International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:1, 2015

A Four Method Framework for Fighting Software Architecture Erosion

Sundus Ayyaz, Saad Rehman, Usman Qamar

***Abstract*—**Software Architecture is the basic structure of software that states the development and advancement of a software system. Software architecture is also considered as a significant tool for the construction of high quality software systems. A clean design leads to the control, value and beauty of software resulting in its longer life while a bad design is the cause of architectural erosion where a software evolution completely fails. This paper discusses the occurrence of software architecture erosion and presents a set of methods for the detection, declaration and prevention of architecture erosion. The causes and symptoms of architecture erosion are observed with the examples of prescriptive and descriptive architectures and the practices used to stop this erosion are also discussed by considering different types of software erosion and their affects. Consequently finding and devising the most suitable approach for fighting software architecture erosion and in some way reducing its affect is evaluated and tested on different scenarios.

***Keywords*—**Software Architecture, Architecture Erosion, Prescriptive Architecture, Descriptive Architecture.

I. Introduction

THE main objective of software architecture is to classify the requirements having considerable influence on application structure. Business risks related in developing a technical solution can be reduced in designing a good architecture as a good design possess the quality of flexibility that can control the natural drift occurring with time in hardware or software technology or user requirements. The overall impact of architecture design decisions and the tradeoffs between quality attributes is considered by an architect. The software architecture should only represent the structure of the system by hiding the implementation details and controlling both the quality attribute and the functional requirements [1].

During the lifetime of any typical software system it undergoes evolution and creation of different prescriptive and descriptive architecture at different times. If a person doesn't have enough knowledge about what the implemented and intended architecture is then the probability of the occurrence of software erosion turns high [2], [4].

For all the software process models, architectural decisions are made early in the development lifecycle as depicted in Fig. 1.

Consider the evolutionary model in Fig. 2, where the architecture design behaves as a core of software system carried out just after the analysis of preliminary requirements.

Sundus Ayyaz is with the College of Electrical & Mechanical Engineering, National University of Sciences and Technology (NUST), Rawalpindi, Pakistan (e-mail: sundus.ayyaz@ceme.nust.edu.pk).

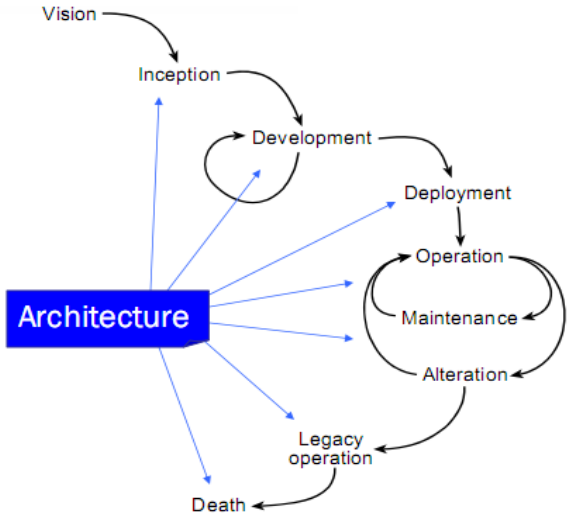

Fig. 1 Software Life Cycle [5]

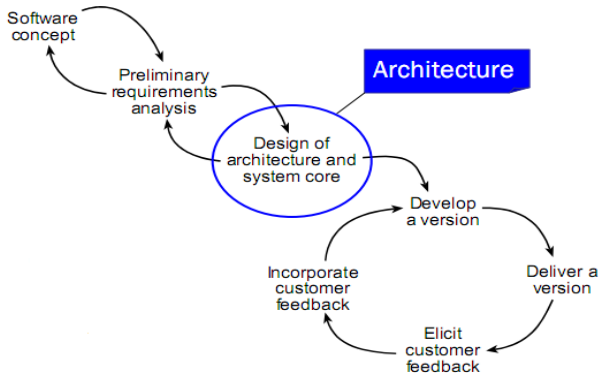

Fig. 2 Evolutionary Model [5]

The failure to follow a proper design results in architecture erosion. Therefore, the architectural decisions affects the whole life time of the software system [3], [5].

A. Architecture Degradation

The ideal case is when prescriptive architecture is equal to descriptive architecture, but with time prescriptive and descriptive change independently with the changing demands of the customer causing architectural degradation. Over time, the design decisions that are straightly applied to descriptive architecture results in either architectural drift (new descriptive decisions do not violate prescriptive) or erosion (descriptive decisions violate prescriptive decisions) [2], [11].

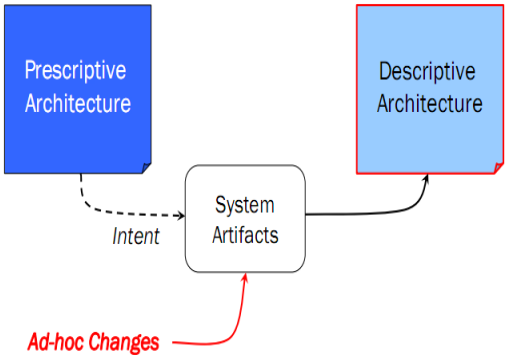

Fig. 3 Causes of Architectural Degradation [11].

B. Architecture Erosion

Software suffers from software architecture erosion, when it undergoes frequent changes during its lifetime due to technological evolution, process optimization or integrating new systems with existing software architecture. Therefore, the internal structure of the software goes through an invariable decay that can occur in any stage of the software development life cycle. At the architectural level, the software erosion can be observed from the deviation of the descriptive architecture from that of prescriptive architecture as the software evolves with time resulting in un-intended modifications (violations) of the software architecture [4,18]. The effect of architecture erosion causes the dissatisfaction of stakeholder's requirements as the changes become difficult to employ on the software and in the worst, it can even lead to failure of software projects. Almost every other project suffers from erosion at some phase in software development cycle unless some effort is done to overcome it. Architecture erosion causes multiple defects in software such as increase in internal complexity with the addition of new functionality, growing time to modify the software and time-to-market, decreased quality, increasing the test effort for maintenance of software etc. at the same time reducing the developer's productivity as much time is spent on understanding the complex existing parts of the software. The end result is: costs rises and productivity falls.

It becomes yet further expensive when software erosion results in a "software landslide", when the measure of erosion attains a point where the software cannot be maintained or improved any further and rewrite becomes the only solution, with all the employed costs and risks. As the situation gets worst, the only possible option remain is to build the software from scratch or in other words 'Rewrite' the software but this decision is exceptionally costly and risky with regards to deadlines or budget. As rewriting involves the new software to achieve all of the functionality that the existing software possess therefore no time is left for making an improvement in the software putting both the project and organization on stake [18]. That is why the software, and mainly its architecture, has to be able to deal with numerous requests for change to permanently stay in working condition [18].

C. Types of Software Architecture Erosion

The general types of software architectural erosion include:

a. *Architectural Rule violations*- For re-architecting or future development certain design rules should be followed e.g. avoidance of strict layering between subsystems [4], [6].
b. *Unreachable Code*- Also known as the dead code which is never executed nor required for any purpose but it is still messing the code base contributes towards architectural erosion [7].
c. *'Copy & Paste' Codes*- Although code duplication is popular for the purpose of reuse and implementation efficiency as copy-paste is the most common method but as the size increases the maintainability cost increases such as a fixing an error or modification in one clone case is likely to have to be disseminated to the other clone examples [4], [8].
d. *Metric outliers*- Include deeper class hierarchies, vast packages and complex code [4]
e. *Dependency*- Between packages and modules reduces reusability, obstructs maintenance, prevents extensibility, limit testability and bounds a developer capability to understand the outcomes of change [13].
f. *Cyclic Dependencies*- Are the worst type of erosion. Cycles tend to sneak into design. For instance, if A and B are placed in an alpha package, and one is placed in a num package, a cyclic dependency between alpha and num exists even though the class structure is acyclic. They should be managed or readily eliminated as they end up in fragile code [9].

D. Symptoms of Software Architecture Erosion

There are certain symptoms that indicate erosion in architectural designs. They are:

a. *Inflexibility*- makes the software difficult to change as a change can cause violation in dependant modules thus exceeding the time to perform that change, therefore the manager's fear so much that they eventually refuse to allow any changes in software [10].
b. *Brittleness*- is closely related to inflexibility causing the software to rupture every time it is modified hence the manager's fear that the software will rupture in some unanticipated way whenever they approve a fix leading towards costly rework [10]. Such software is not viable to maintain as they become worst as every change and bug fix takes considerably longer. In such cases, the developers lose the control on their software and it becomes really hard for them to work with such software and there is a force to rewrite the software.
c. *Serenity*- is the failure to reuse components from same or different software projects as most of the software involves much similar type of modules written by other developers. Serenity appears when the developers find out that the work and risk necessary to split the wanted parts of the software from the unwanted parts are too big to accept and so the software is simply rewritten instead of reused [10].
d. *Reduced Effectiveness and Efficiency*- due to delay in

software releases, budget overruns, quality defects etc.

e. Increase in time, complexity, effort and risk for implementing new functionality and a decrease in productivity and quality [4].

E. Risks of Software Architecture Erosion

There are several risks associated with software architecture [16], for example in the development team where new hires may not understand the system and old employees have to work hard, not resisting the stress, results in high turnover which is a cause for architectural decay as the knowledge of architecture is lost when they leave. Similarly inflexible software is another problem as it is very difficult to enhance and extend it. There is a chance that a modification can even cause an introduction of new bug in software therefore the software must be highly maintainable. The development team also has not a constant relationship with the software's life as there is a possibility that any member can leave the team and the knowledge of the architecture and software associated with him or her also disappears. Software Architecture erodes more when new hires make mistakes and take much time to follow up the project readily. Architecture will further erode when the new hires lacking enough knowledge about architecture would seek to make modifications to the system.

F. Real-time Example of Architecture Erosion

In recent years, many real-time examples of software architecture erosion have been observed, According to Bernhard and Frederic [12], architecture erosion occurs when the system becomes deprecate and congested. The example explains that simple software was created in March 2004 containing only 4 packages, few months later new features were added it was still going fine but in May 2005 a first cyclic dependency appeared, in June 2006 another cyclic dependency was observed and eventually in 2009 the software was surrounded by many intertwines. This project cannot be easily maintained now. The example clarifies that the architecture was perfect at the start, after making some modifications it was still going good but with time its structure was degraded with the introduction of dependency therefore a person cannot actually stop the erosion from getting in to place however measures can be done to fight against it by reducing it to some extent.

II. Proposed Solution

A lot of tools and methods can be used for stopping software erosion; one must work hard to find out the best approach that seems suitable for any particular project such as for the responsibility of a manager, he should create a culture of organization supporting and encouraging employees for the fight against software erosion, if this step is not taken than eventually no one will take interest in erosion. Similarly, for the role of architect or a developer, he should have enough knowledge about different causes of architectural erosion and different approaches for fighting it.

From the study of literature on different causes and problems associated with Software Architecture Erosion it is inferred that the number of practices have been used for stopping architecture erosion but the four most important methods for fighting against architecture erosion is formulated in this paper described as A four Method Framework; including Management Support, Maintenance, Evolvability and Refactoring as key practices.

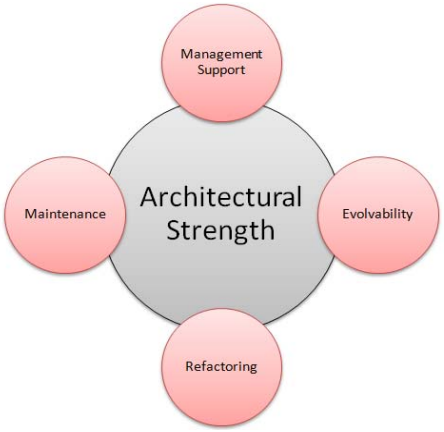

Fig. 4 A Four Method Framework for Fighting Software Architecture Erosion

A. Management Support

For the long term feasibility of the software project, Management obligation is very important for fighting architecture erosion otherwise it would become very much difficult for developer to deal with the problem in a timely manner. Therefore, if management support is provided to developers they can implement different patterns to stop the erosion effects depending upon the availability of tools, domain of the project, maturity of erosion crisis etc giving rise to a culture where fighting erosion is treasured. The culture includes functionalities like the assignment of tasks to individual persons, distribution of architectural knowledge and responsibilities and a regular communication between working group. In this way, different teams focus on different approaches contributing towards long life of software while making the development team feel dominant. Management should also create a supportive and motivating environment where appropriate training should be provided to new hires so that they must understand the system and senior employees would not be over burdened thus avoiding turnover and reducing the risk that the architectural knowledge would be lost with the person leaving the organization. The architectural knowledge should be stored in documented form with sufficient accuracy of good design decisions and using the most appropriate notation to help the new employees get the understanding of prior employees.

B. Maintenance

According to the IEEE standard for software maintenance, maintenance is defined as "Modifying different parts of a software system after delivering it to customer to correct faults, enhance performance or other quality attributes by additional functionality or to adapt the product in a customized

context" [19]. For a good architecture design, its maintenance is very much important as there is always a risk while modifying a piece of crappy software that whether the change would introduced a new bug in a system resulting in faults, therefore faults must be discovered and fixed. There is a risk that a change in a system has caused side effects to other parts in some unintended way e.g. addition of new hardware devices (computers, networks) or software change such as new operating system. Changes in customer requirements usually occurs when business environment changes or when new technology arrives in market or when the expectations of customer augments due to non-functional requirements such as safety and performance of a system or due to market competition therefore changes should be maintained as best as possible. Maintenance can either be performed by architects or developers or an additional maintenance team must be assigned. By boosting quality of architecture or code, maintainability can be improved without significant impact on the software's ability to evolve.

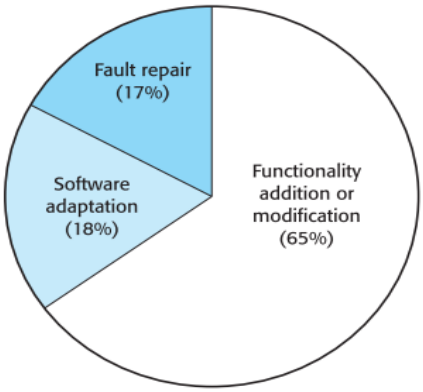

Fig. 5 Distribution of Maintenance Effort [15]

Fig. 5 illustrates that for any software, the changes are always expected and their distribution according to a survey is clarified here thus maintenance is the second major factor for fighting architecture erosion.

Lifetime of software is elongated when there is high maintainability thus lowering development risks. Maintainability is also considered as a quality attribute that only focuses on the existing short-range attempts for changes but does not emphasize on the long-standing conservation of the software. For example, the maintainability of a software system can be enhanced by improving the quality of designs and code but it would not have any sufficient affect on the capability of the software for evolvability. Maintenance activities do not consider the structural modifications as they are not involved in the maintainability of the software systems because any addition to software can result in code clones thus reducing maintainability and leading towards architecture erosion. Therefore evolvability should be considered as a separate quality attribute factor necessary for software architecture consistency.

C. Evolvability

At present, software maintenance and management support for the architecture design are not sufficient for long-life software systems, evolvability must also be considered as a significant factor for fighting architecture erosion. Evolvability is defined as an ability to keep maintainability for long run.

According to Lehman's laws of software evolution [18], the evolution of a software system is aimed at the development of that software during the software lifecycle from its initial phase to closing phase. So there is a need to tackle evolvability unambiguously during the whole lifecycle to extend the productive lifetime of the software system.

Every successful software system is prone to evolution so their architecture certainly drifts. If not any defensive work is taken on, such as building flexibility for future known changes, the software will start to wear away and the cost and risk for development increases.

Evolvability is also considered as a significant quality attribute towards maintaining the architecture design to strengthen the capability of software system to easily adjust to frequently changing needs and stay in working condition. Evolvability also contributes towards improvement of the existing system.

Therefore, for implementing something new in the software its architecture should frequently be visualized as the software is modified and compare the prescriptive architecture with the descriptive one to check how close or different they actually look or any changes are required or not. If vision of prescriptive architecture is not possible, reverse-engineering from descriptive architecture can be used. Many basic free tools as well as advanced commercial tools are available now for the purpose of architecture visualization and checking.

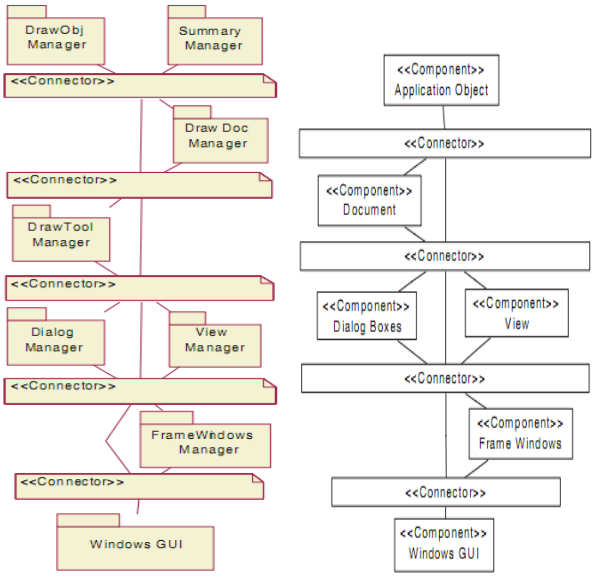

Fig. 6 Prescriptive vs. Descriptive Architecture [14]

A prescriptive architecture is an ideal visualization deduced from the architectural style and behavior whereas the descriptive architecture is extracted from the source code to get the actual components and association between them in the implementation. The goal of following this method is to remove inconsistencies with respect to the implementation. For this purpose different components of the architecture is studied in detail in order for carrying out different evolution requirements.

Fig. 6 visualizes a prescriptive and a descriptive architecture which goes through evolution as new changes are implemented.

D.Refactoring

Refactoring is a method used to reverse software erosion when it is identified. Refactoring is a technique for improvement of the restructuring of code by changing its internal structure without altering its external behavior [20]. It is a process of improvement for existing software. Refactoring is very advantageous for the elimination of architecture destruction, removal of code cycles, trimming off dead code, combining of code clones and for solving metric outliers since they clear the way of other refactoring by fighting software erosion [4]. Many different types of architecture erosion can be eliminated with Refactoring. Fig. 7 shows that most software architectures after a long time look like this and the original architecture image of the software is hardly noticeable. The figure shows that design defects are erased by several small and local corrections, missing parts are attached via knapsack, as a result such architecture erodes to failure before implementation or moving into function as it suffers from the lack of development qualities such as evolvability and maintainability.

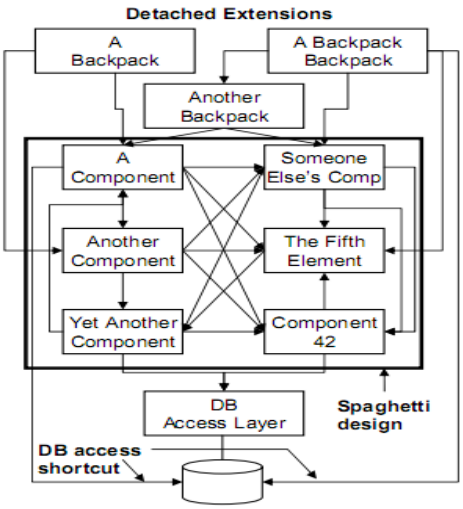

Fig. 7 Software Architecture after much time [17]

The simple steps for architecture refactoring involves, Start creating software architecture in small increments where each increment includes:

1. Top-down improvement activities to specify and complete the software architecture.
2. Bottom-up refactoring activities for plotting and clearing out conflicting or inadequate design decisions [17].

Architecture Refactoring is a means of guidance for architects for classifying possible problem in software architecture and providing solutions for solving such problems.

For architecture refactoring analysis or testing the resulting architecture after it has been refactored is compared with the preliminary architecture, also any of the verification technique can be used for the guarantee of quality [17]. Refactoring can also be reversed for example renaming of entities can also be undone or similarly merging modules can still be unmerged therefore, refactoring patterns can be implicitly understood and applied in reverse direction. Apparently, refactoring should only be applied in that direction which contributes towards improving the quality of architecture and eliminating architectural erosion and it never means that the chain of applying refactoring steps would only be reversed in a planned order.

III. Evaluation and Results

The proposed approach is applied and evaluated on various architectural styles. There are various different types of architecture smells from a huge set of software architectures that led to software architecture erosion. Some of them are; cyclic dependencies, overloaded module responsibilities, different classes with same methods, ambiguous entities in the architecture, module dependency on system, alternate modules, architecture showing more than one solution for a problem, high cohesion between modules, higher layers accessing functionalities of lower layer without any requirement.

In this paper we have evaluated and tested our approach on few examples of software architecture for reducing the design rot and increasing the life of design.

Suppose consider a simple example of cyclic dependency between packages containing the related group of classes in the given architecture. Fig. 8 shows an acyclic package network.

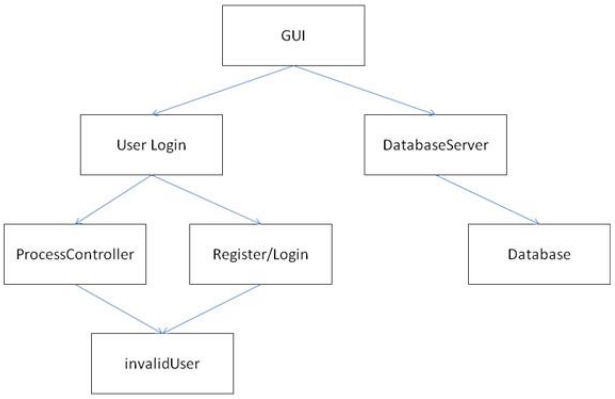

Fig. 8 Acyclic Package Network

Fig. 8 shows no cycles as the GUI depends directly on the package User Login and transfers data to Database Server package. The Register/Login package has no other dependencies as no other package is required therefore the testing and release can be done with nominal amount of work.

Addition of a Cycle in Package Network- when a designer wants to display a message on screen from invalid User Package so it is send up to GUI as message display is controlled by it resulting in a dependency of invalid User on GUI.

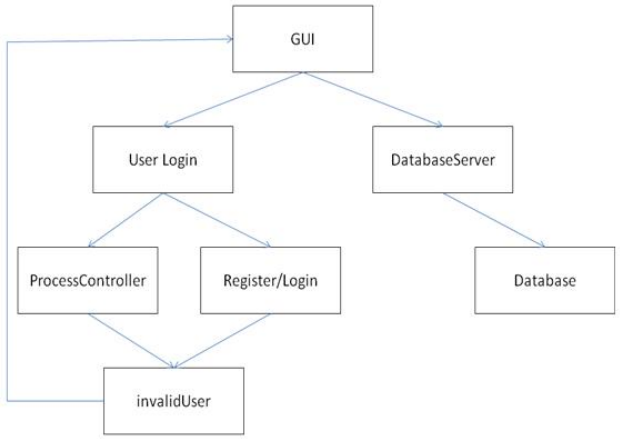

Fig. 9 A cycle sneaks in

As the cycle is introduced between package so any person functioning with Login/Register want to release this package, a test suite is required to be built with all other packages in the architecture such as invalid User, GUI, User Login etc this becomes obviously a terrible situation resulting in a considerable increase in the workload of developers and architects just because of a particular uncontrolled dependency.

This cyclic dependency should be removed from the structure to fight architecture erosion. Refactoring can be applied for breaking the cycle in this scenario; for this, a new package should be added, this change in the architecture should be maintained without any defect. For adding any new feature in the system it should possess Maintenance quality.

Maintenance means architecture design can be continuously modified without introducing any bugs in the system, it allows addition of a new package in the architecture, the classes contained in invalid User package are now placed in that new package which is Message Manager. This change in the architecture can only be welcomed if the system possesses the development quality method Evolvability.

Evolvability in the given architecture means that the system will easily adjust to the new addition in the architecture and remain in working condition for longer run. In short, the system would embrace the change instead of avoiding it.

Management Support is very important for supporting this change in the architecture as an undocumented initial architecture of the system would definitely results in architecture erosion as the architects would never get an idea which module is the basis of cyclic dependency and how it would be resolved.

Now the two packages the invalid User and GUI depends upon this new package.

Hence, this example clarifies that for removing the cycles and breaking dependencies new package should be introduced in the architecture and the classes are to be moved from old package to new packages thus changing the package architecture. The final structure is shown in the following diagram.

The example also shows that the architectural strength of any software depends entirely on these four methods, absence of any one will escort the architecture design towards erosion as refactoring techniques can never be applied to an architecture which is not maintainable and for a long run maintainability of a software it should be evolvable. Correspondingly, no change can be brought to the system without Management Support.

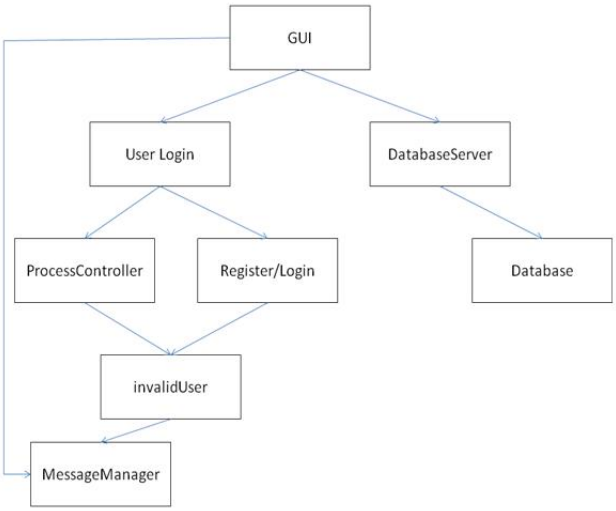

Fig. 10 Architecture after removal of dependency

The result of applying the proposed approach on some other different architectural styles is summarized in the Table I. The Results in the given table shows that the consequence of excluding any one of the proposed method in the devised approach indeed leads to the software architecture erosion.

TABLE I
CAUSES OF SOFTWARE ARCHITECTURE EROSION

Possible Solutions To Architecture Smells	Proposed Approach				Results
	Management Support (Applied? YES/NO)	Maintenance (Applied? YES/NO)	Evolvability (Applied? YES/NO)	Refactoring (Applied? YES/NO)	
Decoupling Layer	YES	YES	NO	YES	System will not adjust to change due to lack of evolvibility, results in shorter life of system
Renaming Entities	NO	YES	YES	YES	Unavailability of initial architecture document results and lack of management support in bringing the change, results in architecture degradation.
Merging Modules	YES	NO	YES	YES	Lack of maintenance results in producing errors in system brought up by the change
Stable Layering Establishment	YES	YES	YES	NO	No refactoring means No change is made in the system to establish stable layering for avoidance of relaxed layering.

IV. CONCLUSION

In this paper, different problems resulting from different types of architectural erosion are presented that leads to high cost, time issues or even failure of costly projects. After deeply analyzing the types, symptoms and risks of software architecture erosion, an optimal approach known as “a four method framework” is devised. By following the proposed approach in designing architectures; one would be able to fight erosion to a greater extent gaining architectural strength in software systems. Management support, Maintainability, Evolvability and Refactoring are the key practices for the achievement of architecture improvement, if all of them should be emphasized orderly. Management support is a very important factor and is needed for the planning and management of the rest of three methods. Architecture smells themselves are the indicators that an effective approach is needed for fighting the nearly occurring architecture erosion in a cost-effective way for getting the best overall effect.

Finally, this paper has indicated the most suitable approach for reducing the effect of erosion by considering that for bringing up any change in the system through refactoring; management support, maintenance and evolvability play a significant role when applied collectively to the architecture but the tools used for removing architecture erosion are still in development stage. Most significant case studies and improved evaluations of the available tools are required so that practitioners can assess different results and adopt the most suitable ones to their circumstances.

REFERENCES

[1] “Chapter 1: What is Software Architecture” Msdn library, Retrieved from <http://msdn.microsoft.com/en-us/library/ee658098.aspx>
[2] Taylor, Richard N., Medvidovic, Nenad, & Dashofy, Eric. (2009). Software architecture: foundations, theory and practice. John Wiley & Sons Inc.
[3] Mehwish Riaz, Muhammad Sulaman, Husnain Naqvi, “Architectural Decay during continuous software evolution and impact of ‘Design for Change’ on software Architecture” published in Journal of Advances in Software Engineering and Communications in Computer and Information Science, 2009, Volume 59, 119-126.
[4] M M Lehman, J F Ramil, P D Wernick, D E Perry, W M Turski, "Metrics and laws of Software Evolution The Nineties View,"metrics,p.f14, Fourth International Software Metrics Symposium(METRICS'97),1997.
[5] John Reekie, Rohan McAdam, "A Software Architecture Primer Paperback." (2009)
[6] Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture, OOPSLA. (2005)
[7] "Dead code detection and removal." Aivosto-Programming tools for Software Developers. Aivosto Oy, Helsinki, Finland, n.d. Web. <http://www.aivosto.com/vbtips/deadcode.html>.
[8] "Code duplication detection." SolidSource- About Software Development and Maintenance. N.p., 17-Feb-2010. Web. <http://www.solidsourceit.com/blog/?tag=code-clone>.
[9] "Software Rot - Manage those Dependencies." kirkk.com. N.p., 06-APR-2009. Web. <http://techdistrict.kirkk.com/2009/04/06/software-rot-manage-those-dependencies/>.
[10] Martin, Robert C.: Design Principles and Design Patterns.
[11] Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy. "Software Architecture: Foundations, Theory and Practice."(2009)
[12] "Stop the Architecture Erosion of Eclipse And Open Source Projects at EclipseCon 2011." Anthony Dahanne’s blog. N.p., 24-MAR-2011 < http://blog.dahanne.net/2011/03/24/stop-the-architecture-erosion-of-eclipse-and-open-source-projects-at-eclipsecon-2011/>.
[13] Knoernschild, Kirk. "That Rotting Design." kirkk.com. N.p., 21-DEC-2009. Web. <http://techdistrict.kirkk.com/2009/12/21/that-rotting-design/>.
[14] Medvidovic, Nenad, and Vladimir Jakobac. "Using software evolution to focus architectural recovery." Autom Software Eng (2006) 13: 225–256. Springer Science.
[15] Dr. Eden, Amnon H. "Software Evolution and Validation." (2008).
[16] "Bad Software Architecture." 0xcafebabe's space. N.p., 22-MAR-2010. Web. <http://huionn.wordpress.com/2010/03/22/bad-software-architecture/>.
[17] Stal, Michael. "Software Architecture Refactoring." Siemens AG Corporate Technology, 2008.
[18] Bode, Stephan: “On the Role of Evolvability for Architectural Design”, 16-AUG-2010.
[19] IEEE Std. 1219-1998, IEEE Standard for Software Maintenance. IEEE Computer Society.
[20] Caroli, Paulo. "Refactoring to Patterns- A practical look into Agile approach on Evolutionary Design." IndicThreads.com Conference on Java Technology 2007, 07-FEB-2008.