VU Research Portal

Teaching a course in Software Architecture

Lago, P.; van Vliet, H.

published in
Proceedings Eighteenth Conference on Software Engineering Education & Training
2005

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Lago, P., & van Vliet, H. (2005). Teaching a course in Software Architecture. In T. C. Lethbridge, & D. Port (Eds.), *Proceedings Eighteenth Conference on Software Engineering Education & Training* (pp. 35-42). IEEE Computer Society.

Download date: 27. Nov. 2021

Teaching a Course on Software Architecture

Patricia Lago and Hans van Vliet
Vrije Universiteit, Amsterdam, The Netherlands
{patricia | hans}@cs.vu.nl

Abstract

Software architecture is a relatively new topic in software engineering. It is quickly becoming a central issue, and leading-edge organizations spend a considerable fraction of their development effort on software architecture. Consequently, software architecture is increasingly often the topic of a dedicated course in software engineering curricula. There are two general flavors as for the contents of such a course. One flavor emphasizes the programming-in-the-large aspects of software architecture and concentrates on design and architectural patterns, architecture description languages and the like. The other emphasizes the communication aspects of software architecture to a variety of stakeholders, thereby acknowledging a broader view of software architecture. In this paper we report our experiences with two master-level courses in software architecture that focus on these communication aspects. We show that, by appropriately focusing the contents of such a course, key aspects of this industrially very relevant field within software engineering can be taught successfully in a university setting.

1 Introduction

Software architecture is becoming one of the central topics in software engineering. In early publications, such as [16], software architecture was by and large synonymous with global design. This view emphasizes design patterns and architectural patterns [3] and the description of the resulting architecture in some Architectural Description Language (ADL) [14]. In a broader view, software architecture involves making tradeoffs 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 a (global) design that meets those requirements. This broader view is quickly becoming the received view [1].

This broader view of what software architecture entails is also reflected in the characteristics of the architecture-centric software development life cycle. We may by and large characterize the pre-architecture life cycle as follows (see also figure 1.(a) and any standard text on software engineering such as [17]):

- Discussions about the system involve a few stakeholders only. Often, it is only the client. Possibly, one or a few user representatives are involved.
- Iteration involves functional requirements only. Once the functional requirements are agreed upon, these are supplemented with non-functional requirements. Together, these constitute the agreed-upon requirements specification for the system to be built.
- In particular, there is no balancing between functional and non-functional requirements. For example, there usually is no discussion to trade off functionality and speed.

In these development approaches, requirements engineering very much is an activity that focuses on the problem space, while the subsequent design phase focuses on the solution space. Conversely, the characteristics of an architecture-centric life cycle are as follows (see also figure 1.(b)):

- The discussions involve many stakeholders: the client, different classes of users, future maintainers of the system, owners of other inter-operating systems.
- Iteration concerns both functional and non-functional requirements.
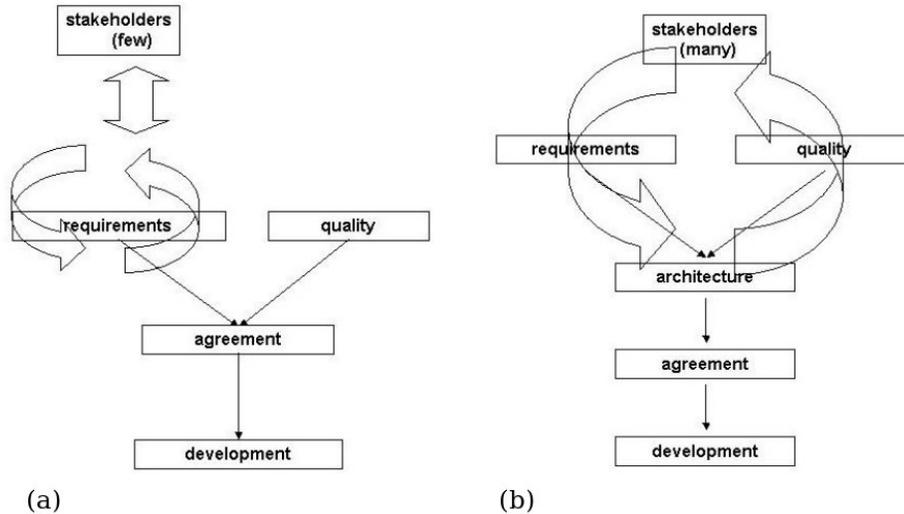
Figure 1. Life cycles: (a) Pre-architecture and (b) Architecture-centric

- In particular, architecting involves finding a balance between these types of requirements. Only when this balance is reached, next steps can be taken.

In the latter 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.

Software developers focus on the transition of the architecture into code. They view an architecture as consisting of components and connectors. The other stakeholders may have a variety of other concerns, and are best served by some type of architecture description that highlights how these concerns are addressed in the architecture. They are typically not served best by a description that looks like a high-level programming language such as typically offered by ADL's, or a formal diagram as offered by UML.

Following this line of thought, the documentation of an architecture is typically split into a small number of views, each of which highlights the concerns of a specific set of stakeholders. This same approach is used in other architecture fields. In house construction, e.g., we use different drawings: one for the electrical wiring, one for the water supply, etc. These drawings reflect different views on the same overall architecture. The same applies to software architecture.

The development and use of different architectural views in a context where the software architect communicates with a variety of both technical and non-technical stakeholders, is **the** central issue in our software architecture course. This is further elaborated in section 2, where we discuss the goals we had for our software architecture course, and how we designed the course to meet these goals. Section 3 next describes two software architecture courses that we gave between September 2003 and February 2004, including some examples from both courses. Section 4 discusses the lessons we have learned and section 5 the related work. Section 6 states our conclusions.

2 Global set-up of the Software Architecture Course

2.1 What's Important in Software Architecture

A software architecture, or rather its description, reflects the major design decisions made. These decisions are made by the architect, taking into account the concerns of the different stakeholders involved. The architect elicits the requirements, both functional and non-functional, from the stakeholders, and devises a solution that accommodates these requirements in a balanced way. Usually, not all requirements of all stakeholders can be met. Architecting then involves negotiations with stakeholders to get a compromise.

In these discussions with stakeholders, the architect uses a description of the architecture which reflects the current set of decisions made, and how these address the concerns of the stakeholders. One possibility is to devise a single description

of the system which addresses all concerns of all stakeholders. This however is likely to result in a very complex document that no one understands. Like with building plans, it is better 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 [9].

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'. Viewpoints can be reused in other projects; these reusable viewpoints are termed 'library viewpoints'. A stakeholder can have one or more concerns, and concerns can be relevant to more than one stakeholder. Clements [4] gives many useful advices as to which views might be appropriate in certain circumstances. An early example of the idea to have multiple views in architecture descriptions is given in [12].

The architect tries to balance the requirements of the various stakeholders involved. In the end, though, 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 [5]. 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 architect presents the architecture and its rationale to the stakeholders. He highlights the major design decisions that led to the architecture. He may use different views of the architecture to illustrate his 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 architect explains 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 the findings of the assessment.

2.2 Goals of the Software Architecture Course

With the above in mind, we decided on the following goals for our course:

- The students should know how to develop different architectural views of an architecture, addressing specific concerns of stakeholders. We used [9] as the model for doing so.
- The students should know of the *wicked* nature of software architecture [2]. 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.
- 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 the architecture and the decisions that led to the architecture to its stakeholders, this once again stresses the communication aspect of software architecture.

The rethinking and examination of one's own professional creations improves one's performance in that profession [7]. Through the studio-like set-up of our course, with a weekly feedback on deliverables (architectural views, lists of scenarios, etc), essential aspects of this reflective practitioner approach are applied.

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.

3 The Software Architecture Courses

We gave the software architecture course twice in two quite different curricula. The first course (discussed in section 3.1) was part of a one-year master program in professional software engineering. It was a very intensive course. It lasted eight weeks, and the students had to spend 20 hours/week on the course (so they took only two courses in parallel). Most of the work was done in the first six weeks. We had guest speakers in week 7, and exam preparation and exam in week 8. A total of 19 students enrolled in the course. They worked in teams of three (and in one case four) people. They had all done a bachelors program at a polytechnic institute before enrolling in the course. We used [1] as text book.

The second course (elaborated in section 3.2) was part of a regular masters program in both computer science and business informatics. It had a duration of 12 weeks, with a Christmas break after week eight. The students had to spend 12 hours/week on this course. A total of 50 students enrolled, approximately evenly divided between the (two-year) master program in computer science and the (one-year) master program in business informatics. They worked in teams of four or five people. Their background was quite varied. A large proportion had done a bachelors at our university. Quite a number of students had done a bachelors at a polytechnic institute, while some students enrolled in the masters program after having done a bachelors in another country. No text book was prescribed, though many students used [1].

None of the students had extensive previous experience with software architecture. For most, this was their first exposure to the topic. Most students had previously followed a software engineering course of some sort.

3.1 The intensive architecture course

Since the work for the intensive course effectively had to be finished within six weeks, we decided not to have the students develop an architecture from scratch. So we started with an existing pile of Java code (approx 75 KLOC). This existing system implemented a car rental system. It used a typical 3-tier architecture that separated the user interface from the business logic and the data layer. We gave the following tasks:

- Reverse engineer the architecture from the (undocumented) source code. We gave no guidelines as to how to do this, nor guidelines as to what the resulting description should look like. Most groups found and used JBuilder in combination with some existing reverse engineering method, such as Dali [1, chapter 10]. In all cases, the architecture was described in a view depicting the major functional elements; see figure 2 for an example. Quite a few of the box-and-line diagrams delivered had unclear semantics. Boxes could denote a (Java) class, logical subsystem, or some other static entity. Lines could denote a calling relationship, an is-contained-in relationship, an is-subordinate-to relationship, etc.

- Develop some (at least two) architectural views and the corresponding viewpoints. All groups developed an improved version of the functional view developed in the previous step. This improved version usually made a more consistent use of various types of boxes and lines. Almost all groups had difficulty in devising a second view. Some groups came up with a rather shallow end-user view with a few icons depicting the user, the computer, and a LAN or WAN connection. Some groups devised a process view [12] showing the dynamic structure of the system in terms of tasks, processes, communications, and the allocation of functionality to run-time elements. The most interesting view we encountered is (partly) depicted in figure 3. This view shows the relationship between business requirements, architectural decisions, and quality aspects. It shows trade offs and supports "what if" scenarios. In this example, a high level of data integrity is chosen, and the impact on other qualities, the proposed architecture, and business requirements is reflected in the coloring scheme.

- Identify the styles and patterns used in the architecture, and discuss their benefits. All groups defined new viewpoints showing how the patterns were used in the architecture: most viewpoints acted as catalogues, pointing out which patterns were used in which subsystems; in these cases the pattern benefits could be discussed in general terms only. Only one group defined viewpoints showing how elements inside subsystems were specializations of elements within a certain pattern; in doing that they could discuss qualities more thoroughly, too.

- Do an architecture assessment. We let half the groups act as architects, and the other half as stakeholders. We did not assign specific assessor roles. We left it to the students to choose or devise a specific assessment method. All groups chose a trimmed-down version of the Architecture Trade-Off Analysis Method (ATAM) [5], whose structure resembled that sketched in section 2. All groups were enthusiastic about the insights they gained in the quality of their architectural description. They also acknowledged now having a much deeper knowledge of the impact their particular set of design decisions had on the architectural solution chosen. One group interestingly noticed the potentially manipulative character of such an assessment. A very assertive architect may overwhelm stakeholders with an overload of confident
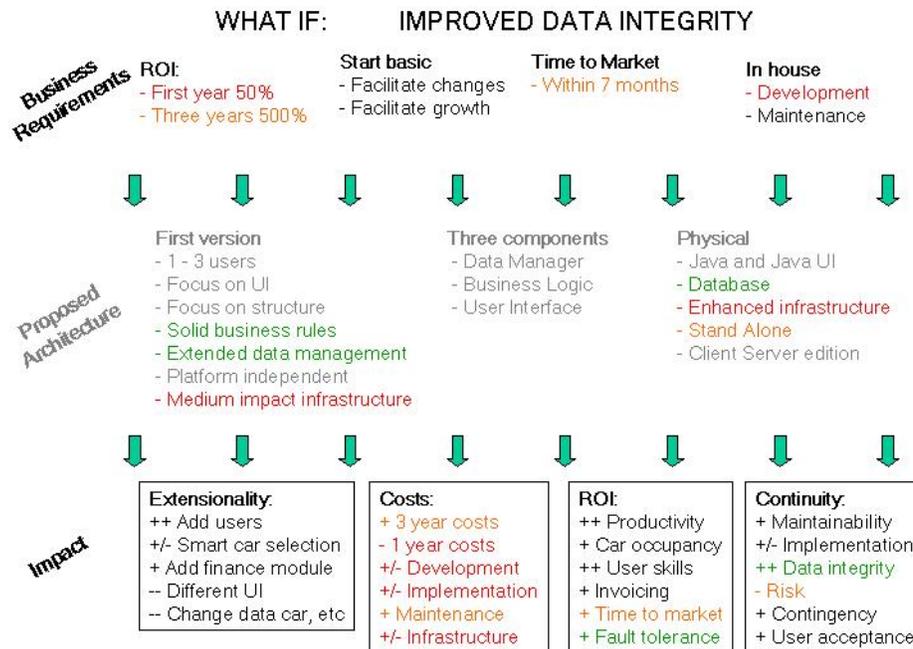
4
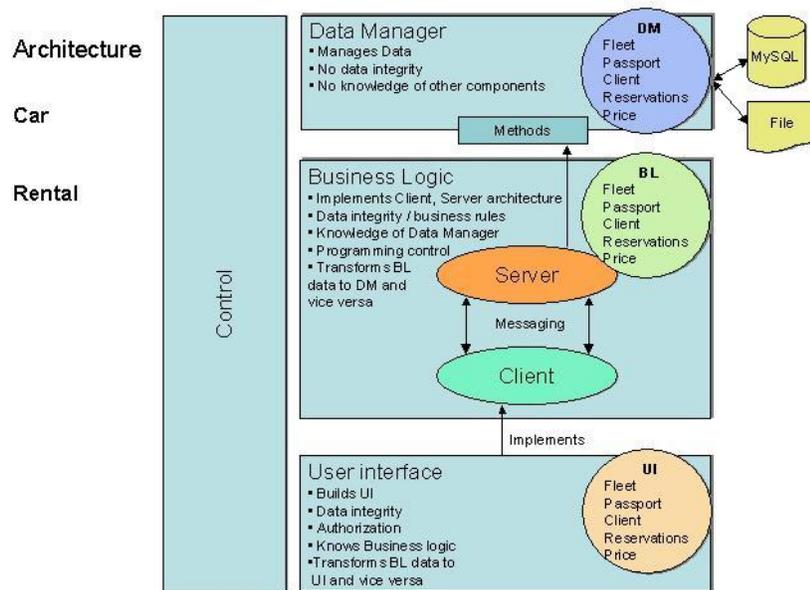
Figure 2. A 3-tier solution

statements, and effectively preclude a productive discussion. On the other hand, having a vision and being decisive are required traits of a software architect [13], [6].

3.2 The regular architecture course

In the regular course, we asked the students to develop a software architecture from scratch. The students were asked to develop an architecture for handling the paperwork in a courthouse; see figure 4 for this assignment. Two groups acted as stakeholders, nine groups acted as architects. One stakeholder group interacted with four architect groups, while the second stakeholder group interacted with five architect groups. The stakeholder groups could devise their own roles. Both these groups decided on roles like IT manager, judge, lawyer, police. One group decided to have the press as one of the stakeholders. Since this resulted in a lot of security problems in the architectures that had to comply with this stakeholder, this role presented quite some problems to the architects that had to deal with it; more on this later on.

For this course, we chose the following tasks:

- Develop an initial architecture. Again, we gave no guidelines as to how to do this, nor guidelines as to what the resulting description should look like. Since most students had previously followed the software engineering course at our department, they were familiar with the notion of MOSCOW: the separation of requirements into Must haves, Should haves, Could haves, and Won't haves. They applied these notions in the requirements elicitation discussions with the stakeholder groups to prioritize requirements. The architect groups that had to deal with the press stakeholder, tended to rate his requirements as low, probably because they had difficulty deciding how to handle them. This resulted in a lot of heated discussions in some of those groups. Similar to the intensive course described earlier, the resulting architecture was described in a functional view resembling the one in figure 2. And again, the semantics of the box-and-lines diagrams was usually unclear.

- Develop at least two architectural views and the corresponding viewpoints. To help the students do this, we presented them with a method for defining IEEE Std 1471 viewpoints [11]. This method has four steps: (1) compile stakeholder profiles, (2) summarize available design documentation, (3) relate this summary to the stakeholder concerns, and (4) define viewpoints. This method forced them to consciously think of stakeholder concerns and how to relate them

5

WHAT IF: IMPROVED DATA INTEGRITY

Business Requirements

ROI:
- First year 50%
- Three years 500%

Start basic
- Facilitate changes
- Facilitate growth

Time to Market
- Within 7 months

In house
- Development
- Maintenance

Proposed Architecture

First version
- 1 - 3 users
- Focus on UI
- Focus on structure
- Solid business rules
- Extended data management
- Platform independent
- Medium impact infrastructure

Three components
- Data Manager
- Business Logic
- User Interface

Physical
- Java and Java UI
- Database
- Enhanced infrastructure
- Stand Alone
- Client Server edition

Impact

Extensionality:
++ Add users
+/- Smart car selection
+ Add finance module
-- Different UI
-- Change data car, etc

Costs:
+ 3 year costs
- 1 year costs
+/- Development
+/- Implementation
+ Maintenance
+/- Infrastructure

ROI:
++ Productivity
+ Car occupancy
++ User skills
+ Invoicing
+ Time to market
+ Fault tolerance

Continuity:
+ Maintainability
+/- Implementation
++ Data integrity
- Risk
+ Contingency
+ User acceptance

Figure 3. A business view

to architectural decisions, something they were not accustomed to, and found difficult. Especially step 3 forced the students to present their results in a concise way, a very necessary skill for a successful architect. We had to guide them through this process, and give examples.

- Do an architecture assessment. We let half the architect groups act as architects, and the other half as stakeholders. In a second assessment round, we reversed these roles. This way, all architect groups played both roles. We asked the two stakeholder groups to define a trimmed-down version of ATAM [5] to be used, and next act play the assessor role during the assessment. In this case, many groups perceived the stakeholders as attacking their solution. As a result, they vigorously defended their design decisions. This considerably improved in the second assessment round, though the learning effect of this second assessment was less than hoped for. At the end though, the students were again very positive about the assessment exercise, for the same reasons given by the students of the other course. In this course, we observed a strong correlation between the quality of the assessment and the specificity of its inputs, viz. the architecture description and the set of scenarios. More specific inputs resulted in a much better assessment.

4 Lessons learned

In discussions on software engineering courses, a recurring theme is whether or not these topics can be taught at all without a "real" case in the accompanying lab assignments. The same issues of course arises when discussing courses on software architecture. In our course that used the Car Rental System, we had real code, but no stakeholders, and no concerns. The architects had to follow lines of thought like "if I were the owner of this car rental company, I might have a concern like . . . ". In one respect, we fared slightly better in the course that featured the courthouse system, in that we assigned certain students the role of stakeholder. But none of them was a real lawyer or judge. Interestingly, one of them had a brother who worked for the police force, and had actual experience with requirements similar to ours. So he interviewed his brother, and came back with a remarkably realistic set of concerns. However, we found that it is not so much the realism of the concerns that matters, but the fact that they vary, and conflict, and need compromises. Especially in the course that featured the courthouse example, this worked quite well. To further improve this, we intend to let a lab assistant participate as stakeholder in the next releases of our course. Also, it is difficult to make the workload of architects and stakeholders equally high. To balance the workload, we assigned the stakeholder groups some extra tasks, such as the preparation of the assessment procedure.

> **Court Online**
>
> The court in Blokker wants to get rid of the large amount of paper that gets produced in their cases. They want a situation in which the record of a case, the hearings of witnesses, court reports, etc., are all stored electronically, and are also available and used in electronic form in the courtroom. Judges and lawyers do not rummage through a large pile of paper, but zap through an electronic file.
>
> Software house VU-Arch is asked to develop an architecture for this system. Important functional requirements for the system are:
>
> - Storage of all parts of a dossier
> - The possibility to add parts to a dossier, change parts by a more recent version, or annotate parts of a dossier, if authorized to do so.
> - Protection against unauthorized access to (parts) of a dossier
> - Parts of a dossier can be in different formats: plain text, pictures, scanned handwritten documents, etc.
> - A function to search a dossier
>
> Next to these functional requirements there are a number of additional things to be decided upon. One may think of whether or not Open Source is required, and whether or not the court should continue its business with the two-person company Salto which earlier provided the court in Blokker with a cheap scanner and accompanying software.

Figure 4. Initial description of the requirements for Court Online

Students find it difficult to develop architectural views. In one respect, this need not come as a surprise, since it is a quite common phenomenon in courses that have a design component [8]. Students often have difficulties in problem solving, specifically in judging how "close" to a final solution their design is. This is even more true if there is no single best solution, as in software design, and especially software architecture. These are 'wicked' problems [2].

Our students mostly had a background in computer science/software development. So it should not come as a surprise that they all developed a functional view first. This comes closest to the traditional global design representation they are familiar with. After some tutoring, they could develop other technical views, along the lines of [12] or [1]. The development of a more business-oriented view remains a challenge. In future courses, we will pay specific attention to this issue.

At first, many students saw the architecture assessment as threatening. This was especially true for the regular architecture course, in which the students designed the architecture from scratch, and thus perceived a strong sense of ownership of the results. This feeling persisted, even though we emphasized from the outset that their grades would not depend on the number of scenarios their architecture could cope with, but only on the degree to which they could actually answer such questions. Students apparently have to learn that not being able to cope with certain situations is the result of (hopefully) explicit decisions and tradeoffs made, and not necessarily a negative statement.

One thing we had not explicitly considered at the beginning is that the quality of the result of an architecture assessment very much depends on the quality of the architectural description. We knew the quality of the scenarios is important, and stressed this point during the course. Scenarios have to be as explicit as possible. A scenario of the form "What if we replace the database system" is not good, while "What if we replace Oracle by DB2" is. The latter allows for more concrete and in depth investigations, and more specific answers. The importance of the quality of the architectural description did not really surface during the course with the car rental system. The students always had the code available there, so if the architectural description did not provide the answer, they could consult the code. This was not true in the course that used the courthouse case. There, the architectural description was all they had. If this description was too global or vague, questions about the impact of scenarios remain vague as well, resulting in quite a bit of handwaving in the argumentation.

5 Related work

There are very few papers that describe experiences with teaching software architecture courses.

Jaccheri [10] 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.

6 Concluding remarks

We do not cover all aspects of software architecture in our course. Based on a careful analysis of the prevalent views of essential aspects of software architecture, we selected topics to deal with these. We devised a set-up which allows us to teach these topics in a university setting.

We achieved the goals set for the course. Though the students generally considered the workload quite high, they also report a very large learning effect for this course. They gain confidence about how to document software architecture for specific purposes and stakeholders, and are able to reason about architectural decisions. Also, they can cope with the fact that alternative architectural strategies exist and that there is no single best solution. Our main challenge for the next iteration of this course is to give the students more guidance in their design-type activities, at the same time retaining a sufficiently broad spectrum of proposed solutions.

We consider the setup of the regular course more successful than that of the intensive course. The main reason is that students there cannot backslide to the code, when the documented views do not suffice. They are forced to think more carefully about the architecture documentation.

Acknowledgements

References

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

[2] D. Budgen. *Software Design*. Addison Wesley, second edition, 2003.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. John Wiley & Sons, 1996.

[4] 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.

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

[6] D. Dikel, D. Kane, and J. Wilson. *Software Architecture: Organizational Principles and Patterns*. Prentice Hall, 2001.

[7] O. Hazzan. The reflective practitioner perspective in software engineering education. *Journal of Systems and Software*, 63(3):161–171, 2002.

[8] J. Hughes and S. Parkes. Impact of Verbalisation upon Students' Software Design and Evaluation. In *Proceedings 8th International Conference on Empirical Assessment in Software Engineering (EASE 2004*, pages 121–134. IEEE, 2004.

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

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

[11] H. Koning and H. van Vliet. A Method for Defining IEEE Std 1471 Viewpoints, 2004. Submitted for publication.

[12] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[13] R. Malveau and T. Mowbray. *Software Architect BOOTCAMP*. Prentice Hall, second edition, 2004.

[14] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions in Software Engineering*, 26(1):70–93, 2000.

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

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

[17] H. van Vliet. *Software Engineering: Principles and Practice*. Wiley, second edition, 2000.