Software Architecture Erosion: Impacts, Causes, and Management

Sharon Andrews *Whites@uhcl.edu*
Associate Professor, Software Engineering
University of Houston Clear Lake
Houston, TX 77058, USA

Mark Sheppard *SheppardM@uhcl.edu*
Software Engineering
University of Houston Clear Lake
Houston, TX 77058, USA

Abstract

As extensive new software projects are becoming more costly with increased development complexity and risks, the use of existing, already developed software projects are now spanning multiple years or decades. Software maintenance durations are increasing along with periodic periods of intense software upgrades due to these lengthy extensions of the life of the software. While the software architecture is set during the project's initial design phase, over the life of the project, architecture erosion can take place due to intentional and unintentional changes, resulting in deviations from the intended architecture. These deviations can manifest themselves in a wide variety of performance, maintenance, and software quality problems ranging from minor annoyances to product failure or unmaintainable software. This paper explores the causes and impacts of software erosion, and recovery approaches to addressing software erosion, laying the framework for future work towards the definition of an Architectural Maturity Model Integration process to assess the architectural maturity of an organization.

Keywords: Software Architecture, Architecture Erosion, Software Maintenance, Software Quality, Software Erosion, Legacy Code, Architectural Maturity Model.

1. INTRODUCTION

Recent trends show that new software projects are increasing in complexity, size, and overall development infrastructure. Examples include increases in lines of code in modern automobiles (10-100 million lines of code) to the estimated 2 billion lines of code within the Google infrastructure [1]. Correspondingly, software development costs have increased along with project duration, staffing, capital resources, and organizational overhead. Along with these factors, the risks associated with cost and schedule overruns, not meeting customer requirements, late to market, or outright project failure has increased. A direct result of increased development costs and associated risks is a desire to extend the life of existing production software through continual maintenance and periodic enhancements resulting in software lifecycles spanning several decades. Typical software upgrades include updates to meet changing technology, new user interfaces, increase capabilities, and increased security needs.

However, software that has long life cycles can experience erosion of the implemented architecture from the intended architecture. This erosion can affect the performance of the software, development activities, routine maintenance, and software quality factors. Extreme cases of erosion can result in brittle software or failure of the software to meet its main objectives with severe consequences, including extensive rewrite activities or an inability to maintain the software.

In its basic definition, software erosion is the difference between the intended software architecture and the existing architecture. Creation of the software architecture takes place within the design phase of this life cycle. Within an architecture design, fundamental properties of a system are embodied within the architectural elements, and the relationships between these elements are defined [2]. The architecture design of the system strives to capture a focused organization of elements to support a defined functionality and solve a specific problem. Good architectures can reduce the overall complexity of the project by separating the design into manageable areas and maintaining separate areas of concern [3].

All software has an architecture, either explicitly or implicitly defined. Architectures reflect a significant decision point in the overall software project. Flexible architecture designs consider requirements that are not defined, and that may change over the project's life. The architecture should be derived from the customer's needs, the environment in which the software will operate, project-specific concerns, and project quality requirements. The project's software is developed within the architecture structure. While the architecture is set to meet the specific requirements and goals, as the software progresses through its lifecycle, due to a variety of reasons, the implemented architecture deviates from the intended architecture [5], [6]. Typically, this deviation is unintentional and, in many cases, not detected until acute systemic conditions arise. Such deviations are called erosions. Corrections for software erosion are often in response to observed system bugs/failures well after erosion has been ingrained into the software. Preventative measures are typically process-oriented [4].

2. SOFTWARE EROSION

Some of the main assumptions surrounding erosion are that erosion occurs during the maintenance phase, where the software spends a majority of its lifecycle, that the erosion is gradual, and that the erosion is unintentional [5]. Certainly, during the maintenance phase, the software can undergo many small changes that can degrade the architecture over time. However, erosion can occur as soon as the architecture is chosen, and before detailed design and implementation begin [6]. Another typical assumption is that erosion is a slow, gradual effect over a long time. That small deviations to the architecture can build up over time and result in ever-increasing levels of erosion [5]. While this is a significant cause of erosion, rapid erosion can also occur during sprint periods where significant changes such as upgrades, platform changes, or user interface changes are taking place in a short time interval. During these sprint periods, numerous changes across the architecture are occurring at the same time and, typically, for a system that must maintain continuous production, time constraints exist, exacerbating typical maintenance phase activities, creating rapid erosion.

Another common assumption is that erosion is an unintended effect associated with maintenance. However, erosion can be intentional and even planned. Some erosion takes place as incurring "technical debt," where short term erosion is accepted with the intent of "fixing" the architecture later, generally when the team will have more time [7]. Technical debt can occur as early as the Requirements phase of the project [8]. Technical debt is generally used to meet schedule pressures by trying to get the product to the customer on-time. However, if the technical debt is not resolved promptly, then the technical debt can increase, resulting in increased unplanned erosion. The costs associated with fixing deferred work can increase the longer the repairs/recovery are deferred. Intentional erosion can also occur as an attempt to fix or repair real or perceived deficiencies in the original architecture. If the architecture does not meet the emergent requirements flowing from the customer's needs, then the design team may try to resolve the discrepancy by intentionally deviating from the intended architecture. However, if an architecture evolution plan is not in place and communicated across the development team, the result may be more erosional than helpful. In practice, it can be difficult to adequately classify the precise cause of the erosion as the overall impact to the architecture can be very gradual and can be the result of multiple effects [5], [7].

2.1 Software Erosion - Impacts

Software erosion may not have noticeable effects in the short-term, but over the long life of the project, erosion can manifest itself in an extensive set of symptoms ranging from minor annoyances and bugs to the customers, operators, and developers, to full failures of the software to meet customer requirements, expectations, and mission goals. The effects of software erosion can affect multiple areas of software development activities (development, test, management, scheduling, finance, etc.) as well as the final integrated product. The observed impacts may present themselves as fairly benign in the early stages of erosion, but the long term erosion can have a range of effects on performance and maintenance. Performance effects can range from relatively minor annoyances to decreased performance, to lost capability, and maximally to full failure, while effects on maintainability can range from minor refactoring needs to increases in code inconsistencies, escalating bug fixes, and maximally reaching an unmaintainable state.

As erosion builds up over the lifecycle of the software project, the impacts of the erosion can manifest in the following ways:

a) Inability to meet requirements
   Features may not work correctly as a result of broken or buggy code. The number of software bugs may increase during attempted repair leading to frustration and/or the inability to meet user goals.
b) Reduced product performance
   Products may still be able to meet the original requirements and goals, but the overall performance of the product may be impacted as the erosion may result in the product performing slowly. Efforts to fix performance issues can erode the system even further.
c) Increased time for updates
   The time required to maintain or update the software may increase, resulting in increased schedule risks and increased time-to-market. All areas of the development process can be affected as code updates must also repair architecture areas that have become corroded by erosion requiring increased levels of refactoring and testing.
d) Increased costs
   Associated with the increased time required for updates, is the related direct costs. Software can become increasingly more expensive to maintain and upgrade. Indirect costs such as missed delivery times, reduced functionality, customer dissatisfaction, loss of business/customers due to the poor performance of the product, and reputation damage can result from the effects of erosion.
e) Degradation of quality attributes
   Quality attributes associated with the software tend to deteriorate with increased erosion. The code becomes more difficult to maintain, modify, test, and deploy. Increased levels of erosion tend to cause erosive effects to accelerate. As erosion levels increase, there is typically a steady degradation of the well-known design principles associated with higher quality architectures and quality code such as separation of concerns, Law of Demeter, Principle of Least Privilege, information-hiding and encapsulation.
f) Brittle code
   As erosion increases, the resulting required repairs can easily degrade the architecture's code elements resulting in less maintainable, brittle software. Brittle code reflects a system in which even small changes to the software can result in increased bugs, loss of functionality, and failure of the software.

2.2 Software Erosion – Technical Causes

Software erosion causes can be split into two broad categories, technical and non-technical. Technical causes are those stemming from directly dealing with the design and implementation of the architecture, while non-technical causes are those stemming from organizational and staffing considerations. Below we list some of the known technical causes for software erosion and address the non-technical causes in section 2.3.

a) Design decisions violate the architecture
   Detailed design decisions may conflict with the intended architecture. This is particularly true in agile-based development where only parts of a project are carefully designed at a

time. Even in plan-based development environments where designs are fully considered before implementation begins, design changes can be necessary due to arising implementation problems as well as changing requirements.

b) Design not properly conveyed
If the design is not properly conveyed to the development team, then it's almost certain that the architecture will not be implemented properly. Poor design documentation, changing staff, and constantly changing requirements or emergent requirements all contribute to communication deficiencies between the design team and the development team, causing deviations from the intended architecture.

c) Architectural style violation
An architectural style violation is any software change that violates the style rules associated with the intended architecture. A typical example is one involving layered architecture interactions restricted to communication between adjacent layers only. Style violations occur when the implementation or maintenance team writes codes that intentionally bypass adjacent layers to directly communicate with non-adjacent layers. Another example, though at a lower level of detailed design, would be the violation of the rule to only access a module implementing a stack through its top-of-stack method. Writing code to directly access the middle or any other location would violate this style rule [9].

d) Orphan elements
Orphan elements are software elements that are no longer used but are still in the software build, such as can occur is with the use of the Swiss Army knife antipattern [8]. This pattern is reflective of the coding method in which non-working code is added to the software build to address all possible requests of the software, including those not called out in the requirements or customer needs. These elements can create confusion for the development team and add needless complexity.

e) Duplicate Code or “Clone elements”
Clone elements are software elements that provide nearly identical functionality or services. [5] Changes in one software element may not have the intended effect since the cloned element has duplicated the service. In this case, changes must be applied to all instances of duplication to ensure the change is propagated through the program.

f) Incorporation of legacy or reusable software
While the incorporation of reusable software is considered beneficial by using readily available software, many times, compromises must be made to fit two different architecture together. Such piecemealed code can create trigger points for future erosional behavior as typically, this results in including clone and orphan elements.

g) Increased coupling
Increased coupling can be both a cause and an effect of increased erosion. Coupling across module and component boundaries can increase with increased levels of erosion. Architecture rule violations can directly lead to an increase in coupling, reflecting the fact that interactions across disallowed architecture element boundaries may increase. This eroded architecture may no longer be an optimal or viable solution to the original problem. With increase coupling, complexity increases along with a decreased understanding of the code by the development team. Also, testing becomes more difficult, leading to increases in the number of bugs.

h) Decreased cohesion
Like coupling, decreased cohesion can be a cause of erosion and an effect of on-going erosion. The architectural components (including architectural elements, modules) may develop co-incidental cohesions where modules of architectural elements are arbitrarily grouped together. An eroded architecture can result in decreased cohesion across the architecture. Architecture elements that were developed to provide specific functionality may be compromised by the erosion. As the software matures or during the maintenance phase, there may be an effort to limit the creation of new elements, which may result in less than ideal cohesion.

i) Increased inheritance hierarchies
Architectures that exhibit deep inheritance hierarchies can lead to the fragile base class problem, in which base classes are difficult to change due to the increased coupling and increased maintenance and testing issues stemming from deep inheritance [10]. While deeper inheritance trees can create a higher reuse potential, such may also present increased complexities of dependence [11]. Specific metrics for optimal inheritance depths are not available, but the upper limits of inheritance levels of 5-6 have been suggested [12].

j) Increased complexity
As complexity within the architecture increases, understanding of the architecture becomes more challenging, resulting in potentially more architecture violations and erosion.

k) Incorrect architecture for the project
While not technically considered erosion, incorrect or bad architecture can cause later erosion. Changes erode the architecture as later design, implementation, and maintenance attempts to overcome the shortcomings of the original architecture. If these changes are not properly managed or repaired, or an architecture evolution is not planned properly, the resulting erosion could be far worse than just maintaining the original bad architecture.

2.3 Software Erosion: Non-Technical Causes

Non-technical causes for erosion can be attributed to the culture and processes adopted within an organization that creates a likely environment for software erosion to occur.

a) Schedule pressures
Schedule pressures to get the product out the door quickly may result in unintentional or intentional architecture erosion.

b) High staff turnover
A continually changing staff leads to team members not understanding the architecture, the development process, or company expectations, thus making changes and improvements to the software without team members experienced with the system or the process in place to change the system which inherently leads to a higher chance for erosion.

c) The software development process followed
The organization may follow a development process that ignores or places no emphasis on architecture or design, e.g., a strict agile-based development process requires the development of a software product in successive small increments without the requirement for completion, or adherence to a product-wide software design or architecture. Such an agile design phase is not explicit but is embedded in code increments, so there exists no defined, documented system-wide architecture. Rather, the architecture is embedded in the resulting project but not visible or documented which typically results in high levels of erosion [6].

d) Organizational has no process
Software development proceeds in an ad-hoc manner from employer to employee. Typically, in such an environment, no well-documented architecture would be produced, and code development and maintenance undertaken would be highly likely to result in erosion.

e) Organizational culture
Some organizations maintain a more rigorous approach to prevent or mitigate architecture erosion through business and software processes and company-wide best practices that lead to more quality-driven product development. One such culture takes place in the domain of critical software system development responsible for producing life-critical software. The more rigorous the organization's approach, the less likely erosion will occur.

2.4 Types of Erosion Management Methods and Relative Cost

There are three broad categories of approaches to address software erosion: prevention, minimization, and recovery [13]. The most common of these are recovery and minimization since prevention is itself a form of minimization. However, minimization methods themselves can be further categorized into two main types, those that are process-based and those that are formal methods based. Each of these approaches is associated with methods requiring varying levels of difficulty of application and costs to an organization. These approaches, costs, and methods are summarized in Table 1, along with a ranking of relative difficulty to implement as well as the general category of methods used. Certainly, with a no-action approach to minimize or prevent erosion, there is no cost or method as erosion is allowed to occur, and no effort is made to restore the architecture or repair erosion effects. Such is applicable to projects that are small, non-critical or have no significant business value. For these projects, the organization or customers are not concerned about the impact on quality or performance factors as the project lifecycle is typically short, or the cost of minimization and recovery outweigh the cost of the project.

Approach	Description Goal of Approach	General Method	Relative Difficulty
No-Action	Erosion is allowed to occur; no efforts made to restore the architecture or repair architecture erosion	None	Low
Recovery	Repair erosion that has occurred	Refactoring to various degrees	Medium
Process-driven Minimization	Minimize the effects of erosion or decrease erosion while reducing cost and time required	Adherence to Established Process & Standards	High
Formal Minimization	Minimize effects of erosion. There is very high cost, time and training involved with these methods relative to informal ones.	ADLs, ACL, Doman language,	Very High

TABLE 1: Approaches, Methods and Costs.

Recovery approaches attempt to repair the erosion. Recovery requires finding the elements of the software that have experienced erosion, then employs methods to repair those elements. Conventional recovery methods involve refactoring the software, component, or module. Process-driven minimization methods are governed by reliance and adherence to a defined process definition that dictates how the software is designed, developed, and maintained. Such processes typically require architecture design documentation, and analysis and monitoring for compliance to all architectural constraints in an attempt to reduce erosion [14].

Other, less commonly used minimization methods are formal methods which rely on formal-method-driven techniques such as Architecture Description Languages (ADLs), and domain constraint languages, domain definition languages [15], frameworks [16], patterns, and other formal techniques to help locate and suggest repairs to architectures [17], [18]. Such methods typically provide syntactic analysis of an architecture description, extracting rules and constraint violations found [19]. These methods are not as commonly used as they require extensive training to be effectively applied and, as such, are costly and time-consuming, though effective, if used properly.

In all approaches to erosion management, effective minimization and recovery approaches require high levels of architecture analysis and documentation along with communication and coordination across the development team and constant and consistent monitoring of software changes to be effective. Such requires an organization to devote resources in the form of cost and time to address erosional concerns. Additionally, escalating degrees of rigor must be incorporated into the software development process to be able to address increasingly higher levels of minimization.

3. RELATED WORK

A great deal of work has taken place with respect to the detection, prevention and reduction of architecture erosion. All such work hinges on the fact that the process of changing the code can easily result in design violations of well-known and established design principles [20] leading to erosion and drift. Finding and tracking such violations is key to controlling erosion. Previous work can be segregated into methods that center around visualization of the erosion, methods that center on the development of architectural models that embody design constraint rules, and methods that rely on the use of dependency structures and query-based languages to define module dependency constraint rules.

Many visualization methods center on the use of specific types of antipatterns. One such work relies on the notion of circular dependencies and subtype knowledge [21] which forms the basis for finding and visualizing antipatterns [22] relating directly to violations of design principles that relate specifically to object oriented design such as the Acylic Dependencies Principle [23] and the Dependency Inversion Principle [24], [25]. Antipatterns such as these can be generically described as code segments which display atomic violations of known and accepted design principles. Visualizations of these violations are one classification of methods of discovery of instances of design rule violations within code, allowing code to be monitored for the emergence and growth of these patterns [26]. Such visualization assists software engineers and architects in assessing, tracing and therefore combating design erosion. Other work that targets the visualization of structural code changes in order to detect design principle violations have been reported within [27], [28], [29], [30], [31].

Approaches within the domain of Model Driven Development depend on the development of appropriate architectural models, as well as a set design rules that form a set of constraints on the design realization. One such approach, described by Harold and Rausch [32], is based on a set of consistency constraints expressed as architectural rules specified as formulas within a defined ontology with models mapped to specific ontologies such that a knowledge representation and reasoning system can be used to check for architectural rule conformance for a given set of models.

Query and relational calculus based methods typically support only a limited set of design artifacts to be checked as well as the inability to support the integration of an architectural description. In one such approach, Dependency Structure Matrices (DSM) [33] are used to capture the dependencies between the modules of systems where lines and row represent modules of a system and matrices indicate dependencies as binary values or weighted numerical values. A similar approach, Lattix LDM [34], provides for the definition of constraints that restrict the values the entities may the assigned. Other approaches rely upon relational calculus approaches for use with object oriented source code like CQL, QL or JQuery [35], [36], [37].

Model-Driven reverse engineering models [38] and Reflexion models [39] are another more sophisticated approach to checking architectural conformance. Such models rely on the architect to manually create an abstract architectural model which defines the architectural components and the dependencies between the architecture components, as well their mappings to the realization source code models. Some examples of this approach are ConQAT and Bauhaus [41], [42].

The choice of which approach, or set of approaches, an organization should adopt as the underlying tool and method support for the detection, prevention and reduction of architecture erosion and drift is a complicated one and certainly would be organization-specific depending on the products being developed, the skill of the development team and the architectural maturity of the organization as is discussed in the remainder of this paper.

4. CONSIDERATIONS OF AN ORGANIZATIONS ARCHITECTURAL MATURITY

The software process in place within an organization governs how an organization recognizes and deals with the development and maintenance of an architecture. This process is a direct reflection of what the authors describe herein as the Architectural Maturity Level of an organization. Architecturally-aware [43] and architecture-centric organizations are by default more mature with regard to controlling and managing erosion and drift. However, such organizations need to meet at least one or more of the requirements presented in Table 2 to be more formally considered architecturally aware and architecture centric. The more of these requirements that are met the more architecturally mature. These requirements form the basis for process steps the organization must establish, require, and support to begin to control erosion and drift of essential critical software.

	Organizational Requirements	**Effect on the organization**
1.	Organization must be architecturally aware	Organization understands the importance of architecture and has an explicitly well-defined process for architecture design and modification in place that employees are expected to follow.
2.	Architecture requirements must be made explicit, documented and traceable to quality requirements and major functional requirements.	The organization's process must be explicit in how these requirements are expressed and maintained, including details such as format and all systems and models required for the expression and understanding of these requirements.
3.	Conformance checking must be performed on all architectures before implementation and during product evolution.	Such conformance checking steps, and the output from each step, must be explicitly defined within the organization's software development process.
4.	Architecture must be updated to reflect new requirements before implementation may be changed	Changes to the product must be controlled so that the architecture is changed to reflect any implementation change.
5.	Any implementation increment must conform to the architecture.	Before a product increment can be released, the code must pass a series of checks that it conforms to the architecture design. The exact steps and tools to effect this must be part of the organizations' adopted development process.

TABLE 2: Organizational Requirements for Architectural Maturity.

An organization that has a defined software development process in place that details exact support for each of these five requirements would certainly be deemed a more architecturally mature organization than those failing to meet one or more of the above requirements for the mature management of architectures. Certainly, the problems of architecture erosion and drift are directly and positively impacted by the adoption of these principles and the causes, impacts and management issues presented herein must be taken into account in an organization's specific process steps and output documents produced by such steps.

5. CONCLUSIONS

Software projects have become more costly to develop from the ground up due to increasing costs, shorter schedules, and increased complexity. Legacy software is exhibiting longer software lifecycles (spanning decades) and undergoes lengthy periods of maintenance and spurts of concentrated upgrades and enhancements. In essence, in many cases it's more cost-effective to reuse or modify legacy software than to go through the design/development process for a new software project. This legacy software requires continuous updates to address changing technology, emergent customer requirements, evolutionary user interfaces, changing deployment strategies, and security concerns. However, bug fixes, enhancements, and upgrades can introduce erosion that gradually builds over the life of the project. Software with longer lifecycles typically suffer from erosional behaviors resulting in negative impacts on performance, maintainability, reliability, and quality. Performance-affecting erosion can span the spectrum from minor annoyances to software that is unable to meet customer needs or that is operationally unstable or even unusable. Maintainability impacts can span from minor refactoring needs to unmaintainable and brittle code. While erosion is typically associated with gradual deviation from the intended architecture during the maintenance phase, erosion can start as soon as the architecture is defined and rapidly progress. Root causes that set the environment for erosion to occur include schedule pressure, incorrect architecture, poor processes in place, staff turnover, and even organizational culture. Organizations that wish to reduce architectural erosion need to plan the management of erosion by way of process for prevention, correction, and reduction. This requires that organizations put into place architecture-centric process steps that address and define how the architecture will be defined, maintained and updated taking into account the causes, impact, and requirements presented herein.

6. FUTURE WORK: AN ARCHITECTURAL MATURITY MODEL IS NEEDED

Organizations vary greatly in their competence in managing architecture design, not to mention erosion and drift. Indeed, for important life-critical software development work, an organization's architectural maturity could be the overriding factor in choosing among competing bids. It is typically accepted that an organization faced with choosing a software contractor for a large critical project would likely choose a contractor with a high level of architectural competence [44]. Therefore, assessment of such an organization's maturity could provide organizations with a way to deem themselves as competent to a certain maturity level, much like the current and well know CMMI-DEV [45] model where an organization can be assessed as being at progressively competent maturity levels concerning their product development processes and methods. Capturing an organization's architectural competence within a multi-dimensional view of competence based on specific architecture processes, and steps within these processes, is needed. Given such an "Architecture Maturity Model Integration (AMMI)" it could be possible to assess an organization to its architectural maturity. The more reliable and higher quality are the products produced, the higher the maturity level. While software architecture is a small aspect of the CMMI it is not explicitly addressed in such a way that an effective AMMI assessment could be applied. Thus, future work is needed towards the definition of a specific architectural maturity model that can provide a means to assess organizations with respect to their architectural maturity and, at the same time, provide process guidance to organizations to develop their architectural maturity. Indeed, the authors have begun laying the foundation for such work with the work reported within which provides a framework of necessary foundational knowledge required for this model.

7. REFERENCES

[1] B. Algaze. "Software is Increasingly Complex. That Can Be Dangerous." Internet: https://www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous, Dec. 7, 2017 [Dec. 20, 2019].

[2] S.A. White. "Software Architecture Design Domain." In Proc. of the Second World Conference on Integrated Design and Process Technology, Vol. 1, Austin, TX. Dec. 1-4, 1996, pp. 283-290.

[3] S. A. White. "The Repository Based Software Engineering Program". in Proc. of the 1996 workshop, A NASA Focus on Software Reuse*.* George Mason University, Fairfax, Virginia pp. 53-62, September 24-27, 1996.

[4] H. Koziolek, D. Domis, T. Goldschmidt and P. Vorst. "Measuring Architecture Sustainability," IEEE Software, vol. 30, no. 6, pp. 54-62, Nov.-Dec. 2013.

[5] M. Dalgarno. (2009, Spring). "When Good Architecture Goes Bad," Methods and Tools [On line] Available http://www.methodsandtools.com/archive/archive.php?id=85, [Dec. 28, 2019].

[6] E. Whiting and S. Andrews. "Drift and Erosion in Software Architecture: Summary and Prevention Strategies," to appear in Proceedings ACM 4th International Conference on Information System and Data Mining (ICISDM). Hilo, Hawaii, May 15-27. 2020.

[7] M. Fowler. "Technical Debt Quadrant". Internet: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html, October 14, 2009 [Dec. 28, 2019].

[8] N. A. Ernst, (2012, Jun). "On the Role of Requirements in Understanding and Managing Technical Debt," 2012 Third International Workshop on Managing Technical Debt (MTD). [On-line]. Available: http://ieeexplore.ieee.org/document/6226002/, [Dec. 28, 2019].

[9] S.A. White and C. Lemus. "Architectural Reuse in Software Development," in Proc. 20th International Computers in Engineering Symposium (ASME-ETCE98) Jan. 1998, pp. 1-8.

[10] L. Mikhajlov and E. Sekerinski. "The Fragile Base Class Problem and Its Solution," Internet: https://pdfs.semanticscholar.org/d9ba/ed252181ac28b6814bd2330b48837747b641.pdf, June, 2997 [Dec. 28, 2019].

[11] Z. Naboulski. "Code Metrics – Depth of Inheritance (DIT)," Internet: https://blogs.msdn.microsoft.com/zainnab/2011/05/19/code-metrics-depth-of-inheritance-dit/, May, 19, 2011 [Dec 20 2019].

[12] R. Shatnaw. "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems," IEEE Transactions on Software Engineering. Vol: 36, Issue: 2 pp. 216–225, 2010.

[13] L. De Silva, & D. Balasubramaniam. "Controlling software architecture erosion: A survey," Journal of Systems and Software. Vol 85, Issue 1, pp. 132-151. Jan. 2012.

[14] R. Terra, M. T. Valente, K. Czarnecki and R. S. Bigonha, "Recommending Refactorings to Reverse Software Architecture Erosion," in Proc. 16th European Conference on Software Maintenance and Reengineering, Szeged, 2012, pp. 335-340.

[15] S. A. White, "A Design Metalanguage for Design Language Creation," In Proc. ASME and API Energy Information Management - Incorporating ETCE, (ASME-ETCE96) Houston TX, Jan. 29 - Feb 2, Vol. I Computers in Engineering, 1996, pp. 135-144.

[16] S.A. White. "A Framework for the development of Domain Specific Design Support Systems". in Proc. First World Conference on Integrated Design & Process Technology, Austin, TX. IDPT- Vol 1, Dec. 6-9, 1995, pp. 37-42.

[17] S. Schröder and M. Riebisch. "Architecture Conformance Checking with Description Logics," ECSA '17: in Proc. 11th European Conference on Software Architecture. Sep. 11–15, pp. 166-172, 2017.

[18] M. De Silva and I. Perera. "Preventing Software Architecture Erosion Through Static Architecture Conformance Checking," in Proc. IEEE 10th International Conference on Industrial and Information Systems (ICIIS0), Peradeniya, 2015, pp. 43-48.

[19] G. Murphy, K. Sullivan, D. Notkin. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," ACM Software Engineering Notes. vol 20, issue 4, pp. 18-28, Oct. 1995.

[20] D. L. Parnas. "Designing software for ease of extension and contraction," IEEE transactions on software engineering, vol. SE-5, no. 2, pp. 128-138, Mar. 1979.

[21] D. Baum, J. Dietrick, C. Anslow, R. Muller. "Visualizing Design Erosion: How Big Balls of Mud are Made," In Proc. IEEE Working Conference on Software Visualization, Madrid, 2018, pp. 122-126.

[22] A. Koenig. "Patterns and Antipatterns," Journal of Object-Oriented Programming, 8(1), Mar-Apr, 1995.

[23] R. C. Martin. Design Principles and Design Patterns. Objectmentor.com, 2000, pp. 34.

[24] R. C. Martin. The Dependency Inversion Principle. C++ Report, 1996.

[25] A. J. Riel. Object-Oriented Design Heuristics. Reading, PA: Addison-Wesley, 1996.

[26] J. Brondum and L. Zhu, "Visualising architectural dependencies," in Proc. 2012 Third International Workshop on Managing Technical Debt (MTD), Zurich, 2012, pp. 7-14.

[27] X. Dong and M. W. Godfrey. "Identifying Architectural Change Patterns in Object-Oriented Systems," in Proc. IEEE International Conference on Program Comprehension. IEEE, pp. 33-42, 2008.

[28] A. Hindle, Z. M. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt. "YARN: Animating Software Evolution," in Proc. 4th IEEE Int. Workshop on Visualizing Software for Understanding and Analysis, 2007, pp. 129-136.

[29] S. Neu, M. Lanza, L. Hattori, and M. D'Ambros. "Telling stories about GNOME with Complicity," in Proc. 6th Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). IEEE, 2011, pp. 1-8.

[30] A. Telea and L. Voinea. "Interactive Visual Mechanisms for Exploring Source Code Evolution," in Proc. 3rd Int. Workshop on Visualizing Software for Understanding and Analysis. IEEE, 2005, pp. 1-6.

[31] S. Herold, M. English, J. Buckley, S. Counsell and M. Ó. Cinnéide, "Detection of violation causes in reflexion models," in Proc. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, 2015, pp. 565-569.

[32] Sebastian Herold and Andreas Rausch. Complementing Model-Driven Development for the Detection of Software Architecture Erosion, in Proc. 5th International Workshop on Modeling in Software Engineering (MiSE). IEEE, 2013, pp. 24-30.

[33] D. V. Steward, "The design structure system: a method for managing the design of complex systems," IEEE Transactions on Engineering Management, vol. EM-28, no. 3, pp. 71-74, Aug. 1981.

[34] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," SIGPLAN Notices. Vol. 40. 10 pp. 167–176, Oct. 2005.

[35] A. Avritzer and E. J. Weyuker, "Investigating metrics for architectural assessment," in Proc. Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262), Bethesda, MD, USA, 1998, pp. 4-10.

[36] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: QL for source code analysis," in Proc. of the Seventh IEEE

International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society, 2007, pp. 3–16.

[37] K. D. Volder, "JQuery: a generic code browser with a declarative configuration language," in Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 3819, Van Hentenryck P. Ed. Berlin: Springer, 2006, pp. 88–102.

[38] S. Rugaber and K. Stirewalt, "Model-driven reverse engineering," in *IEEE Software*, vol. 21, no. 4, pp. 45-53, July-Aug. 2004.

[39] G. C. Murphy, D. Notkin and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," IEEE Transactions on Software Eng. 27(4), 2001, pp. 364-380, Apr. 2001.

[40] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static evaluation of software architectures," in Proc. of the 10th European Conf. on Software Maintenance and Reengineering (CSMR 2006), Mar. 2006, pp. 285–294.

[41] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens, "Flexible architecture conformance assessment with ConQAT," in Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), 2010, p. 247–250.

[42] A. Raza, G. Vogel, and E. Pl¨odereder, “Bauhaus — a tool suite for program analysis and reverse engineering,” in Ada-Europe, ser. LNCS, vol. 4006. Springer, 2006, p. 71–82.

[43] L. Bass, P. Clements, R. Kazman and M. Klein, "Evaluating the Software Architecture Competence of Organizations," in Proc. Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), Vancouver, 2008, pp. 249-252.

[44] Boehm, B., Valerdi, R., Honour, E. 2007. "The ROI of Systems Engineering: Some Quantitative Results," In Proc. 2007 IEEE International Conference on Exploring Quantifiable IT Yields*,* Amsterdam, 2007, pp. 79-86.

[45] CMMI Product Team, “Capability Maturity Model Integration (CMMISM), Version 1.1”, Carnegie Mellon University Software Engineering Institute Technical Report CMU/SEI-2002-TR-012, 2002.