SOFTWARE ARCHITECTURE REFACTORING TOOLS AND TECHNIQUES: A COMPARATIVE STUDY

***Khawar Iqbal Malik[1], Muhammad Ilyas[2], Yaseen Ul Haq[3], Abu Sameer[4], Atiya Shakira[5]**
[1]Department of Computer Science & IT University of Lahore, Sargodha,
[2,4,5]Department of Computer Science & IT University of Sargodha,
[3]Department of Computer Science and Engineering, University of Engineering and Technology Lahore, Narowal Campus, Narowal, Pakistan
Contact: *imkhawarmalik@gmail.com

***ABSTRACT*: *Refactoring is a powerful method to enhance the quality and remove the issues of the software. Refactoring is a phenomenon under the domain of performance engineering in which we can enhance reliability and maintainability of software, code cleaning, enhance reusability, and increase extensibility. Architectural refactoring is the process of changing the architecture without changing the behaviour of the software. An Architectural refactoring normally also includes the code refactoring. Bad smell or code smell in a programming language is a problem that makes it difficult for the software to acknowledge and retain the code. Some refactoring instruments and methods that are used to remove code smells from code are addressed in this document. Here we draw conclusion about how programmer, refactor characterize the assumption taken by each. This debate will help the future scientist to select a suitable method and tool for refactoring architecture.*

Key Words: Software refactoring, bad smell, refactoring tools, refactoring techniques.

1- INTRODUCTION

Refactoring is a composition enhancement method with the exception of altering a system's external behavior. The emergence of a purposeful technique of refactoring and patterns of refactoring helps software developers to influence established alternative approaches when coping with needs for routine refactoring.

They may, therefore, move away from the degradation of the layout [1]. Refactoring may be described as a "coordinated set of deliberate architectural things to do that get rid of a particular architectural smell and improve at least one satisfactory attribute barring altering the system's scope and functionality" [2].

There are now multiple instruments, methods and frameworks for refactoring, each one had specific areas and characteristics of the implementation. In this article, we will attempt to categorize some instruments for refactoring, to use an evolutionary categorization based on tool characterization processes. This taxonomy is focused on modifying methods and variables affecting these processes. This taxonomy's objective is just to place substantive instruments and methods within the software speciation framework, making it simpler to match and merge them. For this research, various tools were selected. Due to the obvious variations among them, these instruments were selected. It would also be important to see how the instruments are as distinct as they appear to be. According to Martin Fowler [2] "Using refactoring, observable conduct is not modified if it alters the inside shape of software program for simpler to apprehend and more cost-effective to modify the code". When refactoring is being used to alter the code, it improves robustness, reliability, and code maintainability. Refactoring is an important part of both the software application enhancement system and extra refactoring machinery is necessary for fast refactoring and behavior preservation. Generally, refactoring is indeed a scheme that alternates a device software application in the same manner that it stays the same outcome, only improving an inside code. Maranzano et al [3] shown that more than a couple of steps are often used to refract the code. These fundamental steps can be described as below.

a. Use the system code module test.
b. Find those code in the package with "odor." or smell
c. Decide how certain code smells can be simplified.
d. To remove its code smell, pick and execute refactoring technique.
e. Do it all over again, simplify/ test till the odor leaves.

This a review paper regarding software architecture refactoring tools and techniques and here in section 2 we discuss the motivation for software refactoring. Here we also discuss the two practical examples of architecture refactoring. Section 3 is regarding dealing with bad smells and design flaws. In Section 4 comparison of tools used to handle refactoring and bad smells. Different obstacles for Architecture refactoring are discussed in section 5. Section 6 is about the conclusion of refactoring, reengineering and rewriting.

2- MOTIVATION FOR SOFTWARE ARCHITECTURE REFACTORING

Typically, professional software systems seem to be complicated and stayed for a number of years. Perceive, for instance, the Windows OS. From the past 25 years, that has risen to about fifty million lines of code. Evolution on this kind of moment and volume scale presents a danger to the software's systemic content. Therefore, to preserve its structural quality of this kind of complicated and changing software program, regular architecture refactoring would be required [4]. Moreover, such refactoring becomes compulsory for all of the software product's achievement as it enables easier development of fresh characteristics [4]. For example, Windows undertook a significant refactoring attempt as it evolved from Windows Vista to Windows OS 7 version [7]. The main objective of this refactoring was always to replace the missing dependencies between components in order to tackle layering breaches and enhance the framework of dependency [5]. An important point is to be noted here that if refactoring methodologies can be used for code then why these could not be applicable for rest of software developing elements like UML diagrams (class, package and object), sequential logic and behavior diagrams and other integrity constraints over databases [6]. Actually, because of its constant development (also recognized as fragmented development), software architecture indicates a potential region for refactoring operations. Software architecture evaluation and refactoring, therefore, should be carried out frequently in all phases [2, 3].

4- DEALING WITH BAD SMELLS

Pieces of code that is wrong in some experience and unsightly to see [4]. Bad smells in different words, it is a code or sketch problem that happens when structural traits of software are described but it does not produce any error at the execution time [5].

3.1 Types of bad smells The type of bad smells in the code is as follows [6, 7].

Duplicate Code: If the same piece of code is available in any location of the program it is called duplicate code. It is more critical especially in the case when you change it at one location and not in another one [6].

Long Method: Where code statements, different loops and variables are a lengthy process in one strategy. The long method will be too long, so understanding the code is a problem [7].

Large Class: A genre that tries to perform too much function is an enormous class. A large type reduces compassion from a system that has many parameters or techniques of example.

Feature Envy: A technique outlined in one form but additionally engaged in the characteristic of distinct categories is the location it is currently positioned and the information is the focus of the envy's attention in function envy. The approach appears happier in separate classes in one section [6].

Long Parameter List: Several parameters have exceeded one technique. Recognizing and ending up with erratic code is hard.

In literature we have seen many term used by researcher for architectural smells, few of them are as follows:

- “architectural bad smells” [28]
- “architecture smells" [29]
- “anti-patterns" [30]
- “architecturally-relevant code smells" [28]
- “contra-indicated patterns" [29]
- “architectural defects" [29]

The refactoring obstacle includes classifying areas that architects may need to enhance. The writers of [6, 7] incorporated "code smells" to define prospective regions for enhancement for code refactoring.

3.2 Architectural Issues due to bad smell

Similarly, architectural odors are markers of architectural issues. The below list illustrates some popular instances:

Duplicate design artifacts: If distinct architecture parts are allocated the same obligations, the principle of DRY (do not repeat yourself) may be breached. Important tasks must be modularized [18], as facet-oriented software design shows.

Unclear roles of entities: Component identities should clarify their duties such that the design is readily understood by the developer/engineer. Similarly, individual parts should be allocated duties and not distributed across various components [17, 18].

Inexpressive or complex architecture: Unnecessary abstractions result from accidental complexity. These abstractions lead to software systems that are complicated and inexpressive. Architecture entities, for instance, may have uncertain or false names, superfluous elements or dependencies, or a granularity that is either too fine or too coarse [15].

Everything centralized: Software engineers might be misinformed towards centralized methods, even if it would be more suitable to organize and decentralize themselves [15]. A decentralized strategy to architecture is much more suitable if the issue is intrinsically decentralized.

Over-generic design: Patterns like the pattern of strategy design enable variation to be deferred to subsequent binding moments. If they are misused, however, they suffer from sustainability and expressiveness [13, 14]. The design of architecture must be as precise as feasible and as clichéd and extensible as needed.

Asymmetric structure: Inner quality of architecture is mostly an indication for symmetric structure, while asymmetry can imply future architectural problems. There can be two types of symmetry: symmetry of behavior and symmetry of structure [10]. Behavioral symmetry primarily deals with launching and start features, like, (i) an open method required close method, (ii) a start operation requiring a hold or rollback technique, or a block requiring a join.

Dependency cycles: The cycles of dependence between architectural elements show an issue because they could have an adverse effect on the ability to test, modify, or expressively [11].

Design violations: Infraction of design policy goals, like the use of relaxed layering rather than a strict layering, must be evaded; otherwise, distinct project engineers would uncontrollably solve this same kind of issue with various solutions, reducing accessibility and expressiveness [12].

Inadequate partitioning of functionality: Some other source of accidental difficulty is the insufficient sequencing of duties to subsystems. In particular, a subsystem's constituents should show high cohesion, whereas subsystem coupling will be low [11]. Otherwise, it may also show incorrect service file systems into components.

Unnecessary dependencies: The number of dependencies must be mitigated in order to decrease complexity. All extra and needless dependencies (i.e. accidental) may influence efficiency and modifiability [15].

Implicit dependencies: If the application of a software package includes frameworks not present in the architectural designs, this can result in several liabilities [15, 16]. Developers could generate a gap among required architecture and applied architecture by adding inherent dependencies to an application without telling anybody about such fresh dependencies.

3.3 Architectural smells classification

We suggest a categorization of the architectural smell in context to each smell according to the breach of certain design rules. We regarded Ganesh et al. [18] suggested categories, based on four concepts of architecture: modularity, hierarchy, abstraction, and encapsulation. The reasoning for these categories, as stated by the researchers [21,18], is that it allows for an acute awareness of the smell and provides a stronger idea of how to refactor the architectural smell.

The criteria for classification that we have chosen are as under:

Modularity [21]: It is the feature of software already divided into a collection of domain-specific and cohesive parts.

Hierarchy: This class or group of abstractions in which an abstraction shows the essential characteristics of an object that varies from all other kinds of stuff and thus provides elegant defined structural boundaries comparable to the perspective of the user [21]:.

Healthy Dependency Structure: A (sub-) system's dependence structure is regarded as unhealthy when it encourages a sequence of system modifications every time it is altered [21].

3.3 Dealing with design flaws

Time constraints in the delivery of software and extra specifications compelled software engineers to change and adapt the architecture continually. Regretfully, for this reason, they did not comply with a systematic strategy, so they could use ad hoc patches and backpacks to develop the scheme. The subsequent software architecture had become over-complicated and indistinct and after a while endured from reduced modifiability.

a) First of all, certain software architecture will not be constructed with "Big Bang" strategy [11], but instead in tiny phases and iterations while each iterations plots one criterion or a tiny set of demands for specific architectural choices.

b) Using a fragmented growth strategy helps to manage hazards by detecting architectural problems early [11].

c) Rather than slicing stone architectural choices, architects should re-evaluate their structure in all iterations, describe prospective design problems and fix them through refactoring. Instead of addressing symptoms, this strategy helps heal the issue**.**

Table 1: Design issues due to bad smell and their description

Design issue	Description
Duplicate design artifacts	Distinct architecture parts are allocated the same obligations, the principle of DRY (do not repeat yourself) may be breached [14].
Unclear roles of entities	Component identities should clarify their duties such that the design is readily understood by the developer/engineer.[15, 16].
Complex architecture	Unnecessary abstractions result from accidental complexity [13].
Everything centralized	Software engineers might be misinformed towards centralized methods, even if it would be more suitable to organize and decentralize themselves [13].
Over-generic design	Patterns like strategy design enable variation to be deferred to subsequent binding moments. If they are misused, however, they suffer from sustainability and expressiveness [11, 12].

5- REFACTORING TOOLS AND COMPARISON

The various tools available for Architecture Refactoring have been discussed in this section. Table 2 presents there comparison of different facts.

Visual Works Visual Works is a famous Cincom-made Smalltalk IDE. As of Visual Works 7.0, the popular Refactoring Browser [31] became the normal Smalltalk browser. Cincom has been working intimately with Refectory. Inc. Incorporate a complete browser refactoring toolset into Visual Works to achieve this. This provides VisualWorks ' market place as the top toolkit for extreme programming.

Smalltalk Browser The first commonly used refactoring instrument was the Smalltalk refactoring browser, which used most of the Smalltalk language refactoring standard classes, techniques, and fields [28]. Smalltalk is a very neat and clean language that processes more automatically than other languages like C++.

The Eclipse The project was intended to build IDE's that could be utilized to generate applications as various Websites etc. The Eclipse has integrated architecture that is upgradeable [29]. The software development kit has many built-in generic characteristics.

Guru Programming language refactoring tool created by Ivan Moore. Its language is object-oriented prototypical, so there is no difference between class and instance objects. Objects are related to relationships of inheritance. It is used in an automatic way to restructure heritage hierarchies and refactor techniques of SELF programs [31].

TogetherSoft ControlCenter 6.0 An application development tool specifically designed to streamline the software development method. Modeling of applications used to ensure the company needs. It provides a good synchronization around application design and code conversion [30].

.

Table 2 Comparison of different Architecture Refactoring tools

Fact	Visual Works	Eclipse	Guru	ControlCenter
Change history	Irrelevant	Parallel/Async.	Un-versioned	Versioned
Frequency	Continuously	Continuously	Occasionally	Continuously
Role	Developer/Designer	Developer/tester	Developer	Developer
Distribution	Local	Local	Local	Local
Automation	Semi-automatic	Semi-automatic	Fully automated	Semi-automatic
Invasiveness	Non-invasive	Non-invasive	Highly invasive	Non-invasive
Effort	Low effort	Low effort	Virtually no effort	Low effort
Locality	Global	Global	Global	Global
Scope	Source code	Source code	Source code	Design/source code
Openness	Source available reflection	Plug-in architecture	Source available	Integration API Wizards
Control	Controlled refactoring	Controlled refactoring	Controlled	Controlled refactoring

The various tools available for Architecture Refactoring have been discussed in this section. Table 2 presents there comparison over different facts.
In literature Ganesh *et al.* [21] also discuss the following tools that are used for the removal of different bad smells from architecture. Here we also provide a comparison of these tools in the context of a platform where these are useful and there supported languages. Few of them are available in commercial versions other few are operative in license-free environment.

Table 3 Architecture Refactoring Tools w.r.t Language and Platform supported

Tool Name	Supported platform	Supported Languages	License	Currently Availability
AI Reviewer [21]	Mac OS, Microsoft Windows, Linux OS	C++ & C	Licensed	Available
ARCADE [22][21]	Mac OS, Microsoft Windows, Linux OS	Java	Free	Not Available
Arcan [21]	Mac OS, Microsoft Windows, Linux OS	Java	Free	Available
Designite [21]	Microsoft Windows	C Sharp (C #)	Licensed	Available
Hotspot Detector [21]	NA	Java	NA	Not Available
Massey Architecture Explorer [25]	Mac OS, Microsoft Windows, Linux OS	Java	Free	Not Available
Sonargraph Commercial [25]	Mac OS, Microsoft Windows, Linux OS	Java, C Sharp C++ & C	Licensed	Available
STAN [21]	Mac OS, Microsoft Windows, Linux OS	Java	Licensed	Available
Structure 101 [27]	Mac OS, Microsoft Windows, Linux OS	Java, .Net, C++ & C	Licensed	Available

Table 4 Comparison of Architectural Smell Classes and Tools used to remove these smells

Architectural Smell Class	Other Names in literature	Description	Variants	Violated Principal	Tools
Cyclic Dependency	Tangle [21], Cross-Package Cycle [23], Cycle of classes [27], Cross-Module Cycle [23]	When more than two architectural sections depend on each other formally or informally.	Strong Circular Dependencies Between Packages [31], Shape detection [24]	Structure, Modularity, Healthy Dependency	Dependency Finder, JArchitect, ClassCycle,NDepend, LDM,
Unstable Dependency	Unstable Interface [23]	Unstable reliance defines a module (component) that focuses on certain components that are less stable than themselves.	Unstable Interface [23]	Structure, Healthy Dependency	Designite, Arcan, Hotspot Detector
Unutilized Abstraction	Super-type Bypass, Policy Detail Dependency [27]	Architectural smell relates to the issue of pointing immediately to a concrete class.		Healthy Dependency Structure, Hierarchy	Designite AI Reviewer
Cyclic Hierarchy	Subtype Knowledge, Unhealthy Inheritance Hierarchy [23]	Due to cyclic dependency between sub and super-type of namespaces		Healthy Dependency, Structure, Hierarchy,	Designite, AI Reviewer, Massey Architecture Explorer
Multipath Hierarchy	Degenerated Inheritance [31].	Such smell arises when different heritage routes link sub-types with their super-types or a specific class with their abstractions.		Hierarchy	Massey Architecture Explorer, Designite
Hub-Like Dependency	Link Overload [20], Hub-like Modularization [28].	These dependencies occur when abstraction depends on large number of other concrete classes or abstractions	Over reliant Class [26]: Dense Structure [28]:	Modularity, Structure, Healthy Dependency	Designite, ARCADE, Arcan, AI Reviewer
Scattered Functionality	Scattered parasitic functionality [20].	Such odor occurs if the same high-level issue has to be realized by different components.		Modularity	Designite, ARCADE
God Component	Concern overload [20], God Class [27]	Such smell indicates that there are too many problems with the component and builds up so much control.		Modularity	ARCADE, Designite, AI Reviewer
Abstraction without disassociation	Unhealthy Inheritance Hierarchy [23]	Architectural smell that leads to a situation where a customer class uses a service defined as an abstract type		Structure, Healthy Dependency	Massey Architecture Explorer, Hotspot Detector

6- BARRIERS OF ARCHITECTURE REFACTORING

The requirement for refactoring programming design appears to be obvious; however software engineers/architects still need to manage different hindrances as fast as they endeavor to acquaint their association with refactoring. Naturally, they will only do it for their interest but they usually sacrifice over many advantages [16]. There are a few diverse areas where obstacles may arise:

5.1 Technology and tools

On one side, the refactoring procedure must be performed manually owing to the lack of instruments actively supporting the refactoring of software architecture, which is often tedious and prone to errors. On the other side, it is even more cumbersome and error-prone to find and cope with architectural issues in subsequent stages. If a catalog or scheme of refactoring patterns is available, software architects can improve the software much more efficiently [18].

5.2 Development process

The mechanism of refactoring must be explicitly incorporated into the general method of growth. Apart from that, project management will not schedule sufficient resources for purposes of rehabilitation. Furthermore, duties for refactoring must be explicitly assigned to various stakeholders, such as testers or software architects [18].

5.2 Practical Implementation

If the design erosion of the software system is sophisticated, tactical refactoring can only cure the symptoms and not the causes, reengineering or even rewriting can be more appropriate and efficient [19]. Such a scenario could be intimated when refactoring activities cannot boost the quality to a higher media (such as bug rates or architecture metrics).

5.4 Management and organization:

Software engineering's primary assets are often regarded by the institution's participants, like product management. Factors like "the design of software architecture should first be correctly conducted so that no problems can ever occur" overlook the reality that all but small projects are rapidly changing. First, at least not in complete detail, software engineers do not originally understand all specifications. Therefore, choices can only take into consideration current knowledge [17]. As an early understanding deepens, it is necessary to check and refine prior decisions.

6 CONCLUSION

As Martin Fowler suggests, architectural refactoring usually includes code refactoring to making it more sustainable without altering its empirical behavior. We concentrate on program entities such as packages, classes, and techniques for code refactoring. Code refactoring is indeed a structure-preserving bottom-up activity while architectural refactoring is a top-down process that is used to enhance the quality of the structure. It affects elements, connectors, modules, interfaces. Architectural refactoring is an intentional method for removing architectural odors without altering the code's features or functionality We addressed the distinct classes of poor smell in software architecture in this article and we also try to address the distinct instruments and compare them over specified guidelines and characteristics with one another. According to this debate, Designite is an instrument that can manage most types of classes of bad smells, but the drawback is it only works with the Windows platform and supports C # language. Similarly, ControlCenter gives the best performance although it's a semi-automatic tool for refactoring.

REFERENCES:

[1] Zimmermann, O. Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, ***32***(2), pp. 26-29 (2015).

[2] Fowler, M. Refactoring: Improving the design of existing code. In ***11th*** *European Conference, Jyväskylä, Finland, June 9 - 13,* (1997).

[3] Kruchten, Philippe, Henk Obbink, and Judith Stafford. The past, present, and future for software architecture. *IEEE software,* **23**(2), pp. 22-30 (2006).

[4] Joshi, Rohit R., and Rajesh V. Argiddi. Author identification: an approach based on style feature metrics of software source codes. *International Journal of Computer Science and Information Technologies***, 4(**4), (2013).

[5] Kim, Miryung, et al. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, **40**(7), pp 633-649 (2014).

[6] Samarthyam, et al. Refactoring for software architecture smells. Conference on **1st** *International Workshop on Refactoring* ACM, (2016).

[7] Terra, Ricardo, et al. Recommending refactoring to reverse software architecture erosion. ***16****th Conference on Software Maintenance and Reengineering 2012*, IEEE, (2012).

[8] Xiao, Lu, et al. Identifying and quantifying architectural debt. *International Conference on Software Engineering '****16*** *38th*, ACM, (2016).

[9] Kumar, M. Raveendra, and R. Hari Kumar. Architectural refactoring of a mission critical integration application: a case study. ***11th*** *India Software Engineering Conference*, ACM, (2011).

[10] Newman, Sam. Building microservices: designing fine-grained systems. O'Reilly Media, Inc, (2015).

[11] Syed, Madiha H., and Eduardo B. Fernandez. The software container pattern. **22nd** *European conference on Pattern Language of Program*, (2017).

[12] Arcelli, Davide, at al. Performance-Driven Software Architecture Refactoring. *International Conference on Software Architecture Companion (ICSA-C)*, IEEE, (2018).

[13] Arcelli, Davide, at al. Antipattern-based model refactoring for software performance improvement. *ACM SIGSOFT conference on Quality of Software Architecture*, (2012).

[14] Trubiani, Catia, and Anne Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. *ACM SIGSOFT Software Engineering Notes* **36**.5, pp.36-36 (2011).

[15] Arcelli, Davide, et al. EASIER: an Evolutionary Approach for multi-objective Software archItecturE

Refactoring. *IEEE International Conference on Software Architecture*, (2018).

[16] Ganesh, S. G., Tushar Sharma, at el. Towards a Principle-based Classification of Structural Design Smells. *Journal of Object Technology*, **12**(2), pp.1-1(2013)

[17] Mo, Ran, et al. Hotspot patterns: The formal definition and automatic detection of architecture smells. ***12th*** *Working IEEE/IFIP Conference on Software Architecture*, IEEE, (2015).

[18] Le, Duc Minh, et al. Relating architectural decay and sustainability of software systems. ***13th*** *Working IEEE/IFIP Conference on Software Architecture*, (2016).

[19] Azadi, Umberto, at el. Architectural smells detected by tools: a catalogue proposal. *International Conference on Technical Debt (2019).*

[20] Le, Duc Minh, et al. An empirical study of architectural change in open-source software systems. ***12th*** *Working Conference on Mining Software Repository*. IEEE, (2015).

[21] Kim, Miryung, at el. A field study of refactoring challenges and benefits. *Foundation of Software Engineering* ***12th*** *proceedings of the ACM SIGSOFT*, (2012).

[22] Baum, David, et al. Visualizing Design Erosion: How Big Balls of Mud are Made**. 6th** *IEEE Working Conference of Software Visualization, (*2018).

[23] Buyya, Rajkumar, at el. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. ***10th*** *IEEE Conference on High Performance Computing and Communication, (*2008).

[24] Hovorushchenko, Tetiana, at el. Intelligent System for Determining the Sufficiency of Metric Information in the Software Requirements Specifications. **2nd** International workshop on *Computer Modelling and Intelligent System*, (2019).

[25] Samarthyam, Ganesh, at el. Refactoring for software architecture smells. *Proceedings of* ***1st*** *Internatinal workshop of Software Refactoring*, ACM, (2016).

[26] Bass, Len, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, (2003).

[27] Lauder, at el. Legacy system anti-patterns and a pattern-oriented migration response Systems Engineering for Business Process Change. Springer, London, pp 239-250, (2000).

[28] Manolescu, Dragos A. Workflow enactment with continuation and future objects. *ACM SIGPLAN Notices* **37(**11), pp.40-51 (2002).

[29] Zimmermann, Thomas, at el. Predicting defects for eclipse. ***3rd*** *International Workshop on Predictor Model in Software Engineering '****07****: International Conference on Software Enginering,* IEEE, (2007).

[30] Simmonds, Jocelyn, and Tom Mens. A comparison of software refactoring tools. *Programming Technology Lab* (2002).

[31] Beck, Kent, and Erich Gamma. Extreme programming explained: embrace change. addison-wesley professional, (2000).

[32] Hamid, Almas, et al. A comparative study on code smell detection tools. *International Journal of Advance Science and Technology* **60**, pp.25-32.