Building Quality into Learning Management Systems - an Architecture-Centric Approach

Paris Avgeriou[1], Simos Retalis[2] and Manolis Skordalakis[1]

[1] National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, Zografou, Athens, 15780, GREECE
E-mail: {pavger, skordala}@softlab.ntua.gr,
[2] Department of Computer Science, University of Cyprus, 75 Kallipoleos St., P.O. Box, 20537, CY-1678 Nicosia, CYPRUS, E-mail: retal@softlab.ntua.gr

Abstract. The design and development of contemporary Learning Management Systems (LMS), is largely focused on satisfying functional requirements, rather than quality requirements, thus resulting in inefficient systems of poor software and business quality. In order to remedy this problem there is a research trend into specifying and evaluating software architectures for LMS, since quality attributes in a system depend profoundly on its architecture. This paper presents a case study of appraising the software architecture of a Learning Management through experience-based assessment and the use of an architectural prototype. The framework of the evaluation conducted, concerns run-time, development and business qualities. The paper concludes with the lessons learned from the evaluation, emphasizing on the compromise between them.

Keywords: Software architecture, architectural evaluation, quality attributes, quality requirements, nonfunctional requirements, Learning Management System, Unified Modeling Language, Unified Process.

1 Introduction

Governments, authorities and organizations comprehend the potential of the Internet to transform the educational experience and envisage a knowledge-based future where acquiring and acting on knowledge is the primary operation of all life-long learners. In order to realize this vision, the use of **Learning Management Systems** is being exponentially augmented and broadened to cover all fields of the new economy demands. LMS are software systems that synthesize the functionality of computer-mediated communications software (e-mail, bulletin boards, newsgroups etc.) and on-line methods of delivering courseware (e.g. the WWW) [1].

LMS that are in use today are either commercial products (e.g. WebCT, Blackboard, Intralearn), or customized software systems that serve the instructional purposes of particular organizations. The design and development of LMS though, is largely focused on satisfying certain *functional* requirements, such as the creation and distribution of on-line learning material, the communication and collaboration between the various actors and so on. On the contrary, the *quality* requirements of LMS are usually overlooked and underestimated. This is due to the fact that even though

quality is always of prime interest to the software vendors, they usually give priority to functionality because it is more tangible and a better argument for marketing purposes. In other words, LMS vendors are competing in a race of implementing as much functionality as possible. This is rather obvious in LMS comparative evaluations, where only functionality is evaluated, and quality requirements are completely ignored [2]. This naturally results in inefficient systems of poor software and business quality. Problems that typically occur in these cases are: bad performance which is usually frustrating for the users; poor usability, that adds a cognitive overload to the user; increased cost for purchasing and maintaining the systems; poor customizability and modifiability; limited portability and reusability of learning resources and components; restricted interoperability between LMS.

The question that arises is how can these deficiencies be remedied, i.e. how can the quality attributes be incorporated into the LMS being engineered? Quality attributes in a software system depend profoundly on its architecture and are an immediate outcome of it [3, 4, 5, 6]. Therefore the support for qualities such as performance, security, availability, and usability should be designed *into the architecture* of the system [5, 6, 7]. These principles have only recently been widely accepted and adopted and have lead to a research trend into defining software architectures that support quality attributes in the field of LMS [8, 9, 10].

Similarly, the key idea behind our endeavor is to design for quality. In specific, this paper presents a case study of applying an evaluation framework to the software architecture of a Learning Management System so that quality can be built inherently into the system. The latter is achieved by appraising the quality of the architecture, in each development iteration, and using the feedback to re-design the architecture in order to enhance the quality of the system. For that purpose, certain criteria, as well as heuristics derived from experience, are adopted for assessing the quality attributes of the system under development and indicating, "where the system is at", in terms of quality. A most significant assistant in this evaluation is an architectural prototype of a Learning Management System that has been engineered to implement the architecturally important design decisions. The conclusions inferred from the evaluation process, concern an estimation of each criterion, complemented with appropriate justification. Furthermore, the evaluation interestingly reveals the compromise between the quality requirements, as they are very tightly inter-connected and are either in conflict or in accordance with each other.

The structure of the paper is as follows: Section 2 very briefly demonstrates the proposed architecture and the architectural prototype. Section 3 introduces an evaluation framework with certain methods and quality attributes and moves on to present the results of the quality evaluation. Finally Section 4 contains conclusions, as well as future plans.

2 A Learning Management System Architecture

The proposed architecture is a result of a prototype *architecting process* that is characterized of five key aspects: it is founded on the higher-level architecture of IEEE LTSC Learning Technology Systems Architecture [8]; it uses a prototype architecture

of a Web-based Instructional System [11] to build a complete business model and refine and constrain the requirements for the Learning Management System; it adopts and customizes a big part of the well-established, software engineering process, the Rational Unified Process (RUP) [7, 12]; it uses the widely-adopted Unified Modeling Language [13, 14] to describe the architecture; and it is fundamentally and inherently component-based. The latter is justified by the fact that great emphasis has been put, not only in providing a pure component-based process, that generates solely components and connectors, but also in identifying the appropriate binding technologies for implementing and integrating the various components. Further study of the architecting process can be found at [15].

2.1 The Architectural Description

The first and most sizeable part of the architectural description is the views of the 5 models dictated by the RUP. Due to lack of space, it is not practical to illustrate even a small representative sample of the numerous diagrams produced in the 5 models. A rather extensive description of the architectural description can be found at [16]. Instead we will only provide the first-level decomposition of the system, by specifying the coarse-grained discrete subsystems in the design model. The decomposition is combined with the enforcement of the "Layered Systems" architecture pattern [17, 18, 19], which helps organize the subsystems hierarchically into layers, in the sense that subsystems in one layer can only reference subsystems on the same level or below. The RUP utilizes the aforementioned architectural pattern by defining four layers in order to organize the subsystems in the design model.

The proposed layered architecture is depicted in Figure 1, which, besides identifying all first-level subsystems and organizing them into layers, also defines dependencies between them, which are realized through well-specified interfaces. The plethora of dependencies between the different sub-systems is indicative of the complexity of LMS. The architectural description continues to decompose each one of these subsystems into smaller subsystem until it reaches the 'tree leaves', i.e. individual classes. Of course, in every subsystem identified, we also design its required and provided interfaces, as well as interaction diagrams that depict the run-time behavior of that subsystem.

Additional issues of the architecture description, such as the legacy systems, the commercial software, the architectural patterns to be used etc. are also quite important for the evaluation to follow and are outlined as following. In the proposed architecture there are a few legacy systems, such as some communication components and some courseware delivery components, but fortunately they were all written in the Java programming language, and thus were relatively easy to integrate into the new system. As far as the commercial systems, we have adopted several of them such as the mySQL RDBMS [http://www.mysql.com] and the Resin Web Server and Servlets engine [http://www.caucho.com] etc. The architectural patterns that have been used, as seen in the catalogues composed in [17, 18, 19] include: the *layered* style as aforementioned; the *Client-Server* style has been used extensively, especially in the communication management components; the *Model-View-Controller* style in the GUI design, which is inherent in all Java Swing UI components; the *blackboard* style in

the mechanisms that access the database in various ways; the *Virtual Machine* and the *object-oriented* style which are both a result of the implementation in Java; the *event systems* style for the notification of GUI components about the change of state of persistent objects.

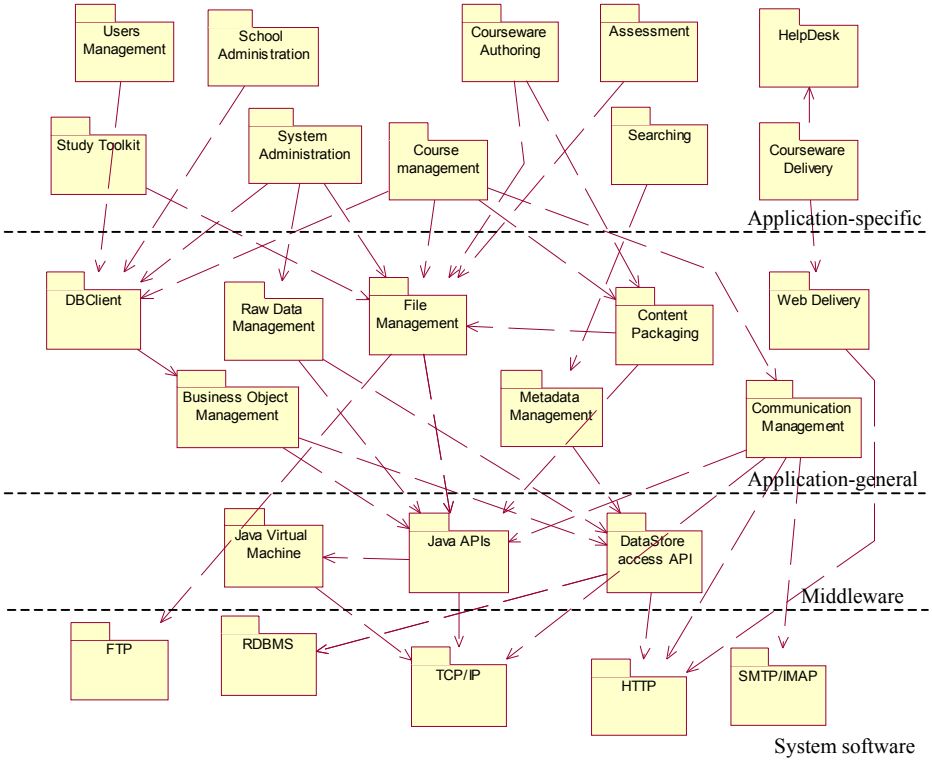

Fig. 1. The layered, component-based architecture of a Learning Management System

2.2 The Architectural Prototype

An architecture is a visual, holistic view of the system, but it is only an abstraction. In order to evaluate the architecture in terms of the quality attributes it promotes, we must build a significant part of it. Therefore, the software architecture must be accompanied with an **architectural prototype** that implements the most important design decisions sufficiently to validate them - that is to test and measure them [3, 6, 12]. The architectural prototype is the most important artifact associated with the architecture itself, which illustrates the architectural decisions and help us evolve and stabilize the architecture.

Therefore, in order to assess and validate the proposed architecture, a prototype was engineered, named “Athena” that implements the main architectural elements. Our choice between Java and Microsoft platforms was the former because it is an open technology, rather than proprietary, and based on a Virtual Machine, thus pro-

moting portability. The specific technologies used are applets, servlets, Java Beans, Enterprise Java Beans, Java Server Pages, as well as the JFC/Swing, RMI, JDBC, 2D Graphics, JMF and JAF Java APIs. The eXtensible Markup Language (XML) was used as the default language for the representation of data that were not stored in the database. About 75% of the total number of components have been implemented or acquired and put into operation, even though some of them do not offer the complete functionality prescribed in the system design.

Finally there was an attempt on adopting international standards within the various components in order to promote interoperability of LMS. For that purpose we have developed the metadata management component conforming to the IEEE LTSC Learning Object Metadata working standard [20] and the assessment component in order to adopt the IMS Question and Testing Interoperability Standard [21].

3 Evaluating the Architecture for Quality

3.1 Theoretical Underpinnings

Software Architectures cannot be classified as either inherently good or bad; instead they are either more or less appropriate to achieve some declared objectives. Therefore architectures can be evaluated according to specific criteria and are designed to fulfill certain quality attributes [3, 6, 19]. It is noted that no quality can be maximized in a system without sacrificing some other quality or qualities, instead there is always a trade-off while choosing on supporting the different quality attributes [3, 6, 19].
The question is how to evaluate the quality attributes of architectures since they are not tangible products but abstract designs that came from the minds of architects. One solution would be to measure the qualities after the system is built but there is an obvious disadvantage in that: it usually takes such an amount of resources to re-engineer the system in order to better support certain qualities, that it is unrealistic to perform [3]. Therefore, since it is too expensive to fix up a system when it is completed, we need to find a way to evaluate the qualities of the system before it is constructed.

The answer to this problem is the *assessment techniques* that have been especially created for the purpose of evaluating the quality attributes of architectures *before* they are implemented into real systems. Therefore these techniques do not estimate the qualities of the actual system, but rather measure the potential of the architecture to fulfill the required quality attributes. For that purpose, in [19] they propose the method of architecture reviews, as well as the Software Architecture Analysis Method (SAAM), which is better demonstrated in [22]. In [23] the Architectural Tradeoff Analysis Method (ATAM) studies the tradeoff between the different quality requirements in architectural evaluation. In [6] the authors perform a thorough and comparative presentation of architecture evaluation through the SAAM, ATAM and ARID methods. In [3], the author identifies the following methods for assessment of software architectures with respect to quality attributes:

- **Scenario-based evaluation** – to evaluate a specific quality attribute, a set of scenarios is created that captures the meaning of that particular attribute.

- **Simulation** – where the main parts of the application are developed, while the rest are only simulated, providing an overall executable system. Therefore the system under evaluation is an implementation of the complete software system at a high level of abstraction.
- Another approach, similar to the simulation method is to use an **architectural prototype**, where only parts of the application are implemented and executed. The simulation and the architectural prototype methods are best for evaluating operational quality attributes, that is qualities that can be measured at the system's run time.
- **Mathematical modeling** – where special-use mathematical models are devised and formalized in order to evaluate quality attributes, especially the ones that concern the operation of the system.
- **Experience-based assessment** - which is rather an intuitive approach based on former experiences of the architects and reasonable argumentation. Even though this is not a formal technique, it is very often used, since the experience of the architects, especially in a certain domain, is priceless, particularly when it is supported by the appropriate line of reasoning.

Regarding the quality attributes themselves, there is also a plethora of qualities proposed by various researchers as well as international standards [24, 25]. Fortunately, these sets of qualities that have been proposed, revolve around the same concepts, even when they are named differently. Probably the most comprehensive catalogue of qualities is given in [19], where four different categories of these qualities are identified:

1 System quality attributes discernable at runtime:

a. **performance** – the responsiveness of the system, the time required to respond to stimuli (events) or the number of events processed in some interval of time. This quality depends highly on the communication and interaction, taking place between components.
b. ***security*** – the system's ability to resist unauthorized attempts at usage and denial of service while still providing its services to legitimate users. It can be strengthened by incorporating specialized components into the system such as authentication servers.
c. ***availability*** – the proportion of time the system is up and running. It is measured by the length of time between failures as well as by how quickly the system is able to resume operation in the event of failure. It can be enhanced by duplicating critical components and connectors that take over when the primary ones fail, and by closely monitoring the system to detect failure. It also depends on the separation of concerns between the components, as well as their modifiability. A closely related quality is ***reliability***, the ability of the system to keep operating over time.
d. ***usability*** – this quality is comprised of other partial qualities: how quick and easy is it for a user to learn to use the system's interface (**learnability**)? Does the system respond with appropriate speed to a user's request (**efficiency**)? Can the user remember how to do system operations between uses of the system (**memorability**)? Does the system anticipate and prevent common user errors (**error avoidance**)? Does the system help the user recover from errors (**error handling**)? Does the system make the user's job easy (**satisfaction**)? Since usability is con-

cerned with human-computer interaction (HCI) issues, the flow of information to the user through the various components is of great significance to this quality attribute. Also the modifiability quality generally assists in achieving usability. Finally efficiency is directly linked to the system's performance.

2 **System quality attributes not discernable at runtime (development qualities):**

a. ***modifiability –*** the ability to make changes quickly and cost-effectively. It is also widely known as **maintainability**. It relies heavily on locality of change, which in turn depends on the encapsulation of functionality and the coupling between components through dependencies.

b. ***portability*** – the ability to run under different computing environments. It depends on the existence of a layer that is interposed between the application and the environment.

c. ***integrability*** – the ability to make the separately developed components of the system work correctly together. It is governed by specification of the components interfaces and their interactions, as well as the separation of concerns between them. A special case of integrability is ***interoperability****:* the ability of a system to work with another system.

d. ***reusability*** – the ability to reuse the system's structure or some of its components again in future applications. It is related to how coupled each component is with the rest; the loosely-coupled components are more reusable. Also the modifiability of the system entails reusability.

e. ***testability*** – the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. It is determined by the level of architectural documentation, the separation of concerns and information hiding.

3 **Business qualities:**

a. ***time to market -*** It is reduced when pre-built components such as Commercial Off The Shelf (COTS) products are purchased or reused from existing development projects. Of course the issue of inserting pre-built components is a matter of integrability.

b. ***cost.*** It can be reduced by reusing pre-existing assets such as components.

c. **projected lifetime of the system -** This quality attribute can be supported if the system scores well on the modifiability and portability attributes. If the system is modifiable and portable it has an extended lifetime but there is also an increase in the time-to-market quality.

d. ***targeted market -*** This is also a quality that depends on other quality attributes, such as portability, usability, performance and of course the functional requirements that are out of the scope of this paper.

3.2 Evaluation of Quality Attributes

The evaluation framework that we shall use to assess the architecture is based on the methods and attributes described in the previous subsection. More specifically the methods used are the 3rd and the 5th, i.e. the evaluation results of the architectural prototype, as well as architectural experience combined with the appropriate line of reasoning.

3.2.1 System Quality Attributes Discernable at Runtime

1. **performance** - This attribute is compromised by the use of the 'layered systems' architectural pattern, which, even though causes the system to be flexible and modifiable, brings a lot of overhead due to inter-component communication. So performance is naturally limited because of the layered nature of the system. The use of Java has an effect on performance as well, since it is an interpreted language. However, by putting a lot of functionality on the client, i.e. implementing a thick client, the system 's performance is greatly enhanced since there is limited client-server communication overhead. In addition, Java performs comparatively better than other similar technologies, like for instance, CGI scripts, where every operation leads to at least an HTTP request. Java applets perform much better since the performance bottlenecks are limited to downloading the bundled classes. So this attribute could be evaluated as fair enough.
2. ***security*** - the sole precaution taken in order to improve security of the system, is choosing communications ports to be non-standard HTTP ports, and place the system behind a firewall so as to block unauthorized requests. On the other hand there is no provision in the architecture about denial of service or IP source address spoofing attacks. Therefore the system is rather vulnerable to attacks.
3. ***availability*** – according to the implementation model of the architecture, there are 7 different server components (application server, WWW server and servlets engine, FTP server, E-mail server, RDBMS, Chat server, Whiteboard server) and they are all independent of each other. Therefore the failure of one server component does not affect the others. Good practice would also be to disperse the server components in different workstations, so that the crashing of one workstation will not affect the others and further improve availability, though that would cause extra communication overhead. However there are no redundant components foreseen in the architecture to take over when the primary ones fail, or an error reporting mechanism. In conclusion the system has a mediocre availability.
4. ***usability*** – This quality attribute is probably the most difficult to assess in terms of the system's architecture, because it concerns the user interface and is mostly subjectively appraised. In general we could claim that the flow of information to the user is straightforward, correct and complete. Efficiency is not rated highly due to the corresponding performance insufficiency. More evaluation results in this quality should be made available when the prototype is tested within its context of use, i.e. with students participating in Open and Distance Learning courses.

3.2.2 System Quality Attributes not Discernable at Runtime

1. ***modifiability*** – Modifiability is met by the proposed architecture since the component-based nature of the system causes it to be inherently modular, making **dependencies explicit** and helping to reduce and control these dependencies [4]. This means that a component can be changed to improve or adapt its functionality if necessary, or it could even be replaced by another new and better component without affecting the overall system. In other words, since the component interfaces are clearly defined, components can be treated as black boxes and a change in a component will not propagate changes to the other components it interacts with. That is after all what locality of change is all about. Even if changes need to occur to a society of components instead of a single one, this society can still be isolated so that

changes to it are made transparently to the rest of the system. Another argument for the good modifiability of the LMS architecture is that the architectural design and the implementation of the system are both performed in object-oriented languages, so if changes occur in either the design or the code it is trivial to transfer them to the other. Furthermore, except from the component nature of the architecture, the layered structuring also leads to separation of concerns and therefore to locality of change. To sum up, the architecture scores pretty high in this quality.

2. ***portability*** – The architectural prototype is to some extent portable since both the client-side and the server-side code are written in Java, which is an interpreted platform-independent language. In other words the Java Virtual Machine plays the role of a portability layer between the Learning Management System and the environment. Of course, in reality, Java does not run on all platforms and therefore 100% portability can't be achieved. In addition, a lot of the GUI is also written in standard HTML, which is apparently platform-independent. As far as the third-party components, such as the MySQL RDBMS or the Resin Web Server and servlets engine, they have also been chosen to be portable or available in multiple platforms. Therefore the architecture can be claimed to be acceptably portable.
3. ***integrability*** – This is also a quality that is satisfied because of the component-based nature of the system, the explicit definition of components and connectors, the predefined protocols of component interaction and the clearly defined interfaces of the different components. In cases where the interfaces of the components under integration are incompatible and cannot be changed for various reasons, e.g. they are COTS products, methods such as gap analysis [26] have been used to leverage the incompatibility. The layered structure of the system also assists in partitioning the functionality into separate components and thus promoting integrability. Finally, since the legacy systems were all written in Java, they did not have to be re-written or wrapped inside Java wrappers, but it did take some adaptation to make them interoperable with the new components.
4. ***reusability*** – According to the same arguments as in the modifiability and integrability quality, the components developed within the proposed architecture, having clearly defined functionality and interfaces, and thus being loosely-coupled, can be reused in different applications, may they be other LMS or not. This was an anticipated result, since reusability and modifiability tend to support each other and the system was evaluated as highly modifiable.
5. ***interoperability*** - This quality attribute is satisfied by the fact that, not only internal component interfaces are identified, i.e. interfaces that allow the system's components to interoperate, but also external ones. For example the Metadata Management System has an external interface, defined as a Java API, that can be used to import or export sets of metadata that conform to an international standard, in our case IEEE LTSC Learning Object Metadata [20].
6. ***testability*** – The proposed architecture promotes testability in a considerable degree for the following reasons: the design is made using object-oriented UML constructs that have a one-to-one mapping to the code, making the architectural documentation clearly articulated and illustrating the exact system built; therefore the testers can understand exactly where the error is caused and why. Furthermore, the concepts of information hiding and separation of concern that have been achieved in the component design, lead to tracing of faults to unique components. Again,

sources of errors are easy to distinguish inside a society of interoperating components. On the other hand, the kinds of errors that have to do with the overall system operation, such as system-wide failures, deadlocks in process synchronization etc. cannot be tested explicitly with the proposed architecture, but rather implicitly by creating test cases from the corresponding use cases. These are, of course, huge classes of errors, but unfortunately don't depend on software architecture.

3.2.3 Business Qualities

1. ***time to market***. Instead of developing all the components from scratch, some of them were located as COTS products, as seen in the architectural prototype description, and that has affected in a great reduction in time to market. Of course the time of integrating COTS in the system is still not minimal, since it takes time to search for them and customize the rest of the system so that they can be properly integrated. Fortunately they were relatively easy to integrate, thanks to the component, layered nature and pre-defined interfaces, as explained above.
2. ***cost***. The use of COTS has also reduced the cost of the system under development. It is noted though that for the sake of our architectural prototype, the COTS were not purchased, since their license allows their use for non-commercial or instructional purposes. If they were indeed bought, then there would probably be a considerably added cost. It is speculated though that still the cost of COTS is less than the cost of developing them from scratch.
3. ***projected lifetime of the system***. Since the system was evaluated to be quite modifiable, it will manage change easily and thus extend its lifetime. Additionally the portability of the system allows for it to claim a bigger share in the market and establish itself in many platforms, thus having better possibilities to last longer.
4. ***targeted market***. Since the system was evaluated highly in the quality attribute of portability, and fairly enough in usability and performance, it is estimated that the system has increased potential for a good market share.

4 Conclusions and Future Work

There is little doubt anymore that a well-specified architecture is able to build quality inherently into a system [4, 6, 7, 12]. Software architecture allows for the evaluation of the system before it is built, thus saving a lot of resources that would have otherwise been unnecessarily spent. It assists the architect into making the right design decisions to correct the development process and finally to achieve the target qualities.

The general conclusion derived from the evaluation presented in this paper, is that the proposed architecture scores pretty high as far as the development qualities are concerned, but it fails to adequately meet most of the run-time qualities. The business qualities are somewhere in the middle: the architecture achieves an acceptable score in the business section. This result makes sense from an architectural point of view, since the development qualities are often in direct conflict with the run-time qualities while, on the other hand, development qualities usually promote business qualities. The controversy between the development and the run-time qualities are further documented in these remarks:

1. The layered nature of the system supports modifiability and integrability but has a considerable cost on performance since there is a lot of communication overhead between independent components.
2. The use of the Java programming language has a negative effect on performance since it is an interpreted language. On the other hand being an interpreted language and relying on a virtual machine, Java is platform-independent, thus allowing portability to an extent. Moreover, Java allows for a direct mapping, from the object-oriented architectural design into the implementation language, thus leading to increased modifiability of the system.

Conversely, the mutual support between the development and the business qualities is illustrated in the following observations:

1. The use of COTS and other third-party components is feasible due to the high integrability and modifiability of the system. This in turn promotes the business qualities, such as reduced time to market as well as reduced cost.
2. The system's modifiability guarantees the effective management of change, therefore it promotes an increased lifetime. Portability also promotes the system's lifetime as well as its targeted market.

It is rather evident that the various qualities of the system are quite mingled and inter-dependent and might support or diminish one another. It is the job of the architect to try and maximize the more desirable ones, and at the same time, minimize the consequent effect for the less desirable qualities. This is quite a challenging problem with many daunting tradeoff issues, but it could be performed more easily and systematically with the adoption and use of a formal evaluation method that provides more hard data and quantifiable results.

Another conclusion is that the evaluation method based on the architectural prototype is best for evaluating quality attributes discernable at runtime. On the other hand experience-based assessment, fits better with development qualities such as modifiability, portability etc.

The work presented in this paper is part of research conducted on the software engineering of a Learning Management System, with emphasis on software architecture. Future work in this area initially includes the adoption of a custom, formal evaluation method to assess the quality attributes and produce more accurate, solid results as well as tradeoff analysis. Furthermore, the feedback from the evaluation presented in this paper is being used to re-engineer the system in order to improve some of the low-score quality attributes. It is of paramount importance to inspect the methods, as well as the effort required to re-engineer the system under development. Finally the adoption and use of architectural patterns [17] will also be investigated with respect to the effect such patterns have on the quality attributes.

References

1. Oleg, S., Liber, B.: A framework of pedagogical evaluation of Virtual Learning Environments. Available online at [http://www.jtap.ac.uk/reports/htm/jtap-041.html], (1999).
2. Avgeriou, P., Papasalouros A. and Retalis, S.: Web-based learning Environments: issues, trends, challenges. Proceedings of the 1st IOSTE symposium in Southern Europe, Science and Technology Education, Paralimni, Cyprus, (2001).

3. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, (2000).
4. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, (1999).
5. Eriksson, H. and Penker, M.: Business Modeling with UML - Business Patterns at work. John Wiley & Sons, (2000).
6. Clements, P., Kazman, R., Clein, M.: Evaluating Software Architecture. Addison-Wesley, (2002).
7. Jacobson, I., Booch, G. and Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, (1999).
8. IEEE Learning Technology Standards Committee: Draft Standard for Learning Technology Systems Architecture (LTSA). Draft 9, (2001).
9. Thorne, S., Shubert, C., Merriman, J.: OKI architecture overview. OKI project document, (2002).
10. Cisco Systems: Blueprint for Enterprise E-learning. white paper, (2002),
11. Retalis S. and Avgeriou P.: Modeling Web-based Instructional Systems. Journal of Information Technology Education, Volume 1, No. 1, pp. 25-41, (2002).
12. Kruchten, P.: The Rational Unified Process, An introduction. Addison-Wesley, (1999).
13. Booch, G., Rumbaugh, J., and Jacobson, I.: The UML User Guide. Addison-Wesley, (1999)
14. Rumbaugh, J., Jacobson, I. and Booch, G.: The UML Reference Manual. Addison-Wesley, (1996).
15. Avgeriou, P., Retalis, S., Papasalouros, A., Skordalakis, M.: Developing an architecture for the Software Subsystem of a Learning Technology System – an Engineering approach. Proceedings of International Conference of Advanced Learning Technologies, Madison, Wisconsin, IEEE Computer Society Press, (2001), pp. 17-20.
16. Avgeriou, P., Retalis, S., Skordalakis, M.: A Software Architecture for a Learning Management System. Post-proceedings of the 8th Panhellenic Conference in Informatics, to be published in the Lecture Notes in Computer Science series, Springer-Verlag, (2002).
17. Buschmann, F., Meunier, R., Rohnert, H., Sommertland, P. and Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons, (1996).
18. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, (1996).
19. Bass, L., Clements, P. and Kazman, R.: Software Architecture in Practice. Addison-Wesley, (1998).
20. IEEE Learning Technology Standardization Committee (LTSC): Draft Standard for Learning Object Metadata, P1484.12/D6.1. http://ltsc.ieee.org, (2001).
21. IMS Global Learning Consortium: IMS Question & Test Interoperability Specification-Best Practice and Implementation Guide, version 1.2.1. http://www.imsproject.org/, (2001)
22. Kazman, R., Abowd, G., Bass, L,; & Clements, P.: Scenario-Based Analysis of Software Architecture. IEEE Software 13, 6, (1996), pp. 47-55.
23. Kazman, R., Klein, M., Clements, P.: ATAM: Method for Architecture Evaluation. TECHNICAL REPORT CMU/SEI-2000-TR-004 ESC-TR-2000-004, (2000).
24. IEEE: Recommended Practice for Software Requirements Specifications, IEEE Std. 830-1993. (1993)
25. ISO/IEC 9126: Information technology-Software product evaluation-Quality characteristics and the guidelines for their use. (1993).
26. Cheesman, J. and Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, (2000).

Acknowledgement. The work described in this paper was performed as part of the MENU project (Model for a European Networked University), which is partly funded under contract NO001ELEARN011of the European Community.