Chapter 1: Introduction

The recent recession, which hit the globe on 15th September, 2008 (the day Lehman brothers filed for bankruptcy), has seriously impacted the business houses world-wide. In order to survive this worst recession, companies are doing every bit to save money or to reduce the costs. Software cost has been a major component of the costs that a company has to incur. So the onus further lies on the software companies to reduce the costs of their products.

Most software cost models are functions of five basic parameters: size, process, personnel, environment (available tools to automate the process), and required level of quality (Royce, 1998). Cost is likely to be more for a large sized product (measured in LOC or function points). Keeping all other parameters constant, a reduction in size can help to reduce the cost of a product. Walker Royce defines size reduction as “to reduce the number of human-generated source lines” or “to reduce the amount of custom developed code” (Royce, 1998). He suggests reuse, object oriented technology, higher order languages, and automatic code generators as some of the mature size reduction technologies. Capers Jones, also points out that a shift from custom development to reuse based development can help to improve the software cost structures (Jones, 2009).

Not only that reuse based software development reduces the development cost, it also shortens the development cycle and thus the time to market. It increases the productivity of programmers. Rather than spending time and effort on mundane tasks, they can focus on more challenging aspects of the application and hence improve its level of quality (Jacobson et al., 1997; Tracz88). In addition several other benefits of reuse have been reported: reduction of project planning overheads, improvements in support and maintenance, better use of resources, and better tackling of system complexity, (Almeida et al., 2007; Llorens et al., 2006). Mohageghi et al. review the industrial studies that link software reuse to quality, productivity, and economic benefits (Mohageghi et al., 2007).

The idea of software reuse is not new. In 1968, McIlroy suggested software reuse as a means for overcoming software crisis (McIlroy, 1968). Software crisis is characterized by two major phenomenon: Lack of ability to produce software within budget and time constraints, and lack of quality in produced software (Kim and Boldyreff, 1996). McIlroy
pointed towards the effective use of reusable software libraries to build large reliable software systems in a controlled and cost effective way.

1.1. Reuse Technologies

Software reuse can be implemented in 3 different forms – Compositional reuse, Product Line Engineering (PLE), and Generator based reuse (Mili et al., 2002). Compositional reuse refers to building an application by assembling already available reusable software components. Component Based Software Engineering (CBSE) paradigm adopts this form to produce software applications. PLE refers to creating a common set of core assets and then using them to create applications that satisfy the requirements of a specific domain (Pohl et al., 2005). Generator based development is applicable for more mature/narrow application domains in which the application developer specifies the variation through parameters and the generator generates the application according to these parameters. In the generator based reuse, domain knowledge is encoded into an application generator. For example, Lex and Yacc are the application generators in the UNIX environment.

To achieve an ideal level of reuse, the desirable trend is to move from compositional to generator based reuse.

1.2. Component Based Software Engineering

Component based software engineering is a systematic approach to develop software applications using already existing software components. The notion of ‘developing an application program by writing code’ has been replaced by ‘building a software system with assembling and integrating existing software components’. It involves use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind (Clemente et al., 2008). The components used in a component based software system may be in-house components or off-the-shelf components (which include open source components or commercial components also known as COTS -Commercial off the Shelf). Component developers develop software components keeping in mind their reuse value across product lines and organizations. These reusable components are reused as is or are adapted to meet the requirements of a different project in a context other than the one anticipated during their development.

This approach is different from the traditional way of software development. Here, the development process has two sides: Development of software components for reuse and
development of software with reusable components as the building blocks (Figure 1.1). Main steps in development for reuse are (Sametinger, 1997):

1. Perform domain analysis
2. Identify the components to be developed
3. Develop the components
4. Evaluate the components so that they can be added to the library
5. Package the components and add to the library.

Main steps in development with reuse (Sametinger, 1997) are as follows:

1. Retrieve components from library (in house or third party) according to some need of the application under development,
2. Evaluate the quality and appropriateness of the components.
3. Adapt a component, if it cannot be reused as-is.

---

Fig. 1.1: CBSE Processes (Hutchinson and Kotonya, 2006)

---

4. Assemble the application
5. Test the integrated assembly

It can be observed in the above discussion that component evaluation takes place at two stages: when components are added to the library of reusable components and when they are selected for use in an application. In the latter case, context of use is also important for evaluation.

1.2.1. Defining a Component

Both the researchers and practitioners in component based software engineering have not yet agreed upon the definition of a component. Different people perceive the concept of a component differently.

Hooper and Chester define a component simply as “anything which is reusable” (Hooper and Chester, 1991). Bertrand Meyer adds another dimension to it and separates a component user from a component developer (Meyer, 1999). He defines a component as “a software element that must be usable by developers who are not personally known to the component’s author to build a project that was not foreseen by the component’s author”. Johannes Sametinger stresses on technical attributes of a software component (Sametinger, 1999). According to him, software components are “Self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status”. Szyperski in his definition of a software component takes into account context of component deployment as well (Szyperski, 1997). He defines a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

Councill and Heinemann’s definition includes a reference to the component infrastructure (middleware) required for seamless integration of software components (Councill and Heinemann, 2001). They define a software component as “a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”.

In the context of object oriented paradigm, Valerio et al. define a component as “a homogeneous set of objects that collaborate to perform a feature or functionality and exposing a component interface that allows to integrate it in a system and make available to the external environment a set of services” (Valerio et al., 2001). However components
are not the same as classes or objects – the traditional object oriented artifacts. Classes are conceptual entities which form a part of the structure of a program. Once implemented as part of the program, they are not required to be accessible from outside. Components are the physical entities which are accessible and pluggable as per the requirements. Both can be assembled to build a new application, but the difference is that components are plugged, and objects are wired. Components generally provide complex functionality, where as objects provide limited functionality. Components offer explicit interfaces: required interfaces, and provided interfaces. But objects explicitly mention only their provided interfaces (set of public methods), their required interfaces are hidden in implementation.

1.2.2. Component Characteristics

Several attempts have been made to characterize the components in order to better understand and classify them which can further help in efficient component storage and retrieval and component selection as well.

Szyperski identifies 4-tiers at which there are different reasons to use components in different forms (Szyperski, 2003). In tier 1, organizations choose software components for economic reasons. Components at this level are the source components which include architectural, design, and source code artifacts. Tier 2 focuses on use of partial design and implementation information across multiple products of the same domain i.e. product lines. These components are known as build time components. In third tier, components are not consumed during development-time but during run-time. Such components are called deployable components. The fourth tier deals with the use of components to handle changes in the engineered solution in an open environment. Components are dynamically available as services at distributed remote locations which can be obtained on-demand, installed and integrated with the existing solution.

Bertrand Meyer characterizes components on the basis of four viewpoints: level of software process task, level of abstraction, level of execution, and level of accessibility (Meyer, 2003). At different levels of software life cycle, components exist in different forms. It may be a software requirements specification document in the analysis phase, a design pattern in the design phase, or an executable piece of code at implementation level. It may represent an abstraction of a function, or data with fine granularity as a class or coarse granularity as a cluster of classes or a complete system. A component is static if it is integrated into a system at compile/link time, and has to be recompiled after every modification. It is replaceable if it is static but its variants can be substituted dynamically.
A component in the dynamic category can be integrated into the system at the time of execution. No source code is available mostly for components in the commercial category. Level of accessibility criteria distinguishes components, with source code available to component users, from components whose source code is not available or is available on demand only. Components in the former category are available in the open market and are called Open Source Software Components. Open source components are available free of cost under different types of licenses such as GNU, PDS (Public Domain Software) etc. Components in the latter category are available in the commercial market and are acquired for a fee. Such components are known as Commercial off the Shelf (COTS).

Other criteria to classify components can be age (level of maturity), level of reuse, context (application domain), technology/infrastructure support, ability to plug and play with other components as well as with the underlying platform, role (active or passive- GUI v/s database component) (Yacoub and Mili, 1999).

Sametinger identifies component characteristics by means of different types of interfaces a component uses to communicate with the user, other components, or with the environment (Sametinger, 1997). They include: type of user interface (command line or graphical), data interface (textual, file, or data base I/O), program interface (functional composition, object oriented composition, or open platform composition), and component platform (hardware, OS, and programming environment).

Morisio and Torchiano characterize software components on the basis of source (origin, cost, and property), customization (required modification, possible modification, and interface), bundle (packaging – static or dynamic, partial or total delivery, size), and role (functionality, and architectural level- support or core) (Morisio and Torchiano, 2002).

The component characterization framework suggested by Sassi et al. groups characteristics into: general (cost, date of first release, and change frequency), structural (name and number of services), behavioral (pre/post-conditions and state-transition diagrams), architectural (component type and architectural style), quality of service (nonfunctional properties and possible modification), technical (conformance to standards), and usage (similar components and use cases) (Sassi et al., 2003).

Kienle et al. present taxonomy to characterize software components as well as component based systems (Kienle et al., 2007). They use the following criteria to characterize software components: origin, distribution form, customization mechanisms, interoperability mechanisms, and packaging. Origin of the component specifies the source
of the component i.e. in-house or off the shelf component. Distribution form is based on
the availability and modifiability of source code – Black box (no source code available, no
modification possible), white box (source code available, modifications possible), glass
box (source code visible but no modifications possible). Sametinger specifies another
distribution form i.e. Gray box in which limited source code is visible, and only that
portion is modifiable (Sametinger, 2003). Customization mechanisms are available at two
levels: non-programmable and programmable. Non-programmable customization allows
using command line switches, configuration files, or check boxes to customize a software
component. In programmable customization, application programming interface (API) or
scripting languages are used to modify or extend the behaviour of a component. Another
characteristic of a component is its ability to interoperate with other components in the
application. There may be no interoperability information available for a component or it
may have programmable interfaces to enable interoperability with other components.
Components may be packaged differently as standalone or non-standalone. Standalone
components can be directly executed without any prior customization or integration.
Whereas a non-standalone component has to be customized or integrated before it is
executed.

1.2.3. Issues and Challenges of using Components

CBSD is still not a very popular paradigm of development with the software developers.
There are several obstacles to successful adoption of the CBSD in software development
organizations. Kunda et al. have studied the human, social, and organizational factors
responsible for making the CBSD application difficult in organizations (Kunda et al.,
2000). Neumann elaborates on the risks of predictable compositions of software
components. There are several problems related to composability of components including
scalability, certification, quality assurance, incompatible policy matters, inadequate
requirements specifications, poor software engineering practices etc. (Neumann, 2006).
Voas has identified and analyzed five sources of headaches in dealing with library
components (Voas, 1998). Apart from technical risks, Hasselbring et al. talk about liability
risks of using third party components (Hasselbring et al., 2006). From judicial point of
view (in the European Union), the vendor providing a component based software solution
to his customers is to be held legally responsible for malfunctioning in any component of
the solution. The customer is not bound to localize the defect to a particular component of
the solution. Application vendor has to pay the claims. The vendor can, in turn, ask for
damages from the specific component supplier but for that he has to establish the fact that the supplier’s component is problematic.

Component based software development brings with it its own issues and challenges. Several technical as well non-technical issues need to be addressed in order to have successful implementation of this paradigm (Kalagiakos, 2003). Some of the issues are outlined below:

- Creating tools, techniques, and well defined processes to support essential component activities such as component specification, component certification, component search and retrieval, component selection, component composition, component integration, and component version management.
- Component repository management - A rich repository of reusable components is essential for successful implementation of component based development. Further the component users should be able to locate needed components easily and quickly. So there is need to design efficient algorithms for storing and retrieving components from a repository.
- Risk analysis and management – Components acquired from external sources carry the risks of unpredictable quality, architectural mismatch, and uncertainty of future support from component suppliers (Vitharana, 2003). In addition to this there are legal risks involved. So it is necessary to identify the risks, and manage them in advance so as to facilitate seamless integration of software components.
- Support for evolving third party components – Successful software requires modification from time to time to accommodate changes in domain as well as in technology. As software components evolve, problems arise due to inadequate support from the component vendor, delay in identification of modification requirements and their implementation, conflicts between needs and priorities of different component users. So there is need to manage component evolution otherwise it may result in higher maintenance burden and lead to other quality issues such as reliability.
- Establishing Trust in third party components – There is a need to define mechanisms to establish trust in third party software components. A component user has every

---

reason to not to believe the component developer/supplier regarding component quality attributes till sufficient documentary proof is not made available.

- Component quality assessment - Quality of existing components in general and of third party components in particular has been an issue of great concern. Bertrand Meyer suggests that foremost importance should be given to quality of software components especially acquired from third parties (Meyer, 1999). He stresses that quality of a component based application is equal to the quality of its worst component. Here issues that need to be explored include: component characterization, component documentation, availability of component related information, component testing, component certification, component quality models, and predictable assembly of components.

- Software reuse metrics and models – Software metrics in the reuse context may be divided into five categories-
  1. Metrics which measure the extent of reuse within a software application,
  2. Metrics which measure the consequences (economic benefits) of reuse in an application,
  3. Reuse library (repository) metrics
  4. Metrics which measure the ability to use a software component in a context other than that for which it was originally developed, also known as reusability metrics.
  5. Metrics which measure the quality of a reuse based application.

There is need to define metrics based on the formal specifications so that they can be theoretically as well as empirically validated.

The emergence of component integration technologies such as the Common Object Request Broker Architecture (CORBA), Sun’s JavaBeans Model, and Microsoft’s Component Object Model (COM) and Distributed COM (DCOM) has enabled the reuse of software components across development groups. Moreover, it allows components to be freely exchanged between and across application domains and development contexts—to such an extent that it is now feasible to recognize the existence of software component markets (Ravichandran, 2003)

1.2.4. Component Assessment and Quality

One of the most important activities in component based software development is the selection of suitable components. Component selection further consists of two sub-
activities: component search and retrieval, and component evaluation. As indicated above component evaluation should take place in both the processes of component based software development life cycle. Component evaluation is a critical activity for selection of suitable components. A component needs to be evaluated for its technical as well as business value. The proposals discussed in this section point towards the need to evaluate design of a component, and act as motivation for this research.

Mili et al. propose that for the purposes of component evaluation, it is important to know how easy it is to use the component and how it performs its function (Mili et al., 2002). Information, in the latter case, is inherent in the structure of the component. It may also be made explicitly available through metric values, which is at present a rare practice. Availability of such information depends on the type of reuse (black box or white box), type of component (analysis, design, or source code), and the type of component user (active or passive). Type of required information also depends upon the interest of the component user such as reliability v/s performance.

Sametinger stresses that good design is essential for the quality of a component (Sametinger, 1997). Almeida et al. are also of the opinion that a software component needs to have a modular design (Almeida et al., 2007). Szyperski introduces a maturity model for software components and calls it Component Maturity Model (Szyperski, 2003). At the initial level of maturity, the solutions should be at least maintainable i.e. have modular designs. At the second level of maturity, organization should be able to implement internal reuse and develop product line solutions. The maturity model gives two options for composing component based application at the third level of maturity – open composition in which the components are acquired from open market, and closed composition in which components are obtained from a closed pool of suppliers. In both the cases components need to be upgradable dynamically i.e. at run time. The last level of maturity restricts component solutions to be open and dynamically upgradable.

Bertrand Meyer classifies the quality improvement approaches in two orthogonal dimensions: Technical v/s Management, and A Priori v/s A Posteriori (Meyer, 2003). A priori approach ensures use of proper methods while developing the software component. In technical sense, it is related to use of formal methods. A posterior approach is applied once the software is ready. White box testing and static analysis are the technical methods useful in this approach.
Existing components are seldom available as ready-to-use building blocks (Ravichandran and Rothenberger, 2003; Yu, 2009). Instead, these components need to be adapted and/or modified to meet the specific requirements of the new product being developed. In addition to this, a component also has to be upgraded to meet new requirements from time to time. Therefore, software components need to be designed with maintainability as an important consideration, otherwise releasing new versions of the software component may become an expensive and complicated task for the component producers (Almeida et al., 2007). Hence, reusability and maintainability are two important properties of a software component.

It can be concluded that in order to evaluate a component, it is important to evaluate its internal design/implementation as reusability and maintainability of a component depends upon the ease with which it can be modified for use in unforeseen contexts, and also for future updates. With this background, the problem for this research is defined next.

1.3. Problem Definition

This research focuses on software components created using object oriented design methods. It is a widely accepted fact that software solutions developed using OO design methodology, are easy to use, test, maintain, refine, and extend (Wirfs-Brock, 1990). This could be attributed to object oriented designs which are natural, resilient to change and easily evolvable (Blaha and Rumbaugh, 2005). Object oriented technology provides concepts such as abstraction, inheritance, polymorphism, encapsulation, coupling, and cohesion to design reusable software components. However Rumbaugh et al. argue that object orientation is not a magic formula to ensure reusability (Rumbaugh et al., 2002). Reusability of a software component in multiple applications has to be planned before hand and efforts need to be put in to create a general design that can be adapted or extended easily for unforeseen situations (Cohen, 1998). Software metrics help to measure structural properties of an artifact. This research employs object oriented metrics (Chidamber and Kemerer, 1994) to understand the extent to which the object oriented concepts are followed while designing an object oriented reusable software component. It analyses changes in metric values as the software component evolves over a period of time. The software component selected for analysis is a chart builder known as

---

JFreeChart, developed using JAVA language. It is an open source reusable software component available in the well known online repository [www.sourceforge.net](http://www.sourceforge.net). It has an active community and a large user base. A total of 43 initial versions of the software component released in the time period from 2000 to 2008 have been studied.

### 1.4. Objectives

As a reusable software component, or for that matter any software product, evolves quality of its design degrades if it is not managed from time to time. This research aims to evaluate the design of a reusable component by analyzing the trends in its object oriented metric values over a period of time. Main objectives of this research are:

1. To investigate the design of an evolving software component (JFreeChart) with the help of metric trends.
2. To study the extent to which OO concepts are followed by comparing metric values with the thresholds suggested by researchers/practitioners as well as identified by using statistical methods.
3. To test the hypothesis that during the maintenance activity:
   - Added classes have better properties (size, coupling, cohesion etc.) than removed classes
   - Unchanged classes have better properties than changed classes.
   - Maintenance activity takes place at shallow levels of hierarchy.
   - Maintenance activity results in extensive modifications in the organization of the inheritance hierarchy.

### 1.5. Significance of study

It is commonly believed that component users need not bother about the internal structure of a reusable software component. It is also recommended to not to provide the internal details to component users as they may create unnecessary context dependencies on the internal design of the software component which may make component replacement difficult in the future. The team responsible for creating reusable components controls the development process and resources required to develop the components. The development team or component broker may paint a rosy picture of an otherwise low quality component (Mari and Eila, 2003; Li et al., 2008). This may lead to exploitation in the component market which is already facing the problem of lack of trust in third party components.
Suppose a software component is designed with no sub-classing, and high levels of coupling between classes. This will result in a large sized component (because of less use of inheritance), if size is measured in LOC. Moreover if size is taken as a factor for price determination, the component may have a high price tag as well. At the time of maintenance, the change effort will increase due to (unnecessary) coupling so maintenance cost will also increase. Metrics can be used to check as up to which level an object oriented software component uses the concepts of object oriented technology that promises reusable and maintainable designs.

Metrics based analysis also helps to know how others are developing reusable software components.¹ Such an attempt can guide developers to know as to what goes inside a reusable component which can further motivate them to develop easily reusable and maintainable designs.

1.6. Overview of the Thesis

Rest of the thesis is organized as follows:

Chapter 2: Literature Survey
Chapter 3: Experimental Setup
Chapter 4: Results and Analysis
Chapter 5: Conclusions and Future Directions

Second chapter presents the literature survey. Metrics in component based software engineering are studied in two dimensions: system level metrics, and component level metrics. Object oriented metrics are also classified into: System level, Package level, and Class level. This chapter gives a detailed analysis of the validation studies for the object oriented metrics. It also mentions the studies which applied object oriented metrics to evaluate reusable components.

Third Chapter explains the metrics that are taken up for further analysis in this research. Several metrics have been chosen to measure different concepts of the object oriented paradigm such as size, inheritance, abstraction, coupling, cohesion, information hiding, polymorphism. Metrics are considered at three levels of granularity: system, package, and class level. Metrics from MOOD (Metrics for Object Oriented Design) metric set (Abreu et

al., 1996) are selected for system level analysis. Robert Martin’s package metrics (Martin, 2003) are used for next level analysis. At class level, metrics have been picked up from different sources.

The chapter also discusses the tools used to collect the metrics. Borland’s Together 2008 SP 2 is used to collect system level metrics. For package level and class level metrics, SDMetrics tool is used. SDMetrics can collect metric information at design level only. So the source code of the software component was reverse engineered using BOUML, a reverse engineering tool, to generate the design level information.

Some of the metrics such as Number of Hidden Operations (NumHidOps), Number of Overridden Operations (NumOvrOps), and Coupling Between Classes (CBC) which SDMetrics does not collect, have been defined in this research by extending the existing metric definition file of the tool. This research points out some anomalies in the existing design level class cohesion metrics (CAMC, NHD, and SNHD) and proposes a modified version of the NHD metric. It is called NHD Modified (NHDM). An automated collection of these metrics is realized by writing XML scripts and by developing programs in the C language. SPSS statistics tool is used for analysis of metrics data.

Fourth Chapter presents the statistical analysis of the metrics data collected from an open source reusable software component available at www.sourceforge.net. It begins with a description of the software component used for analysis. The chapter presents trends in metric values as the software component evolves. Metrics are used to analyze different object orientation concepts such as complexity, abstraction, inheritance, information hiding, polymorphism, coupling, and cohesion. This chapter also contains a comparison of the metrics values obtained for this component with metric values suggested by researchers and practitioners and also calculated by using statistical methods.

Component releases from JF 0.9.20 to JF 1.0.2 were further analyzed to understand the large amount of change in size. Classes were divided into four categories – added Classes, removed classes, unchanged classes, and changed classes. It is observed that unchanged classes have better properties than changed classes, and added classes have better properties than removed classes. Inheritance metrics are analyzed to investigate the changes in the hierarchical arrangement of classes during evolutions of the software component.

Fifth Chapter presents the conclusion and gives future directions.