MOLDING ARCHITECTURE AND INTEGRITY MECHANISMS EVOLUTION
An Architectural Stability Evaluation Model for Software Systems

Octavian-Paul Rotaru
University "Politehnica" Bucharest Romania

Keywords: Architecture Evolution, Data Integrity, Mathematical Model, Perturbation Theory, Requirements, Software Architecture, Stability.

Abstract: The architectural stability of a software system is a measure of how well it accommodates the evolution of the requirements. The link between integrity mechanisms and application's architecture starts right from the moment the requirements of the application are defined and evolves together with them. The integrity mechanisms used will evolve when the application's requirements are modified. Apart from the possible architectural changes required, adding a new requirement to an application can trigger structural changes in the way data integrity is preserved. This paper studies the architectural stability of a system based on an integrity oriented case study and proposes a mathematical model for architectural evaluation of software systems inspired from the perturbations' theory. The proposed mathematical model can be used to mold the evolution of any software system affected by requirements changes; to find the architectural states of the system for which a given set of requirements is not a trigger (doesn't provoke an architectural change); and to find the architectural configuration which is optimal for a given set of requirements (evolves as less as possible).

1 INTRODUCTION

"All mankind is of one author, and is one volume; when one man dies, one chapter is not torn out of the book, but translated into a better language; and every chapter must be so translated... as therefore the bell that rings to a sermon, calls not upon the preacher only, but upon the congregation to come: so this bell calls us all: but how much more me, who am brought so near the door by this sickness.... No man is an island, entire of itself... any man's death diminishes me, because I am involved in mankind; and therefore never send to know for whom the bell tolls; it tolls for thee." (John Donne, Meditation XVII, from "Devotions upon Emergent Occasions", 1624)

This famous meditation of Donne's elaborates on a major idea of the Renaissance, valid in life as well as in science that people are not isolated from one another, but mankind is interconnected.

Similarly software systems all over the world are nowadays interconnected. Home computers, servers, applications, databases, modules and components are all connected in a way or the other, they are interacting and they are influencing each other.

When an Internet user uploads a file to a server the operation influences the web server in many ways. An upload routine is employed in order to get the file on a temporary location; the database access module is afterwards called in order to store the file into the database; a mailing daemon will send an email to the user in order to notify him that the upload ended successfully; etc. Maybe some other servers are also notified that the file is uploaded, for example a search engine in order to index it, or a load balancing module is inquired to decide what server to use. The file upload operation can also influence the load of the server, or can limit its available space, and these are just a few examples of what can happen behind the scene.

Not only systems are influencing each other, but also at a smaller scale different characteristics of a software system are doing the same.

Usually data integrity is perceived as an a priori service offered by the database. However, data integrity assurance is not referring only to the integrity of the information stored in the database, but to the integrity of all the data flows transferred to and from the application or between the different application's modules as well.

Rotaru O. (2006).
MOLDING ARCHITECTURE AND INTEGRITY MECHANISMS EVOLUTION - An Architectural Stability Evaluation Model for Software Systems.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - ISAS*, pages 426-431
DOI: 10.5220/0002469704260431
Copyright © SciTePress

There are many available integrity assurance mechanisms and strategies that can be used by application. The choice is clearly influenced, even if not unique, by the specificities of each application. Apart from the application characteristics, the environment and the exterior interactions are also influencing the choice of the optimal integrity strategy. Finally, it all depends on the requirements, implemented in the application, that are influencing the architecture and its interactions, and consequently the way the data integrity is realized.

Integrity is not a database characteristic, but a characteristic of any software system. By data integrity I am designating the integrity of the data in the entire computational context, seeing the system as an ensemble from which no piece can be ignored.

2 CASE STUDY

2.1 Overview

The link between integrity mechanisms and application's architecture starts right from the moment the requirements of the application are defined and evolves together with them. The integrity mechanisms used will evolve whenever the application's requirements are modified. Apart from the possible architectural changes required, adding a new requirement to an application can trigger structural changes in the way data integrity is preserved.

The architecture's influence on the integrity assurance mechanism will be examined based on a case study on an application that will evolve in time by varying its functional and non-functional requirements. At each stage the influence of the requirements on the architecture and in turn on the integrity control mechanisms will be evaluated. It is worth noting that the description of each stage of the case study will also contain context related information, and not only the requirements.

The stages can be considered versions created during the development of the application that are created keeping in mind only the current requirements, ignoring the future requirements that will be introduced at later stages. The way requirements are incrementally added without considering the big picture of all the requirements that will be added in each and every version is not a good development practice, but it serves the purpose of this case study. Usually, the development versioning for an application is decided from the beginning, choosing the most suitable architecture, after studying all the requirements and the context, keeping in mind possible future extensions.

In our case the requirements will be added incrementally, trying as much as possible to preserve the existing architecture and to always make the minimal impact change, treating each version as being the last one. Practically, while creating this use case I will put on the hat of a bad application architect or designer, which has no vision for the future of the application.

2.2 Stages

2.2.1 Stage I

Requirements: The application is single user and single instance and it offers only data visualization services. No data is modified or inserted by the application. The data transferred between the application and database is made through a secured environment.

Consequences: An architecture chosen based on the above requirements and excluding any future development, can be only 2-Tier. The logical or physical separation of functionalities has no justification in this context. Figure 1 presents the architecture of the application at this stage.

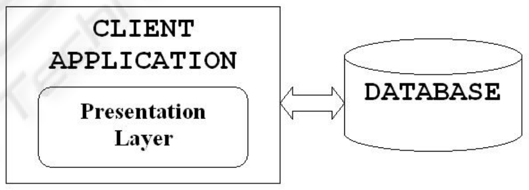

Figure 1: First Stage's Architecture.

2.2.2 Stage II

Requirements: The transfer of data between the application and the database is done through an open environment and therefore it is required to verify if the information was transmitted correctly.

Consequences: The physical architecture of the application stays the same like at Stage I, 2-Tier.

Additionally, the application needs to control the integrity of the information received from the database.

Checking the integrity of the transferred data is affecting both the structure of the database and the application's architecture. The introduction of these additional services will modify the logical architecture of the application, inducing a new layer that will handle the data verifications. The application will now have two functional levels: Presentation and Verification, as presented in Figure 2.

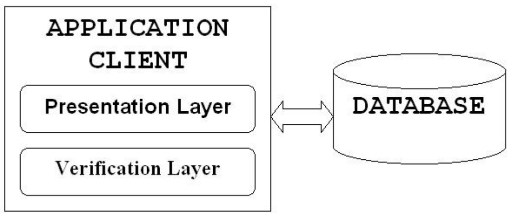

Figure 2: Second Stage's Architecture.

2.2.3 Stage III

Requirements: The application should maintain the confidentiality of the transferred data by protecting it from unauthorized accesses.

Consequences: Digitally signing the data is not enough to insure that unauthorized third parties do not access it. Since the communication is done through and open unsecured environment, it is necessary to crypt the information in order to insure its inviolability. The confidentiality requirement can be implemented either by using the native support of the database, if existent, or by implementing an encryption algorithm.

The algorithm and the encryption key are chosen depending on the required security degree. Either symmetric algorithms, like DES, or asymmetric algorithms, like PGP, can be used. Also, most of the commercial DBMS's provide native support for secured communication protocols like SSL.

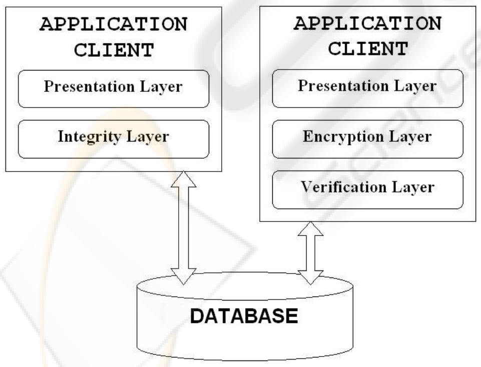

Figure 3: Stage III–Candidate Architectures.

Similarly with the digital signature, the encryption of the transferred date adds a computational overhead. The physical architecture of the application remains unchanged, but the logical architecture of the application can change in order to accommodate the encryption mechanisms in a separate Encryption layer (Figure 3, right). The encryption mechanisms can also be clubbed together with the data verification mechanisms, forming an integrity layer (Figure 3, left).

2.2.4 Stage IV

Requirements: The application can be simultaneously used from different computers (multi-user).

Consequences: Keeping in mind that the application only reads data from the database, the physical architecture can stay 2-Tier. Also, since all clients read data without modifying it, an exclusion mechanism is not required. Such a requirement will not at all influence the application's architecture.

However, the same requirement added after the stage in which the application already starts to modify data will for sure influence the architecture. This also proves that not only the integrity mechanisms and the architecture are correlated having the requirements as a catalyst, but also the order in which requirements are added to the system is relevant.

2.2.5 Stage V

Requirements: Additionally, the application will allow its users to modify the record stored in the database and to add new records.

Consequences: In this stage it is still not required to take care of the operational integrity of data since the effects of each and every operation are available immediately after being done.

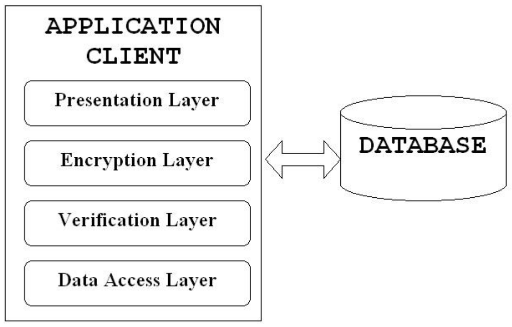

Figure 4: Fifth Stage's Architecture.

The physical architecture of the application remains unchanged, while the logical architecture is affected by the introduction of a new layer dedicated to the data access.

The separation of all data access related functionalities into a separate layer is the best solution that can be applied, the application's architecture becoming the one from Figure 4 that contains the following layers: Presentation, Verification, Encryption, and Data Access.

2.2.6 Stage VI

Requirements: The application needs to execute work units or groups of operations whose result to be available only if the entire group was processed successfully. Such an operation group should not overwrite during its execution data that was already modified by another.

Consequences: This stage is the first in which it becomes necessary to preserve the operational integrity of data. The application's work units composed of multiple operations defined in the requirement are actually translated into database transactions. The clients will therefore require the ability to impose locks on the records that they read with the purpose of modifying so that no other instance will attempt to simultaneously do it. The transaction execution can be done either by using the native database support or by embedding the transaction mechanisms in the application. Also, there is a third option of implementing the transactions as database stored procedures.

In case the transactions are processed using the native database support or implemented as stored procedures the 2-Tier physical architecture of the application remains unchanged. However, if the transactional support is implemented directly in the application, its physical architecture will modify, becoming 3-Tier. The additional tier will be dedicated to transaction management.

Practically, the choice is about the best transactional model for the application, the available options being: TP-Less, TP-Lite and TP-Heavy.

As per the initial assumption that at each stage the lowest impact solution will be chosen, the native transactional support of the application will be used, going for a TP-Less transaction processing strategy.

2.2.7 Stage VII

Requirements: The application needs to cope with strict performance criteria, related both to the processing speed and the bandwidth required for data transfer from and to the database.

Consequences: Such non-functional requirements will trigger modifications in the way transactions are processed.

There are situations in which the use of the native transactional support of the database does not fulfill the required performance criteria, especially because of the big amount of data transferred to and from the client in order to be processed. The data set resulted after each operation will be sent to the client who will take the decision of continuing the transaction or not.

The transaction's implementation as stored procedures will diminish the quantity of data transferred through the physical communication environment. I/O operations are probably the bottleneck of any software systems, and they are even slower if network transfer is also involved.

2.2.8 Stage VIII

Requirements: The application needs to support a number of clients bigger than the maximum number of connections accepted by the database.

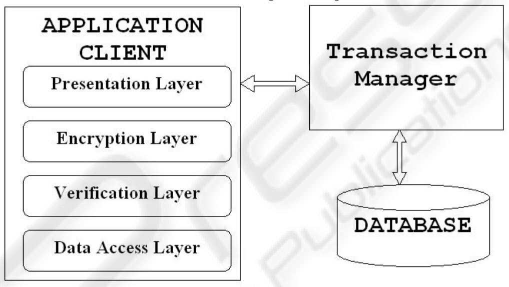

Figure 5: Eighth's Stage Architecture.

Consequences: The adoption of the above stated requirement involves changes in the physical architecture of the application. The transaction processing strategy needs to change, going to TP-Heavy and consequently adding a new tier. The introduction of the additional tier consisting of a transaction manager, one of the simplest types of application servers, will change the physical architecture to 3-Tier (Figure 5).

A transaction manager is doing a multiplex operation between n clients and m connections, where usually n is greater than m using a waiting queue.

3 MATHEMATICAL MODEL

A software system like the one used for the above case study mathematically modeled. Apart from modeling the state of the system as any point of time, its evolution can also be modeled.

Architecturally, a system evolves only when exposed to stimuli or events, which in such a case are new or modified requirements. Depending on the new or modified requirements, the system will change or not the architectural state, both physically and/or logically. Therefore, the evolution of a software system during successive development cycles can be modeled as a discrete set of states X, where $x \in X$ is any discrete state of the system.

Let Γ be a discrete set of requirements that can be implemented in the studied software system and $\alpha \in \Gamma$ a requirement that will be implemented in the system at a certain time, then $\Gamma(x)$ is the set of requirements, both functional and non-functional, that can be implemented in the system at a certain moment.

It is obvious that not all the potential requirements can be implemented at any moment, mainly because of dependencies between them. Also, some of the requirements can become impossible to implement at certain moments because of the way the system evolved, reaching some technical limitations.

The modeling at the system can be started at any moment, the initial modeling state of the software system, x_0, being chosen accordingly.

Considering the implementation of a requirement from the set of possible ones in a certain state of the system $\Gamma(x)$ to be equivalent with the application of a stimulus to any kind of general system, the software system will make a state transition depending both on the current state and the applied stimulus:

$$x_n = f\left(x_p, \theta\right),$$

where $x_n, x_p \in \Gamma; \theta \in \Gamma$

For any state x_k of a system, a requirement $\alpha \in \Gamma(x_k)$ activated in x_k has a lifetime $c_\alpha(x_k)$. The lifetime $c_\alpha(x_k)$ represents the effort required to implement the requirement, keeping in mind the state of the system.

To simplify the model, the requirements are considered to be implemented one after the other, a new requirement being introduced only if the system is not inside a development cycle $c_\alpha(x_k)$.

Every state x_k of the system can potentially have a requirement or a sequence of requirements that lead to o a state transition. Let's call such requirements' sequences transition triggers because they trigger a state transition, and use D as notation for them. The sum of the lifetimes of all the requirements from a trigger is the lifetime of the transition trigger:

$$c_D(x_k) = \sum_{\delta \in D} c_\delta(x_k).$$

A trigger is an ordered set. The order of the requirements inside the set D is strict. A change in the order of the requirements can provoke the state transition to happen earlier or not happen at all.

Starting from the points stated above, the minimum lifetime of between two architectural states of a system can be defined as being:

$$c_D^*(x_k, x_j) = \begin{cases} \infty, \textit{ if } \Delta_D(x_k, x_j) = \varnothing \\ \min\left(c_{\Delta_D(x_k, x_j)}\right), \textit{ otherwise} \end{cases}$$

Where $\Delta_D(x_k, x_j)$ is the set of transition triggers that will trigger the state transition from state x_k to state x_j.

Furthermore, based on the minimum lifetime between a state of a system and any other state, the minimum lifetime in a state can be defined as being the minimum of all the minimum lifetimes between that state and any other state of the system, excepting the previous states:

$$\mathrm{K}^*(x) = \min\left(c_D^*(x\ , x_j)\right),$$

Where $x_j \in X$.

The minimum lifetime of a system in a state defines the architectural quality of the system in that state. As the value of the minimum lifetime of the system in a certain state is bigger, the system is more robust, flexible and more resistant to new requirements' implementation.

Therefore the architectural flexibility of a system can be defined in a given context as being proportional with the minimum lifetime of the system's state that corresponds to the evaluated architecture. An ideal architecture is one that is able to cope with additional requirements without changing. It corresponds to an infinite value of the minimum lifetime of a state.

The lifetime of a requirement, which actually mean the effort required for its implementation, can be used as comparison criteria between two systems with similar functionalities but having different architectures. The impact of the same requirement can be different when applied as a stimulus to different software systems. For example, in one of the systems the requirement can trigger a state transition while in other systems the state will remain unchanged.

4 CONCLUSIONS

The versioned development of an application is used as a case study, adding a new requirement at each stage and always tailoring the solution as if no further development will be done, treating each stage as the last one.

The case study distinguishes a strong relation between architecture and integrity, by analyzing the organizational influence and the impact of the application's requirements on the integrity mechanisms.

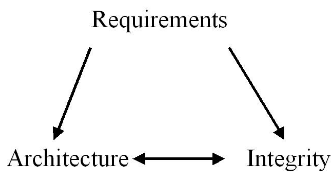

Figure 6: Relations between Architecture, Integrity and Requirements.

The main conclusion that can be drawn from the case study presented here is the existence of a dependency triplet, pictured in Figure 6: Requirements – Architecture – Integrity. The architecture of the software system and the way data integrity is realized and controlled are tightly coupled together, the set of requirements to be implemented in the system being the catalyst of this relation.

Adding new requirements at each phase at the case study triggered architectural changes. The logical structure of the application was the most affected by the additional requirements introduced at every step. The logical layering of the application doesn't serves directly the requirements additionally implemented in a software system, but gives more clarity to the system's architecture, makes it more scalable and introduces a clear separation between concerns at system level.

The architectural evolution of an application is studied, stimulated by the introduction of new requirements at each stage, distinguishing the bijective relation between architecture and the integrity assurance mechanisms that are used. The relation is injective controlled by the stimuli, in our case the requirements. The integrity mechanisms and the multi-level data access architectures are correlated. Integrity assurance proves to be an integrant part of any architecture.

REFERENCES

Octavian Paul ROTARU, "Database Integrity Assurance Mechanisms in Multi-Level Architectures. Pattern and Components in Databases", *PhD Dissertation, scientific adviser Prof. Dr. Eng. Mircea Petrescu*, University "Politehnica" Bucharest, 2005.

Mehdi Jazayeri, "OnArchitectural Stability and Evolution", Ada-Europe 2002, Vienna, Austria.

D. L. Parnas, "Designing Software for ease of Extension and Contraction", IEEE Transactions on Software Engineering, 5(2), pp. 128-138, March 1979.

D. L. Parnas, "Software aging", Proc. of International Conference on Software ENgineering (ICSE94), Sorento, May 1994, pp. 279-287.

M. Shaw, D. Garlan, "Software architectures: perspectives on an emerging disciplines", Prentice Hall, Englewood Cliffs, NJ, 1996.

Rami Bahsson, Wolfgang Emmerich, "Evaluating Softaware Architectures: Development, Stability, and Evolution", the Proceedings of ACS/IEEE International Conference on Computer Systems and Applications, Tunis, Tunisia, July 14-18-2003 IEEE Press.

Muhamad Ali Babar, Ian Gorton, "Comparison of Scenario-Based Software Architecture Evaluation Methods", 11th Asia-Pacific Software Engineering Conference (ASPEC'04), pp. 600-607.

Gasiorowicz, Stephen, 2003. Quatum Physics, 3rd edition. John Wiley & Sons Inc.

Cohen-Tannoudji, Claude, 1989. Photons and Atoms: Introduction to Quantum Electrodynamics. John Wiley & Sons Inc.