

**IDENTIFYING SOFTWARE ARCHITECTURE
EROSION THROUGH CODE COMMENTS**

Vidudaya Neranjan Bandara

(168206G)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2018

IDENTIFYING SOFTWARE ARCHITECTURE EROSION THROUGH CODE COMMENTS

Herath Mudiyanseelage Vidudaya Neranjan Bandara

(168206G)

Thesis submitted in partial fulfillment of the requirements for the
degree Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2018

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books)

Signature:

Date:

H.M.Vidudaya Neranjan Bandara

The above candidate has carried out research for the Masters thesis under my supervision.

Name of the supervisor: Dr. Indika Perera

Signature of the supervisor:

Date:

Abstract

Software architecture erosion or the as-implemented architecture is not complying with the as-intended architecture is one of the major problems faced by many organizations. There is no easy way to trace design decisions or tracking back or reconstructing those decisions by looking at the source code level elements is one of the major reasons for software architecture erosion. Other than that the mistakes or carelessness of the programmer may lead the system to an eroded status eventually. Lack of domain knowledge, lack of knowledge about intended architecture and unable to identify possible violations of as-intended architecture (by identifying architectural degradation) are some other reasons for software architecture erosion.

There are various methodologies and tools for architecture conformance checking and analyzing the static architecture and provide comparison results which can be used to determine whether the architecture of a system is altered or not [10]. Most of them require high end tool support and providing the implemented architecture and the intended architecture each time the analysis needs to done.

As the main research objective it identified a missing area of software architecture conformance checking methodologies and analyzed and identified a way to prevent software architecture erosion using that. This research is more focused on unconventional usability of the code comments and how it can be leveraged to capture the architecture of the application and how it can be used as an effective architecture conformance checking mechanism.

This research states a methodology which uses Java Doc comments to inject architecture specific information into the code base and a mechanism to capture them and compare them with a pre-defined architecture rule set. An empirical and theoretical evaluation has been done to prove this concept actually works in real life scenarios. It opened up a new area of architecture conformance checking to the future researchers of the field of software architecture.

ACKNOWLEDGEMENT

My sincere appreciation goes to my family for the continuous support and motivation given to make this thesis a success. I also express my heartfelt gratitude to the research supervisor Dr. Indika Perera, for the supervision, advice and continues guidance given throughout to make this research a success. Also I am grateful for the support and advice given by Dr. Malaka Walpola, by encouraging continuing this research.

My special thanks go to Mr. Umesh Indith Liyanage and Dr. Rasika Withanawasam for sharing their knowledge regarding software architecture and software architecture conformance.

Finally I wish to thank the academic and nonacademic staff of Department of Computer Science and Engineering and colleagues of MSC'16 for the support and encouragement given.

TABLE OF CONTENTS

DECLARATION	i
Abstract	ii
ACKNOWLEDGEMENT	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	vii
LIST OF TABLES	ix
LIST OF ABBREVIATIONS	x
Chapter 1 INTRODUCTION	1
1.1 Background	2
1.2 Software Architecture and Software Engineering	3
1.3 Problem Statement	3
1.4 Motivations to Solve the Problem	4
1.5 Research Objectives	5
1.6 Overview of the Document	6
Chapter 2 LITERATURE REVIEW	7
2.1 Software Architecture Erosion and Industry Software	8
2.2 Industrial Examples of Design and Architecture Erosion	8
2.3 Causes of Erosion	9
2.4 Effects of Architecture Design Erosion on a System	11
2.5 When to Decide the Software Is Eroding and Needs to Be Repaired	12
2.6 What Kind of Solutions Are Applied To Fix an Eroded System	12
2.7 Architecture Reconstruction Techniques	13
2.8 Architecture Refactoring	14
2.9 Preventing Architecture Erosion	14
2.10 Preventing Software Architecture Erosion through Static Architecture Conformance Checking	15
2.11 GRASP ADL Based Static Architecture Conformance Checking Tool for Java	17

2.12	Reflexion Modeling and Inverting Reflexion for Design Control (An Industrial Case Study of Architecture Conformance).....	19
2.12.1	Reflexion Modeling	20
2.12.2	Inverting Reflexion for Design Control	21
2.13	Traceability Model For Viewing Architectural Tactics Using Code Comments	22
Chapter 3	METHODOLOGY.....	26
3.1	Identifying a Possible Architecture Capturing Mechanism	27
3.2	Identifying Possible Architecture Violation Types to Consider For a Proof of Concept	28
3.3	Developing a Proof of Concept	29
3.4	Evaluation of the Proof of Concept	29
Chapter 4	SOLUTION ARCHITECTURE AND IMPLEMENTATION.....	30
4.1	Presentation of Rules for Style Invariants and Prescriptive Architecture ...	31
4.1.1	Presenting the Rules via XML	31
4.1.2	Presenting the Rules via JSON	32
4.2	Mapping Source Elements to Architectural Information	33
4.2.1	Mapping Elements Using Java Annotations	34
4.2.2	Mapping Elements Using JavaDoc Tags	35
4.2.3	Comparison of Java Annotations and JavaDoc.....	36
4.3	Solution Architecture	37
4.3.1	Validating System Architecture Using JavaDoc Tags in the Source Code	38
4.3.2	How Descriptive Architecture Information Is Identified	39
4.4	Prototype Implementation	41
4.5	Generating Views for the Reports	43
4.6	Style Invariants Checker Added As a Separate Dependency through Maven	43
Chapter 5	EVALUATION.....	49
5.1	Empirical Evaluation the Correctness Of the SIC.....	50
5.1.1	Evaluate the Correctness of the Loaded Prescriptive Architecture by SIC	51
5.1.2	Evaluation of the Style Invariants Checker Validation Components.....	52

5.1.3 Evaluation of the Success Path	53
5.1.4 Evaluation of the Failure Paths	54
5.2 Performance Testing of SIC	57
5.2.1 Performance Testing By the Complexity of the Prescriptive Architecture Rule Set	57
5.2.2 Performance Testing By the Size of the Code Base	59
5.3 Analytical Evaluation Of The Impact Of SIC When It Is Added To A Continuous Integration Flow	62
Chapter 6 CONCLUSION	64
6.1 Research Contribution	65
6.2 Research Limitations	66
6.3 Future work and Conclusion	66
REFERENCES.....	67
APPENDIX A	70
APPENDIX B	72

LIST OF FIGURES

Figure 2-1 : Simple GRASP Specification	16
Figure 2-2 : Design of the GRASP based static conformance checking tool	18
Figure 2-3 : GRASP specification containing mapping information.....	19
Figure 2-4 : The inverted Reflexion Modeling process	21
Figure 2-5 : XML based design pattern specification for Adapter pattern	23
Figure 2-6 : JSON based design pattern specification for Adapter pattern.....	23
Figure 2-7 : Minimal XSD needed to represent design pattern information	24
Figure 2-8 : Using Java annotations to map design pattern specifications	24
Figure 3-1 : Typical code review process with a reviewer other than the developer itself.....	27
Figure 3-2 : Development process after the introduction of the solution	29
Figure 4-1 : Representing simple layered architecture invariants and details about the architecture using XML	32
Figure 4-2 : Representing simple layered architecture invariants and details about the architecture using JSON.....	33
Figure 4-3 : Use Java annotations to present the details of the architecture and other information like style invariant	34
Figure 4-4 : Use JavaDoc tags to present the details of the architecture and other information like style invariant.	35
Figure 4-5 : Solution Architecture	38
Figure 4-6 : Folder and package structure.....	41
Figure 4-7 : Simple_layer_access.json.....	42
Figure 4-8 : The basic Mojo class for the Maven plug-in.....	44
Figure 4-9 : Pom.xml configuration for the plug-in.....	45
Figure 4-10 : How to include the plug-in to an application through maven plug-in configuration	46
Figure 4-11 : Run the scan using the SIC plug-in.....	47
Figure 5-1 : Basic flow of Style Invariants Checker.....	51
Figure 5-2 : Basic components of Style Invariant Checker	51
Figure 5-3 : Prescriptive Architecture rules can be presented in a JSON file format	52
Figure 5-4: Application's system structure.....	54
Figure 5-5 : Code section from a layer, access a functionality of another layer which is not allowed to access directly	55
Figure 5-6 : Results of the SIC execution when a code section from a layer, access a functionality of another layer which is not allowed to access directly	55
Figure 5-7 : Average execution time of SIC with the complexity of the prescriptive architecture rules	58

Figure 5-8 : Case 1, Change of the Average Execution Time with the size of the code base.....	60
Figure 5-9 : Case 2, Change of the Average Execution Time with the size of the code base.....	61
Figure 5-10 : Case 3, Change of the Average Execution Time with the size of the code base	62
Figure 5-11 : Continues Integration flow before the introduction of Style Invariants Checker	63
Figure 5-12 : Continues Integration flow after the introduction of Style Invariants Checker	63

LIST OF TABLES

Table 4-1 : Using Annotations vs JavaDoc tags	36
Table 5-1: Loaded prescriptive architecture validation test results	52
Table 5-2 : Expected and Actual results when executing the SIC on applications with different violations	56
Table 5-3 : Average execution time of SIC with the complexity of the prescriptive architecture rules	58
Table 5-4 : Case 1, Change of the Average Execution Time with the size of the code base.....	60
Table 5-5: Case 2, Change of the Average Execution Time with the size of the code base.....	60
Table 5-6 : Case 3, Change of the Average Execution Time with the size of the code base.....	61

LIST OF ABBREVIATIONS

Abbreviation	Description
IDE	Integrated Development Environment
ARM	Architecture Reconstruction Method
ADDRA	Architectural Design Decision Recovery Approach
ADL	Architecture Description Languages
RM	Reflexion Model
HLM	High Level Model
SM	Source Model
SIC	Style Invariants Checker

Chapter 1
INTRODUCTION

1.1 Background

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [1]. There are many benefits obtained by the software architecture. It is used to present a common abstraction of the system, and it will provide a basis for communication among various stakeholders. Architecture of the software captures the earliest design decisions which continue to have profound implications on remaining development, deployment and maintenance phases of the system [2].

Software architecture can be seen in two different views. Prescriptive Architecture and Descriptive Architecture. A system's prescriptive architecture captures the design decisions made prior to the system's construction while a descriptive architecture describes how the system has been built, it is the as-implemented or as-realized architecture.

When a software system is initially built there may not be a huge difference between the prescriptive and the descriptive architectures. That means the system is implemented according to the intended architecture, but there will be large number of prescriptive and descriptive architectures created during the lifespan of the project due to various reasons. When the already implemented system is evolved, its prescriptive architecture is modified appropriately and it will lead to do the corresponding changes to its descriptive architecture. In many cases the system is often directly modified without analyzing the impact to the prescriptive architecture.

With the time there will be a huge notable difference between the prescriptive architecture and the descriptive architecture. This failure to update the prescriptive architecture will results in potential dangers. The resulting discrepancy between a system's prescriptive and descriptive architecture is referred to as architectural degradation.

The architectural degradation comprises of two related phenomena, Architectural Drift and Architectural Erosion. Drift in the architecture happens when introducing principal design decisions into a systems descriptive architecture that are not included in, encompassed by or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture design decisions. Erosion happens when introducing architectural design decisions into a systems descriptive architecture that violates its prescriptive architecture.

From the above two, architectural erosion is more dangerous than the drift. Because sometimes a drift may be intentional or can be corrected with low cost, but the

erosion may not be identified until the end of a major code release and the only viable solution left may be to rewrite the entire code base from the scratch.

Throughout the last few decades many researches were conducted and many methodologies and tools were created in order to mitigate architecture erosion. Removing the architecture erosion completely is still an unsolved problem in the field of software architecture.

1.2 Software Architecture and Software Engineering

As per the book Rational Unified Process: An Introduction, by Philippe Kruchten, An architecture is the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides this organization - these elements and their interfaces, their collaborations, and their composition. So architecture of a software intensive system is not just software. It is a combination of software, hardware and human resources. The performance related design decisions, usability / HCI (human computer interaction) related design decisions, security related design decision and many more are included in a software architecture. In one way software engineering is the process of making sure that the software is made to its architecture correctly.

Software engineering is a study of engineering to the design, development and maintenance of a software product. At the initial stage of the software industry there were lots of issues related to low-quality software products. Then there were problems regarding timeliness, budgets and reduces level of quality and correctness. Most of the software products faced the challenge of constantly changing customer requirements. The demand for this specific area of engineering called 'Software Engineering' emerged to cater those requirements and as a solution for those above mentioned issues.

This research can be considered as a study related to a common problem occurred in between the software architecture and software engineering.

1.3 Problem Statement

Software architecture erosion can be a result of many bad practices and mistakes. Most of the software development organizations hire programmers to maintain a system or develop a system. Those programmers will start to change the code and

then the system itself without knowing that they may violate the intended software architecture although the tasks that assigned to them are completed.

Few major reasons for software architecture erosion are,

- Developers lack domain knowledge
- Developers lack knowledge about intended architecture
- The organizations best practices are not followed correctly
- Hard to notice that the changes done by a developer may lead to architectural degradation

The problem this research tried to focus comprise as follows.

There is no easy way to identify that the ready to commit code contains elements that may lead the system to an eroded state other than carefully reviewing it or using tools which are costly and do not guarantee 100% results.

1.4 Motivations to Solve the Problem

Finding a viable solution to the above stated problem and by doing that help the software developers to write a better code that protects the intended architecture of the system is the main motivation of this research.

The solution includes a tool which compares the intended architectural information with the descriptive architectural information of a system at any given time. Basically that tool will be able to,

- Identify the possible architectural violations and notify them to the developer
- Integrate with a continuous integration mechanism to be a part of a deployment cycle
- Integrate with a build management tool like Maven

Though there are many researches going on in this area of the field there are so much left to discover. With the exponential growth of the technologies and the industries, new problems and new perspectives of solving the same problem come up.

There are existing methodologies and tools which try to solve the above stated problem, with some limitations,

- They require the code base to be in a standard pattern
- They might add extra code changes other than the actual business logic
- Cost is high

- Customizing the tool is not easy

This research motivated to focus on mitigating the above mentioned limitations and came up with a solution that actually does not have those limitations.

1.5 Research Objectives

The main research objective is to find a solution to the above stated problem while mitigating the limitations of the existing methodologies and tools. To achieve that major objective there are several sub objectives fallen in line. Throughout the research period they were successfully achieved.

First objective was to survey and analyze the developer perspectives about the research idea. This was necessary because to mitigate the limitations of the existing tools and methodologies first it needed to be examined properly. The best way to do so was to get the inputs from the actual developers who use them.

The second objective was to survey the relevant research literature in this research area. This helped the research to identify the missing areas of software architecture conformance checking and provide solutions to them. Understanding the existing solutions for the same problem and identifying how to optimize them was another plus point of the literature survey.

Third objective was to define a methodology to solve the problem and identify a suitable tech stack to implement the solution. This was followed by the next objective to implement the actual tool that used as a proof of the underline concept.

As the final objective a comprehensive evaluation has been done in both theoretically and euphorically in order to prove the concept actually works.

In this research above mentioned sub objectives were successfully achieved and they lead to a successful research outcome, which is the tool named SIC (Style Invariants Checker).

1.6 Overview of the Document

This document consists of six chapters. The first chapter comprised the introduction to the research by presenting a background to the underline domain of software architecture and software architecture erosion. It will present a brief overview to the problem which the research trying to find a solution. The motivations to solve the stated problem and the objectives and milestones set to achieve that objectives are also listed in the introduction chapter.

The second chapter comprised the findings of the related literature. It includes descriptions regarding the software architecture erosion and the impact to the industry level software. The causes of erosion and what kind of solutions available to prevent it is also focused on this chapter. Findings of this chapter helped this research to identify the potential methodologies and areas that are not yet considered to solve the problem of architecture erosion.

The third chapter discuss about the identified methodology to solve the problem of software architecture erosion. It will comprise of the possible architecture capturing mechanisms identified and possible scope for this research. The methodology also talks about the mechanism to develop a proof of concept to the identified solution.

Fourth chapter comprised the information regarding the solution architecture and the implementation of the proof of concept. This chapter also contain in depth details about finding a solution to the problem of software architecture erosion using developer code comments. At the end of this chapter there is a section that describes the prototype implementation to the tool Style Invariant Checker.

Fifth chapter has the information regarding the evaluation of the research methodology which includes empirical and theoretical evaluation followed by the performance testing. Sixth chapter conclude the research by stating the research contributions along with the limitations of the research.

Chapter 2
LITERATURE REVIEW

2.1 Software Architecture Erosion and Industry Software

With the massively increasing complexity and the size of software, the existing software development methods and tools beginning to be weak and less useful. This is practically a true statement when it comes to maintaining software [3]. In the Design Erosion : Problems and Causes paper they are illustrating that despite thirty decades of research and despite the many suggested approaches it is still inevitable that a software system eventually erodes under pressure of the ever changing requirements.

2.2 Industrial Examples of Design and Architecture Erosion

In the software industry design erosion is quite common and the naive solution for that is redeveloping the software system from the scratch. Almost in all the cases the redevelopment effort was high.

An example of a project for this scenario is the famous Mozilla web browser. After experiencing a fierce competition from the Microsoft's Internet Explorer Netscape decided to release their own browser as open source. Nearly after 6 months the developers of open source Netscape came to the conclusion that the original netscape source code was eroded beyond repair. Obviously the decision was to start from the scratch. Yet during two years of redevelopment, requirements had changed sufficiently to retire a part of the system before the system was even finished [3].

Another example is the Linux kernel. Among the many reasons why it took nearly two years to develop kernel 2.4 after the previous stable release 2.2 is that the old 2.2 code needed massive restructuring in order to incorporate the new designs and requirements.

In those scenarios the preferred solution was to redevelop or restructure the source code base. Though it was a successful approach a massive effort needed to be put on in identifying the signs of design erosion early enough to be able to take such action. Redeveloping software is a time consuming and a very expensive procedure, and failing to get the right decision at the right time may cost an organization a fortune.

2.3 Causes of Erosion

Software architecture erosion can be caused by a number of problems associated with the way the software is commonly developed.

- Traceability of design decisions

It is very important to know the prescriptive architecture of the system in order to change the system when dealing with new requirements. Problems can occur when the notations commonly used to create software lack the expressiveness needed to express concepts used during design. There are no or lack of proper documentation about the implemented functionality or the design, which eventually leads to making guesses about the system.

- Maintenance cost is increasing

During the software evolution the maintenance task becomes increasingly effort consuming due to the fact that the complexity of the system keeps growing. Those task can be both time consuming and costly. This may eventually cause the developers to take suboptimal design decisions either because they do not understand the architecture or because a more optimal decision would be too effort consuming

- Accumulation of design decisions

Due to the hierarchical nature of design decisions high level architectural decisions are followed by many low level architectural design decisions. The design decisions are accumulated and interact in a way such that revision of one would force reconsideration of all of the others. When a programmer decide to change a design for any reason, then they must consider the system as a whole and take a optimal strategic decision which eventually consider all other decisions affected by that or they must work with a system design which is not going to be optimal

- Iterative methods

A primary goal of the system architecture design is to create a design that can accommodate future changes to the system easily. This conflicts with the iterative nature of many software development methods (ex: in agile) since these methodologies typically incorporate new requirements that may have an architectural impact, during development where a proper design requires knowledge about these requirements in advance

- Lack of continuous refactoring

Refactoring should be done regularly, if not then small design or implementation issues, architectural smells or decision inconsistencies will be accumulated, and consequently the software qualities will degrade

- Uncertainty about the evolution of the system

Most of the times when the creation of prescriptive architecture takes place the designers are uncertain about the possible future goals of the system. What are the future extensions, the possible future integrations and migrations are not visible during the primary phases due to various reasons. This may lead the prescriptive architecture and hence the descriptive architecture hard to maintain

- Release pressure

With the increasing number of change requests and the tight schedules the developers are forced to complete the tasks assigned to the as soon as possible. Though the task completed without a problem and passed the user acceptance tests that doesn't mean the fix or the change made to the system was the optimal one. The developer may be unintentionally change the system architecture and cause the system to early eroded state

- Changing requirements

This cannot be stopped. Requirements are in their nature are subjected to change. What we can do is to build the system so that it could withstand the changes in the future. Knowing the possible changes or the possible requirements can help the designers to do a better job

- Lack of knowledge about early design decisions

This happens due to both lack of documentation and staff turnover. If the code base is not self explanatory (when the system is growing cannot expect it to be self explanatory always) then there should be a proper documentation or the developers needs to be in touch with the production and maintenance. If not it's hard to understand and maintain the intended purpose of the system

2.4 Effects of Architecture Design Erosion on a System

When a software system getting large and getting mature with the time it will tend to degrade from the original architecture and erosion will eventually happen. Identifying the software erosion with the time as early as possible will help to recover from it or delay it. There are symptoms of a deteriorating system [4]

- Code quality

Quality problems of a source code may include unnecessarily complex or lengthy functions, abuse of language features, wrong use of infrastructure features etc. When an experienced developer feels that the code is too complex for its intended purpose or there are so many boilerplate codes here and there it might be too late to recover it. Sometimes a well developed code base may have violated a major design decision with its latest change. So with a review it can be identified as early as possible and take actions to solve the problem

- Uncertainty about specifications

Most of the times undocumented changes added to the system effectively making the existing design specifications obsolete. When it feels like there is a great deal of uncertainty about the system specification and the architecture design it might be a good time to take time to resolve those problems and after that move forward

- Regressions

Fixes for defects often introduce new problems. Some of them are visible or produced new bugs as soon as the new deployment goes and some of them will remain few years to show symptoms

- Deployment problems

Since the original design of the system was developed aiming to cater a certain set of deployment configuration steps it might not hold on with a new and changed set of deployment steps. This changing of the environments keeps happening and we cannot stop that. These deployment problems can be identified when the system shows symptoms regarding deployment issues

2.5 When to Decide the Software Is Eroding and Needs to Be Repaired

After identifying the symptoms of software architecture erosion sometimes it will be kept as a low priority task to do the repairs or take the necessary actions to resolve the problem due to various reasons like cost limitations and time limitations. Sooner or later it needs to be done [14]. There are some tipping points to consider taking that decision.

- By analyzing defect densities

Defect figures in a particular software component, which are larger than the norm, automatically trigger management to initiate proper action. Quick bug fixes and less number of reviews are the major reason for the increased defect density. Those reasons are a clear indication of eroding software

- By conducting proper system evaluations

Most of the times the decision to evolve the software was taken after an internal evaluation. If there are problems with the existing software and if it feels like the software is not in a good condition to evolve, then a proper adjustment should be made

- When the new requirements arrives

If the system is not in a shape to cater the new requirements then probably the system is no longer useful. Might be a case where the software erosion is in place

2.6 What Kind of Solutions Are Applied To Fix an Eroded System

After it has been determined that a system is eroded and the possible causes have been identified we can attempt to make the repairs and prevent future damage to the system. As identified there are few obvious things that can be done [4]

- Redevelopment of the software

If the software is in a state that there are no options to repair it to a workable system then we might have to redevelop it from the scratch. Because changing the existing system might not end up giving the best

possible outcome, sometimes it might not satisfy the minimum requirements of the system

- Restructuring

There might be a case where a restructuring of the system is enough to make the erosion delay. If so it might be suitable for certain scenarios

- Strong focus on design

As identified before lack of up to date design and specifications is usually one of the problems with eroded systems. In scenarios like that recovering and updating the designs is an integral part of the attempt to address the problems and a key to the success of the whole operation

- Modularization and object orientation

There might be a case where the source code is in a bad shape and that there are many dependencies between the various modules and components in the system. In both cases object oriented type mechanisms such as encapsulation, information hiding and delegation can be applied to improve the structure of the system

- Reverse-engineer the system and recover the architecture from the source code

Sometimes the prescriptive architecture is no longer available and in such a case it will be much harder to recover from it. If there is a possibility to reverse engineer the source code and find the original architecture then there is a chance of recovery

2.7 Architecture Reconstruction Techniques

There are different tools and techniques for reconstructing architecture from source code or runtime artifacts. The Architecture Reconstruction Method (ARM) [5] is one of the first semi-automatic methods for architecture recovery from source code. This method is based on pattern matching idea to identify a set of patterns provided by the user in a reverse engineered model of the implemented architecture.

Reflexion models by Murphy et.al [6] is used to map a hypothetical model of the intended architecture to the results from a static analysis of the source code. A clustering technique (Classes and packages as the elements of the clustering

technique) to refine the reflexion models and create a high level abstraction of the system architecture was done by Lung et.al [7]. Pattern matching language called Architecture Query Language was used by Sartipi [8] to develop a software architecture recovery method. Jansen et.al [9] emphasizes recovering architectural design decisions based on differences in architecture design across different versions of the system. This method is called Architectural Design Decision Recovery Approach (ADDRA) and it does not focus on the structure of the software. As the first phase it will generate an architectural view for each version of the system. Then the differences are inspected to identify the architecture changes and the various design decisions.

2.8 Architecture Refactoring

This is used to reconcile the prescriptive and descriptive architectures of a system. Architecture refactoring is often done to improve design fragments of software architectures which can have a negative impact on system maintainability. Because some requirement change and that might results in the adoption of a design solution which is inappropriate for that context, or the new solution might result in undesirable behavior. In such situation, refactoring the architecture is necessary to remove the issues known as architectural smells and prevent their accumulation which may end in design erosion. Various tools and IDE's have been developed to support code refactoring but almost all of them do not emphasize the architectural level concerns. It is kept as a developer responsibility to deal with the issues with the architecture.

2.9 Preventing Architecture Erosion

As many suggested the first step towards preventing design erosion is having a more mature software development process. As identified by Bengtsson et-al [4] the most important factor in such a development process which can prevent design erosion is proper documentation of the architectural design decisions. This documentation should be accessible to all the developers and stakeholders and it should give them up to date knowledge about the architecture of the system.

"Even when the code is designed so that changes can be carried out efficiently, the design principles and design decisions are often not recorded in a form that is useful to future maintainers. Documentation is the aspect of software engineering most neglected by both academic researchers and practitioners. It is common to hear a

programmer saying that the code is its own documentation. Even highly respected language researchers take this position, arguing that if you use their latest language, the structure will be explicit and obvious." - Parnas

Not only the prescriptive architecture, the requirements itself should be documented as well. Len Bass [11] prescribes a detailed documentation of requirements; it has qualities in form of a precise description template which is ready for rigorous analysis, as well as documenting the design architecture through different architectural views suitable for different stakeholders. [12]

The prescriptive architecture should be clearly visible and understandable to all the developers and architects of the system. New members and maintainers should be properly introduced to the intended architecture so that they won't break it unintentionally. The source code should be reviewed by the senior developers as well as the architects who know the intended architecture.

There are researches about considering the linkage between architecture and implementation. It might provide a basis for monitoring architectural compliance at any time. The reflexion models technique developed by Murphy et al [6] which compares a reconstructed model of the implemented architecture to a hypothetical model of the design intended by the architects. Then the two models can be analyzed to find possible deviation between intended and implemented architectures. There is a research and a survey on extending this idea to provide a tool support for mapping and deviation analysis [13]. It assumes that the intended architecture exists, then repeatedly refines the implemented architecture as development progresses, and compares it against the intended architecture.

2.10 Preventing Software Architecture Erosion through Static Architecture Conformance Checking

There are researches that are trying to assess the conformity of the implemented architecture to the intended architecture and can provide a strategy for detecting software architecture erosion and thereby prevent its negative consequences.

When considering the current state-of-the-art of software architecture research and popular industry practices on architecture erosion, it obviously appears that such solution strategy is much needed to address the ever increasing demands for large scale complex software systems [12].

As discussed earlier architecture erosion controlling approaches can be divided into three broad categories. Namely minimizing the architecture erosion, preventing the architecture erosion and repair an eroded system [13]. We are talking about prevention approach when it comes to detecting the architecture erosion and try to avoid erosion. Static architecture conformance checking belongs to that category as well.

Static code analysis does not require the application to be running. It can be seen as a methodology which can provide very quick and early feedback to developers. [2]. In order to perform architecture conformance checking we need to have the architecture of the system in a formal manner or according to a standard format. So that the analyzing or processing in a programmatic way is easily possible. Architecture Description Languages (ADL), which try to provide unambiguous definition of the systems architecture, are formal languages that can be used to represent architecture of a software intensive system [15] .

GRASP [16] [17] is a textual architecture description language capable of capturing the ‘rationale’ behind architectural design decisions. It is somewhat similar to a high level programming language syntax or a pseudo code. GRASP can parse a given set of specification and create an object model from it. In any GRASP specification ‘architecture’ element is the top most one. As described in the Figure 1, ‘layer’ element will represent the layered architecture and ‘because’ key word will link each architectural element to relevant rationale.

```
architecture Simple
{
  rationale R1() {
    reason #'Use layered architecture style';
    reason #'Achieve clear separation of
      concerns';
  }
  system SimpleSystem
  {
    layer Utilities because R1()
    { }
    layer Simulator over Utilities because R1()
    { }
    layer UserInterface over Simulator,
    Utilities because R1()
    { }
  }
}
```

Figure 2-1 : Simple GRASP Specification

Once the as intended architecture captured in a manner which supports programmatic processing of the architecture specification, needs to define a mapping, which contains as to how architectural elements map to implementation units in the actual source code. Static architecture conformance checking is mostly concerned with module view of the architecture, which deals with the organization of the source code elements. Once the architecture and the mapping are in place actual source code has to be analyzed. To do that one can build a model of the source code and then evaluate rules related to architecture elements against this model. Another approach will be to reverse engineer an architectural model using the mapping information and then compare this with the original architectural specification to find deviations. Since to do static architecture conformance the application doesn't needs to be in execution, the above can be done easily.

2.11 GRASP ADL Based Static Architecture Conformance Checking Tool for Java

As researchers say, given that GRASP is a textual ADL is would be easy to integrate a static architecture conformance checking tool based on GRASP, with existing build, continuous integration tools and frameworks [14].

The approach the suggested first need to have the information on what Java classes a particular GRASP architecture element maps to. Once all the mapping of architectural elements are known, it is possible to check whether associations between Java classes are valid compared to the associations between architectural elements.

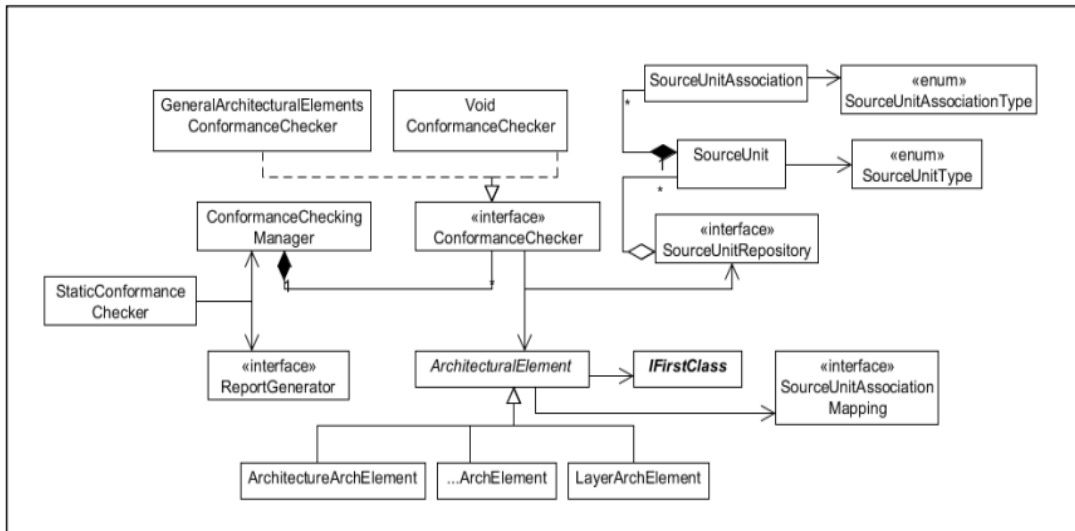


Figure 2-2 : Design of the GRASP based static conformance checking tool

They have used annotation support in GRASP for mapping architectural elements to Java classes. Figure shows example of an annotated GRASP specification. There the mapping information contains which Java classes a given architectural element maps to and which associations of those classes needs to be considered during conformity checking.

```

@conformance(include="example.layer.**")
@conformance(excludeAssociations="java.**")
architecture Simple
{
  rationale R1() {
    reason #'Use layered architecture style';
    reason #'Achieve clear separation of concerns';
  }
  system SimpleSystem
  {
    @conformance(include=
      "example.layer.utilities.**")
    layer Utilities because R1()
    { }
    @conformance(include=
      "example.layer.simulator.**")
    layer Simulator over Utilities because R1()
    { }
    @conformance(include=
      "example.layer.userinterface.**")
    layer UserInterface over Simulator, Utilities
      because R1()
    { }
  }
}

```

Figure 2-3 : GRASP specification containing mapping information

2.12 Reflexion Modeling and Inverting Reflexion for Design Control (An Industrial Case Study of Architecture Conformance)

In the paper An Industrial case study of Architecture conformance, they reports on a case study which is designed to evaluate an approach for monitoring architectural drift during software development. This approach is designed for application within a forward engineering context. [18] In this paper they have identified that the degradation of a software system's design, as a result of its very own implementation and the continuous evaluation is a well documented phenomenon known as architectural drift.

They report on a 2-year ongoing case study designed to evaluate a lightweight process for monitoring the as-implemented or descriptive architecture of a software system during its development phase. That process is based on an inverted version of Reflexion Modeling [19] as suggested by Hochstein [20] and Knodel [21]

They argue that the architectural conformance process should be explicitly defined and should be applicable during the initial development of the software system. The reason is, if not, by the time the software is developed, it may have already drifted substantially from its initial architecture. They further argue that it should be applied by professional software developers to large commercial systems during development and the developers should then be given the results of the process to guide subsequent development.

2.12.1 Reflexion Modeling

This is a semi-automated, diagram-based structural summarization technique that programmers can use to aid the comprehension of software systems.

This technique has 6 process steps,

- The software engineer creates a hypothesized architectural model, the High-Level Model (HLM)
- A dependency graph of the subject system's sources is extracted, creating the Source Model (SM)
- The software engineer creates a mapping assigning SM entities to HLM entities
- The relationships in the HLM are compared with relationships in the SM. Results of that comparison is presented in the resulting Reflexion Model (RM). The following relationships are represented in this model
 - A solid edge represents a relationship present in both, the HLM and the SM. (convergence)
 - A dashed edge represents a relationship present in the SM, not present in the HLM. (divergence)
 - A dotted edge represents a relationship present in the HLM, not present in the SM. (absence)
- By analyzing these relationships in the RM, engineers can either alter the mappings, the HLM, or the SM (through re-factoring the source code)
- Steps 4 and 5 are repeated until the recovered model is consistent

2.12.2 Inverting Reflexion for Design Control

The proposed architecture conformance checking based on a inversion of the above mentions steps.

- Before implementation of the system commences, designer creates a HLM representing system's as-designed architecture
- During the implementation phase, developers or/and architects, update the mappings to reflect developer changes to the code base
- At any time, a RM can be generated to verify conformance of the implementation
- Subsequently the resulting model is analyzed to reveal the following
 - Where the implementation is consistent with the design (convergences)
 - Where the implementation violates the design (divergences and absences)
- Of most interest are the violations, i.e. the absence of any edge in the SM where one is expected, or the presence of an edge where one is not. Engineers may choose to take one of the following actions to correct the issues
 - The violation may be corrected by updating the code base (changing the SM)
 - Mappings may be updated, reassigning an SM entity to a different HLM entity
 - The 'as implemented' design may be considered acceptable and the HLM updated accordingly
 - The violation may be accepted and documented for later consideration
- Steps 3 and 4 are applied repeatedly until the project is completed

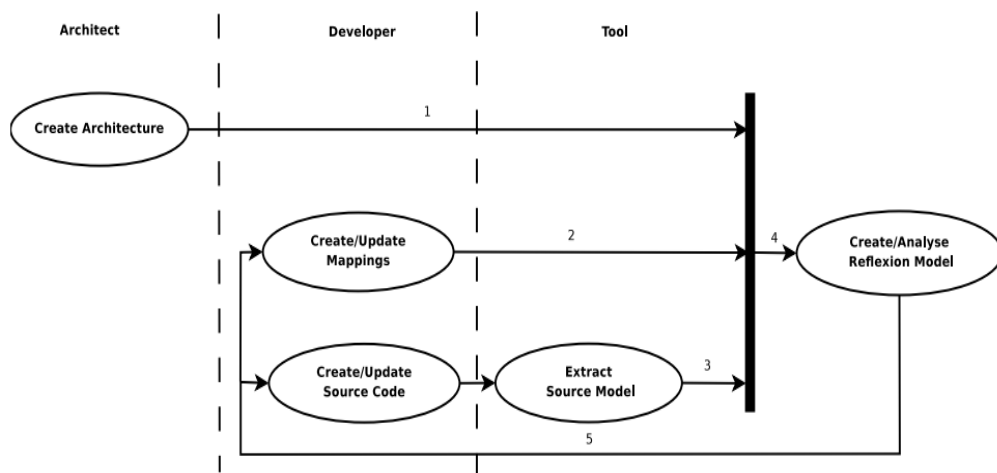


Figure 2-4 : The inverted Reflexion Modeling process

2.13 Traceability Model For Viewing Architectural Tactics Using Code Comments

There is a research which focuses on an analysis into a model to retain knowledge of architectural tactics of a software project along with its source code, by a non-intrusive, centralized way using code comments. A tool is implemented to read the information in the model and generate documentation views based on the information present in the model [22].

They expect to have several benefits from it, including,

- Preserve the unified knowledge of architectural tactics, design patterns and quality goals of a software project in an assured and centralized way
- Provide comprehensive visualization to easily correlate the above: quality goals, tactics, and the design patterns; thereby eliminating the abstractness of the relationships of those elements
- Using the visualizations, function as a traceability mechanism to guard against architecture degradation at every system build
- Facilitating easy ramping up of new team members by helping them to easily understand the "architecture vision" of the software system

The primary mechanism here is the designed framework will read metadata and provide high-level descriptions about the prescribed architecture of the software system. The idea of DesignDoc, the tool developed as a proof of concept, is to be a pluggable tool to view architectural tactics adopted in a software project while serving as a traceability mechanism to detect any degradation of the architecture.

Design patterns defined in Gamma et al. (1995) are essentially formations of source elements: i.e. classes, interfaces, class attributes and class methods. They have leveraged this idea and tried to represent and retain that information of design patterns using a suitable data model. In this case XML and JSON formats are considered. Among those XML is preferred since it is more readable and extensible in hierarchical and relational data representation found in this context.

```

<pattern name="adapter_pattern">
  <classes>
    <class name="adapter" required="true">
      <interfaces>
        <interface name="target" required="false">
          <methods>
            <method name="adapt" required="false"></method>
          </methods>
        </interface>
      </interfaces>
      <methods>
        <method name="adapt" required="true"></method>
      </methods>
      <attributes>
        <attribute name="adaptee" required="true"></attribute>
      </attributes>
    </class>
    <class name="adaptee" required="true"></class>
  </classes>
</pattern>

```

Figure 2-5 : XML based design pattern specification for Adapter pattern

```

{
  "name": "adapter_pattern",
  "classes": [
    {
      "name": "adapter",
      "required": "true",
      "interfaces": [
        {
          "name": "target",
          "required": "false",
          "methods": [
            {
              "name": "adapt",
              "required": "false"
            }
          ]
        }
      ]
    },
    {
      "name": "adaptee",
      "required": "true"
    }
  ],
  "methods": [
    {
      "name": "adapt",
      "required": "true"
    }
  ],
  "attributes": [
    {
      "name": "adaptee",
      "required": "true"
    }
  ]
}
]
}

```

Figure 2-6 : JSON based design pattern specification for Adapter pattern

Considering the pattern, class, interface, method and field relationships in, a XML Schema Definition (XSD) can be created to describe the elements of a design pattern XML.

```
<pattern name="adapter_pattern">
  <classes>
    <class name="adapter" required="true">
      <interfaces>
        <interface name="target" required="false">
          <methods>
            <method name="adapt" required="false"></method>
          </methods>
        </interface>
      </interfaces>
      <methods>
        <method name="adapt" required="true"></method>
      </methods>
      <attributes>
        <attribute name="adaptee" required="true"></attribute>
      </attributes>
    </class>
    <class name="adaptee" required="true"></class>
  </classes>
</pattern>
```

Figure 2-7 : Minimal XSD needed to represent design pattern information

They have considered two possible ways to create a mapping between source elements and design pattern elements. One is addition of Java annotations to source code elements and other is commenting source elements with Javadoc tags.

```
/**
 * An adapter class to handle conversion of {@link MerchantLookupResponse}
 * to {@link MerchantRegistrationRequest}.
 *
 * @author Umesh Liyanage
 */
@DesginDoc(pattern="adapter_pattern", classifier="adapter")
public class MerchantRegsitrationRequestAdapter {

    @DesginDoc(pattern="adapter_pattern", classifier="adaptee")
    private MerchantLookupResponse lookupResponse;

    @DesginDoc(pattern="adapter_pattern", classifier="adapted")
    public MerchantRegistrationRequest getRequest(MerchantLookupResponse
lookupResponse) {
    }
}
```

Figure 2-8 : Using Java annotations to map design pattern specifications

This idea about creating a mapping between source code and the relevant specification (design patterns or the intended architecture) can be used to track design degradation as well.

Chapter 3
METHODOLOGY

Developing software to last long and have no major problems is a complex task. One major threat is software architecture erosion. It can happen in many ways, simply put if one developer makes some sloppy mistake the whole software might be at trouble [24]. It might not be straight away, but take a long time, but by that time it might be too late to recover from that.

The primary objective of this research is to identify a way to detect software architecture erosion and what causes it by analyzing the code implemented by the developers of the software. This can either be done at the code commits stage or at the local development environments [23]. So the intended methodology will need to have a way to capture the architectural specifications or the descriptive architecture from the code which is implemented or committed by the developer and compare it with the original architecture. The architecture conformance can be take place after that.

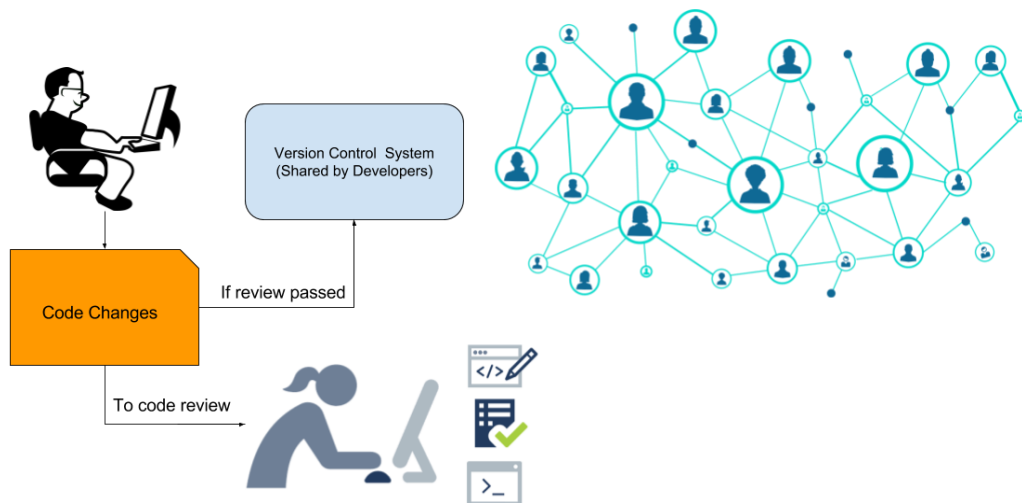


Figure 3-1 : Typical code review process with a reviewer other than the developer itself

3.1 Identifying a Possible Architecture Capturing Mechanism

With the literature survey about the past and present researches, several methodologies have been identified to cater the need of having a mapping between source code elements and the architectural specifications.

- Analyzing the source code using a Reflxion model

- Using a architecture description language to capture and preserve information about the architecture
- Using an inverted Reflexion model
- Using a framework to represent and retain architectural information using a suitable data model (XML, JSON with Javadocs)

For this research the latter was considered. As identified using a framework to represent and retain architectural information using a suitable data model is promising method with its own benefits.

- There will be no source code changes which leads to unnecessary dependencies
- Easy to understand and implement accordingly
- There are still few research methodologies to discover in this area

There were few possible ways to represent the descriptive architecture in a code base.

- Using Javadoc annotations
This was not chosen because with this the source code will get changed
- Using Javadoc comments
Since there will be no source code changes other than the doc comments this was selected.

These two was analyzed and compared to identify the best suitable methodology to identify and represent the descriptive architecture for this research.

Then a Doclet program was used to capture that information represented in the code. Doclet programs work with the Javadoc tool to generate documentation from code written in Java. Doclets are written in the Java and use the Doclet API, which provides an environment that allows clients to inspect the source-level structures of programs and libraries, including API comments embedded in the source.

3.2 Identifying Possible Architecture Violation Types to Consider For a Proof of Concept

There are many architecture violations to consider but for this research it was identified that considering the architecture style invariants violations would be a suitable candidate. An invariant is a condition that can be relied upon to be true during execution of a program, or during some portion of it, for an example one architect can decide that a layered architecture has an invariant stating that one layer can access only the below and above layer only.

3.3 Developing a Proof of Concept

In order to prove that this research idea actually matters and can actually contribute the software product line and software development life cycle, a tool was implemented. A project which uses a layered architecture style was used as the project under analysis. The tool was used to detect the architecture violations induced to that source code.

To check the viability of using this approach to enhance the continuous integration and continuous delivery a maven plug-in was created. It can be integrated to the local build process and the continuous integration process (Using a jenkins plug-in).[25][26]

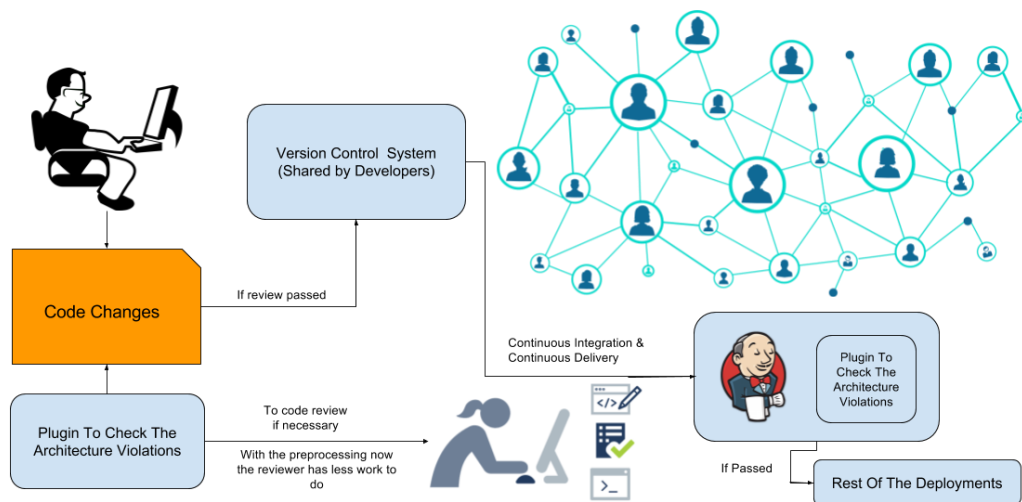


Figure 3-2 : Development process after the introduction of the solution

3.4 Evaluation of the Proof of Concept

The implemented version was evaluated in order to check its performance and capability of identifying the violations correctly. Both empirical and theoretical evaluations have been done in order to solidify the evaluation. Improvement and enhancements were identified and suitable conclusions were made with the results.

Chapter 4
SOLUTION ARCHITECTURE AND IMPLEMENTATION

Checking architectural style invariants and confirming they are intact can lead to a way to identify whether the prescriptive architecture is degraded or not. [27] Violations of style invariants massively contributed towards architecture erosion. [28]

As identified,

- The solution should not add intrusions to the source code. I.e. the actual source code should not include any additional dependencies
- There should be a way to define a rule set to identify the prescriptive architectural aspects
- It should be easy to extend to capture other types of invariants and architectures
- It should be a single dependency and easy to configure
- Should operate in acceptable processing times and resource utilizations

4.1 Presentation of Rules for Style Invariants and Prescriptive Architecture

Basically the conformity includes comparing the prescriptive architecture rules with the descriptive architecture, i.e. the architecture identified by the existing source code. So presentation of the intended prescriptive architecture is quite important. [29] It can be a one single presentation for the overall architecture or can be separate rules for style invariants and others. [33]

4.1.1 Presenting the Rules via XML

XML (eXtensible Markup Language) is a software and hardware independent markup language for storing and transporting data. Data represented through XML is both human and machine readable. The design goals of XML emphasize simplicity, generality, and usability across the Internet. [30] XML has come into common use for the interchange of data over the Internet also.[31]

Figure shows representing simple layered architecture invariants and details about the architecture using XML.

```
simple_layer_access.xml x
1 <layers>
2   <layer>
3     <name>presentationlayer</name>
4     <server>applicationlayer</server>
5     <client></client>
6   </layer>
7   <layer>
8     <name>applicationlayer</name>
9     <server>businesslayer</server>
10    <client>presentationlayer</client>
11  </layer>
12  <layer>
13    <name>businesslayer</name>
14    <server>dataaccesslayer</server>
15    <client>applicationlayer</client>
16  </layer>
17  <layer>
18    <name>dataaccesslayer</name>
19    <server></server>
20    <client>businesslayer</client>
21  </layer>
22 </layers>
```

Figure 4-1 : Representing simple layered architecture invariants and details about the architecture using XML

4.1.2 Presenting the Rules via JSON

JavaScript Object Notation or JSON is a standard file format which can be used for multiple purposes.[32] It uses human readable and machine readable text to transmit objects of data which are commonly attribute-value pairs and array data types.

It was derived from JavaScript but now widely used from others too. Now it's language independent data format.

Figure shows representing simple layered architecture invariants and details about the architecture using JSON.

```
simple_layer_access.json ×
1  {
2  "layers" : [
3  {
4    "name" : "presentationlayer",
5    "client" : "",
6    "server" : "applicationlayer"
7  },
8  {
9    "name" : "applicationlayer",
10   "client" : "presentationlayer",
11   "server" : "businesslayer"
12 },
13 {
14   "name" : "businesslayer",
15   "client" : "applicationlayer",
16   "server" : "dataaccesslayer"
17 },
18 {
19   "name" : "dataaccesslayer",
20   "client" : "businesslayer",
21   "server" : ""
22 }
23 ]
24 }
```

Figure 4-2 : Representing simple layered architecture invariants and details about the architecture using JSON

4.2 Mapping Source Elements to Architectural Information

Source or the code base is the most important artifact here. The solution should introduce a way to extract the architecture information from the code and compare the extracted architecture (i.e. the descriptive architecture) with the defined architecture.

Although there are complex ways to extract those details as mentioned before, in this research it's mainly focused to check other ways to do the same.

One identified such solution is to introduce some source code changes to be done by the developer or the architect during the development time. This method will require an initial training to use the tool and write a code which complies with that tool.

As identified by previous researchers there are many ways to do so. This research only focuses on two out of them.

- Mapping elements using Java Annotations
- Mapping elements using JavaDoc tags

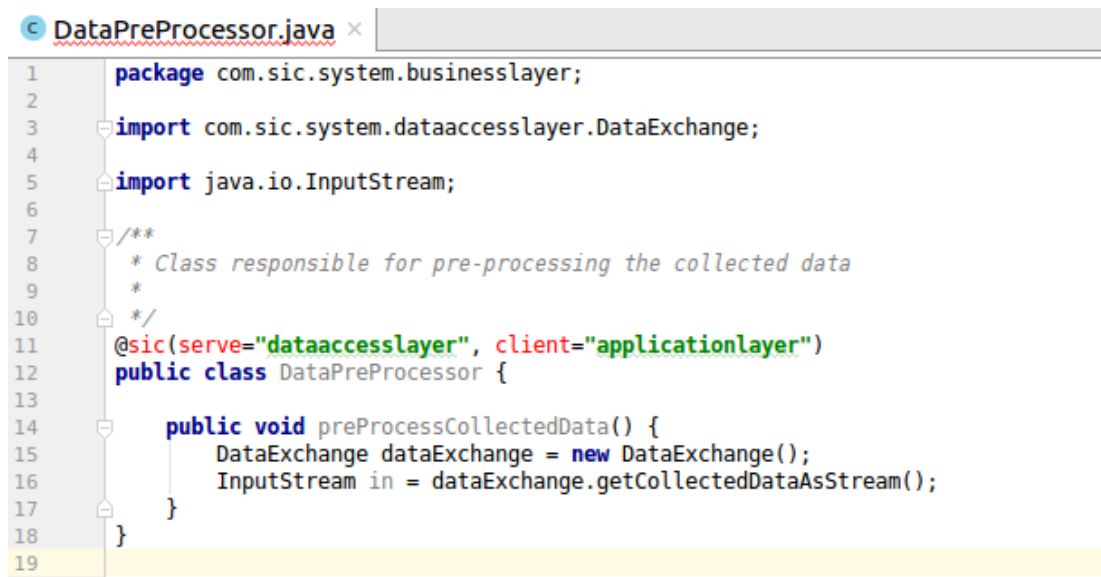
4.2.1 Mapping Elements Using Java Annotations

Java annotations are used to provide metadata for a Java code. Since its metadata, Java annotations do not directly affect the execution of the code. Even though it is stated like that, some types of annotations can actually be used for that purpose. Java annotations were added to Java from Java 5.

There are three typical use cases of Java annotations. For the purpose of Compiler instructions, Build-time instructions and Runtime instructions. [34] When using in the build time, the build process includes generating source code, compiling the source, generating XML files like deployment descriptors and packaging the compiled artifacts. An automated build tool like maven can be used to building the system, and those build tools may scan the Java code to identify specific annotations and generate source code or other files based on these annotations.

Java annotations will not be present in the code after compilation. It is possible to define custom annotations which are available at runtime. Those annotations can be accessed via Java Reflections also [35]. By using that kind of an approach we can extract some information about the source code, like architecture details states like in this case.

The figure shows how we can use Java annotations to present the details of the architecture and other information like style invariant. [36]



```
1 package com.sic.system.businesslayer;
2
3 import com.sic.system.dataaccesslayer.DataExchange;
4
5 import java.io.InputStream;
6
7 /**
8  * Class responsible for pre-processing the collected data
9  *
10 * */
11 @sic(serve="dataaccesslayer", client="applicationlayer")
12 public class DataPreProcessor {
13
14     public void preProcessCollectedData() {
15         DataExchange dataExchange = new DataExchange();
16         InputStream in = dataExchange.getCollectedDataAsStream();
17     }
18 }
19
```

Figure 4-3 : Use Java annotations to present the details of the architecture and other information like style invariant

4.2.2 Mapping Elements Using Javadoc Tags

As Oracle state Javadoc is a tool for generating API documentation in HTML format from doc comments in source code.

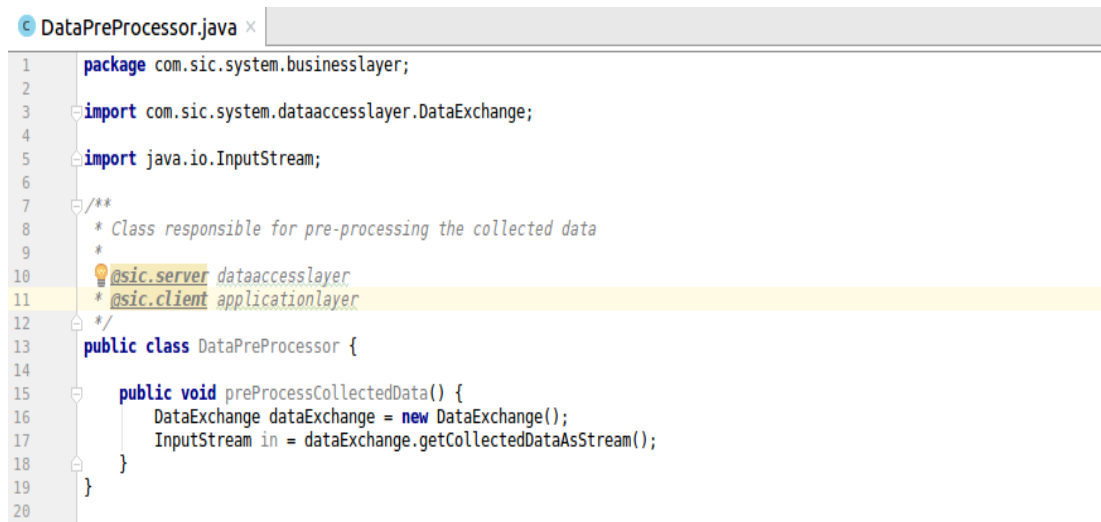
Javadoc uses a format called "doc comments" and it's a common industry standard for documenting classes. Some IDEs (Integrated Development Environments) like Eclipse, Netbeans, JIdea automatically generate Javadoc HTML reports.

Javadoc also provides a good API for creating doclets and taglets, which allows users to analyze the structure of the Java application.

As Oracle state The Doclet API (also called the Javadoc API) will provide a mechanism for the users to inspect the source-level structure of programs and libraries, which has embedded javadoc comments in the source. That can be used by a third party tool to do any kind of documentation or checking the program itself.

Doclets are typically invoked by javadoc can use this API to write out program information to any format. The javadocs' standard doclet is called by the default and writes out documentation to HTML files.

The figure shows how we can use Javadoc tags to present the details of the architecture and other information like style invariant.



```
1 package com.sic.system.businesslayer;
2
3 import com.sic.system.dataaccesslayer.DataExchange;
4
5 import java.io.InputStream;
6
7 /**
8  * Class responsible for pre-processing the collected data
9  *
10 * @sic.server dataaccesslayer
11 * @sic.client applicationlayer
12 */
13 public class DataPreProcessor {
14
15     public void preProcessCollectedData() {
16         DataExchange dataExchange = new DataExchange();
17         InputStream in = dataExchange.getCollectedDataAsStream();
18     }
19 }
20
```

Figure 4-4 : Use Javadoc tags to present the details of the architecture and other information like style invariant.

4.2.3 Comparison of Java Annotations and Javadoc

Most of the research literature has identified similarities and dissimilarities of these two. Below table contains a comparison of the two approaches.

Considering the below factors, Javadoc tags was chosen as the optimal approach for mapping source elements to design pattern specifications.

Table 4-1 : Using Annotations vs Javadoc tags

	Using Annotations	Using Javadoc tags
Extendibility to add custom design patterns	High (Java annotations provide implementation level flexibilities to easily extend the solution)	Medium (Passing auxiliary data to The tool from Javadoc tags is limited)
Portability of The tool across projects	Medium (Target application must use a Java version above Java 1.5)	High (Target application can be of any Java version)
Ease of setting up Same tool in a project	(Only need to add the tools' library to the application as a dependency)	(Only need to add the tool's library to the application as a dependency)
Learn ability of using The tool	Low (Developers need to know about using Java annotations)	High (Only need to know about adding Javadoc tags)
Ease of using The solution in an application	Medium (Needs to annotate source elements)	High (Only need to comment the source elements)
Level of intrusion to the code	High (Have to insert annotations which are Java code elements processed in the compiler)	Very Low (Only code comments)
Overall performance of processing in The tool	Less (High overhead to scan all the sources and filter out candidate source elements)	More (Javadoc is fast and has minimal overhead to filter out)

	based on annotations)	sources with matching tags)
Ease of integrating with Javadoc to generate Javadoc documentation	Less	Naturally more
Cross-platform portability (i.e. implementing The tool on another implementation platform like C#, C++)	None (Java annotations are specific to Java and will not be able to manipulate by any other implementation language)	Low (Javadoc only work on Java source files but commenting scheme of using @comment tags may still be portable to any platform if needed. It will be a matter of reading them from source files)
Relative effort to implement	More	Less

4.3 Solution Architecture

As identified the solution system architecture of the Style Invariants Checker (SIC) will include few main components,

- Source code parser
- Rule processor
- Architecture confirmation validator
- Style invariants checker
- View module

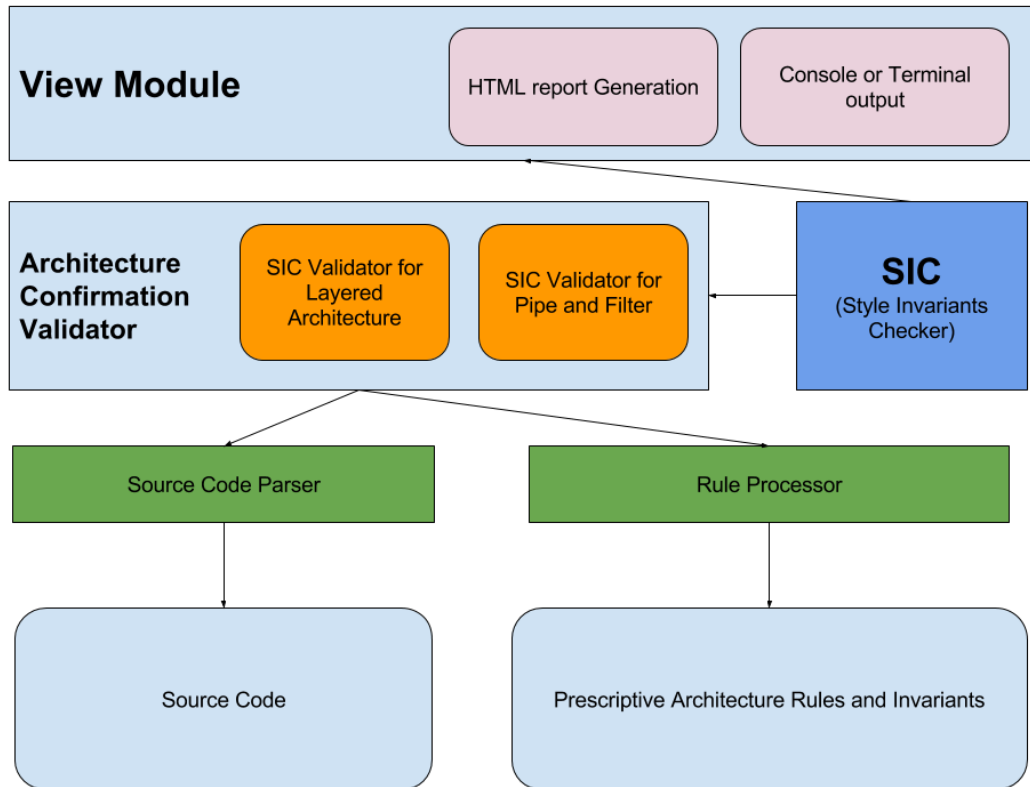


Figure 4-5 : Solution Architecture

4.3.1 Validating System Architecture Using JavaDoc Tags in the Source Code

As identified above JavaDoc tags which implemented for custom purposes, can be used to inject some information about the descriptive architecture of the system [37]. This information injecting should be done by the developer or the system software architect at the time of developing.

For an example if the system follows a layered architecture there will be several style invariants.

Below are some typical style invariants of layered architecture.

- Limiting the layer interactions to adjacent layers only
 - This can be further customized (by violating the fact that only adjacent layer interactions are allowed) such that any layer can use the services of any below layer or something similar to that.
- Below layer should support the requirements of the immediately above layer

When these style invariants add up it is hard to manually decide whether the system comply with the actual prescriptive architecture or not [38]. Especially when the system is quite complex and large and handled by large number of developers.

Using this research methodology the architects can design certain rules using the prescriptive architecture. Then when developing how to inject the architectural decisions made to the source code itself.

In a simple case like simple layered system, it can be done by simply mentioning what are the packages the class access and what are the packages this class should provide support for. Packages can be categorized as relevant layers for the ease of development. In a complex scenario we can add more detailed information like what are the functions access certain layers and how those happens in the code.

4.3.2 How Descriptive Architecture Information Is Identified

As Oracle describes the Doclet API (also called the Javadoc API) provides a mechanism for clients to inspect the source-level structure of programs and libraries, including javadoc comments embedded in the source. Doclets are programs written in the Java™ programming language that use the doclet API to specify the content and format of the output of the Javadoc tool. By default, the Javadoc tool uses the "standard" doclet provided by Sun™ to generate API documentation in HTML form. However, we can supply our own doclets to customize the output of Javadoc as you like. We can write the doclets from scratch using the doclet API, or we can start with the standard doclet and modify it to suit our needs. A simple custom doclet can be like below,

```
import com.sun.javadoc.*;

public class ListClass {
    public static boolean start(RootDoc root) {
        ClassDoc[] classes = root.classes();
        for (int i = 0; i < classes.length; ++i) {
            System.out.println(classes[i]);
        }
        return true;
    }
}
```

What this doclet does is it will take the classes upon which Javadoc is operating and prints their names to standard out.

As mentioned before custom tags may also be created. Here is an example usage of custom tags.

Let's say we want use a custom tag, say **@invariants**, in your documentation comments in addition to the standard tags like **@param** and **@return**. To make use of the information in our custom tags, we need to have our doclet use instances of `Tag` that represent our custom tags. One of the easiest ways to do that is to use the `tags (String)` method of `Doc` or one of `Doc`'s subclasses.

```
import com.sun.javadoc.*;

public class ListTags {
    public static boolean start(RootDoc root) {
        String tagName = "invariants";
        writeContents(root.classes(), tagName);
        return true;
    }

    private static void writeContents(ClassDoc[] classes, String
tagName) {
        for (int i=0; i < classes.length; i++) {
            boolean classNamePrinted = false;
            MethodDoc[] methods = classes[i].methods();
            for (int j=0; j < methods.length; j++) {
                Tag[] tags = methods[j].tags(tagName);
                if (tags.length > 0) {
                    if (!classNamePrinted) {
                        System.out.println("\n" + classes[i].name() +
"\n");
                        classNamePrinted = true;
                    }
                    System.out.println(methods[j].name());
                    for (int k=0; k < tags.length; k++) {
                        System.out.println("    " + tags[k].name() + ":
"
+ tags[k].text());
                    }
                }
            }
        }
    }
}
```

The tag for which this doclet searches is specified by the variable `tagName`. The value of the `tagName` string can be any tag name, custom or standard. This doclet writes to standard out, but its output format could be modified, for example, to write HTML output to a file.

So it is clear that by using custom tags we can extract some information associated with them. That is how the information about the descriptive architecture is gathered.

4.4 Prototype Implementation

For the purpose of proof of concept let's consider a simple layered architecture of a system. Where we have few layers like below,

- Presentation layer
- Application layer
- Business layer
- Data access layer

Folder and package structure may look like below,

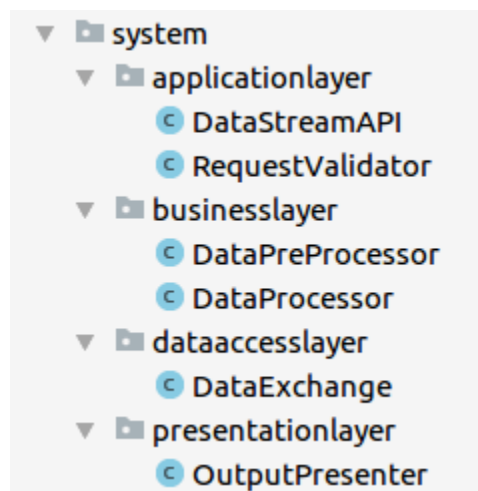


Figure 4-6 : Folder and package structure

Few of the classes implemented for this is included in the Appendix A.

To store the details about prescriptive architecture we can use a JSON as below.



```
1  {
2  "layers" : [
3  {
4    "name" : "presentationlayer",
5    "client" : "",
6    "server" : "applicationlayer"
7  },
8  {
9    "name" : "applicationlayer",
10   "client" : "presentationlayer",
11   "server" : "businesslayer"
12 },
13 {
14   "name" : "businesslayer",
15   "client" : "applicationlayer",
16   "server" : "dataaccesslayer"
17 },
18 {
19   "name" : "dataaccesslayer",
20   "client" : "businesslayer",
21   "server" : ""
22 }
23 ]
24 }
```

Figure 4-7 : Simple_layer_access.json

To process that rules and store it in memory we can a Java program which use Doctet and JavaDoc annotations to extract the data. Java code for a basic implementation for an analyzer is included in Appendix B.

4.5 Generating Views for the Reports

For the scenario mentioned as the example it will create an output like below. Since it is printing to the standard output we can see below on the console itself.

```
Loading source files for package com.sic.system.presentationlayer...
Loading source files for package com.sic.system.businesslayer...
Loading source files for package com.sic.system.dataaccesslayer...
Loading source files for package com.sic.system.applicationlayer...
Constructing Javadoc information...
```

```
Analyzing classes of presentationlayer layer
```

```
Class OutputPresenter PASSED
```

```
Analyzing classes of businesslayer layer
```

```
Class DataProcessor PASSED
```

```
Class DataPreProcessor PASSED
```

```
Analyzing classes of dataaccesslayer layer
```

```
Class DataExchange PASSED
```

```
Analyzing classes of applicationlayer layer
```

```
Class DataStreamAPI PASSED
```

```
RequestValidator violates the layered architecture by accessing dataaccesslayer as a client
```

```
Process finished with exit code 0
```

4.6 Style Invariants Checker Added As a Separate Dependency through Maven

At the early stage of the research the SIC was designed to run as an inbuilt component of the application. Later it was ported out and resulted in a separate plug-in for the application which gave the flexibility to separately modify the SIC and maintain. In order to achieve this maven plug-in was created with the SIC functionality.

A plug-in should be created using a suitable methodology and as per this case Apache maven was selected, as it is the most widely used software project management and build automation tool. Developing a plug-in for maven is much suitable since most of the industry applications use Maven as the underline project management tool.

Plug-in creation using Maven is fairly easy than the others since it has a large community support and a good shared resources to follow on. The figure 4-14 shows the main Mojo class of the maven plug-in.

```

1  sicmavenplugin > src > main > java > sic > plugin > SicMain
2  SicMain.java x
3  import com.sun.tools.javadoc.Main;
4  import org.apache.maven.plugin.AbstractMojo;
5  import org.apache.maven.plugin.MojoExecutionException;
6  import org.apache.maven.plugin.MojoFailureException;
7  import org.apache.maven.plugins.annotations.Mojo;
8  import org.apache.maven.plugins.annotations.Parameter;
9  /**
10     * This is the main executor class of SIC
11     */
12     @Mojo(name = "scan")
13     public class SicMain extends AbstractMojo {
14         /**
15          * The javaSourceLocation
16          */
17         @Parameter(property = "scan.javaSourceLocation", defaultValue = "./src/main/java")
18         private String javaSourceLocation;
19         /**
20          * The javaSubPackageLocation
21          */
22         @Parameter(property = "scan.javaSubPackageLocation", defaultValue = "./src/main/java")
23         private String javaSubPackageLocation;
24         /**
25          * The layerRuleFileLocation
26          */
27         @Parameter(property = "scan.layerRuleFileLocation", defaultValue = "/")
28         private String layerRuleFileLocation;
29
30         public void execute() throws MojoExecutionException, MojoFailureException {
31             getLog().info("charSequence: " + "scanning the project .....");
32             Support.layerRuleFileLocation = layerRuleFileLocation;
33
34             Main.execute(new String[]{"-doclet", "sic.plugin.SimpleLayerAccessCheck", "-docletpath"
35                                     , ".",
36                                     , "-sourcepath"
37                                     , javaSourceLocation
38                                     , "-subpackages"
39                                     , javaSubPackageLocation});
40
41             if (!Support.isBuildPassed) {
42                 throw new MojoFailureException(Support.buildFailureMsg);
43             }
44         }
45     }

```

Figure 4-8 : The basic Mojo class for the Maven plug-in

In order to create the plug-in we had to add few dependencies through maven. Figure 4-15 shows the project pom.xml with the required dependencies.


```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>sic.plugin</groupId>
8     <artifactId>sic-maven-plugin</artifactId>
9     <version>1.0-SNAPSHOT</version>
10
11    <packaging>maven-plugin</packaging>
12
13    <name>Sample Parameter-less Maven Plugin</name>
14
15    <dependencies>
16        <dependency>
17            <groupId>org.apache.maven</groupId>
18            <artifactId>maven-plugin-api</artifactId>
19            <version>3.0</version>
20        </dependency>
21        <dependency>
22            <groupId>com.googlecode.json-simple</groupId>
23            <artifactId>json-simple</artifactId>
24            <version>1.1.1</version>
25        </dependency>
26        <dependency>
27            <groupId>com.sun</groupId>
28            <artifactId>tools</artifactId>
29            <version>1.4.2</version>
30            <scope>system</scope>
31            <systemPath>${java.home}/../lib/tools.jar</systemPath>
32        </dependency>
33
34        <!-- dependencies to annotations -->
35        <dependency>
36            <groupId>org.apache.maven.plugin-tools</groupId>
37            <artifactId>maven-plugin-annotations</artifactId>
38            <version>3.4</version>
39            <scope>provided</scope>
40        </dependency>
41    </dependencies>
42 </project>

```

Figure 4-9 : Pom.xml configuration for the plug-in

In order to this plug-in to be used by other applications this needs to go into the maven repository. After that the application which needs to access the plug-in should set the pom.xml configurations as shown in the figure 4-16.

```
abcmanager pom.xml
abcmanager x
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>com.sic.abcmanager</groupId>
8   <artifactId>abcmanager</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11   <build>
12     <plugins>
13       <plugin>
14         <groupId>sic.plugin</groupId>
15         <artifactId>sic-maven-plugin</artifactId>
16         <version>1.0-SNAPSHOT</version>
17         <configuration>
18           <layerRuleFileLocation>/resources/sicrules/simple_layer_access.json</layerRuleFileLocation>
19           <javaSubPackageLocation>com.sic.system</javaSubPackageLocation>
20         </configuration>
21         <executions>
22           <execution>
23             <phase>compile</phase>
24             <goals>
25               <goal>scan</goal>
26             </goals>
27           </execution>
28         </executions>
29       </plugin>
30     </plugins>
31   </build>
32 </project>
```

Figure 4-10 : How to include the plug-in to an application through maven plug-in configuration

To run the plug-in goal defined we need to use the **mvn sic:scan** command. Figure 4-17 shows an example instance of running the maven goal.

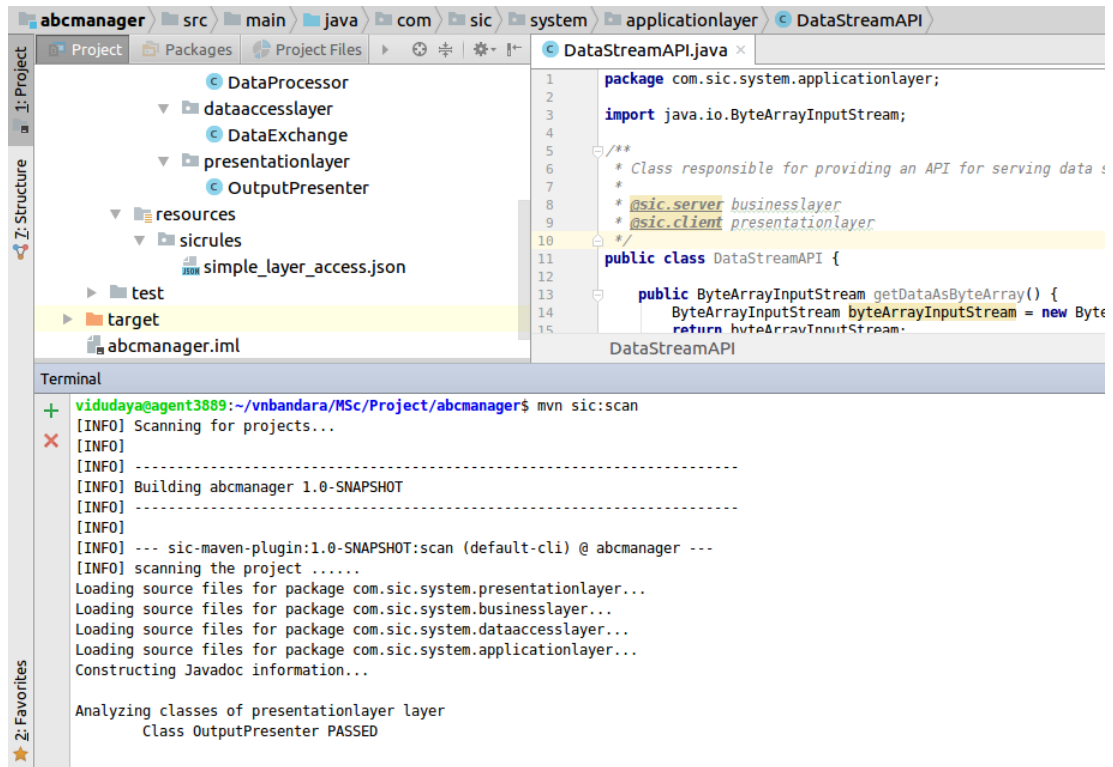


Figure 4-11 : Run the scan using the SIC plug-in

An example result of the maven goal execution is shown below. As it shown the scan took place with an actual build for the project.

```

[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building abcmanager 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- sic-maven-plugin:1.0-SNAPSHOT:scan (default-cli) @ abcmanager ---
[INFO] scanning the project .....
Loading source files for package com.sic.system.presentationlayer...
Loading source files for package com.sic.system.businesslayer...
Loading source files for package com.sic.system.dataaccesslayer...
Loading source files for package com.sic.system.applicationlayer...
Constructing Javadoc information...

Analyzing classes of presentationlayer layer
Class OutputPresenter PASSED

```

```
Analyzing classes of businesslayer layer
  Class DataProcessor PASSED
  Class DataPreProcessor PASSED

Analyzing classes of dataaccesslayer layer
  Class DataExchange PASSED

Analyzing classes of applicationlayer layer
  RequestValidator violates the layered architecture by accessing
  dataaccesslayer as a client
  Class DataStreamAPI PASSED

[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 0.939 s
[INFO] Finished at: 2017-12-31T11:56:19+05:30
[INFO] Final Memory: 10M/160M
[INFO] -----
[ERROR] Failed to execute goal sic.plugin:sic-maven-plugin:1.0-SNAPSHOT:scan
(default-cli) on project abcmanager: layered architecture violations detected !!!
[ERROR] RequestValidator violates the layered architecture by accessing
dataaccesslayer as a client
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read
the following articles:
[ERROR] [Help 1]
http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```

Chapter 5
EVALUATION

The evaluation of this methodology and the actual implementation of it rely on few factors. Few of them are the skill level of actual human resources who is going to test this manually and the complexity of the applications which the SIC is applied to.

The main reason of this research is to identify and propose a methodology to decrease architecture erosion of an enterprise level application. Many researches have been done in this area and this research will focus on a new tactic to identify architecture erosion. In here the research was done to identify how can JavaDoc comments help to reduce the violations to the prescriptive architecture when developing a complex system.

There can be many reasons to violate the prescriptive architecture by a developer. As per this research we narrowed it down to check whether some of the pre defined architecture style invariants are violated or not. Style invariant is a rule that should not be violated throughout the development process.

The evaluation strategy for the system that was developed to test this methodology (hereby called as SIC - Style Invariant Checker) consists of few phases.

- Empirical evaluation the correctness of the SIC
- Performance testing of the SIC
- Analytical evaluation of the impact of SIC when it is added to a continuous integration flow

5.1 Empirical Evaluation the Correctness Of the SIC

Basic flow of the SIC to check the descriptive architecture against the prescriptive architecture is as in the figure 5-1.

- Will load and read the prescriptive architecture rule set
- Will extract the descriptive architecture details from the application code base
- Will compare the prescriptive and descriptive architectures
- Will produce s set of results

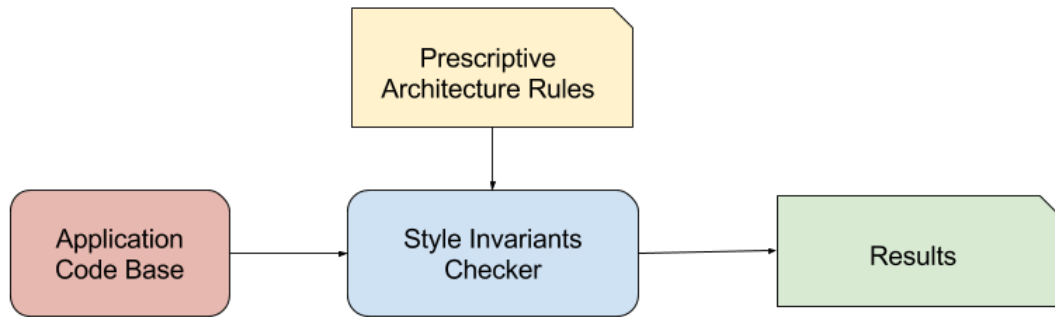


Figure 5-1 : Basic flow of Style Invariants Checker

The SIC itself contains few components as shown in the figure 5-2.

- Prescriptive Architecture Rules Processor
- Invariants Violation Checker
- Result Generator

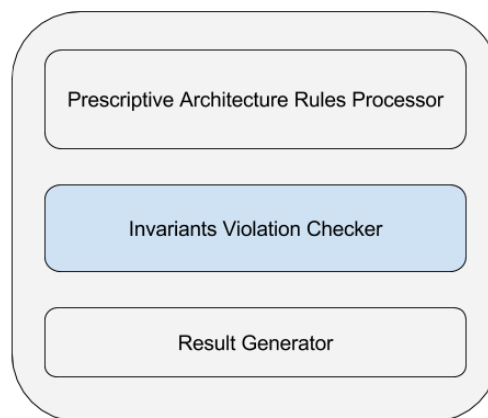


Figure 5-2 : Basic components of Style Invariant Checker

5.1.1 Evaluate the Correctness of the Loaded Prescriptive Architecture by SIC

For this we need to evaluate the correctness of the Prescriptive Architecture Rules Processor of the SIC, since it is the component, which the actual prescriptive architecture is loaded to the SIC. This must be correct to continue further rule validation on SIC.

The prescriptive architecture rules are pushed to the system via JSON file.

```

simple_layer_access.json x
1  {
2  "layers" : [
3  {
4    "name" : "presentationlayer",
5    "client" : "",
6    "server" : "applicationlayer"
7  },
8  {
9    "name" : "applicationlayer",
10   "client" : "presentationlayer",
11   "server" : "businesslayer"
12  },
13  {
14   "name" : "businesslayer",
15   "client" : "applicationlayer",
16   "server" : "dataaccesslayer"
17  },
18  {
19   "name" : "dataaccesslayer",
20   "client" : "businesslayer",
21   "server" : ""
22  }
23  ]
24 }

```

Figure 5-3 : Prescriptive Architecture rules can be presented in a JSON file format

We can test the results using a dedicated tests suite. We need to test whether the provided rules are actually being loaded or not. This evaluation strategy took few steps.

- Define 10 different prescriptive architecture rules documents (.json files)
- Loaded the rules via the SIC
- Evaluated the loaded results are correct or not

Table 5-1: Loaded prescriptive architecture validation test results

	Json 1	Json 2	Json 3	Json 4	Json 5	Json 6	Json 7	Json 8	Json 9	Json 10
Passed	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

The results (As in the table 5-1) shows that the Prescriptive Architecture Rules Processor can load the given prescriptive architecture rule set without a problem in all the cases.

5.1.2 Evaluation of the Style Invariants Checker Validation Components

In this section the SIC is tested under actual code base of an application. For this a sample application was developed with below properties.

- The application uses the layered architecture style
- The invariants to consider are defined as
 - A layer can only access the immediately above and below layers

Apart from the above invariants to the architecture, we introduced few organizational specific coding standards to be strictly followed in order to get a better result with SIC. SIC is supposed to extract the descriptive architecture using Javadoc comments using custom Javadoc tags specified for that architecture design. So developers need to strictly adhere to those as well. If not the SIC will throw an exception and result it as a failure case, a violation to the architecture.

5.1.3 Evaluation of the Success Path

For this case we used an application which correctly adheres to the above mentioned guidelines. So this case should verify that the SIC can identify that an application is being built correctly so far.

The results generated are as below, and that was the expected outcome as well.

```
Loading source files for package com.sic.system.presentationlayer...
Loading source files for package com.sic.system.businesslayer...
Loading source files for package com.sic.system.dataaccesslayer...
Loading source files for package com.sic.system.applicationlayer...
Constructing Javadoc information...

Analyzing classes of presentationlayer layer
  Class OutputPresenter PASSED

Analyzing classes of businesslayer layer
  Class DataProcessor PASSED
  Class DataPreProcessor PASSED

Analyzing classes of dataaccesslayer layer
  Class DataExchange PASSED

Analyzing classes of applicationlayer layer
  Class DataStreamAPI PASSED
  Class RequestValidator PASSED
```

5.1.4 Evaluation of the Failure Paths

The most important case is to evaluate whether the SIC is able to identify the violations to the intended architecture.

The application's system structure is as in the figure 5-4.

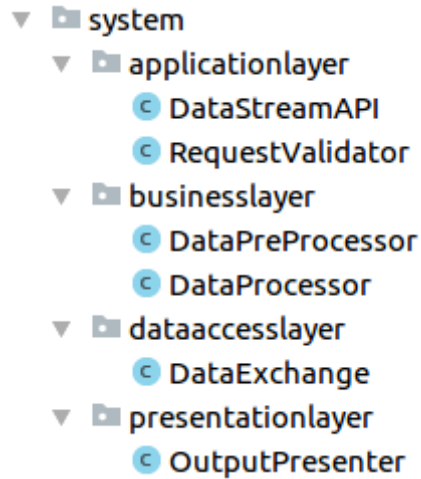


Figure 5-4: Application's system structure

Few cases where violations happen were identified first.

A case where a code section from a layer, access a functionality of another layer which is not allowed to access directly. Figure 5-5. Figure 5-6 shows the result of the SIC execution on that application.

```

RequestValidator.java x
1  package com.sic.system.applicationlayer;
2
3  import com.sic.system.dataaccesslayer.DataExchange;
4
5  import java.io.InputStream;
6
7  /**
8   * Class responsible for validating the request for data
9   *
10  * @sic.server dataaccesslayer
11  * @sic.client presentationlayer
12  */
13  public class RequestValidator {
14
15      public boolean validateDataRequest(Object request) {
16          boolean isValid = false;
17
18          // This is not a good practice
19          // (Trying to access the Data access layer functionality from app layer)
20          DataExchange dataExchange = new DataExchange();
21          InputStream in = dataExchange.getCollectedDataAsStream();
22
23          // validate 'request' using 'in'
24
25          return isValid;
26      }
27  }
28

```

Figure 5-5 : Code section from a layer, access a functionality of another layer which is not allowed to access directly

```

/usr/lib/jvm/java-7-oracle/bin/java ...
Loading source files for package com.sic.system.presentationlayer...
Loading source files for package com.sic.system.businesslayer...
Loading source files for package com.sic.system.dataaccesslayer...
Loading source files for package com.sic.system.applicationlayer...
Constructing Javadoc information...

Analyzing classes of presentationlayer layer
Class OutputPresenter PASSED

Analyzing classes of businesslayer layer
Class DataProcessor PASSED
Class DataPreProcessor PASSED

Analyzing classes of dataaccesslayer layer
Class DataExchange PASSED

Analyzing classes of applicationlayer layer
Class DataStreamAPI PASSED
RequestValidator violates the layered architecture by accessing dataaccesslayer as a client

Process finished with exit code 0

```

Figure 5-6 : Results of the SIC execution when a code section from a layer, access a functionality of another layer which is not allowed to access directly

For the other cases, the obtained results are shown in the below table. It summarizes the approach carried out to do each validation test and the expected and actual results. So for the case where we check the style invariants violation in a layered architecture style application, the SIC tend to perform in a significantly accurate manner.

Table 5-2 : Expected and Actual results when executing the SIC on applications with different violations

Scenario	Expected Result	Actual Result	Evaluation
The prescriptive architecture rule file is missing	An exception should be thrown with the relevant error message	An exception thrown with the message “The Prescriptive Architecture Rules file not found”	Correct
A class file in the application system package does not have a @sic.server tag or a @sic.client tag	An exception should be thrown with the relevant error message	An exception thrown with the message “The DataPreProcessor class does not have the required doc tags with the relevant information”	Correct
A class in a middle layer does not have the @sic.client tag	An exception should be thrown with the relevant error message	An exception thrown with the message “The DataStreamAPI class does not have the required @sic.client tag with the relevant information”	Correct
A class in a middle layer has a misspelled @sic.client tag name	An exception should be thrown as it was not found, with the relevant error message	An exception thrown with the message “The DataStreamAPI class does not have the required @sic.client tag with the relevant information”	Correct
A class in a middle layer has an empty @sic.client tag name	An exception should be thrown, with the relevant error message	An exception thrown with the message “The DataStreamAPI class does not have the required @sic.client tag with the relevant information”	Correct
A class has a syntax error	An exception should be thrown	Relevant exception was thrown after the SIC analysis	Correct

	but the code analysis should be done to detect architecture violations	performed	
An internal error occurred with the SIC	An exception should be thrown and the code analysis results should not be published	An Exception with the internal error was thrown	Correct

5.2 Performance Testing of SIC

For the evaluation of the performance of the SIC couple of workload factors were used [39]. The complexity of the prescriptive architecture rule set and the size of the code base or the number of Java class files in the code base.

This performance testing was done in a personal computer with following properties.

- Processor : Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
- Random Access Memory : 6.00 GB
- System Type : 64-bit Operating System
- Operating System : Linux (Ubuntu 12.0)

5.2.1 Performance Testing By the Complexity of the Prescriptive Architecture Rule Set

For this the same application with layered architecture was used. The size of the code base kept constant to 24 java classes. The classes under test were 20 altogether.

The complexity of the rule set was changed by adding few additional rules which are relevant to the layered architecture and the some relevant to organizations customizations, which includes rules relevant to development best practices.

Table 5-3 : Average execution time of SIC with the complexity of the prescriptive architecture rules

Complexity of the rule set (Number of rules)	Average Execution Time (milliseconds)
4	20
6	20
10	22
25	26
30	30

Here we can see that the execution time tend to increase with the complexity of the prescriptive architecture rule set. That variation is not significant. As the figure 5-7 shows we can predict that the execution time will increase with the complexity of the rule set when goes into rule sets with much higher complexities.

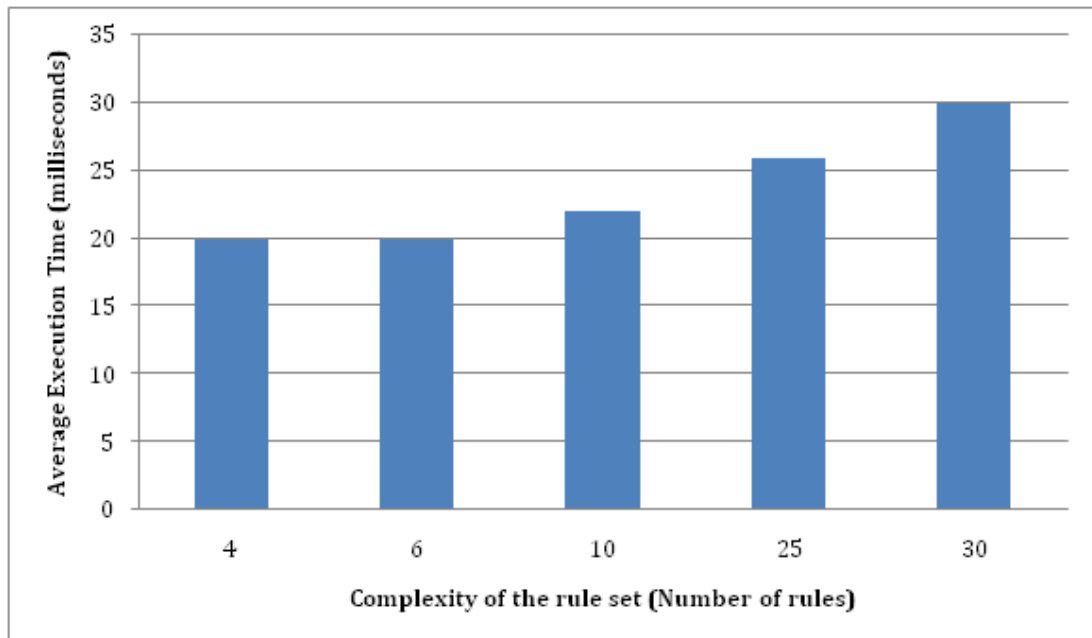


Figure 5-7 : Average execution time of SIC with the complexity of the prescriptive architecture rules

5.2.2 Performance Testing By the Size of the Code Base

Evaluating the performance of the SIC against the size of the code base is very important since this kind of a tool is much more useful to a fairly large and complex applications. For this testing the same application code base was used with systematically increased code base. SIC will load every class file under the considered package into the memory at some point and do the analysis. So when the number of classes increased a significant amount of increase in the execution time was expected. It was not increased by a very large number.

The test was again conducted on three scenarios where the complexity of the prescriptive architecture is differed. This was done to check how the complexity of the prescriptive architecture affects the run time along with the increased code base.

For that we used three different prescriptive architectures.

- The code base size was increased systematically and the prescriptive architecture has 4 rule sets - Case 1
- The code base size was increased systematically and the prescriptive architecture has 10 rule sets - Case 2
- The code base size was increased systematically and the prescriptive architecture has 15 rule sets - Case 3

The observations show that when the complexity of the prescriptive architecture changed then the execution time is also getting affected by that.

The execution time of this scenario will affect on the following factors,

- The processor which host the execution
 - This can be considered as an obvious case
- The memory capacity of the machine
 - This can be considered as an obvious case
- The complexity of the prescriptive architecture
 - Obtained from the above evaluation results
- The performance level of the rule processor
 - In the case of SIC this rule processor has to be implemented by a senior developer, probably by an architect. So that will also impact the performance of it since the implementations may differ from one to another.

Table 5-4 : Case 1, Change of the Average Execution Time with the size of the code base

Size of the code base	Average Execution Time (milliseconds)
20	35
50	64
100	95
150	130
200	170

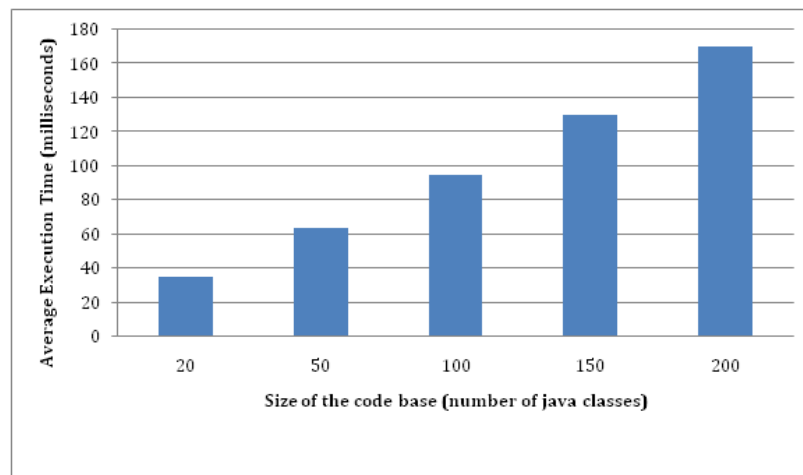


Figure 5-8 : Case 1, Change of the Average Execution Time with the size of the code base

Table 5-5: Case 2, Change of the Average Execution Time with the size of the code base

Size of the code base	Average Execution Time (milliseconds)
20	55
50	94
100	125
150	160
200	195

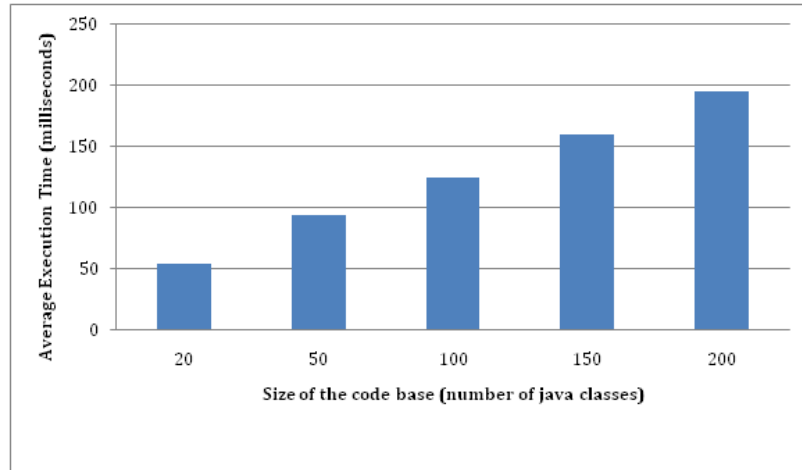


Figure 5-9 : Case 2, Change of the Average Execution Time with the size of the code base

Table 5-6 : Case 3, Change of the Average Execution Time with the size of the code base

Size of the code base	Average Execution Time (milliseconds)
20	70
50	110
100	135
150	175
200	205

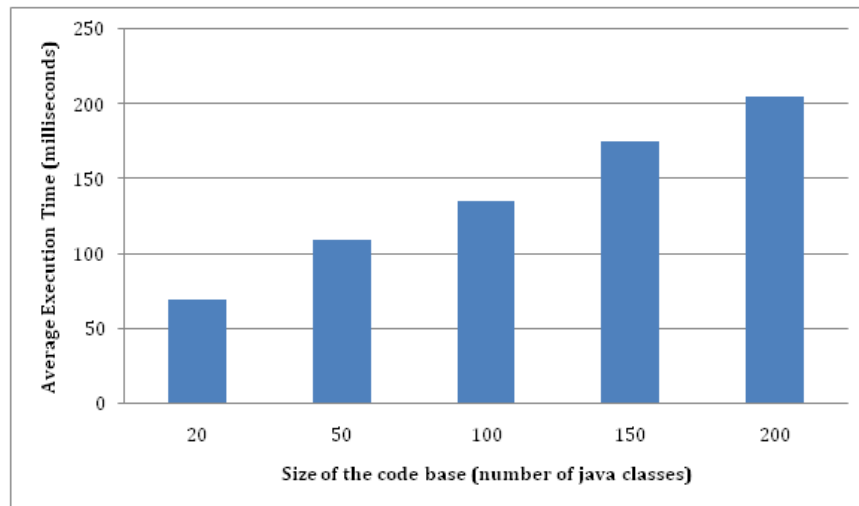


Figure 5-10 : Case 3, Change of the Average Execution Time with the size of the code base

5.3 Analytical Evaluation Of The Impact Of SIC When It Is Added To A Continuous Integration Flow

Since SIC is developed to practically detect the architecture violations of a code base change it is ideal to plug-in this to a proper continuous integration flow. SIC is ported as a Maven plug-in and the developer can do a local analysis for his changes to the code base. Then after the code goes into the shared repository the code will again get scanned for the architecture violations.

Before it is introduced to the CI flow it was like in the figure 5-11.

This phase 2 might take days with the availability of the reviewer resulting long waiting time to deploy the code in pre production environment. Results of that review will depends on the skill level of the reviewer and his quality of the understanding about the prescriptive architecture of the system.

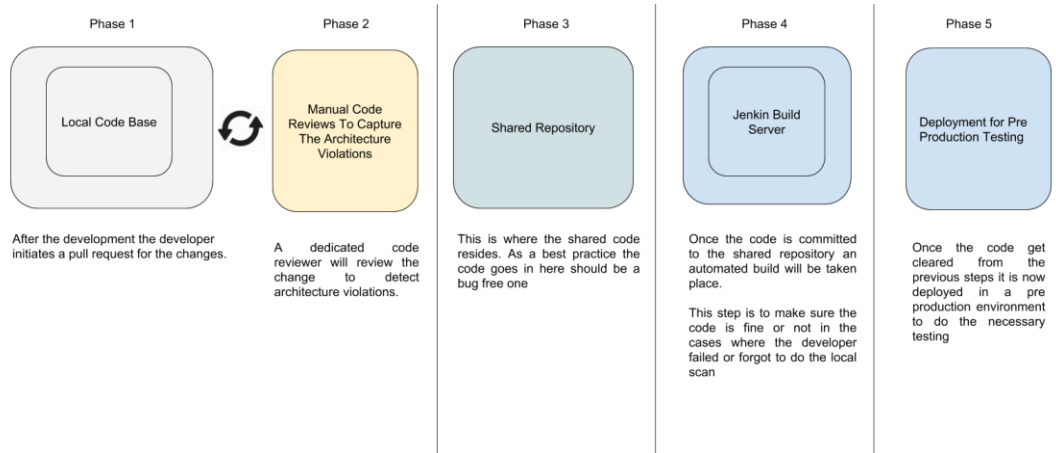


Figure 5-11 : Continues Integration flow before the introduction of Style Invariants Checker

After Introducing the SIC maven goal (Figure 5-12), the phase 2 can be enhanced or completely removed. SIC will have a comprehensive set of rules to check the descriptive architecture match the prescriptive architecture or not. It won't have the problems like when it is done by another reviewer.

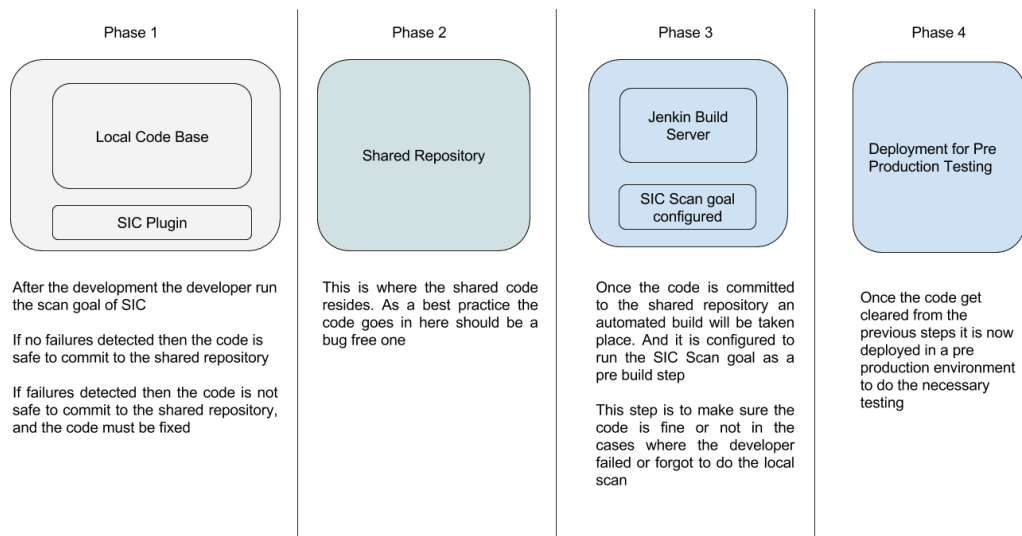


Figure 5-12 : Continues Integration flow after the introduction of Style Invariants Checker

Chapter 6
CONCLUSION

Development of a durable and maintainable software application is a challenging task for any development organization. Achieving nonfunctional quality goals like protecting the system architecture throughout the software development cycle, protecting the organization wise quality standards and fast time to market or deployment is directly affecting the above mentioned goal. There have been many researches carried out towards this area focusing on developing methodologies to protect the systems by retaining and properly managing the non functional quality goals.

6.1 Research Contribution

This research focuses on finding out an approach to preserve the prescriptive architecture using comments added by the developers throughout the development cycle. With this approach an organization can develop a standard to write code within the organization, and that includes users to inject special architecture specific information to the code that they write using JavaDoc comments. Then they have to develop a rule processor to process the code base against the prescriptive architecture information. This tool can be configured to support the systems' architecture in the beginning of the design development process. The prescriptive architecture information can be configured through a file with predefined rule set, preferably in JSON format.

The developed proof of concept is the Style Invariants Checker (SIC). SIC as per the POC version will take a JSON file with a predefined rule set which contain rules for a standard layered architecture pattern. There is an analyzer component to analyze the descriptive architecture and compare it with the predefined rule set. Then it will produce the results which will be use in the decision making process. This POC was evaluated both empirically and analytically to prove that this kind of a methodology can actually work in software development practice.

Yet actually using this methodology in a real complex software project is somewhat challenging task. Not because of the technical barriers (we have proved that it can actually be done using a SIC like tool) but because of the changes that needs to be done to the software development life cycle. If an organization needs to adopt this approach for a project then first they need to derive strong prescriptive architecture documentation. Then derive a rule set to be checked or define the architecture style invariants, and then identify how to enforce those rules within the code base. That is basically defining the required JavaDoc tags and where to use them in order to augment the codebase with additional information. After that each and every member of the development team needs to adhere to those guidelines. If an organization can make sure that these are in place then they can actually get the real benefit of this approach.

6.2 Research Limitations

Few limitations for this research are that this research methodology and the POC system was not tested in extreme conditions like within real life complex software applications. To be able to test it under those conditions above mentioned barriers should be cleared and a significantly larger development team is required [40]. That development team should be trained to understand what needs to be done and how to properly use this framework. To get actual progress over a significant time period we need to use this POC (SIC tool) in that project team for a fairly long time. It was not feasible because with the time constraints for this research.

For this research as a programming language Java was considered. Since most developers and organizations use Java as the preferred programming language for their development tasks. So this research methodology was actually limited to the resources and tools which support Java programming language.

6.3 Future work and Conclusion

In this research a basic tool called Style Invariants Checker (SIC) was implemented as a proof of concept. It can be enhanced in few ways to add extra advantage to using this approach. So far the JavaDoc tags to be placed inside the code base were externally defined, and the developers have to know those beforehand. If we can introduce an IDE plug-in with all those pre defined tags to be placed, then it would be much easier for the developers when writing code with introduced standard guidelines. Additionally if we can introduce a framework, specific to each project with the prescribed architecture then it would be much easier to use this approach in real life complex projects. Since SIC uses the JavaDoc comments added by the developer, A dedicated framework can actually inject some of the default architectural information into the code base so the work needs to be done by the developer gets reduced.

So far this SIC framework supports projects written in Java language only. This framework can be extended to support other implementation languages like C#, C++ etc. The descriptive architecture capturing part and the comparison needs to be done in that language.

This research was meant to find out the feasibility of identifying software architecture erosion through code changes done by the developer. Using a methodology which augment the code base with added comments to the codebase and use a set of predefined information to compare the prescriptive architecture with the descriptive architecture was a successful one. With the results of the evaluations we can conclude that augmenting the code with less complex information like JavaDoc comments can actually be used to decrease the architecture erosion up to a significant extend.

REFERENCES

- [1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice* (2nd Edition). Addison-Wesley Professional, 2 edition, April 2003
- [2] M. De Silva and I. Perera, "Preventing software architecture erosion through static architecture conformance checking", in *IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, 2015.
- [3] Parnas, D. L. Software aging. In *Proceedings of the 16th international conference on Software engineering* (Los Alamitos, CA, USA, 1994), ICSE '94, IEEE Computer Society Press, pp. 279–287.)
- [4] van Gurp, J., Brinkkemper, S., and Bosch, J. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.* 17 (July 2005), 277–306.
- [5] Guo, G. Y., Atlee, J. M., and Kazman, R. A software architecture reconstruction method. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)* (Deventer, The Netherlands, The Netherlands, 1999), WICSA1, Kluwer, B.V., pp. 15–34.
- [6] Murphy, G. C., Notkin, D., and Sullivan, K. J. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.* 27, 4 (Apr. 2001), 364–380.
- [7] Lung, C.-H., Zaman, M., and Nandi, A. Applications of clustering techniques to software partitioning, recovery and restructuring. *J. Syst. Softw.* 73, 2 (Oct. 2004), 227–244.
- [8] Sartipi, K. Software architecture recovery based on pattern matching. In *Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 2003), ICSM '03, IEEE Computer Society, pp. 293
- [9] Jansen, A., Bosch, J., and Avgeriou, P. Documenting after the fact: Recovering architectural design decisions. *J. Syst. Softw.* 81, 4 (Apr. 2008), 536–557
- [10] Abi-Antoun, M., Aldrich, J. (2009). Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*. ACM, New York, NY, USA, 321-340. [Online]. Available from: DOI=<http://dx.doi.org/10.1145/1640089.1640113>
- [11] Len Bass, Paul Clements, R. K. *Software Architecture in Practice*. 2000
- [12] M. Mirakhorli, "Preserving the Quality of Architectural Tactics in Source Code", 2014.
- [13] Rosik, J., Le Gear, A., Buckley, J., and Ali Babar, M. An industrial case study of architecture conformance. In *Proceedings of the Second ACM-IEEE*

- international symposium on Empirical software engineering and measurement (New York, NY, USA, 2008), ESEM '08, ACM, pp. 80–89
- [14] Bennett, K. (1996). Software evolution: past, present and future. *Information and software technology*, 38(11), 673-680.
- [15] Paul C. Clements. “A survey of architecture description languages”. In *Proceedings of the Eighth International Workshop on Software Specification and Design*. IEEE Computer Society Press, 1996
- [16] L. de Silva, “A Rationale-based Architecture Description Language using the Oslo Modelling Platform,” Master’s thesis, University of St Andrews, 2008
- [17] L. de Silva and D. Balasubramaniam, “A model for specifying rationale using an architecture description language,” in *Software Architecture. Proceedings of the 5th European Conference on Software Architecture (ECSA 2011)* (I. Crnkovic, V. Gruhn, and M. Book, eds.), pp. 319–327, Springer Berlin Heidelberg, 2011
- [18] J. Knodel, D. Muthig, and M. Naab. Understanding software architectures by visualization—an experiment with graphical elements. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 39–50, Washington, DC, USA, 2006. IEEE Computer Society
- [19] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *Computer*,30(8):29–36, 1997.
- [20] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. *sew*, 00:137, 2003.
- [21] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 279–294, Washington, DC, USA, 2006. IEEE Computer Society.
- [22] U. Liyanage and I. Perera, "Traceability Model For Viewing Architectural Tactics Using Code Comments", 2017.
- [23] Continuous Integration and Its Tools. (2014). *IEEE Software*, 31(3), pp.14-16.
- [24] Smith, T. (2010). Protecting the process [source code management]. *Engineering & Technology*, 5(4), pp.51-53.
- [25] Quibeldey-Cirkel, K. and Thelen, C. (2012). Continuous Deployment. *Informatik-Spektrum*, 35(4), pp.301-305.
- [26] CSS-Tricks. (2018). Why You Should Use Continuous Integration and Continuous Deployment | CSS-Tricks. [online] Available at: <https://css-tricks.com/continuous-integration-continuous-deployment/> [Accessed 22 Feb. 2018].
- [27] Kim, J. and Garlan, D. (2010). Analyzing architectural styles. *Journal of Systems and Software*, 83(7), pp.1216-1235.

- [28] Monroe, R., Kompanek, A., Melton, R. and Garlan, D. (1997). Architectural styles, design patterns, and objects. *IEEE Software*, 14(1), pp.43-52.
- [29] Mehta, N. and Medvidovic, N. (2003). Composing architectural styles from architectural primitives. *ACM SIGSOFT Software Engineering Notes*, 28(5), p.347.
- [30] Thongkum, S. and Vatanawood, W. (2014). An Approach of Software Architectural Styles Detection Using Graph Grammar. *International Journal of Engineering and Technology*, 6(2), pp.123-127.
- [31] Darshan, K. and P., S. (2017). Json is Efficient over the XML in Native Application. *International Journal of Computer Applications*, 165(8), pp.14-17.
- [32] Pandey, M. and Pandey, R. (2017). JSON and its use in Semantic Web. *International Journal of Computer Applications*, 164(11), pp.10-16.
- [33] Architectural Design Patterns for Language Parsers. (2014). *Acta Polytechnica Hungarica*, 11(5).
- [34] Lyon, D. (2010). Semantic Annotation for Java. *The Journal of Object Technology*, 9(3), p.19.
- [35] Ahuja, K. (2018). Are annotations bad?. [online] Java Code Geeks. Available at: <https://www.javacodegeeks.com/2015/08/are-annotations-bad.html> [Accessed 22 Feb. 2018].
- [36] dzone.com. (2018). How Do Annotations Work in Java? - DZone Java. [online] Available at: <https://dzone.com/articles/how-annotations-work-java> [Accessed 22 Feb. 2018].
- [37] ZHANG, L. (2008). Software Architecture Evaluation. *Journal of Software*, 19(6), pp.1328-1339.
- [38] Garlan, D. and Shaw, M. (1994). *An Introduction to Software Architecture*. Pittsburgh: Carnegie Mellon University.
- [39] Weyuker, E. and Vokolos, F. (2000). Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12), pp.1147-1156.
- [40] Marshall, A. (1991). A conceptual model of software testing. *Software Testing, Verification and Reliability*, 1(3), pp.5-16.

APPENDIX A

Few of the Java classes used for the proof of concept implementation.

- DataExchange.java

```
1 package com.sic.system.dataaccesslayer;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5
6 /**
7  * Class responsible for communicate with persistent data storage
8  *
9  * @sic.client businesslayer
10 */
11 public class DataExchange {
12
13     public InputStream getCollectedDataAsStream() {
14         InputStream inputStream = () -> { return 0; };
15
16         return inputStream;
17     }
18 }
19 }
```

- DataPreProcessor.java

```
1 package com.sic.system.businesslayer;
2
3 import com.sic.system.dataaccesslayer.DataExchange;
4 import java.io.InputStream;
5
6 /**
7  * Class responsible for pre-processing the collected data
8  *
9  * @sic.server dataaccesslayer
10 * @sic.client applicationlayer
11 */
12 public class DataPreProcessor {
13
14     public void preProcessCollectedData() {
15         DataExchange dataExchange = new DataExchange();
16         InputStream in = dataExchange.getCollectedDataAsStream();
17     }
18 }
19 }
```

- DataProcessor.java

```
1 package com.sic.system.businesslayer;
2
3 import java.io.IOException;
4 import java.io.OutputStream;
5
6 /**
7  * Class responsible for processing the collected data
8  *
9  * @sic.server dataaccesslayer
10 * @sic.client applicationlayer
11 */
12 public class DataProcessor {
13
14     /**
15      * @return processed data output stream
16      * @throws IOException
17      * @mytag neranjan
18      */
19     public OutputStream getProcessedDataAsOutputStream() throws IOException {
20         OutputStream outputStream = new OutputStream() {
21             @Override
22             public void write(int b) throws IOException {
23             }
24         };
25         return outputStream;
26     }
27 }
28 }
```

- **DataStreamAPI.java**

```
DataStreamAPI.java x
1 package com.sic.system.applicationlayer;
2
3 import java.io.ByteArrayInputStream;
4
5 /**
6  * Class responsible for providing an API for serving data stream requests from UI
7  *
8  * @sic.server businesslayer
9  * @sic.client presentationlayer
10 */
11 public class DataStreamAPI {
12
13     public ByteArrayInputStream getDataAsByteArray(){
14         ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(new byte[100]);
15         return byteArrayInputStream;
16     }
17 }
18
```

- **OutputPresenter.java**

```
OutputPresenter.java x
1 package com.sic.system.presentationlayer;
2
3 import java.io.OutputStream;
4
5 /**
6  * Class responsible for display the processed results
7  *
8  * @sic.server applicationlayer
9  */
10 public class OutputPresenter {
11
12     /**
13     * @param outputStream
14     */
15     public void showProcessedData(OutputStream outputStream) {
16         // presenting data
17     }
18 }
```

- **RequestValidator.java**

```
RequestValidator.java x
1 package com.sic.system.applicationlayer;
2
3 import com.sic.system.dataaccesslayer.DataExchange;
4
5 import java.io.InputStream;
6
7 /**
8  * Class responsible for validating the request for data
9  *
10 * @sic.server dataaccesslayer
11 * @sic.client presentationlayer
12 */
13 public class RequestValidator {
14
15     public boolean validateDataRequest(Object request){
16         boolean isValid = false;
17
18         DataExchange dataExchange = new DataExchange();
19         InputStream in = dataExchange.getCollectedDataAsStream();
20
21         // validate 'request' using 'in'
22
23         return isValid;
24     }
25 }
26
```

APPENDIX B

First a structure to store the layer data,

```
class SimpleLayer {
    private String name;
    private String server;
    private String client;

    public SimpleLayer(String name, String server, String client) {
        this.name = name;
        this.server = server;
        this.client = client;
    }

    public String getName() {
        return name;
    }

    public String getServer() {
        return server;
    }

    public String getClient() {
        return client;
    }

    @Override
    public String toString() {
        return "SimpleLayer{" +
            "name='" + name + '\'' +
            ", server='" + server + '\'' +
            ", client='" + client + '\'' +
            '}';
    }
}
```

Then to extract the prescriptive architecture.

```
static Map<String, SimpleLayer> layerRulesMap = new HashMap<String,
SimpleLayer>();

public static void processRules() throws IOException, ParseException {
    JSONParser parser = new JSONParser();

    //Use JSONObject for simple JSON and JSONArray for array of JSON.
    JSONObject data = (JSONObject) parser.parse(
        new FileReader("/path/to/simple_layer_access.json")); //path
to the JSON file.

    String json = data.toJSONString();

    JSONArray msg = (JSONArray) data.get("layers");
    Iterator<JSONObject> iterator = msg.iterator();
    while (iterator.hasNext()) {
        JSONObject layerJson = iterator.next();
        String layerName = (String) layerJson.get("name");
        String serverLayer = (String) layerJson.get("server");
        String clientLayer = (String) layerJson.get("client");

        SimpleLayer layer = new SimpleLayer(layerName, serverLayer,
clientLayer);
        layerRulesMap.put(layerName, layer);
    }
}
```

For a simple layered system we can create a doclet with below details.

```
static Map<String, SimpleLayer> layerRulesMap = new HashMap<String,
SimpleLayer>();

    public static boolean start (RootDoc root) throws IOException,
ParseException {

        processRules ();

        processLayers (root.specifiedPackages ());

        return true;

    }
```

```

private static void processLayers(PackageDoc[] packages) {
    for (PackageDoc layer : packages) {
        String nameSplit[] = layer.name().split("\\.");
        String layerName = nameSplit[nameSplit.length - 1];
        System.out.println("\nAnalyzing classes of " + layerName + "
layer");
        SimpleLayer layerRule = layerRulesMap.get(layerName);
        ClassDoc[] classes = layer.allClasses();
        boolean isPass = true;
        for (ClassDoc classDoc : classes) {
            isPass = true;
            Tag[] serverTags = classDoc.tags("sic.server");
            Tag[] clientTags = classDoc.tags("sic.client");
            for (Tag tag : serverTags) {
                if (tag.text().length() > 0 && !tag.text().equals
(layerRule.getServer())) {
                    System.out.println("\t" + classDoc.name()
+ " violates the layered architecture by
accessing " + tag.text() + " as a client");
                    isPass = false;
                }
            }
            for (Tag tag : clientTags) {
                if (tag.text().length() > 0 && !tag.text().equals
(layerRule.getClient())) {
                    System.out.println("\t" + classDoc.name()
+ " violates the layered architecture by
accessing " + tag.text() + " as a server");
                    isPass = false;
                }
            }
            if (isPass) {
                System.out.println("\tClass " + classDoc.name() + "
PASSED");
            }
        }
    }
}

```

```

public static void processRules() throws IOException, ParseException {

    JSONParser parser = new JSONParser();

    //Use JSONObject for simple JSON and JSONArray for array of
JSON.

    JSONObject data = (JSONObject) parser.parse(

        new FileReader
("/home/vidudaya/vnbandara/MSc/Project/SicAlpha/src/main/resources/sicr
ules/simple_layer_access.json")); //path to the JSON file.

    String json = data.toJSONString();

    JSONArray msg = (JSONArray) data.get("layers");
    Iterator<JSONObject> iterator = msg.iterator();
    while (iterator.hasNext()) {

        JSONObject layerJson = iterator.next();

        String layerName = (String) layerJson.get("name");
        String serverLayer = (String) layerJson.get("server");
        String clientLayer = (String) layerJson.get("client");

        SimpleLayer layer = new SimpleLayer(layerName, serverLayer,
clientLayer);

        layerRulesMap.put(layerName, layer);

    }
}

```