CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Software planning and development is one of the most challenging issues in today’s business environment. It is necessary for the organizations, which make use of Information and Communication Technologies (ICT) to align their business processes to compete with global business. The business organizations have perceived that a large amount of existing software frameworks fail due to unstructured architectural design. The value as well as quality must be taken into account, while choosing a software evolution approach. In supporting business goals, the system’s effectiveness will be reflected in terms of business value. Further the system quality will be influenced by business processes environment and the application software. Research suggests that refactoring is considered as best-practice for creating, reusing, maintaining the software systems. Indeed, programmers practice regularly with refactoring tools in two different occasions- normal program development phase and secondly whenever design problems arise.

Today’s Industry is using object-oriented languages for their software development as they provide more reusability, modularity. Also source code is easily understandable, maintainable at low costs that allows the precious development resources to be used economically in the business development. However at the same time, there is a need of new requirements be introduced more efficiently with less problems. These new requirements should be fit effectively without altering the internal organization of the original software design with less maintenance effort. This necessitates, refactoring tools for bump less modification of the existing software. Refactoring tools also aid avoiding code duplication and to remove the code smells in the system. For this purpose an effective refactoring process is required, which aids to nondestructive changes to the source code’s structure and to improve code clarity and maintainability.

1.2 SOFTWARE REUSE

Software reuse, as we know, encompasses reapplication of existing knowledge about software systems in order to diminish the efforts and time required to develop and maintain a different system. Software maintenance akin to software reuse. In this,
existing knowledge of a software system stands employed to advance it and come up with another similar variety that enhances it or outspreads the same. Software reuse procedures have been absorbed with operating at the code level. Besides this, design-level reuse has been another area of concern. However, in reusing a code there are limitations. The code performs and functions superlatively when the domain is slender and completely comprehended and the technology beneath this principle happens to be unchanging. At times, the software’s design lends itself to possibilities of reuse even when the code may not be flexible. However, a key problem with reuse at design-level is that there is a lack of properly spelt out exact design’s representation system.

Reuse does not happen by coincidence. We have comes with plans to reuse software and consider for software that are prone to reclamation. Reuse, naturally, necessitates and calls for the right tools, the right attitude, and appropriate techniques [100]. Nifty techniques as well as tools that provision software reuse include generational and compositional approaches [16]. The approach that is generation-based, incidentally, endeavors to reuse those specific patterns that lead to the construction of exact or customized versions. Plugging components together is the method used in composition-based model of reuse. Modification of those components is minimal or none at all so as to fashion target software systems. Code skeletons, subroutines [20] or functions are the components here. Application generators and some program transformation systems [17] are example as of generation-based systems.

It is common knowledge that designing of reusable software is rather painstaking. Reusable software, by and large, is the result of several design repetitions more popularly known as iterations. A large chunk of iterations take place after the software is reused. The requisite changes impinge the design of the software has to be reused as well as the design of other that use such. Construction of software program is facilitated in modifying or changing it, making further iterations in design easier, and making the software entity relatively re-utilizable. It is true that there are few methodical studies aimed at the validation of such an entitlement, it is a sturdy conclusion by experts that object-oriented software entity is less complicated when we try to alter or modify than conservative ones[61].

A few changes to this object-oriented software is accomplished easily by appending new subclasses. We can also add innovative maneuvers on prevailing classes, but retain a good chunk of the principal software without many modifications. Nevertheless,
object-oriented software is actually more difficult to make changes to than it might initially seem to be. Changing an object-oriented system to remove the code smells, per se, often calls for making modifications to the abstractions that are embodied in present object classes and the associations amongst those classes. This stage, as a result, takes in structural changes namely moving variables as well as other functions among classes. Similarly, splitting a complicated class into numerous classes is done by partitioning.

Naturally, when a change to the said structure is effected to a class or a set of classes, equivalent changes have to be effected somewhere else in a program. This is because of dependencies like typing, scoping and naming. It is laborious and arduous to track down the dependencies. Moreover, we need more time on our hands to continually update the program. There is no assurance that it will be free from flaws either. It is very difficult to realize the reusability benefits of object-oriented programming, as we need to have automated support in the task of effecting structural changes [46, 87].

1.3 MOTIVATION

Software reuse is one of the significant issue. It is the recycle of previously written code and is a means to the intensification of software development productivity besides quality of a software. The pecuniary benefits accruing from maintenance as a natural corollary to recycling are offset by the savings from developing something afresh. That it requires a lot money and manpower to develop software is one of the major motivations to lean towards the reuse and improvement of existing software designs. Object-oriented programming, compared with more traditional software development approaches, tends to make refactoring a feasibility by making the structural information required to program refactoring more explicit.

Refactoring plays a crucial part in the sphere of object-oriented programming. Well-known figures in the object-oriented programming cultures for sure attached premium worth to designing or redesigning software so as to achieve reusability. However designing of a program entails re-writing. In other arenas, restructuring a program may be a niftier approach to advance its design.

1.4 CHARACTERISTICS OF REUSE APPROACHES

The characteristics of reuse approaches are listed below:-

1. A reusable component gets identified: Not as modest as finding a particular match, it comprises discovering the greatest comparable element.
2. Familiarize about the constituent or component: Considering and making out what a constituent fixes, is imperative to using it, but growing one such cerebral archetype requires great effort and imagination [35].

3. Changing a module or a set of constituents: Comprehending the changes that are essential has been proved be laborious. Not many effective instruments have proven supportive help, these modifications call forth reorganization.

4. Composing the constituents: The process of composing, in itself, tends to be labor intensive. This is so if a component meets the twofold purpose of being a valuable self-regulating entity while at the same time is employed to create other suitable composite structures.

1.5 FEATURES OF OBJECT ORIENTED SYSTEMS FOR REUSE

A major chunk of object-oriented languages syndicates quite a few structures that back reusing: data abstraction, class inheritance and polymorphism.

They make available data abstraction with the help of their classes. Defining class offers encapsulation. Occasionally it may also offer information hiding, and can often help as components worth recycling.

Class inheritance, as we know, can be understood as a system that allows for the operations. In addition, we can inherit the structure of a class and recycle subclasses. Occasionally, we take recourse to inheritance to denote specialization hierarchies, where the subclasses are specializations of the superclass. One more well-known aspect of inheritance is that it supports programming by difference; the existing software can be reused by creating a subclass and embody the differences in the recently spelt out subclass.

Polymorphism is another potent technique of object-oriented languages as this is the notion that a procedure can be invoked on an object even when we do not seem to know its exact type.

1.6 CODE SMELLS

The outline of code smells is presented in this section, which will cause the performance of the code to be degraded.

In Martin Fowler’s view, the defined the code smell typically relates to some deeper problem in the system [34]. The software developers due to inexperience, time bounds
and lack of understanding follow modest coding procedures. This will lead to code smells in the system.

Initially, the term anti-pattern was initiated by Opdyke [81]. Code smells are the indicators of the presence of anti-patterns in a given code. Even though there is a small gap between anti-patterns and code smells, both the terms are capable of being put or used in the place of each other. Anti-patterns not an error but are thought of alarming programming practice.

Knowledge of the in experienced software developer’s leads to the involvement of code smells for solving an exact problem. Code smells indicates weakness in design, but technically not wrong. The presence of the code smells leads to bugs in near future, are to be seen as flaws in a design. They can be structural problems of a software too. Refactoring addresses these specific problems.

Code smell taxonomy was propounded by Martin Fowler. In [67], Mantyala identified twenty-two bad smells. Based on their resemblance, he separated code smells into seven broad groupings. For example, based on the resemblance code smells which have redundant data like lazy class, data class, duplicated code, dead code and speculative generality come under one of the category called dispensable.

Other classification of the code smells by researchers is smells within a class (code duplication, long method, comments, and switch statement) and smells outside the class (middle man, data class) [38].

There are seven categories of code smells - **Bloaters** are the code, methods and classes in vast size, **Object-Orientation Abusers** are due to inadequate application of the principles object-oriented programming, **Change Preventers** which makes program development more complex, **Dispensable** whose nonexistence makes the code clean and effective, **Couplers** that represents excessive coupling between classes or excessive delegation, **Encapsulators** such as Message chains and **Incomplete Library Class** which may stop catering to the needs of the user.

An array of software tools has been advanced to automatically detect bad smells though with differing capabilities and approaches. To see and find out if a segment of a code suffers from a malaise of bad smell(s) can be decided as it can be subjective. However, still there is a lack of proper yardsticks or norms.
1.7 REFACTORING

Refactoring is a widely known procedure among the fraternity of software engineering professionals. Martin Fowler defined as the procedure of improving the structure of a program internally albeit not disrupting the external behavior [14]. Recurrent refactoring of the code benefits programmers to make the code comprehendible, to spot and fix the bugs. This step makes it suitable for the topping up of innovative features and to program quickly. Above all that, this technique expands the design of the software. This in turn results in the improvement of overall quality of a software program.

Several suitable tools are arrived at to refactor the code in an automatic fashion. Refactoring smacks of resemblances to reverse engineering or for that matter, agile software development. One of such nimble software development models eXtreme Programming (XP), proposed by Beck [13] looks upon refactoring as one of its crucial features. Endless improvement of the design of the software via refactoring is a reality as it supports in the development and incremental evolution of the software.

Diverse researchers have suggested differing steps for the refactoring. Tom Mens and others have [73] defined six actions in the refactoring procedure. They are: Identification of the locale suitable for refactoring. Next comes deciding what specific kind of refactoring(s) should be functionally applied to the areas identified, ensuring that the applied refactoring preserves behavior and after that we proceed to assess the consequences of refactoring on the value of the software process. Finally, we have to assess the consistency between the code and other software artifacts.

1.7.1 Categories of Refactoring

Refactoring can be categorized based on level of automation into three-Fully manual refactoring which is done manually by programmers. Semi-automatic refactoring which is done by using various refactoring tools available in the market such as ReShaper, CodeRush and Just-Code and Automatic refactoring – refactorings which are prepared beforehand and applied to the whole codebase by running script.

1.7.2 Key Advantages of Refactoring

Refactoring provides four key advantages which are mentioned below.
1. The design of a software is improved by refactoring: Without refactoring, the design of the program degenerates over a period of time. When new things are added into the code, developers often do not get to have full knowledge of the original or intended design which often leads to duplicate code.

2. Refactoring makes software easier to understand: When a developer wants to refactor a certain part of the code, he firstly needs to have knowledge about what the code is doing and then he can change it which further raises his level of understanding the code.

3. Refactoring helps in finding defects: When a developer refactors a certain part of the code base, he or she gains deep knowledge about it, which in return enables him to see possible defects.

4. Refactoring helps program faster: Poor design of the software often slows developers down because they spend lots of time trying to understand the code itself and in fixing defects. The benefits of refactoring are that it helps developers create software expeditiously since this effort stops the design of the system from crumbling.

1.7.3 Refactoring Strategies

Refactoring should be a continuous process. It should be not something that is done in specific or chosen times. However, there are some guiding rules that state when to refactor, are listed below.

1. The Rule of Three: The first time something is done, it is done just like that without much effort or serious thought or plan but the second time it is slightly different and possible improvements are thought of. The third time is the time for refactoring.

2. When a function is being added: There are two possible reasons why the code should be refactored in this case. First is to better understand the code that needs to be changed when the new feature is added. The second reason is when the current code design makes adding the feature much more complicated than it could be with better design.

3. When a bug needs to be fixed: This comes from a reason that when the bug is found, it is a sign that the code was not palpable to a developer and see the bug to begin with.

4. When a code review is being done: Useful in small review groups consisting of the author of the code and the reviewer who recommends plausible changes. Together, they resolve if these changes should be applied into the code or not.
5. When a design or code smell is identified: The other time when the refactoring should also be done whenever a design or code smell is found in the code.

6. Targeting error-prone modules: It is one of the effective strategies of refactoring. These are modules that everybody is afraid of which is a good sign that they are more error-prone than others.

7. Target high-complexity modules: These are modules that have the highest complexity ratings. The quality of the code improved dramatically when the focus was on these kinds of modules.

8. Attempting to define an interface amid a clean code in relation to an ugly code and then move code over the interface: This strategy is effective in dealing with old systems. Every time the developer makes changes to a legacy system, some part of that code should be moved into the clean part of the code. If every developer in the team do this, the quality of the code will be significantly improved.

1.7.4 Application of Refactoring

1. **Pull out a reusable component**: For example, for quite a few years an industrial process control system served user requirements fruitfully. A customer is in need of a new product as there always this growing need to support new process tactics. But the user interface was well-matched only with former systems. These outworn designs originally designed and implemented without foreseeing major developments to come. They limited the user interface functions being intertwined with other, archaic purposes. They require refactoring the existing software and necessitate extracting the user interface component.

2. **Improve consistency between components**: It may be recalled that different project members work on two components of a software system to implement them separately. They don’t factor in the fact that originally these components were believed to be separate and unique. That they shared a common abstraction is not taken in to consideration. In order to reduce future maintenance costs and to make the design of the system easy to comprehend, refactoring the system is thought of as this step throws open commonalities between the two mechanisms.

3. **Supporting the iterative design of an Object-Oriented Application Framework**: An object-oriented application framework is to be perceived as an abstract design of an application. This consists of an abstract class for each key component. It is a central
object-oriented procedure to enable design-level reuse [110]. Respectable frameworks can be seen as the outcome of many design repetitions namely iterations and of course, plenty of efforts which call for structural changes.

For instance, the application framework for handling files in the operating system [62] commenced as an application of only one file format. Then, it had be stretched to manage added filing formats. Never the less, in the preliminary execution the more universal, common abstractions got mixed up with features specific to the file format it was designed to support. As a result, refactoring was called for to isolate the shared abstractions in order to expand and advance the design of the framework and there by facilitating reuse.

The research espoused at this juncture concentrated on the third area of how these refactorings are able to support the iterative design of object-oriented application frameworks.

The proposed work also covers the other two areas. This is required to improve consistency among components. Reusable components are extracted and improved when a framework matures.

1.8 CHALLENGES IN REFACTORIZING SOFTWARE SYSTEMS

1. Focusing on Manual Inspection for Selection for Refactoring

As of today, there are several nifty tools like the Refactoring Browser to boot, we can also observe other entities like IntelliJ IDEA and Eclipse. These handy tools support refactorings in a relatively safe and automated fashion. They do ensure that the refactoring is preserving behavior. These tools spare the task of the developer because it need not be manually transformed. They do not focus on the places where the code should be refactored. Hence the user is forced to realize time-consuming manual inspections as these manual verifications are required to decide where exactly these codes can be applied to accomplish refactoring. In addition, there are several other handy tools designed to perform automatic code inspections. They include LINT for C, LINT for Java, SmalltalkCodeCritic provided by the Refactory Browser and TogetherCC. They help to locate and spot exact locations places where the code has bugs or other technical defects. These include error handling, misuse of operators, unused code or entities that do not follow the usual norms or coding standards.
2. Lack of Addressing Advance Design Issues

It has to be borne in mind that sophisticated design issues are not paid attention to like the bad smells defined in Fowler’s Refactoring book. The time and effort needed to locating and identifying the right place to refactor, spotting the proper refactoring and the parameters involved and monitoring to improve the overall quality. This then becomes a painstaking responsibility of the developer.

3. The daunting task of working on voluminous code bases and copious amounts of inter-component dependencies

There is a need for coordination with other teams or developers. It also entails that one has to take up confirming the correctness of the said program after refactoring.

4. Non availability of tool support intended to refactor change integration

Code review tools, we can see, target refactoring edits. More advanced refactoring engines assist the user in defining new refactoring types. That it is a laborious and cumbersome process, merging and integrating these refactorings. Naturally, the users are often disinclined to opt for refactoring.

1.9 PROPOSED WORK

One of the possible solutions is to refactor or modernize existing systems into a new platform or version in existence.

For refactoring existing systems, one has to recognize and comprehend the present system in all ramifications. To investigate the present system, researchers offered their contributions to this field to suggest diverse methodologies. That understanding the software calls for proficiency regarding crucial functional aspects of the domain in addition to the platform itself, the technology besides of course, the architecture. The whole process revolves around time and effort. It is very difficult to get the skilled resource personnel with the requisite knowledge of the system. Therefore, automating the whole process of understanding, analyzing, and detection of code smells in the software systems is required.

Firstly, we analyzed various techniques proposed for understanding software systems and proposed a methodology(GATAOOS) for understanding object oriented software systems and to detect code smells in the system.
Refactoring is a transformation achieved from miniature modification, conserving their behavior in the meantime. Though every transformation (called a 'refactoring') effects miniscule changes, a succession of transformations tends to yield a substantial restructuring [14]. The supposition is that as refactoring is slight, it's less prone to be erroneous. Despite each small refactoring, the system works well as usual. This feature eliminates the risk of fatal system break down even as the process of restructuring continues unabated [84].

Subsequently, we have proposed a novel refactoring tool called GA Factor. GA Factor automatically performs static analysis for analyzing the flow of data of the code that saves the programmer from doing error-prone work and refactors the code to eliminate the code smells.

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.

Finally proposed a complete automated verifiable mechanism which certifies all the functional components of application and various processes involved during the verification phase. The mechanism ensures the external behavior of the refactored system is intact with existing system.

1.10 CONTRIBUTIONS

In this dissertation, we investigated and implemented the above proposed work, stated as three objectives as listed below.

1.10.1 Objective-1: To Understand and Analyse of Object Oriented Software Systems

It is necessary to analyze the existing system for better understanding the business logic and its functionalities. We have analyzed various techniques for understanding software systems and a methodology is proposed for understanding software systems and to detect code flaws in the system.
Overview of ANTLR Parser:

There are many open source tools that facilitate an understanding of the systems. ANTLR can be said be a replacement to the Purdue Compiler Construction Tool Set (PCCTS). This was first advanced in 1989. This tool is under vigorous development. ANTLR, as we see, can perform superior analysis of a software system. ANTLR (ANother Tool for Language Recognition), is a leading generator for parsing activity which helps us in studying, analyzing, executing. It is also handy in the translation of binary or text files. ANTLR develops a parser which can be of use in building parse trees with the support of the grammar input.

Limitations of Existing ANTLR Parser

1. Using this tool we cannot call methods as argument to the other method call.
2. Whenever a nested methods call is invoked this tool finds difficulty in identifying the hierarchy of the method called.
3. In method or constructor overloading parser again faces the problem of addressing the exact call hierarchy.

To achieve this objective, we proposed a methodology, which analyses OO Systems using ANTLR in multi-threading environment and to detect code flaws in the system. Multiple instances are created and they take manifold files as input. They deliver effective outputs for a superior systems analysis and detects the bad smell in a code (especially in java).

Proposed Architecture for Understanding Object Oriented Software Systems (GATAOOS)

To accomplish this objective, a prototype tool, GATAOOS is developed. This takes as input a set of java classes. The defects in the design appear as the results in the java system in the form AST which is accomplished via visualization.

A parser generator, ANTLR, produces the code that is employed in parsing the input code. ANTLR specialty is that it accepts bulky class of grammars. It moves to shape the parse trees in case of a specified input of the program.

In due course, ANTLR may be combined with multi-threading environment, capable of creating multiple instances, multiple files as input. At the end it provides active outputs for improved analysis of the system. The bad smell symptoms get apprehended and underscored in the GATAOOS that gather the statistical results.
The tool will have twin phases designed to identify the smells. In the first phase, the tool tries to parse 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 parsing 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.

The parser by performing analysis of source files (java) generates Abstract Syntax Trees (AST), which are interfaces between parser and later stages of the compiler. Syntactic structure of a source program is preserved by AST. The code smells can be identified by traversing these trees.

After the completion of parsing, Tree traversal is done by an analyzer. The traversal of the AST make it possible to identify the code smells which are preserved for later stage. Using the repository and Grok scripts, it possible to derive complex code smells which can be presented by explicitly calling a method that parses a java file.

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 Duplicated code (33.1%). We have calculated the density of smells detected by tool for better analysis. It was observed that proposed tool identified effectively 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 verify the effectiveness of the tool.

1.10.2 Objective-2: To Model a Generic Automated Refactoring Tool for Object Oriented Software Systems

We proposed a novel refactoring tool called GAFactor. The proposed system reminds to the programmer that the automatic refactoring is available and if the programmer accepts then GA Factor complete the refactoring automatically.

The refactoring process model performs a series of minor transformations. Though each transformation does a wee bit, they are significant. However, a sequence of
transformations is designed to 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 step, which is small and less likely to go wrong. This process also keeps the system fully operational after slight refactoring. This in turn helps in diminishing the likelihoods that a system can get debilitated during the restructuring activity.

The refactoring technique is selected by evaluating maintainability metrics and decision tree is constructed based on these metrics. Some of the object oriented metrics considered in this proposed work are Weight Method for Class (WMC), Lack of Cohesion in Method (LCOM) and Coupling between Object Classes (CBO) which is AC (Afferent coupling) and EC (Efferent coupling). The approach uses principle of optimality. These metrics were used to evaluate the effectiveness of the refactoring paths. All the refactoring paths for a refactoring technique are optimal, then 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 paths. Each altered source code segment will be calculated for software maintainability metrics. 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.
1.10.3 Objective-3: To Verify the Automated Refactoring Tool for Object Oriented Software Systems

There is a need to verify the refactoring process, because 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 system before refactoring intact after going through the process of migration. We have to provide verifiable the object-oriented software refactored system that certifies all the functional components of an application.

We have calculated the maintainability metrics for the refactored code and compared with metrics of the code before refactoring, to ensure the effectiveness of the system.

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 as increase in LCOM, decrease in CBO and WMC which improve the quality of the system.

1.11 ORGANIZATION OF THE THESIS

The remainder of the thesis is ordered in the following manner.

Chapter 2: Literature Survey

This chapter provides systematic literature review on taxonomy of the code smells, refactoring techniques and tools in existence for analysis of software system, identification of code smells or anti- patterns and refactoring with respect to code smells. Various researchers proposed different mechanisms for code smell detection, diverse tools in refactoring systems. This survey summarize the existing tools /frame works proposed and elucidates finally findings of survey.

Chapter 3: Generic Automated Tool for Analysis of Object Oriented Software Systems (GATAOOS)

In this chapter we have proposed a tool called GATAOOS for analyzing OO Systems using ANTLR Parser in multi-threading environment. The proposed tool will have twin phases to design to identify the smells. In the first phase, the tool tries to parse entire the java source code files and garners the mandatory data including class names, variable declarations and method declarations. In the subsequent phase, it employs the statistical data with the view to parsing all the code again in order to identify the smells
existing in each of them. For every smell, the parsing and analysis methods are driven to identify or spot the code smell occurrence from a source code. Finally we have developed the Algorithms for detecting the code smells like Data Class, Primitive Obsession, Duplicate Code and Message Chains. We have taken different java projects as test cases and analyzed the tool by finding density of the code smells for each project. The implementation and results are presented.

**Chapter 4: A Novel Refactoring Tool for Object Oriented Software Systems (GAFactor)**

This chapter address 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 very little, but a series of transformations can yield a substantial restructuring using Dynamic programming approach. The implementation and results are presented.

**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 have compared the metrics of the software system under consideration before and after applying the refactoring techniques.

**Chapter 6: Conclusion and Future Scope**

This chapter gives the conclusion and indicates the future research directions.

**1.12 SUMMARY**

This chapter presented the overview of software reuse, code smells and refactoring. This chapter explored various research issues of refactoring techniques. Some of these major research issues have been addressed in refactoring the Software Systems Next chapter is confined to describe, taxonomy of code smells, refactoring techniques in use, the related work of analysing the existing software systems, code smell detection techniques, existing refactoring tools and its vulnerabilities.