A Pattern Language for Documenting Software Architectures

Paris Avgeriou[1], Nicolas Guelfi[2], Reza Razavi[2]

[1]Fraunhofer IPSI
CONCERT division,
Dolivostrasse 15 D-64293
Darmstadt, Germany
paris.avgeriou@ipsi.fraunhofer.de

[2]Software Engineering Competence Center
(SE2C), University of Luxembourg,
6, rue Richard Coudenhove-Kalergi L-1359
Luxembourg-Kirchberg, Luxembourg
{ nicolas.guelfi, reza.razavi}@uni.lu

Abstract

The process of creating the architecture of a software system results in a documentation, which is recognized as a key artifact for stakeholder communication, early analysis of the system, support for quality attributes and trouble-free maintenance. The problem of software architecture documentation remains to a large extent unsolved; however the past few years, significant advances have been made in the field from research academic and industrial centers. This paper introduces an approach for recording the results that have been achieved hitherto in the field of documenting software architectures, by formatting them in the shape of patterns. We aim at assembling knowledge and experience in the field from industry and academia, with respect to the few issues that the community has reached consensus. Furthermore, by codifying this knowledge and experience in the form of patterns, we hope for a wider dissemination of architectural documentation concepts and practices to the community and thus a further advance of the field.

1 Introduction

People 's ideas on the field of software architecture range from those who consider it as simple 'box and arrow diagrams' to those who claim that software architecture is a panacea that will revolutionize software development. Nevertheless industry and academia have reached consensus that investing on architecture is of paramount importance to the project success [3, 5, 7, 12, 13, 14, 18, 28]. In most iterative and incremental approaches, from heavyweight processes, e.g. the Rational Unified Process, to agile processes, e.g. the Extreme Programming paradigm, architecture plays an undoubted, pivotal role [18, 22]. Moreover there is an undoubted tendency to create an engineering discipline on the field of software architecture if we consider the published textbooks, the international conferences devoted to it, and recognition of architecting software systems as a professional practice [9].

Documenting an architecture means to "write it down" [9]: to design, describe or specify all the architectural elements, diagrams, models, decisions, rationale or anything else that might concern the architecture. Therefore, we do not distinguish between 'designing' and 'documenting' an architecture. Despite the attention drawn to this emerging discipline, there has been little guidance, regarding how to document a software architecture. Evidently there have been advances in the field, especially concerning Architecture Description Languages, design and evaluation methods, as well as reusable architectural artifacts such as architectural patterns and frameworks. But a software architecture needs to be rigorously documented if we expect to profit from its advantages such as communication of stakeholders, early analysis of the system, support of quality and trouble-free maintenance. Unfortunately the problem of documenting software architectures has not been solved [9]; on the contrary we are still at early stages of addressing it [21].

On the bright side, there is growing consensus nowadays about certain aspects of the task of software architecture documentation, things that experience has proven right over the years [14]. There has been extensive publication of these concepts and practices in numerous textbooks and research papers and we believe it is worth assembling and documenting them in the style of patterns. We thus hope to provide experience and knowledge on this field, in digestible and inter-related chunks and therefore help software architects, especially inexperienced ones. On the side, we hope for a broader dissemination of software architecture concepts to the software engineering community.

The structure of the rest of this paper is as follows: the second section attempts a short literature review on the subject of software architecture documentation. The third section briefly explains a special category of systems, Learning Management Systems, whose architectural documentation will be used as a 'running example' inside the patterns. The fourth section contains part of the pattern language for this field. Finally the fifth section wraps up with conclusions derived from this work.

2 The state of the art

There have been numerous approaches from academia, industry and international bodies, on what the documentation of a software architecture entails and what process should be followed to perform the actual documentation. These approaches that are briefly portrayed in this section, are the sources that have been used to find common ground and subsequently mine the patterns presented in the following section.

IEEE has developed a **Recommended Practice for the architectural description of software-intensive systems** [14]. This standard mainly contains a framework of concepts in order to facilitate the adoption of architectural principles and practices in the industrial and research community. It remains at a general level of prescription but does provide a common denominator for tackling the task of architectural documentation. For the specific category of Open Distributed Processing Systems, an ISO/IEC committee has developed the **Reference Model for Open Distributed Processing (RM-ODP)** [15, 24]. This standard guides development teams into designing architectures that support distribution, interoperability and portability. The Open Group has also developed **The Open Group Architectural Framework (TOGAF),** which supports some of the IEEE 1471 standard concepts and practices such as identification of stakeholders and their concerns, views as instances of viewpoints etc.

In academic research centers, even though work on software architecture has been carried out for almost a decade, few results have been derived for documenting architectures. Bass et al. [3], Shaw and Garlan [28] and Bosch [5] have early identified the problem of documenting a software architecture. Despite the fact that these first approaches did not attempt to tackle this problem, they did identify basic principles such as the explicit support of qualities by the architecture and the indispensable use of reusable architectural assets such as patterns (see also [6]), reference models and reference architectures. Clements et al. in [9] have gone a step further and specify how the selection of views should be performed and how they should be documented. They also indicate the application of architectural patterns according to the selected views, as well as the incorporation of horizontal issues that apply to all views.

Industrial research centers have also worked on the issue of architectural documentation, codifying experience from industrial case studies. One of the earliest works which deals with architectures of information systems in enterprises is the Zachman framework [29], a two-dimensional matrix that associates views with stakeholders. Hofmeister et al. [13] propose a set of 4 views for documenting the architecture assisted by the use of the Unified Modeling Language (UML). Furthermore, the Rational Unified Process (RUP) [18, 25], mandates that an architecture documentation should contain the architecturally significant elements from all the models, organized into a number of predefined views.

Finally the documentation of software architectures has always been concerned with the definition of the appropriate notations or languages for modeling the various architectural artifacts. As a consequence, a different genre of languages has emerged over the past ten years: **Architecture Description Languages (ADLs)**, which aim at formally representing software architectures [3, 8, 19]. Unfortunately these languages have never been broadly used in industry and most of them lack support by appropriate tools. However the recent trend is the use of the widely accepted UML as an ADL, either by extending it per se, or by mapping existing ADLs onto it [20, 26].

3 A 'running example'

Throughout the description of the patterns, we demonstrate small parts of a case study that concerns the architectural documentation of a Learning Management System (LMS). In particular, the 'example' clause of each pattern description gives characteristic details about this case study, by focusing on fragments of the architectural documentation, that are related to the specific pattern. We decided not to show the entire case study [2], as it is out of the scope of this paper, but only to show concrete examples of the patterns proposed. In this section we briefly explain the nature of these systems in order to make the examples in the patterns more comprehensible.

A vast number of Learning Management Systems (e.g. WebCT, Blackboard, LearningSpace, VirtualU, ATutor) exist nowadays. LMS are used to support on-line courses in higher education institutes, but also in K-12 schools and vocational training organizations. They support a number of **features**, that can be classified into the following groups:

- **Course Management,** which contains features for the creation, customisation, administration and monitoring of courses.
- **Class Management,** which contains features for user management, team building, projects assignments etc.
- **Communication Tools,** which contains features for synchronous and asynchronous communication such as e-mail, chat, discussion fora, audio/video-conferencing, announcements and synchronous collaborative facilities (desktop, file and application sharing, whiteboard).
- **Student Tools,** which provide features to support students into managing and studying the learning resources, such as private & public annotations, highlights, bookmarks, off-line studying, log of personal history, search engines through metadata etc.

- **Content Management,** which provide features for content storing, authoring and delivery, file management, import and export of content chunks etc.
- **Assessment Tools,** which provides features for managing on-line quizzes and tests, project deliverables, self-assessment exercises, status of student participation in active learning and so on.
- **School-Management**, which provide features for managing records, absences, grades, student registrations, personal data of students, financial administration etc.

The users of LMS can be classified into three categories:

- The *learners* that use the system in order to participate through distance (in place and/or time) to the educational process.
- The *instructors*, being the teachers and their assistants that use the system in order to coach, supervise, assist and evaluate the learners
- The *administrators* of the system, who undertake the support of all the other users of the system and safeguard its proper operational status.

4 A pattern language for documenting software architectures

Figure 1 depicts a map of the pattern language as well as the most important relationships between the proposed patterns. This pattern language attempts to tackle the complex problem of documenting software architectures and the *intended audience* for the language is software architects. This pattern language does not imply a waterfall-like up-front architecture design approach, but rather supports iterative and incremental approaches, ranging from agile to heavyweight processes. The patterns in grey background will be elaborated in the remainder of this section while the patterns in white background constitute future work. The thumbnails of all the patterns are the following:

- The BIG PICTURE, is a model of the environment that the software system inhabits in, that shows the interaction between the software system and its environment and should be consistent with the software system's architecture. It is the starting point for documenting a software architecture.
- The STAKEHOLDERS AND THEIR CONCERNS, is a list of all the categories of people that are involved in the system and their corresponding interests, that should constitute a checkpoint for the effectiveness of the architecture document. The starting point for deriving STAKEHOLDERS AND THEIR CONCERNS is the BIG PICTURE.
- STAKEHOLDER PRIORITIZATION, indicates that stakeholders who are more important than others should have greater priority in getting their concerns addressed in the views.
- The architect must select the VIEWS ACCORDING TO THE CONCERNS of the stakeholders, so that each concern is addressed by at least one view.
- The SPECIFICATION OF VIEWS, is an unambiguous definition of the semantics of the different views that architects must conform to when producing the CONTENTS OF VIEWS.

- The process of creating the CONTENTS OF VIEWS, follows carefully the SPECIFICATION OF VIEWS, focusing upon what exactly should be included in a view and what not.
- The ROADMAP OF THE DOCUMENTATION, is a detailed plan of navigating through the documentation, with hints on how it should be read by the stakeholders.
- The architect should specify WHO READS WHAT, by indicating which stakeholders are meant to read each part of the documentation. Thus the stakeholders are guided through the documentation and are able to understand the specific parts that are related to them.
- The DOCUMENT SPLIT gives an alternative solution to the one in WHO READS WHAT, in case the architecture document is too voluminous and bulky. In this pattern, instead of delineating each part, the architectural document is divided into small manageable chunks, each one addressed to a specific stakeholder.
- COMPONENTS AND CONNECTORS is the most significant of all views, since they describe coarse-grained, run-time units of computation or data storage (components), interacting through special mechanisms (connectors). It is of paramount importance to treat connectors as first-class entities, just like components [28].
- The explicit INTERFACE SPECIFICATION for both COMPONENTS AND CONNECTORS is one of the most significant tasks in architectural documentation and deserves special attention.
- The RELATION BETWEEN VIEWS is performed after the CONTENTS OF VIEWS have been populated, and performs consistency tests between the views.
- The architect strives to achieve the STAKEHOLDERS VALIDATION, which happens if their concerns have been addressed by the corresponding CONTENTS OF VIEWS.
- QUALITY TRADEOFF ANALYSIS tackles the issues of performing a tradeoff analysis in order to decide on which qualities should be supported according to the prioritized STAKEHOLDERS AND THEIR CONCERNS.
- The ARCHITECTURAL RATIONALE should be captured in the CONTENTS OF VIEWS, so as to ensure the effective implementation of the system and its subsequent evolution.
- ARCHITECTURAL PATTERNS should be applied per view, in order to solve specific design problems in each view and at the same time capture the ARCHITECTURAL RATIONALE through the use of patterns.
- TOOL SUPPORT is an essential part of architectural documentation since it can automate some of the wearing tasks and facilitate for architectural analysis.

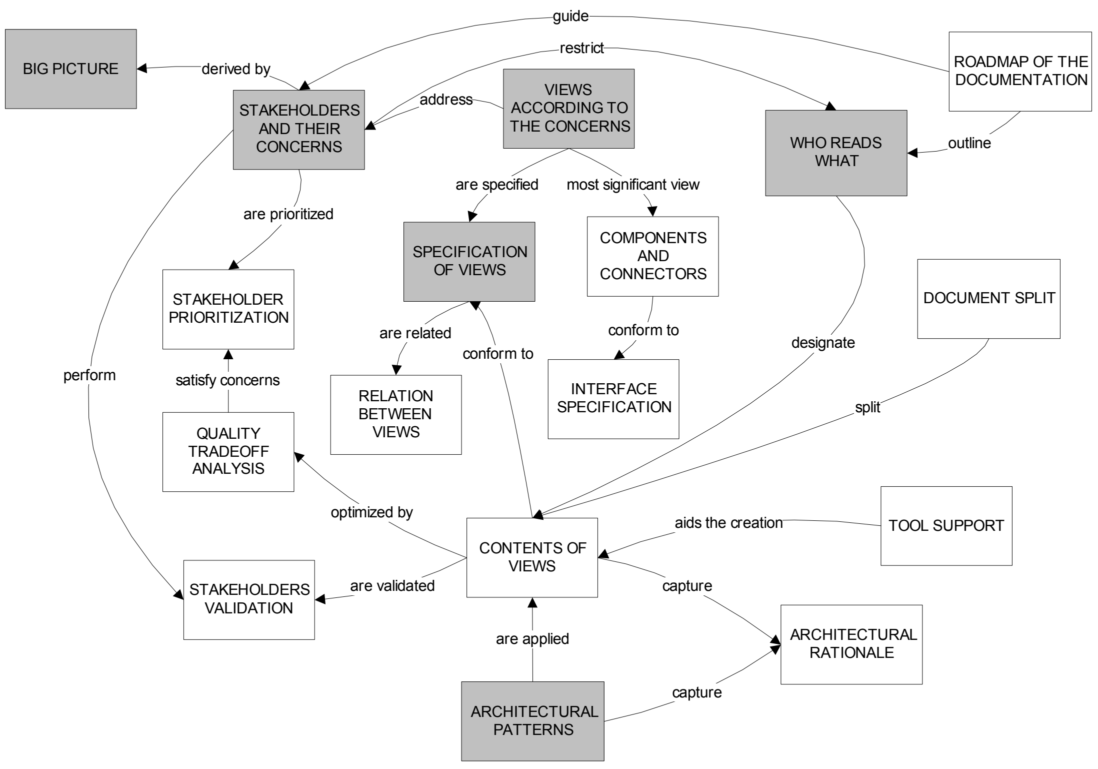


Figure 1 – Map of the pattern language and relationships between the patterns

4.1 BIG PICTURE

Context You just took up the task of documenting a software architecture.

Problem **Some software architects develop and evolve systems in isolation from the environment in which they will be integrated such as a company or an organization. Others attempt to take into account this environment but fail to do so effectively. Both cases result in unanticipated and often problematic influence of the environment to the software system and vice-versa. How do you manage the bi-directional dependency between the software system and its environment?**

Forces

- No software system is an island. On the contrary, a software system inhabits into an environment, is influenced by it and consequently affects it.
- An understanding of what the environment of a software system is and how they interact with each other is crucial, in order to effectively design and evolve the software system.
- There are several different factors that comprise the environment of a software system which may be technical, business, social, economic etc.
- The environment is sometimes too complex and sizeable and therefore it is too difficult or expensive to comprehend its internal workings.
- The environment is itself a dynamic system and may change over time.
- It is difficult to understand how exactly the environment influences the software

system, since it is not necessarily a software issue; it may not even be a technical issue but it could have socio-economic causes.

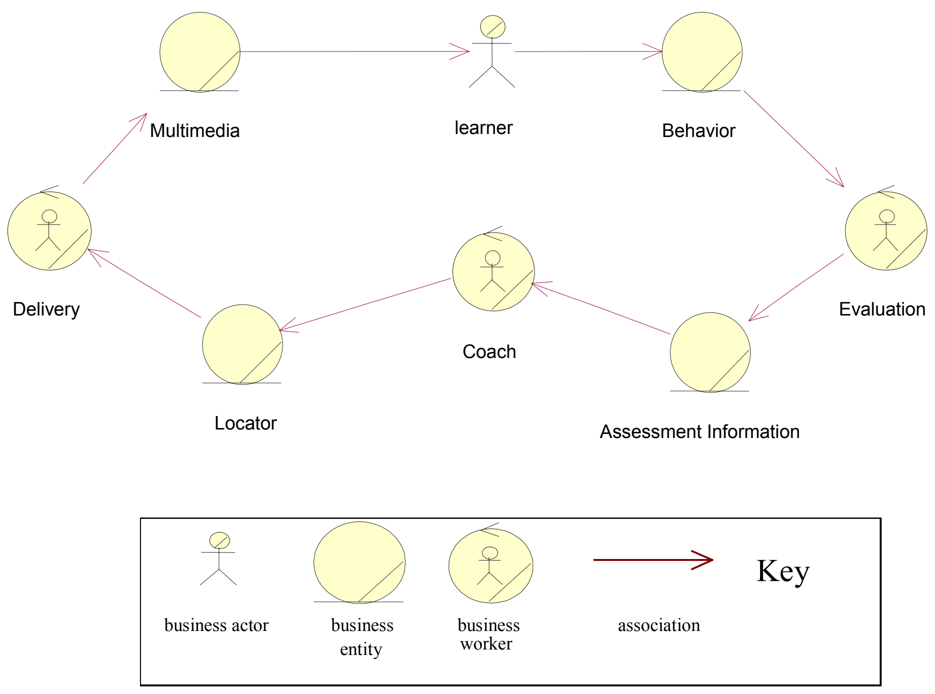
Solution **Therefore: Model the environment by focusing on all the significant issues that may influence the software system under development. Use this model as the starting point to derive functional and non-functional requirements for the software system. Ensure the consistency between the model of the environment and the rest of the software architecture during the evolution cycles.**

A common solution to defining the environment of a software system is *business modeling* or *domain modeling*, which is an engineering discipline concerned with modeling complex systems. There are also other approaches for defining the environment that influences the software system, such as Global Analysis [13]. As a minimum the definition of the environment should be comprised of: the **resources** (e.g. people, material, information, products), the **processes** (activities performed within the environment), the **goals** that are being served and the **rules** that constrain the environment. Emphasis should be given on the resources and the processes that are directly connected to the software system under development, since they can easily lead to the definition of the STAKEHOLDERS AND THEIR CONCERNS. It is also of paramount importance to specify the *interfaces* between the system with its environment, so that the points of interaction between the two are made clear. The different technical, business, social and economic matters can be modeled by splitting the description of the environment into different *views*, just like we perform a selection of VIEWS ACCORDING TO THE CONCERNS during the architectural description itself.

It should be expected that influence between software system and its environment is applied in both ways: first the environment shapes the development of the system; then the system is used inside the environment and affects it; subsequently the environment has an effect on the maintenance and next version of the software system and so on. If the environment is well recorded in the architectural description, this endless cycle of influencing can be better understood and managed. Of course, the model of the environment needs to be updated in each evolution cycle; otherwise it will be rendered obsolete.

Example In [1] we had demonstrated a business model for a Learning Management System, based on the Learning Technology Systems Architecture (LTSA) standard of the IEEE Learning Technology Standardization Committee. For that purpose we used the business modelling concepts from the Rational Unified Process and the Business Modelling Extensions of UML. We present as an example, a small part of the business model in Figure 2, namely the realization of the “unreliable learning” business use case, which shows that humans are considered unreliable learners and feedback systems may be required to guide them towards desirable learning behavior. It is called “the feedback and coaching loop”, through which, the required learning experiences are maximized and the detrimental are minimized. The *learner* receives *multimedia* information (e.g. web pages) from the *delivery* process (e.g. a web server) and expresses some *behavior* (e.g. gives answers to multiple-choice questions) that is assessed by the *evaluation* process (e.g. automatic evaluation of a multiple-choice quiz). The *coach* can be a person or a software system, e.g. an Intelligent Tutoring System. The *coach* may determine the "current position" of the learner from the *assessment information* that comes out of the

Evaluation and in sequence decide on appropriate action (e.g., delivery of particular learning content) to achieve the desired target (pedagogical objectives). The coach may then send *locators* (e.g., references to lessons, experimentation tools, suggestions) to the *Delivery* system in order to achieve the new targets. The *learner* is a business actor since in this case a learner is an external entity that interacts with and profits from the Learning Technology System by learning. The *Evaluation*, the *Coach* and the *Delivery* are business workers, i.e. abstractions that act within the business to realize the business processes. Finally *Multimedia*, *Behavior*, *Assessment Information* and *Locator* are all business entities that represent artefacts handled or used by the business workers.



- Figure 2 – the realization of the "unreliable learning" business use case

The application of the pattern entails the following positive and negative consequences:

Benefits

- The environment into which the software system inhabits is specified and the diverse factors are highlighted through different views such as business, social, technical etc.
- The influence of the environment upon the software system and vice-versa is documented.
- The model of the environment can be used for deriving the requirements for the software system.
- Some of the STAKEHOLDERS AND THEIR CONCERNS can be derived from the environment description.
- A single document, the architectural documentation contains both the description of the software system architecture and the environment into which it inhabits in.

Liabilities

- It is likely that the description of the environment will be somehow incomplete since there are factors of the environment that are impossible to predict beforehand.

- Documenting the environment of a software system may be considered an 'overkill' in some cases, especially in small or medium-sized systems. Modeling the environment is not a trivial process; it is a project on its own.

Pattern Sources The Rational Unified Process mandates that the environment of the software system should be defined using a business modeling technique and takes it to the next level by proposing to derive the functional requirements of the software system from the business model. IEEE 1471 standard [14] and other approaches from research and academia [3, 11, 13, 22], outline the importance of specifying the environment that influences the system under development. Clements et al. [9] propose the use of *context diagrams* to show the relation between the system and the environment it interacts with. Eriksson and Penker [11] suggest an approach for modeling the *business architecture* of a software system's environment through multiple views and UML extensions, and relating it to its software architecture. The Zachman framework [29] mandates that all views should be described from the 'business' perspective of the information system owner. Bosch [5] instructs the specification of the interfaces between the software system and its environment.

4.2 STAKEHOLDERS AND THEIR CONCERNS

Context You have defined the environment of the software system, so you have a good understanding of the BIG PICTURE.

Problem **Some software architecture documents are hardly understood by anyone, except for the architect, because s/he does not think of an intended audience. How do you specify the intended audience of the architectural documentation and how do you satisfy this audience?**

Forces

- An architectural documentation should be read and validated by all the people that have concerns over the systems, and they should be able to validate the architecture with respect to their concerns. The documentation should be written with an intended audience in mind.
- Several categories of people are interested in the system and have specific concerns about its development. They care about different things and look at the system from different angles, e.g. technical, financial, usage, managerial.
- The different concerns that different people have about the system may be contradictory by nature.

Solution **Therefore: Make an explicit list of all the different stakeholders of the software system and elicit their concerns. These system stakeholders also serve as the intended audience of the architecture documentation. Thus, use this list of stakeholders and concerns as a pivotal point for the architectural documentation and strive to achieve the STAKEHOLDERS VALIDATION as the definitive criterion for the success of the envisioned architecture.**

After seeing the BIG PICTURE, you have a fairly complete idea about all the 'key players' who have a saying in the development of the system. Starting from this set of 'key players' you can identify even more actors that have some concerns over the system. The stakeholders may include the architect(s), the developers, the clients and finally the end-users, who are not necessarily the same as the clients. Other stakeholders that can be taken under consideration are the project managers, the administration of the development organization, international standardization committees, reviewing or auditing committees, national or international legislation bodies and so on.

Concerns on the other hand should include all issues that are related to the system's development, maintenance and of course usage, as long as they are of importance to one or more stakeholders. A very important category of concerns that needs to be included are the *qualities* of the system under development, such as those described in [3]. For example the application developers may be concerned with the modifiability or the portability of the system; the application users may be concerned with the performance and the usability of the system; the project manager may be concerned with the cost and the time to market.

In some cases, it is not clear who are the *real* stakeholders of the system. For example in large enterprises, they may be implicitly defined by corporate practices and rules, that development teams are not aware of. To make matters worse, it is often difficult to extract the concerns from the stakeholders due to reasons of different background, lack of communication, inability of stakeholders to express them correctly etc. There is no 'silver bullet' for the actual elicitation of the stakeholders and their concerns, therefore development teams are encouraged to utilize their preferable requirements engineering method.

The goal of the elicitation of stakeholders and their concerns is twofold: to select the VIEWS ACCORDING TO THE CONCERNS and to achieve the STAKEHOLDERS VALIDATION of views so as to make sure their concerns are properly addressed. Naturally, not all stakeholders are supposed to read the entire architectural documentation, but you must explicitly specify WHO READS WHAT.

Finally, it should be expected that due to the different viewpoints that stakeholders look at the system, some of them will have contradicting concerns. This is a normal problem, and these concerns should be early identified, in order to perform a QUALITY TRADEOFF ANALYSIS when populating the CONTENTS OF VIEWS. A STAKEHOLDER PRIORITIZATION should also be performed before the tradeoff analysis in order to assign weights to the concerns, according to how important the stakeholders are.

Example In our case study, two significant categories of stakeholders considered for the development of a Learning Management System are defined as following:

- **Users of the system,** which include students, professors, administrative staff of the educational institute, teaching assistants, courseware authors, system administrators.
- **Acquirers of the system,** which include universities or in general higher educational institutions, K-12 educational institutions, and companies or organizations that perform employee training.

Two indicative *concerns* that the first category of stakeholders, i.e. the users have about the system are the following:

- What are the tasks or functionalities that the framework offers to the different categories of its users, e.g. courseware delivery, communication mechanisms, evaluation techniques?

- What is the usability of the system with respect to its different categories of users, e.g. professors setting up online courses or students engaging in learning activities?

The application of the pattern entails the following positive and negative consequences:

Benefits

- The stakeholders that have concerns over the system are identified and categorized and their point of view is specified. They constitute the audience of the architectural documentation.
- The concerns about the system are expressed from the stakeholders' point of view, and especially the qualities that play a critical role for the system development.
- The crucial tasks of selecting the VIEWS ACCORDING TO THE CONCERNS and performing STAKEHOLDERS VALIDATION will be based on the STAKEHOLDERS AND THEIR CONCERNS.

Liabilities

- There is always a possibility that one or more important stakeholders have not been discovered.
- Some important concerns may not have been identified, while some others may be misunderstood.
- The plethora and complexity of different concerns requires a thorough organization of the architectural documentation.
- The possible contradictory nature of certain concerns will inevitably lead to a QUALITY TRADEOFF ANALYSIS at a later point, which is usually challenging. Nevertheless, the STAKEHOLDERS VALIDATION may fail if they do not agree with this analysis, thus forcing another iteration of architectural documentation.

Pattern Sources

The IEEE 1471 standard recommends the specification of the stakeholders and concerns of the system under development. It mandates that at least, the users, acquirers, developers and maintainers of the system are identified and also prescribes a minimum set of concerns. Bass et al. in [3], Clements et al. in [9, 10] and the Open Group Architectural Framework propose the explicit identification of the stakeholders and their concerns. The Rational Unified Process focuses particularly on the identification of the stakeholder and the elicitation of their concerns. The Zachman framework [29] defines a fixed set of stakeholders for an information system and indicates the way that each view should be looked at from each of the stakeholders' perspectives.

4.3 VIEWS ACCORDING TO THE CONCERNS

Context

You have seen the BIG PICTURE comprised of the software system and its environment, and identified the system's STAKEHOLDERS AND THEIR CONCERNS. It is now time to look at the system per se.

Problem

In mature engineering disciplines, the architecture of a system under development is organized in views. In software engineering this often leads to

unfitting, awkward views that fail to support understanding and communicating the system architecture. How do you organize the architectural documentation of a complex software system in a set of views?

Forces

- Software systems can be overwhelmingly complex and multifaceted. Thus they can't be presented at a single glance; instead, they should be looked at from different views.
- Having a fixed set of views to document the architecture of your system, helps you to focus on the CONTENTS OF VIEWS, rather than on what views you should choose.
- The same set of views is not adequate to describe all systems. The complexity and diversity of software systems require a set of views that is customized to the individual needs of each system.
- Finding the right views to describe a system can be a complex task, especially if the architecture documentation is performed from scratch.

Solution

Therefore: Select a set of views that are specific to your software development project, so that these views address the concerns of the different stakeholders.

The set of views must be chosen on the basis of who are the STAKEHOLDERS AND THEIR CONCERNS. Since in every software development project, unique, custom stakeholders and concerns are defined, it is normal to also select different views in order to address these concerns. The goal is to select the views that will satisfy as many concerns of the stakeholders as possible. Most probably some concerns will be contradicting to each other and thus not all of them can be satisfied. In these cases, you must perform a QUALITY TRADEOFF ANALYSIS among these concerns, decide which concerns to favor at the expense of others and justify all that in the architectural documentation. At the end of the selection of the views you should verify that all concerns of all stakeholders are either addressed by at least one view, or given proper justification for not being addressed. Moreover, the architects need to explain the rationale behind choosing each view.

Some views that commonly appear in software architecture documentation concern the following:

- The run-time decomposition of the system in terms of components and connectors
- The dynamic behavioral aspects of the systems
- The processes, the threads and concurrency issues
- The functional requirements usually in the form of use cases
- The external environment that hosts the software system
- The code artifacts usually associated to logical artifacts from other views
- The data that is used in the system
- The deployment of the system into hardware and network components
- The project management and especially the assignment of tasks

During this selection phase, the views should be nothing more than abstract ideas of architectural artifacts, such as those mentioned above. As soon as they are selected though, you must perform the SPECIFICATION OF VIEWS, that is specifying their semantics in an unambiguous way.

Example In our case study, six views were selected to address the concerns of the stakeholders:

- The *use case* view, that shows how the system interacts with the external environment that it inhabits in.
- The *logical* view, that shows the decomposition and behavior of the system in a logical level of abstraction.
- The *implementation* view, that shows the artifacts of code that comprise the system
- The *data* view, that shows the persistent data that are stored and manipulated by the system.
- The *deployment* view, that shows the physical topology of the system.
- The *user experience* view, that shows the graphical user interface that end-users will operate.

The application of the pattern entails the following positive and negative consequences:

Benefits

- The complexity of the system is leveraged by organizing its description into multiple views.
- The right set of views is chosen since it corresponds to the stakeholders' concerns.
- The views selected to represent a system are customized to that particular system and are therefore able to better address the concerns in each case.

Liabilities

- It may be difficult in some cases to select the views, just by looking at the stakeholders' concerns.
- It is possible that there are no known views that address specific concerns and therefore the architects need to define them.
- In most cases not all concerns can be satisfied by the views selected, and therefore a QUALITY TRADEOFF ANALYSIS will be necessary.

Pattern Sources The IEEE 1471 standard recommends the definition of viewpoints that are the templates used to define the view. The standard does not prescribe any specific set of views but lets the architects free of choosing their own views according to the stakeholders and their concerns. The latest SEI approach on this issue also does not mandate a specific set of views, leaving it to the architect to decide, again with respect to the stakeholders and their concerns [9]. Rational's Unified Process uses a predefined set of views, namely Kructhen's 4+1 views [17, 18]. Other examples are the 4 views model proposed by a Siemens research team [13], and the 6 views refined over 5 stakeholders suggested by the Zachman framework [29].

4.4 SPECIFICATION OF VIEWS

Context You have selected the VIEWS ACCORDING TO THE CONCERNS of the stakeholders and you aim to produce the CONTENTS OF VIEWS.

Problem **The actual CONTENTS OF VIEWS are often created arbitrarily, in an ad-hoc fashion, according to the experience and the intuition of the architect. This can be a cumbersome task, leading to unambiguous interpretations by the system stakeholders. How do you know the precise semantics of a view and what exactly that view should contain?**

Forces

- Architects have *intuitive* ideas about what views should contain. For example they know that a structural decomposition view aims at a logical decomposition of a system into subsystems. However such anecdotal notions are not good enough and architects may interpret them in their own arbitrary way.
- Stakeholders need to read through the CONTENTS OF VIEWS in order to make sure that their concerns are being tackled. If they don't know the meaning and purpose of a view, they may not be able to understand the contents and the STAKEHOLDERS VALIDATION may fail.
- Views need to be unambiguously defined in order to allow for their exchange and reuse between different organizations, teams and projects.
- Deciding on what a view should contain can be a daunting task.

Solution **Therefore: Use a specification for each view, which should be precise enough to allow you to unambiguously produce its contents, and the stakeholders to comprehend these contents. Make sure the CONTENTS OF VIEWS conform to these specifications.**

Adopt a well-established specification for each view that matches the notion of that view in your particular project. If necessary, modify the view specification to bring it semantically closer to the specific needs of your project. If no such view specification exists, make your own specification based on relevant views and your architectural experience. The specification of views should be comprised of at least the following:

- **Metadata information**, version of the view definition, author, organization etc.
- **Stakeholders,** whose concerns are addressed by this view.
- **Concerns** that this view addresses
- **Rationale** that explains the precise way that the concerns are addressed by the viewpoint. This is one of the key aspects in specifying a view since it safeguards the fact that the viewpoint actually tackles the concerns it aims at.
- **Methods** that will be used by architects to author the contents of the view. These may include design techniques, notations, languages, analysis techniques for testing the artifacts, templates, standards, patterns or anything else that can be used in the view
- **RELATION BETWEEN VIEWS.** Usually views are not independent of each other but often have tight relationships between them. For example subsystems in the logical view are directly connected to code artifacts in the implementation view,

and the latter are associated with hardware and network nodes from the deployment view, onto which they are deployed. Architects provide to a large extent added value by explicitly showing these relationships between views since they provide insight into the entire architecture.

When the field of software architecture matures enough, it is expected, that there will be a plethora of libraries of views specification made available to public use. At present few libraries do exist, e.g. in [13, 17, 24]. For the time being, architects try to reuse some parts or entire specifications of views and probably customize them in order to fit the specifics of their own projects. It must be stressed that specifying views from scratch requires enormous effort and resources and in practice, it does not often takes place.

Example In our case study the specification of the *use case view* has the following form (the metadata information is omitted):

The *stakeholders* to be addressed by the view are the users, acquirers, and developers.

The *concerns* to be addressed by the view are:

- Who are the external entities that interact with the system?
- What are the tasks or functionalities that the system offers to those external entities actors?
- What are the relationships between the above tasks?

The constructed views shall use the UML as a modelling language and especially use-case diagrams. Specifically they will present a subset of the Use-Case Model, presenting the architecturally significant use-cases of the system; therefore they will contain a subset of the Software Requirements Specification (SRS) document. They will describe the set of scenarios and/or use cases that represent some significant, central functionality, as seen from external actors. They will also describe the set of scenarios and/or use cases that have a substantial architectural coverage (that exercise many architectural elements) or that stress or illustrate a specific, delicate point of the architecture.

This view addresses the aforementioned concerns in the following manner:

- The actors of the use case model represent the external entities that interact with the system and the use cases provide value to these actors.
- The use cases represent the functionalities that the system carries out in order to provide value to the actors.
- The relationship between the use cases is clearly specified in the use case model, either with general UML relationships between the use cases, or with more specific ones such as the <<extend>> and <<include>> stereotyped dependencies between use cases.

This view will provide inputs to the logical view, which will perform system modelling in a conceptual level. In specific, the logical view will show how the use cases are realized through a collaboration of classes and interfaces, both statically in the form of class diagrams and dynamically in the form of sequence/collaboration diagrams.

The source, for this view is the Rational Unified Process, v. 2001.03.00.23, Rational Software Corporation, part of the *Rational Solutions for Windows* suite, 2000.

The application of the pattern entails the following positive and negative consequences:

Benefits

- The architects understand how to create the CONTENTS OF VIEWS and the task of architecting is made easier.
- The stakeholders comprehend the CONTENTS OF VIEWS and can validate them, if their concerns have been addressed.
- The specification of the view is an adequate fit to the project needs.
- The view definitions are reusable across projects, teams and organizations

Liabilities

- The available libraries of view specifications are still very scarce.
- It is really difficult to perform the view specification on your own, especially with respect to what notations and languages should be used in each view.
- There is no guarantee that the documentation of the architecture will comply to the specification.

Pattern Sources The IEEE 1471 standard mandates the specification of the views into what they call *viewpoints*, which should contain, more or less, all the things prescribed in this pattern. Clements et al. in [9] suggest the specification of views in the form of *view templates* that also include the aforementioned issues.

4.5 ARCHITECTURAL PATTERNS

Context You are creating the CONTENTS OF VIEWS, always having in mind to address the STAKEHOLDERS AND THEIR CONCERNS. You are trying to solve several design problems in the system under development.

Problem **Applying architectural patterns can be problematic, especially when they should be distributed in the different architectural views. Often the architects are puzzled by where to apply which pattern, and sometimes they just re-invent the wheel. How do you apply architectural patterns while documenting the architecture?**

Forces

- Reusing architectural design experience is always an essential in software development, saving time and money and preventing from re-inventing the wheel. There are a number of architectural patterns [6] that solve recurrent problems in architectural design and can be reused in different contexts.
- The application of architectural patterns characterizes large portions of the system and helps to set up a common terminology between different stakeholders.
- It is not straightforward what is the relation between patterns and different views and how patterns can be utilized with respect to the views.

Solution **Therefore: Work on each view and try to identify architectural design problems that can be solved by architectural patterns, pertaining to this**

specific view. Explicitly document the patterns selected in each view, so as to capture the ARCHITECTURAL RATIONALE through the patterns and foster comprehension and communication of architectural decisions.

Architectural patterns should have a special place in the architectural documentation since they provide solutions to architectural-level problems, they capture the rationale of architectural decisions and they affect system-wide quality attributes. Each pattern is appropriate for a specific view and can be applied to organize the architectural elements in this view. For example, in the structural decomposition view, the 'Layered Systems' pattern [3, 6, 28] helps organize the logical subsystems hierarchically into layers, in the sense that subsystems in one layer can only reference subsystems on the same level or below. Therefore you should select certain architectural patterns for each view that solve the problems encountered in that particular view.

Record the architectural patterns being used in each view, so that stakeholders can easily distinguish them and reference them while communicating with each other. The goal is to use patterns as a common vocabulary, so that all stakeholders can speak the same language while discussing various aspects of the system. For example stakeholders should understand each other when mentioning that subsystem X adopts the 'Model-View-Controller' or that component Y is implemented with 'Pipes and Filters'.

The incorporation of architectural patterns in the documentation should be performed depending on the nature of the patterns and the views they belong to. In practice several notations and languages can be used: informally natural language or box-and-arrows diagrams can be used; if more formality is required, widely accepted languages such as the UML can be used; in case more precise semantics are required, Architecture Description Languages or other techniques from Software Engineering Formal Methods can be applied.

Example In our case study we have applied several architectural patterns in the logical view. In specific, the architectural patterns that have been used, as seen in the catalogue composed in [3, 6, 28] include: the *layered* style in the decomposition of the high-level subsystems, hierarchically into layers, in the sense that subsystems in one layer can only reference subsystems on the same level or below; the *Client-Server* style in several components and especially in the communication management components (e.g. e-mail, chat); the *Model-View-Controller* style in the Graphical User Interface design; the *blackboard* style in the mechanisms that access the database in various ways; the *event systems* style for notification of GUI components about the change of state of persistent objects.

The application of the pattern entails the following positive and negative consequences:

Benefits

- Architectural design experience in the form of architectural patterns is reused saving precious resources.
- ARCHITECTURAL RATIONALE is captured through the application of patterns.
- A common language is established among stakeholders for the description of the architecture using patterns.
- In each view, there are several architectural patterns applied, that solve

problems pertinent to the specific view.

Liabilities

- Associating specific design problems with architectural patterns in specific views can be quite difficult.
- There may be certain views that no architectural patterns can be applied.

Pattern Sources Beck and Johnson in [4] first argued that patterns generate architectures by providing an explicit way to document design decisions. Buschmann et al. in [6] initiated the field of pattern-oriented software architecture and strongly argue for documenting architectures with patterns as building blocks. Clements et al. in [9] suggest that categories of patterns, which they call *styles*, correspond to the categories of views that are being used to document the architecture. The Rational Unified Process mandates the use of patterns in documenting the architecture and their explicit enclosure in the architectural documentation. Several other sources in the literature [3, 5, 13, 16, 22, 28] have given great emphasis on the use of patterns in the architectural documentation.

4.6 WHO READS WHAT

Context You have created all the CONTENTS OF VIEWS and aim to submit the document for STAKEHOLDERS VALIDATION.

Problem **Having all the information in a single architectural document makes it difficult for stakeholders to understand it. How do you communicate the architectural documentation to all the stakeholders so that it can be comprehended by them?**

Forces

- An architectural documentation is a complex, voluminous, technical document.
- It is crucial to achieve the STAKEHOLDERS VALIDATION of the views by having them read through the architectural documentation and agreeing or consenting that their concerns are addressed. If their concerns are sacrificed in favor of other concerns, the stakeholders should be provided with a proper justification.
- The stakeholders come from different backgrounds, such as technical, financial, or management environments and do not all 'speak the same language'. They also may not have the time to carefully study the entire documentation.
- If too many technical details are in the architectural documentation, there is an increased risk that stakeholders will not understand it.
- If technical details are omitted from the architectural documentation, then it loses its value as a guide of understanding, evaluating, developing and evolving the system.

Solution **Therefore: Decide on which part of the architectural documentation should be read by which stakeholders and include clear indications to delineate this. Make sure that the stakeholders have the knowledge and background to understand the details of each part that is related to them.**

The criteria for deciding if a stakeholder should read a part of the document is whether s/he has a concern that is being addressed in this part. Also make sure that a part of the document that is of interest to a specific stakeholder contains explicit information on how exactly the stakeholder's concern is addressed.

Include a ROADMAP OF THE DOCUMENTATION that should be read by all stakeholders because they contain general information, e.g. information on the document's structure, an overview of the architecture etc. This roadmap is a good place to insert details about which parts different stakeholders should read in the architectural documentation. Nevertheless, if the whole Software Architecture Document is too large and bulky, you should consider a DOCUMENT SPLIT into smaller manageable texts that can be better handled by the stakeholders.

Often there are large sections of the document, e.g. a whole view, that are of interest to several stakeholders, but not all of them are either concerned or even capable of understanding the whole section. In that case it is better to try and separate the parts that contain individual pieces of information that address specific concerns, and denote them to be read by the corresponding stakeholders. Do not let the stakeholders try to extract information from large parts of text and diagrams, especially when they don't have the background to comprehend all of them.

Example In our case study, the learners are the most significant users of the system and they are considered to lack technical knowledge in software development issues, which renders most of the architecture documentation inappropriate. So in the architectural documentation there are specific parts highlighted that are meant to address their concerns and also are written in a non-technical fashion. For example their concern in the functionalities performed by the system is addressed in the use case view, where there is a particular section that outlines the use cases as a number of steps that users need to carry out. Another concern of paramount importance to learners is the usability of the system, which is addressed mainly in the user experience view, by showing screenshots and thus demonstrating the graphical user interface design. Learners, of course, can't entirely evaluate the usability of the system by looking at screenshots, but can at least have an indicating view of the user interface, even before the system is developed.

The application of the pattern entails the following positive and negative consequences:

Benefits

- STAKEHOLDERS VALIDATION of the views can be performed since they read only the parts of the documentation that address their concerns, and can be assured that their concerns are tackled.
- Stakeholders have the necessary background to understand the parts of the architectural documentation they are reading.
- Stakeholders only read a fraction and not the whole architectural documentation and thus are not required to spend too much time or effort on it.
- The architectural specification contains all the technical details that are necessary for a rigorous architectural documentation, and these will be read by only the appropriate stakeholders.

Liabilities

- Separating the parts of the documentation with respect to what stakeholder should read what part can be difficult and time-consuming
- There are parts of the documentation that can't be distinctly separated because the information is overlapping.

Pattern Sources The IEEE 1471 standard addresses the issue by denoting in each viewpoint, the stakeholders whose concerns are addressed in this viewpoint as well as the concerns themselves. Clements et al. in [9] suggest showing to each stakeholder only what he needs to know in each view and in what detail. They also introduce the notion of a *view packet* as the smallest collection of architectural documentation artifacts that should be shown to a specific stakeholder. The Zachman framework [29] instructs that each view is specifically written for each of the five stakeholder categories, involved in the development of an information system.

5 Conclusions

The field of software architecture is still immature and currently there are more questions asked than answered. The task of documenting software architecture has a long way ahead before it becomes a systematic, disciplined practice based on sound engineering principles. This pattern language, intended for software architects, has made an early effort to codify commonly accepted concepts and practices in the field of documenting software architectures, in the form of patterns. Even though the contents of these patterns have been written in numerous textbooks and research papers and have been discussed in conferences and workshops for years now, we believe that their recording in the form of pattern yields the following advantages:

- The patterns offer valuable experience in small digestible chunks in favour of software architects, especially inexperienced ones, which lack the time to read complete textbooks and research papers.
- The patterns offer the compilation of experience in software architecture documentation collected from different sources from academia, industry and international bodies. Interested parties can find everything in one place rather than processing multiple heterogeneous material on the subject.
- We have tried to find common ground between the various approaches that have been proposed in order to express concepts and practices in a uniform way thus eliminating differences in terminology and viewpoint. Each pattern presented in this paper has at least three sources that have proposed the same concepts and practices.
- The standardized description format that patterns follow, is usually more appealing to people as opposed to textbooks and research papers which are not easily and comfortably read by everyone.

Acknowledgements

We thank all those who have collaborated with us in various architecture-centric projects, and helped us put the patterns to action. We also thank our EuroPLOP 2004 shepherd, Jorge L. Ortega Arjona, for his constructive and insightful feedback and the writers' workshop members, especially Kevlin Henney, Frank Buschmann, Andreas Rüping, Wolfgang Herzner, and Arno Haase for a truly intense, valuable and encouraging session.

References

1. P. Avgeriou, S. Retalis, N. Papaspyrou, "Modeling a Learning Technology System as a Business System", *Software and Systems Modeling*, Volume 2, No. 2, pp 120-133, Springer-Verlag, July 2003.
2. P. Avgeriou, Describing, Instantiating and Evaluating a Reference Architecture: A Case Study, *Enterprise Architect Journal*, Fawcette Technical Publications, http://www.ftponline.com/ea/, June 2003.
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, 1998.
4. Beck, K. and Johnson, R., Patterns Generate Architectures, In European Conference on Object-Oriented Programming 1994, pages 139-149. Springer-Verlag Lecture Notes in Computer Science 821.
5. Bosch, J., Design and Use of Software Architectures. Addison-Wesley, 2000.
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, John Wiley & Sons, 1996.
7. Clements, P., Kazman, R., Clein, M., Evaluating Software Architecture, Addison-Wesley, 2002.
8. Clements, P., "A Survey of Architecture Description Languages", *Proceedings of the 8th International Workshop on Software Specification and Design*, pp. 16-25, Schloss Velen, Germany, 22-23 March 1996.
9. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.
10. Clements, P. and Northrop, L., Software Product Lines - practices and patterns, Addison-Wesley, 2002.
11. Eriksson, H. and Penker, M., Business Modeling with UML, Business Patterns at work, John Wiley and Sons, 2000.
12. Garland, J. and Anthony, R., Large Scale Software Architecture, John Wiley & Sons, 2003.
13. Hofmeister, C., Nord, R. and Soni, D., Applied Software Architecture, Addison-Wesley, 1999.
14. IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE std. 1471-2000, 2000.
15. ISO/IEC 10746-1, 2, 3, 4 | ITU-T Recommendation X.901, X.902, X.903, X.904. "Open Distributed Processing - Reference Model". OMG, 1995-98.
16. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999.
17. Kruchten, P., "The 4+1 view model of architecture", *IEEE Software*, November 1995.
18. Kruchten, P., The Rational Unified Process, An introduction, Second Edition Addison-Wesley, 2001.
19. Medvidovic, N., Taylor, R.N., "A classification and comparison framework for software architecture description languages". *IEEE Transactions on Software Engineering*, vol.26, (no.1), p.70-93, Jan. 2000.

20. Medvidovic, N., Rosenblum, D., Redmiles, D. and Robbins, J., “Modelling Software Architectures in the Unified Modeling Language”, *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 1, pages 2-57, January 2002.
21. N. Medvidovic, P. Avgeriou, and N. Guelfi, editors. UML’04 Workshop on Software Architecture Description & UML, Lisbon, Portugal, October 2004.
22. Nord, R., Tomayko, J. and Wojcik, R., Integrating Software-Architecture-Centric Methods into Extreme Programming (XP), Carnegie Mellon University Technical Report, CMU/SEI-2004-TN-036, September 2004.
23. Open Group (The), The Open Group Architectural Framework Version 8.1, http://www.opengroup.org/, December 2003.
24. Putman, J., Architecting with RM-ODP, Prentice Hall, 2001.
25. The Rational Unified Process, Rational Software Corporation, part of the Rational Solutions for Windows suite, 2001.
26. Robbins, J., Medvidovic, N., Redmiles, D.F. and Rosenblum, D.S., “Integrating architecture description languages with a standard design method”. *Proceedings of the 1998 International Conference on Software Engineering*, 1998.
27. Rueping, A., Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects, John Wiley & Sons, 2003.
28. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice Hall, 1996.
29. Zachman, J., A Framework for Information Systems Architecture, IBM Systems Journal, Volume 26, Number 3, 1987.