On Architectural Stability and Evolution

Mehdi Jazayeri
Technische Universität Wien
jazayeri@tuwien.ac.at

Abstract

Many organizations are now pursuing software architecture as a way to control their software development and evolution costs and challenges. A software architecture describes a system's structure and global properties and thus determines not only how the system should be constructed but also guides its evolution. An important challenge is to be able to evaluate the "goodness" of a proposed architecture. I suggest stability or resilience as a primary criterion for evaluating an architecture. The stability of an architecture is a measure of how well it accommodates the evolution of the system without requiring changes to the architecture. As opposed to traditional predictive approaches to architecture evaluation, I suggest retrospective analysis for evaluating architectural stability by examining the amount of change applied in successive releases of a software product. I review the results of a case study of twenty releases of a telecommunication software system containing a few million lines of code to show how retrospective analysis may be performed.

Keywords: software architecture, software evolution, architecture evaluation, architectural stability, retrospective analysis, software visualization

1. Introduction

After many years of research on software architecture, interest in software architecture seemed to suddenly explode in the latter half of the 1990s. Both industry and academic interest in software architecture has been intense, as witnessed by the many books, conferences and workshops on software architecture. Some of the recent milestone publications are the paper by Perry and Wolf [Perry92] that proposed a systematic framework for the study of software architecture, Shaw and Garlan [Shaw96] that tried to present software architecture as a systematic discipline, Kruchten[Kruchten96] that proposed a model of software architecture that enlarged the view of software architecture from dealing only with static structure to consisting of four different aspects.

Prior to these works, Parnas had laid the foundations of software architecture in a number of seminal papers. The problems he addressed in these papers in the 1970s and 1980s took two decades to be recognized as real problems by the software industry. Parnas not only identified the key problems, he also developed a set of related architectural concepts for addressing these problems. These concepts are still fundamental for understanding and solving the problems of software construction. I

will review some of these concepts here because they form the underpinnings of any work on software architecture, including this one.

The most fundamental of these ideas was the principle of "information hiding" [Parnas72]. This principle gives the designer a concrete method for decomposing a design into modules. The modularization is on the basis of design decisions. Each module protects, or encapsulates, or "hides" an important design decision. The motivation behind the method is that if important design decisions are encapsulated in separate modules, then changing a design decision in the future will affect only the associated module and not require changes that are scattered throughout the software. The same motivation is behind the development of object-oriented development, but information hiding is a more fundamental and basic notion.

The problem of *change* is a key challenge to software engineering. The ideas of "design for change" and "anticipation of changes" [Parnas79] were the motivations for Parnas's work and information hiding was a concrete design method to implement them. Parnas also proposed a set of relations and structures as a basis for designing and documenting a system's architecture. The "uses" relation [Parnas74] describes a relationship among modules of a system. A module M1 uses module M2 if a working copy of M2 must be present for module M1 to satisfy its specification. This fundamental relationship supports the design of software structures that are organized as a hierarchy of modules. One important implication of such a design is that it makes it possible to define working subsets of the hierarchy. This means that we can not only build the software incrementally, but also that we can build potentially useful subsets of the software.

The subsettable software designs and information hiding lead almost naturally to the idea of a family of designs. In [Parnas76], Parnas describes the reasons for designing families of programs rather than single programs and gives a concrete approach for doing it. The idea is to clearly identify the design decisions that are common to family members and those that distinguish among family members. Naturally, design decisions can be hidden by appropriate modules and we can build the right family member by using the module that captures the right decision. The idea of program families was inspired by the IBM System 360 architecture, which indeed billed itself as a family architecture. The architecture was shared by all the members of the family. Particular "realizations" of the architecture exhibited different properties, primarily in terms of performance. Today, the concept of program families is pursued under the topic of software product families or product-line architectures.

2. Architecture and Evolution

Despite the concern with "change" and accommodating changes, most of the early definitions of software engineering focus explicitly on construction and only implicitly, if at all, with the phenomenon of software "evolution." By and large, software processes and design techniques concentrate on construction. Yet we know from experience that evolution is a key problem in software engineering and exacts huge costs. Anecdotal evidence even hints that companies spend more resources on maintenance (i.e. evolving their software) than on initial development. Probably the

earliest work to deal explicitly with software evolution is that of Lehman[Lehman80, Lehman85]. Even Parnas finally got to deal explicitly with the issue of evolution in his paper on software aging [Parnas94], where he posits a set of hypotheses and insights on why software needs to evolve and how we can deal with the challenges. Recently, Bennett and Rajlich[Bennet00] proposed a software development process that considers the evolution and retirement of software as explicit phases in the software lifecycle.

We can argue that, as Parnas foresaw in his early work on "change," evolution is the underlying, if implicit, motivation for much of the recent software development research. For example, product-line architectures are a systematic approach to controlling software evolution. They try to anticipate the major evolutionary milestones in the development of the product, capture the properties that remain constant through the evolution and document the variability points from which different family members may be created. The approach gives a structure to the product's evolution and possibly rules out some unplanned evolutions, if the architecture is respected. Incremental software processes, such as the unified process, are also ways to structure the software's evolution through prescribed steps. The assumption is that evolution is helped by the feedback gained from releases of the early increments.

3. The Role of Software Architecture

There are many definitions for software architecture. Definitions usually concentrate on structural properties and the kinds of decisions that an architect must make. A common view is that architectural decisions are those that have to be made before concurrent work on the system can be started. That is, the architectural decisions span the system components and determine their overall relationships and constraints. Once these decisions have been made, work on the individual components may proceed relatively independently. The architectural decisions determine many of the significant properties of the system and are difficult to change because they span the whole system. Therefore, one of the major implications of a software architecture is to render particular kinds of changes easy or difficult, thus constraining the software's evolution possibilities. Despite the importance of evolution, and the impact of the software architecture on evolution, it is surprising that most definitions of software architecture do not explicitly mention evolution. In fact, it can be argued that the primary goal of a software architecture is to guide the evolution of the system, the construction of the first release of the system being only the first of many milestones in this evolution.

4. Architecture Evaluation and Assessment

Because of the key role that architecture plays in the life of a system, it is important to evaluate or assess a system's architecture. Reviews and inspections are accepted evaluation methods in software engineering. In such an evaluation, we have to decide what factors we are evaluating and what the goals of the evaluation are. Depending on the definition of software architecture and its goals, it is possible to define an

evaluation procedure. Typically, such an evaluation is qualitative and is itself difficult to evaluate.

A representative architecture evaluation method is the Architecture Tradeoff Analysis Method (ATAM), developed at the Software Engineering Institute [Kazman99]. ATAM tries to help elicit the goals of the architecture and then evaluates the architecture against those goals. The procedure is said to take 3 or 4 calendar days. It is aimed at evaluating how well the architecture meets its quality goals such as performance, reliability, security, and modifiability.

We call such kinds of evaluations *predictive*. They try to anticipate how well an architecture will perform in the future. While useful, predictive evaluations suffer from inherent uncertainty. How do we know what to assess? Even if we did know, how do we assess it? How sure can we be of the results?

A way to answer these questions is to apply *retrospective* evaluation. I will describe retrospective evaluation here in the context of evolution. First, we start with the assumption that the software architecture's primary goal is to guide the system's evolution. Retrospective evaluation looks at successive releases of the product to analyze how smoothly the evolution took place. Intuitively, we want to see if the system's architectural decisions remained intact throughout the evolution of the system, that is, through successive releases of the software. We call this intuitive idea "architectural stability."

There are many ways we can perform such an analysis but all rely on comparing properties from one release of the software to the next. This implies that some architectural information must be kept for each release. For example, we might compare the uses relation in successive releases. If the relation remains substantially unchanged, we can conclude that it was a stable (or robust) architecture that supported evolution well. There are virtually any number of quantitative measures we can make depending on what aspect of the architecture we are interested in evaluating.

Retrospective analysis can have many uses. First, we can empirically evaluate the software architecture's stability. Second, we can calibrate our predictive evaluation results. Third, we can use the results of the analysis to predict trends in the system's evolution. Such predictions can be invaluable for planning the future development of the system. For example, a manager may use previous evolution data of the system to anticipate the resources needed for the next release of the system, or to identify the components most likely to require attention, or to identify the components needing restructuring or replacement, or, finally, to decide if it is time to retire the system entirely. In the next section, we describe a case study that shows some simple examples of retrospective analyses.

5. Case Study

We have applied three different kinds of retrospective analyses to twenty releases of a large telecommunication software system. In the first, we compared simple measures such as module size, number of modules changed, and the number of modules added in the different releases. In the second, we tried to detect coupling among modules by

discovering which modules tend to change in the same releases, and in the third, we used color visualization to "map out" the system's evolution. In this section, we give an overview of these experiments. The details may be found in [Gall97, Gall98, Gall99]. The telecommunication system under study consists of over ten million lines of code. The system is organized into *subsystems*, each subsystem consists of *modules*, and each modules consists of *programs*. We had access to a database that contained information about the system but not the code itself. The size of components is recorded as the number of subcomponents it contains. For example, the size of a module is the number of programs it contains.

5.1 The first analysis: Simple metrics

In the first set of analyses, we simply plotted various basic size-related metrics according to releases. For example, Fig. 1 shows the growth in the number of programs in the system. It shows a steady but stabilizing growth of the system. It appears to show a stable system, possibly approaching a dormant state, getting ripe for retirement.
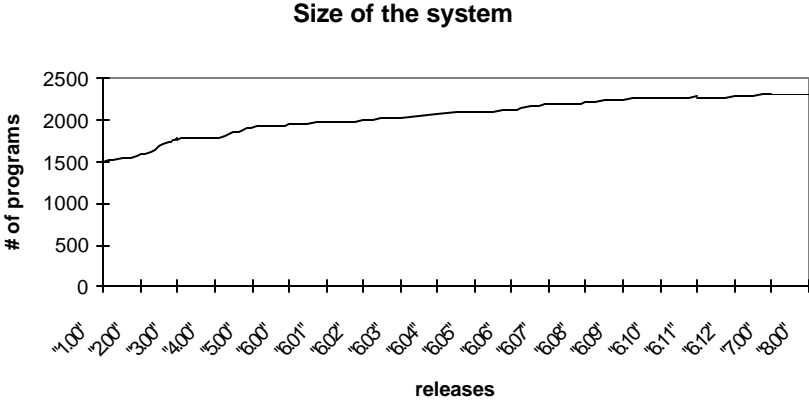
Size of the system

Fig. 1 Growth of the system in size

Fig. 2 shows the number of *added* programs. Here we see that a decreasing number of programs are added at each release, with a curiously higher number in every other release. This phenomenon could be due to the way the releases were planned. The manager of the project should be able to interpret the result and decide whether it was expected. In any case, the fact that the number of additions is decreasing also points to a stabilization of the system.

Fig. 3 plots two related numbers: the percentage of programs added and the percentage of programs changed in each release. The figure seems to indicate that in one release "many" programs are added and in the next "many" are changed. We don't know if there is any correlation between the programs added and those changed

in the next release. But the figure certainly invites many questions that should be of interest to the manager.

Finally, in Fig. 4 we show the growth in size of three different modules. We see that two of the modules are relativ ely stable while the third is growing significantly. This figure indicates that it is not enough to study the data only at the system level. It is possible that undesirable phenomena at a lower level, in this case at the module level, mask each other out at the system level. Certainly the growth of Module A compared to the other modules should ring an alarm bell to the manager.

These figures show how simple metrics plotted along the releases of a system can reveal interesting phenomena about the evolution of the system. Unusual and anomalous evolutions of components can be easily spotted. Any deviations from expectations should be investigated.
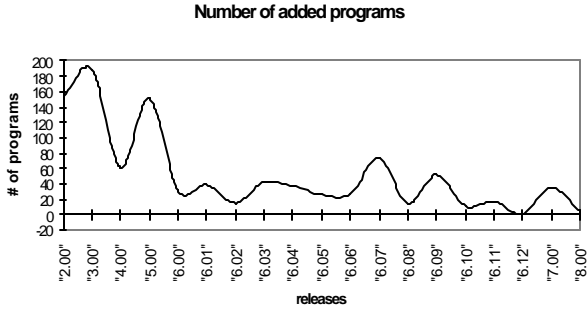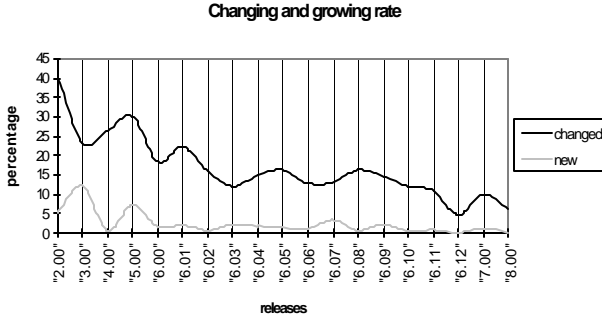
Number of added programs

Fig. 2. No. of added programs per release

Changing and growing rate

Fig. 3. No. of changed and added programs per release
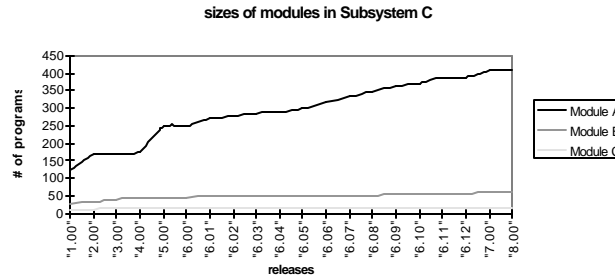
sizes of modules in Subsystem C

Fig. 4. Growth of size of modules in one subsystem

5.2 The second analysis: Hidden module coupling

In the second experiment, we tried to uncover potential (hidden) dependencies among modules or programs of the system. The idea was to discover if there are certain modules that always change during the same release. For example, Table 1 shows two particular programs that are changed in nine releases together. In the two other releases, one is changed but not the other. We developed a number of analysis techniques for discovering and correlating "change sequences". If two modules are always changed in the same sequences of releases, it is likely that they share some possibly hidden dependencies. The longer the sequence, the higher is the likelihood of cross-dependencies. Such analysis can be easily performed and can reveal a great deal about the architecture. In fact, the goal of the architecture is to minimize such dependencies so that change and evolution is isolated in different modules. If changes are required in many modules, the architecture suffers from lack of stability.

$SUB_2 = <1\ 2\ 3\ 4\ 6\ 7\ 9\ 10\ 14>$										
A.aa.111	1	2	3	4	6	7	9	10	14	17	19
B.ba.222	1	2	3	4	6	7	9	10	14	16	18

Table 1. Coupling among subsystems A and B

5.3 The third analysis: Color visualization

In this study our goal was to make the results of retrospective study more apparent and easy to grasp. We used visualization techniques to summarize the large amount of data that could be plotted and displayed. In particular, we explored the use of color in such visualizations. Due to the need for color, the reader is urged to look at an on-line version of this paper to view the figures in color. We use color percentage bars to display a history of a release. For example, Fig. 5 represents a module by a bar in each release. The bar contains different colors. The colors represent different version numbers of programs in the module. For example, in the first release, when all programs are at version 1, the bar is a single color. By comparing the bars for different

releases, the eye can quickly observe the amount of changes from one release to the next. Large variations in color indicate a release that is undergoing lots of change, possibly indicating an unstable architecture. Fig. 6 shows the change maps for modules A through H of the system. Such maps can be used to quickly identify problematic modules. The color maps for different modules may be quickly compared to get a sense of how module evolutions relate to each other. Such maps could be used as a "fingerprint" of a module to show its evolution. It is possible to spot different types of evolution and modules that share certain patterns of evolution. A predictive evaluation of the architecture, if effective, should be able to anticipate the kind of fingerprint a module should produce during its evolution.
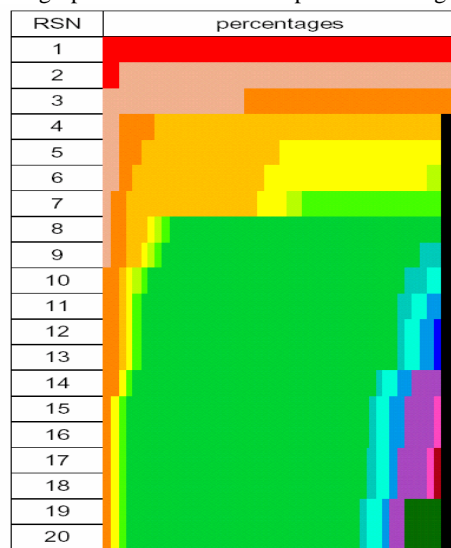

Fig. 5. Visualizing evolution with percentage bars

6. Retrospective analysis

The case studies of the previous section show a glimpse of how retrospective analysis may be applied and exploited. The information from retrospective analysis may be used for forward engineering as well, primarily by using the information from the past to predict the requirements of the future, for example, in answering questions such as how much code changes the next release will entail and how much it will cost. The tools we have used are simple and require very little information to be kept for each release of the software. Yet, such data is not commonly maintained, analyzed, or exploited. The key point is that the tools must maintain data across releases to enable the reasoning and analysis about the software's evolution. This means that software reengineering tools must be enhanced to deal with releases explicitly to be able to

support retrospective evolution analysis. Because of the huge amount of data involved, visualization techniques seem to be useful.

An example tool that can be used for evolution analysis is the Evolution Matrix [Lanza01] which visualizes the evolution of classes in an object-oriented system. The evolution of various metrics about a class may be displayed. Size and color are used to represent the metrics. The evolution analysis applied to a large number of classes has led to classifying different types of classes based on their evolution patterns. Lanza [Lanza01] has observed classes that he categorizes as supernova (suddenly explodes in size), pulsar (grows and shrinks repeatedly), white dwarf (shrinks in size), red giant (continues being very large), and idle (does not change). Such a tool can be a powerful aid in retrospective analysis. For example, a large number of idle classes would indicate a stable architecture. (Clearly, idle classes could also indicate dead code so analysis had to be done carefully.)

7. Summary and conclusions

We have argued that a primary goal of a software architecture is to guide the evolution of a software product. To evaluate how well a software architecture achieves this goal, we can analyze the architecture for adherence to certain rules that we believe support evolution. But there is more that we can do. Using appropriate analysis tools, we can try to evaluate an architecture's "stability" or "resilience" by observing the actual evolution of the associated software product. We call this kind of analysis "retrospective" because it looks back on the software product's releases. We have shown the results of some simple tools that can help in retrospective analysis. These tools combine simple metrics and visualization to summarize in a compact form the evolution patterns of a system, thus enabling the engineer or manager to check the reality against the expected results.

In principle, predictive analysis and retrospective analysis should be combined. Perfect predictive evaluations would render retrospective analysis unnecessary. If we are not sure of perfection, however, retrospective analysis is necessary to validate our predictions and detect deviations from plans.
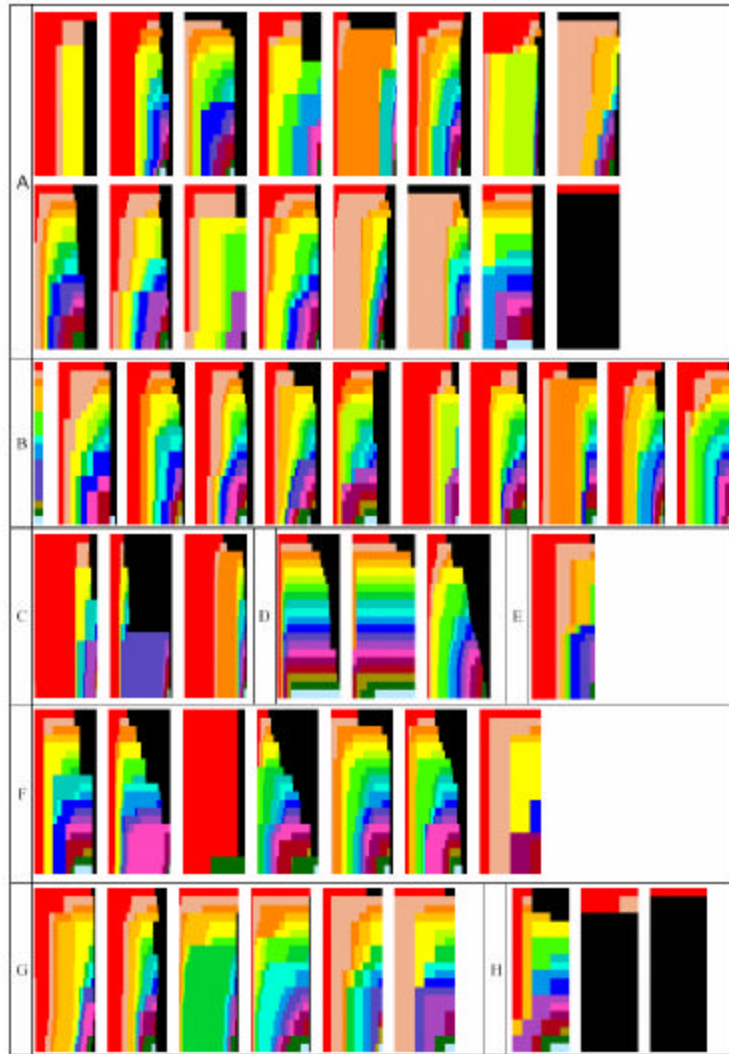
Fig. 6. Evolution of modules A through H in terms of programs

8. References

[Bennett00]

        K. Bennett and V. Rajlich, "A staged model for the software lifecycle," *Computer* 33(7): 66–71, July 2000.

[Gall97]

H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, "Software Evolution Observations Based on Product Release History," Proceedings of International Conference on Software Maintenance (ICSM '97), Bari, Italy, IEEE Computer Society Press, Los Alamitos, CA, September 1997.

[Gall98]

H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release histories," Proceedings of International Conference on Software Maintenance (ICSM '98), Washington, DC, IEEE Computer Society Press, Los Alamitos, CA, November 1998.

[Gall99]

H. Gall, M. Jazayeri, and C. Riva, "Visualizing software release histories: the use of color and third dimension," Proceedings of International Conference on Software Maintenance (ICSM '99), pp. 99–108, Oxford, UK, IEEE Computer Society Press, Los Alamitos, CA, September 1999.

[Jazayeri00]

M. Jazayeri, A. Ran, and F. van der Linden, Software Architecture for Product Families: Principles and Practice, Addison-Wesley, Reading, 2000.

[Kazman99]

R. Kazman, M. Klein, and P. Clements, "Evaluating software architectures for real-time systems."

[Kruchten95]

P. B. Kruchten, "The 4+1 view model of architecture," *IEEE* Software, 29(11): 42–50, November 1995.

[Lanza01]

M. Lanza, "The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques," Proceedings of IWPSE (International Workshop on Principles of Software Evolution), 2001, Vienna.

[Lehman80]

Lehman M.M., "Programs, life cycles and laws of software evolution," *Proceedings of the IEEE*, pp. 1060-1076, September 1980.

[Lehman85]

Lehman M.M. and Belady L. A., *Program evolution*, Academic Press, London and New York, 1985.

[Parnas72]

D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Communications of the ACM, 15(12): 1053–8, December 1972.

[Parnas74]

D. L. Parnas, "On a buzzword: hierarchical structure," Proceedings IFIP Congress (1974), North-Holland, Amsterdam, 1974.

[Parnas76]

D. L. Parnas, "On the design and development of program families," IEEE Transactions on Software Engineering, 2(2): 1–9, March 1976.

[Parnas79]

D. L. Parnas, "Designing software for ease of extension and contraction," IEEE Transactions on Software Engineering, 5(2):128–138, March 1979.

[Parnas94]

     D. L. Parnas, "Software aging," Proc. International Conference on Software Engineering (ICSE 94), Sorrento, May 1994, pp. 279-287.

[Perry92]

     D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," ACM SIGSOFT Software Engineering Notes, 17(4): 40–52, October 1992.

 [Shaw96]

     M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.