Introducing the Software Architectonic Viewpoint

Alessandro Maccari[1], Galal H. Galal[1,2]
1: Software Architecture Group, Nokia Research Center, P. O. Box 407, FIN – 00045 NOKIA GROUP
2: School of Informatics and Multimedia Technology, University of North London, 166-220 Holloway Road, London N7 8DB.

Abstract: Managing evolution of complex software architecture is a continuous challenge in industry. Systems such as mobile handsets undergo a continuous increase in complexity, while the fast market evolution imposes quick integration of new features. Being able to easily manage software architecture evolution is the basis for shorter time-to-market and faster product release. The term "viewpoint" has become familiar with the publication of the IEEE standard 1471-2000 on recommended practices for architectural modelling. Based on the classical 4+1 view model, we have elaborated our own set of viewpoints in order to support our domain-specific architectural modelling needs. We hereby justify the introduction of the architectonic viewpoint, which models the evolutionary aspects of software architecture. The term, as well as the rationale behind it, is inspired from architecture as in buildings. We describe the viewpoint and the way it links to the others we use. Additionally, we briefly elaborate on the other viewpoints that we use for architectural modelling of mobile telephone software architecture. We provide basis for discussion and further research into the matter.

Keywords: software architectonics, architectural evolution, the architectonic viewpoint, architectural viewpoints.

1. INTRODUCTION

A large proportion of the software development work for systems such as mobile phones consists of maintenance and evolution of complex architectures and integration of new features in existing systems at market

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

speed. Proper architectural modelling plays an essential role in such tasks: well-structured and exhaustive architectural documentation is a key aid for successful management of the evolution of the software architecture and the roadmapping of system features.

Architectural modelling based on different views has been used for a long time. The milestone of the discipline was set back in 1995 by Philippe Kruchten's paper on the 4+1 view model [Kruchten]. While proposing some improvement and extensions, we believe that the core of the model is still valid, and will refer to it throughout this paper. A summary of the viewpoints that we use appears in section 2.

We believe that the viewpoints thereby described, and the views extracted from them, provide a fairly good static description of software architecture. However, when it comes to managing evolution, it can be very hard to extrapolate the necessary information from static views.

A workshop recently held with the ECOOP conference [ECOOPworkshop] focused on architectural evolution. Its participants debated how to model architecture so to be able to manage and, in theory, also predict evolution. (Clearly, accurate predictions of the evolution of software, and of any other system for that matter, is impossible; however, the discussion produced interesting hints to identify the parts of the system that are more likely to evolve over time, as opposed to those that will hardly change during the system's future life.) Motivated by the conclusions thereby reached, we present the idea of an architectonics viewpoint, which classifies software components according to their relative stability characteristics into different layers that have different likelihood and scales of change. The architectonic viewpoint is the focus of this paper, and concerns the evolutionary aspect of software architecture. The architectonic viewpoint advocates the organisation of software into layers that share similar likelihood of change, thus localising the need to propagate changes and minimising the disturbance to other layers that need not be disturbed. It is justified in section 5 and described in section 0.

Additionally, we present three other viewpoints that we regularly use (section 7). Telecom systems, and mobile handsets in particular, have complex runtime behaviour due to events from the environment and the mobility of the devices. Understanding such behaviour (at both user and operating system level) is crucial for writing effective test cases, and this is the reason why we propose the dynamic view and task view.

Finally, we introduce the notion of the organizational view, which models the structure of the developing organization. While this view is trivial in most cases, when working in large, multi-site organizations such as Nokia it becomes crucial.

We conclude the paper with a list of topics for validation and further research, and locate this work within the context of previous research.

2. OUR BASIC VIEWPOINT SET

We have elaborated the 4+1 view model to fit the specific needs posed by real-time, embedded telecom systems, such as mobile telephones. We presented our flavour of the model [Riva] during a workshop at the last ICSE conference [ICSEworkshop]. In brief, we model architectures by means of four basic viewpoints, resulting in the following views:

a) Requirements view (which includes a domain model and a requirements model, mainly in the form of use cases);
b) Conceptual view (made of architecturally significant entities, stereotypes, constraints and interaction patterns);
c) Logical view (major logical components and relevant provide service/ require service relationships between them);
d) Implementation view (source code or target modules that implement the logical view elements).

Our model does not include a deployment view (yet), since it has proved to be rather trivial in the case of our mobile handset software architecture.

The main improvements with respect to the original model are:

a) The Requirements view contains not only use cases (engineered functional requirements) but also non-functional requirements and, most importantly, a domain model [Jackson], which we find of invaluable help in modelling the reality and the problem domain;
b) The Conceptual view, containing architecturally significant constraints and patterns, is not explicitly present in Kruchten's model (we believe it's meant to be implicit in the logical view, although the author is the best person to ask!).

We briefly elaborate on the requirements and conceptual views in sections 3 and 4 respectively.

In addition to these basic four viewpoints, we introduce others that we have found useful for our purposes.

The runtime viewpoint deals with the user-visible behaviour of the system. We assert that such behaviour cannot be effectively modelled by means of use cases or any other traditional requirements modelling technique. We elaborate on this topic in section 7.1.

The task viewpoint has to do with operating system task allocation. It is discussed in section 7.2.

The organizational viewpoint models the organization that develops a certain software system, together with the dependencies between the different teams and units. It is briefly elaborated in section 7.3.

3. REQUIREMENTS VIEW

In the 4+1 view model, the "+1" part is the one about requirements, and contains only functional requirements expressed in terms of use cases. The use cases provide the "glue" between the other views, in that they model the business reasons for a certain system to be built. However, we advocate that it is not sufficient to model only functional requirements in order to understand all the architectural constraints for a system.

First, qualitative requirements may have substantial consequences at the architectural level. An example is quality of service, which in some cases is enforced by network standards. The architecture of the protocol software is usually influenced by quality of service, since products cannot be commercialised if the requirement is not met.

Additionally, we believe that architecture is a solution to certain problems. These can partly be summarized as requirements, but this is not sufficient. A certain knowledge about the problem domain is essential in order to devise a good solution [Jackson].

An example from the mobile phone domain is a message. Digital mobile phones have always been capable of sending and receiving messages. Until very recently, messages were simply made of (almost) ASCII text, up to 160 characters long. In the past two years, the usage of messages has boomed, and Nokia has launched products that support picture messages, where a picture can be attached to the message. In the future, with the availability of the so-called third-generation networks, higher bandwidth will allow the transmission of multimedia messages, which we guess will be very similar to today's emails and include attachments of various kinds (documents, pictures, sounds), that can be "viewed" or "played" by certain applications.

Clearly, the concept of message is no more as simple as it used to be. Understanding what a message is (and how it can evolve in the future, see section 4) is crucial for architecting the messaging software in mobile handsets. The answer to the question "what is a message?" (and to similar, even more complicated questions, such as "what is a call?") lies at the heart of the domain model.

We feel that a well-defined domain model is an essential part of the architecture, since it allows the understanding of the system's architectonic nature and its implications on the evolution of the software system (see section 4).

Therefore, our requirements view contains requirements (both functional and qualitative) and a domain model. We usually prefer use cases [Cockburn] as a means to model functional requirements.

It is to be noted that even with the addition of a domain model and of qualitative requirements (the -ilities), our requirements view does not contain any description of the system structure, and thus still qualifies to be a "+1" view in the Kruchten sense.

4. CONCEPTUAL ARCHITECTURE VIEWPOINT

We devised the conceptual viewpoint (that infers the conceptual view) after realising that the logical view in the 4+1 view model did not explicitly contain a number of things:

a) the constraints on the types of components that can exist and on the relationships between components that are allowed by the architectural rules (usually part of the architectural style);
b) the architectural patterns (or design patterns that have been applied at the architectural level);
c) the main system-level rules that software developers and architects must follow when building new parts of the system (e.g. when to use a certain type of component, architectural heuristics).

We felt that such information should be part of a view of its own. That view contains what everyone who works with a particular software architecture needs to know, and should be included in the basic training for developers and, most importantly, chief designers and architects.

An example from the mobile phone domain is the Symbian [Symbian] user interface architecture. Symbian is an operating system platform that is targeted to high-end mobile handsets with a rich functionality and an advanced graphical user interface. The Symbian platform is developed independently of the target products, thus making an ideal example of a product family software platform.

Applications are built using a flavour of the module-view-controller design patterns, which allows an application engine to run independently of its UI appearance. This way, an application can be adapted to different user interface styles by rewriting only the view code, without touching the engine code. This may sound like a requirement for application development, but instead it has architectural implications, since all applications that run on Symbian platforms must conform to this rule. This is a general architectural rule, which poses constraints on all applications.

5. THE ARCHITECTONIC VIEWPOINT

The architectonic viewpoint derives from the concept of systems and software architectonics [Galal & Paul, Galal99, Galal2000] focuses on the evolvability concern. The term architectonic derives from similar use of the term in the area of building architecture [Frampton], where the term was used to refer to lightness vs. heaviness of constructional elements. This way of considering construction also reflects the relative ease of changing, or transporting or adapting various elements of buildings. It aims at modelling software systems a way that makes explicit the difference in the nature of software components in terms of their relative likelihood of change. However, seeking to categorise such components, perhaps arranged into layers, rather than attempting to predict the precise nature of change, we believe, best achieves this analysis. We turn to the Architecture discipline for an informative analogy. In a book about how buildings learn and adapt over time, Stewart Brand [Brand] refers to the layers of change that comprise buildings. Brand identifies six 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). Figure 1below illustrates this view.

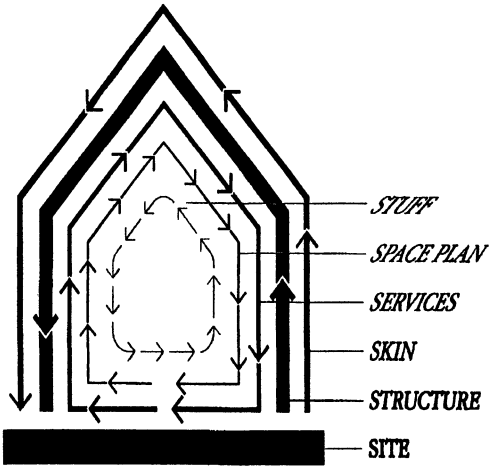

Figure 1. Shearing layers of change (courtesy of Phoenix Illustrated).

The view derived from this architectonic viewpoint is fundamentally *normative*, i.e. it is based on longitudinal studies of the types of changes that

normally affect buildings after construction and delivery to clients, as a result of use and adaptations by their users in the western culture. Buildings that accommodate such changes gracefully 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 and other individual constructional elements such as doors and windows: rather, the de-coupling 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 in each entails.

It is important at this juncture to point out that the placement of types of building components into particular layers is fundamentally a cultural choice. For example, the nomad's site is the most volatile and continuously changing aspect of his habitat, whilst the fabric of his tent remains the same for a long time. Frampton illustrates this by referring to a variety of anthropological evidence, pointing out to ethnographic studies that have demonstrated the constancy of the light vs. heavy differentiation of constructional efforts across cultures [Frampton].

So again, we encounter a view of architecture that demarcates categories of building blocks according to their *architectonic* or relative stability characteristic, this time with the role of *cultural specificity* and variances spelled out.

Our conclusion at this point is that the way in which architecture constraints an artefact of any sort is very much dependent on the *culture* that spawns it. What is *heavy* is more constraining than what is *light*. The choice lies within the culture tradition that uses or indeed develops the building, or in our case, software. This has been recently reflected in the writings on *Enduring Business Themes* (e.g. [Fayad]) and how types of domain constructs are implicit, essential and rely on intuition for their discovery. Enduring Business Themes are also the most stable concepts in a given problem domain, which to us is generated by surrounding business and organisational cultures. There is therefore a need to give close investigative attention to such cultural and domain-bound aspects and the choices they spawn, and to the way in which they allow or constraint the evolution of the software artefact.

For example, the particular Conceptual architecture view that we reported above is also a result of cultural (and business) choices, which leads to

certain allowances towards 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 that appears in Figure 1. This complies with the view that the aforementioned ECOOP workshop converged upon: namely that software architecture is primarily an expression of constraints that are often deeply embedded into the context of the system, as it relates to both developers and consumers.

When focussing on the evolvability concerns, the view that software architecture should be less concerned with structural elements and more with categories of software components (in the large-grain) becomes particularly useful. Stratifying such categories according to their relative rigidity, scale and speed of change can help the architecting effort by making the 'architectonic' nature of the software artefact as whole clearer.

The layering according to change should mirror that of the domain model. For instance, in the messaging example we quoted above, messages are evolving from simple, text-only, short messages to rich messages with attachments (email-like) and to multimedia messages. However, the fundamental functional need to send SMS messages will remain for long time (at least due to backwards compatibility). The evolution in the domain should be mirrored in the software architecture, where the messaging software is not likely to evolve much in the part where it handles SMS messages. The architectonic view should uncover this mapping and separate the core, stable parts from the ones that are likely to require future changes.

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 with respect to the particular situation (read: culture) that we are referred to.

We refer to this view of systems as the architectonic view. We believe that there is not much that is fundamentally new here in this type of differentiation: the SPARC database model, modern operating systems, and the ISO OSI reference model follow the same principle. What is new is our reference to the way in which the specific problem domain can affect and spawn such categorisation for each individual, substantive domain.

6. DESCRIBING THE ARCHITECTONIC VIEWPOINT:

We provide below a description of the architectonic viewpoint according to the precepts of the IEEE-Std-1471-2000 standard [IEEE] for the architectural description of software systems. According to this standard, an

architectural viewpoint is a standard or a template for constructing a view. The architectonic viewpoint that we propose is mainly concerned with facilitating the description of, reasoning about and communication of the rationale underlying evolvability-centred views about the software system in question. Below is our template, including an explanation in square brackets where the slot heading in the template might be less than obvious. We wish to stress at this point that the instantiation of the viewpoint into view is fundamentally grounded in the problem domain that gives rise to the software artefact (that aims to satisfy one of its needs). The software architect must stop to ponder, inevitably in consultation with the stakeholders of the domain, as to what aspects of the domain are the most stable, thus corresponding to the idea of enduring business themes, and what is less stable. In our view, it is not possible to do this by merely referring to standard practice of say, isolating common areas of change like user interface, but that the architectonic nature of the problem domain must be investigated and in some way mapped to the software architectural layout.

6.1 The architectonic viewpoint

Ontological origin [by this we mean the essential nature of the view and its origin, rather than the result of performing an ontological study on a given domain of discourse]: *Normative*, from the study of extant software artefacts over a significant number of instances.

Epistemological status: *Contingent* on relevant stakeholders consensus that in itself is in flux and continually achieved. This makes the process for reaching and maintaining the consensus paramount, as well as fully recording its underlying rationale and connections to any existing contextual analyses.

Appropriate methodology: *Hermeneutic / interpretive*, but also fully *grounded* in available data to aid traceability, sense making and forward-projection (possibly using a set of organisation or domain-wide change scenarios).

Stakeholders: Developers, Maintainers, Clients and Sponsors.

Concerns: *Evolvability* and *adaptability* in the face of unplanned changes to requirements or concrete environmental evolution scenarios.

Viewpoint: *Architectonic*, meaning the categoric differentiation of layers according to any or some of the following attributes: stability, constraining power, ease of change, likelihood of change, scale of effect, magnitude of change. The term derives from its use in architecture (as in buildings) to study and differentiate categories of constructional elements according to the degree of heaviness (or lightness) of their attributes, which relates to their

degree of permanence, symbolic value, grounding in context (physical, business and social) and ease of change and movement (portability).

View: *Layers* of software (we suggest a small number; perhaps a maximum of 6) arranged according to the degree of stability or likelihood of change.

Known inconsistencies/ operationalisation issues:

a) there are often conflicts between the architectonic view and the performance and validity views of real-time systems;
b) different stakeholders may adopt different architectonic views for the same viewpoint to suit their particular positioning and aspirations.

Rationale: for numerous man-made and natural systems, and especially that succeed in adapting to varying circumstances, it is observed that different parts of the system change at the varying rates, scales, speeds or costs. Examples of this are abound in biological systems: animals, forests, eco-systems and so on.

6.2 Justification for the architectonic viewpoint

Given a certain software system, if we stratify its components into layers according to its change-based characteristics, we observe that it is possible to distinguish layers that are more stable than others, and perhaps last the lifetime of the system, while others change more frequently. The intuition is that if the stratification into layers is valid from the point of view of the original problem domain, (i.e. there is a degree of isomorphism or correspondence in the mapping between the architectonic nature of the problem domain, or its environment if you will, and the architectonic view of the software), the latter becomes significantly easier and cheaper to maintain, and with fewer modification risks.

Fundamental to this point of view is the distinction between constructional elements (and associated construction plans), which we do not view as the primarily architectural, and the overall *architecture* that acts at a more *global level* and thus includes *integration with and correspondence to the surrounding context.* This view does not regard schemes that discuss individual buildings blocks and their inter-relationships as architecture, as these are better viewed as structural representations. Software patterns are examples of these largely structurally-bound representations, although we realise the debate that this view might trigger.

7. OTHER VIEWPOINTS

We believe that the viewpoints listed in the previous sections can infer valuable architectural views for most kinds of systems. Some other viewpoints, however, become useful when dealing with embedded telecommunications systems in general, and mobile handsets in particular.

The participants of the aforementioned ECOOP workshop agreed that the number of views should be as low as possible. We hereby propose three specific viewpoints that we find useful in our specific organization and domain. The choice to adopt such viewpoints arises mainly from company-specific modelling needs. Further research is needed to establish whether the viewpoints we use would suffice also in other domains or in other organizations working in the same domain. Also, as this section constitutes a summary of our best practice (rather than the result of research work), we are aware of the probable incompleteness of our viewpoint set. As usual in a practical setting, it represents a good compromise between rigor and practical applicability.

7.1 Dynamic (runtime) viewpoint

Use cases typically model functional requirements based on user goals [Cockburn]. Architecturally significant functional requirements for our mobile handset software often span through several (unrelated) user goals, and therefore cannot be modelled with use cases. This brings up the need for feature description models, a topic that has been overlooked in the literature.

When dealing with features, it is important to model the user-visible behaviour of the system when seemingly unrelated features interact, for example by means of interruption, blocking or dependency. The topic of feature interaction has been subject of previous research [Lorentsen].

Feature interaction is present 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. Modelling such interaction patterns is useful to generate complete system-level test cases.

An example from the mobile phone domain is the one where an incoming call is signalled while the user is playing a game. In this case, the game is interrupted, so that the user can handle the incoming call (for instance, by answering, diverting or rejecting it). In the meantime, the state of the game must be saved, so that playing can be resumed after the incoming call handling is terminated. This scenario has to do with two unrelated user goals: "play a game" and "handle an incoming call", which correspond to two different use cases.

The case is interesting, since an incoming call can interrupt almost all the activities that can take place in a mobile phone, at almost all points in time. Therefore, it is not feasible (and definitely not convenient) to model all the combinations. Instead, modelling effort should focus on patterns of interaction, which should be linked to the main combinations of user goals. In the incoming call case, for instance, the interruption does not always happen in the same way. A slightly different case is when an incoming call interrupts an ongoing call and is put in wait. The user can choose whether to handle the interruption or ignore it, a possibility he doesn't have when playing games.

In order to address this modelling need, we created a dynamic (runtime) viewpoint, in which we model the interruption priorities and patterns, the blocking rules and the dependencies between the various system features.

The dynamic view wherefrom inferred is all about requirements, and thus may be included in the requirements view. We prefer to keep it separate, since the need for this kind of view does not exist in other domains.

7.2 Task viewpoint

The view derived from the task viewpoint models the allocation of components into operating system tasks. It constitutes part of the process view in the 4+1 view model. We felt the need to isolate the particular problem of task allocation because of the stringent real-time requirements that our systems must fulfil (partly because of international standards on, for example, maximum call establishment time). We will not elaborate any further on the task viewpoint, as it is not the focus of this paper.

7.3 Organizational viewpoint

The organizational viewpoint models the structure of the organization that is in charge of software development. While it is trivial in most cases, when working in large, multi-site organizations such as Nokia it is worth some more attention.

Conway's law [Conway] asserts that the structure of a software system mirrors that of the developing organization. Experience and wisdom confirm that it holds a fair amount of truth, especially for large, multi-site organization such as Nokia. In particular, decisions on architectural evolution should be made while keeping in mind the organizational boundaries, since it is usually the case that components and subsystems that are developed in different sites are designed and architected differently, and should be as loosely coupled as possible so as to be able to evolve independently.

Just like the requirements and dynamic view, the organizational view does not model the system architecture directly, but helps understand the context in which the system is developed.

8. CONCLUSION AND FURTHER RESEARCH

We believe that the investigation of the following issues would be very useful for the activity of architecting software to enhance its evolution:

a) Architectonic domain modelling, which aims at producing a model of how the problem domain in its wider sense, which also includes the software development organisation, influence or dictate the evolution paths of the software in question. One of the authors is currently engaged in a UK government-funded project to carryout this work. The aim is to explore the extent to which the architectonics of a given domain influence or dictate the architecture of the software artefact serving it [Addis & Galal]. Within the CAFÉ project [CAFÉ], Nokia plans to explore several areas pertaining to architectonics for product families, including domain modelling and assessment for architectural evolution.
b) The historical impact of legacy architecture on the ease, or otherwise, of evolving software to accommodate new requirements. This should be used to inform further software architecting or re-architecting decisions.
c) Investigating the impact of the organisational view and making it explicit and subject to debate and re-organisation when necessary.
d) Seemingly, the objectives of producing and maintaining architectural views vary between organisations. No single architectural view can address all such objectives. At the same time, the proliferation of architectural views is probably harmful since this can increase complexity (thus defeating the main purpose of having architectures in the first place: to simplify the management of complexity), as well as increasing the opportunities for inconsistencies. This calls for a conscious effort by the software development organisation to state its prime objectives from having architectural representations, and prioritising these. Thus the prioritisation and stratification of the architectural views, perhaps as a hierarchy of constraints, to rationalise and re-organise when necessary becomes important. Thus developing a kind of meta-architecture framework, where the architecture of the constellation of architectural views in an organisation is explicitly addressed, modelled and managed.
e) It is vital to develop representations that can support reasoning and debate about the architectonic attributes of the problem domain and the software that aims to address it. Such representations need to be both

open to all stakeholders and meticulously maintained and version-managed to aid reasoning about past evolution behaviour of systems.

f) More research effort should go into studying the histories of domains and corresponding software over a period of time. This can be the start of global effort to classify domains, software and systems according to aspects of architectonic behaviour and profiles. However, this is a large research programme that requires substantial commitment and resources.

g) We asserted that the number of architectural views that are used to describe a certain system should be as small as possible. However, in our examples we realized it was necessary to add at least three domain-specific viewpoints. This makes for a large number of views in the architectural document. The research question whether there should be a maximum limit to the number of views should be addressed. From anecdotal evidence, it appears that the number of views depends on the domain, but more research is needed on this point.

h) It is necessary to find out methods how the cross-effect of layers upon each other can be investigated and brought under greater discipline.

i) Perhaps the biggest challenge concerning the architectonic viewpoint is its practical validation. While at Nokia we have started using it in an experimental way, several other trials are necessary before its usefulness is proved. Ideally, such trials would come from both toy case studies (e.g. during university courses) and from real cases, ideally extending to different domains than ours. Early results from the study of other real-life systems suggest the utility of the architectonic view in isolating software components by layers that differ in their rates of change.

9. ACKNOWLEDGEMENTS

Some of the work that resulted in this paper was funded by the EU project CAFÉ (Σ! EUREKA 2023 / ITEA - ip00004) and by the EPSRC (Engineering and Physical Sciences Research Council, England).

10. REFERENCES

[Addis & Galal] T. Addis & G. H. Galal, *Using problem-Domain and Artefact-Domain Architectural Modelling to Understand System Evolution.* 9th European Conference on Information Systems, Bled, Slovenia June 27-29, 2001. Pp 298-303.

[Brand]: S. Brand, *How buildings learn: What happens after they're built.* 2nd ed. 1994, London: Phoenix Illustrated.

[CAFÉ]: "from Concepts to Applications in System Family Engineering" (Σ! EUREKA 2023/ ITEA - ip00004), see http://www.extra.research.philips.com/euprojects/cafe/

[Cockburn]: A. Cockburn, Structuring Use Cases with Goals, Journal of Object-Oriented Programming, September 1997 (part 1) and November 1997 (part 2).

[Conway]: see the Free Online Dictionary of Computing: http://burks.bton.ac.uk/burks/foldoc/18/25.htm

[ECOOPworkshop]: Fourth International Workshop on Object-Oriented Architectural Evolution, co-located with the 15th European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary June 2001. See http://prog.vub.ac.be/OOAE/

[Fayad]: M. Fayad Accomplishing Software Stability, Communications of The ACM, 2001 Vol. 45, No. 1, pp. 111-115.

[Frampton]: K. Frampton and J.e. Cava, *Studies in Tectonic Culture - The Poetics of Construction in Nineteenth and Twentieth Century Architecture*. 1995, Cambridge, Massachusetts: The MIT Press.

[Galal & Paul]:G. Galal, & Paul, R. J. Systems Architectonics. Mini-track on the Philosophical Foundations of Information Systems. In W.D. Haseman & D. L. Nazareth (Eds.) Proceedings of the Fifth Americas Conference on Information Systems (AMCIS'99) August 13-15, 1999, University of Wisconsin-Milwaukee, Milwaukee, WI, USA. pp 627-629

[Galal99]: G. H. Galal, On the Architectonics of Requirements, the Requirements Engineering Journal, Viewpoints, 1999, Vol 4, No. 3, pp 165-167.

[Galal2000]: G. H. Galal, Software Architectonics: Towards a Research Agenda, 2nd Workshop on Object-Oriented Architectural Evolution. In Object-Oriented Technology (ECOOP'2000: 14th European Conference on Object-Oriented Programming), Sophia-Antipolis and Cannes, France, June 12-16, 2000.

[IEEE]: see http://standards.ieee.org/ A thorough discussion of this standard is at: http://www.incose.org/delvalley/Hilliard_11_14_00.pdf

[ICSEworkshop]: International Workshop on Describing Software Architecture with UML, co-located with the 23rd International Conference on Software Engineering (ICSE), Toronto (CA), May 2001. See: http://www.rational.com/events/ICSE2001/index.jsp

[Jackson]: M. Jackson, *Software Requirements and Specifications, a lexicon of principles, practice and prejudices,* Addison-Wesley, 1995.

[Kruchten]: P. Kruchten, Architectural Blueprints – The 4+1 View Model of Software Architecture, IEEE Software, November 1995, 12 (6), pp.42-50.

[Lorentsen]: L. Lorentsen, A-P. Tuovinen, J. Xu, Modelling Feature Interactions in Mobile Phones, presented at the Workshop on Feature Interaction in Composed Systems, co-located with the 15th European Conference on Object Oriented Programming (ECOOP 2001), Budapest, Hungary, June 2001.

[Riva]: C. Riva, J. Xu, A. Maccari, Architecting and Reverse Architecting in UML, presented at the International Workshop on Describing Software Architecture with UML, co-located with the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, 15 May 2001.

[Symbian]: Symbian Ltd., see http://www.symbian.com/