A Community of Learners Approach to Software Architecture Education

Remco C. de Boer, Rik Farenhorst, and Hans van Vliet
Dept. of Computer Science
VU University Amsterdam, the Netherlands

E-mail: [remco, rik, hans]@cs.vu.nl

Abstract

The wicked nature of software architecture calls for educational methodologies that deviate from the traditional active lecturer-passive student relation. In this paper we present our experiences with applying a Community of Learners approach, in which students are treated as partners in the knowledge development process, to software architecture education. In our course the students are actively involved, either as architects or as stakeholders, in the design of an architecture for a large and realistic system. In this Community of Learners, the students not only learn about but also experience the important issues in software architecture. Students indicate that, although the workload is high, the required effort is outweighed by the learning effect of their active involvement.

1 Introduction

Software architecture design is a 'wicked problem'[16]. The term originated in research into the nature of design issues in social planning problems. Properties of wicked problems in this area are similar to properties of software architecture design:

- There is no definite formulation of a wicked problem. The architecture design process can hardly be separated from the preceding requirements engineering phase.
- Architecture design has no stopping rule. There is no criterion that tells us when *the* architecture is finished.
- Solutions to architecture design problems are not true or false. At best, they are good or bad. Architecture design involves making trade-offs, such as those between speed and robustness. As a consequence, there is a number of *acceptable* solutions, rather than one best solution.
- Architecture design problems do not have a well-defined set of potential solutions. Software architects cannot fall back on a set of ready-made solutions, but have to apply tactics, practices and creativity to arrive at a satisfactory solution.
- Every architecture design problem is a symptom of another problem. Solving one problem may very well result in an entirely different problem elsewhere.

The wicked nature of software architecture calls for educational methodologies that deviate from the traditional active lecturer / passive student relation. In a Community of Learners, students are treated as partners in the knowledge development process [5]. They are not passive listeners in an auditorium, but are involved in a process of inquiry learning, a cyclic process consisting of (1) conducting research, (2) sharing the outcomes, and (3) drawing conclusions to be applied in a subsequent task. A basic underlying idea of the Community of Learners concept can be paraphrased as follows: you do not learn to play soccer by watching many soccer games on TV; you have to practice soccer instead.

In this experience report, we discuss our application of the Community of Learners concept to teaching software architecture. In an inquiry learning cycle, students develop an architecture, discuss their solution with other students, and usually find out their solution is not satisfactory yet. This stimulates them to reflect on the feedback obtained, and find alternatives.

978-0-7695-3539-5/09 $25.00 © 2009 IEEE
DOI 10.1109/CSEET.2009.10

IEEE computer society

2 Related Work

There is a great deal of literature on the Community of Learners approach. [5] is a seminal paper. Much of the literature is on applying these principles in primary or secondary schools, although Community of Learners principles can also be applied in higher education. In this paper, which is a sequel to [13], we incorporate lessons learned in our education of software architecture. Especially the focus on the Community of Learners concepts embodied has been considerably strengthened.

The Community of Learners approach is very similar to Problem-Based Learning [2]. A distinguishing aspect of the Community of Learners approach that makes it especially attractive in our case is the collaborative learning aspect. Student teams learn about architectural viewpoints by discussing the viewpoints other teams have chosen. They learn about architectural tradeoffs by discussing the tradeoffs other teams have chosen. And so on.

Jaccheri [12] describes a course given at the Norwegian University of Science and Technology (NTNU) in 2001. The goals for this course were similar to ours: generate architectural alternatives, describe an architecture accurately, evaluate an architecture. The course emphasized the influence of quality considerations on the architecture (by making performance-driven, maintenance-driven and usability-driven changes to the architecture), but did not emphasize the use of different architectural views.

Muller [15] discusses his experiences with teaching systems architecting. The course objectives partly overlapped with ours: raising awareness with the non-technical context in architecting, documenting and reviewing architectures. The course has been given 23 times to experienced people within Philips.

Shaw et al. [18] discuss a more general design course capturing many issues "that lie between the client's problem and actual software development". It has a wider focus than our software architecture course, and covers such issues as problem frame analysis, use case modeling, design, and analyzing business, economic and policy constraints. The overarching objectives of the course are "for the students to be able to explain the major forces, both technical and contextual, that shape and constrain the solutions to their clients' problems, to evaluate and address the ways these forces constrain the software implementation, and to select and apply appropriate techniques for resolving the constraints and selecting a preliminary design", and these are very similar to our's. The course involves real-life problems provided by real clients. Different student groups solve different problems and not, as in our case, alternative solutions to the same problem, but many Community of Learners concepts are present in their course as well. Bareiss and Griss [1] apply similar ideas to the complete software engineering curriculum.

Männistö et al. [14] also discuss a software architecture course. The major aim of the course was to be industrially relevant, and is given to relatively mature students. Topics treated are similar to ours, but Community of Learners aspects are not explicitly designed into their course.

3 What's Important in Software Architecture

In early publications, such as [17], software architecture was by and large synonymous with global design. In a broader view, software architecture involves making trade-offs between quality concerns of different stakeholders. As such, it becomes a balancing act reconciling the collective set of functional and quality requirements of all stakeholders involved, eventually resulting in an architecture that meets those requirements. This broader view is becoming the received view [3].

This broader view of what software architecture entails is also reflected in the characteristics of the architecture-centric software development life cycle. The characteristics of an architecture-centric life cycle are as follows:

- The discussions involve many different stakeholders.
- Each iteration concerns both functional and non-functional requirements.
- In particular, architecting involves finding a balance between these types of requirements.

In this view, software architecture has to bridge the gap between the world of a variety of, often non-technical, stakeholders on one hand – the problem space –, and the technical world of software developers and designers on the other hand – the solution space.

In the software architecture, the global structure of the system is decided upon. This global structure captures the early, major design decisions. Whether a design decision is major or not can really 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 [9]. Viewed this way, the architectural design process is about making the important design decisions. Next, these important design decisions need to be documented.

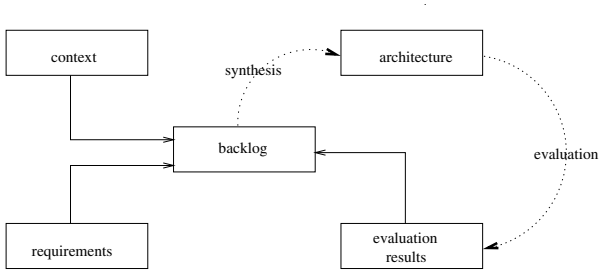

Figure 1. Global workflow in architecture design (adapted from [10])

3.1 Software Architecture Design

Architecture design methods give little guidance for the precise order and kind of development steps [10]. This is very much a matter of the architect's expertise. The global workflow common to these methods is depicted in Figure 1. At the center, the *backlog* is depicted. The backlog contains a list of issues to be tackled, open problems, ideas that still have to be investigated, and so on. In (architecture) design projects, the notion of a backlog is usually not represented explicitly. Yet, it is always there, if only in the head of the architect.

In each step of the architecting process, one or a few items from the backlog are taken and used to transform the architecture developed so far. The result of this transformation is evaluated (usually rather informally), and this evaluation may in turn change the contents of the backlog. New items may be added (for instance new problems), items may disappear or become obsolete, and the priorities of backlog items may change.

The architecture design process is very much driven by the architect's experience, much more so than by any of the so-called software design methods. Experienced architects *know* how to handle a given issue, rather than that some method tells them how to perform a design iteration. Design methods that are applied at the more detailed levels of design usually give much more guidance than those for architecture design methods. But this guidance is used by inexperienced designers mostly. Since architecture design is usually done by experienced designers, the amount of guidance given, and needed, is less. Attention then shifts to techniques for documenting the *result* of the design process: the decisions, their rationale, and the resulting design.

3.2 Architecture as a Set of Design Decisions

If architecture is the set of design decisions, then documenting the architecture boils down to documenting the set of design decisions. This is usually 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.

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. We use a template based on [19] for the type of information that is important to capture for each design decision.

3.3 Software Architecture Documentation

In discussions with stakeholders, architects use a description of the architecture which reflects the current set of decisions made, and how these address the concerns of the stakeholders. As with building plans, it is common practice to make different "drawings", each of which emphasizes certain concerns of certain stakeholders. In software architecture, this idea is put forward in the IEEE recommended practice for architecture description [11].

Central terms of reference in IEEE 1471 are 'views', 'viewpoints', 'stakeholders' and 'concerns'. An 'architectural description' consists of 'views' that are made according to a 'viewpoint'. A viewpoint prescribes the contents and models to be used in its views, and also indicates the intended 'stakeholders' and their 'concerns'. A stakeholder can have one or

more concerns, and concerns can be relevant to more than one stakeholder. Clements et al. [6] give many useful advices as to which views might be appropriate in certain circumstances.

3.4 Software Architecture Assessment

In the end, the stakeholders have to decide whether they are satisfied with the proposed architecture. A software architecture assessment is meant to do exactly this: assess to what extent the architecture meets the various concerns of its stakeholders [7]. It is conducted by one or a few assessors. Further participants are the architect(s) and the major stakeholders of the system. Very generally, the structure of such an assessment is as follows:

- The architects present the architecture and its rationale to the stakeholders. They highlight the major design decisions that led to the architecture. They may use different views of the architecture to illustrate their points.
- The stakeholders next devise a series of scenarios that best express their concerns. A maintainer may devise scenarios that describe possible changes or extensions to the system. A security officer may devise scenarios that describe possible threats to the system. And so on.
- For each of these scenarios, or a carefully selected subset if there are too many of them, the architects explain how the architecture fares with the situation described, and what changes are needed, and against which cost, to accommodate the situation described.
- The assessment team writes a report describing their findings.

4 Software Architecture Course

Students who participate in our software architecture course already possess a fair amount of software engineering knowledge; our course is part of the Computer Science Master's curriculum, and has a course in software engineering as prerequisite. With the background of our students and the important topics from Section 3 in mind, we have formulated four goals for our course:

1. The students should know how to develop different architectural views of an architecture, addressing specific concerns of stakeholders. We use [11] as the model for doing so.
2. The students should know of the *wicked* nature of software architecture [16]. A software architecture is never right or wrong, but at most better suited for certain situations. It involves making a large number of trade-offs between concerns of different stakeholders. There may be different *acceptable* solutions, and the solution eventually chosen depends on how the balancing between stakeholder concerns is made.
3. The students should know how to document design decisions and their rationale, for which we provide the template from [19] as a model. Design decisions should be visible in the views used to document the architecture. Decisions need to be discussed with the stakeholders. Ultimately, stakeholders must buy in on the major decisions made.
4. The students should know how to do an assessment of an architecture. This gives them the opportunity to learn and appreciate a set of architectural decisions and trade-offs made. This provides insight into the boundaries of the architectural solutions, the consequences for an architecture if another set of concerns had been chosen, as well as an overall impression of the quality of the architectural description. Since an assessment involves explaining to the stakeholders the architecture and the decisions that led to the architecture, this once again stresses the communication aspect of software architecture.

By letting students develop their own architectural viewpoints and views, and letting them decide which concerns to address, we obtain a series of different solutions to the same problem. This gives the students the opportunity to learn from different solutions, and appreciate these differences in terms of quality priorities set. It emphasizes the very nature of the intrinsic design-type problem.

4.1 The Software Architecture Course 2006-2007

In its present form, we have given this course four times. In the remainder of this paper we give examples from one particular course year to illustrate our approach to teaching software architecture. In the course that ran from late October 2006 to early February 2007, students were asked to develop an architecture for an electronic billing system for national public transport. The case description issued at the beginning of the course is given in Figure 2.

There were 35 students who participated in the course. They were organized into ten groups: six groups of architects and four groups of stakeholders. Two stakeholder groups (who named themselves *Translink* and *Public T*) represented the public transport consortium that provides the architect groups with functional and quality requirements. The other two stakeholder groups (called *The Patient* and *Huns*) represented social organizations that demand the new system has specific quality characteristics, e.g. regarding privacy and accessibility. Of the six architect groups, three groups interacted with *Translink* and *Huns*, while the other three architect groups interacted with *Public T* and *The Patient*.

Electronic billing for the national public transport system
A consortium of public transport companies wants to modernize the way in which financial transactions for public transport are carried out. They want to do this by replacing the paper tickets currently in use with electronic ones. This change introduces the need for a new, uniform, billing infrastructure in place at all public transport companies. Since the public transport companies do not have the required expertise, realization of the new billing system is delegated to a third party.
Some organizations look at this project with suspicion. Privacy organizations are concerned, especially about potential privacy implications of the new system. The government demands that accessibility of public transport is not negatively impacted by the new system.

Figure 2. Case description

Both the stakeholders and the architects had to fill in their own roles. The architects could get access to a lot of information about this particular system from newspaper clippings and the Internet. At the time we gave this course, this particular system was being implemented in the Netherlands. Delivery got delayed, of course. Since public transport is a national issue, politics entered the debate. The general public feared that prices would rise because of this new system. More recently, severe security and privacy issues have arisen. As of now, the system still has not been delivered.

Based on the case description and the type of stakeholder group (social organization or public transport consortium) they were assigned to, the stakeholders had to define their own roles. The stakeholders were in principle free to select any roles they liked, although most of them remained close to the example roles we provided. Nevertheless, this freedom for the stakeholders resulted in stakeholder groups of the same type having different needs and concerns. Hence the three architect groups who worked for one combination of stakeholder groups experienced different forces and constraints than the three other architect groups.

A particular point of attention for the stakeholders was that they had to make sure to have mutual conflicts. During the development of their architectures, architect groups had to handle those conflicts, and either take sides or reconcile them. In their architecture descriptions, their decision making and the options chosen had to be made visible. An example conflict was between the financial and marketing managers of *Public T*. The financial manager only wanted the essential features to be implemented, to save money. The marketing manager on the other hand wanted the system to gather all kind of information from travelers for marketing purposes. And of course, overall the public transport consortium stakeholders and the social organizations stakeholders had opposing interests.

In the first week of the course, the stakeholders picked a role and started gathering domain knowledge. In the week thereafter, the stakeholders augmented their roles by specifying their goals and tasks, the concerns they had in their roles, their needs, their relationships with other stakeholders, and the business and system quality attributes they deemed important. In the next two weeks, the stakeholders were interviewed by the architects and reported on those interviews, especially on the effects of the interviews on their roles (e.g. additions to the stakeholder roles because interviewers asked about things that were not yet in the stakeholder descriptions, changes in concerns due to new insights, etc.). Next, they prepared and executed an assessment of the architectures designed by the architect groups. At this point in time, the stakeholders had on purpose not yet been introduced to the theory of architecture evaluation. After the Christmas break, and after assessment theory and the ATAM assessment procedure had been introduced [3], the stakeholders prepared and executed another assessment. Their final report was a reflection on the course. Notice the role change of the stakeholder groups around the Christmas break. During the final weeks of the course, these groups had to take the lead in the architecture assessment round.

In the second week the architects were asked to design a preliminary architecture, based solely on the case description and their own insights and experiences. The week thereafter the architects interviewed their stakeholders with the aim of eliciting the information the stakeholders had written down in their profiles a week earlier. Based on the elicited information, the architects spent the next three weeks selecting viewpoints and describing a proper architecture related to the goals, needs, and concerns of their stakeholders, all the while going back to their stakeholders when necessary for clarification, follow-up

interviews, et cetera. Based on guidelines we provided them, the architects used the last week before the Christmas break to improve one of the views in their architecture description, while in the meantime they participated in the first assessment meeting led by their stakeholders. After the Christmas break, the architects participated in the ATAM assessment and performed two refinement iterations in which they addressed the results of the stakeholders' assessments of their architecture as well as any feedback on their work thus far they had received from us.

At our weekly two-hour lectures, generally half of the time was spent on some new theory, such as architecture design, architecture documentation, architecture assessment for which we mainly used [3]. The other half was filled by the students, who took turns in presenting their progress. Just before the end of the course, three architects from the organization that was actually implementing the system gave a guest lecture about their solution.

5 Mapping to Community of Learners concepts

In a Community of Learners, students are confronted with the *big ideas* from the field. In the case of our software architecture course, these big ideas are the ones discussed in Section 3: the wicked nature of software architecture, software architecture design, architectural decisions, architecture documentation, and architecture assessment. These topics are taught in class, and drive the weekly student assignments.

In a Community of Learners, students follow the cycle of *inquiry learning*, consisting of the following steps:

1. **Conduct research.** For example, in one assignment, architect groups have to define their architectural viewpoints. We require them to not only use technical viewpoints as listed in [3], but include a more business oriented viewpoint as well. In another assignment, stakeholder groups have to decide on the form of the assessment procedure.
2. **Share the outcome.** This occurs in class reporting, in discussions with stakeholder groups, and in weekly reports to the tutors. The viewpoints chosen by the architect groups have to be documented, presented in class, and communicated to their stakeholder counterparts. This sharing of results involves reflection and the drawing of conclusions. For instance, a viewpoint chosen may insufficiently capture the concerns of stakeholders. The architects then have to think of a better viewpoint, to satisfy the stakeholders. Likewise, one architect group might see that another architect group valued stakeholder concerns differently and as a result came up with a different set of viewpoints to document. Reflection brings knowledge to a higher level. Students look back to their experiences and draw valuable lessons from them.
3. **Apply results in a consequential task.** After the architects have defined viewpoints, the corresponding views are developed in a next task. Likewise, after the assessment procedure is decided upon by the stakeholders, an assessment following that procedure is executed.

Inquiry learning is about *learning science*, in our case software architecture. By being actively involved in the generation and testing of hypotheses (for example by stating that a particular view captures certain major design decisions, and next discussing the view with stakeholders) a more thorough insight is provided for than a mere lecture about characteristics of possible architectural views. Inquiry learning is also about *learning to do science*. It is about methods to do research: how to effectively cooperate with others, how to communicate findings, and the importance of providing rationale. Finally, inquiry learning is about *learning about science*. It gives students a critical attitude, and shows them that providing arguments is important. Architect groups have to make their design decisions explicit and convince their stakeholders of the trade-offs being made. It was interesting (both to us and the students) to note that architect groups who interacted with different stakeholder groups experienced different constraints and hence came up with different solutions. For example, the architect groups who interacted with *Huns* consistently chose to strictly separate personal and travel data, while the architect groups that interacted with *The Patient* could get away with simpler solutions, such as the option to have anonymous cards. Apparently, *Huns* enforced heavier constraints on privacy.

We follow the *guided discovery* approach to inquiry learning, a middle ground between didactic teaching and unbounded discovery learning [4]. In guided discovery, the teacher is a facilitator, guiding the discovery by students. We do not prescribe which viewpoints to use, but give feedback on viewpoints chosen. Often, snippets of theory are dealt with based on what happens in the classroom. For instance, in one meeting different architect groups discussed their viewpoints. Some groups had 3 viewpoints, while others felt that 4 were needed. When asked, it turned out that, in almost all cases, the number of views equaled the number of group members. This gave us the opportunity to discuss Conway's law [8].

In our weekly lectures, students take turn in leading the discussion. An architect group might discuss the trade-offs they have made: stakeholder concerns that are being dealt with and those that are not, for which reasons, and how this is manifested

in their architecture. This presentation then gives rise to a discussion with other architect groups that may have chosen for a different balance. This results in a talk across groups, where groups learn from other groups.

In a Community of Learners, students and teachers each possess certain expertise. Fairly soon, student groups knew (much) more about the electronic billing case than the teachers did. They found that similar systems had been developed in other countries (e.g., Hong Kong), and knew of the specific challenges of the Dutch situation. Likewise, the stakeholder groups had carefully studied their background and concerns. They knew about applicable privacy regulations, ongoing political discussions, and the position of consumer organizations. Near the end of the course, architects from the company that was responsible for the development of the actual Dutch electronic billing system gave a guest lecture about their solution. Initially, they approached the lecture from a "spectator theory of knowledge" point of view and started by presenting their solution. They soon discovered, though, that the students formed a worthy discussion partner, and the mood of the lecture shifted from a mere presentation of highlights to a discussion of the architectural challenges involved.

6 Discussion

The Community of Learners approach provides for an ideal nursery for knowledge workers, such as software architects and, more generally, university graduates. Some of the strengths attributed to the guided discovery approach in a Community of Learners [4] clearly manifested themselves in our course:

- **Distributed expertise:** students and teachers share their knowledge with each other. Variability arises opportunistically. If one architect team feels the need to pay careful attention to privacy issues, a viewpoint addressing privacy will emerge.
- **Adult teachers are not the only sources of knowledge:** peer groups soon know more about the case than the teachers do.
- **Everyone is both teacher and learner:** students teach each other, *and* the teachers, what they know about the electronic billing case and its major architectural issues.

Curiosity is an important motivating factor in a Community of Learners context. In a previous year, we asked the students to develop an architecture for an archiving system for a government agency. Although the case was derived from an actual system, it did not really ring a bell with the students. The electronic billing case was much more interesting to our student population. It was in the news every day. As students, they had a personal stake as well (privacy issues, the cost to students of the new billing system). This further strengthened their eagerness to acquire knowledge and learn.

One weakness of the guided discovery approach that we encountered in another year is the limited knowledge capital. In particular, a weak stakeholder group affects the work of all architect groups it has to deal with. A stakeholder group that does not develop strong concerns about the system to be developed allows architect groups to come up with an easy solution and a lower-quality architecture description, and makes them get away with handwaving arguments about their solution and its rationale.

We fully achieved the goals set for our course. After the course, the students knew (1) how to develop different architectural views, (2) the wicked nature of software architecture, (3) how to document design decisions, and (4) how to do an assessment. Group reflections at the end of the course also point out that the Community of Learners approach does pay off.

The students generally consider the workload high, but they also report a very large learning effect. One easy measure of student's appreciation of a course is class attendance. Often, attention is large in the first few lectures, and then gradually drops, sometimes even to less than 50%. We experienced an overall attendance of more than 90%.

The VU University Amsterdam wants to promote a broad application of the Community of Learners approach in various courses. Our course was selected by the university as an example to others. Part of the lectures was videotaped and turned into a short movie. Interviews with students and teachers featured in various education-related publications of the university.

7 Conclusions

In our course, students learn about the wicked nature of software architecture. They learn that there are different architectural strategies and that there is no single best solution. They learn how to document a software architecture for different audiences, and how to reason about their design decisions.

Application of the Community of Learners concepts considerably helped us to achieve our goals. Wicked problems cannot be solved by following a strictly sequential and fixed number of steps. The inquiry learning cycle of the Community

of Learners approach instead provides for a good fit with the wicked nature of software architecture design. In a Community of Learners approach, both students and teachers have certain knowledge. In particular, the students are partners in the knowledge development process. This considerably increases their commitment to the course, and provides for an ideal nursery for knowledge workers such as software architects. By applying the Community of Learners concepts, the students not only learned science (i.e. software architecture), they also learned to do science, and about science.

References

[1] R. Bareiss and M. Griss. A Story-Centered, Learn-by-Doing Approach to Software Engineering Education. In *Technical Symposium on Computer Science Education (SIGCSE'08)*, pages 221–225. ACM, 2008.

[2] H. Barrows and R. Tamblyn. *Problem-based learning: An approach to medical education*. Springer, 1980.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, second edition, 2003.

[4] A. Brown and J. Campione. Guided Discovery in a Community of Learners. In K. McGilly, editor, *Classroom lessons: Integrating cognitive theory and classroom practice*, pages 230–270. MIT Press, 1994.

[5] A. Brown and J. Campione. Psychological theory and the design of innovative learning environments: On procedures, principles, and systems. In L. Schauble and R. Glaser, editors, *Innovation in learning: New environments for education*, pages 289–325. Lawrence Erlbaum Associates, Inc., 1996.

[6] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2003.

[7] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

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

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

[10] C. Hofmeister, P. Kruchten, R. Nord, H. Obbink, A. Ran, and P. America. A General Model of Software Architecture Design Derived from Five Industrial Approaches. *Journal of Systems and Software*, 80(1):106–126, 2007.

[11] IEEE. IEEE Recommended Practice for Architecture Description. Technical report, IEEE Standard 1471, IEEE, 2000.

[12] M. Jaccheri. Tales from a Software Architecture Course Project. On-line at http://www.idi.ntnu.no/~letizia/swarchi/eCourse.html, 2002.

[13] P. Lago and H. van Vliet. Teaching a Course on Software Architecture. In T. Lethbridge and D. Port, editors, *Proceedings 18th Conference on Software Engineering Education & Training*, pages 35–42. IEEE Computer Society, 2005.

[14] T. Männistö, J. Savolainen, and V. Myllärniemi. Teaching Software Architecture Design. In *Proceedings Seventh Working IEEE/IFIP Conference on Software Architecture*, pages 17–124. IEEE Computer Society, 2008.

[15] G. Muller. Experiences of Teaching Systems Architecting. In *INCOSE 2004*, 2004.

[16] H. Rittel and M. Webber. Dilemmas in a General Theory of Planning. In *Policy Sciences, Vol. 4*, pages 155–169. Elsevier, 1973.

[17] M. Shaw. Toward Higher Level Abstractions for Software Systems. In *Proceedings Tercer Simposio Internacional del Conocimiento y su Ingerieria*, 1988.

[18] M. Shaw, J. Herbsleb, I. Ozkaya, and D. Root. Deciding What to Design: Closing a Gap in Software Engineering Education. In P. Inverardi and M. Jazayeri, editors, *Software Engineering Education in the Modern Age*, pages 28–58. Springer Verlag, 2006.

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