

Patterns for documenting software architectures

Paris Avgeriou, Nicolas Guelfi, Reza Razavi

*Software Engineering Competence Center (SE2C), University of Luxembourg,
6, rue Richard Coudenhove-Kalergi L-1359 Luxembourg-Kirchberg, Luxembourg
{paris.avgeriou, 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 architectural design in the early phases of the lifecycle is of paramount importance to the project success [1, 3, 5, 10, 11, 12, 16, 23]. 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 [7].

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; on the contrary we are still at early stages of addressing it [7].

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 [12]. 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 their broader dissemination to the software development 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 an example inside the patterns. The fourth section contains a catalog of patterns for this field and the fifth section outlines some more patterns that remain as future work. Finally the sixth section wraps up with conclusions derived from this work.

2 Literature Review

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** [12]. 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)** [13, 20]. 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. [1], Shaw and Garlan [23] and Bosch [3] 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 [4]), reference models and reference architectures. Clements et al. in [7] 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. Hofmeister et al. [11] propose a set of 4 views for documenting the architecture assisted by the use of the Unified Modeling Language. Furthermore, the Rational Unified Process (RUP) [16, 21], 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 designing 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 [6, 1, 17]. 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 Unified Modeling Language as an ADL, either by extending it per se, or by mapping existing ADLs onto it [18, 22].

3 The example used in the patterns

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 Patterns for documenting software architectures

Figure 1 depicts the relationships between the proposed patterns and the order in which they should be applied. The semantics of the arrows between the patterns is that the pattern at the beginning of the arrows should be applied before the patterns at the end of the arrows. It is noted that in the description of the patterns, the pattern names are in uppercase letters, so as to distinguish them inside the text.

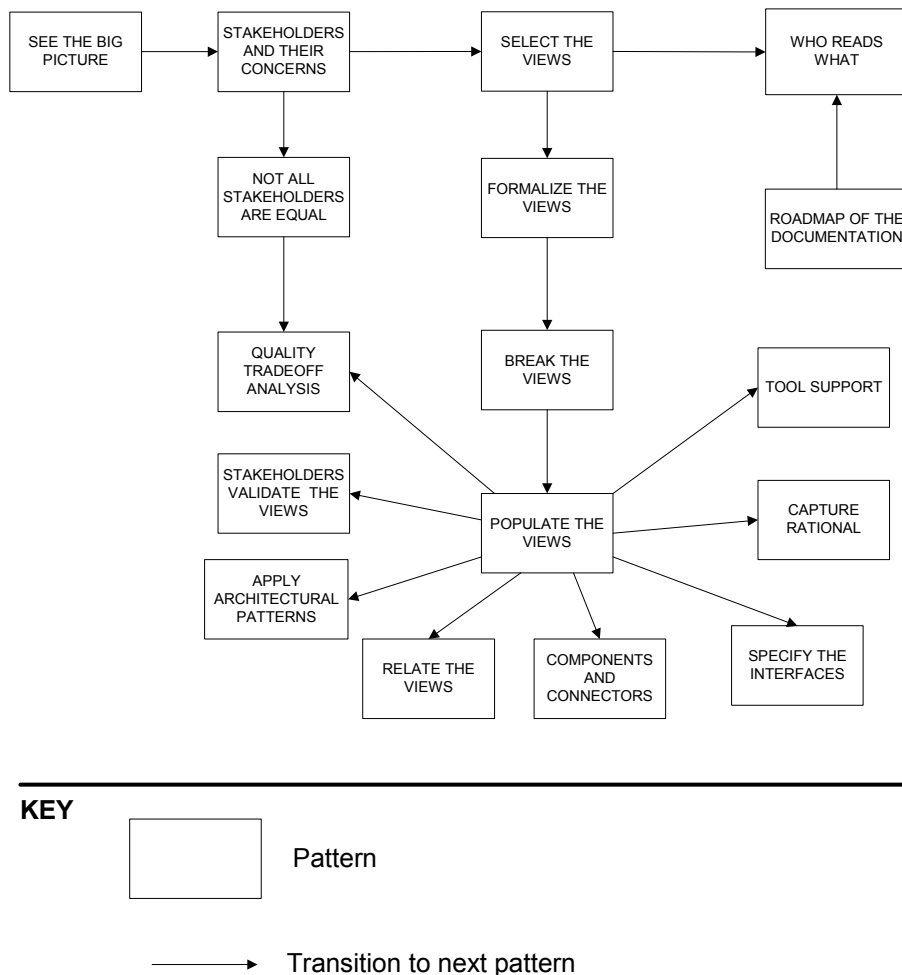


Figure 1 - The patterns and their ordering

4.1 SEE THE BIG PICTURE

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

Problem: No software system is an island. On the contrary, a software system inhabits into an environment and is influenced by it. How can you take into account this environment?

Forces:

- There are several different factors that comprise the environment of a software system.
- It is difficult to understand how exactly the environment influences the software system.
- The environment should be recorded in the architectural documentation so that its influence upon the documentation is established.

Solution: *Define the environment by explicitly specifying all the issues that are important and can influence the system under development.*

The environment of a software system includes technical, business, social and economic issues. 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 [12]. 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. The different technical, business, social and economic matters can be modeled by splitting the description of the environment into different *views*, just like we break up the architectural description into separate views.

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.

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 the Unified Modelling Language. We present as an example, a small part of the business model in Figure 2, namely the human resources that interact with the Learning Management System in a web-based e-learning scenario. The *student* is considered to be a business actor, which interacts with the Learning Management System, and is a specialization of the LTSA Learner, which can specify his/her own learning preferences. The *professor* and the *tutor* are both business workers that extend the Coach process, perform some of the Evaluation process functions and manipulate the Performance, Learning Preferences and Assessment Information entities. The Evaluation process apparently assesses the Performance of the learner with

respect to a learner activity and produces Assessment Information. In turn, the Coach process can determine the "current position" of the Learner and decide on appropriate action to achieve the desired learning target. Finally the *system administrator* is also a business worker and a specialization of the Coach process as s/he performs some complementary administrative tasks.

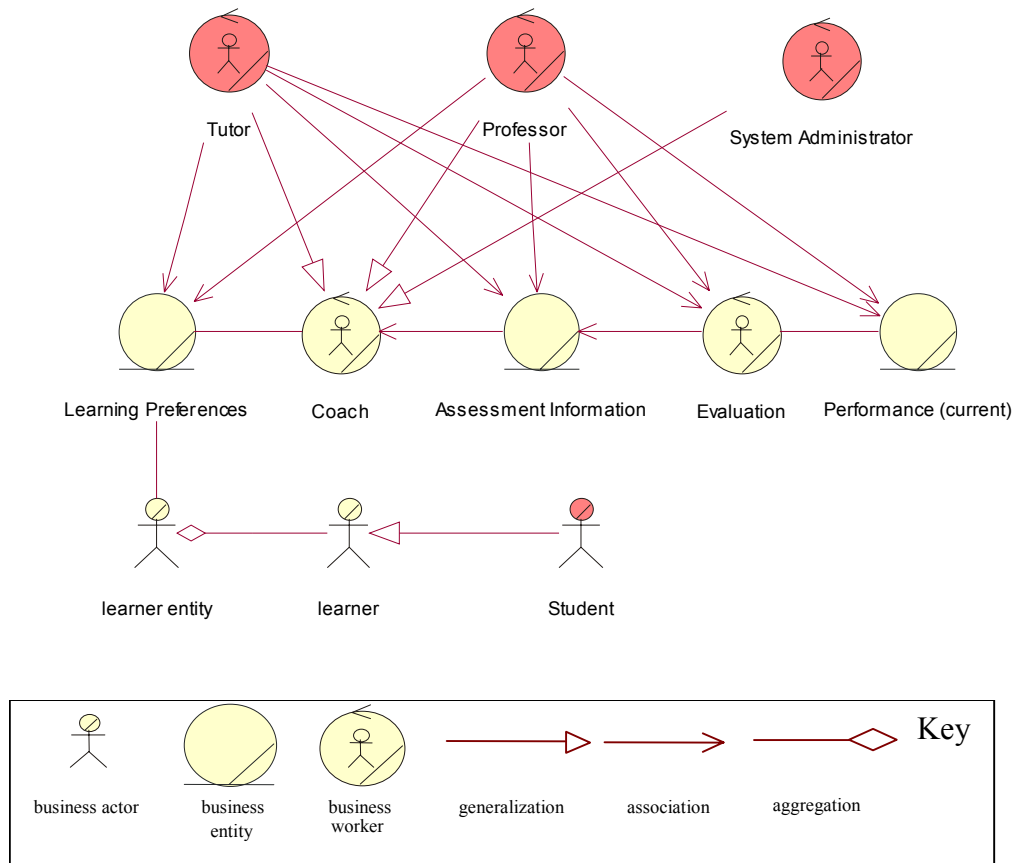


Figure 2 – The human subsystem related to the LTSA business model elements

Consequences:

- + The environment into which the software system inhabits is formally 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 is documented
- + The STAKEHOLDERS AND THEIR CONCERNS are easily identifiable 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.
- The description of the environment before the development of the system, may be rendered obsolete since the environment is itself a dynamic system and may change over time
- 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.

Known uses: 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 [12] and other approaches from research and academia [1, 9, 11, 19], outline the importance of specifying the environment that influences the system under development. Clements et al. [8] propose the use of *context diagrams* to show the relation between the system and the environment it interacts with. Eriksson and Penker [10] 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.

Related Patterns: STAKEHOLDERS AND THEIR CONCERNS

4.2 STAKEHOLDERS AND THEIR CONCERNS

Context: You have defined the environment of the system and it is time to look at who has concerns over the system and what are these concerns.

Problem: Several categories of people are interested in the system and have specific concerns about its development. How do you make sure that their interests are addressed in the system under development?

Forces:

- Stakeholders care about different things and look at the system from different angles, e.g. technical, financial, usage, managerial.
- In some cases it is not clear *who* are the stakeholders of the system. For example in large enterprises, stakeholders may be implicitly defined by corporate practices and rules, that development teams are not aware of.
- When trying to elicit concerns from the stakeholders, we may not know what we are looking for and thus ask them the wrong questions.
- Just as in requirements engineering, 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.
- Sometimes the concerns of the stakeholders are contradictory by nature.

Solution: *Explicitly identify the stakeholders and elicit their concerns.*

Each stakeholder should have at least one concern in order to be included in the stakeholders list. After SEEING THE BIG PICTURE, the architect has a fairly complete idea about all the key players who have a saying in the development of the system. As a minimum, the stakeholders should 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 [1]. 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.

There is no 'silver bullet' for the actual elicitation of the concerns from the stakeholders, therefore development teams are encouraged to utilize their preferable requirements engineering method.

Stakeholders are intended to read and validate the contents of the architectural documentation so as to make sure their concerns are properly addressed. Naturally, not all stakeholders are supposed to read the entire architectural documentation, but the architect 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 VIEWS of the architecture.

Example:

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

- a) **Users of the system**, which include students, professors, administrative staff of the educational institute, teaching assistants, courseware authors, system administrators.
- b) **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?

Consequences:

- + The stakeholders that have concerns over the system are identified and categorized and their point of view is specified
- + Stakeholders express their concerns about the system from their point of view
- 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 tradeoff analysis at a later point

Known uses: 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 [1], Clements et al. in [7, 8] 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.

Related Patterns: SELECT THE VIEWS, SEE THE BIG PICTURE, WHO READS WHAT.

4.3 SELECT THE VIEWS

Context: You have defined the environment of the system and identified the stakeholders and their concerns. It is now time to look at the system per se.

Problem: Software systems can be overwhelmingly complex and multifaceted. How can you manage the modeling of such systems?

Forces:

- The system can't be presented at a single glance since it is too complex and multifaceted; instead, it should be looked at from different views.
- Finding the right views to describe a system can be a complex task.
- The same set of views are not adequate to describe all the systems.

Solution: *Select different views so that the system can be explored from different angles and all the concerns of the different stakeholders are addressed by these views.*

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, the architect 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 the architect 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 static structural decomposition of the system in terms of components and connectors
- The run-time, 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, the architect must FORMALIZE THE VIEWS, that is specifying their exact contents in a formal way.

Example:

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

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

Consequences:

- + 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
- 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 tradeoff analysis will be necessary

Known uses: 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's decision, again with respect to the stakeholders and their concerns [7]. Rational's Unified Process uses a predefined set of views, namely Kruchten's 4+1 views [15, 16]. Another example is the 4 views model proposed by a Siemens research team [11].

Related Patterns: STAKEHOLDERS AND THEIR CONCERNS, FORMALIZE THE VIEWS, STAKEHOLDERS VALIDATE THE VIEWS

4.4 FORMALIZE THE VIEWS

Context: You have selected a set of views you are going to use.

Problem: How do you formally specify what a view should contain in order to make sure it correctly addresses the concerns of the stakeholders for which it is aimed at?

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 this is not enough for architects to design complex software systems.
- Stakeholders need to read through the contents of the views in order to validate that their concerns are being tackled. Without a formal specification of the views they may not be able to understand the semantics of the contents.
- Views need to be formally 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:

Specify the views in a formal manner so that the architects are assisted in designing the system, the stakeholders can unambiguously comprehend them, and they can be reused in other contexts.

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 to other 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. The relationship between views should be performed outside their formal specification as indicated in RELATE THE VIEWS.

Finally let it be noted that, when the field of software architecture matures enough, it is expected, that libraries of views definitions will be composed and made available to public use. When that time comes, architects will be able to reuse these definitions of views and

probably customize them in order to fit the specifics of their own projects. At present few such definitions do exist, e.g. in [11, 15, 20].

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:

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

The constructed views shall use the Unified Modeling Language as a modeling 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.

Consequences:

- + The architects understand how to perform the architectural documentation in each view and the task of architecting is made more easier.
- + The stakeholders comprehend the contents of the views and can verify whether their concerns are addressed.
- + The view definitions are reusable across projects, teams and organizations

- Even if the above template is strictly respected, it is still quite difficult to perform the actual specification, 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.

Known uses: The IEEE 1471 standard mandates the formalization 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 [7] suggest the formal definition of views in the form of *view templates* that also include the aforementioned issues.

Related Patterns: STAKEHOLDERS AND THEIR CONCERNS, SELECT THE VIEWS, RELATE THE VIEWS, QUALITY TRADEOFF ANALYSIS.

4.5 WHO READS WHAT

Context: You have designed the architecture, neatly organized into views.

Problem: An architectural description is a complex, voluminous, technical document. How do you communicate it to all the stakeholders so that it can be comprehended by them?

Forces:

- It is crucial to make sure that stakeholders read through the architectural documentation and agree or consent 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 study carefully 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 and developing the system.

Solution: *Include clear indications in each part of the architectural documentation, with respect to which stakeholders should read this specific part. 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.

Naturally there will be 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.

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.

Consequences:

- + Stakeholders read only the parts of the documentation that address their concerns, and can validate the tackling of their concerns
- + 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 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.
- 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 so that different stakeholders can go through different parts.

Known uses: 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 [7] 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.

Related Patterns: SELECT THE VIEWS, STAKEHOLDERS AND THEIR CONCERNS, ROADMAP OF THE DOCUMENTATION

4.6 APPLY ARCHITECTURAL PATTERNS

Context: You are designing the system according to the specification of the views.

Problem: How do you apply architectural patterns while designing and 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 [4] 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 how patterns can be utilized with respect to the views selected.

Solution: *Employ architectural patterns in each view that entail solutions to system design problems particular to that view, and effectively document them so as to foster comprehension and communication.*

Architectural patterns should have a special place in the architectural documentation since they provide solutions to architectural-level problems, and thus affect system-wide properties. 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 [1, 4, 23] 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 the architect 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 documentation of architectural patterns in the designs 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, 5, 24] 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.

Consequences:

- + Architectural design experience in the form of architectural patterns is reused saving precious resources
- + A common language is established among stakeholders for the description of the architecture using patterns
- + Architectural patterns are employed in those views, whose interaction of elements they are meant to describe
- 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

Known uses: Buschmann et al. in [4] initiate the field of pattern-oriented software architecture and strongly argue for documenting architectures with patterns as building blocks. Clements et al. in [7] 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 designing the architecture and their explicit enclosure in the architectural documentation. Several other sources in the literature [1, 3, 11, 14, 19, 23] have given great emphasis on the use of patterns in the architectural documentation.

Related Patterns: SELECT THE VIEWS

5 Future patterns

In the future we also intend to write or finalize the following patterns:

- **ROADMAP OF THE DOCUMENTATION**, which suggests providing a roadmap on the documentation with hints on how it should be read by the stakeholders.
- **COMPONENTS AND CONNECTORS**, which implies that the core of the architectural documentation should be a view describing components interacting through connectors. It is of paramount importance to treat connectors as first-class entities, just like components [24].
- **SPECIFY THE INTERFACES**, which instructs that the explicit specification of the interfaces of both components and connectors is one of the most significant tasks in architectural documentation.
- **RELATE THE VIEWS**, which deals with relating views among themselves, after they are populated with elements, and performing consistency tests between them.
- **BREAK THE VIEWS**, which tackles the complexity of each view by decomposing it into manageable chunks.
- **POPULATE THE VIEWS**, which guides the architect into filling the views with the necessary elements.
- **CAPTURE RATIONALE**, which mandates the justification behind each design decision taken inside the views.
- **STAKEHOLDERS VALIDATE THE VIEWS**, which urges the architect to engage the stakeholders in the decision-making process of selecting the views.
- **TOOL SUPPORT**, which prescribes the automation of architectural documentation tasks with the support of CASE tools
- **QUALITY TRADEOFF ANALYSIS**, which tackles the issues of performing a tradeoff analysis in order to decide on which qualities should be supported and which should not. This pattern is related to **NOT ALL STAKEHOLDERS ARE EQUAL**, which indicates that stakeholders who are more important than others should have greater priority in getting their concerns addressed.

6 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 paper 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 people interested in the field but reluctant or short of time for reading complete textbooks.

- 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 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.

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. Bosch, J., *Design and Use of Software Architectures*. Addison-Wesley, 2000.
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. and Stal, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.
6. Clements, P., Kazman, R., Klein, M., *Evaluating Software Architecture*, Addison-Wesley, 2002.
7. 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.
8. 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.
9. Clements, P. and Northrop, L., *Software Product Lines - practices and patterns*, Addison-Wesley, 2002.
10. Eriksson, H. and Penker, M., *Business Modeling with UML, Business Patterns at work*, John Wiley and Sons, 2000.
11. Garland, J. and Anthony, R., *Large Scale Software Architecture*, John Wiley & Sons, 2003.

12. Hofmeister, C., Nord, R. and Soni, D., Applied Software Architecture, Addison-Wesley, 1999.
13. IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE std. 1471-2000, 2000.
14. 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.
15. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999.
16. Kruchten, P., "The 4+1 view model of architecture", *IEEE Software*, November 1995.
17. Kruchten, P., The Rational Unified Process, An introduction, Second Edition Addison-Wesley, 2001.
18. 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.
19. 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.
20. Open Group (The), The Open Group Architectural Framework Version 8.1, <http://www.opengroup.org/>, December 2003.
21. Putman, J., Architecting with RM-ODP, Prentice Hall, 2001.
22. The Rational Unified Process, Rational Software Corporation, part of the Rational Solutions for Windows suite, 2001.
23. 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.
24. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice Hall, 1996.