Software Architecture Knowledge Management

Hans van Vliet
Department of Computer Science
VU University Amsterdam
The Netherlands
hans@cs.vu.nl

Abstract

Software architecture is a recognized and indispensable part of system development. Software architecture is often defined in terms of components and connectors, or the "high-level conception of a system". In recent years, there has been an awareness that not only the design itself is important to capture, but also the knowledge that has led to this design. This so-called architectural knowledge concerns the set of design decisions and their rationale. Capturing architectural knowledge is difficult. Part of it is tacit and difficult to verbalize. Like developers, software architects are not inclined to document their solutions. Establishing ways to effectively manage and organize architectural knowledge is one of the key challenges of the field of software architecture. This architectural knowledge plays a role during development, when architects, developers, and other stakeholders must communicate about the system to be developed, possibly in a global setting. It also plays a role during the evolution of a system, when changes are constrained by decisions made earlier.

1. Introduction

Software architecture is a recognized and indispensable part of system development. Until recently, research and industry primarily considered software architecture as the high-level design, captured in terms of components and connectors. Such a software architecture is usually represented in different views (most often some kind of box-and-line diagram), where each view highlights certain concerns of certain stakeholders. This is akin to the way different drawings are used in house construction: one for the electrical wiring, one for the water supply, etc. A well-known set of architectural views is the '4+1 model', now part of RUP [15]. One of its views is the logical view, which describes a system in terms of major design elements and their interactions. This is the traditional representation of a global design. Another view from this model is the deployment view, which contains the allocation of tasks to physical nodes. When modeling and representing a software architecture this way, emphasis is on modeling and representing the *solution* aspects.

In a software architecture, the global structure of the system is decided upon. This global structure captures the early, major design decisions. It might thus be useful to not only capture the solution chosen, in terms of components and connectors, but also the decisions, including their rationale, that led to that solution. These early design decisions and their rationale are important since their ramifications are felt in all subsequent phases. Going one step further, we may even claim that a software architecture *is* the set of design decisions [3].

If architecture is the set of design decisions, then documenting the architecture boils down to documenting the set of design decisions. This is generally not done, though. We can usually get at the *result* of the design decisions, the solutions chosen, but not at the reasoning behind them. Much of the *rationale* behind the solutions is usually lost forever, or resides only in the head of the few people associated with them, if they are still around.

So the reasoning behind a solution is not explicitly captured. This is implicit knowledge, essential for the solution chosen, but not documented. At a later stage, it then becomes difficult to trace the reasons of why things are the way they are. Architectural decisions are like material floating in a pond. When not touched for a while, they sink and disappear from sight. These sunken decisions are the most difficult to change at a later stage. In particular, during evolution one may stumble upon these design decisions, try to undo them or work around them, and get into trouble when this turns out to be costly if not impossible.

The shift in attention from a solution-oriented view to a decision-oriented view of software architecture has spawn new lines of research. In this paper, I discuss several research areas we are working on in the context of the GRIF-

1530-0803/08 $25.00 © 2008 IEEE
DOI 10.1109/ASWEC.2008.69

Element	Description
Decision topic	How can we measure the distance that a user drove on a route for which payment is required?
Decision	Use of SmartBoxes. Dutch cars have a SmartBox built in, drivers of non-Dutch cars can rent one at the border.
Status	Approved.
Assumptions	a) The system must also fee drivers of non-Dutch cars b) The Ministry of Finance bears a large fraction of the cost of SmartBoxes c) Camera's have an accuracy of at most 98%
Alternatives	a) Vignettes for non-Dutch cars b) The use of camera's at highway entries and exits, instead of SmartBoxes
Rationale	With the on-board SmartBox, the accuracy of the system is larger than what can be achieved with camera's. It also is easier to treat Dutch and non-Dutch cars the same way. Using GPS, it is easier to extend the system to other roads besides highways.
Implications	Non-Dutch cars need to be equiped with a SmartBox too. They can be rented against a reasonable deposit.
Related decisions	How to detect fraud

Table 1. Example of a design decision

FIN project[1]. First, architectural design decisions have much in common with requirements. Adequate support for eliciting, capturing, and managing architectural design decisions may thus profit from results in the field of requirements engineering, and vice versa. Second, architectural design decisions are a form of knowledge. Supporting software architects in their daily work thus becomes a knowledge management issue, and tool support in this area resembles knowledge management tool support. Finally, a specific usage of architectural knowledge occurs in the context of multi-site development, where it can be used to alleviate some of the challenges of global software development.

2 What's in a Design Decision?

Software design is a 'wicked problem'. Solutions to wicked problems are not true or false. At best, they are good or bad. The software design process is not analytic. It does not consist of a linear sequence of design decisions each of which brings us somewhat closer to that one, optimal solution. Software design involves making a large number of trade-offs, such as those between speed and robustness. As a consequence, there is a number of *acceptable* solutions, rather than one best solution. Wicked problems also have no stopping rule. There is no criterion that tells us when *the* solution has been reached, i.e., when *all* decisions have been taken. Finally, every wicked problem is a symptom of another problem. Resolving one problem may very well result in an entirely different problem elsewhere: taking a design decision incurs new problems that require additional decisions.

[1] `http://griffin.cs.vu.nl`

A design decision addresses one or more issues that are relevant for the problem at hand. There may be more than one way to resolve these issues, so that the decision is a choice from amongst a number of alternatives. The particular alternative selected preferably is chosen because it has some favorable characteristics. That is, there is a rationale for our particular choice. Finally, the particular choice made may have implications for subsequent decision making.

The type of information we would like to capture for each design decision directly follows from the above. Table 1 gives an example of a design decision for an electronic billing system for road usage. It concerns the choice for using a SmartBox, an On Board Unit that receives coordinates through a GPS system and transmits those coordinates together with vehicle and timing information to a signal receiver. The template is derived from [21].

Design decisions are often related. The example decision from Table 1 is related to one or more decisions about how to detect fraud. For instance, we may decide to have fixed and/or mobile cameras for this purpose. Generally, a given design decision may constrain further decisions, exclude or enable them, override them, be in conflict with them, and the like [16]. These relationships between design decisions resemble the kind of relationships that may exist between requirements. And likewise, the notations and tools used to capture this information are very similar as well; see also section 3.1.

It is an illusion to want to document all design decisions. There are far too many of them, and not all of them are that important. Unfortunately, whether a design decision is major or not really can only be ascertained with hindsight, when we try to change the system. Only then will it show which decisions were really important. A priori, it

is often not at all clear if and why one design decision is more important than another [13]. For instance, separating the user interface from the processing part is generally considered good design. If, at a later stage, changes occur in either part, we will be glad to have made this decision. If no such changes occur, the decision was not all that important, after all. The on-board SmartBox of our road pricing system transmits timing information together with GPS coordinates. This timing information may be used at a later stage if we want to extend the system with a feature to handle speed violations. But again, if no such extension occurs, the decision wasn't that important either. Worse even, we may then consider it a bad decision with hindsight, for example because of privacy concerns.

Experienced software architects and designers have at their disposal a vast number of useful knowledge chunks. These knowledge chunks are called in when studying or developing an architecture. The more experienced an architect is, the more useful chunks he has at his disposal. He will choose one that fits the current situation. He will, of his own, generally not build up and document a complete design space.

Software architects often operate in an opportunistic mode. If they know a feasible solution, they apply it, and they then do not consciously consider possible alternatives. In particular, they do not search for optimal solutions, but for satisfactory ones [22].

In [10], we analyzed the complete design space of an electronic commerce system. In the end, three feasible solutions remained. For each of these feasible solutions, trade-offs had to be made, and certain requirements had to be relaxed on the way to the final solutions. After the exercise, we confronted an experienced architect in the domain of electronic commerce with our analysis. He told us he knew these three solutions existed. But he also told us he did not (anymore) know of the trade-offs made on the way to these solutions. This knowledge had vaporized.

We noted earlier that many design decisions remain undocumented. There are different types of undocumented design decisions:

- The design decision is implicit: the architect is unaware of the decision, or it concerns 'of course' or tacit knowledge. Examples include earlier experience, implicit company policies to use certain approaches, standards, and the like.
- The design decision is explicit but undocumented: the architect takes a decision for a very specific reason (e.g. the decision to use a certain user-interface policy because of time constraints). The reasoning is not documented, and thus is likely to vaporize over time.
- The design decision is explicit, and explicitly undocumented: the reasoning is hidden. There may be tactical company reasons to do so, or the architect may have personal reasons (e.g. to protect his position).

So, explicitly documenting all design decisions will not easily become a reality. But we may try to document the really important ones. And supporting technology should provide the necessary incentives.

3 The GRIFFIN project

GRIFFIN is a joint research project of the VU University Amsterdam and the University of Groningen, both in the Netherlands. The research is carried out in a consortium with various industries, both large and small. These industries provide us with case studies and give regular feedback. The domains of these case studies range from a family of consumer electronics products to a highly distributed system that collects scientific data from around 15,000 sources to a service-oriented system in a business domain.

The GRIFFIN project develops notations, tools and associated methods to extract, represent and use architectural knowledge that currently is not documented or represented in the system. In GRIFFIN, Architectural Knowledge is defined as the integrated representation of the software architecture of a software-intensive system) or a family of systems), the architectural design decisions, and the external context/environment. The project emphasizes sharing architectural knowledge in a distributed, global context. Some of the results so far can be found in [4, 5, 6, 8, 9, 12, 14].

Typical research questions addressed within GRIFFIN are:

- What knowledge does an architect use or need?
- What types of assumptions are made in an architecture?
- Which types of decisions are important to capture explicitly?
- How to "extract" important architectural knowledge?
- How to collect important architectural knowledge for later reuse and analysis?
- How does an architecture knowledge ontology look like?
- How to share architectural knowledge in a distributed setting?
- How to describe architectural knowledge?
- How to manage architectural knowledge?

In the following subsections, I discuss three research areas we are working on in the context of the GRIFFIN project:

- The type of architectural knowledge to be captured. This leads to a discussion on the nature of architectural design decisions and their relation to software requirements.
- The use of architectural knowledge in organizations, in particular development organizations that operate in a global setting.
- The management of architectural knowledge, and the type of tool support required to make effective management of architectural knowledge a reality.

3.1 Architectural Design Decisions versus Requirements

We have constructed a core model of architectural knowledge: a reference model in which different specifications of what architectural knowledge entails can be expressed [8]. This model describes among others a 'decision loop' that relates individual architectural design decisions. With this decision loop, we can discuss the relationship between requirements and architectural design decisions.

In the road pricing example, one of the requirements could be that the cost of a trip should depend on the distance driven. This requirement asks for one or more decisions to be taken. That is, the requirement introduces one or more *decision topics*, most notably 'how to measure the distance driven'. This decision topic can be addressed through various alternatives, such as SmartBoxess or toll gates along roads. When one of these alternatives is chosen, for instance SmartBoxes as in the example in Table 1, it becomes an *architectural design decision*, which immediately leads to new decision topics ('how to detect fraud' – after all, someone may decide to not install a SmartBox, or even remove the one installed).

An architectural design decision (often seen as a 'how') for a requirement (a 'what'), inevitably introduces new 'whats'. If a 'what' is 'cost should depend on the distance driven' and the 'how' is 'by means of a SmartBox' then one of the new 'whats' is 'fraud should be detected'. In this decision loop, concerns lead to decisions which in turn lead to new concerns. The issue of fraud detection could just as well be regarded a requirement as an architectural design decision, depending on who you ask. The issue may be coined by the architect as a consequence of the decision to use SmartBoxes. The issue may equally well crop up in requirements discussions about policy enforcements for this type of system.

That 'how' questions do not only play a role in architecture, but also in requirements engineering, is illustrated by an approach called 'goal-oriented requirements engineering', or GORE [17]. Given a requirement, by asking *why* that requirement is needed, higher level requirements and goals can be detected. The other way around, by asking *how* a goal or requirement can be satisfied, new — lower level — requirements can be identified. In this way a hierarchical structure of requirements and goals emerges. In GORE, the decision loop described above appears in a requirements engineering context. Or, in the words of Van Lamsweerde, "quality goals are used to compare alternative options and select preferred ones" [18]. This is remarkably similar to how concerns are used to rank alternative solutions and choose the eventual design decision in the decision loop from our architectural knowledge theory [8].

Architectural design decisions and requirements thus are closely related. One might even say that, depending on your point of view, you could regard architectural design decisions as requirements, or architectural significant requirements as architectural design decisions[2].

The similarity of requirements and architectural decisions also means that the fields may profitably use each other's methods and techniques. This seems especially promising in the area of management. Management of requirements has already received considerable attention in the requirements engineering community. More recently, the same can be observed in the architecture field. Both fields have to do with regarding the collection of requirements/architectural decisions not as a mere accumulation of statements, but as an information repository. Both fields aim to impose a structure on this collection that captures the connections (traces) between individual requirements or architectural design decisions.

3.2 Software Architecture and Multi-site Development

The essence of a collocated team is that many things are shared. The team shares some set of tools and uses the same process. Team members have the same background and share an understanding of the job to be done. Team members share information, amongst which architectural knowledge, at frequent informal meetings in the lobby or at the coffee machine. Perry et al [20] observed that developers in a particular organization spent up to 75 minutes per day in unplanned personal interaction. Sometimes, development teams share the same office area on purpose, to stimulate interaction and speed things up. A shared room for instance is one of the practices of agile development.

These mechanisms all disappear when work is distributed. The challenges that face global software development all have to do with distance: temporal distance, geographical distance, and socio-cultural distance. The challenges of global software development have to do with [1]:

[2]Not all requirements are architecturally significant. Specifically, many functional requirements are not.

- communication and collaboration between team members,
- coordination between tasks, and
- control of the work.

Part of overcoming these challenges has to do with adequate knowledge management, including the management of architectural knowledge. A lot of knowledge of a software development organization is kept in unstructured forms: FAQs, mailing lists, email repositories, bug reports, lists of open issues, and so on. Lightweight tools like wikis, weblogs, and yellow pages are other examples of relatively unstructured repositories to share information in global projects.

In the knowledge management literature, a distinction is often made between the *personalization strategy* and the *codification strategy*. The personalization strategy emphasizes interaction between knowledge workers. The knowledge itself is kept by its creator. One personalization strategy is to record who knows what, as in yellow pages. Each person then has his own way to structure the knowledge. The threshold to participate is usually low, but the effort to find useful information is higher. In the codification strategy, the knowledge is codified and stored in a repository. The repository may be unstructured, as in wikis, or structured according to some model. In the latter case, the structure of the repository can be used while querying. An advantage of a structured repository is that the information has the same form. A disadvantage is the extra effort it takes to cast the information in the form required. A hybrid strategy may be used to have the best of those different worlds [11, 2].

Part of the relevant knowledge concerns the architecture. Not only is the architecture important information to be shared between sites in a global development project, the architecture may also serve as kind of a proxy for communication, coordination, and control tasks. One may claim that less informal meetings are needed when all is neatly documented in the architecture documentation. Similarly, coordination is regulated by specifying allowed dependencies between subsystems in the architecture.

So the software architecture can be used to reduce the need or communication in a multi-site development project. Typically, people that have to interact a lot are located at the same site. Such not only reduces the need for communication, but also makes the coordination of work a lot easier. For instance, experts in a certain area, such as user interfaces, are put together at the same site. User interfaces for all systems of an enterprise then are developed at that site. Alternatively, the gross structure of the system, the software architecture, can be used to divide work amongst sites. This can be seen as a variation of Conway's Law [7] which states that the structure of a product will mirror the organization structure of the people who designed it. If three groups are involved in the development of a system, that system will have three major subsystems. This way of decomposing work is the programming-in-the-large equivalent of Parnas' advice to consider a module as a "responsibility assignment rather than a subprogram" [19]. One may thus study what architecture fits a certain global setting or, the other way around, what global organization is required to successfully realize a given architecture.

In [5], we studied how two different organizations used architecture to overcome some of the challenges in global software development. One organization heavily relied on so-called architectural rules: principles and statements about the software architecture that must be complied with throughout the organization. The other organization emphasizes process measures such as periodic travel of key individuals and frequent highly communicative meetings. The study shows that a heavy reliance on formal architecture is not a panacea: especially cultural challenges and team collaboration challenges are not addressed this way.

3.3 Software Architecture Knowledge Management

Software architecting is a knowledge-intensive process, involving many different stakeholders. As the size and complexity of systems increases, more people get involved, and architecting means collaboration. There often is not a single all-knowing architect. Instead, this role is fulfilled by more than one person. To be able to take well-founded decisions, all stakeholders need to have the relevant architectural knowledge at the right place, at the right time. Sharing architectural knowledge is crucial, in particular for reusing best practices, obtaining a more transparent decision making process, providing traceability between design artifacts, and recalling past decisions and their rationale.

In [12], we describe a prototype support suite for architectural knowledge sharing. The suite provides services to share both codified knowledge and personalized knowledge. For knowledge that is not subject to frequent changes, a codification strategy is useful. Architects may then retrieve proven solutions and reuse them. On the other hand, much architectural knowledge is not stable until consensus has been reached. For such knowledge, a personalization strategy is useful. It enables architects to find who knows what.

Architects take decisions and develop solutions. But experience shows that they do not naturally document their decisions, let alone codify relevant architectural knowledge. If we still aim to support architectural knowledge sharing, we have to find other ways to achieve our goals. We are currently following two roads to bridge this gap between what

architects are inclined to invest and what an architectural knowledge support suite might offer them:

- develop services that enrich unstructured information to obtain codified knowledge,
- obtain the codified knowledge as a side effect of the normal architecting activities.

We explain the former through an example.

Suppose two architects working on the architecture of our road pricing system discuss different alternatives for measuring the distance a user drives. They may use a blog for their discussion. The unstructured documentation thus collected is not very useful to most other stakeholders involved, but a structured summary is. We may conceive of a text mining service that at some point in time opportunistically analyzes the blog discussion and determines that a decision on distance measuring has been taken. A summary of this architectural knowledge is captured and sent to the lead architect using an RSS feed.

This way, unstructured information is enriched and becomes codified knowledge. In a similar vein, the semantic structure of a collection of unstructured documents can be grasped using a technique called Latent Semantic Analysis (LSA). In [9], we applied LSA to transform a set of texts into a collection that contains architectural knowledge elements and the intrinsic relations between them. The basic idea behind LSA can be explained as follows: if the terms 'architecture' and 'system' often occur in the same document, then another document, in which the term 'system' does occur, but the term 'architecture' does not, probably is about architecture as well. LSA has for instance been applied in the area of programming, where programmers get suggestions as to how to continue coding of a method based on the statements entered so far and knowledge of a large collection of other methods. This is similar to suggestions made by Amazon and similar sites: "Readers who ordered 'The State of Africa' by Martin Meredith also ordered 'The White Man's Burden' by William Easterley". We may conceive of architectural advice based on an LSA-type analysis of previous architectural documents: "architects confronted with built-in devices also have taken fraud prevention measures".

A second way to obtain codified knowledge is as a side effect of "normal" architecting activities, such as writing an architectural description. As an example, consider the way archiving used to work at a typical government agency in the past, and how it works nowadays. In the past, documents prepared by professionals were eventually moved to the basement of the government agency, where archiving clerks manually added archiving codes to those documents. Nowadays, professionals use a computerized system, and the archiving codes are added "automatically", under the hood of the system, without the professionals being aware that in fact they do the archiving themselves. Our aim is to provide similar support to software architects.

4 Conclusions

In this paper, I have discussed the shift in attention from a solution-oriented view to a decision-oriented view of software architecture. This shift has spawn new research challenges. Some of these are being worked on in the context of the GRIFFIN project.

Architectural design decisions have much in common with requirements. Adequate support for eliciting, capturing, and managing architectural design decisions may thus profit from results in the field of requirements engineering, and vice versa. The precise relation between the two fields needs further study.

Architectural design decisions are a form of knowledge. Supporting software architects in their daily work, in a collocated as well as a global setting, thus becomes a knowledge management issue, and tool support in this area resembles knowledge management tool support. The most challenging issue here seems to be to provide for effective support without requiring that architects pay massive amounts of time codifying what they do. Mining architecture artifacts seems one promising area to pursue.

Acknowledgements

This research has been partly sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. I thank my fellow researchers in the GRIFFIN projects for their contributions: Paris Avgeriou, Remco de Boer, Viktor Clerc, Rik Farenhorst, Anton Jansen and Patricia Lago.

References

[1] P. Ågerfalk and B. Fitzgerald. Flexible and Distributed Software Processes: Old Petunias in New Bowls? *Commun. ACM*, 49(10):27–34, 2006.

[2] M. Ali Babar, R. de Boer, T. Dingsoyr, and R. Farenhorst. Architectural Knowledge Management Strategies: Approaches in Research and Industry. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge (SHARK-ADI07)*. IEEE Computer Society, 2007.

[3] J. Bosch. Software architecture: The next step. In *Software Architecture, First European Workshop (EWSA)*, pages 194–199. Springer, LNCS 3047, 2004.

[4] V. Clerc, P. Lago, and H. van Vliet. Assessing a Multi-Site Development Organization for Architectural Compliance. In *Proceedings 6th Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 2007.

[5] V. Clerc, P. Lago, and H. van Vliet. Global Software Development: Are Architectural Rules the Answer? In *Proceedings Second IEEE International Conference on Global Software Engineering (ICGSE 2007)*, pages 225–234. IEEE Computer Society, 2007.

[6] V. Clerc, P. Lago, and H. van Vliet. The Architect's Mindset. In *Proceedings QoSA 2007*. Springer, LNCS 4880, 2007.

[7] M. Conway. How Do Committees Invent. *Datamation*, 14(4):28–31, 1968.

[8] R. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen. Architectural Knowledge: Getting to the Core. In *Proccedings QoSA 2007*. Springer, LNCS 4880, 2007.

[9] R. de Boer and H. van Vliet. Constructing a Reading Guide for Software Product Audits. In *Proceedings 6th Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 2007.

[10] H. de Bruin, H. van Vliet, and Z. Baida. Documenting and Analyzing a Context-Sensitive Design Space. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development and Maintenance, Proceedings 3rd Working IFIP/IEEE Conference on Software Architecture*, pages 127–141. Kluwer Academic Publishers, 2002.

[11] K. Desouza, Y. Awazu, and P. Baloh. Managing Knowledge in Global Software Development Efforts: Issues and Practices. *IEEE Software*, 23(5):30–37, 2006.

[12] R. Farenhorst, P. Lago, and H. van Vliet. EAGLE: Effective Tool Support for Sharing Architectural Knowledge. *International Journal of Cooperative Information Systems*, 16(3/4):413–437, 2007.

[13] M. Fowler. Who Needs an Architect. *IEEE Software*, 20(5):11–13, 2003.

[14] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Using Architectural Decisions in Practice. In *Proceedings 6th Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society, 2007.

[15] P. Kruchten. *The Rational Unified Process, An Introduction*. Addison-Wesley, third edition, 2003.

[16] P. Kruchten, P. Lago, and H. van Vliet. Building up and Reasoning about Architectural Knowledge. In C. Hofmeister, I. Crnkovic, and R. Reussner, editors, *Quality of Software Architectures, Proceedings 2nd International Conference*, pages 43–58. Springer, LNCS 4214, 2006.

[17] A. v. Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings 5th International Symposium on Requirements Engineering (RE'01)*, pages 1–13. IEEE, 2001.

[18] A. v. Lamsweerde. Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In *Proceedings 12th IEEE International Requirements Engineering Conference*, pages 4–7. IEEE, 2004.

[19] D. Parnas. On the Criteria to be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[20] D. Perry, N. Staudenmayer, and L. Votta. People, Organizations, and Process Improvement. *IEEE Software*, 11(4):36–45, 1994.

[21] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[22] C. Zannier and F. Maurer. Social Factors Relevant to Capturing Design Decisions. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge (SHARK-ADI07)*. IEEE Computer Society, 2007.

Dr. Hans van Vliet is Professor in Software Engineering at the VU University, Amsterdam, The Netherlands, since 1986. He got his PhD from the University of Amsterdam. His research interests include software architecture and empirical software engineering. Before joining the VU University, he worked as a researcher at the Centrum voor Wiskunde en Informatica (Amsterdam). He spent a year as a visiting researcher at the IBM Almaden Research Center in San Jose, California. He co-authored over 100 refereed articles. He is the author of “Software Engineering: Principles and Practice", published by Wiley (3rd Edition, 2008). Together with Paul Clements, he is the editor of the Software Architecture Session of the Journal of Systems and Software. He is a member of IFIP Working Group 2.10 on software architecture.