CONTROLLING SOFTWARE ARCHITECTURE EROSION TO SUPPORT MAINTAINABILY

Dissertation submitted in fulfilment of the requirements for the Degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

PALLAVIT SHARMA

11207607

Supervisor

MOHIT ARORA

School of Computer Science and Engineering (14 Bold)

Lovely Professional University

Phagwara, Punjab (India)

June 2017

ABSTRACT

Software Architecture Erosion is a problem faced by many organization in the software industry. It happens when 'as-implemented' architecture does not conform to the 'as-intended' architecture, which result in low quality, complex, hard to maintain software. Architecture erosion makes the software system more complex, more prone to errors and less maintainable. In this I have created two architecture one is based on spring MVC and hibernate framework and the other is simple java code. Based on design principles such as separation of concerns, single responsibility principle, principle of least knowledge I will compare the two architecture with the help of tools i.e. SonarQube and JArchitect in order to find the cyclic-dependencies and architectural violations between the two architectures. This will help us to make the system more reliable and maintainable. In this we will then remove cyclic-dependencies, code smells, vulnerabilities, and violations from source code by modifying the source code of software system.

DECLARATION STATEMENT

I hereby declare that the research work reported in the dissertation entitled "CONTROLLING SOFTWARE ARCHITECTURE EROSION TO SUPPORT MAINTAINABIILITY" in partial fulfilment of the requirement for the award of Degree for Master of Technology in Computer Science and Engineering at Lovely Professional University, Phagwara, Punjab is an authentic work carried out under supervision of my research supervisor Mr Mohit Arora. I have not submitted this work elsewhere for any degree or diploma.

I understand that the work presented herewith is in direct compliance with Lovely Professional University's Policy on plagiarism, intellectual property rights, and highest standards of moral and ethical conduct. Therefore, to the best of my knowledge, the content of this dissertation represents authentic and honest research effort conducted, in its entirety, by me. I am fully responsible for the contents of my dissertation work.

Pallavit Sharma

11207607

SUPERVISOR'S CERTIFICATE

This is to certify that the work reported in the M.Tech Dissertation entitled "**CONTROLLING SOFTWARE ARCHITECTURE EROSION TO SUPPORT MAINTAINABIILITY"**, submitted by **Pallavit sharma** at **Lovely Professional University, Phagwara, India** is a bonafide record of his original work carried out under my supervision. This work has not been submitted elsewhere for any other degree.

Signature of Supervisor

Mohit Arora

Date:

Counter Signed by:

1) **Concerned HOD:**
   HoD's Signature: _____

   HoD Name: _____

   Date: _____

2) **Neutral Examiners:**

   External Examiner

   Signature: _____

   Name: _____

   Affiliation: _____

   Date: _____

   Internal Examiner

   Signature: _____

   Name: _____

   Date: _____

ACKNOWLEDGEMENT

It is not until you undertake research like this one that you realize how massive the effort it really is, or how much you must rely upon the selfless efforts and goodwill of others. I want to thank them all from the core of my heart.

I owe special words of thanks to my supervisor Mr. Mohit Arora for his vision, thoughtful counselling and encouragement for this research on **"Controlling Software Architecture Erosion to support Maintainability".** I am also thankful to the teachers of the department for giving me the best knowledge guidance throughout the study of this research.

And last but not the least, I find no words to acknowledge the financial assistance & moral support rendered by my parents and moral support given by my friends in making the effort a success. All this has become reality because of their blessings and above all by the grace of almighty.

Pallavit Sharma

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Checklist for Dissertation-III Supervisor

Name: _____ UID: _____ Domain: _____
Registration No: _____Name of student: _____

Title of Dissertation:

_____

☐ Front pages are as per the format.
☐ Topic on the PAC form and title page are same.
☐ Front page numbers are in roman and for report, it is like 1, 2, 3…….
☐ TOC, List of Figures, etc. are matching with the actual page numbers in the report.
☐ Font, Font Size, Margins, line Spacing, Alignment, etc. are as per the guidelines.
☐ Color prints are used for images and implementation snapshots.
☐ Captions and citations are provided for all the figures, tables etc. and are numbered and center aligned.
☐ All the equations used in the report are numbered.
☐ Citations are provided for all the references.
☐ **Objectives are clearly defined.**
☐ Minimum total number of pages of report is 50.
☐ Minimum references in report are 30.

Here by, I declare that I had verified the above mentioned points in the final dissertation report.

Signature of Supervisor with UID

CHAPTER 1
INTRODUCTION

Software architecture is the major structures of a software system, and it pact with how numerous software processes uphold to accomplish their functions or tasks. The architecture of a system describe its extensive constituents, their interconnections, and how they work together with each other, In other words we can say that, Architecture work as a model for a system. It works a detachment to control the system complications and settle an intercommunication and collocation mechanism amongst constituents. It specify a framed clarification to meet all the technological and operative requirements, while reform the similar factor property like performance and security. It is about the architecture of programming concentrated frameworks, characterized as ―any framework where software contributes basic impacts to the outline, development, sending, and advancement of the framework overall.

Software architecture consists a set of guidelines to expand, establish as well as drilling a software framework for a specific assignment. It characterizes an extremely up level design of enormous software systems also the far reaching architecture of a system in a perfect in addition to methodical way. The primary target of a software architectural perspective of system is to sort the real parts of a system, to recognize the relationships amongst the segments, components as well as describe them in like manner. These alleged components are gathered collected by connectors that provide the match or connection amongst various segments [2]. The connectors can thusly be segments themselves. In general sense software architecture is a specific way in the design and improvement of programming. It contains groupings of choices which relies on upon incalculable angles in a wide scope of software improvement. Each of these choices can have intense ampules on the overall accomplishment of the product. One of the real issues in software systems development today is quality. A quality trait is a non-functional characteristic of a component or a framework. ISO/IEC 9126-1 [17] characterizes a product quality model. As indicated by this definition, there are six classifications of attributes: usefulness, unwavering quality, convenience, effectiveness, viability, and versatility.

1.1 Objectives of designing software architecture

The objectives [20] of software architecture is to classify the prerequisites which will influence the structure of the application. Goals for designing software architecture are as follows:

Table 1.1: Objectives of software structural design

Goals	Description
Platform independent	Software structural design should not rely on upon a suitable equipment stage. This will run programming on any implanted frameworks or PCs with slightest conceivable details oblige.
Hardware modularity	Hardware segments must be part in little areas and might other intercommunicate to one another via a wired or remote medium. Software design might determine tenets and system for intercommunication among different equipment parts. This will give system with capacity of highlight expanding without much exertion.
Increased productivity	Structure of programming ought to be very much portrayed, so it will be simpler to include distinctive elements.
Code maintainability	Code ought to be compatible and all around organized so it will be less demanding to channel and oversee.
Testability	Organized software ought to give an all-around characterized function interfaces to end client, this facilitates testability of a specific component.
Investigate and Debug	It is additionally simple to distinguish bugs or escape clauses inside an all-around organized and modular code. Software architectures ought to coordinate analytic and troubleshoot highlights while outlining a product framework. In this way end client can straightforwardly connect with modules for demonstrative elements.
Simplicity	The support and usage of the software architecture must be in the easy to use way.

Appropriate for outsized team	Every designer or analyzer ought to have the capacity to take a shot at free modules parallel. This is conceivable because of measured and organized nature of software framework characterized by software architecture.
Availability	It characterizes the bit of time framework is useful and working. It can be uniform as the rate of the aggregate framework downtime over predefined period. It is influenced by framework bugs, association issues, fiendish assaults and framework stack.
Security	A system's capacity to manage fiendish assault from outer or inner of the system.
Performance	Expanding framework's proficiency as to reaction time, throughput, asset usage and characteristics which typically strife with each other
Lifetime	The time period of the item it is full of life formerly its retirement
Scalability	It is ability of system to deal with an expansion in system stack
Concurrency	It is the property of system in which a few errands are implementing together and communicating with one another. These assignments might be implementing on the different centers in a similar chip
Cost	The cost of building, keeping up and working the system.
Usability	Convenience incorporates matter of fulfillment of clients from utilizing the system.

The product architecture is a standout amongst the most basic articles inside the life expectancy of a product system. Choices made at the design level have a straight influence on the accomplishment of business objectives, functional as well as quality necessities [19]. It comprises of arrangement of choices which relies on upon many calculates an extensive variety of software advancement. Each of these choices can have huge effect on the general achievement of the software.

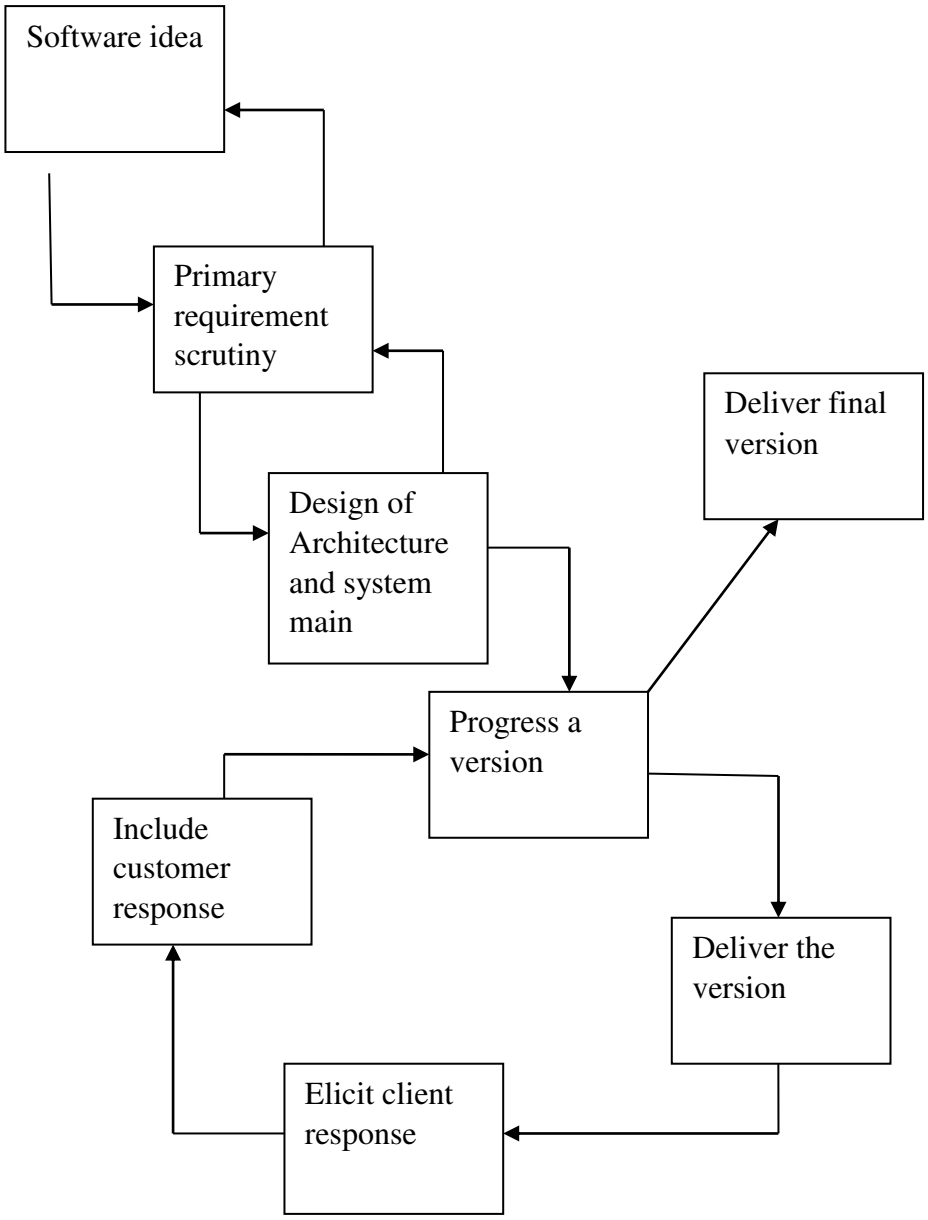


Figure 1.1: Software architecture lifecycle

1.2 Software frameworks

A product structure is a strong or theoretical stage where ordinary code with bland usefulness can be particularly specific or superseded by engineers or customers. Structures show up as libraries, where a very much characterized portrayed application program interface (API) is reusable wherever inside the product a work in progress [18]. Software structure comprises of frozen spots and hot spots. Frozen spots define the general architecture of a product framework, in other words its fundamental segments and the connection between them. These stay unaltered (frozen) in any

instantiation of the application structure. Hot spots signify those parts where the programmers using the framework add their own code to add the functionality particular to their personal project. Framework is an application that is finished aside from the real usefulness, you connect to the usefulness and you have an application, they are extremely valuable to designers. It contains the entire thing you have to make an application. In reality you can frequently insignificantly make an ostensible application with not very many lines of source that does literally nothing yet it gives you window administration, sub-window administration, set of decisions, catch bars, and so forth.

Certain elements make a framework unique in relation to other library forms, including the following:

- **Default Behavior:** Before customization, a framework demonstrations in a path specific to the specialist's activity.
- **Inversion of Control:** Not at all like different libraries, the general stream of control inside a structure is locked in by the system as opposed to the guest.
- **Extensibility**: A client can grow the framework by specifically substituting default code with client code.
- **Non-modifiable Framework Code**: A client can grow the framework however not change the code. The inspiration driving software framework is to make less difficult the advancement condition, enabling planners to devote their endeavors to the venture necessities, instead of managing the system's commonplace, repetitive capacities and Libraries.

In an object-oriented environment, a structure comprises of unique and substantial classes. Instantiation of such a framework includes shaping and sub-classing the present classes. When working up a solid software framework with a software framework, creators utilize the problem areas as per the particular requirements and necessities of the system. Software framework depend upon the Hollywood Principle: "Don’t call us, we’ll call you". This suggests the client characterized classes (for instance, novel subclasses), get messages from the predefined structure classes. Architects by and large handle this by executing superclass dynamic techniques. While structures for the most part allude to wide programming advancement stages, the term can likewise be utilized to depict a particular system inside a bigger software design condition. While structures usually allude to wide programming improvement stages,

the term can likewise be utilized to depict a precise system inside a bigger programming condition. For instance, numerous Java frameworks.

Architecture Core Activities

- Architecturally significant requirements.
- Architecturally Analysis.
- Architecturally Synthesis.
- Architecturally evaluation.

There are four basic exercises in software architecture plan. These fundamental engineering exercises are accomplished iteratively and at various phases of the early software improvement life-cycle, and also over the advancement of a framework:

- **Architectural Analysis** is the strategy of understanding the earth in which an arranged system or system will work together and deciding the prerequisites for the system**.**
- **Architectural Synthesis** or design is the way toward making architecture. Given the necessities controlled by the examination, the present condition of the outline and the results of any assessment exercises, the plan is made and made strides.
- **Architecture Evolution** is the procedure of maintaining and adjusting current programming design to meet necessity and ecological changes. As software design gives a critical structure of software system, its advancement and upkeep would basically affect its essential structure. Thusly, design advancement is worried with including novel usefulness and also keeping up current usefulness and framework execution. Architecture requires unsafe supporting exercises. These supporting exercises occur all through the centre software architecture process. They comprise of information administration and correspondence, plan thinking and result making, and documentation.
- **Architecture supporting activities** Software architecture supporting activities are agreed out through basic software architecture exercises. These supporting exercises bolster a product modeler to finish examination, union, evaluation and movement. For example, a draftsman needs to assemble information, settle on decisions and report amid the investigation stage.

There are frameworks that cover specific territories of use headway, for instance, JavaScript/CSS systems that aim the presentation (view) layer of the application, and there are others that handle a more noteworthy measure of the active parts of the presentation. Instances of structures that are starting at now offered by benchmarks bodies or associations incorporate:

- **Resource Explanation Framework**, an arrangement of rules from the World Wide Web Association for in what way to characterize some Internet asset, for example, a Web webpage in addition to its substance.
- **Internet Business Framework**, a gathering of programs that frame the mechanical establishment for the SAP item from SAP, the German organization that business sectors a venture asset administration line of items.
- **Sender Policy Framework**, a well-defined methodology as well as programming for making e-mail additional safe and sound.

Advantages

- Recycle code that has been pre-developed and pre-tried. Supports the dependability of the novel application and lessen the programming and testing exertion, and time to advertise.
- A framework can help develop better programming hones notwithstanding suitable utilization of configuration examples and new programming apparatuses.
- A framework can give new usefulness, better execution, or better quality without extra programming by the framework user.
- By definition, a system offers you with the way to extend its conduct.

Disadvantages

- Creating a framework is hard as well as timewasting (i.e. expensive).
- The learning arc for a novel framework can be sharp.
- In excess of time, a framework can come to be more and more difficult.
- Structures often enhance to the magnitude of programs, a phenomenon named “code bloat”.

1.3 Architecture patterns and styles

An architectural pattern is set of guidelines that form an application. It consolidates design for instance, customer/server, service-oriented architecture (SOA), section based design, layered architecture as well as message bus architecture. The design styles for application is picked in perspective of outline, key models, real advantages, and data. These styles portray diverse parts of utilizations. Some engineering styles characterize arrangement designs, some characterize structure and configuration issues, and others define communication factors. The applications as a rule utilize a mix of more than one of the style.

Table 1.2: Architectural styles

Category	Architecture styles
Communication	Service-Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain Driven Design
Structure	Component-Based, Object-Oriented, Layered Architecture

1.3.1 Client/Server architectural style

Client-server architecture concentrates on administrations distinctive customers need to perform. This architecture is particularly fit when the equipment is sorted out as various nearby PCs (e.g. individual workstations) and one central asset, for example, a record tree, database, or a group of effective central computation PCs..

A two level structural pattern is a customer/server structure that has an application which is situated at the server side that is gotten to specifically by various customers. The client/server architecture style clears up the association amongst a client and the servers which can change from one to various or we can express that, in a product customer-server system, there may be a couple of clients in one PC, and even the server can continue running on a comparative PC.

The limits of two level customer/server style of design is, penchant for application information and in addition computational rationale to be firmly consolidated on the server, which can unfavourably influence framework extensibility, versatility notwithstanding its reliance on a focal server, which additionally thusly influences framework unwavering quality undesirably. For conquering these limits, the

customer-server architecture style changed into a further broad 3-Tier structural style likewise perceived as (N-Tier) style of architecture. The main benefits of client/server architectural style are [20]:

- **Higher security**. All information is put away on the server, which for the most part offers a more prominent controller of security than customer machines.
- **Centralized data access**. Since data is put away just on the server, get to and updates to the data are far calm to direct than in other architectural styles.
- **Ease of maintenance**. Parts and also duties of a computing system are scattered among a few servers that are perceived to each other through a system. This guarantees a customer stays uninformed and furthermore unaffected by a server repair, overhaul, or movement.

1.3.2 N-Tier / 3-Tier architectural style

N-level/3-level are architectural style give insights about the partition of functionality into different portions. Each piece is a level which can be originate on particular single PCs. N-level/3-level style is created by methods for the segment based method, being a substitute for message-based method. It incorporates the utilitarian disintegration of administration parts, their applications and appropriated arrangement. Each level is detached or have his own autonomous presence from every single other level present, with the exception of quick above and underneath. Advantages of the N-level/3-level structural style are adaptability, versatility, openness plus practicality.

1.3.3 Component-Based Architectural Style

Component-based architecture defines [22] a way to deal with system design and improvement. It involves the deterioration of the design into unmistakable utilitarian or coherent segments. These constituents bring out all around characterized correspondence gauges containing strategies, properties notwithstanding occasions. It gives a higher measure of deliberation contrasted with question arranged outline techniques. It doesn't concentrate on issues, for example, correspondence conventions and shared state.

A huge standard of the component-based style is the use of segments that are recyclable, free, epitomized, not setting particular and in addition extensible. The conventional sorts of constituents incorporate lattices, catches, colleague, utility capacity and lined part. The key advantages of the part established building style are

simplicity of the organization, cheap price, simplicity of the advancement, recycle ability as well as lightening of specialized multifaceted nature. Component-based architecture style is measured to make an intricate design, that permit to effortlessly supplant notwithstanding refresh singular segments. The main principle [20] of the component-based style is the use of components that are:

- **Reusable.** Segments are usually intended to be reused in divergent situations in various applications. In any case, a few constituents might be intended for a specific errand**.**
- **Replaceable.** Segments might be promptly supplanted with other alike segments.
- **Not setting particular.** Segments are intended to work in various conditions and in addition settings. Point by point data, for example, state information, ought to be passed to the part as opposed to being fused in or gotten to by the segment**.**
- **Extensible.** A part can be extended from existing segments to give new conduct.
- **Encapsulated.** Parts uncover interfaces that enable the guest to use its usefulness, notwithstanding not uncover data of the inside procedures or any inward factors or state.
- **Independent.** Segments are intended to have minor conditions on different parts. In this way segments can be sent into any appropriate condition without influencing different components or systems.

The following are the main benefits of the component-based architectural style:

- **Ease of arrangement.** As new perfect forms wind up noticeably available, you can substitute existing adaptations with no impact on alternate segments or the system overall.
- **Reduced cost.** The use of outsider segments grants you to spread the cost of improvement and in addition upkeep.
- **Ease of advancement.** Segments actualize surely understood interfaces to offer characterized usefulness, allowing improvement without affecting different parts of the system.
- **Reusable.** The utilization of reusable segments implies that they can be utilized to spread the advancement notwithstanding support taken a toll over a few applications or system.
- **Mitigation of specialized many-sided quality.** Segments ease multifaceted nature through the utilization of a part compartment and its administrations. Illustration

part benefits contains segment initiation, lifetime administration, strategy lining, eventing, and also exchanges.

1.3.4 Domain Driven Design (DDD)

Domain Driven Design (DDD) [23] remains an object-oriented methodology. It is built up on the structure area, its parts, the mode it works, plus the connections among them. It intends to empower software structures that are an acknowledgment of the hidden area by characterizing a space show passed on in the dialect of framework area specialists. The space display legitimizes the arrangement. The fundamental backings of the Domain Driven Design style are testability, extensibility and correspondence. The following are the main benefits of the Domain Driven Design style:

- **Communication.** All gatherings inside an advancement group can utilize the area display notwithstanding the elements it characterizes to convey business learning and prerequisites with a typical business space dialect, without requiring specialized language.
- **Extensible.** The area model is now and then particular and adaptable, making it simple to raise to date and reach out as conditions and necessities change**.**
- **Testable.** The space demonstrate items are generally coupled and strong, allowing them to be all the more effortlessly tested.

1.3.5 Object-Oriented Architectural Style

The object oriented architecture [24] style is an arrangement of outline standards in software development that spotlights on separating a system into individual and reusable parts, or protests. Questions regularly comprise of information fields and in addition methodology. Objects are normally occasions of classes, and a program can be ponder to be a gathering of items associating with each other. This is as opposed to them or regular procedural programming where a program is more like a rundown of subroutines. Articles are partitioned, autonomous that are daintily coupled. They speak with each other by means of interfaces, technique calls, and sending and accepting messages.

The normal uses of the protest arranged style grasp characterizing a question model that backings complex logical operations and true antiquities inside a system domain. The key principles of the object-oriented architectural style are:

- **Abstraction**. This lets you to decrease a mind boggling operation into a speculation that keeps the base attributes of the process. For instance, a active interface can be a perceived as the description that backings information get to operations with straightforward strategies, for example, Get and Update. Another type of deliberation could be metadata used to give a mapping between two configurations that hold organized information.
- **Composition.** Items can be assembled from different questions, and can conceal these interior articles from different classes or uncover them as basic interfaces.
- **Inheritance**. Items can get from different protests, and utilize usefulness in the base question or overrule it to execute new conduct. Besides, legacy makes support likewise refreshes less demanding, as changes to the base protest are spread naturally to the acquiring objects.
- **Encapsulation.** Objects uncovered usefulness just through techniques, properties, and occasions, and conceal the center points of interest, for example, state and factors from different articles. This makes it less demanding to refresh or substitute items, the length of their interfaces are perfect, without influencing different objects in addition to code.
- **Polymorphism.** This enables you to overrule the conduct of a base sort that backings operations in your application by executing new sorts that are interchangeable with the existing object.
- **Decoupling.** Articles can be decoupled from the end client by characterizing a theoretical interface that the protest executes and the end client can get it. This lets you to give another executions without influencing clients of the interface.

The main benefits of the object-oriented architectural style are that it is:

- **Understandable.** It maps the application promote nearly to this present reality objects, making it more fathomable.
- **Reusable**. It offers for reusability through polymorphism and additionally deliberation.
- **Testable.** It conveys for upgraded testability through exemplification.

- **Extensible**. Exemplification, polymorphism, and reflection affirms that an adjustment in the portrayal of data does not influence the interfaces that the question uncovered, which would limits the ability to impart and additionally communicate with different items.

- **Highly Cohesive.** By finding just related techniques and components in a question, and utilizing disparate items for various arrangements of elements, you can finish an abnormal state of attachment.

1.3.6 Layered architecture style

This architectural style [25] is finest suited for presentations that incorporate particular classes of administrations which can be organized progressively. It stresses on gathering the connected functionality inside an application hooked on unmistakable conspicuous layers that are fixed vertically on top of one another. Every layer is practically interrelated through other by a typical part. The communication amongst layers is express also delicately coupled. This architectural style is an altered pyramid of recycle where every layer adds up to the duties as well as reflections of the layer specifically underneath it.

This architectural pattern is comprehensively used to model message-passing circumstances. This style know how to be utilized to structure various sorts of programming frameworks. The non-parallelism can without much of a stretch be portrayed by a layered model. On the off chance that there is parallelism of parts, we can place the parallel procedures in single layer. The layered engineering style is likewise suitable for unpredictable and configurable framework issues. Regular standards for plans that utilization of the layered structural style include:

- **Abstraction.** Layered structural design abstracts the perspective of the framework as entire while sufficiently giving subtle element to comprehend the parts and obligations of individual layers and the connection between them.

- **Encapsulation.** No supposition should be made about information sorts, techniques as well as properties, or usage amid plan, as these components are not uncovered at layer limits.

- **Visibly characterized functional layers**. The division amongst usefulness in every layer is flawless. Higher layers, for example, the introduction layer send guidelines to inferior layers, for example, the business as well as information layers, and may

respond to occasions in these layers, enabling information to stream together up and additionally down between the layers.

- **High cohesion**. Very much characterized duty confinements for each layer, and guaranteeing that each layer incorporates usefulness straightforwardly identified with the undertakings of that layer, will boost union inside the layer.
- **Reusable.** Bring down layers have no conditions on higher layers, conceivably allowing them to be reusable in further situations.
- **Loose coupling.** Correspondence between layers depends on deliberation and occasions to give free coupling between layers.

The core welfares of the layered structural design style are:

- **Abstraction.** Layers enable alterations to be made at the theoretical level. You can upsurge or reduction the level of reflection you use in each layer of the progressive stack.
- **Isolation.** Grants you to segregate innovation moves up to individual layers keeping in mind the end goal to limit chance and limit effect on the general framework.
- **Manageability.** Partition of centre concerns helps to distinguish conditions, and sorts out the code into more sensible segments.
- **Performance.** Conveying the layers over numerous physical levels can enhance versatility, adaptation to internal failure, and execution.
- **Reusability.** Parts advance reusability. For instance, in MVC, the Controller can frequently be reused with other perfect Views keeping in mind the end goal to give a part particular or a client altered view on to similar information and usefulness.
- **Testability.** Enhanced testability emerges from having very much characterized layer interfaces, and in addition the capacity to switch among various usage of the layer interfaces. Isolated Presentation designs allow you to fabricate taunt protests that copy the conduct of solid questions, for example, the Model, Controller, or View amid testing.

1.4 Design principles of software architecture

Design is important to all product building exercises and is the focal incorporating action that ties the others together [21]. The key rule that will manufacture an architecture that obeys to demonstrated standards, limits costs and in addition support necessities, and advances ease of use and extendibility. The key principles are:

- **Separation of concerns**. Isolate your application into particular components with as pitiful cover in handiness as would be reasonable. The imperative component is minimization of c interaction focuses to fulfil high connection and low coupling. Nevertheless, segregating usefulness at the wrong limits can achieve high coupling and versatile quality between parts in spite of the way that the contained functionality inside a component.

- **Single Responsibility standard.** Each segment ought to be in charge of just a specific element or functionality, or accumulation of bound together usefulness.

- **Principle of Least Knowledge (otherwise called the Law of Demeter or LoD).** A segment or question ought not to think about interior parts of different segments or protests.

- **Don't repeat yourself (DRY**). You ought to just need to determine expectation in one place. For instance, as distant as application plan, specific functionality have to to be actualized in just single fragment; the functionality have not to be repeated in other part.

- **Reduce upfront design**. Just outline what is fundamental. At times, you might need upfront exhaustive outline notwithstanding testing if the cost of improvement or a dissatisfaction in the plan is in elevation. In unlike circumstances, particularly for flexible improvement, you can keep away from big design upfront (BDUF). On the off chance that your application prerequisites are uncertain, or if there is a likelihood of the plan advancing after some time, withdraw from endeavouring recklessly. This principle is sometimes known as YAGNI ("You ain’t gonna need it").

When arranging a system, the aim of a product modeller is to confine the many-sided quality by secluding the plan into different zones of concern. For example, the user interface (UI), business dealing with, and data get to all address different area of concern. Inside each locale, the segments you configuration focus on that specific area

and ought not to blend code from various regions of concern. The going with anomalous state principles will consider the broad assortment of segments that can impact the effortlessness of arranging, completing, and passing on, testing, as well as keeping up your application:

Design principles

- **Keep configuration designs predictable inside each layer.** Inside a sensible layer, where possible, the plan of segments should be dependable for a specific operation. For instance, in the event that you select to utilize the Table Data Gateway example to make a protest that goes about as a passage approach to tables or perspectives in a database, you ought not hold onto another example, for example, Repository, which utilizes a not an indistinguishable worldview for getting to information from well as instating business elements.
- **Do not copy functionality inside an application.** There should be just a single segment giving a specific usefulness—this usefulness ought not to be reproduced in whatever other constituent. This makes your segments strong and in addition makes it less demanding to improve the segments if a specific component or usefulness change. Duplication of usefulness inside an application can roll out it intense to actualize improvements, fall in clearness, notwithstanding present potential irregularities.
- **Prefer composition to inheritance.** Wherever conceivable, utilize piece over legacy while reusing usefulness since legacy rises the reliance amongst parent and kid classes, along these lines controlling the reuse of youngster classes. This additionally decreases the legacy chains of importance, which can turn out to be extremely hard to manage.
- **Establish a coding style and naming tradition for improvement.** Verify whether the association has built up coding style and in addition naming norms. If not, you should build up regular norms. This offers a steady model that makes it less demanding for colleagues to look at code they didn't compose, which prompts better practicality.
- **Maintain system quality utilizing mechanized QA systems amid improvement.** Utilize unit testing and other computerized Quality Analysis systems, for example, reliance examination and static code investigation, amid advance. Depict clear behavioral and execution measurements for segments and sub-frameworks, additionally utilize computerized QA apparatuses for the time of the manufacture

procedure to guarantee that nearby outline or usage decisions don't unfavorably influence the general system quality.

- **Consider the operation of your application**. Figure out what measurements and agent information are required by the IT foundation to affirm the effective organization notwithstanding operation of your application. Planning your application's parts and sub-frameworks with a reasonable comprehension of their different operational prerequisites will fundamentally ease add up to organization and operation. Utilization of automated QA instruments for the time of advancement to guarantee that the right operational information is conveyed by your application's parts and sub-systems.

Application layers

- **Isolate the areas of concern.** Breakdown your application into specific elements that cover in usefulness as meager as possibly will be normal the situation being what it is. The essential favorable position of this approach is that a segment or usefulness can be streamlined unreservedly of various components or usefulness. What's more, in the event that one component comes up short, it won't make different elements bomb too, and they can run unreservedly of each other. This approach furthermore makes the application less requesting to grasp and arrange, and empowers organization of complex dependent frameworks.
- **Be explicit about how layers communicate with each other**. Allowing each layer in an application to be connected with or have endless supply of alternate layers will bring about an answer that is all the more empowering to comprehend and also oversee. Make unequivocal choices about the conditions between layers and additionally the information stream between them.
- **Use abstraction to execute free coupling between layers**. This can be accomplished by characterizing interface segments, for example, an exterior with prestigious contributions to expansion to yields that make an interpretation of solicitations into an organization comprehended by segments inside the layer. Furthermore, you can likewise utilize Interface sorts or dynamic base classes to characterize a typical interface or shared deliberation (reliance reversal) that must be executed by interface segments.
- **Do not blend diverse sorts of segments in the same logical layer.** Begin by distinguishing diverse zones of concern, and after that gathering parts related with

every range of worry into sensible layers. For instance, the UI layer ought not to contain business preparing segments, but rather ought to contain parts used to deal with client contribution to expansion to process client demands.

- **Keep the information arrange reliable inside a layer or part**. Blending information configurations will make the application more hazardous to execute, augment, and maintain. Each and every time you have to change information starting with one configuration then onto the next, you are essential to execute interpretation code to accomplish the operation and bring about a preparing overhead.

1.5 Software architecture erosion

Software architecture erosion or disintegration is determined as the aspect that happens when the executed design of a product framework go amiss from its normal design. The actualized architecture is the model that has been executed or worked in inside low-level plan builds and the source code. The term solid design additionally alludes to the implemented architecture [26]. The proposed architecture is the result of the design configuration handle, otherwise called the conceptual architecture [26] or the arranged design. The deviation itself is not caused by venomous human activities yet rather by routine repair and change work normal for a developing programming framework.

The general impact of structural design choices as well as the exchange offs between individual qualities is inspected by a planner. The software architecture ought to just speak to the structure of the framework by concealing the execution subtle elements in addition to controlling both the quality trait as well as the practical necessity.

During the lifespan of some usual software system it experiences advancement plus making of various prescriptive and clear architecture at various circumstances. On the off chance that a man doesn't have sufficient learning about what the actualized and proposed engineering is then the likelihood of the event [30] of software disintegration turns high.

The impact of architecture disintegration causes the dis-fulfilment of partner's prerequisites as the progressions wind up noticeably hard to utilize on the product and in the most exceedingly bad, it can even prompt disappointment of programming

undertakings. Practically every other venture experiences disintegration at some stage in software improvement cycle unless some exertion is done to conquer it.

In another normally referred to case of design disintegration, Godfrey and Lee [27] depict their examination of the extricated models of the Mozilla web program (which in this manner developed into Firefox) and the VIM content manager. Both these product items demonstrated a substantial number of undesirable interdependencies among their center subsystems. Actually, the gravely dissolved design of Mozilla caused noteworthy postponements in the arrival of the item and constrained engineers to revise some of its center modules starting with core modules from scratch.

Architecture disintegration causes numerous bugs in programming, for example, increment in inner unpredictability with the expansion of new usefulness, developing time to change the product and time-to-market, diminished quality, expanding the test exertion for upkeep of programming and so on in the meantime lessening the designer's efficiency as much period is spent on understanding the difficult present portions of the software. The final product is: costs rises and efficiency falls.

The contextual analysis of Garlan and Ockerbloom [28] portray various issues (e.g., extreme code, poor execution, need to alter outer bundles, need to rethink existing usefulness, superfluously confounded devices) caused by structural befuddle that hampered effective reuse. Each of these issues requires alterations of the product with the danger of digressing from the planned architecture. Software architecture disintegration can be limited by consistence checking. Design consistence as a measure to which degree the executed engineering in the source code fits in with the arranged design (i.e., a consistence of 1.0 or 100% implies that there are no building infringement, 0.0 or 0% the inverse). The figuring partitions the quantity of agreeable conditions by the aggregate number of conditions between segments. Engineering consistence checking is the way to gauge this. The consistence of the design can be checked statically (i.e. without executing the code) and powerfully (i.e. run time). The three main static architecture compliance checking approaches of a system are:

- **Reflexion models:** Reflexion models think about two models of a product framework against each other, normally, a compositional model (the arranged or proposed design) and a source code show (the genuine or actualized engineering). The correlation requires a mapping between the two models to be thought about, which is a human-based assignment.

- **Relation conformance rules**: Relation conformance rules empower indicating permitted or illegal relations between two parts. They can distinguish comparable deformities as reflexion models, yet the mapping is done naturally for every conformance check.
- **Component access rules:** Component get to rules empower indicating basic ports for segments, which different parts are permitted to call. These guidelines help to expand the data stowing away of parts on a level, which won't not be conceivable inside the physical architecture of the implementation language.

It turns out to be yet further costly when software disintegration brings about a "product avalanche", when the measure of disintegration achieves a point where the product can't be kept up or enhanced any further and modify turns into the main arrangement, with all the utilized expenses and dangers. As the circumstance gets most exceedingly terrible, the main conceivable alternative remain is to fabricate the product sans preparation or at the end of the day "Rework" the product however this choice is extraordinarily expensive and dangerous with respect to due dates or spending plan. As revising includes the new programming to accomplish the majority of the usefulness that the current programming have hence no time is left to make a change in the product putting both the venture and association on stake. That is the reason the product, and for the most part its engineering, must have the capacity to manage various solicitations for change to forever remain in working condition.

Architecture disintegration prompts the steady of the engineering quality of software systems. With the end goal of this overview we characterize designing quality as a subsumption of compositional honesty (i.e. fulfillment, rightness and consistency), conformance to quality characteristic prerequisites, and selection of sound programming building standards. Building nature of a framework may not generally liken to the nature of system performance. A well-performing system may have a seriously eroded architecture. Be that as it may, such a framework is amazingly delicate and has a high danger of separating at whatever point alterations are made..

1.5.1 Types of Software Architecture Erosion

The most common types of software architectural disintegration consists of:

- **Architectural Rule violations-** For re-architecting or encourage advancement a few plan principles ought to be taken after e.g. maintain a strategic distance from strict layering between subsystems

- **Unreachable Code-** It is also known as the dead code which is not ever executed nor required for any utilization but rather it is as yet messing the code base contributes towards structural disintegration.
- **'Copy & Paste' Codes-** In spite of the fact that code duplication is well known with the end goal of reuse and execution productivity as duplicate glue is the most well-known strategy yet as the size builds the practicality cost increments, for example, a settling a blunder or alteration in one clone case is probably going to must be dispersed to the next clone illustrations.
- **Metric outliers**- Include further class progressive systems, huge bundles and complex code.
- **Dependency-** Between bundles and modules lessens reusability, blocks upkeep, anticipates extensibility, constrain testability and limits a designer ability to comprehend the results of progress**.**
- **Cyclic Dependencies-** Are the most exceedingly terrible kind of disintegration. Phases tend to snitch into plan. For example, if A and B are put in an alpha bundle, and one is set in a various bundle, a cyclic reliance amongst alpha and numbers exists despite the fact that the class structure is a cyclic. They ought to be repaired or promptly wiped out as they wind up in delicate code.

1.5.2 Symptoms of software architecture erosion

There are certain side effects that show disintegration [29] in architectural designs. They are:

- **Inflexibility-** It rolls out the product hard to improvement as a change can cause infringement in ward modules hence surpassing an opportunity to play out that change, along these lines the administrator's dread so much that they in the long run deny to permit any adjustments in software.
- **Brittleness-** It is firmly identified with rigidity making the product crack each time it is altered henceforth the administrator's dread that the product will burst in some unforeseen way at whatever point they endorse a fix driving towards exorbitant improve. Such programming is not reasonable to keep up as they turn out to be most exceedingly bad as each alteration and bug settle takes significantly more. In such cases, the designers lose the control on their product and it turns out to be truly difficult for them to work through such software in addition to there is compel to revamp the software.

- **Serenity-** It is the inability to recycle segments from identical or diverse software ventures as the vast majority of the product includes much comparative kind of modules composed by different designers. Tranquility shows up when the engineers discover that the work and hazard important to part the needed parts of the product from the undesirable parts are too huge to acknowledge thus the software is just modified rather than reused.
- **Reduced Effectiveness and Efficiency-** because of deferral in software discharges, spending overwhelms, quality bugs and so on.

1.5.3 Threat of software architecture erosion

There are a few danger connected by programming design for instance in the improvement group where novel contracts might not comprehend the framework as well as old representatives need to buckle down, not contradicting the anxiety, brings about high turnover which is a reason for structural consumption as the learning of engineering is lost when they assent. So also unbendable software is another issue as it is extremely hard to improve and grow it. Quite possibly an alteration can even origin a presentation of novel bug in software along these lines the product must be exceedingly viable or repairable. The advancement group additionally has not a steady association with the product's life as there is a probability that some part can left the group plus the learning of the architecture and software related with them likewise vanishes. Software Architecture disintegrates increasingly when new contracts commit errors and set aside much opportunity to catch up the venture promptly. Engineering will additionally disintegrate when the new contracts sufficiently lacking learning about design would try to make changes to the framework.

1.5.4 Real-time example of architecture erosion

Lately, numerous genuine runtime cases of software structural design disintegration have been perceived. Architecture erosion shows up when the system progresses toward becoming disapprove and glutted. The illustration Describes that simple software was made in March 2004 containing just 4 bundles, couple of months after the fact new elements were included it was all the while going fine however in May 2005 a first cyclic reliance showed up, in June 2006 another recurring reliance was watched and in the end in 2009 the product was encompassed by many interweaves. This venture can't be effortlessly repaired or looked after at this point. The case resolve that the structural design was faultless at the beginning, in the wake of making a few

adjustments it was all the while running great yet with time its structure was debased with the presentation of reliance subsequently a man can't really prevent the disintegration from getting into place however measures should be possible to battle against it by expel it to some development.

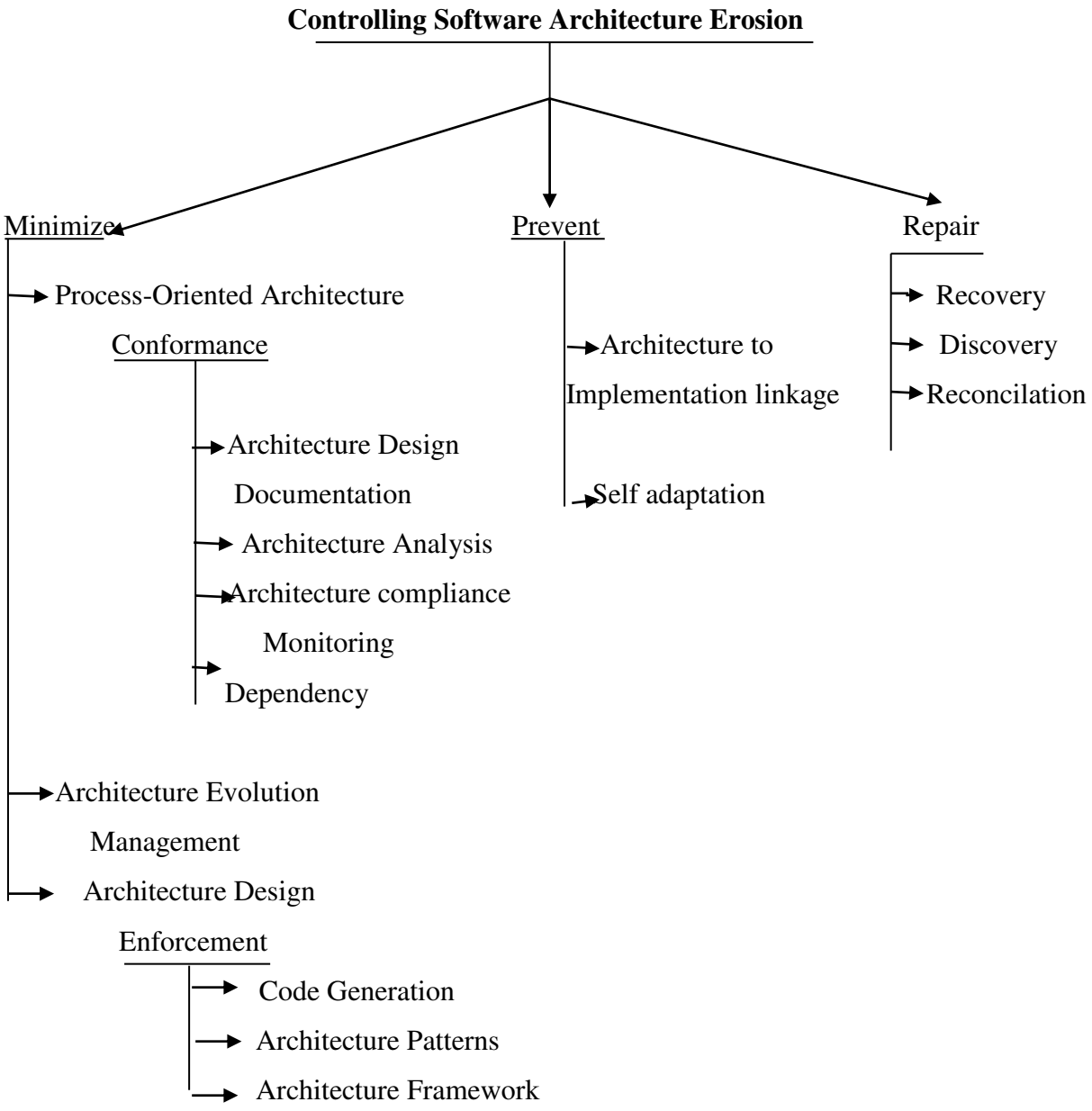


Figure 1.2 controlling software architecture erosion

CHAPTER 2
REVIEW OF LITERATURE

Palmen et.al (2011), "A Systemic Methodology for Software Architecture Analysis and Design" proposes that another systemic strategy for examination and plan of architecture design that addresses some real constraints in the present best in class. It present another product engineering strategy that methodologies the product design area with systemic procedures and we demonstrate expressly relevant considers software architecture definition. It present the idea of example for the relevant condition, which serve close by engineering designs as the essential vocabulary for architecture depiction and examination. Our investigation approach utilizes a probabilistic demonstrating and choice formalism to guide software architecture evolution. Patterns are a fitting devices for measuring the dynamic ideas appropriate to architecture, for example, "calculated uprightness" and "wellness for reason" as opposed to breaking down those architectural concept to a smaller scale level determination with the end goal of estimation and control [1].

De Silva and Perera (2015), "Preventing Software Architecture Erosion through Static Conformance Checking" propose some of apparatuses which help to overcome from issue of disintegration. The apparatus which is used to overcome from this issue is static engineering conformance checking device. Design conformance checking suggests assessing the comparability of the executed plan to the proposed engineering and can give a framework to perceiving software architecture disintegration and prevent its negative results. This apparatus identify building limitations pollutions and along these lines help to maintain a strategic distance from the product plan disintegration. This device depends on GRASP ADL. Handle is a printed engineering portrayal dialect fit for taking the "justification" behind building outline decisions. It underpins an arrangement of compositional parts for catching the engineering of framework. E.g.: System Element Layer Element, Component Element, Link, Element, Connector Element, and Interface Element [3].

Terra, Valente et.al (2012), "Recommending Refactoring's to Reverse Software Architecture Erosion" recommends the essential plan of a proposal framework whose rule explanation behind existing is to give refactoring principles to engineers and maintainers during the undertaking of turning around a building breaking

down process. The paper formally delineates the approach which offers proposals to evacuate architectural infringement recognized by DCL (Dependency Constraint Language) which is space particular dialect and is utilized for depicting the auxiliary limitations among the modules in programming framework. This dialect is straightforward, simple reasonable sentence structure. Conformance apparatus is utilized which is dclcheck and dclfix [4].

Kamran Sartipi (2003)," Software Architecture Recovery based on Pattern Matching" recommends an pattern-based recuperation philosophy whose goals can be determined as far as the structural properties that are very much characterized through a architectural pattern. The proposed architectural pattern is fixated on architecture description dialects (ADLs) and is incrementally made by means of an intuitive strategy that permits to incorporate the learning from the application domain and system documentation. The result of the recuperation can be specifically tried in opposition to the recuperation destinations through: conformance checking with the available documentation that affirms the decay of the fundamental framework usefulness into segments, deciding the particularity nature of the recouped design to affirm the recuperation of a viable framework, conformance with the segment and connector size and sort limitations do by the example [5].

Pruijt and Brinkkemper (2013), "Architecture Compliance Checking of Semantically Rich Modular Architectures" propose the Architecture Compliance Checking (ACC) is a way to deal with confirm he conformance of executed program code to abnormal state models of engineering outline. ACC is utilized to forestall building disintegration amid the advancement and development of a software system. Static ACC, in view of static programming examination strategies, concentrates on the secluded design and particularly on tenets compelling the measured components. A semantically rich modular design (SRMA) is expressive and may contain modules with various semantics, similar to layers and subsystems, compelled by standards of various sorts. To check the conformance to a SRMA, ACC-apparatuses ought to bolster the module and manage sorts utilized by the draftsman. This paper presents necessities in regards to SRMA bolster and a stock of normal module and lead sorts, on which premise eight business and non-business apparatuses were tried [6].

De Silva and Balasubramaniam (2012), "Controlling Software Architecture Erosion" recommends the methodologies and advancements that have been proposed throughout the year to control software architecture disintegration or to distinguish and

re-establish design that have been dissolved. These methodologies incorporate the instruments, systems and procedures, which are grouped into three classifications viz., limit, counteract and repair design disintegration. These procedures are as per the following: process-oriented architecture conformance, architecture design enforcement, architecture evolution management, self-adaptation, architecture to implementation linkage, and architecture restoration techniques consisting of recovery, discovery and reconciliation [7].

Chanda and Liu (2015), "Intelligent Analysis of Software Architecture Rationale for Collaborative Software Design" proposes that the gathering of partners trading their perspectives with a specific end goal to settle on plan choice since architecture rationale behind different outline choices is not completely caught and henceforth influences the practicality of software system. Keeping in mind the end goal to catch and keep up the Software Architecture rationale for examination a keen software architecture rationale basis catch framework has been composed that enables distinctive member or partners to take an interest in an online discourse to determine configuration issue cooperatively. This paper utilizes three distinct methodologies. Right off the bat, we decide aggregate assessments of a gathering on various perspectives and recognize perspectives which have picked up a huge consideration into the online exchange. Besides, propose a technique to build up a traceability network that connections different software architecture components to its related software necessities. Thirdly, we perform printed investigation of partners' perspectives to decide the points that are generally talked about [8].

Caracciolo, Lungu et.al (2015), "A Unified Approach to Architecture Conformance Checking" suggests that software architecture disintegration can be controlled by documenting design decision and using architecture description language (ADLs) techniques [9].

Maqbool and Babri (2004), "Bayesian Learning for software Architecture Recovery" proposes the utilization of Bayesian learning technique for automatic recovery of as software system's architecture, given fragmented or obsolete documentation. In this utilize programming modules with known characterizations to prepare the Naive Bayes classifier. After this we utilize the classifier to put new cases, i.e. Software modules, into appropriate sub-systems. At that point they will assess the execution of the classifier by directing investigations on a product framework, and contrast the outcomes got and a manually prepared architecture. To thinking and

finishing up results Bayesian learning utilizes the likelihood based approach. Bayesian learning strategies depend on Bayes hypothesis. One of the Bayesian learning method is Naïve Bayes classifier that has been successfully applied to solve many problems like text classification, speech/image recognition [10].

Dargomir and Lichter (2012), "Model-based Software Architectural Evolution and Evaluation" proposes a automatic assessment of software architecture, intended to bolster the architecture based advancement of software systems at different deliberation levels. It gives recoup, imagine and assess the software architecture of a system. This paper comprises of two noteworthy objectives: **Architecture monitoring and representation:** It starts with mapping the source code artefacts of a software system to architecture components beginning from the system's architecture view, by labelling the source code as needs be. **Architecture evolution and assessment:** It address the advancement and assessment of the architecture of a software system. The trigger of any advancement is change ask. The change demand can be caused, e.g., by the expansion of new prerequisites or by the choice to structurally re-consider worsened parts of the system [11].

Budi et.al (2010), "Automated Detection of Likely design Flaws in Layered Architecture" suggests that Layered Architecture is a good principle for separating concerns to make systems more maintainable. One example of layered architecture is a separation of classes into three groups boundary, control, and entity these are refer to as three analysis class stereotypes in UML. When the classes of different stereotypes interact with each other, and properly design, the overall system would be maintainable flexible and robust. Whereas poor design results in less maintainable system which is more prone to errors. It provides a frame work which can automatically labelled classes as boundary, control or entity, and detect design flows of rules associated with each stereotype. There are two common rules variants: - Robustness rule and well-formedness rules [12].

Herold, Counell et.al (2015) "Detection of Violent Causes in Reflection Model" describes that Reflection Model is a technique used to detect architecture violation that occurs during software architecture erosion. In this abnormal state architecture is characterized by the designer which contains box and bolts. At that point designer will characterize the mapping of abnormal state architecture to genuine source code then tool analyse the source code and it will produce the Reflection Model. This paper focuses on the technique which will help us to automatically detect the causes of

violation of reflection model and detection of typical symptoms for these causes. Reflection model tools JITTAC. Common causes for violations in a reflection model: Architecturally misplaced software units and Call-back. Symptoms for violation causes are: Structural Symptoms (It basically describe the structural pattern in reflection model, the mapping and the code) and Measurable Symptoms (It describes the quantifiable properties of the elements in structural symptoms) [13].

Herold et.al (2013), "Checking Conformance with Reference Architectures" this paper shows a report about the use of rule based architecture conformance checking approach which examine a industrial reference for the German open organization. The limitations for usage of reference architecture are formalized as architecture guidelines empowering programmed conformance checking device bolster. The above approach can recognize and keep away from design disintegration if connected over the product life cycle. Way to deal with design conformance checking are: (an) indicating the reference engineering as a meta-model to such an extent that the engineering of a framework can be demonstrated and (b) formalization of building elements required to be checked for conformance as intelligent expressions [14].

Eyck, Helleboogh et.al (), "Using code analysis tools for architectural conformance checking" In this paper, we examine a couple of code examination devices that can be utilized to check static conformance of a system to its architecture that offer support for Java and consider their abilities for compositional conformance checking: Architecture Rules, Macker, Lattix DSM, SonarJ, Structure101 and XDepend. Checking software architecture conformance is vital to keep that the framework erodes after some time and to defend the quality. At the point when a system is executed or transformed, it is difficult to evaluate whether the real implementation conforms in with the design. Architecture conformance checking is the verification whether a framework conforms in with its planned architecture, which is crucial to protect the quality characters of the system. A few methodologies exist to support architecture conformance checking. One way to deal with accomplish conformance is to combine an Architectural Description Language (ADL) general-purpose programming dialect. A case of this approach for the Java programming dialect is Arch-Java. Since engineering elements are top of the line segments in the tongue and fill in as a starting point for usage, design conformance is approved by the lingo itself. Along these lines compositional information is safeguarded inside the code, be that as it may it requires the utilization of a committed dialect to fabricate the framework. [15].

Knodel, Popescu (2007), "A comparison of Static Architecture Compliance Checking Approaches" This paper separate three static architecture consistence checking approaches reflexion models, relation conformance guidelines, and component access rules by looking over their pertinence in 13 specific measurements. The three fundamental static engineering consistence checking methodologies of a framework: Reflexion models: Reflexion models partners two models of a product structure against each other, regularly, an outline model (the orchestrated or arranged plan) and a source code display (the genuine or executed design). Relation Conformance Rules: it empowers determining allowed or denied relations between two sections. They can recognize practically identical deformations as reflexion models, yet the mapping is done naturally for each conformance check. Component Access Rules: empower determining straightforward ports for segments, which diverse parts are allowed to call. These guidelines help to grow the data concealing without end of segments on a level, which won't not be possible inside the physical architecture of the execution dialect [16].

CHAPTER 3
PRESENT WORK

This report gives a comprehensive review of the architectural, utilizing various diverse architectural perspectives to portray distinctive parts of the framework. It is proposed to catch and convey on the critical architectural decisions which have been made on the system.

It likewise exhibits a review of strategies and innovations that have been proposed throughout the years either to anticipate architecture disintegration or to distinguish and restore architectures that have been eroded. These methodologies, which incorporate apparatuses, procedures and techniques, are basically grouped into three non-specific classes that endeavour to limit, counteract and repair architecture disintegration. Inside these general classifications, each approach is additionally separated mirroring the abnormal state techniques embraced to handle disintegration. These are: process-architecture conformance, architecture evolution management, engineering outline authorization, architecture design enforcement, architecture to implementation linkage, self-adaptation and architecture restoration techniques consisting of recovery, discovery and reconciliation. Some of these procedures contain sub-classes under which study results are exhibited. We talk about the benefits and shortcomings of every technique and contend that no single methodology can address the issue of disintegration. Further, we investigate the likelihood of joining methodologies and present a case for further work in building up a comprehensive structure for controlling architecture disintegration.

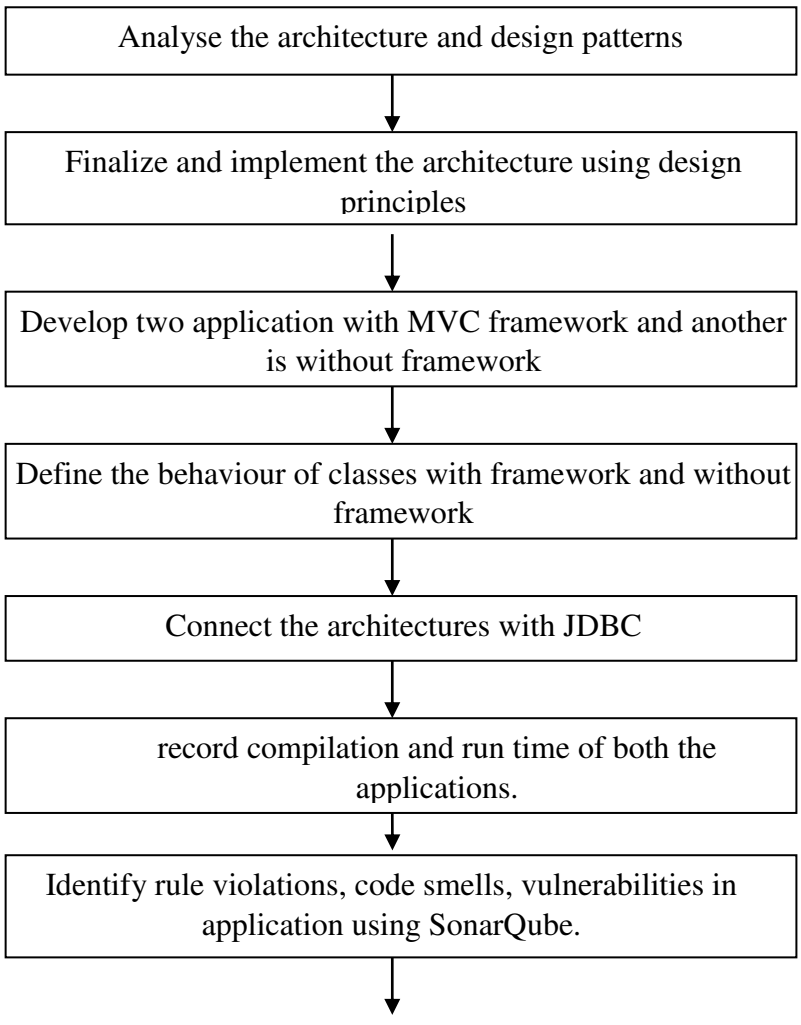
3.1 Problem formulation

By reviewing number of research papers, we concluded some of the important aspects of software architecture problems which occurs regularly in software development. There are some problems which we identified which are architectural smells, code smells, code duplication, technical debt and possible relations of similar types of problem which leads to Architectural problems. Here we are focusing about the code smells, technical debt and code duplication majorly and which can be solved with the help of through refactoring of code, placing proper design principles and design patterns. To yield maximum architecture resources and software designers must

estimate the size of technical debt. This technical debt can be handled by framework also, so that developer can write the code in better environment.

Following is the research methodology:

1 Analyze architecture and design the patterns.
2 Finalize and implement the architecture based on design principles and patterns.
3 Develop the one application without framework and one with spring MVC framework.
4 Define the behavior of the classes to check the dependencies and violations rules.
5 Connect the implemented architectures with the JDBC and record their compilation and run time.
6 Validate the compliance of both the application by checking violations of rules, validate configuration with the help of tools SonarQube and JArchitect.
7 Check for code smells, vulnerabilities, number of issues in application, technical debt using SonarQube and JArchitect.
8 Remove the issues manually in order to make application less prone to errors so that new version of application can be produced.



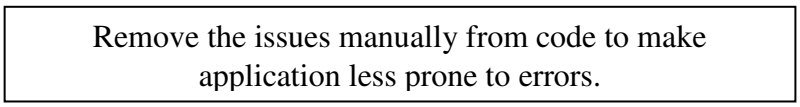


Figure 3.1: Flowchart of methodology

3.2 Objectives of the study

Architectural level of understanding is crucial for software specifically when the structure is to be reformed to meet varying requirements. In case of the old or legacy software system, architectural documentation is not available, an attempt must be made to recuperate the architecture from the source code.

- Our main objective is to rectifying an eroded software system architecture by aligning the implemented architecture with the proposed architecture or planned architecture. This process involves the support of architecture discovery (it is used to build the planned architecture from system requirements, documents specifications and system use cases) and recovery (used for recovering the implemented architecture). To recover we will use some tools which will help to obtain the executed architecture from the source code or source artefacts.
- To reduce the complexity of system in order to increase maintainability, security as well as performance of software system.

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Experimental results

In order to control software architecture disintegration or erosion I have developed two projects. One is developed using Spring MVC (Model-View-Controller) framework and another one is developed with simple java without beans. The project "SpringMVCHibernateCRUD" is developed using spring MVC framework and follows all the design principle such as Separation of concerns, single entity responsibility and principle of least knowledge. The other project "Project-2" is developed without using design principles and design patterns.

These two projects are developed in eclipse. Both the projects consists of an interface in which user will enter some of its basic information as shown in the figures given below. These projects consists of four different classes UserController.java (contains the servlets), UserDao.java (contains the logic for database operation), User.java (contains the POJO (Plain Old Java Object)), Database.java (contains the class for initiating database connection). The project "SpringMVCHibernateCRUD" which consists of Spring MVC framework is developed using Maven. Apache Maven is a product project administration and understanding tool. In view of the idea of a project object mode (POM), Maven can deal with a project's build, announcing and documentation from a focal snippet of data. Maven takes care of managing dependencies, developing a deployable component, runs application in Tomcat, creating sample report.

The comparison of these projects are done with the help of SonarQube and JArchitect. The SonarQube is an open source tool for quality administration and is committed to persistently examining and measuring the technical quality of source code, from project portfolio down to the technique level, and following the introduction of new Bugs, Vulnerabilities, and Code Smells in the Leak Period. Bugs are code that is more expected not providing the intended behaviour. Examples such as null-pointer dereferences, memory leakages, and logic errors. Code smells are smelly codes which are difficult to maintain and introduce bugs. For example, complex code, duplicate code, codes which are not covered by unit testers. SonarQube helps to find and track

the security Vulnerability in code. For example, SQL injections, passwords, badly managed errors.

JArchitect is a java static analysis and code quality tool. JArchitect assist in a large amount of code metrics, permits for visualization of dependencies with the help of directed graphs and dependency matrix. User can also write custom rules and query code by using CQLinq (Code Query over LINQ), calculates Technical debt, checks for Quality Gates, Issues Management, generate custom reports, explore existing architecture, harness test coverage data.

Figure 4.1: Java code without beans

The figure 4.1 is the architecture which is based on java code which does not consists of beans and spring MVC framework.

Figure 4.2: Java code without beans output

Figure 4.3: Controller class code

The figure 4.3 shows the code for the controller class which controls the data taken as input from the user and putting the input into the database.

Figure 4.4: Spring MVC and hibernate output

The Spring Web MVC system gives Model-View-Controller (MVC) engineering in addition to prepared parts that can be utilized to construct adaptable as well as approximately coupled web applications. The figure 4.4 is based on spring MVC framework and consists of beans. The objects that shape the foundation of your application and that are overseen by the Spring IoC holder are called beans. A bean is a protest that is instantiated, unite, and generally overseen by a Spring IoC holder. These beans are made with the setup metadata that you supply to the container. For example, in the form of XML <bean/>.

Figure 4.5: Spring MVC framework database

The figure 4.5 shows the output of the spring MVC framework in MySQL Workbench. This shows that how the values are entering into the database.

Figure 4.6: Sonarqube output

Figure 4.6 shows the rating of the four different projects. The Project-2 which is developed without any architecture pattern and does not follow any design principle shows 'E' rating in reliability, 'B' rating in security and 'A' rating in maintainability. Where the project SpringMVCHibernateCRUD which is developed using MVC (Model-View-Controller) architecture pattern and follows design principles shows 'A' rating in reliability, 'A' rating in security and 'A' rating in maintainability.

Figure 4.7: Spring MVC output (SonarQube)

Figure 4.7 shows the number of vulnerabilities, bugs, code smells, and duplicated blocks in SpringMVCHibernateCRUD project which is developed by using MVC architecture pattern and follows design principles. It consists of 4 code smells and also shows the technical debt to remove the code smells.

Figure 4.8: Spring MVC output (SonarQube)

Figure 4.8 shows the issues in the given project and effort needed to remove single issue and code smell.

Figure 4.9: Spring MVC output (SonarQube)

Figure 4.9 shows the location where the error occurs. It will tell us the package and the class where the code smell will occur and also tell us whether it is minor or major.

Figure 4.10: Spring MVC output (SonarQube)

Figure 4.11: Spring MVC output (SonarQube)

Figure 4.12: Spring MVC output (SonarQube)

Figure 4.13: Project-2 output (SonarQube)

Figure 4.14: Project-2 output (SonarQube)

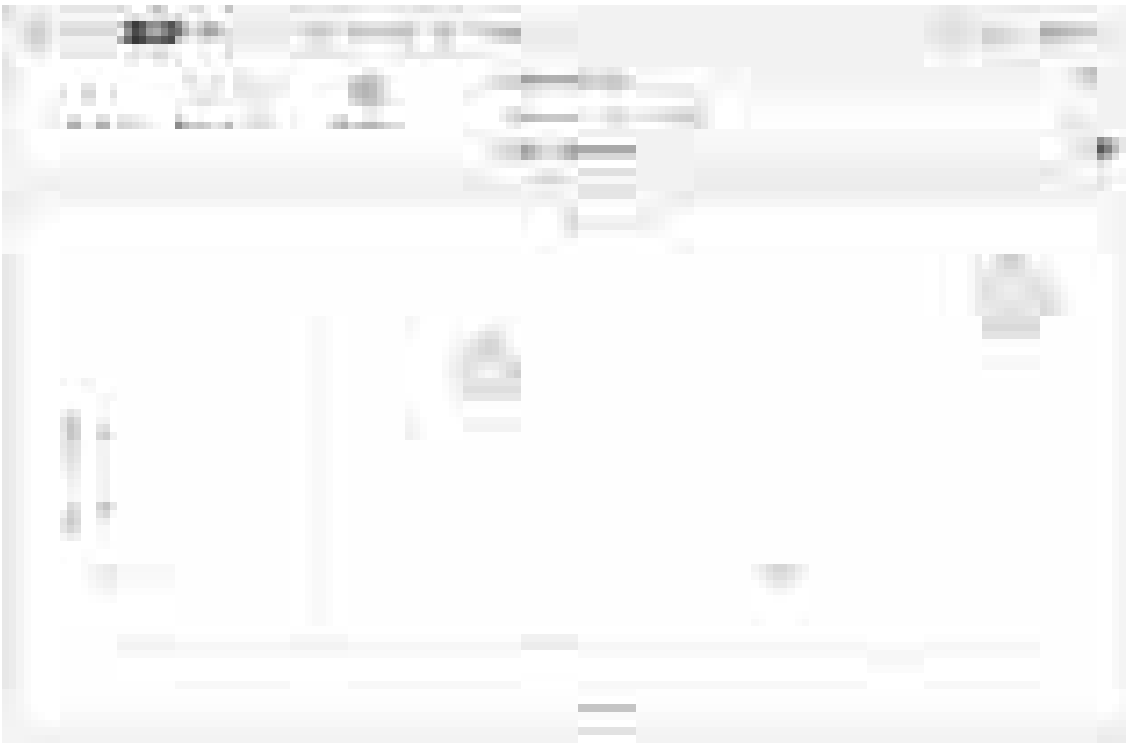


Figure 4.15: Project-2 output (SonarQube)

Figure 4.16: Project-2 output (SonarQube)

Figure 4.17: Spring MVC output (JArchitect)

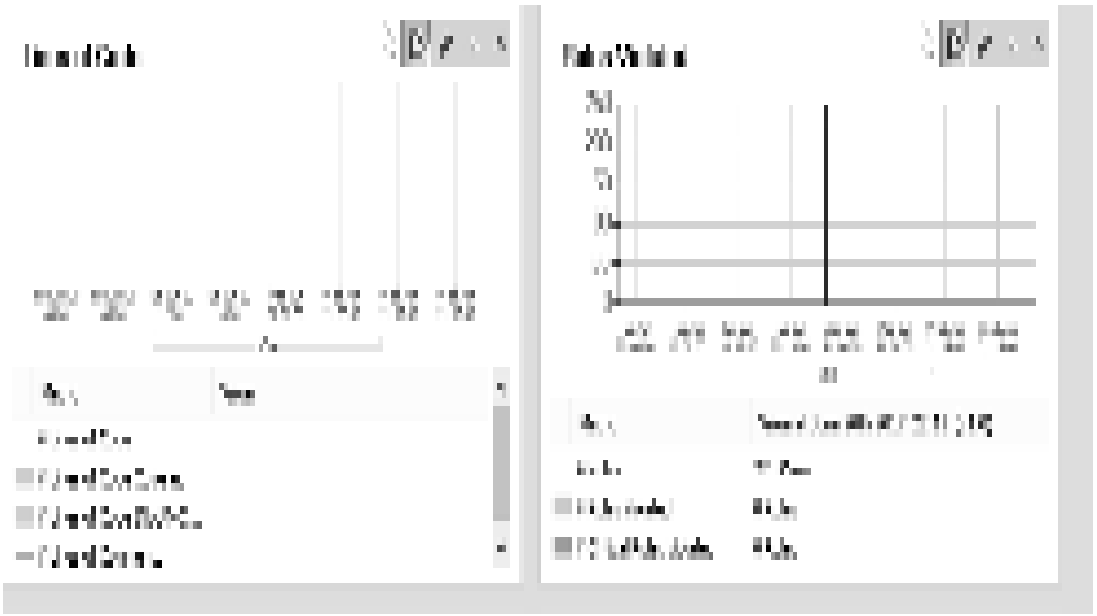


Figure 4.18: Spring MVC output (JArchitect)

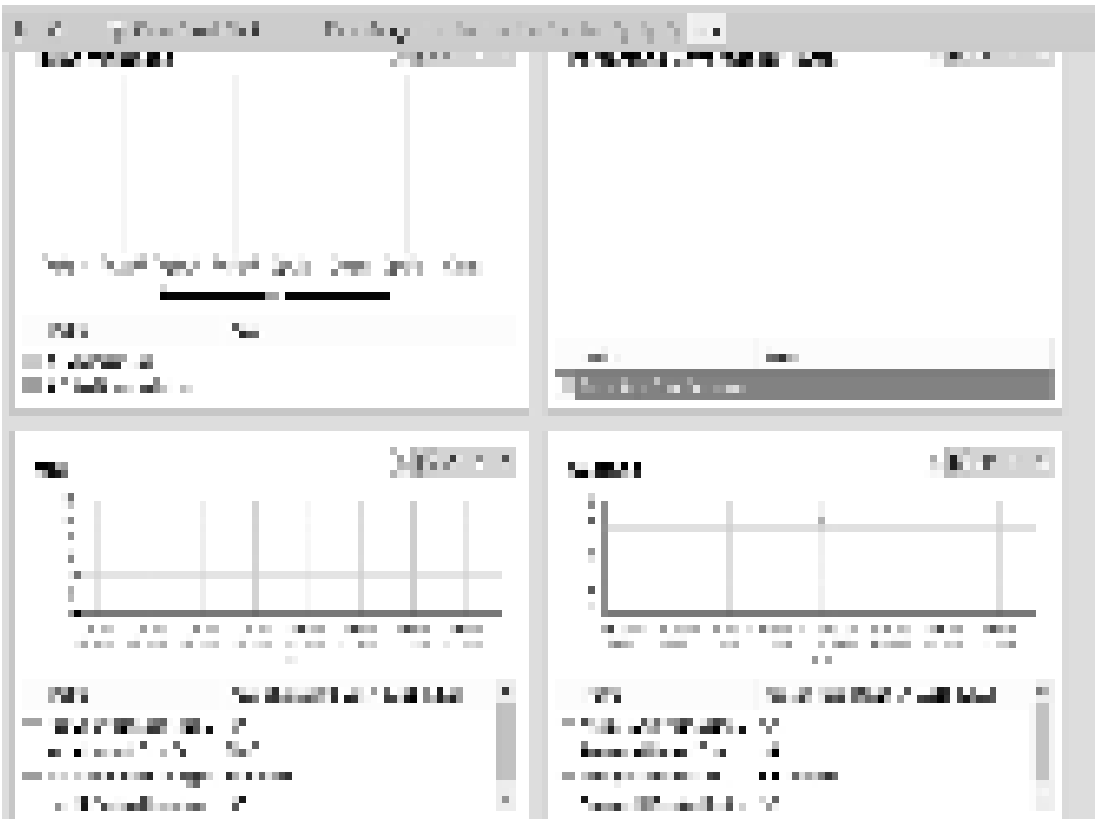


Figure 4.19: Spring MVC output (JArchitect)

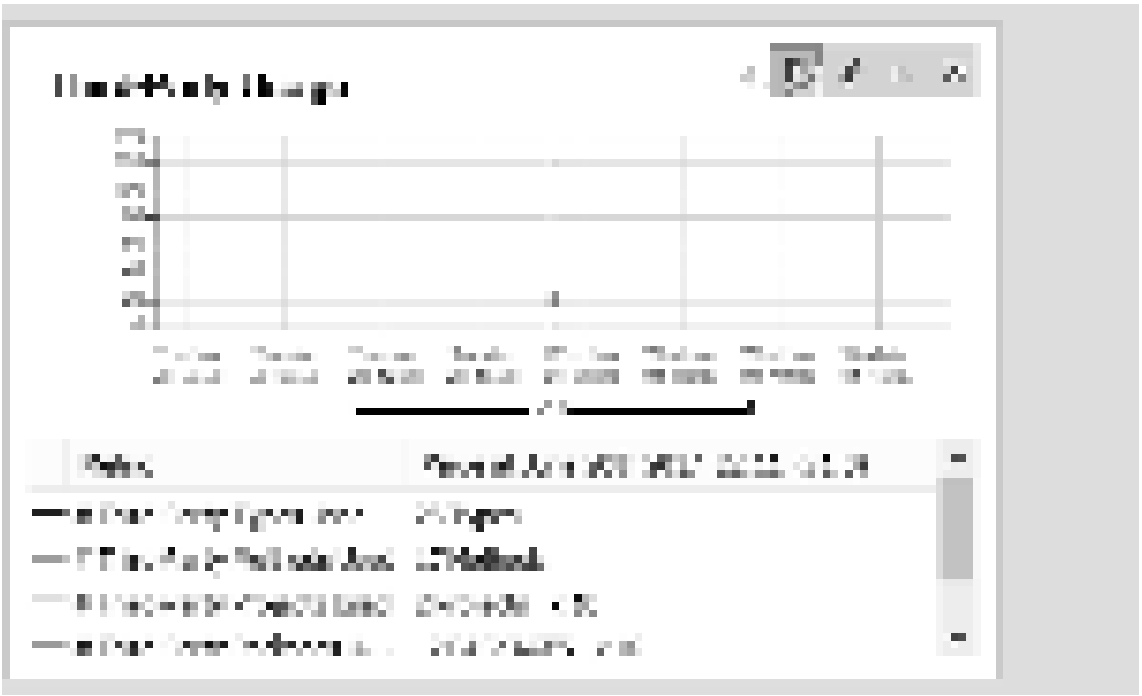


Figure 4.20: Spring MVC output (JArchitect)

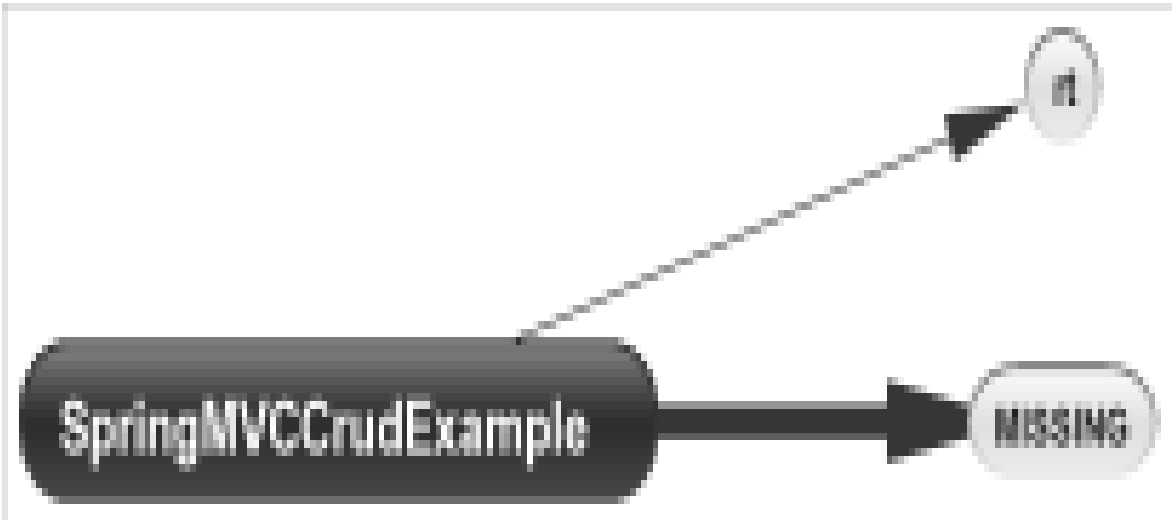


Figure 4.21: Dependency graph (JArchitect)

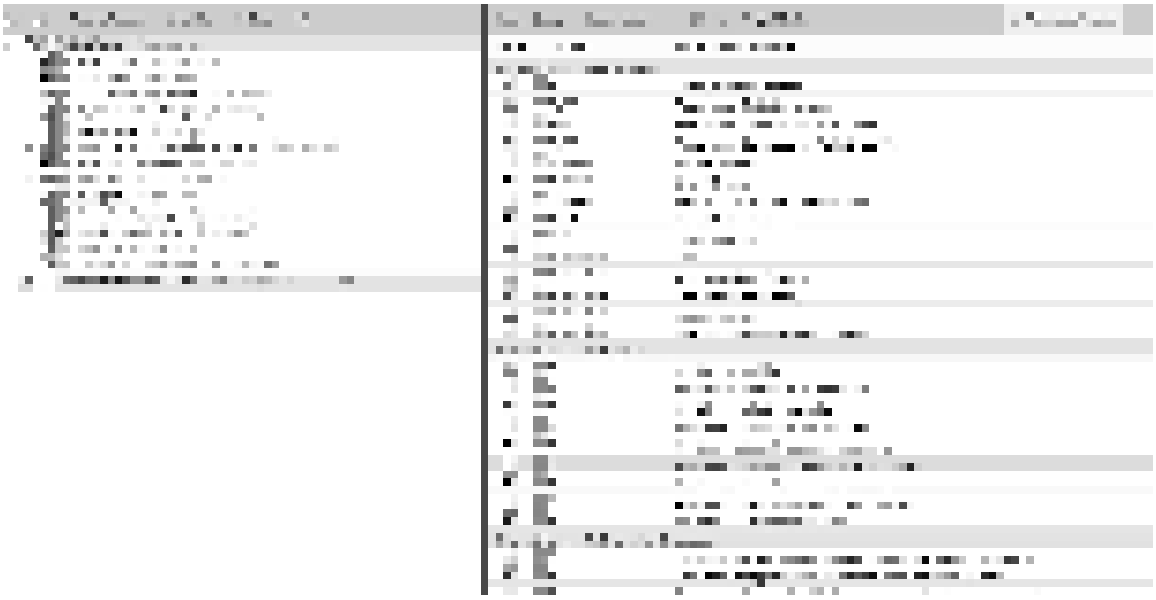


Figure 4.22: Code query and rules (JArchitect)

Figure 4.23: Dependency matrix (JArchitect)

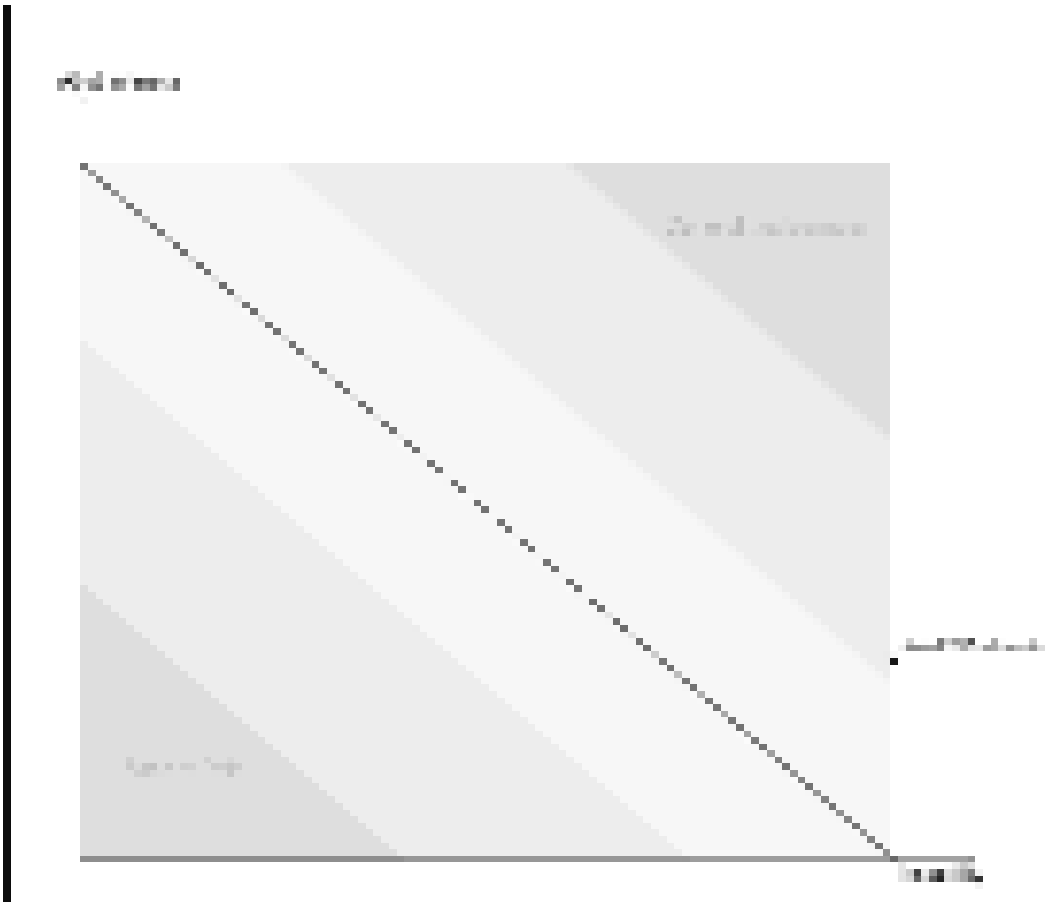


Figure 4.24: Abstractness vs. Instability (JArchitect)

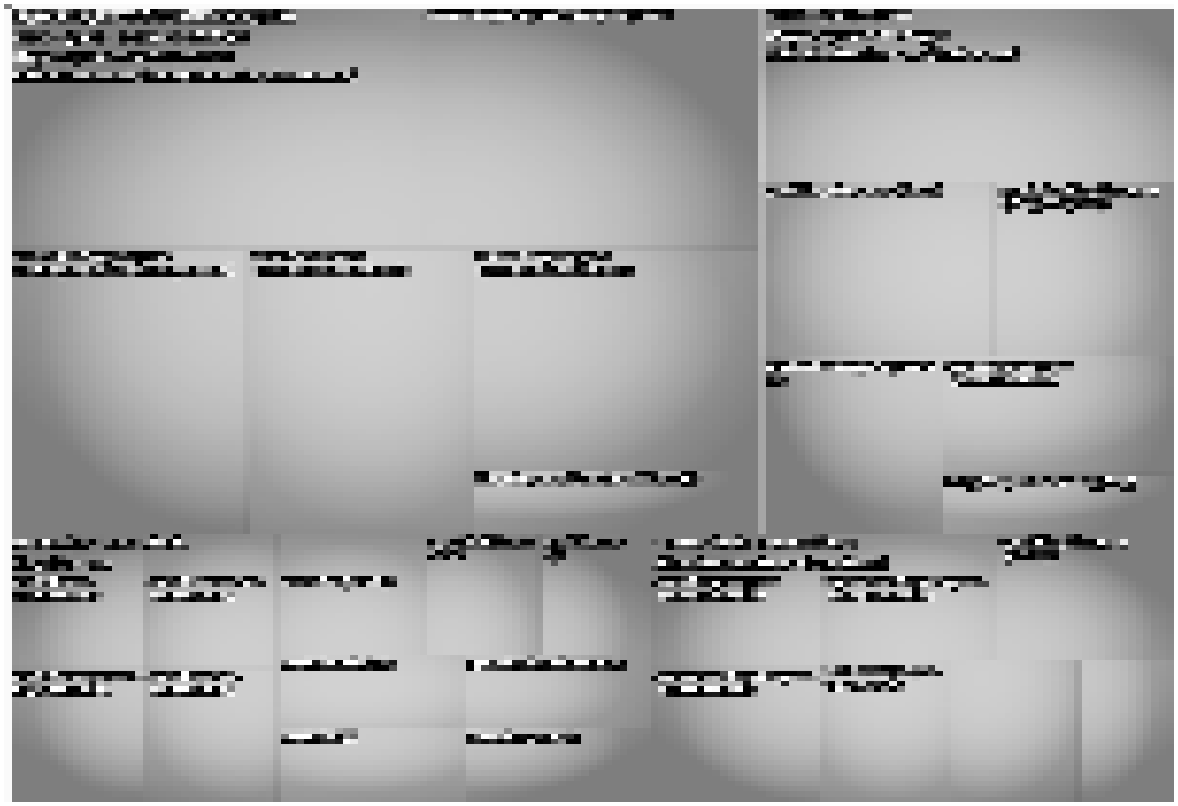


Figure 4.25: Metric tree (JArchitect)

Figure 4.26: Project-2 output (JArchitect)

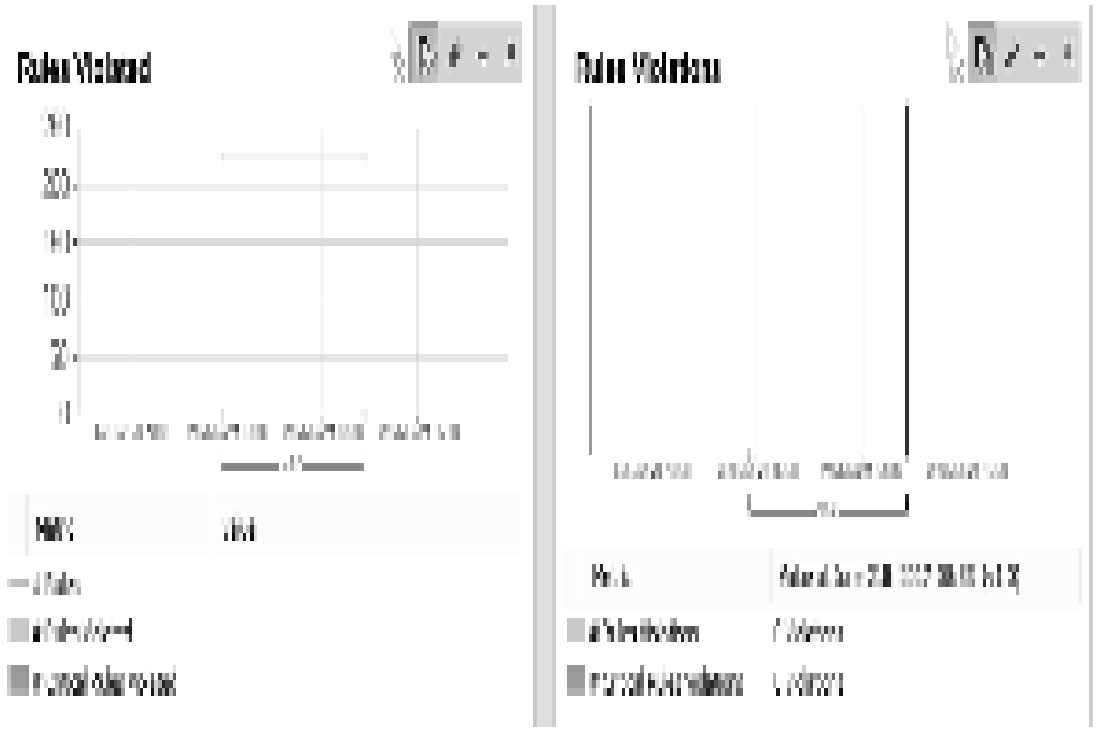


Figure 4.27: Project-2 output (JArchitect)

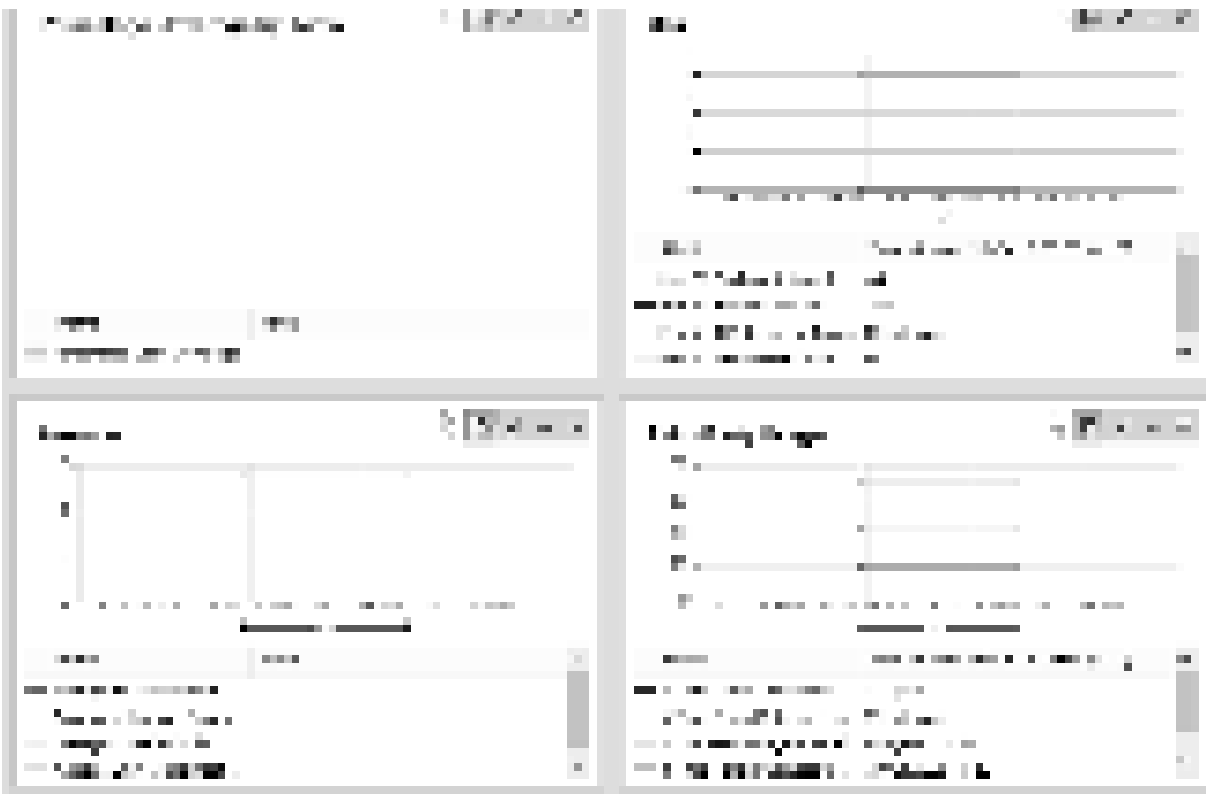


Figure 4.28: Project-2 output (JArchitect)

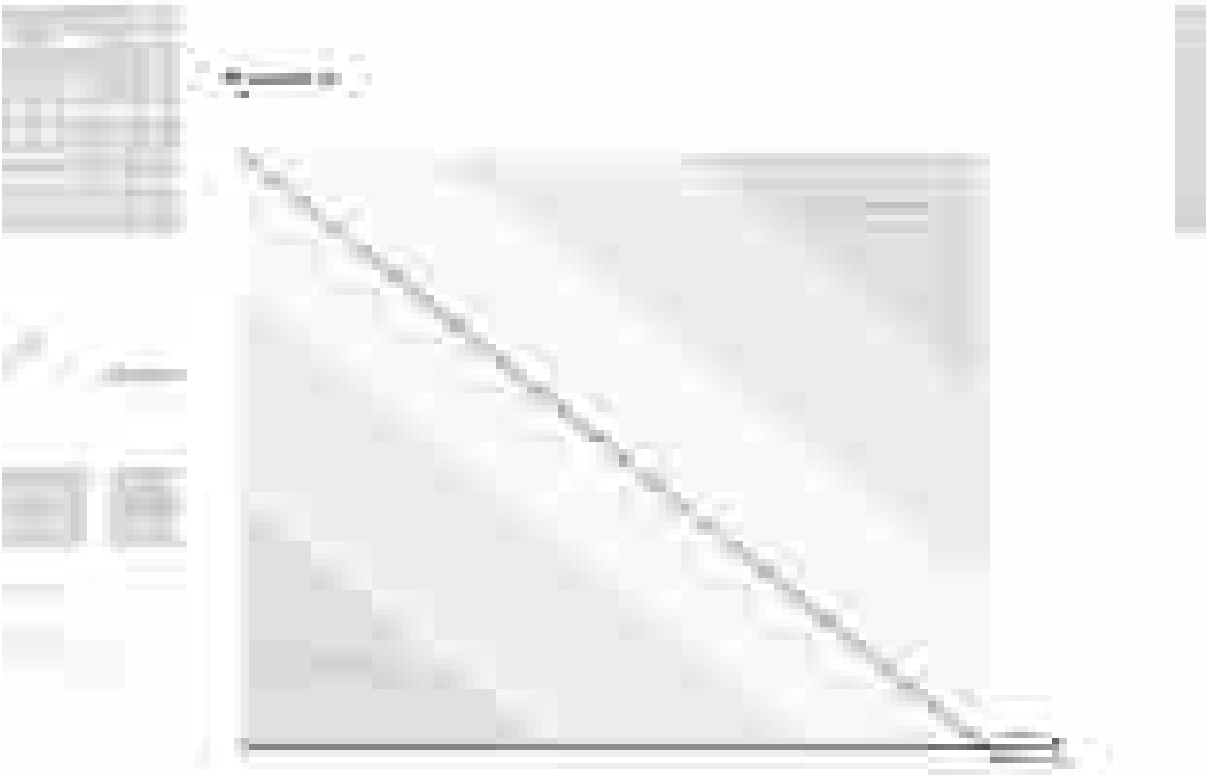


Figure 4.29: Abstractness vs. Instability (JArchitect)

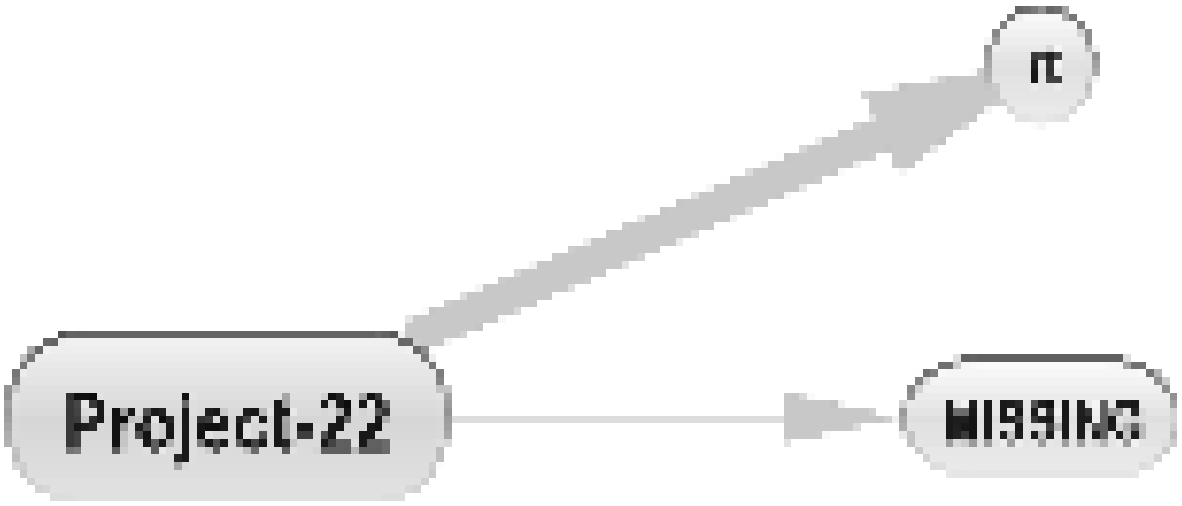


Figure 4.30: Dependency graph (JArchitect)

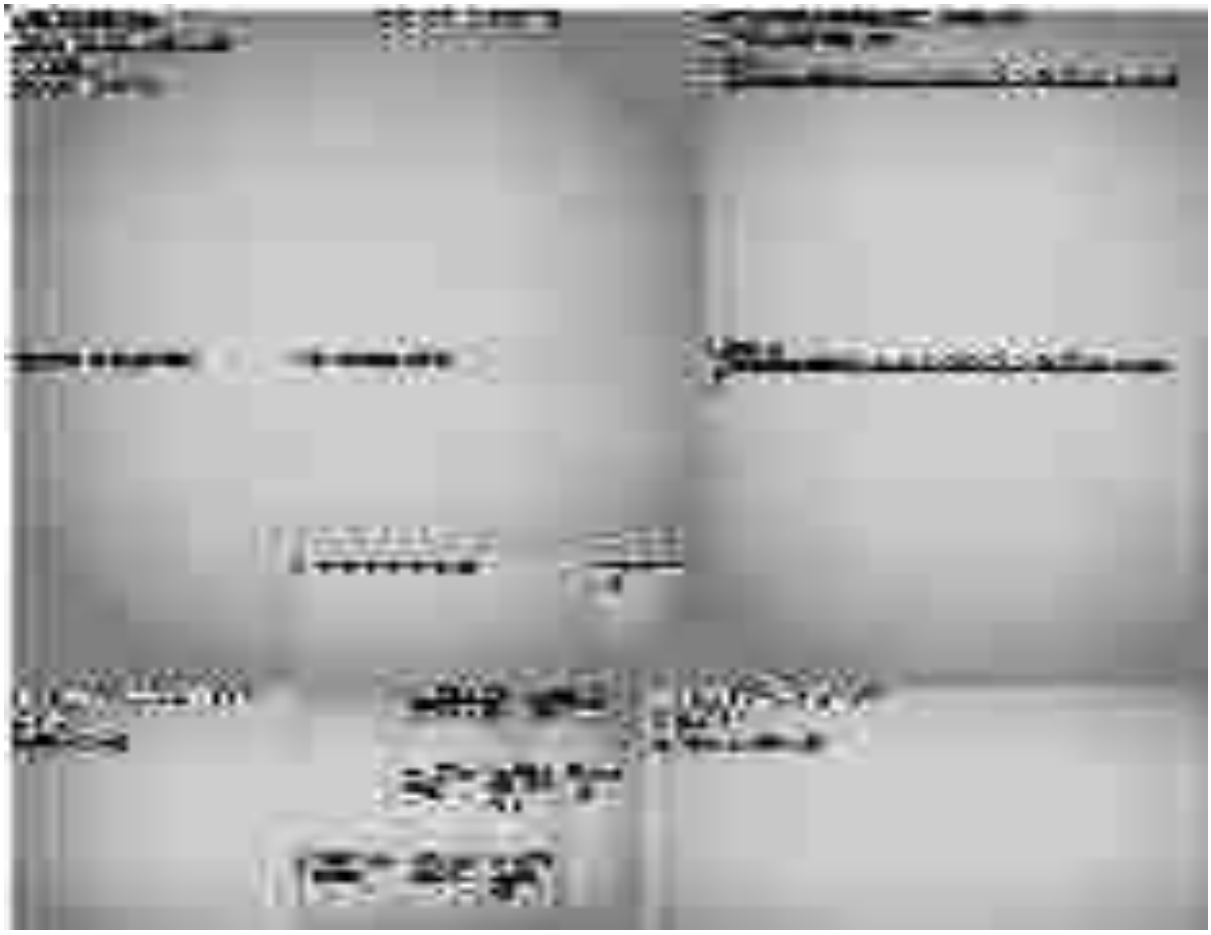


Figure 4.31: Metric tree (JArchitect)

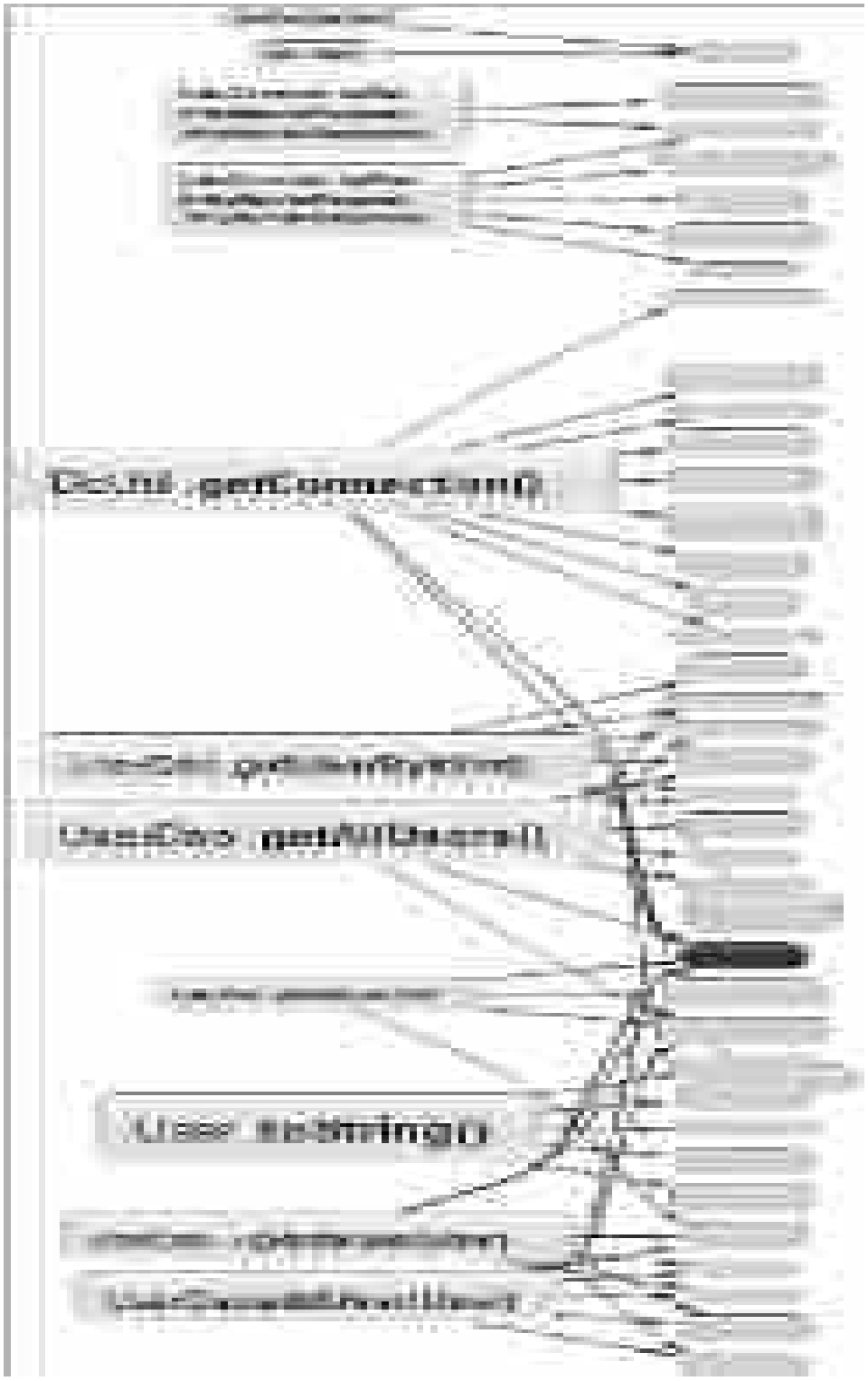


Figure 4.32: Dependency graph (JArchitect)

4.2 Comparison with existing technique

Table 4.1 comparison of Sonaruqbe and JArchitect

	SonarQube	JArchitect
Version	6.2	4.0
License	Lesser GNU General Public License	Proprietary
Purpose	Continuous inspection of code quality.	Static analysis tool for Java code
Strengths	Offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.	Dependency Visualization, Software metrics (82+), Declarative code rule over LINQ query
Operating System	Cross Platform	Multiplatform
Integration	Integrates with Maven, Ant, Gradle, MS Build and continuous integration tools like Bamboo, Jenkins, Hudson,	Integrates with Maven, Ant, Gradle, MS Build
Rule Categories	Language: Type: Tag: Repository: Status: Beta: Deprecated: Ready:	

	Find bugs	**PMD**	**CheckStyle**
Version	3.0.0	5.2.2	6.1.1
License	Lesser GNU Public License	BSD-style license	Lesser General Public License
Purpose	**Potential Bugs** finds - as the name suggests - bugs in Java byte code	**Bad Practices** looks for potential problems, possible bugs, unused and sub-optimal code and over-complicated expressions in the Java source code	**Conventions** scans source code and looks for coding standards, e.g. Sun Code Conventions, JavaDoc

Strengths	- finds often real defects - low false detected rates - fast because byte code - less than 50% false positive	- finds occasionally real defects - finds bad practices	- finds violations of coding conventions
Weaknesses	- is not aware of the sources - needs compiled code	- slow duplicate code detector	- can't find real bugs
Number of rules	408	234	132

Rule Categories	**Correctness** **Bad practice** **Dodgy code** **Multithreaded Correctness** **Performance** **Malicious Code Vulnerability** **Security** **Experimental** **Internationalization**	***JSP*** - Basic JSF - Basic JSP **XSL** - XPath in XSL **Java** - Design - Coupling - Jakarta Commons Logging - Basic - Strict Exceptions - Security Code Guidelines - Java Logging - Android - Controversial - Comments - Type Resolution - Empty Code - String and StringBuffer - Code Size - Braces - Unused Code - Unnecessary - J2EE - JavaBeans - Migration - Import Statements - JUnit - Naming - Finalizer - Optimization - Clone Implementation **Ecmascript** - Basic Ecmascript - Unnecessary - Braces **XML** - Basic XML	**Annotations** **Block Checks** **Class Design** **Coding** **Duplicate Code** **Headers** **Imports** **Javadoc Comments** **Metrics** **Miscellaneous** **Modifiers** **Naming Conventions** **Regexp** **Size Violations** **Whitespace**

S. No	Features	Developer Edition	Build Machine
1	Analysis of Application Projects, Code Source and Third-Party Projects	Yes	Yes
2.	Dashboard, Smart Technical Debt Estimation (On windows version only), Quality Gate (On windows version only)	Yes	Yes
3.	Automatic Report (HTML + javascript) Production through JArchitect.Console.exe	No	Yes
4.	Possible Integration into the Build Process	No	Yes
5.	Warnings about the Health of the Build Process	No	Yes
6.	Interactive UI: Dependency Graph	Yes	Yes
7.	Interactive UI: Dependency Matrix	Yes	
8.	Interactive UI: Metrics Visualization through Treemaping	Yes	
9.	Build Comparison / Code Diff	Yes	Yes
10.	Running an Analysis from Power Tools	Yes	Yes

CHAPTER 5
CONCLUSION AND FUTURE SCOPE

The huge number of various procedures, in both scholarly research and modern practice, shows that manageability of software systems is a vital issue in computer science and strengthens our conviction that interests in programming resources should be secured. Be that as it may, as examined under every arrangement, none of the accessible strategies separately gives a compelling and far reaching answer for controlling architecture disintegration. We chiefly concentrate on Repair strategy for end of software architecture Erosion. This paper additionally quickly evaluated the present condition of structural plan which impacts on software design issues. This work has looked at software architecture, its different definitions, objectives, prerequisites and styles. We reviewed numerous compositional styles or examples that risen out of involvement of software architects in the business and research on software architecture.

5.1 Conclusion

This paper provides an overview of current approaches for dealing with the problem of architecture erosion. I have presented a lightweight classification framework to categories these methods primarily for easier analysis of their efficacy. In this paper I have proposed technique to control architecture erosion. In this I have created two architecture one is based on spring MVC and hibernate framework and the other is simple java code. Based on design principles such as separation of concerns, single responsibility principle, principle of least knowledge I will compare the two architecture with the help of tools i.e. SonarQube and JArchitect in order to find the cyclic-dependencies and architectural violations between the two architectures. This will help us to make the system more reliable and maintainable.

5.2 Future scope

We are still comparing the two architectures one is pattern (MVC) without framework and the other is MVC with spring and hibernate framework. The comparison is done based on the architecture design principles such as separation of concerns, single responsibility principle, etc. The comparison will show that using the MVC spring and hibernate framework we can minimize the architecture erosion. More

to go with JACOCO, In future we are going to implement the same thing using JACOCO code analysis.

"*JaCoCo should provide the standard technology for code coverage analysis in Java VM based environments. The focus is providing a lightweight, flexible and well documented library for integration with various build and development tools.*"

One of most interesting

One of the most interesting thing regarding JACOCO is that it work for functional as well non-functional characteristics and it also lightweight and works also for VM. Today's world is belongs to cloud, so everywhere we found virtual machines if we are writing any code in cloud environment then this tool is very useful. Our next focus is analysis of code in cloud environment using these tools.

References

[1] P. Petrov and U. Buy, “A systemic methodology for software architecture analysis and design,” *Proc. - 2011 8th Int. Conf. Inf. Technol. New Gener. ITNG 2011*, pp. 196–200, 2010.

[2] P. U. Chavan, M. Murugan, and P. P. Chavan, “A review on software architecture styles with layered robotic software architecture,” Proc. - 1st Int. Conf. Comput. Commun. Control Autom. ICCUBEA 2015, pp. 827–831, 2015.

[3] M. De Silva and I. Perera, “Preventing software architecture erosion through static architecture conformance checking,” *2015 IEEE 10th Int. Conf. Ind. Inf. Syst. ICIIS 2015 - Conf. Proc.*, pp. 43–48, 2016.

[4] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, “Recommending refactorings to reverse software architecture erosion,” *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR*, pp. 335–340, 2012.

[5] K. Sartipi, “Software architecture recovery based on pattern matching,” *Softw. Maintenance, 2003. ICSM 2003. Proceedings. Int. Conf.*, pp. 293–296, 2003.

[6] L. Pruijt, C. Köppe, and S. Brinkkemper, “Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 220–229, 2013.

[7] L. De Silva and D. Balasubramaniam, “Controlling software architecture erosion: A survey,” *J. Syst. Softw.*, vol. 85, no. 1, pp. 132–151, 2012.

[8] N. Chanda and X. (Frank) Liu, “Intelligent Analysis of Software Architecture Rationale for Collaborative Software Design,” ???, pp. 287–294, 2015.

[9] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, “A Unified Approach to Architecture Conformance Checking,” *Proc. - 12th Work. IEEE/IFIP Conf. Softw. Archit. WICSA 2015*, pp. 41–50, 2015.

[10] O. Maqbool and H. A. Babri, “Bayesian Learning for Software Architecture Recovery,” *2007 Int. Conf. Electr. Eng.*, pp. 1–6, 2007.

[11] A. Dragomir and H. Lichter, “Model-based software architecture evolution and evaluation,” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2012, vol. 1, pp. 697–700.

[12] A. Budi, D. Lo, and S. Wang, “Automated Detection of Likely Design Flaws in Layered Architectures,” no. July, pp. 7–9, 2011.

[13] S. Herold, M. English, J. Buckley, S. Counsell, and M. O. Cinneide, “Detection of violation causes in reflexion models,” 2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2015 - Proc., pp. 565–569, 2015.

[14] S. Herold, M. Mair, A. Rausch, and I. Schindler, “Checking conformance with reference architectures: A case study,” Proc. - IEEE Int. Enterp. Distrib. Object Comput. Work. EDOC, pp. 71–80, 2013.

[15] J. Van Eyck, N. Boucké, A. Helleboogh, and T. Holvoet, “Using code analysis tools for architectural conformance checking.”

[16] J. Knodel and D. Popescu, “A Comparison of Static Architecture Compliance Checking Approaches 1,” 2007.

[17] ISO/IEC 9126-1, Software Engineering - Product Quality - Part 1: Quality Model, 2001.

[18] N. M. Edwin, “Software Frameworks , Architectural and Design Patterns,” no. July, pp. 670–678, 2014.

[19] Bosch, J (2000), "Design and Use of Software Architectures" , Addison-Wesley Professional.

[20] Microsoft, Microsoft Application Architecture Guide. 2nd ed. Microsoft Press, 2009.

[21] Freeman, P. The Central Role of Design in Software Engineering. Software Engineering Education Freeman, P. and Wasserman, A. eds. Springer-Verlag: New York, 1976

[22] T. Abdellatif, S. Bensalem, J. Combaz, L. De Silva, and F. Ingrand, “Rigorous design of robot software: A formal component-based approach,” Rob. Auton. Syst., vol. 60, no. 12, pp. 1563–1578, 2012.

[23] G. S. Kumar, K. Rameetha, and K. P. Jacob, “A generic software architecture for a domain specific distributed embedded system,” Int. Conf. Softw. Eng. Theory Pract. 2007, SETP 2007, pp. 41–46, 2007.

[24] P. Iñigo-blasco, F. Diaz-del-rio, M. C. Romero-ternero, D. Cagigas-muñiz, and S. Vicente-diaz, “Robotics software frameworks for multi-agent robotic systems development,” Rob. Auton. Syst., vol. 60, no. 6, pp. 803–821, 2012.

[25] W. Michael, “A Layered software architecture for Hard Real Time (HRT) embedded systems Monterey , California,” 2002.

[26] J. B. Tran, R. C. Holt, Forward and reverse repair of software architecture, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, IBM, 1999, pp. 12–20.

[27] M. W. Godfrey, E. H. S. Lee, Secrets from the monster: Extracting Mozilla’s software architecture, in: Proceedings of the International Symposium on Constructing Software Engineering Tools, pp. 15–23.

[28] Garlan, D. A. & Ockerbloom, J. R., (1995) “Architectural mismatch: Why reuse is so hard”. IEEE Software, 12(6):17–26, Nov 1995

[29] T. Abdellatif, S. Bensalem, J. Combaz, L. De Silva, and F. Ingrand, “Rigorous design of robot software: A formal component-based approach,” Rob. Auton. Syst., vol. 60, no. 12, pp. 1563–1578, 2012.

[30] S. Herold, M. Mair, A. Rausch, and I. Schindler, “Checking conformance with reference architectures: A case study,” Proc. - IEEE Int. Enterp. Distrib. Object Comput. Work. EDOC, pp. 71–80, 2013.