CHAPTER 5
A VERIFIABLE MECHANISM FOR GENERIC AUTOMATED REFACTORING TOOL

This chapter addresses the complete automated verifiable mechanism which certifies all the functional components of application and various processes involved during the verification phase. We are particularly interested in complex program transformation based on a sequence of refactoring operations provided by eclipse tools. In the previous chapter, we already discussed about refactoring which is usually a process of improving the underlying design and architecture of java systems that subsequently can improve their performance and maintainability. Many of the old version java technologies are no longer supported, hence the need for migration.

However, the refactoring tools are not correct in every possible case, and programmers cannot trust them. One has to make sure that the functionality of the java system remains intact after going through the process of migration. Hence there is a need to build verifiable refactoring tools, which has to make sure that the behavior of refactored the system is intact with existing system.

Moreover, substantial experiential proof is also offered on how refactoring is of momentous benefit in restructuring automatically-generated program code to be employed in bigger systems. Automated refactoring with an archetype implementation demonstrates that our techniques quite simple to put in to practice as they necessitate only negligible modifications to current refactoring tools. Moreover, this we prototype is a principal item for machine-driven verification, that provides perceptible advantages in software development in practice.

5.1 INTRODUCTION
Software refactoring should be perceived as a procedure of progressively refining the structure of prevailing software via modifications and changes that are behavior-preserving. These minor modifications are cumbersome and daunting. The widely prevalent ones are nifty uncomplicated exercises like renaming variables or breaking up functions into smaller constituents to diminish their complexity. Maneuvers of that kind are easy to automate. They are accessible aplenty in today’s scenario. Sadly enough, even such ostensibly up-front operations as renaming turn out to be tough to
put into use appropriately for contemporary programming languages. Especially object-oriented languages like Java have intricate perusal rules that can intermingle in unanticipated fashions and make it pretty stiff to bring about a name change without presenting annoying duty of capturing the name [90].

Moreover, tools that perform refactoring do get employed for numerous languages, but it appears improbable to try a direct authentication of these kinds of shrink-wrapped or canned operations. An object language with its appropriate semantics is commonly deficient and the operations tend to be exceedingly intricate programs. In our research work, we proposed a novel method for automated software refactoring that carry out momentous refactoring on programs that get composed in an object language that is rich. This technique certainly benefits the refactoring mechanism from mechanized verification. To achieve this, initially we proposed the two modularization axes: Initially, the refactor implementation will be broken up into slighter mechanisms to be substantiated autonomously. It can be demonstrated that the distinct apparatuses explicit properties dealing with correctness. They are not dependent on the presence of a complete prescribed semantics of the object language. Next, specified refactoring will be verified for the object language. Finally, this tool and its evidence of accuracy should be easy to apply adaptations to work on extended object languages, which provides powerful mechanisms for automated reasoning and certified refactored code.

5.2 CONCEPTUAL ARCHITECTURE FOR VERIFIABLE REFACTORED SOFTWARE SYSTEM

The system uses component called switch which helps in toggling between existing software system and refactored systems in a convenient and effective manner allowing the user to migrate easily. The main functionality of this switch component in migration is to provide backward compatibility.

Moreover, if the developer want to invoke or to undo the changes to existing software system code that has already made then he can use same switching as depicted in Fig: 5.1. The network barrier (Switch) component has several sub components such as Router, Dashboard, and Façade which helps in achieving different functionalities as depicted in Figure. Let us give brief each sub-component.
Router is the heart of the switch which is responsible mainly for routing the requests to either existing software system or refactored systems. Moreover, Router will identify the transaction path based on the defined mappings at the dashboard level. We will use constant values 0 and 1 to identify the route for existing software system and refactored systems respectively as shown in Fig: 5.2.

![Dashboard Diagram](image)

**Fig: 5.1 Conceptual Architecture for Verification Mechanism**

Second sub-component is facade which provides backward compatibility for Existing Software System applications. Whenever there is a need to refactor systems, there also definite need to provide backward compatibility for software system. This operation can be performed by façade which is shown in Fig-5.4. Once the response from the refactored system is received, it is validated against the configuration file to check whether there is a need to provide backward compatibility, if yes apply the configurations on the response received from the refactored system so that it will be apple to apple match to the legacy system The logic for implementation of Façade is shown Fig 5.3.

Final one is dashboard which is mainly used by the switch for mapping that identifies the transaction path of all the request that are sent by the consumer or application programmer as shown in Fig:5.6.

The main goal of the transformation phase in GAFactor is to develop a new system that has been improved in some way. What that improvement goal is depends on the individual project, but it could include improved quality attributes (higher modularity, lower complexity, less replicated code etc.,).
As a part of this phase, the quality metrics must be determined both at the beginning of
the transformation and then throughout the iterative transformation steps. In this way,
it can be verified that the transformations are making the desired improvements.

```
public class Façade {
    public void m1() {
        // make all calls into the existing system,
        // complexity hiding
    }
    public string m2() {
        // make all calls into the existing system,
        // the return gets converted
        return rval;
    }
}
```

Fig: 5.2 Working Process of a Router

Fig: 5.3 Logic for implementation of Façade
5.3 PROPOSED VERIFICATION METHODOLOGY

This section describes architecture for verifiable refactored System that shows extended work of previous result for generating the correct code which we had implemented the refactoring (GAFactor) described in the previous section. The following sections elaborate on each step of the validating refactored process which includes various components Certification Router, Certification Engine, Certification Database, Rules Engine, and Certification Dashboard as shown in the Fig 5.5. Let us discuss about these various components in detail.

Certification Router: Router component acts as a façade between the consumer applications and the actual existing system. This component clones the actual request coming from the consumer application and routes the actual request to the existing system and the cloned request to the refactored system. It also stores the actual request in the certification database which later will be used to perform debugging.

Certification Engine: Once the request is sent to both existing and refactored system, they will hit the backend layer. Once Certification Engine gets the response from the backend service and clones the response computed by each of the existing and refactored system and stores it in the certification database. Then the existing system will send the actual response to the user and refactored system does nothing. Once the Existing system response and the refactored system response is stored in the certification database, the certification engine will pick them up, tie them to the actual request and compares the responses. In the process of comparison the certification
engine talks to the rules engine to check any particular rules are defined for the application and based on that will do the comparison and inserts the results into certification database.

**Certification Database:** This will hold all the output data of the certification engine, certification router and the certification database. Certification_Route_Master contains all the information about the routing details about Existing and refactored systems.

**Certification_results_master:** It will have all the certification results and also references request and response masters of the Existing and the refactored applications

**Rules Engine:** Rules engine consists of all the possible deviations of refactored system response from the Existing system response. The possibilities include approved design deviations during the migration phase, known bugs from the existing application, change requests implemented in the refactored system, improved functionality of the refactored system.

Approved design deviations are the design and architectural changes that were made as part of the migration process. It includes renaming the method, change in the error handling framework, change in the data model of the application to make it in line with the data model of the organization, change in the way we make backend service calls.

Known bugs from the existing system are the bugs or issues or defects that are identified as part of the independent validation and verification process that will be conducted during any migration phase of the application.

Quality management system hinges on Verification and Validation as they are crucial parts. These independent procedures that are carried out together to examine if the products, services, or systems do meet necessities and stipulations and that it accomplishes its envisioned purpose.

"Independent" (or) I precede the words "verification" and "validation" to indicate that the verification and validation are to be completed by an impartial third party. It is sometimes said that validation can be expressed by the query "Are you building the exact entity?" and authentication by "Are you doing it properly and accurately?" The convention of these nomenclatures varies. Sometimes they are even used as substitutes. IV & V allow the avid researcher try to detect and put on record any faults or a computer program’s irregularities in a system that results in an inappropriate or unforeseen result,
or leads to queer off beat behavior. Maximum infections ascend due to human errors or blunders in the design of a program, the source code or its frameworks.

Fig: 5.5 Architecture for Automated Verification Mechanism

When the migration plan is on for the Existing applications all the change requests that are planned for the system functionality will not be made to the Existing system, in turn they are made to the refactored system. These changes requests are to be tracked and added to the rules engine to let know the certification engine about the change request planned for the refactored system.

Improved functionality of the refactored system includes the changes made to the system keeping in mind of the performance issues and other design and implementation challenges encountered in the existing system. These changes are to be added to the
rules engine to let know the certification engine about the improvements planned for the refactored system.

Fig: 5.6 Dash Board View

5.4 CASE STUDY OF SOFTWARE REFACTORING USING OBJECT LANGUAGE

At this juncture, we try to deliberate on the case study of software refactoring using object language, modest but extensively employed refactoring. Rename refactoring, as we see, permits the programmer to constantly modify the name of a program entity, throughout the program, while at the same time averting an undue introduction undesirable name capture that would result in alteration of the semantic principles of the programs. For a mundane instance, let us consider the Java program in Fig 5.7. Let us suppose there is a need for renaming the parameter a to x in order to suitably document its function as an initial value of the field with a similar name. As the parameter that has been renamed shadows the field x, pointers to that particular field need to be qualified by this. Consequently, the refactored program would in all probability appear like the one in Fig 5.7.
To be accurate, the ensuing steps will have to be done by the refactoring engine:

1. Ascertain and confirm that the renaming leads to a precise Java program: renaming a as b. It would not be feasible as a local variable by same nomenclature happens to be there already.

2. Bring declaring ‘a’ to reflect the name change up-to-date. After that, bring up-to-date every single reference to ‘a’ to refer to x in its place.

3. Meet the requirements to qualify all references to a variable that is shaded by the renaming.

By and large, these actions fall into the two groupings of analysis and transformation. Analyses have to be there to regulate feasibility of the renaming and to identify all references to the renamed variable. At the same time, however, we have to identify every variable that will be shaded, and to decide as to how (and if) they can be fit to avoid name capture. When these conversions accomplish the required changes on the basis of the inputs garnered by such analyses.

The analysis part would be tougher than the transformation per se., naturally, a recurrent cause of bugs in more sophisticated refactoring tools. For instance, while renaming a local variable x to y, it gives the impression of being secure merely replace every simple name x in the variable’s scope by y in step 2 above. Java, as we see, would not permit nesting lexically those local variables which are identically named. As a result, if the checking in step 1 passed, it can be presumed that no local variable y is declared in x’s scope that could capture a reference to the renamed variable. Regrettably, this heuristic flops as it falters in taking local classes into consideration.
In Fig: 5.8, an instance of a local variable x with a field y declared in its scope can be seen. Invoking method m on an object of dynamic type A will print the value of x, i.e. 42, to the console. Though instantaneous syntactic objections are absent while renaming x to y that could be trapped during the checking in the first step, (nesting of similar fields and local variables of the same name are permissible in Java). Refactoring in this context is unsuitable as the reference to x inside the anonymous class would be taken, and references to local variables cannot be competent enough to qualify. However, several popular IDEs fail in checking for this possibility and proceed with the refactoring, yielding the program in Figure-5.8, where calling m now prints 23.

It is true that all-embracing testings’ would for sure detect several bugs like this and sooner or later guarantee that, on most programs, refactoring engine works correctly. In all probability, it would encounter hindrances. Being a peculiar language, Java may be many tricky cases that go beyond the grasp of or detection and consideration of the most meticulous analysis implementer.

A proper authentication can give us comprehensive a guarantee that every possible consequence and probabilities are taken care of. But verifying and attesting

```java
class A {
    void m() {
        final int x = 42;
        new Object() {
            int y = 23;
            System.out.println(x);
        }
    }
}
```

```java
class A {
    void m() {
        final int y = 42;
        new Object() {
            int y = 23;
            System.out.println(y);
        }
    }
}
```

**Fig: 5.8 Case Study-2: Sample Java class - Wrong Refactoring**

the accuracy and suitability of a refactoring for Java is a daunting task. Preservation of its semantics, the most obvious correctness criterion addresses the issue that performing a refactoring (such as Rename) on a program P yields a program P0, the two programs P and P0 should be semantically equivalent. Whether this is a viable criterion at all, however, is not clear; A formal semantics of Java is yet to come into being. Though the Java Language Specification comprises a vigilant informal
description of all aspects of the language, formalizing this description can be arduous because the specification that results in defects and faults

```java
class A {
    public static void main(String[] args) {
        System.out.println(A.class.getName());
    }
}
```

**Fig: 5.9 Case Study-3: Java class - Reflection**

It has to be mentioned here that some structures of the java assuring behavior preservation for refactoring is impracticable: Let us take into consideration a simple program in Fig-5.9, where the foremost technique employs java reflection capabilities in order to attain a string representation of the name of class A’s, and print it to the console.

If asked to rename A to B, how should a refactoring engine respond? Mere rename exercise of the class will produce a program printing B instead of A, i.e. this transformation is not preserving behavior. At the same time, however, it would undoubtedly not be necessary to change the print statement to System.out.println("A");

Even as the carried-out alteration succeeds in preserving behavior here, it is much too invasive, and in general it is difficult to decide on where and how those kinds of modifications have to be performed to preserve precision in the semantics area.

**5.5 RESULTS AND DISCUSSION**

This section presents the result analysis of the proposed method to know the feasibility study of the theoretical work. Some of the object oriented metrics considered in this proposed work are Weight Method for Class, Lack of Cohesion in Method and Coupling between Object Classes (Efferent Coupling, Afferent Coupling).
This approach is appraised by comparing software maintainability of source code which is applied with the sequence refactoring techniques to changed Source code without sequencing refactoring techniques. All these metrics were used to appraise the efficacy of the projected technique. It is also observed that the proposed system provides better refactoring results, which is useful to the application developer. The following graphs visualizes the comparison of the metrics before and after refactoring for a given test case for different refactoring techniques.

![Graph: Composing Methods](image1.png)

**Fig: 5.10 Comparison of Metrics before and after Refactoring: Composing Methods**

![Graph: Moving Features between the Objects](image2.png)

**Fig: 5.11 Comparison of Metrics before and after Refactoring: Moving Features between Objects**
Fig: 5.12 Comparison of Metrics before and after Refactoring: Simplifying Conditional Expressions

Fig: 5.13 Comparison of Metrics before and after Refactoring: Organizing Data
Fig: 5.14 Comparison of Metrics before and after Refactoring: Simplifying Method Calls

![Simplifying Method Calls](image)

Fig: 5.15 Comparison of Metrics before and after Refactoring: Generalization

The experimental results shows for the given test case before and after refactoring technique- Composing Methods - WMC decrease from 123 to 98, LCOM increased from 4.3 to 5.101, AC decreases from 22 to 19 and EC from 14 to 11. Composing
Methods - WMC decrease from 123 to 98, LCOM increased from 4.3 to 5.101, AC decreases from 22 to 19 and EC from 14 to 11. Similarly for other refactoring techniques there as increase in LCOM, decrease in CBO and WMC which improve the quality of the system as shown in the Table 5.1.

<table>
<thead>
<tr>
<th>Refactoring Techniques</th>
<th>WMC</th>
<th>LCOM</th>
<th>AC</th>
<th>EC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>Composing Methods</td>
<td>123</td>
<td>98</td>
<td>4.3</td>
<td>5.101</td>
</tr>
<tr>
<td>Moving Features between the Objects</td>
<td>123</td>
<td>84</td>
<td>4.3</td>
<td>5.101</td>
</tr>
<tr>
<td>Simplifying conditional Expressions</td>
<td>54</td>
<td>36</td>
<td>0.78</td>
<td>1.944</td>
</tr>
<tr>
<td>Organizing Data</td>
<td>40</td>
<td>35</td>
<td>0.98</td>
<td>1.987</td>
</tr>
<tr>
<td>Simplifying Method Calls</td>
<td>29</td>
<td>27</td>
<td>0.65</td>
<td>1.971</td>
</tr>
<tr>
<td>Generalisation</td>
<td>88</td>
<td>78</td>
<td>1.2</td>
<td>2.657</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of Metrics Before and After Refactoring

5.6 SUMMARY

This chapter addresses the complete automated verification mechanism that certifies all the functional components of application and various process involved during the certification phase. This mechanism mainly saves the cost and time compared to the manual Quality Assurance approaches. Moreover, this mechanism does not need any human intervention, will provide a 100% bug free certification. This approach is appraised by comparing software maintainability of source code which is applied with the sequence refactoring techniques to changed Source code without sequencing refactoring techniques. All these metrics were used to appraise the efficacy of the projected technique.
CHAPTER-6

CONCLUSION AND FUTURE WORK

The software evolution is one of the challenging issues in today’s business environment. It is necessary for the organizations, which make use of Information and Communication Technologies (ICT) will have to align their business processes to compete with global business. The business organizations have perceived that a generous amount of existing software frameworks fail due to unstructured architectural design. The value as well as quality must be taken in to account, while choosing a software evolution approach.

Research suggests that refactoring is considered as best-practice for creating, reusing, maintaining the software system

Modern Object Oriented Programming languages provides more reusability, modularity, and understandably, maintainable with low costs, which allows the precious development resources to the business development. But, at the same time, there is a need of new requirements that can be introduced more efficiently with less problems. These new requirements should be fit cleanly without changing the internal structure of the original design with less maintenance effort. Also, Code smells are the indicator of the presence of anti-patterns in the code, the presence of which will lead to the bugs in the near future. Efficient refactoring process is required which aids to nondestructive changes to the structure of the source code and to improve code clarity and maintainability.

6.1 SUMMARY OF CONTRIBUTIONS

We have proposed an effective tool named GATAOOS to do the software archeology outspreading several of the open source procedures in vogue. ANTLR is unified in a multi-threading environment aimed at producing manifold instances. This stage then takes multiple files as input to provide operative outputs for improved analysis of OO systems. The tool will have twin phases designed to identify the smells. In the first phase, the tool tries to parse the given source code files and garners the required data including class names, declared variables and methods. In the subsequent stage, it employs this data with the view to parse all the code again in order to identify the code smells or design flaws existing in the system. The parsing and analysis approaches are
driven simultaneously to identify or spot the code smell occurrence from a source code. Finally, we have analyzed the proposed tool GATAOOS by considering 18 open source java projects with better complexity to detect the code smells. Each one is tested for presence of smells- Data Class, Primitive Obsession, Duplicate Code and Message Chains. It was observed that, the Data Class Code smell detected in most of projects (88.8%) followed by Message Chains (61.1%), Primitive Obsession (44.4%) and Duplicate Code (33.1%). We have calculated the density of smells detected by tool for better analysis, it was observed that proposed tool identified the Data Class code smell whose average density is 2.5 followed by Message Chains (1.52), Primitive Obsession (0.86) and Duplicate Code (0.36). Finally, each test case verified manually to assess the effectiveness of the tool.

Next, we have proposed a novel refactoring tool called GAFactor using Dynamic Approach. This GAFactor system detects a developer’s java code, reminds to the programmer that the automatic refactoring is available and if the programmer accepts then GAFactor complete the refactoring automatically. GAFactor automatically performs static analysis for analyzing the flow of data of the code that saves the programmer from doing error-prone work. The refactoring process model performs a series of small transformations, by preserving the behavior of the software. Even though each transformation perform little, but a series of transformations can produce a substantial restructuring using dynamic programming approach. The main purpose of using this approach is that, it provides the flexibility to application developer to take the decision based on each refactoring process step, which is small and less likely to go wrong. This process also keeps the system fully functional after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. The refactoring path is evaluated with various maintainability metrics and decision tree is constructed based on these metrics as Weight Method for Class [WMC], Lack of Cohesion in Method [LCOM] and Coupling between Object Classes [CBO] (Efferent Coupling and Afferent Coupling). The approach uses the principle of optimality. If all the refactoring paths, for a refactoring technique are optimal, the solution is optimal and those refactorings are applied on given source code. However the choice is given to the programmer to commit the refactored code.

Source code which encompasses code smells will be explored for places to be changed and refactoring techniques to be applied. Each position will be applied refactoring
techniques to create possible refactoring techniques usage paths or simply refactoring path. Each altered source code segment will be calculated for software maintainability metric. Selected source code will be a base source code to be applied refactoring techniques on the remained positions. All positions are already changed, maintainability metrics for all refactoring paths for a specific refactoring technique were calculated. From the experimental results we have obtained the cumulative metrics—(WMC,LCOM,AC, EC) for Composing Methods- (98, 5.101, 19,11), Moving Features between the Objects(84,5.126,20,14), Simplifying Conditional Expressions (36,1.944, 9, 5), Organizing Data (35,1.987,8,4), Simplifying Method Calls( 27,1.971,7,7) and Generalization(78,2.657,8,8). Now Programmer can take a decision to commit for the refactored code by comparing these metric values with threshold values selected.

It is observed that, the proposed GAFactor tool refactors the code effectively and from results, we can conclude that for each refactorings performed, cohesion (LCOM) is very high due to the sub optimal solutions. Moreover, the proposed method provides low WMC, due to less time complexity among member functions within the class in all the cases and minimum CBO.

Finally we have implemented the complete automated verification mechanism that certifies all the functional components of application and various process involved during the certification phase. This mechanism mainly saves the cost and time compared to the manual Quality Assurance approach. Moreover, this mechanism does not need any human intervention, will provide a 100% bug free certification. We have evaluated our approach by comparing software maintainability of source code before and after applying the refactoring techniques. All these metrics were used to evaluate the effectiveness of the proposed method. It is also observed that the proposed system provides better refactoring results.

The experimental results shows for the given test case before and after refactoring technique- composing methods - WMC decrease from 123 to 98, LCOM increased from 4.3 to 5.101, AC decreases from 22 to 19 and EC from 14 to 11. Similarly for other refactoring techniques there is increase in LCOM, decrease in CBO and WMC, which improves the quality of the system.
6.2 FUTURE WORK

The different applications may require dissimilar thresholds for the metric to be considered. Machine learning techniques are suitable for selecting the threshold. Using our approach for improving quality will produce better results. But the main issue is how to maintain the consistent accuracy due the overhead added by metrics measurements.

The proposed approach is implemented in Java programming language. So, the performance of the approach may be changed for other programming languages and platforms. The assumption in the approach is that there will be extensive test cases for the functionalities of the classes. If there are not enough test cases to reflect the functionalities, the dynamic analysis will not be fruitful in determining the client usage of subclass’s methods. However, ensuring the coverage of test cases is a different research topic.

The proposed approach is experimented on a limited set of projects. How the proposed approach would work for other projects is one of the future works. The accuracy of the proposed approach is determined by manually inspecting the source codes. Even though there are different standard processes for identifying the code smells from the source code and refactoring the code, there is more scope to improve the software quality of the system.