4th Workshop on Object-Oriented Architectural Evolution

Tom Mens[1] and Galal Hassan Galal[2]

[1] Postdoctoral Fellow of the Fund for Scientific Research - Flanders
Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be
[2] School of Informatics and Multimedia Technology, University of North London
166-220 Holloway Road, London N7 8DB, United Kingdom
galal@acm.org

Abstract. The aim of the fourth workshop on Object-Oriented Architectural Evolution was to discuss into more detail a number of important issues raised during the previous workshop: the relationship between domain analysis and software architecture, the importance of architectural views and layering techniques, and the applicability of existing object-oriented principles and evolution techniques. This paper summarises the results of the debates held about these issues, reports on convergences of view taken place during the workshop, and suggests some research topics that are worthwhile to pursue in the future.

1 Introduction

The workshop on *Object-Oriented Architectural Evolution* was co-located with the 15th *European Conference on Object-Oriented Programming* (ECOOP 2001), which took place at the Eötvös Loránd University in Budapest, Hungary, June 2001. This workshop was the fourth in a series of consecutive ECOOP workshops in the area of software architecture and its evolution [1–3]. Previous workshops have proved very successful and stimulating, culminating in reports that contained novel and exciting views on what software architecture is, or should be, and how architectural issues may be approached from fresh perspectives. Past workshops also incorporated relevant experience reports and suggestions for future research in the area of evolving software architectures, especially object-oriented ones.

One full day was allocated for the workshop (Monday, June 18, 2001). In preparation to the workshop, participants were requested to provide partial answers to a list of questions that were suggested by the organisers as likely candidates to stimulate discussion.

In total, 11 submissions were received, 10 of which were accepted for workshop participation [16]. A total of 12 people (including the organisers and representatives of 8 submissions) actually attended the workshop. The workshop participants came from research institutes in 8 different countries: Belgium, Brazil, Denmark, Finland, France, Germany, Latvia and United Kingdom.

2 Workshop Preparation

2.1 Q & A-style

The nature of the workshop was intended to be incremental, building further on the results of the last three years. To ensure an active collaboration between the participants, the call for participation was built up in a Q & A-style. After a briefing of the results of the previous workshop [3], a number of tentative open questions was suggested to which participants needed to provide an answer before the workshop.

We are convinced that this way of soliciting submissions greatly contributed to the success of the workshop. The process of asking questions is a well-known hermeneutic cognitive process in philosophy. It made it much easier to compare points of divergence between participants' opinions, and enabled us to detect points of agreement on particular topics. Therefore, we believe that a Q & A-style would be an interesting alternative for other workshops as well.

2.2 Categories of questions

Since the previous workshop [3] emphasised the need for *domain analysis*, as well as the importance of *architectural views* in combination with a *layering mechanism*, we decided to raise specific questions with the aim to explore these issues in more detail. The questions were subdivided into 5 categories:

1. Domain analysis
   (a) What is the precise relationship between domain modelling and architectural design/modelling?
   (b) How can domain analysis be used to derive a better (i.e. less change-sensitive) software architecture?
   (c) Can we predict certain types of architectural evolution based on a given domain analysis? Which ones? How?
2. The use of multiple architectural views
   (a) Should there be a predefined set of architectural views, or do the kinds of views depend on the problem domain?
   (b) Is there a relationship between the different architectural views? Should we allow for explicit constraints between the views? How? Why (not)?
   (c) Is there a correspondence between the architectural views and the architectural styles that can be used in those views?
3. Layered approach
   (a) How should the different architectural layers be related? Should we put explicit constraints between them? How?
   (b) Should there be a limited set of layers depending on the architectural view taken, or can there be an unlimited number of layers?
   (c) How can layering ease the transition from a software architecture to the (object-oriented) software implementation?
   (d) (How) can architectural styles other 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
   (a) How can views be used to guide/constrain/facilitate changes to the architecture and the implementation?
   (b) Does it make sense to distinguish inter-view, intra-view, inter-layer and intra-layer evolution? What is the meaning of this?
   (c) Is a multi-layered-view approach 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
   (a) 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?
   (b) Can object-oriented software engineering principles 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?
   (c) How can one determine whether (part of) a given software architecture is stable?

3 Contributions

3.1 Additional questions and workshop topics

Besides the questions mentioned above, several contributions proposed additional questions and topics to be discussed during the workshop:

The need for *separation of concerns* was addressed by various authors.

Harald Störrle, *Janis Osis* and *Stephen Cook* addressed the need for formalisms in the context of architectural evolution. This gave rise to a variety of questions. How formal does an architectural specification need to be? Does greater formality make evolvability easier to achieve? Should a domain analysis be formal?

Stephen Cook asked whether it is possible to assess the evolvability of alternative architectures objectively? Are existing design concepts such as cohesion and coupling relevant to architecture evolvability? If so, how should they be measured?

Christian Wege proposed to look at the software development process from an architecture point of view. More precisely, he posed the following questions. 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 software 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?

Albertina Lourenci emphasised the use of semiotic, hermeneutic and autopoietic reasoning and domain modelling to build more expressive and evolutive software architectures [12].

3.2 Controversial statements

With the aim to stimulate discussions, some participants deliberately made some controversial statements in their contributions. For example, *Serge Demeyer* postulated the following challenging assumptions:

- A software architecture is something that fits on a single page.
- There is no such thing as architectural drift (erosion).
- Architectural recovery is not a viable research topic.

3.3 Clusters of interest

All received submissions [16] were double reviewed by the organisers. By compiling all contributions in a so-called *contribution matrix* (see Table 1), and comparing the addressed topics, the organisers were able to divide the contributions into three major clusters of interest. The first relates to the possible links between the original problem domain and the software architecture(s) that can be related to it. The second addresses the range of available architectural views and layering techniques, and their relevant merits and contribution to the evolvability issue. The third relates to the applicability of existing object-oriented principles and evolution techniques for improving architectural object-oriented software evolution.

Question	Respondent 1	Respondent 2	Respondent 3	Respondent 4	Respondent 5
1. Domain Analysis					
1.(a)	Osis				Lourenci et al.
1.(b)		Andrade et al.	Wege		Cook et al.
1.(c)		Osis			Cook et al.
2. Use of multiple architectural views					
2.(a)	Demeyer	Nowack et al.	Störrle	Maccari	
2.(b)			Störrle	Maccari	
2.(c)			Störrle	Maccari	
3. Layered Approach					
3.(a)		Andrade et al.			
3.(b)		Andrade et al.			
3.(c)				Maccari	
3.(d)	Demeyer	Briot et al.			
4. Impact of multi-layered view approach on architectural evolution					
4.(a)			Wege	Nowack et al.	
4.(b)		Andrade et al.			
4.(c)	Demeyer	Briot et al.			
5. Applicability of existing techniques					
5.(a)	Demeyer	Andrade et al.	Wege	Nowack et al.	Cook et al.
5.(b)	Briot et al.	Andrade et al.	Störrle	Maccari	Lourenci et al.
5.(c)	Osis				Cook et al.

Table 1. Contribution Matrix

4 Warm-up presentations

In the morning, after a general introduction and getting acquainted, two warm-up presentations were given. The topics were chosen according to the bias of the organisers, with the aim of stimulating discussions.

4.1 Designing for Architecture Evolvability

The first talk, presented by *Stephen Cook* based on joint work with Rachel Harrison and Brian Ritchie, was entitled "Designing for Architecture Evolvability: some conclusions from a Management Information System case study". During this presentation, a number of interesting ideas and guidelines were suggested:

It is important to look at evolvability as a viewpoint. Software designers' way of thinking changes significantly once they are encouraged to think about the evolvability of their designs, in terms of the aspects of change that are likely to affect their designs. This says something about the nature of training that needs to be provided to software designers. This idea of consciously thinking about evolvability is useful for all stakeholders involved in the software engineering process, including business analysts, domain experts, software developers and even project managers.

Patterns might be useful to localise evolution and improve evolvability of architectures.

Established software engineering principles (such as separation of concerns, abstraction, refinement) are useful to solve architectural problems. In order to achieve this, there is a need for an architectural language that explicitly incorporates these principles. Such an explicit language or formalism is also needed during domain analysis.

4.2 Reflecting on Architectural Evolution

The second talk, presented by *Palle Nowack* based on joint work with Lars Bendix, was entitled "Reflecting on Architectural Evolution: questions from change management and conceptual modelling". Among others, he raised the following issues:

- A software architecture should essentially remain stable. We only have architectural evolution if a software system evolves in a discontinuous way.
- Changes to the software architecture can come from a variety of different sources: from changes in the problem domain, application domain, solution domain (e.g., changes in technology and infrastructure) and development domain.
- Instead of layering, separation of concerns and the associated principles of coupling and cohesion should be used, because they constitute commonly accepted architectural principles.
- Techniques from software configuration management [**?**,17, 6, 15, 8] can be used to enhance tool support for architectural evolution. For example, one should try to identify the right units (operators and operands) for describing architecture operations. While these concepts are well-understood for configuration management, this is not the case for architectural change.

5 Summary of the debate

This section summarises the discussions that were held during the remainder of the day. Subsection 5.1 summarises the debate held on the first cluster of interest identified in section 3.3, namely the relationship between domain modelling and software architectures. Subsection 5.2 addresses the second cluster of interest, namely the use of architectural views and architectural layering. The third cluster of interest was only discussed very briefly, but instead a discussion ensued about the importance of the software process. This is summarised in subsection 5.3.

5.1 Domain analysis and software architecture

The first cluster of interest focused on the relationship between domain analysis/modelling and the derivation of a software architecture from it. This issue basically boils down to how to map elements of the domain to elements of a certain software architecture in a way that enhances the latter with respect to certain agreed priorities or quality attributes.

Domain analysis First we looked in more detail at the domain analysis, and brainstormed about what should be included in the domain model. As a very general guideline, *the domain model contains only those things or constraints that are imposed on the technical design from the outside*. Things that require an explicit decision from the software engineer (software architect, software designer or software developer) reside at a lower level and should be excluded from the domain model.

More precisely, the domain model can include the following information, although the exact content will vary enormously from organisation to organisation, and even from software system to software system:

- Business and organisational issues, organisational patterns, organisational structures, social structure, company culture
- Processes (interactional processes, design processes, planning processes, ...)
- IT infrastructure, in as far as this cannot be freely chosen (hardware configuration, network infrastructure, operating system, development environment)
- Stakeholders (includes users, domain experts, software engineers, designers, developers, managers, ...)
- Change scenarios (including design alternatives and quality attributes)
- The architecture of the domain: as a separate concept that needs some deliberation as to the optimal way of mapping into the architecture of the software artefact. The domain architecture is the set of domain concepts that describe it at a relatively high level of abstraction and their organisation. The assumption being that a mapping between the domain and the software artefact will enable the latter to change in tune with the domain more easily.
- Various relevant concepts that do not belong to any of the heading above

Note that the information contained in the domain model should not be restricted to a fixed moment in time. *The domain model should contain all relevant information from past activities, as well as future scenarios.* Indeed, it is important to introspect past

activities to reflect upon future activities with the purpose of establishing the degree of variability of various elements of the domain and the architectures that support it. This also became apparent from Stephen Cook's presentation. When thinking about future evolvability, people start making different decisions.

Software architecture As a second part of the discussion, we addressed the question of what is an architecture and what should we do with it? The discussion started by statement of Serge Demeyer that "architecture is in the eyes of the beholder", which he linked to his controversial requirement that "a software architecture should fit on a single sheet of paper" (see subsection 3.2). The discussion then went to approve that the nature of software architecture depends on the purpose behind having an architecture in the first place. Since there are typically many stakeholders involved, with each stakeholder having a specific purpose in mind, it then becomes vital to make an architectural representation explicit so as to enable sharing it amongst interested parties.

It is also important to develop a shared representation of such an architecture and to maintain it over time, so that changes are reflected in the representation and so that stakeholders can inspect the change against conflicts with their own purposes. Nokia, for example, has an architecture team with a variety of experts. A meeting-point document on architecture is regularly produced.

From domain model to software architecture As a third part of the discussion, we tried to identify how the information contained in the domain model can help us to build a suitable software architecture. To this extent, we needed to address several questions. Which concepts in the domain model correspond to which parts of the software architecture? How do we determine which architectural style to use? How do we decide which views to take and what to include in each view?

It was mentioned that an architecture does not have to 'fix' everything specified by the domain model. The skill of the architect is to select from a large amount of data the elements that can or should be mapped onto the architecture.

5.2 Architectural views and layering techniques

The second cluster of interest focused on architectural views and layering techniques.

Architectural views. The workshop participants agreed that architectural views are necessary, although the number of views should be kept small. This necessity of views is also acknowledged by the recent IEEE 1471 standard on architecture descriptions [10, 14]. It states that a view is a collection of models that represent one aspect of an entire system. A view applies to only one system, not to generalizations across many systems. The standard introduces the concept of viewpoints to capture common descriptive frameworks across many systems. Viewpoints are the vehicles for writing reusable, domain-specific architecture description standards. They establish the languages or notations used to create a view, the conventions for interpreting it, and any associated analytic techniques that might be used with the view. An architecture description must define the viewpoint for each view it contains.

Kruchten [11] proposes a 4+1 view model of software architectures using five concurrent views that each address a specific set of concerns of interest to different stakeholders in the system. The *logical view* supports the functional requirements, i.e., the services the system should provide to its end users. The *development view* describes the software's static organisation in its development environment. The *process view* describes concurrency and synchronisation aspects. The *physical view* describes the mapping of the software onto the hardware. Finally, the *"+1" view* illustrates architectural decisions using use cases or scenarios.

Allesandro Maccari expanded this into the following set of views used within Nokia. The *logical view* comprises the major logical components (expressed as subject areas or top-level areas) and their interactions. The *implementation view* corresponds to the development view of [11]. Additionally, there are 4 views that are not available in [11]. The *dynamic view* is a domain-specific view that deals with feature interaction, a very important issue in the domain of telecommunication software [13]. The *task view* allocates components into tasks. The *organisational view* maps the software architecture into the developing organization. Finally, the *conceptual view* represents the core layer of the architecture, which is impossible to change without rebuilding the entire software system. It contains the architectural rules and architecturally significant interaction patterns that need to be adhered to by the software system.

One important question is: *how detailed should each of these views be relative to each other?* Maccari mentioned *Conway's Law* [5], essentially stating that the architecture of the artefact mirrors the architecture of the team that developed it (and, more generally, the structure of a system tends to mirror the structure of the group producing it). The same comment was also made as regards the structure of the client organisation: quite often, the architecture of telecommunications software reflects the billing structure! The role of the chief architect is to balance the work organisation, and the architecture of the artefact.

In the context of architectural evolution, another important question was raised: *which views give you the most information about how the software will change?* The answer to this question was the conceptual, logical and dynamic view. The conceptual view acts as a boundary on design decisions to ensure, among many things, economic development, outside which you lose control over design decisions. The logical view maps the features of the system (as viewed by the user or the marketing department) to a high-level view of software areas. The dynamic view deals with feature interaction, and is also useful to use scenario-based software architecture assessment to compare and evaluate architectural proposals.

Next, the discussion focused on the question: *which information should be contained in the core layer of a software architecture?* In other words, what are the hard parts in the architecture we are contemplating? We expect the answer to be embedded in some way in a comprehensive understanding of the problem domain.

A representation of the core layer (i.e., the conceptual view) was suggested by Maccari: it is basically a set of constraints that act as a boundary. Within the boundary, software evolution remains manageable. Once the boundary has been crossed, the situation becomes unknown, and evolution of the software may become very difficult, if not impossible. Hence, the core layer of an architecture defines what shouldn't be changed,

while it enables other kinds of change. Note that what is considered to be architectural and what is not changes with time, and according to the priorities and policies of the software development organisation.

Obviously, it is a difficult problem do decide which constraints should be included in the core layer, because the number and nature of constraints varies with the nature of the domain under consideration as well as the various technological offerings. Therefore, *each constraint in the core layer should be accompanied by a careful justification, possibly in the form of a concrete change scenario*. These change scenarios can be defined based on the domain model, and correspond to decisions one cannot afford to change because they are prohibitively expensive. Obviously, these decisions can vary dramatically from organisation to organisation, or even from system to system, which largely stresses the importance of a comprehensive understanding of the problem domain.

At Nokia, resources required for testing (primarily testing time) are a major driver of the choice of constraints and rules in the conceptual view to achieve a degree of predictability of integration testing resource requirements.

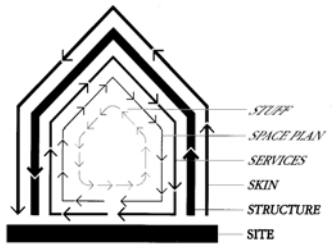
Layering. There was some disagreement about what it means to have a layered architecture, or even whether the notion of layering is beneficial from an architectural point of view. A number of alternative techniques were suggested: coupling and cohesion, composition and decomposition, clustering, and separation of concerns. One participant argued that the point of having an architecture is to think explicitly about the areas of high variability and isolate them into layers or areas that need to be loosely coupled from the rest of the architecture.

A potentially useful layering mechanism could be achieved by categorising the architectural elements in *collections that share the same likelihood of change*. This corresponds to the revolutionary definition of software architecture that was agreed upon during last year's workshop [3].

An advantage of this kind of layering mechanism is that, with normal evolution, changes would be limited to a single layer, causing ripple effects to only adjacent layers, with the architecting aim being isolating such effect to the relevant layer as far as possible. This facilitates the natural evolution of a system. Nevertheless, in his contribution, Serge Demeyer presented a counter example where numerous designers spent the best of their talents to devise a layered architecture. Unfortunately, when the actual software system was constructed it turned out that the neat arrangement of layers did not work out in practice because ripple effects always caused changes in at least two layers.

The Architectonic View. Galal presented a diagram that he used in the first workshop of this series to illustrate his architectonic view of software architecture. This view focuses on the adaptability and maintainability of software through an architectural emphasis. The view is based on Brand's ideas about how buildings learn and adapt over time [4]. Brand's *shearing* layers perspective of buildings refers to the layers of change that comprise buildings. Brand identifies 6 layers in a building that change at different rates. From the slowest to the fastest these are: Site, Structure, Skin, Services, Space

Plan, and Stuff (meaning things like furniture, decorations, light fixtures and appliances). This view is fundamentally *normative*, i.e., it is based on a study of the types of changes that typically affect buildings, after construction and delivery to clients, as a result of adaptations by their users. Buildings that gracefully accommodate such changes are the ones that please their users most and remain useful for longer. Such buildings are capable of accommodation of unforeseen uses because the layers that make them up are loosely coupled. These layers *slip past* each other: changes to one layer do not necessitate changes to others. Note here that the low coupling is not at the level of individual bricks or other individual constructional elements: rather, the decoupling referred to is at the level of categories, or layers, of such elements. The constructional elements are categorised according to the degree of susceptibility to, or speed of, change that they share. The categorisation also relates to the degree in which each layer constrains others, and to the scale of disruption that the change of each entails.



Another view congenial with Brand's is that of the architectural theorist and critic Kenneth Frampton [9]. He uses the term *architectonic attributes* to refer to the *light* versus *heavy* characteristics of constructional elements. He uses vernacular examples, such as the Greek temple, to show how the architectonic attributes of various construction technologies have been used to retain certain values that are germane to certain *cultures*. Frampton observes how the architectonic role of constructional material can be reversed from one culture to another: the relative lightness or heaviness of constructional elements is a function of the culture, its particular condition, and the expressions that it tries to achieve. So again, we encounter another view of architecture that demarcates categories of building blocks according to their relative stability characteristic, this time with the role of *cultural specificity* and variances spelled out. This position reflects that which was articulated by Maccari on how Nokia's software architecture reflected things such as the organisation of the development team, and the billing structure of mobile communications; also how the constraints implied by the architecture are these that the software development organisation wished to preserve as far as practicable.

The point that was made here is that the way in which architecture constrains an artefact of any sort is very much dependent on the culture that spawns it. What is *heavy* and stable is more constraining than what is *light*. The choice lies with the culture tradition that uses or indeed develops the building, or in our case, software. There is

therefore a need to investigate these cultural and organisational choices, as well as the way in which they allow or constrain certain types of software evolution. For example, the conceptual architecture view that we reported is also a cultural and business choice, which leads to certain allowances to and certain constraints on how the software can be feasibly evolved. In the Nokia example therefore, the conceptual architecture is more akin to the "site" or "structure" layer in the figure.

We submit at this juncture that there is substantial validity to the view that software architecture should be less concerned with structural elements and more with categories of software system's components, in the large-grain. Stratifying such categories according to their relative rigidity, scale and speed, of change can help our architecting effort by making the *architectonic* nature of the software object as whole clearer and thus shareable by other stakeholders. This clarity means that the impact of various architectural decisions can be studied more carefully, and in conjunction with the relevant stakeholders. The aim is also to support the understanding and consequent design of systems, so that adaptability properties are maximised, but with respect to the particular situation (read culture) that we refer to. This view of systems was referred to as the architectonic view of software architecture.

5.3 Process considerations

Much of current practice tends to add architecture at the end of the software design process if at all. This practice is not recommended as the designers' motivation to do a proper architecting job will be weak (since architecture, as an after thought will be produced mostly to satisfy a documentation standard, without being allowed to affect other design considerations). An architecture description should be the culmination of a process, not as something that we start with, and neither as something to be added later (this goes somewhat against the ethos of the software patterns movement, which tends to place the architecture up front). Architectural thinking and consideration should permeate the whole software design process, where iterations lead to the refinement of architecture against a selected set of purposes, or quality attributes.

There was also an analogy to be drawn between process and art. The traditional way of software engineering is like classical music, with strict rules of composition and tightly planned and directed performance. On the other hand, eXtreme Programming (XP) can be likened to improvised Jazz, where musicians create new tunes and idioms on the fly in response to fellow Jazz musicians. Palle Nowack pointed out, however, that even for Jazz improvisation, there are broad compositional rules that must be followed so that the result is indeed improvised Jazz that is pleasant to hear. This can be regarded as a kind of compositional architecture.

This brings us to problems with operationalising a sound architecting process: how do you get to software architecture? The issue of stakeholders comes up again here, as the utility of architecture to various stakeholders needs to be evident, so that the stakeholders are motivated to participate fully in the architecting process. This begs the question of who should the architects be exactly?

Since a large number of architectures are usually computationally equivalent, it is non-functional requirements (such as evolvability, reusability, scalability, flexibility) that drive the major part of software architecture. Wege (who is a proponent of XP [18])

reminded the group that XP states that designers should not think ahead, and focus on satisfying the immediately evident required functionality instead.

The group agreed that not all aspects of architecture are relevant to all types of software and that we must not generalise: the architectural implications for embedded systems are different from business systems. It was also stressed that architecting process issues are critical to getting from analyses of the domain to software architecture. From the viewpoint of operationalising the architecting process, the importance of representation formalisms was highlighted. Appropriate representational formalisms and primitives that cover a wide range of issues, but that also remain accessible to the variety of stakeholders present, need to be established and monitored.

6 Conclusions and recommendations

This section summarises the convergences of views that have taken place during the workshop, and clearly shows that our worskhop series has made considerable progress.

6.1 Requirements for a domain model

All participants agreed that a domain model should not be overly complex, in that it should only contain constraints that are imposed on the software engineer from the outside, and that do not require any decision from the software engineer. There was also a consensus that a domain model should contain relevant information from past activities, as well as future scenarios.

6.2 Communicative role of a software architecture

The communicative role of a software architecture was considered to be essential by all participants. The architecture document (which should be small) plays a vital role in the communication and resolution of views and enabling debate about architecture. A software architecture is a means to share information between the various stakeholders (not only software developers, but also the users, business analysts and project managers).

6.3 Evolutive role of a software architecture

It is also important to *share a software architecture over time*, so that changes are reflected in the representation and so that stakeholders can inspect the change against conflicts with their own purposes. In this light of architectural evolution, it is absolutely essential that *a software architecture needs to be specified explicitly*. Whether this is best achieved informally, using natural language or a freehand drawing, or formally, using an architectural description language or modelling notation, remains an open issue.

Note that the need to take evolution of an architecture into account is also acknowledged by the IEEE 1471 standard [10, 14]. It defines architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution."

6.4 The constraining role of software architecture

The tentative definition of software architecture that was proposed during the previous workshop [3] was reconsidered. Since it is very important that a software architecture should be robust towards evolution and change as little as possible, this definition focused on the quality requirement of robustness towards changes: *A software architecture is a collection of categories of elements that share the same likelihood of change. Each category contains software elements that exhibit shared stability characteristics.* Obviously, this definition focuses on an evolution viewpoint for software architectures. For other stakeholders, that have another viewpoint (e.g., security), a different definition might be more appropriate.

All participants of this year's workshop agreed with this definition, and amended it with an extra requirement: *A software architecture always contains a core layer that represents the hardest layer of change. It identifies those features that cannot be changed without rebuilding the entire software system.* To determine precisely what are the hard layers in the architecture that we are contemplating, a comprehensive understanding of the original problem domain is essential.

Although the core layer can be specified in a variety of ways, a specific representation was suggested : *The core layer of an architecture is basically a set of constraints*. This set acts as a boundary, within which software evolution remains manageable. Note that the architectural constraints are also subject to evolution due to changes in the domain or technology, so this boundary may change over time. Therefore, each constraint must be carefully justified in the form of a concrete change scenario that illustrates why a certain kind of change may be prohibitively expensive.

6.5 Need for a formal architecting process

The group stressed that architecting process issues are critical to go from domain analysis to software architecture. From the viewpoint of operationalising the architecting process, the importance of representation formalisms was highlighted. These formalisms should cover a wide range of issues, but should also remain accessible to the variety of stakeholders present.

7 Future plans

Obviously, there are still many outstanding questions that could not adequately be addressed during this workshop due to time constraints. The following questions were considered important by the participants, and might be the topics of next year's workshop.

What can we learn from real-world industrial case studies about managing software evolution? Concrete examples from practice are needed, including process issues, documentation and methodology.

Is architectural evolution essentially different from ordinary software evolution? How can we ease the transition from a software architecture to a software implementation?

Where do formalisms fit in? Which aspects of domain analysis and software architecture can/should be formalised? Which aspects are better left informal? Does greater formality make evolvability easier to achieve?

What existing techniques and tools can/should we use to support architectural evolution? This includes established object-oriented software engineering techniques (such as separation of concerns, refinement, abstraction and composition) as well as established evolution techniques (such as software configuration management).

How do proceed from the technical architecture to refactoring of the architecture?

8 List of participants

Workshop organizers. *Galal Hassan Galal* (University of North London, UK); *Tom Mens* (Vrije Universiteit Brussel, Belgium).

Authors of a workshop submission that actually attended the workshop. *Jean-Pierre Briot* (Université Pierre et Marie Curie, Paris, France); *Stephen Cook* (University of Reading, UK); *Serge Demeyer* (Universiteit Antwerpen, Belgium); *Albertina Lourenci* (University of São Paulo, Brazil); *Allesandro Maccari* (Nokia Research Center, Finland); *Palle Nowack* (Aalborg University and University of Southern Denmark, Denmark); *Janis Osis* (Riga Technical University, Latvia); *Christian Wege* (University of Tubingen and DaimlerChrysler AG, Germany).

Last-minute workshop attendees without a submission. *Claudia Pons* (Universidad Nacional de La Plata, Argentina); *Pascale Rapicault* (Université de Nice Sophia Antipolis, France).

Co-authors of a workshop submission that did not attend the workshop. *Luis Andrade*, *João Gouveia* and *Georgios Koutsoukos* (Oblog Software SA, Portugal); *Lars Bendix* (Aalborg University, Denmark); *José Luiz Fiadeiro* (Universidade de Lisboa, Portugal); *Rachel Harrison* (University of Reading, UK); *Frédéric Peschanski* (Université Pierre et Marie Curie, Paris, France); *Brian Ritchie* (CRC Rutherford Appleton Laboratory, UK); *Harald Störrle* (Ludwig-Maximilians-Universität München, Germany); *Michel Wermelinger* (Universidade Nova de Lisboa, Portugal); João Antonio Zuffo (University of São Paulo, Brazil).

9 Acknowledgements

We express our extreme gratitude to all workshop participants, whose interesting contributions and active discussions made the workshop a real success.

The workshop was supported by the *Scientific Research Network on Foundations of Software Evolution*. This is a research consortium coordinated by the Programming Technology Lab of the Vrije Universiteit Brussel (Belgium), and it involves 9 research institutes from universities in 5 different European countries (Belgium, Germany, Austria, Switzerland, and Portugal). The consortium is financed by the Fund for Scientific Research - Flanders (Belgium).

References

1. Borne, I., Brito e Abreu, F., De Meuter, W., Galal, G. H.: Techniques, tools and formalisms for capturing and assessing architectural quality in object-oriented software. In: Demeyer, S., Bosch, J. (eds.): ECOOP 1998 Workshop Reader. Lecture Notes in Computer Science, Vol. 1543. Springer-Verlag, Berlin Heidelberg New York (1998) 44–71
2. Borne, I., Demeyer S., Galal, G. H.: Workshop on Object-Oriented Architectural Evolution. In: Moreira, A., Demeyer, S. (eds.): ECOOP 1999 Workshop Reader. Lecture Notes in Computer Science, Vol. 1743. Springer-Verlag, Berlin Heidelberg New York (1999) 57–79
3. Borne, I., Galal, G. H., Evans, H., Andrade, L. F.: Workshop on Object-Oriented Architectural Evolution. In: Malenfant, J., Moisan, S., Moreira, A. (eds.): ECOOP 2000 Workshop Reader. Lecture Notes in Computer Science, Vol. 1964. Springer-Verlag, Berlin Heidelberg New York (2000) 138–149
4. Brand, S.: How buildings learn - What happens after they're built. 2nd edn. Phoenix Illustrated, London (1994)
5. Conway, M.: How do Committees Invent? Datamation Journal (April 1968) 28-31.
6. Conradi, R. (ed.): Software Configuration Management. Proc. Symp. SCM-7 (Boston, MA, USA). Lecture Notes in Computer Science, Vol. 1235. Springer-Verlag, Berlin Heidelberg New York (1997)
7. Estublier, J. (ed.): Software Configuration Management. Selected Papers Symp. SCM-4 and SCM-5. Lecture Notes in Computer Science, Vol. 1005. Springer-Verlag, Berlin Heidelberg New York (1995)
8. Estublier, J. (ed.): System Configuration Management. Proc. Symp. SCM-9 (Toulouse, France). Lecture Notes in Computer Science, Vol. 1675. Springer-Verlag, Berlin Heidelberg New York (1999)
9. Frampton, K., Cava, J.E.: Studies in Tectonic Culture - The Poetics of Construction in Nineteenth and Twentieth Century Architecture. MIT Press (1995)
10. IEEE Software Engineering Standard 1471: Recommended Practice for Architectural Description for Software Intensive Systems. IEEE Computer Society Press (October 2000).
11. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12(6)**, IEEE Computer Society Press (1995) 42–50
12. Lourenci, A.: An evolutive architecture reasons as a semiotic, hermeneutic and autopoietic entity. Proc. Int. Workshop on Principles of Software Evolution, Vienna (September 2001)
13. Maccari, A., Tuovinen, A-P.: System family architectures: current challenges at Nokia. In: van der Linden, F. (ed.): Software Architectures for Product Families, Proc. 7th Int. Workshop Database Programming Languages. Lecture Notes in Computer Science, Vol. 1951. Springer-Verlag, Berlin Heidelberg New York (1999) 107–
14. Maier, M.W., Emery, D.E., Hilliard, R.: Software Architecture: Introducing IEEE Standard 1471. IEEE Computer **34(4)**, IEEE Computer Society Press (April 2001) 107–109
15. Magnusson, B. (ed.): System Configuration Management. Proc. Symp. SCM-8 (Brussels, Belgium). Lecture Notes in Computer Science, Vol. 1439. Springer-Verlag, Berlin Heidelberg New York (1998)
16. Mens, T., Galal, G. H.: Submissions for the 4th ECOOP workshop on object-oriented architectural evolution. Technical Report, Programming Technology Lab, Vrije Universiteit Brussel (2001) http://prog.vub.ac.be/OOAE/ECOOP2001/ooaesubmissions.pdf
17. Sommerville, I. (ed.): Software Configuration Management. Proc. Symp. SCM-6 (Berlin, Germany). Lecture Notes in Computer Science, Vol. 1167. Springer-Verlag, Berlin Heidelberg New York (1997)
18. Wege, C., Lippert, M.: Diagnosing evolution in test-infected code. In: Succi, G., Marchesi, M. (eds.): Extreme Programming Examined, Proc. 2nd Int. Conf. eXtreme Programming and Flexible Processes in Software Engineering. Addison-Wesley (2001) 127-131