Chapter 5: Conclusions and Future Directions

The concept of reuse helps to control cost, schedule, and quality of a software product. In this approach, a software application is assembled using already existing reusable software components. However quality requirements for these reusable components are stringent as application quality depends upon the quality of its worst constituent (individual components, connectors, and middleware). In addition to own attributes several other factors such as architectural constraints or context dependencies, affect the quality of a reusable software component. Unfortunately there is no general consensus on the quality characteristics of software components but there are numerous studies that relate quality attributes of a software product to its design. Goodness of design is one of the prerequisites for realizing a quality product or for that matter a software component.

As a software component matures it goes through several rounds of testing. Every (re)user verifies the software component before selecting it for integration in his/her application. Several rounds of testing ensure that software component becomes defect free after a period of time. Therefore it can be expected that as a component matures or evolves, its design improves as well. Otherwise as per Lehman’s seventh law of evolution, quality of software degrades if it is not managed in successive releases.

The research in this thesis revolves around the concept of design evaluation of a reusable software component as it evolves. The software component under study is a JAVA based industrial strength software program, called JFreeChart, available at www.sourceforge.net, one of the largest repositories of open source software. It is not an end application in itself, but can be integrated as a component while assembling new applications.

Research literature suggests several metrics to evaluate design of a software component from two different perspectives: internal design and external or interface design. But most of such metrics proposals are still in infancy and are not supported by any rigorous validation studies. This study analyzes 43 versions of the JFreeChart software component using object oriented metrics at three levels: system, package, and class. Object oriented metrics help to measure the usage of basic concepts of the object oriented paradigm such as abstraction, inheritance, polymorphism, coupling, and cohesion while designing applications based on this paradigm. These concepts of the paradigm are empirically
validated to be related to creating reliable, resilient, and easily maintainable designs. Main objective of this thesis is to investigate as to what extent the concepts of object orientation are incorporated in the design of this software component and how the usage of these concepts improves or degrades as the design evolves over a passage of time. Another objective is to analyze the maintenance activity to investigate the suitability of the chosen metric set to differentiate between classes of different categories such as unchanged, changed, removed, and added in between any two releases of the software component. This research also investigates a few more hypotheses regarding the maintenance activity.

Following conclusions are derived after analyzing the trends in metric values:

- System level metrics attribute hiding factor (AHF) and method hiding factor (MHF) show that the software component incorporates the concept of information hiding. As the component evolves, AHF almost achieves its ideal value i.e. 100%. MHF is high for the initial releases and as the functionality increases MHF decreases in successive versions. MHF has to be low for increased functionality as more public members are required to support more functionality. However, MHF still remains within the suggested range of 8%-25%.

- Attribute inheritance factor (AIF) and method inheritance factor (MIF) also show significant use of inheritance. Both the metrics decline for the later versions with fall in AIF value very sharp from JFreeChart 0.9.20 to JFreeChart 0.9.21. This may be due to increase in average class size as the denominator for these metrics is the sum of attributes of all the classes.

- Polymorphism factor (PF) also follows a downward trend across the releases. There is considerable use of method inheritance (as shown by MIF metric) but only a small number of inherited methods are redefined in sub classes. Redefinition of a large number of inherited methods is not at all desired as well.

- At system level, the metric coupling factor (CF), for measuring coupling between classes, also shows a decreasing trend. Actually this metric does not take into account inheritance based coupling. As seen with the help of class level coupling metrics, coupling between classes across the hierarchy is very less in comparison to coupling between classes within the same hierarchy. Downward trend of this metric also shows the same information. Anyhow this is a good design practice.

- Package level metrics measure the design quality of packages. The package size metric shows that on average package size has increased but is still manageable and
all the packages have been assigned almost the same number of classes i.e. package responsibilities are now more evenly distributed.

- Average package cohesion has increased over time though it is below the range suggested in the research literature.

- Over the passage of time, packages have become more concrete i.e. their level of abstraction has decreased. In the initial versions abstraction level was high. Anyhow abstraction and instability both have become well balanced over the period of time as shown by the trend in normalized distance from the main sequence (Dn) metric.

- Average class complexity has increased in successive releases. Distribution of functionality in various classes was almost equal in the beginning but in the latest releases some of the classes are assigned more functionality than others.

- Class interface size was initially small but it has grown over the period of time. This information is also available from system level metric MHF. There is a tradeoff, as the functionality increases interface size also increases. However a larger interface size is not a good design property as it leads to more class interactions and hence difficulty in maintaining the design.

- Coupling metrics at class level indicate that nature of class coupling is good. A few classes act as utility or library classes and export information to a large set of classes. Over the time, the variation in classes for export coupling has increased that may be due to the fact that classes which export are core classes and as the number of classes increases across various versions, variation also increases. Coupling between classes across the hierarchies is less in comparison to coupling between classes within the same hierarchy.

- Inheritance tree has become broad and shallow towards the recent versions. In the initial releases it was narrow and deep. A broad and shallow tree is better for reusable software as it indicates a general design and can be extended easily for unforeseen situations.

- Existing design level class cohesion metrics suffer from various anomalies. An improved of the existing cohesion metric NHD is proposed in this research and named as NHDM (NHD Modified). This metric is theoretically as well as empirically validated.
Cohesion metrics does not indicate good class design throughout the various releases. NHD metric does not show much variation during the design evolution. It takes value in the higher range around 0.60. Other cohesion metrics such as CAMC and NHDM show a decreasing trend in metric values towards the latest versions. Change in values of NHDM metric is sharper than that of CAMC that may be due to more variation in values of NHDM metric (CAMC has false positives as well). Initially CAMC is high (may be because of smaller class size in the beginning) but starts decreasing towards the end (class size is large in this period).

However SNHD metric shows a positive trend for cohesion of classes for the recent versions. SNHD metric indicates that cohesion level of classes lies in the negative range (-1, 0). Intuitively variants (with self parameter) of cohesion metrics show better level of cohesion for classes. SNHD\textsubscript{s} lies in the range (0, 1) so metric values for classes with expectedly better levels of cohesion lie in the upper range. For cohesive classes, NHD is more near to its maximum than its minimum. It indicates that SNHD is not a true measure of cohesion as it shows that NHD is close to its minimum for cohesive classes. It also shows a positive trend towards the recent versions.

CAMC and NHD metrics both are largely influenced by class size. NHD is large for classes large in size whereas CAMC is large for classes smaller in size and vice-versa.

CAMC and NHDM (the metric introduced in this research) follow each other very closely. NHDM has the largest amount of variation among all the cohesion metrics.

A comparison of metric values for the reusable software component release JFreeChart 1.0.11 with metric threshold values as suggested by researchers and practitioners indicates that except a few (such as package cohesion) metrics, all others are well within the prescribed limits. Comparison with thresholds identified using statistical techniques also shows that number of outliers is very small.

Identified metrics successfully differentiate the quality of added classes from removed classes and unchanged classes from changed classes. Only exception is the first evolution from JFreeChart 0.9.20 to JFreeChart 0.9.21, in which metric values are not as desired. Cohesion metrics NHD and NHDM\textsubscript{s} behave inconsistently.
Other metrics in this category consistently report better cohesion level for expectedly better set of classes (added and unchanged).

- Maintenance activity takes place at deeper levels of hierarchy. Classes that are changed or removed belong more evenly to different levels of hierarchy in comparison to the unchanged or newly added classes. It contradicts the hypothesis that developers mostly do not touch the classes deep in the hierarchy tree for maintenance activities. Developers do not remove or add classes from specific locations of the hierarchy that are convenient to manipulate.

- Maintenance activity does not result in any major changes in the organization of the inheritance hierarchy before and after an evolution.

The research work presented in this thesis can be extended in future in various dimensions such as:

- This research work can be further explored by analyzing the metric trends for a larger set of reusable software components from various domains. Measurement results obtained from such analysis may be used to determine the metric thresholds to guide development of reusable components.

- These metrics can also be validated on commercial components to understand the differences in characteristics of commercial and open source software components.

- Another potential area for future research is to define component interface metrics using a formal specification language such as unified modeling language (UML). It is necessary to automate the metric collection process to validate the metrics empirically.

- No doubt a lot of work has been reported to define a component quality model but still there is no consensus on the quality attributes that a reusable software component should have. So in future research work can be done to define component quality factors for different types of components (such as general or safety critical) and link with corresponding metrics to measure presence/absence of the quality attributes. It is also important to specify this quality information in a universally acceptable format.