CHAPTER 5

REVERSE ENGINEERING ASSISTANCE

5.1 INTRODUCTION

The ability to reverse engineer object-oriented systems has become a vital matter in today’s software industry. Formal methods are nowadays the most rigorous way to develop software. However, people find it difficult to use and understand the existing formal notations. The chapter proposes an approach to circumvent this shortcoming by providing Software quality indications for UML based projects developed in Java.

5.2 REVERSE ENGINEERING

For evolving object-oriented software, the design changes are the critical factor. A reverse engineering technique should exploit whatever information is available concerning the changes of a piece of code. The change based reverse engineering would be more scalable, because it focuses reverse engineering efforts on changing pieces of code instead of the full code base. The research involves in implementing a modular reverse engineering technique for object oriented medium sized projects designed through UML for Java environment.

The research analyzes the drift in design from the earlier version of the software to the current version after a Quality indication supplied by the Design Phase Tool. The Design Phase Tool assists Reverse engineering tasks as it supplies the change information in the software which is captured in the design stage itself for later verification. The research takes critical analysis of
the following object oriented metrics, class size, method size, Effort Estimate, and inheritance.

The research differs from Moose metrics, which is a related work \cite{24} in reverse engineering which lays more emphasis on reliable statistical information for analysis and also the easy implementation to the development environment. The research has implemented the Reverse engineering procedure through an extension to the Design Phase Model developed with Java.

The Software Assessment Model assists the reverse engineering tasks as it supplies the change of information in the software which was captured in the design stage itself for later verification. The research takes critical analysis of the following object-oriented metrics, such as the class size, method size, Effort Estimate and inheritance. The research differs from MOOSE (Lanza 2003) metrics, a related work in reverse engineering, which gives more emphasis on reliable statistical information for analysis and ease of implementation to the development environment. The reverse engineering procedure was implemented as an extension to the Software Assessment Model.

Reverse engineering existing software systems has become an important problem that needs to be tackled. It is defined (Chikofsky & Cross 1990) and Cross as “the process of analysing a subject system to identify the system’s components and their relationships and to create representations of the system in another form or at a higher level of abstraction”. Reverse engineering forms the prerequisite for the maintenance, reengineering and evolution of software systems. Since an unwary modification of one part of a system can have a negative impact, it is first necessary to reverse engineer the software system before it can be modified or reengineered. Sommerville
(2000) and (Alan Michael 1995) estimates that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system.

It would thus seem advisable to rewrite software systems as soon as it fails to fulfil their requirements. However, certain software systems are too valuable to be replaced or to be rewritten, because of their sheer size and complexity which makes such a feat too expensive for the owning company in terms of time and money. In case of such legacy software systems, it is more advisable to first reverse engineer and then maintain, reengineer and evolve such systems. By adapting them to new requirements (Casais 1998), the lifetime of these systems can be extended which increases the return of investment of their owners. Indeed, the longer a software system can be used, the better it pays off for the company that developed it.

The research has mainly concentrated on the reverse engineering of object-oriented legacy systems, because, most current software systems are written in languages implementing this paradigm. This becomes vital, since it is not age that turns a piece of software into a legacy system, but the rate at which it has been developed and adapted. Moreover, early adopters of object-oriented technology are discovering that the benefits they expect to achieve by switching to objects have been very difficult to realize (Dekel 2002). The adopters also find themselves with present and future legacy systems implemented with object-oriented technology.

Moreover, reverse engineering object-oriented software systems comes with additional challenges (Wilde & Huitt 1991) compared to non object-oriented systems, such as polymorphism, late-binding and incremental class definitions.
5.2.1 Reverse Engineering Concern

Before a software system can be reengineered and maintained, the system must be reverse engineered, for example a mental model of the software needs to be built which allows for making informed decisions. Reverse engineering software systems is difficult due to their sheer size and complexity. However, it is a prerequisite for their maintenance, reengineering and evolution. Maintaining and evolving existing software systems is difficult because of several reasons, such as the accelerating turnover of the developers, the increasing size and complexity of software systems and the constantly changing requirements of software systems. These legacy systems are large, mature and complex systems, which are valuable to a company and must therefore be maintained and evolved (Rugaber & White 1998). However, as (Parnas 1994) assessed, most legacy systems suffer from typical problems:

- The original developers are no longer available and often they are inexperienced developers without system or domain knowledge required to maintain and evolve the systems.

- Outdated software development methods or programming languages have been originally used to develop the system and it is hard to find people knowing these techniques or willing to master them.

- Extensive modifications and patches have been applied to the system, triggering a phenomenon called the architectural drift, which makes it hard to understand the original design of the system.

- The documentation is outdated, incomplete or completely missing. Since legacy systems tend to be large, thousands of
lines of poorly documented code are no exception and hence there is a definite need for effective approaches which helps in program understanding and problem detection.

Moreover, compared with procedural systems, the reengineering and reverse engineering of object-oriented software systems pose many additional challenges (Wilde & Huitt 1992) such as polymorphism, late binding, incremental class definitions by means of inheritance and the dynamic semantics of self. Furthermore, in object-oriented systems, the domain model is spread over classes residing in different hierarchies and subsystems. Indeed, the problem of reverse engineering the object-oriented software systems needs to be tackled.

5.2.2 Object Oriented Reverse Engineering

Reverse engineering can be defined as part of the reengineering lifecycle (Rajlich & Bennett 2000); (Dekel 2002) define reengineering as “the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form”. Forward engineering, on the other hand, is defined as “the traditional process of moving from high-level abstractions and logical implementation independent design to the physical implementation of a system” (Chikofsky & Cross 1990). Therefore, forward engineering is about moving from high-level views of requirements, towards concrete realizations. Reverse engineering is about going backward from some concrete realization to more abstract models and reengineering is about transforming concrete implementations to other concrete implementations.

• Forward engineering can be understood as being a process that moves from high-level abstract models and artifacts to increasingly concrete ones.
• Reverse engineering reconstructs higher-level models and artifacts from the code.

• Reengineering is a process that transforms from one low level representation to another, while recreating the higher-level artifacts along the way (Dekel 2002).

5.3 GENERAL PRACTICES IN REVERSE ENGINEERING

The goal of reverse engineering a software system is to build progressively refined models of the system (Storey et al. 1999) which enables to make informed decisions regarding the software. This is not a complex problem for small software systems, where code reading and inspection is often enough. In the case of legacy software systems which tend to be large, thousands or millions of lines of poorly documented code are no exception. This becomes a hard problem due to their sheer size and complexity. In order to build a progressively refined mental model of a software system, the reverse engineer must gather information about the system which helps in this process.

There are many approaches to reverse engineering software systems, such as:

Reading the existing documentation and source code: Various people have investigated code inspection, code reading and code review practices (Demeyer & Stephane 1999); (Demeyer et al. 2000). Using this approach is difficult when the documentation is obsolete, incorrect or not present at all. Reading the source code is widely used in practice, but does not scale up, as reading millions of lines of code would take weeks or months without necessarily increasing the understanding of the system by the reader. Moreover, at the beginning of a reverse engineering process, one does not
seek detailed information, but rather wants to have a general view of the system.

**Running the software is to generate and analyse execution traces:** The use of dynamic information gathered during the execution of a piece of software, has also been used in the context of reverse engineering (Richner & Ducasse 1999); (Richner & Ducasse 2002), but has its drawbacks. In terms of scalability, traces of a few seconds can become very big and interpretation of thousands of message invocation can hide the important information which is searched for.

**Interviewing the users and developers:** This aspect gives an important insight into a software system, but is problematic because of the subjective viewpoints of the interviewed people. It is always hard to formalize and reuse the insights proposed by the user. Moreover, it is also hard to find developers who have been a part of the development team over long periods of time and thus possess knowledge about a software system’s complete lifetime.

Various tools (visualizers, slicers, query engines, etc.) and techniques (visualization, clustering, concept analysis, etc.) can be used to generate the high-level views of the source code. Tool support is provided by the research community in various ways and visualization tools like (Muller 1986) and (Storey & Muller 1995) are widely used.

**5.3.1 Phases in Reverse Engineering Process**

Before applying reverse engineering, it is essential to decide the primary goals and to differentiate the secondary goals. This work presents a short canonical list of primary goals that have been inferred from the
engineering experience which is a subset of the goals Reverse Engineering Life Cycle is shown in Figure 5.1.

5.3.2 Reverse Engineering Achievements

The goals of a reverse engineer are to identify and locate the important inheritance hierarchies. The reverse engineer also gathers answers to the question about the size of the subject system. Moreover, it helps to detect exceptional classes in terms of number of methods or number of attributes.
5.4 REVERSE ENGINEERING TECHNIQUE

According to (Parnas 1978), software engineering deals with the construction of the multiversion software. The software undergoes a number of revisions either to enhance the functionality or to fix bugs. The ability to perform corrective, adaptive or perfective maintenance (Black 2001) in an existing object-oriented design has been associated with many closely related terms such as changeability (Arisholm et al. 2001) maintainability (Ghezzi et al. 2002) and robustness to changes (Gamma 1995) extensibility (Riel Arthur 1996) and flexibility (Martin 2002).

A number of terms have also been used in order to describe the key features in design such as rigidity, fragility, inflexibility and limited resolvability. In a context, where all of the above terms are usually used, are prone to changes that are unavoidable in software development and that anticipation of these changes by a software designer is essential.

Figure 5.2 Reverse Engineering Model
In order to characterize several aspects of a software system a large number of metrics has been proposed (Fenton & Neil 2000). Since the initial work of (Chidamber & Kemerer 1994), in the field of software metrics has been also expanding to the object-oriented domain as well, many of these metrics have been both theoretically justified and empirically validated, while others lack a systematic validation on real-life industrial software. However, it is believed that most of the existing metrics evaluate the degree of object-orientation or measure the static characteristics of the design, which are not always helpful in answering the question as whether a specific design is good or bad (Kirsopp et al. 1999). When attempting to answer such a question, an expert would assess the conformance of the design to the well established rules of thumb, heuristics and principles. This work attempts to standardize this process by quantifying the change proneness of each class in a design.

The goal is to assess the possibility that each class will change in future. The proposed model is to evaluate the change proneness of the process and it has been fully automated using Java. The statistical analysis has been proved to have improved correlation between the extracted probabilities and the actual changes in each of the classes in comparison to a prediction model that relies simply on the past data. Rather, it should be applied when several successive versions of an application are available. In order to calculate these probabilities and axes of change, the change in one class that affects the design of other classes is to be identified.

The Reverse engineering approach is an extension of the previous Software Assessment model designed to provide quality indication for UML based small and medium sized object-oriented java projects. The concept was assisted through a software model designed for this research (Software Assessment Model). The model uses a modular approach to object-oriented metrics and records the classes that comply with object-oriented standards and
indicate visual suggestion for classes, which has deviation. The model also has provisions for recording the results of the Software Assessment evaluation with metrics evaluation, which results in a database for future reverse engineer’s reference.

The research model shown in Figure 5.2 provide straightforward values of classes, attributes, methods, depth of inheritance, number of methods in a class, number of attributes in a class and their deviations from the object-oriented metrics to the reverse engineer. The values are presented in a straightforward way in this model, which have some advantages over the other approaches.

The research model suggests certain metrics which has a great influence on Reverse Engineering metrics. They are 1.Class Size Metrics, 2.Inheritance Metrics, 3.Number of Methods Metrics, 4.Number of Attribute Metrics, 5.Depth of Inheritance.

5.4.1 Related Works

Several researchers have contributed towards reverse engineering from the year 2002. Some of the works like Polymetric view based on code crawler (Niertrasz et al. 2005) as well as Moose Metrics (Lanza 2003) are referred. Both the research has taken huge sized projects and has generated visualized information to the reverse engineer. The proposed model uses small and medium sized projects, which provides direct values with object-oriented metrics.
Limitations of Polymetric approach

**Layout Algorithm.** The approach presented here relies heavily on an efficient layout algorithm in terms of space and readability. Especially, in the case of very large classes, having hundreds of methods, it may happen that the only real statement that can be made is that the class is large (the Giant blueprint). However, patterns often occur in such classes providing important pieces of information.

**Functionality.** The blueprint of a class can give the viewer a ‘taste’ of the class at one glance. However, it does not show the actual functionality that the class provides. The approach proposed here is thus complementary to the other approaches used to understand classes.

**Collaboration.** The approach does not address collaboration aspects between classes for the time being.
**Static Analysis.** The approach presented here does not make use of dynamic information, which ignores runtime information about the methods that actually get invoked in a class. This is especially relevant in the context of polymorphism and switches within the code. In this sense, the class blueprint can be seen as a visualization of every possible combination of method’s invocation and attribute accesses.

Mere visualization of the classes without measurement was found to be the major drawback of this model represented in Figure 5.3. Hence, the proposed model tries to overcome the above stated limitations by measuring and providing the value of classes, attributes methods, DIT, number of methods in a class and the number of attributes in a class.

Table 5.1 summarizes related works

**Table 5.1 Related Works**

<table>
<thead>
<tr>
<th>Author Name</th>
<th>Approach</th>
<th>Metrics</th>
<th>Limitations</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michele Lanza and Stephane Ducasse</td>
<td>Polymetric view – A light weight visual approach – using code crawler for C++ - a non linear approach [Polymetric]</td>
<td>Class metrics, Method metrics, Attribute Metrics</td>
<td>Can be used one or two weeks of a reverse engineering process. Requires traditional approach and systematic</td>
<td>2002-2003</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Method/Model</td>
<td>Metrics</td>
<td>Emphasis</td>
<td>Project Period</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>---------</td>
<td>-------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Duraisamy et al</td>
<td>HLDQI</td>
<td>All the above metrics and also emphasis on CBO and RFC.</td>
<td>Medium sized projects</td>
<td>2004-2006</td>
</tr>
</tbody>
</table>

### 5.4.2 Reverse Engineering Support

Most of the legacy systems tend to be large with hundreds and thousands of lines of poorly documented code. There is a definite need for effective approaches that help in program understanding and problem detection. Since the object-oriented paradigm does not support a sequential reading order the domain model is distributed across classes, hierarchies and subsystems, the reverse engineer needs to know where to look into the system to understand its structure.

This approach helps the reverse engineer to get a mental picture by viewing the small or medium sized object-oriented system. This is enhanced by means of a Software Assessment Model with traditional metrics as well as new metrics to capture the complete object-oriented programming issues. The model captures the approach of the system in the design stage itself in a modular way and indicates the imperfections. The model also records the major structure of information along with design imperfections as history of
the project. This valuable piece of information can surely assist reverse engineer during any of reengineering or maintenance requirement. The model supplies real values for the system, which can greatly assist the reverse engineer to quickly perform the reverse engineering task in terms of

- Providing an easy interface to investigate the object-oriented system and gather important information like the Effort Estimate, Effort Estimate complexity and variations from the object-oriented standards.
- Providing object-oriented metrics values of the class.
- Providing both project based and individual class based investigation approach to gather information on the object-oriented systems.
- Providing record option of the results for future maintenance or reverse engineer reference requirements.

**Software Quality Indicator**

The Quality Indicator methodology suggest a practical standard during the Software Assessment of a small and medium sized object-oriented system and validate the design with the standard values. The standards were based on the practical feedback collected from several industrial object-oriented developers and followed by industries.

Based on the quality of design, Quality Indicator suggests the following: If, the input design (Java source code generated by UML)

- exceeds the Quality Indicator standards, it strongly recommends for redesign.
• is equal to Quality Indicator standards, it needs still more optimization in the design.

• is less than Quality Indicator standards, it strongly agrees to Quality Indicator and is considered to be stable, maintainable and easily understandable with reduced maintenance cost and also extends the functionalities for any reverse engineering or reengineering requirements.

The model gives a visual indication (red color) for variations from the Quality Indicator standards. During the design imperfection scenario, the tool also assists in calculating the redesign cost efforts based on metrics suggested (Marinescu et al. 2000) for better optimization of the project at an earlier stage of development phase.

The research has provided an exploratory nature of its approach to deal with reverse engineering. The model provides options for the reverse engineer to deal with the reverse engineering work. The options are class level or method level views of the metrics as well as project level views of metrics. It also has the following aspects:

**Simplicity:** The tool provides features to work with modular nature of the project. Taking the object-oriented system from individual class, it can also consider the following constraints like limited man power and short time constraints. It provides statistics of individual classes, project as a whole and also provisions for regenerating the results for maintaining the consistency of the results.
Scalability: The research is designed to approach small and medium sized project up to 10,000 Lines of Code. The future aspect is to scale the research to Huge Industrial projects.

Language Independence: The system is developed with Java which is a platform independent language. The architecture of the tool is also carefully designed, not to include any platform’s specific functionalities.

5.5 PERCEPTION

The tool available at present for reverse engineering provides only a limited metrics for visualization and consolidating the idea. Also, it has an implicit drawback of identifying individual classes or methods with design imperfections. The Quality Indicator research proposes the reverse engineering efforts through a model which is designed to reduce the trouble.

The model supports various views and exhibits different layouts. The tool has the following views viz., Class Metric view, Method Metric View and Chart View. A library management software where admin can add/view/delete librarian and librarian can add/view books, issue, view issued books and return books. The class metrics view displays the class name and the root of the class with all the metrics captured by the model as shown in Figure 5.6 and 5.7.
Figure 5.4 Method Data - ViewIssuedBooks.java

Figure 5.5 Method Data - IssueBookForm.java
The Method view, which provides an Individual Method Data view of the object-oriented metrics, providing an individual metrics view of the class is shown in Figure 5.4 and Figure 5.5.

Figure 5.6 Class Data - ReturnBookDao.java
Figure 5.7 Class Data - ViewLibrarian.java

The model also supports the Project Chart view, which provides a chart view of the metrics for various classes. Saved Metrics view provides the facility to load the saved metrics view of a previously validated project for the system analyst as shown in Figure 5.8.
In this chapter, The Reverse engineering approach is an extension of the Design Phase Model designed to provide Design phase quality indication for UML based small and medium sized Object Oriented projects developed with Java. The concept is assisted through a Software tool designed for the Design Phase Model. The model uses a modular approach to object oriented metrics and records the classes that comply with object oriented standards and indicate visual suggestion for classes, which has deviation. The model also has provisions for recording the results of design phase evaluation with results of metrics in database for future reverse engineer reference.

The research model provides straightforward values of classes, attributes, methods, depth of inheritance, number of methods in a class,
number of attributes in a class and their deviations from the Object oriented metrics to the reverse engineer. The values are presented in our model in a straightforward way. The Research Model suggests the Class Size Metrics, Inheritance Metrics, Number of Methods Metrics, Number of Attribute Metrics, and Depth of Inheritance as some of the metrics that has greater influence on reverse engineering.