strongly believe that each development context calls for certain views. The choice of primary abstractions (entity/component concept and relation/connector concept) for each view is determined by the needs of the specific development task. This depends on e.g. the domain (e.g. real-time embedded systems versus business applications), the stakeholders, the platform choice, the organizational context, the developers experience, the choice of tools, etc.

Because we believe that each view is determined by the choice of primary abstractions, we can investigate the views from a conceptual modeling approach. Each architectural concept (architectural abstraction, such as a component or a synchronization mechanism) can be related to other architectural concepts through the concept formation mechanisms (abstraction mechanisms): classification/exemplification, specialization/generalization, aggregation/decomposition.

2 The layered approach

We oppose to the unclear definition of the layer concept. In our opinion layering can mean different things:

- A very general and typical architectural pattern is the division of a system into: user interface, functionality (e.g. Business logic), model (e.g. Model of problem domain), and system interface (i.e. Interfaces to external systems). In this case the inspiration is clearly from the domain of distributed business applications. This is an application of the decomposition abstraction mechanism.
- In the Layer pattern of [Buschman&al1996], layering is explained as "an abstraction". In this case the inspiration is clearly from layered protocol stacks. Each layer corresponds to a specific "level of abstraction". In that description it is very unclear what constitutes a level of abstraction. Is it perhaps a combination of specialization and decomposition levels?

A much better term than layering is *separation of concerns*. Separation of concerns (and the associated principle of low coupling) and it's inverse, unification (and the associated principle of high cohesion), constitute key architectural principles.

3 Impact of multi-layered view approach on architectural evolution

If we want to talk about multiple views and/or multi-layered approaches, we also need to propagate evolution to other views/layers. Or at least to be able to register that there is an inconsistency in our model caused by component xxx and having impact on components yyy and zzz. If we can give some kind of formal descriptions for our different views/layers we might also be capable of having a tool automatically translate evolutionary changes to one into an update to another view/layer. This would give the great advantage that we can work with only one single representation for our views/layers and avoid all the well-known problems in trying to synchronize several representations.

4 Applicability of existing techniques

In our opinion it is obvious to look at the applicability of existing techniques from software configuration management. If evolution of software can be managed and controlled leading to better quality and consistency,

so can software architectures. However, in order to apply CM techniques, the SWA world first needs to come up with some structure and contents on which the well-known change management techniques can be applied. They also need to establish what kind of functionality they require from the change management. Finally, there might be restrictions on form and/or functionality for the known techniques to work.

5 Summary

There is not a fixed set of correct architectural views. The architectural abstractions identified and described within each view should be subject to the concept-formation processes known from OO modeling in order to yield sufficiently rich descriptions. The concept of layering is unclear and should be replaced with the principle of separation of concerns. Manageable software evolution requires tractability between layers. Techniques, strategies, and principles from software configuration management can and ought to be applied in the management of intra-view as well as inter-view dependencies.

6 References

[Buschman&al1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stahl. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

SUBMISSION BY STÖRRLE

Harald Störrle (Dr. rer. nat., Dipl. Inform.)
stoerrle@informatik.uni-muenchen.de
Ludwig-Maximilians-Universität München, Institut für Informatik/PST
Oettingenstr. 67, 80538 München, Germany
Tel. ++49-89-2178-2134, Fax. - 2175 or -2152, Office: E10
http://www.pst.informatik.uni-muenchen.de/personen/stoerrle

I shall focus on the second question concerning (multiple) architectural views.

First of all, however, I want to emphasize my belief in the absolute necessity of multiple views in software architecture. Then, I would like to express my belief that (models of) software architectures *must* be described by the devices of the UML: I shall point out that architectures should be described abstractly, and that the only practically viable abstract notation is the UML.

First, why should we describe (software) architectures abstractly, or, why should we care for architectural models rather than (only) for architectures (that is, real code). The purpose of any model (Latin: scale) is to allow to conduct certain operations on a "smaller scale" (and thus cheaper, faster, easier, ...) than on the real thing. It is as simple as that. A model does not replace the real thing, just like with buildings: sometimes just nice to look at, sometimes useful, sometimes indispensable, but not quite the real thing. This whole point might seem like flogging a dead horse to some, but in many years I have met only very few practitioners who truly understand and acknowledge the importance of models and modeling.

Second, why should we use UML. Obviously, one of the prime purposes of software architectures is to facilitate communication: between developers, between teams, between clients and architects, between users and developers and so on. In general, it is about communication among people of very different backgrounds and across both time and space. It is vital that the medium of communication is universally understood - or, where this is not achievable, as widely understood as possible. This includes a universal meaning, at least for a basic conceptual framework. There is no other such framework in a comparable position as the UML. In the preface of the last UML conference proceedings, the UML has already been dubbed "the lingua franca of the SE community", and rightfully so. Against this background, all deficiencies the UML still does have are irrelevant.

Please note also the difference I make between viewpoints (generic "kinds" of views) and views (concrete "instances" thereof), following to IEEE P1471. I shall not follow the P1471 in another, very important way. Consider the definition of the relationship between view and viewpoint: it is said to be, somehow, and instance-relationship. But "instance" in what sense? Looking at the UML, there are at least two such notions: the kind of relationship between the metaclasses Class and Object, say (or UseCase and UseCaseInstance and so on), and on the other hand, the relationship between a class in an object-oriented programming language and the UML metaclass Class (or the latter and the MOF concept of Metaclass). For simplicity, I call these two kinds of instance-relationships as horizontal and vertical, respectively.

Now, I suggest that the instance-relationship between view and viewpoint is vertical rather than horizontal. So, if view is to become a concept in the UML metamodel, viewpoint should correspond to what will be called UML profile or preface - though these two concepts are not yet defined entirely satisfactorily. In other words, viewpoint should not (as the P1471 suggests) be a concept in the metamodel.

1 Should there be a predefined set of architectural views, or do the kinds of views depend on the problem domain?

In general, the ensemble of viewpoints (!) relevant for some development project are specific and characteristic for that project. Determining the set of viewpoints (declaring it, and only it, as relevant) is a fundamental, instrumental and indispensable step in the requirements elicitation phase. Making it explicit is, in my experience, equally revealing as defining use-cases.

However, there are some views that are likely to occur in very similar shapes under almost all circumstances. For these, particularly rich support in terms of tools and techniques can be provided. So, there is a trade-off between the availability of tools, knowledge, training, experience and so on, versus the degree of fitness to the respective project. In other words, it is not (only) the problem domain that determines the ensemble of viewpoints, but also the solution domain, and to a quite considerable degree.

2 Is there a relationship between the different architectural views? Should we allow for explicit constraints between the views? How? Why (not)?

There are definitely explicit and strong relationships between different viewpoints (and thus between the views realizing these viewpoints), and they may be formulated in a variety of ways, depending on the descriptive formalisms used for describing the views (!) themselves. For instance, in my PhD-thesis, I start from concepts similar to ROOM, use the UML as the concrete and abstract syntax, and provide formal Petri-net semantics for the dynamic models. This way, I can formally define consistency within and among views formally. For example, one may define that a StateMachine and an Interaction of a behavioural run must be consistent with each other, that is, the Interaction actually is a run of the StateMachine. Or one may have both an interfaces and a behavior view, where consistency may mean that the behaviors of these two must be equivalent.

But then, what does "equivalent" mean? In concurrency theory, there are hundreds of suitable equivalence relations (e.g. traces, bisimulations, partial words). So, which relationships *exactly* hold within/among views is, to a degree, dependant on the application and solution domains again. So, for a distributed architecture, one is likely to come across true concurrency phenomena, implying a partial word-based relationship. But then, there may be a global clock, so that interleaving may be enough. Or we might not have the computing power to deal with any of these, or don't care (for the time being), and start with simple traces.

To facilitate experimentation, I have defined implementation/equivalence relationships with parameters for (a) a notion of concurrency and (b) a set of relevant signals (or observations). All consistency conditions I define on views/viewpoints thus have parameter. Which of these are appropriate under what circumstances must be determined empirically - up to now, I can offer only (plausible) suggestions.

There might be also relationships between viewpoints, but if views are on the metalevel and viewpoints on the meta-metalevel, these are of completely different nature than those discussed here, and beyond the scope of my conjectures.

3 Is there a correspondence between the architectural views and the architectural styles that can be used in those views?

This question calls for definitions of "style" and "using a style". In my point of view, a style is a kind of UML Subsystem containing certain specification elements (i.e. and UML ModelElement), so that using a style means: creating something that can be shown to have an implementation or refinement relationship to such a Subsystem. That is, another Subsystem that is an Abstraction of the Subsystem (in the UML-terminology).

SUBMISSION BY COOK ET AL.

Assessing the Evolvability of a Financial Management Information System

Stephen Cook & Rachel Harrison
University of Reading
{S.C.Cook, rachel.harrison}@reading.ac.uk
Brian Ritchie
CLRC Rutherford Appleton Laboratory
Brian.Ritchie@rl.ac.uk

Applied Software Engineering Research Group
School of Computer Science, Cybernetics & Electronic Engineering
University of Reading
PO Box 225, Whiteknights
Reading RG6 6AY
United Kingdom
Tel. : +44 (0)118 931 6423
Fax : +44 (0)118 975 1994
URL : http://www.rdg.ac.uk/~sis99scc/

We have recently commenced an assessment of the evolvability of a financial management information system (MIS) called FRS. This is a bespoke, intraweb application that generates financial reports for a laboratory's project managers from a data warehouse built on the laboratory's accounting system. The purpose of the assessment is to advise the developers of FRS on what kind of revised architecture can be expected to have improved stability despite incomplete knowledge about FRS's future evolution.

We would like to offer tentative answers to some of the questions raised for this workshop, based on the initial phase of this study. We expect that by the date of the workshop we will be able to refine these answers and show how they follow from our empirical findings.

1 Domain Analysis

We believe that domain analysis is critical to predicting important kinds of architectural evolution but this cannot completely substitute for prototyping architectures. For example, for the domains that are relevant to FRS, we can identify three main sources of change:

1. MISs are characteristically subject to changes from two, potentially conflicting, directions:
   a. The information suppliers can change the structure, semantics or availability of the MIS's input data.
   b. The information consumers can change their requirements for the structure, semantics or presentation of the MIS's output reports.

   However, these changes and any conflicts between them cannot always be evaluated analytically. It is often necessary to construct at least a proof of concept architecture to reveal inconsistencies or incompleteness in requirements. (We suggest that this is analogous to the problem of satisfiability in specifying functions in formal languages [1]; in complex cases, the most practical way to prove or disprove the satisfiability of a specification may be to attempt to implement it.)

2. Financial information is subject to characteristic changes that can impact the architecture of applications that use it. For example, recent changes in the laboratory's accounting rules (from "cash" to "accrual" conventions for valuing assets) invalidated most of FRS's reports and required repetitious edits of their SQL query definitions (because the existing architecture provides no shared abstractions for such business rules).

3. Project management reporting requirements also evolve in characteristic ways, most often by new project partners or funders bringing novel approaches. For example, the laboratory's management board expects telephone costs to be grouped with other accommodation costs as infrastructure expenditure; however, one funding agency who is concerned about the cost of highly distributed project teams wants telephone costs to be grouped with travel. This would require an architecture that allows dynamic associations between definitions of accounting categories and reports according to the project and the report's audience.

On the other hand, some aspects of the FRS domain are very stable. The structure of a purchase order or an invoice is much less likely to change than, say, the rules for applying VAT to its item lines. So it would be inefficient to make provision for everything in an architecture to evolve; we need to target the high risk modules, i.e. those modules that are most likely to change and where unanticipated change would be most expensive. Many of these evolution hotspots will be domain specific, found through users' scenarios [2].

2 Applicability of Existing Techniques

One of the secondary objectives of our study is to assess whether existing techniques for evaluating the evolvability of architectures are sufficient. We can't answer this question definitively until our study is further advanced but the initial indications are that a great deal can be achieved by selectively combining existing best practices. Indeed, simply adopting a "design for evolvability" mindset brings significant insights and worthwhile improvements.

We think that The Open Group Architecture Framework (TOGAF) [3] offers a useful baseline methodology to build on, so we are using it to structure our study. However, we also expect that we will have to supplement TOGAF with some specialised techniques. For example, to achieve the targeting of architectural evolvability, we need some method of identifying evolution hotspots; we plan to assess whether techniques such as Software Architecture Analysis Method (SAAM) [2] can be integrated successfully with TOGAF to fill this requirement. There are also shortcomings in the usability of existing techniques. For example, the Architecture Description Markup Language component of TOGAF currently lacks any visualisation tools, limiting its usefulness for describing larger systems.

We have also found that existing techniques and documents can sometimes be reinterpreted from an architecture evolvability viewpoint in a form of reverse engineering. For example, when we reviewed the existing documentation of FRS, we found significant implicit information about expected changes in requirements that had not been incorporated into the current architectural design.

3 Suggestions for Additional Open Questions

3.1 How can the evolvability of alternative architectures be assessed objectively? Are existing design concepts such as cohesion and coupling relevant to architecture evolvability? If so, how should they be measured?

We have done some initial work that suggests that the use of architectural styles could provide the basis for a kind of cohesion metric but we need to establish whether this approach is useful.

3.2 How formal does an architectural specification need to be? Does greater formality make evolvability easier to achieve?

The take-up of formal ADLs has been poor, as with most applications of formal methods. TOGAF, in contrast, provides a structured but informal ADL. What aspects, if any, of architecture evolvability would benefit from greater formality?

4 References

[1] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, et al. Proof in VDM: A Practitioner's Guide. Springer Verlag, 1994.

[2] R. Kazman, L. Bass, G. Abowd and M. Webb "SAAM: a method for analyzing the properties of software architectures". In Proceedings of the 16th International Conference on Software Engineering, 1994. IEEE Computer Society / ACM Press, 1994, p.81-90.

[3] The Open Group, 2000. "The Open Group Architectural Framework (TOGAF) version 6". [URL: http://www.opengroup.org/togaf/]

SUBMISSION BY OSIS

Janis Osis
osis@egle.cs.rtu.lv
Leading researcher
Institute of Applied Computer Systems
Riga Technical University
Meza iela 1/3 - 502
Riga, LV-1048
LATVIA

1 What is the precise relationship between domain modelling and architectural design/modelling?

The answer is short: precise domain model gives precise architectural model. This answer creates other questions. How much does it cost? Which kind of domain model should be used? Perhaps fits the *topological* or *categorical* model?
By my experience, *topological models* can give good results. My colleagues and I are using and have been improving it for more than thirty years. This approach is based on the assumption that a complex system can be described in abstract terms as a topological space (X,Q), where X is a finite set and Q is a topology, given in the form of an oriented graph. The semantics of the model can be formally changed by continuous mapping. The model was used for solving different tasks of pattern recognition, technical and medical diagnostics. The model gives formal class structure with attributes and main operations for object-oriented system analysis. Unfortunately the publications (summary more then hundred) are in Russian. The fundamental of them are [Osis1969, Osis&Al1991]. In English: [Osis&Sukovskis1997, Osis&Beghi1997].

Another important problem is of course that there doesn't exist a precise domain model. In reality we can start from the position that the problem domain is a black box and then move to a more precise model. It is very hard to construct a complete analytical mathematical model of complex system.

1.1 How can domain analysis be used to derive a better (i.e. more stable) software architecture?

Domain analysis gives the structure of the domain, which must be mapped to architectural structure. The structure is one of the factors upon which depends the stability. Full knowledge about structure helps minimize the risk of loose the stability.
As far as topological modeling is at the beginning of software life cycle, I look upon with hope that a complete formal model will help achieve more precise results on the following stages: analysis, design (architecture), testing, and implementation.
Another argument for getting more precise architecture is following the UML process, but I can't give reasonable explanation in a short way. The UML (Unified Modeling Language) is declared as a graphical language for visualizing, specifying, **constructing**, and documentation of a software system. It is adopted as a standard by OMG since November 1997.

2 How can one determine whether a given software architecture is stable?

Stability of a system depends on many factors. If the architecture of a software system is under consideration, one can desire stability, but not determine it. As far as stability depends on structure of problem domain, inputs determined by the designer or "architect", and other factors, one can only check-up the stability of given system. One can study the factors and try to decreases the influence of those that are destabilizing the system most. These are main ideas from Common Control Theory. I delivered a university course on Control Theory for many years and my notices aren't speculative from my point of view. If the management of software architecture is a branch of engineering science they must be used and should be useful for object-oriented architectural evolution.

By the way, the question *"How can one determine whether a given software architecture is stable?"* shows how wrong the term "software architecture" is. An architect is not interested in stability. As artist he is more interested to create something unexpected, then stable. The engineers are carrying about the stability of the projects drawn by the architects. Therefore by software development we deal with **software construction** instead of architecture.

3 References

[Osis1969] Osis J.J. Topological Model of System Functioning. J. of Latvian Academy of Science "Automatics and Computer Science". Riga, 1969, 6, pp. 44 - 50.

[Osis1991] Osis J. J., Gelfandbein J. A. et al. Diagnostics based on Graph Models (with examples from automotive and aviation techniques). Moscow, Transport Publ., 1991, p. 245.

[Osis&Sukovskis1997] Osis J., Sukovskis U., Teilans A. Business Process Modelling and Simulation Based on Topological Approach. Proc. 9th European Simulation Symposium. Passau, Germany, 1997, pp. 496 - 501.

[Osis&Beghi1997] Osis J., Beghi L. Topological Modelling of Biological Systems. Proc. 3rd IFAC Symposium on Modelling and Control in Biomedical Systems. Elsevier Science Publ., Oxford, UK, 1997, pp. 337 - 342.

SUBMISSION BY ANDRADE ET AL.

Luís Andrade, Georgios Koutsoukos, João Gouveia
Oblog Software SA
Alameda António Sérgio 7, 1 A
2795 Linda-a-Velha, Portugal
{landrade,gkoutsoukos,jgouveia}@oblog.pt

José Luiz Fiadeiro
Dep. de Informática, Faculdade de Ciências, Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

Michel Wermelinger
Dep. de Informática, Fac. de Ciências e Tecnologia, Univ. Nova de Lisboa
2825-114 Caparica, Portugal
mw@di.fct.unl.pt

A Two-Layer Approach to Architectural Evolution

Our answers to the organizers' questions are based on our work on the language-independent concept of Coordination Contracts and its associated OO-based technology, which have been described in detail in the last year's workshop and in the papers cited below. In the following we address questions 1.2, 3.2, 4.2, 5.1 and 5.2.

1 Position

In the behavioural view of a system there should be at least two layers: one to handle the interactions, the other to handle computations. This separation of concerns has been advocated for a long time in the Coordination Languages and Software Architecture communities. These layers, besides other advantages, greatly facilitate the evolution of a system since often the behavioural changes can be brought about by modifying the way components interact, instead of changing the components themselves. The former can be achieved by superposing the new functionalities on the components, while the latter has side-effects on all components that use the services provided by the changed components. For this approach to evolution to be effective, both for system design and implementation, two requirements must be met.

First, the layering must be strict, in the sense that the top layer (which handles the coordination) makes use of the services of the bottom layer (which handles the computation), but the latter must not be even aware there is a top layer. If the components have some built-in dependencies on the way they will be coordinated, it will be harder to evolve the system. Achieving this strict separation requires a very strong discipline during domain analysis. The stakeholders have to be quite clear about what is "stable" (i.e., belongs to the core business functionality) and what is "unstable" (i.e., is likely to change in the future).

Second, the coordination aspect must be encapsulated in a conceptual unity with a precise semantics and a technological unity to facilitate implementation. The conceptual unity proposed by Software Architecture is the notion of connector. In our case, it is coordination contracts. The corresponding technological unity is a design pattern, based on other known design patterns such as the Proxy and the Chain of Responsibility, that handles the superposition of the contract's functionality onto the coordinated components in a transparent way to the contract designer.

Within this two-layer approach, there are three kinds of evolutions. The intra-layer evolution for coordination consists just in creating and removing connector (or contract) types, and plugging and unplugging connector/contract instances among existing components. The intra-layer evolution for computation deals with adding and removing uncoordinated components, while the inter-layer evolution deals with changes that involve both layers (like removing a coordinated component, which also requires all attached connectors/contracts to be removed). We feel that (re)configuration languages and techniques as developed for distributed systems and architectural description languages will be helpful to specify and implement those change operations in an easy way for the system designers and administrators.

2 Conclusion

To sum up, software evolution at the architectural level within an OO framework raises many problems and our partial answers cover the conceptual, technological, and methodological dimensions. Conceptually, we advocate, like many others, the separation of coordination from computation, encapsulating each in a layer, and making interactions first-class entities by encapsulating them into special-purpose constructs, called coordination contracts. In this way, changes that occur most often (namely those on the active interactions between components) can be handled more easily. As for the technological aspect, we advocate the use of established techniques like design patterns and reconfiguration languages to make the transition between design and implementation more amenable, and to help people cope with the inherent complexity of managing multiple components interacting in many possible ways. Last but not least, the development of such evolvable systems must start with a careful domain analysis to sort out which are the core functionalities to be provided by components, and which are the "transient" functionalities to be provided by coordination contracts.

3 References

[1] L. Andrade, J. Fiadeiro, J. Gouveia, A. Lopes and M. Wermelinger. "Patterns for Coordination", in Coordination Languages and Models, LNCS 1906, pp. 317-322. Springer-Verlag, 2000.

[2] L.Andrade and J.Fiadeiro. "Coordination Technologies for Managing Information System Evolution", in Proc. CAiSE'01, Springer-Verlag.

[3] J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro. "Tool Support for Coordination-Based Software Evolution", in Proc. TOOLS Europe 2001, Prentice-Hall.

[4] L.Andrade and J.Fiadeiro. "Coordination: the evolutionary dimension", in Proc. TOOLS Europe 2001, Prentice-Hall.

SUBMISSION BY BRIOT AND PESCHANSKI

Jean-Pierre Briot
Directeur de recherche CNRS
e-mail: Jean-Pierre.Briot@lip6.fr -- http://www.lip6.fr/oasis/~briot
tel: +33 (1) 44 27 36 67 -- fax: +33 (1) 44 27 70 00

Frédéric Peschanski
PhD student
e-mail: Frederic.Peschanski@lip6.fr
tel: +33 (1) 44 27 87 53 -- fax: +33 (1) 44 27 70 00

Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Paris 6 - Case 169
8, rue du Capitaine Scott
75015 Paris, France

Some quick answers to some of your questions:

1 The layered approach

Actually I am not sure we should restrict to a layered architectural style. This seems quite restrictive to me.
(By the way, in your text when you say layered approach, do you mean layered architectural style ?).

2 Impact of multi-layered view approach on architectural evolution

Can a multi-layered-view approach be beneficial for checking or enforcing the conformance of a software implementation to its architecture? Does it become simpler to synchronise an architecture and its corresponding implementation?

Prior to conformance between an architecture and its implementation, we also need to check the consistency of a description. One aspect is compatibility of components and how to actually connect them. This is usually done via type checking but in a very static way. (The assemblage of components is not supposed to change then.) We are, like you, convinced that tracking the dynamic aspects (evolution) of the architecture is fundamental.

Frédéric and I are actually currently working on some model and type inference mechanism for incrementally (and dynamically) inferring types of connections between components as the programmer may dynamically add, remove components or connections. (In this work we focus on evolution of the architecture, not on evolution of the behavior of components. This latter is a completely different issue that we also address in other works, e.g., at the agent level). In our work, no assumption is made about a particular architectural style. A simple model of typed event (sent asynchronously) is used, thus completely neutral to the architectural style.

We currently have a prototype, a technical report and a paper currently under submission about the model (this is recent work). We may send you some refs if you need.

On previous work on dynamic adaptation of software architectures (distributed components, e.g. transparently installing protocols between components), here are a few references: [Peschanski1999, Peschanski2001]

3 Applicability of existing techniques

Can object-oriented techniques such as design patterns, frameworks and inheritance be used to facilitate evolution, or to ease the transition from a software architecture to a software implementation?

For inheritance, I do not think so. This is an implementation technique (sharing descriptions and code) at the component level and not at the architecture level.

Answer is yes for the others.

Sorry for being short. I realized the deadline lately… We can elaborate more following this message and obviously during the workshop if you allow us in :)

4 References

[Peschanski1999] F. Peschanski, "Comet - A Component-based Reflective Architecture for Concurrent and Distributed Programming," presented at OOPSLA'99 Workshop on Reflection and Software Engineering, Minneapolis MN, USA, October 1999.

[Peschanski2001] Peschanski, "Architecture réflexive à base de composants pour la construction d'applications concurrentes et réparties," presented at Langages et Modèles à Objets (LMO'2001), Mont-Saint-Hilaire PQ, Canada, January 2001. Proceedings edited by Hermés Science Publications, Paris, France.

SUBMISSION BY WEGE

Christian Wege
wege@acm.org, http://www.purl.org/net/wege
DaimlerChrysler AG

1 How can domain analysis be used to derive a better (i.e. more stable) software architecture?

One of the most important jobs of an IT department of a typical IT user is to limit the IT related costs. This means on the one hand to minimize the number of technologies deployed inside the company. On the other hand this means to ensure that chosen technologies or components fit to what already is in place. Given the premise to rather use existing components than invent them, the architecture of new systems stems to a large degree from the used infrastructure. It is the duty of the IT department to ensure the stability of this technological base.

Domain analysis in this case starts with horizontal components that can be used by a great variety of systems. For example standardize on one web application server product, one portal server product, one message-oriented middleware product, etc. In a second step the most important vertical components can be covered. For example e-mail, calendaring, task list, etc… These vertical components leverage the services provided by the horizontal components (e.g. user authentication, web server, etc.). From the perspective of an IT department, domain analysis stops here in many cases. Now the main task would be domain integration. Much of the functionality to support the business processes is already implemented and has to be leveraged for new systems. For example for building an employee portal the time account management system must be accessible through a portlet.

For a more detailed discussion of this topic please see my position paper "System Lines - Elements of the Software Product Lines Approach for the Construction of Corporate Information Systems"

2 How can views be used to guide/constrain/facilitate changes to the architecture and the implementation?

Talking about a system in architecture terms is a tool to reduce complexity. Concentrating on a specific view on the system helps to highlight specific aspects. Some information is stated in formal architecture documents, some can be requested from the developers who know the system, some can be derived from the actual implementation.

For example, changing the portal server product for a given portal implementation can benefit from several views:

- derive from the implementation the delta to the released portal server product to see the places of customisation
- from a feature point of view find the components which are most critical for a migration because much of the portal implementation relies on this
- from a customisability point of view find the places which are likely to contain customisations on the one hand and to give advice for how to develop in a platform-neutral way for the ongoing development on the old platform

3 Applicability of existing techniques

I would like to answer in a broader sense than to a detailed question in this category.

To a great deal the evolvability of a system is dependent of the used development process. Predicting the likelihood of change for elements in a system is limited because it depends on events located in the future.

A combination of existing techniques like refactoring, aggressive unit testing, continuous integration, etc. to a sound development process like eXtreme Programming or SCRUM can - in my opinion - help build a system which is able to adapt to future change without predicting the nature of the change. (see for example http://www.extremeprogramming.org/)

4 New questions

There is still much work to do to be able to trace the effects of the development process in the product from an architecture point of view. On the other hand an identification of those elements of an architecture that would

most likely change can help concentrate the resources within such a development process to the critical parts. [Wege&Lippert2001] shows how to identify the evolution in code that is the result of such a development process.

How do you identify those areas within an architecture which need special support by the development process?

Which hints/guidelines for building the architecture can the architect derive from the used development process?

What could a round-trip between these two aspects look like? E.g. how should the development process be adapted to meet the needs of the architecture?
(see for example Alistair Cockburn: Just-in-time methodology construction http://members.aol.com/humansandt/papers/jitmethy/jitmethy.htm)

5 References

[Wege&Lippert2001] Christian Wege, Martin Lippert. Diagnosing evolution in test-infected code. Accepted for publication at XP2001, Cagliari, Italy.

SUBMISSION BY MACCARI

Alessandro Maccari
alessandro.maccari@nokia.com
Nokia Research Center
Software Architectures Group
PO Box 407 - FIN 00045 Nokia Group (Finland)

1 Architectural views

There should be a basic set of views that are essential in all systems I can think of. This idea was first proposed by Philippe Kruchten in his classic "4+1 view model" paper [1]. To model our mobile phones software architecture we use the following views:

a) *requirement view* (made of domain model plus use case model, both included in the "+1" part of Kruchten's model),
b) *conceptual view* (made of architecturally significant entities and stereotypes, plus architecturally significant interaction patterns)
c) *logical view* (major logical components and relevant service provide / require relationships between them)
d) *implementation view* (source code or target modules that implement the logical view elements)

I have not seen an example of software architecture of a complex system that can be described completely without including all the above listed views. According to the type of system, and to its architecturally significant requirements, it may be necessary to add other types of view. For instance, architecturally significant requirements for our mobile phones software cannot be modelled satisfactorily only with use cases. The interaction of the various user features needs to be modelled as well [2]. Therefore, we need to produce a feature interaction model, in which we model the interruption priorities, the blocking rules and the modifications that the activation of a certain feature brings to other features. A demo of our feature interaction model is currently submitted to another ECOOP workshop. Feature interaction models are present only in systems, such as telecommunications switches and mobile phones, where externally generated events and user settings can change the way the system responds to a large number of user stimuli. Examples of views that may be needed for selected systems are:

e) *dynamic view* (incl. detailed feature interaction patterns, execution architecture models)
f) *task view* (allocation of components into tasks)
g) *organizational view* (software architecture mapped into the developing organization)

Kruchten has logical, development, process, physical + scenarios. The mappings I can think of are:

- scenarios (PK) included in requirement view (AM)
- logical view (PK) corresponds to logical view (AM) and partly to conceptual view (AM)
- process view (PK) more or less included in dynamic view (AM)
- development view (PK) corresponds to organizational view (AM)
- physical view (PK) is part of the implementation view (AM)

It can be noticed that the main addition that I make to PK's views is the conceptual view.

2 Relationship between the views

The basic views I list are similar to those contained in the Rational Unified Process. Therefore, the process represents the most immediate link between different views. For example, the use cases that are part of the use case view map into the "provided service" interfaces elements that are part of the logical view. I think that an overly tight mapping poses unnecessary constraints to the development of the software architecture. Since the various views lie at different levels of abstraction, mappings are needed, but should not couple the various views too tightly. In a way, it should be possible to change layers of the system at lower level of abstraction (such as the structure and texture, see [3]) without touching the higher level views (such as the architecture). However, there should be a way to map the higher-level artefacts to packages of lower level artefacts. I guess using something similar to the UML stereotypes may be a good way to do so.

Correspondence between architectural views and architectural styles. There definitely is a correspondence, in that the choice of a certain style or paradigm may introduce the need to have additional views (other than the basic ones). However, I do not have a precise idea of how tight the correspondence should be. I think this should be debated in the workshop.

3 References

[1] P. Kruchten, "Architectural blueprints - the 4+1-view model of software architecture", IEEE Software 12 (6), November 1995.

[2] A. Maccari, A-P. Tuovinen, "System family architectures: current challenges at Nokia", in Frank van der Linden (ed.), Software Architectures for Product Families, Springer LNCS 1951.

[3] M. Jazayeri et. al., Software architectures for product families - principles and practice, Addison-Wesley, 2000.

SUBMISSION BY LOURENCI AND ZUFFO

Albertina Lourenci
E-mail: Lourenci@lsi.usp.br
Phone: ++55113818 5254
Postdoctorate researcher
Laboratory of Integrated Systems

João Antonio Zuffo
e-mail: jazuffo@lsi.usp.br
Coordinator of the Laboratory of Integrated Systems
Department of Engineering of Electronic Systems

Politechnic School
University of São Paulo (Brazil)
http://www.lsi.usp.br/~lourenci

1 Should there be a predefined set of architectural views, or do the kinds of views depend on the problem domain?

More and more experts from all fields are unfolding cognitive mechanisms that mimic those from a larger encompassing psychological architecture that regulates behaviour with its constituent array of problem-solving specializations from domain-specific reasoning replacing the mainstream ones associated with the domain general psychological architecture that cannot guide behaviour in ways that promote fitness and evolution. A priori seen in this context any mechanism in the domain-general architectural view cannot produce fit solution unless it is embedded in a constellation of specialized mechanisms that have domain-specific procedures or operate over domain specific representations or both (1).

Hence my domain dependent model to design and plan sustainable cities called *The Model of Primary and Secondary Waves* has this flavour. It is an application of the catastrophe theory to the biological development (2). It was adapted to generate sustainable cities. The processes of the *Primary Waves* called homeostasis, continuity, differentiation and repeatability, intrinsic to the elements activities, structural systems, building systems, environmental comfort, ecohydraulic installations, etc. that shape the sustainable architectonic object describe its interaction with the environment. The design processes of the *Secondary Waves* were created demonstrating that architecture is a language consisting of the planes function and form and the stratas substance of the function, form of the function, substance of the form, form of the form.

The striking feature here is the separation of concerns in the former, and the composionality of concerns in the latter as well as the model's evolvability.

The *Primary and Secondary Waves* generate architectural and urban design. The architectonic object defines the urban ecosystem and is defined by the urban ecosystem. So the *Tertiary Waves* responsible for planning emerge. The underlying geometric modeling is based on symmetry groups of the plane and the dotless planes (fractals) inspired by the graphic work of M.C. Escher. Its action is local, however coordinated to the global, due to the smooth transition of form allowed through the subgroup relationships of the crystallographic groups of the plane. They build infinitely recursive relationships, hence allowing unlimited creation of form (4).

Quaternary waves may correspond to its virtual reality and so on. We may call the waves architectural views. Likewise the architectural views may be generated.

2 What is the precise relationship between domain modeling and architectural design/modeling?

Curiously the prototype based programming language Self and its subjective version Us (5) allows a straightforward implementation of my ecodesign model. The separation and compositionality of concerns is achieved through layers, however layers that remain as objects. The architectural graphical editor implemented in Self allows even the simulation of free hand sketch, the most important step in design conceptualization. If a full-fledged version of it is successful, the architecture is entwined tightly with domain and implementation.

The complexity of the domain model enables the creation of the urban ecosystem as a single organism, an autopoietic entity that is distributed in time and space by recursive partitioning into parts that are conceived similarly structurally speaking to tune in within the whole. The parts into which the urban ecosystem is recursively partitioned include the concept of a sustainable planet, continent, bioregion, cities, boroughs, neighborhoods, buildings etc.

Both for programming in the small and in the large, an evolutive architectural model is a must. The domain model only cares about the single generation of an architectonic object and its surroundings.The recursive generation ability must be grasped by the architecture.
Moreover it is important to follow the trials in generating the myriad of free plans through the subgroup relationships of the crystallographic groups of the plane.A free plan is generated for each element and must be integrated in a single final free plan for each architectonic object. Each free plan of an apartment must transform smoothly to the free plan of the next apartment. The form of the building must conform to the form of the next building and so on. These relationships function like music scales. So a careful analysis of more pleasing transitions of form to the eye must be the final outcome. So myriad of architectonic scales may be discovered. These must be managed at the architectural level as intralayers.
Of course the specific computational concerns such as distribution, concurrency, graphical editors, printing belong to the architectural level as well as to the language of implementation.
I believe object oriented techniques such as design patterns may allow this architectural modeling with relative ease. Of course I intend to create my own design patterns especially applying Peirce's general theory of the sign. This theory allows to model the Dynamic and the Immediate Object, namely the transcendent and immanent aspects of a thing. The model as conceived above care only about the scientific aspects of the design, hence the Immediate Object (6).
Traditionally architectural configurations or topologies are connected graphs. I intend to introduce concepts such as tiling design patterns to reason on the components and connectors of the architecture (7) keeping faithful to the algebraic/geometric nature of the ecodesign model.

Issues of layers, multiple architectural views are intertwined with the answers above.

3 References

(1) Leda Cosmides and John Tooby: Origins of domain specificity: the evolution of functional organization in Minds, brains and computers. Edited by R. Cummins and D.D. Cummins
(2) Zeeman, E.C.: Primary and Secondary Waves in developmental biology. Catastrophe theory. Selected papers 1972-1977.Addison Wesley Publishing Company, Inc. 1977
(3) Mehmet Aksit: Composition and separation of concerns. Computing Surveys 28 A 12/ 96
(4) Albertina Lourenci: A proposal of a prototype based object oriented knowledge system for designing and planning sustainable cities. PhD thesis. Faculty of Architecture and Urbanism University of São Paulo. 11/1998
(5) Randall Smith and David Ungar: A simple and unifying approach to subjective objects in Theory and Practice of object systems. Vol. 2(3), 161-178, 1996
(6) Vincent Colapietro: Is Peirce's general theory of the sign really general? In Transacations of the Charles Saunders Peirce Society. Spring 1987.
(7) David H. Lorenz: Tiling design patterns. A case study using the interpreter pattern.Proceedings from OOPSLA'97.206-217