Volume 3, Issue 7, July 2013 ISSN: 2277 128X
**International Journal of Advanced Research in Computer Science and Software Engineering**
Research Paper
Available online at: www.ijarcsse.com

# Analysis of Software Design Erosion Issues

**Er. H. P. S. Dhami**
Associate Professor, Department of CSE
RBIENT, Hoshiyarpur, India

**Dr. Anuj Kumar**
Professor, Department of CSE
AIMT,Noida, India

***Abstract-Design erosion in software products is a common problem and most of the software systems are affected by it. We have found that invariably, no matter how ambitious the intentions of the designers were, with the passage of time as these systems age, it becomes ever more difficult to make changes. Eventually it is more feasible to redesign and replace or at least refactor the software than it is to continue with the regular maintenance to extend the life of the existing design. In this paper we have identified a number of potential causes for this phenomenon. The analysis & study of some commercial software, various erosion problems & issues, development approaches, and observations related to design erosion problems.***

## I. Introduction

Most of the software systems start with precise & understandable software architecture. But then, developers are often forced to modify the system under increasing time pressure. With the ever increasing size and complexity of software, the weaknesses of existing software development methods and tools are beginning to show. This becomes evident when it comes to maintaining the system. The strategic core architecture becomes vague or even completely lost. Design erosion cause breakdown of the original software architecture. The software crisis was identified during a NATO workshop [Naur & Randell 1969]. Since then, many approaches [J.Gurp & J. Bosch 2002], have been suggested to solving the software crisis, many of which are still applied today. In this paper we intend to illustrate that despite more than forty years of research and many suggested approaches, it is still inevitable that a software system eventually erodes under pressure of the ever changing requirements. Some examples of approaches are the architecture development method discussed in [Bosch 2000], the software development method Extreme Programming [Beck 1999] and many others. However, we have reasons to belief that such approaches still do not fully address the identified issues.

Design erosion is a problem that affects mostly all large software systems. The phenomenon is also known as architectural drift [Parnas 1994], software aging [Perry & Wolf 1992] or architecture erosion [Jaktman, Leaney, Liu 1999.]. Essentially the problem is that as software evolves, the software is incrementally changed to meet new requirements, fix defects or optimize quality attributes (adaptive, corrective and perfective maintenance [Swanson 1976]). However, these requirements may conflict with requirements in earlier iterations or may change the assumptions under which design decisions in earlier iterations were made. When faced with such requirement conflicts, there are two strategies for adapting the system to incorporate the changes: the optimal architecture strategy and the minimal effort strategy The example discussed in this paper serves both as an illustration of design erosion and related problems and as a starting point for future research. Furthermore, we present two strategies for incorporating change requests, identification of design erosion problems & related issues and finally, discussing & analyzing them.

### A. *Examples Studied - Commercial Software*

I.T industry is increasingly relying on a growing quantity of ever-larger software, and design erosion presents a serious problem. Design erosion is quite common and the diagnosis of its occurrence is often used as a motivation for redeveloping systems from scratch. Affected software cannot be easily replaced or repaired. Failing to do so, however, may cause maintenance cost to rise and limits the flexibility of the affected software. In some circumstances, requirement changes caused by new business models, technologies or desired features might be too far-reaching. In these cases, it is sometimes much more effective to throw away the old system and build a new one from scratch. In most cases such redevelopment requires a massive effort. Ultimately, eroded software may threaten the existence of the company that produces it as well as the existence of companies that use the software.

A well known example of commercial project, where this happened is the Mozilla web browser. Around ten years ago, Netscape was experiencing fierce competition from Microsoft's Internet Explorer. They decided to release their own browser as open source and started working on transforming it into the next generation browser. After a year of development the developers of the open source Netscape came to the conclusion that the original Netscape source was eroded beyond repair and they started from scratch. After two years of development an enormous amount of code was released and some of it was retired yet again (despite it was written from scratch). Apparently during redevelopment, requirements had changed sufficiently to retire a part of the system before the system was even finished. More than ten years later the Mozilla project is still working on this browser and updating it as per the new requirements. Another example is the version of the Linux kernel, this product was developed as an open source project. It took approximately one year to develop kernel 3.2(used in Ubuntu 12.04 LTS, Debian 7 Wheezy and Slackware 14.0) after the previous stable release (version 3.0). The reason is that much of the old 3.0 code needed massive restructuring in order to incorporate the new requirements. By redesigning large parts of the old kernel, the performance was enhanced and new requirements could be met. A similar effort can be expected for the next stable release (i.e. 3.9 kernel series and version 3.9.3). A third example of software erosion studied in [Bosch 1999a][Svahnberg & Bosch 1999]. A company (say xyz ltd.) that produces network devices, which replace PC's as a means to offer network connectivity for common PC peripherals like printers, scanners, DVD/CDs, etc. At first, this company only had a printer server; however, support for other devices was added over the time. At some point the developers realized that in order to support new types of devices, a radical restructuring of their software was needed. Rather than patching up the existing software it was decided to build a new architecture. After two years of development (while simultaneously maintaining the old software), they were ready to release products based on the new software. Now this new architecture (after a few years of successful use) was slowly being replaced by a third generation of software (they were migrating from their proprietary OS to an embedded Linux version). In the above examples, the redevelopment of the software can be considered a success. However, considering the effort needed to do so, it can easily be imagined that some companies are less fortunate in identifying the signs of design erosion early enough to be able to take such action. Redeveloping software (also referred to as the revolutionary approach), is a very expensive, lengthy and exhaustive procedure and failing to see it is essential, can be fatal to a software producing company.

Another issue that we have experienced in all the above cases is that, the redevelopment of the software was partly successful. The Linux development can be characterized as a continuous effort to perfect the system, often resulting in large parts being replaced by new code, Mozilla has already seen some of its components rewritten and the said company (xyz ltd.) is already working on its third generation of software.

## II. PROBLEMS AND ISSUES

On the basis of the Commercial Software cases that we have studied (e.g. [Bengtsson & Bosch 1998], [Bosch et al. 1999b] and [J.Gurp, J. Bosch 2002]) and the above examples, we have identified that design erosion is caused by a number of problems associated with the way software is commonly developed.

- **Inappropriate use of methods and tools**. Inappropriate use of process models, methods and tools and further poor selection testing methods can lead to design erosion. These factors are more significant.
- **Non-Traceability of design decisions.** It means that the notations generally used to create software, lack the precision and explicability needed to express concepts used during designing phase. Consequently, design decisions are difficult to track and reconstruct from the system.
- **Escalating maintenance cost**. The maintenance tasks become increasingly costly and effort consuming due to the fact that the complexity of the system keeps growing. This may cause developers to take sub-optimal design decisions either because they do not understand the architecture or because a more optimal decision would be too effort demanding.
- **Gathering of design decisions**. Design decisions accumulate and interact in such a way that whenever a decision needs to be revised, other design decisions are not reconsidered. In this case cyclic dependency problem may occur. A consequence of this problem is that if circumstances change, developers may have to work with a system that is no longer optimal for the requirements and that cannot be fixed cheaply.
- **Iterative methods**. The focus of the design phase is to create a design that can accommodate future requirements. This conflicts with the iterative nature of many development methods (rapid prototyping, extreme programming, etc.) since these methodologies typically incorporate new requirements that may have an architectural impact, during development whereas a proper design requires knowledge about these requirements in advance.
- **Architectural Drift**. Architectural drift refers to design decisions forgotten or unused by the development team, while Architectural erosion refers to changes that are in conflict with previous design decisions. The first version of all software is a relatively compact version. However, the design, maintainability and flexibility were less than ideal. The designer address these issues in the succeeding versions by changing the code structure; adding new classes; moving blocks of code; etc. During the evolution of new system or even at the time of refactoring, designers can easily violate design rules and constraints arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift [Perry, Wolf 1992] and its associated problems (e.g. increased maintenance costs). As design rules and constraints influence future design decisions, they have a steering influence on the future direction of the architecture. The design in most of the latest version still implements the same functionality as in first version. Thus, it is much larger and complex. The added structure provides some additional flexibility over the previous version. It also makes the new version harder to understand. This may lead to architectural drift.
- **Vaporization of Design Decisions**. The design erosion, complexity, and high costs of change, are some of the fundamental problems of software architecture. These problems are partially due to knowledge vaporization. Currently, almost all the knowledge and information regarding the design decisions on which the architecture is based on (e.g. results of domain analysis, architectural styles used, trade-offs made etc.) are implicitly embedded in the architecture, but lack a first class representation.

  The architectural design decisions are an important part of Architectural Knowledge (AK). Failure to document architectural design decisions can lead to AK vaporization and higher maintenance costs. We observed that in a larger system there will be even more of these design decisions vaporization.

On studying the problems various new facts were also observed that,

- One of the causes of design erosion might be that the actual developers were no longer working on the system (or had left the company). The employees take all their undocumented knowledge with them. New employees have to build up their knowledge alone. They don't have the possibility to increase their knowledge by taking part in architecture workshops, because for one or two person they can't be conducted. Thus, ignorance problems occur, which are linked to a lack of basic knowledge in the field. It is the knowledge related to facts and factual information, sometimes called "declarative" knowledge.
- An interesting aspect noted is that the development of the software was transferred a number of times to different groups. Some development groups are relatively inexperienced and consequently, the quality of their work was relatively low. Lack of experience in the field may lead to Misuse problems. Therefore, many of the design decisions taken early in the development of system was poorly documented and could not be understood methodically. Particularly the maintenance of system became more problematic after the person who designed that component left the company. The problem was surfaced once the development was transferred to a more experienced team.
- Some problems may occur due to new or changing technologies such as OS, Framework, Programming language, Hardware. Other aspects of problems could be Code clone, Dead code, and Metric outliers (Huge packages, Deep class hierarchies, Complex code).
- At the time of developing the system, changes may occur that require immediate updating and documentation of the software design. In some cases time-pressure had caused developers to bypass the proper process for defect fixing (which includes documenting the changes and designing a fix).
- During the evolution time of system, defects are found and fixed (in [8], called corrective maintenance). An appropriate way to fix a defect is to analyze the defect, design a solution, implement and test the solution. Unfortunately, time-pressure or cost considerations may prevent developers to properly follow this process. Often this results in quick fixes that addresses the issues but that may also introduce additional issues.

The consequences of erosion are that the systems are harder to maintain- readability, testability, side effects by the changes and the code is hard to test and understand. Thus, the systems maintenance cost increases and they become noneconomic. A noneconomic software system is a system where the maintenance costs are higher than a new implementation of the current system. This case of erosion is the worst that can happen to software.

## III. SOFTWARE DEVELOPMENT APPROACH - OPTIMAL VS. MINIMAL

Assuming an iterative development method, we can distinguish two conventional strategies for adapting the software system to incorporate the request for changes:

- **Optimal design strategy.** Include all the essential changes to the software artifacts to get an optimal system for the new set of requirements. Technically, no compromises are made with respect to design quality and incurred cost. The design of the software is improved in such a way that the new requirements can be incorporated without compromising the design integrity. The advantage of this approach is that the new changed system is optimal for the requirements because any conflicts with decisions in the previous version are resolved. This means that future changes can be incorporated at a relatively low cost. However, redesigning a system can take a lot of time and generally takes a lot of effort. Although this strategy typically results in a good design, the associated cost may make it infeasible for few changes.
- **Minimal effort strategy**. Including the change in the next cycle of the development while preserving of the old system as much as possible. Generally, complicated design changes can be avoided by stretching a bit the design rules of the existing design. Though, it may have consequences for the quality of the design, this strategy can be very effective in meeting the requirements on short notice. The advantage of this approach is the relatively low cost of each iteration.

Both strategies are unfeasible in general. In the optimal strategy the cost is too high, whereas the minimal strategy causes problems for future changes. However, we tend to look upon these strategies as two extremes in a spectrum of approaches.

We may conclude that it is inevitable that in real world systems the first strategy i.e. optimal design strategy is not always feasible. Consequently, cost considerations or time constraints sometimes compel developers to take less than ideal design decisions. Over the time, these less than ideal design decisions accumulate, resulting in what we call design erosion. Eroded software systems are typically hard to understand due to the many sub-optimal design solutions that have accumulated and complicated the design. Consequently, additional changes become harder and eventually may even become infeasible. When this happens, the only way to resolve the situation is to either repair (e.g. using refactoring techniques) or replaces the software. Both types of resolutions typically require a significant effort. We discussed number of real-world projects that were affected by design erosion. In these examples, the subsequent effort to repair/replace the software spanned several years.

## IV. STUDY OF RELATED WORK

In [Perry, Wolf 1992], a distinction is made between architecture erosion and architectural drift. Architectural erosion, according to Perry and Wolf, is the result of 'violations of the architecture'. Architectural drift, on the other hand is the result of 'insensitivity to the architecture' (the architecturally implied rules are not clear to the software engineers who work with it).

Parnas, in his paper on Software Aging [Parnas 1994], observes similar phenomena's. Although he does not explicitly talk about erosion, he does talk about aging of software as the result of bad design decisions which in turn are the result of poorly understood systems. In other words: erosion is caused by architectural drift. As a solution to the problem Parnas suggests that software engineers should design for change and should pay more attention to documentation and design review processes. He also claims that no coding should start before a proper design has been delivered.

In [Jaktman, Leaney, Liu 1999], a set of characteristics of architecture erosion is presented. Some of these characteristics are also identified in our study. In their case study, Jaktman et al. aimed to gain knowledge about how architecture quality can be assessed. Assessing architecture erosion is an integral part of his assessment.

To avoid taking bad design decisions, developers can consult a growing collection of patterns (e.g. [Gamma et al. 1995] and [Bushman 1996]). An approach to countering design erosion is refactoring [Fowler et al. 1999]. Refactoring is a process where existing source code is changed to improve the design. Fowler et al. present a set of refactoring techniques that can be applied to a working program. Using these techniques violations of the design can be resolved. Unfortunately, some of the refactoring techniques can be labor intensive, even with proper tool support (e.g. [Roberts, Brant, Johnson 1997]).

The notion of architectural design decisions is not part of the current perspective on software architectures. [Jansen & Bosch 2005] identified several problems due to this, including high costs of change, design erosion, and limited reuse, which are primarily caused by the vaporization of these design decisions into the architecture. To address these problems, proposed a new perspective on software architecture, where software architectures are described as set of design decisions. In many of the suggested approaches towards (e.g. Parnas' suggestions) solving the software crisis, it is assumed that if engineers work harder and/or more efficiently and/or use better tools, the problems will disappear. [Gurp, Bosch 2002] disagree with this assumption and demonstrate in his paper that design erosion is inevitable with the current way of developing software. Good methods only contribute by delaying the moment that a system needs to be retired. These approaches do not address the fundamental problems that cause design erosion. Rather than fight the symptoms we should start to address the causes. Gurp & Bosch evaluated an extensive example of evolutionary design to assess what happens to a system during evolution. The example clearly demonstrates how design erosion works. Design decisions taken early in the evolution of a system may conflict with requirements that need to be incorporated later in the evolution. In the example taken in their case study, they reversed several of such decisions. However, in large industrial systems such a thing is often infeasible due to the radical, system wide impact of such changes. Gurp & Bosch, in their analysis of design efforts, have found evidence of architectural drift, vaporized design decisions and design erosion. Causes identified for these problems ranged from the accumulation of multiple design decisions (i.e. certain design decisions were taken because of earlier design decisions, even if these were wrong decisions) to limitations of the OO paradigm. An optimal design strategy for the design phase does not deliver an optimal design.

## V. OBSERVATIONS AND ANALYSIS

The main goal of studying the various versions of commercial software was to observe and analyze what happens when a system is evolved as new requirements are added. By putting a strong emphasis on such requirements as flexibility, reusability and maintainability, the system began to show similar problems as those typically found in industrial cases. We will discuss various problems & will recommend few suggestions to lessen them.

- **Appropriateness in process model, organization, methods, and tools**. It is possible to avert or delay design erosion, if your process model, organization, methods, and tools are appropriate. Testing, organisation issues, and many other aspects play an important role. In product lines these factors are even more important because of the significantly larger impact of any change.
- **Traceability of design decisions**. As discussed above in section 2 (Problems & Issues) it is recommended that standard notations and proper documentations must be used at the time of creating and modifying or fixing a bug in the software. Consequently, design decisions become much easier to track and reconstruct the system.
- **Gathering of design decisions**. Design decisions accumulate and interact in such a way that whenever a decision needs to be revised, other design decisions must also be reconsidered. So that cyclic dependency can be avoided. As a result of this no such problem occurs under such circumstances change.
- **Use of continuous review and assessment techniques**. The system architect should not only focus about creating system, it must also constantly review, assess and improve existing system. As change is the rule, architectures grow and change continuously. To keep them sustainable, review and assessment techniques are of utmost importance. An architecture review consists of different phases: clarification or scoping phase, analysis or information phase, evaluation phase, and feedback phase (most important). The findings can be used to refactor and improve the system performance.
- **Iterative methods.** Our software systems' architecture must be open to allow tactical changes in system, when priorities are changing, then even the strategic design might be subject to change. The various development methods of iterative nature should accommodate such changes.
- **Architectural Drift.** As discussed that architectural drift occurs due to non or incomplete implementation of design decisions by the development team. The designs must be documented in detail by the actual designers, so that the architectural implied rules become clear to the software engineers who work with it. Various analysis tools (like Findbugs, Sonargraph, Structure101) should be used. Functionality should be checked by Static Code Analysis, Architecture Conformancy Check and Source code Metrics (Average Component Dependency-ACD, Normalized cumulative component dependency -NCCD).

- **Vaporization of Design Decisions**. To address the above discussed problems, [Jansen & Bosch 2005] recommended a new perspective on software architecture: define software architecture as the composition of a set of architectural design decisions. This reduces the knowledge vaporization of design decision information, since design decisions have become an explicit part of the architecture.
  The architectural design decisions are an important part of Architectural Knowledge (AK). To reduce AK vaporization, [Tofan, Galster, and Avgeriou, 2011] propose to apply the Repertory Grid Technique (RGT) to make tacit knowledge about architecture decisions explicit. An architect can use the RGT to extract decision alternatives and concerns, and to rank each alternative against concerns.
- **Refactoring.** We must make software systems open, to allow tactical extensions. Using additional use cases or when priorities are changing, then even the strategic design might be subject to change. Refactoring will be an important tool for architects.
  A lot of the code refactoring in between the versions involve a lot of mechanical changes (e.g. cutting and pasting lines of code), locate erosion point (using Reflexion models [Murphy, Gail C.; Notkin, David; Sullivan, Kevin 1995] or DCL-Dependency Constraint language) [Terra, Ricardo; Valente, Marco Tulio; Czarnecki, Krzysztof; Bigonha, Roberto S. (2012] and later replace violations with architectural consistent source code. This suggests that some of these refactoring can be automated as for instance is done for some refactoring in [Roberts, Brant, Johnson 1997]. Later design decisions become more difficult because the earlier design decisions have to be taken into account. Even in a small prototype, Gurp & Bosch had to deal with the legacy of the first few versions when going from version 4 to version 5. This caused them to move around a lot of code. There has been some promising research into code refactoring (most notably [Fowler et al. 1999] and [Roberts, Brant, Johnson 1997]). However, more advanced, preferably automated, refactoring would be useful.

As part of our observations and discussions, few ideas are recommended to further improve the design and development process:

- **Testing Process**. To establish an efficient and trustworthy safety net for the developers, tests need to become more stable and trustworthy. With this, the architects can introduce what they call "pain-driven development". That is, when a developer introduces or changes code that breaks the tests, he will be notified immediately to correct it. It is recommended that too much of manual testing should be avoided and more automation is desired.
- **Restructuring and Refactoring goals.** The components of the software need to be de-coupled from the core and the removal of the overlapping and duplicated code. It is felt that the system should have a clearer separation of concerns were modularization should reflect business segments and the system should better separate business and platform related code.
- **Continuous Monitoring of Quality**. The architects proposed a principle [Hanssen, Yamashita, Conradi and Moonen 2010] that they refer to as "quality-from-now", meaning that any change to the code should be analyzed at development time, to check that it does not conflict with defined rules of good design. This can, for example, be achieved using a tool like NDepend, by defining CQL rules to detect code smells and monitor potential problems nearly constantly during development. It is believed that this approach would considerably reduce the fear of changing the code.

As pointed out in this paper, most existing development methods are flawed because they iteratively accumulate design decisions. Since it is inevitable that requirements change over time, it is also inevitable that sooner or later design erosion occurs (because some of the earlier decisions become invalid). Current research focuses on fighting the symptoms (i.e. design erosion) rather than the problems. New methodologies such as extreme programming [Beck 1999] address this by adopting a stepwise refinement strategy with frequent releases. However, there are issues with respect to, among others, planning and cost management of projects using such methods.

## VI. CONCLUSION

An important conclusion is that even an optimal design strategy without any compromises e.g. cost, for the system does not deliver an optimal design. The reason for this is the changes in requirements that may occur in later evolution cycles. Such changes may cause design decisions taken earlier to be less optimal. Design erosion is inevitable with the current way of developing software. Refined methods only contribute by delaying the moment that a system needs to be withdrawn or retired. These approaches do not address the fundamental problems that cause design erosion. Rather than fight the symptoms we should start to address the causes.

An issue that we intend to explore is that of the design method. It seems that the current practice of software development is to create a design in advance. However, as noted earlier this conflicts with the iterative nature of many development methods. New requirements are constantly added to the system and as discussed they often conflict with design decisions taken in earlier iterations or in the design phase. We believe such conflicts are the primary cause for the phenomena of design erosion.

## VII. Future Work

As it is inevitable that requirements change over time, it is also inevitable that sooner or later design erosion occurs, as with passage of time some of the earlier decisions become invalid or obsolete. Current research focuses on fighting the symptoms (i.e. design erosion) rather than the problems.

## REFERENCES

[1] Beck 1999. K. Beck, *"Extreme Programming Explained"*, Addison Wesley 1999.

[2] Bengtsson & Bosch 1998. P. Bengtsson, J. Bosch, "Scenario-based Software Architecture Reengineering", *Proceedings of the 5th International Conference on Software Reuse*, pp. 308-317, June 1998.

[3] Bosch 1999a. J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

[4] Bosch 2000. J. Bosch, "Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach", Addison Wesley, ISBN 0-201-67494-7, June 2000.

[5] Bosch et al. 1999b. J. Bosch, P. molin, M. Matsson, P.O. Bengtsson, "Object Oriented Frameworks - Problems & Experiences", in *"Building Application Frameworks"*, M.E. Fayad, D.C. Schmidt, R.E. Johnson (editors), Wiley & Sons, 1999.

[6] Bushman 1996. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[7] Fowler et al. 1999. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring - Improving the Design of Existing Code", Addison Wesley, 1999.

[8] Gamma et al. 1995. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading MA, 1995.

[9] Gurp, Bosch 2002. J. van Gurp, J. Bosch, "Design Erosion: Problems & Causes", *Journal of Systems & Software*, 61(2), pp. 105-119, Elsevier, March 2002.

[10] Hanssen, Yamashita, Conradi and Moonen 2010. Hanssen, G.K. ; Yamashita, A.F. ; Conradi, R. ; Moonen, L., "Software Entropy in Agile Product Evolution", *Proceedings of the 43rd Annual Hawaii International Conference on System Sciences (HICSS), 2010, pp.* 1 – 10, ISSN : 1530-1605.

[11] Jaktman, Leaney, Liu 1999. C. B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", in *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[12] Jaktman, Leaney, Liu 1999. C.B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architec-ture - A Maintenance Assessment Case Study", *The First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer Academic Publisher, 22-24 February 1999, San Antonio, TX, USA.

[13] Jansen & Bosch 2005. A. Jansen, J. Bosch, "Software Architecture as a Set of Architectural Design Decisions", *Proceedings of 5th Working IEEE/IFIP Conference on* Software Architecture, 2005. ISBN: 0-7695-2548-1, pp. 109 - 120, WICSA 2005.

[14] Lieberherr 1989. K.J. Lieberherr, I.M. Holland, "Assuring Good style for Object Oriented Programs", *IEEE Software*, September 1989, pp. 38- 48.

[15] Lieberherr 1996. K. Lieberherr, "Adaptive Object-Oriented Software - The Demeter Method", PWS Publish-ing company, 1996.

[16] Murphy, Gail C.; Notkin, David; Sullivan, Kevin (1995): Software reflexion models: bridging the gap between source and high-level models. In SIGSOFT Softw. Eng. Notes 20 (4), pp. 18–28.

[17] Naur & Randell 1969. P. Naur, B. Randell. "Software Engineering: Report on 1968 NATO Conference", *Nato*, 1969.

[18] Parnas 1994. D. L. Parnas, "Software Aging", *Proceedings of ICSE 1994*, pp 279-287.

[19] Perry, Wolf 1992. D. E. Perry, A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIG-SOFT Software Engineering Notes*, vol 17(4):40–52, 1992.

[20] Roberts, Brant, Johnson 1997. D. Roberts, J. Brant, R. Johnson, "A Refactoring Tool for Smalltalk", *Theory and Practice of Object Systems* Vol 3(4): 253-263, 1997.

[21] Svahnberg & Bosch 1999. M. Svahnberg, J. Bosch, "Characterizing Evolution in Product-Line Architec-tures", *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*, pp. 92-97, October 1999

[22] Swanson 1976. E. B. Swanson, "The dimensions of Maintenance", *proceedings of the 2nd International Conference on Software Engineering*, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976.

[23] Terra, Ricardo; Valente, Marco Tulio; Czarnecki, Krzysztof; Bigonha, Roberto S. (2012): Recommending Refactorings to Reverse Software Architecture Erosion. In Tom Mens, Anthony Cleve, Rudolf Ferenc (Eds.): 16th European Conference on Software Maintenance and Reengineering (CSMR 2012): IEEE. Available online at http://homepages.dcc.ufmg.br/~mtov/pub/2012_csmr_era.pdf.

[24] Tofan, Galster and Avgeriou, 2011. Dan Tofan, Matthias Galster, and Paris Avgeriou. "Reducing Architectural Knowledge Vaporization by Applying the Repertory Grid Technique", *Proceedings of the 5th European conference on Software architecture ECSA'11, Springer-Verlag Berlin, Heidelberg, 2011*. pp. 244 – 251, ISBN: 978-3-642-23797-3.

[25] Van Gurp & Bosch 1999. J. van Gurp, J. Bosch, "On the Implementation of Finite State Machines", in *Pro-ceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 172-178, 1999.