CHAPTER 2
LITERATURE SURVEY

2.1 INTRODUCTION
This chapter provides a detailed discussions of techniques used for migration of OOOSS TO CBS. We segregated these techniques and highlighted major development related to legacy systems, software re-engineering, reverse engineering, software reuse, object oriented systems and component based systems. The proposed research is based on broad and detailed literature survey.

2.2 SOFTWARE EVOLUTION AND RE-ENGINEERING
Bigger organizations allocate majority of their software budget for evolving the pre-existing software systems instead of developing them from scratch. Requirement change is the main driver for software evolution and evolution persists throughout the system life span. Software evolution is the process by which software systems reform architecture and realize properties from existing systems. It is the continuous change to make the software systems capable to adapt the newest technologies and includes various activities such as maintenance, refactoring, updating in architecture and re-engineering [13]. Re-engineering is discussed relatively more actively in research for system evolution. It involves restructuring the system, organizing, redocumenting and migrating the system to a new programming paradigm by modifying its structure and value of data. For re-engineering, one should have knowledge of various phases of re-engineering, related tools and techniques [14]. Involved cost and different re-engineering experiences have to be examined to design a completely new model.

It is generally a difficult task to change the programming language of the old systems and migration is also not a simple task. Though, from a business perspective, software re-engineering may be the only feasible way to ensure that legacy systems can still be kept in service. It may not be a wise decision economically to adopt any other approach to system evolution. To understand this justification, an overview of legacy systems issues is necessary. Many industries use these systems, even with high maintenance costs, because they are valuable to them and consisted of important business rules that are hard to recover in an eventual re-engineering effort [18]. Market competitions, changing business requirements and
revolution in technologies lead to persistence modification of legacy systems resulting in unstructured code. The feasible solution to deal with these issues is modernization or renovation of legacy [19]. It helps software industries to respond to the changing needs of the software communities.

To experience the perception of industry practitioners on legacy systems modernization, we interacted with some of the software developers those working on legacy systems in telecom companies. They outlined following points while discussing the need for modernization of these systems:

- It is difficult to train new users on legacy systems.
- Lack of expertise on legacy technologies
- Databases cannot be easily maintained on legacy systems (RDBMS was introduced later)
- All the legacy languages are not in use like COBOL, FORTRAN and BASIC.
- Many modern operating systems don’t support legacy systems.
- Important business rules may be embedded in the software but may not be documented elsewhere.
- There is rarely a complete specification of the legacy system. The original specification may have been lost.

The lifetime of a legacy system can be increased by renovating or re-engineering it. Several attempts for definition of re-engineering are made, typically by Chikofsy et al. [15]: “It is an examination of the design and implementation of already developed software systems with a purpose to redesign and reshape the systems into hopefully better or suitable software using advanced tools and techniques”, Jacobson et al. [17]: Re-engineering = Reverse engineering + Delta + Forward Engineering. These are the most accepted definitions in literature. Reverse Engineering is the analysis of subject systems to identify its elements or modules and their interrelationalship to produce a representation of the systems in another form or at a higher level of abstraction. Forward engineering on the other side is the conventional process of logical implementation of the design to the physical implementation of a system [16]. Figure 2.1 depicts the idea:
2.3 CHALLENGES IN OBJECT ORIENTED SOFTWARE DEVELOPMENT

Object oriented software development (OOSD) gained the maximum popularity between 1980 and 2000 [20]. Most of the software languages and the development cycles are even designed relative to OOSD. Various process models, prototype models are defined to control the development process and to utilize the software system capabilities in an effective way. The core component of OOSD is the object model [21]. In this model, the classes are defined to represent the entities or the modules. Each class is defined with relative variables and function specification. But while implementing such design model for large and complex software systems development, a lot of issues and challenges are faced in development methodologies. Some of the most discussed challenges are:

2.3.1 Software Reuse

The reusability of existing artifacts or the available code modules or the libraries is the primary requirement in software development to improve the performance and reliability of software development. In object oriented programming, the reusability is accomplished through inheritance. This kind of reusability is complicated than library level reuse. The dependency observation and the composition refinement cannot managed by such kind of reusability. These types of reusability become the code level connectivity which requires complete efforts for software testing. Each flow and the access behaviour need to be rechecked which is quite time consuming and complicated as the number of involved classes increases. The decision becomes more critical when software maintenance is performed [21]. The object oriented design process cannot answer the reusability in an effective way.
2.3.2 Prototyping
Prototyping in software development helps to improve the sharing and robustness of the software systems. The representation of the software system in structural form is required to represent the working model relative to the software design. The framework is required to define the software system requirement, design time constraint, extensible adoption of the system. In object oriented languages, the libraries can be defined in code or at object level. But as the complex software system is composed, the framework formation is not acceptable. The configuration of the integrated elements and the interactive map is difficult to achieve. It shows that the prototype is the inclusive, feature of object oriented systems, but when the configuration level or the dynamic updating to the existing system is required, the prototype cannot be used effectively and it requires the changes from the initial level. The design features, code features and the framework constraints are modified from the earlier level. As the changes performed, the required test process and validation also need to perform again [21].

2.3.3 Concurrent Process
Concurrency in object oriented projects is the critical design time and long standing issue. These kind of programming requires the thread based handling of concurrent processes. The multi-thread model is defined as the active object model with specification of concurrent control and communication flow. The concurrent processes share the common memory or resources by setting up the relative status locks. The operational synchronization is required to characterize the behaviour of associated processes. The iteration between the processes is defined with parallel incoming and outgoing communication. The object oriented programming (OOP) is not able to formulate such kind of conceptual and dynamic interaction. The status, locking on different objects, resources and memory cannot be represented through design diagrams supported by object oriented languages [17].

2.4 COMPONENT BASED SOFTWARE DEVELOPMENT (CBSD)
Research shows that the object oriented concept itself is not an effective solution with rapidly changing business needs and requirements of ongoing applications. Many information-based legacy systems contain similar or even identical things, which are developed from scratch again and again. With the huge growth of software size and complexity, the conventional approach of developing software systems from scratch is more expensive and can take a longer time to complete. Critical applications with strict time limits may loose the market due to the delay in
the development process. The quality control and quality assurance of the produced software system is also not feasible that discouraging use of the new technologies. In order to satisfy the quality requirements of modern large scale software systems, new paradigm has been introduced that provides the development of flexible, reliable, evolvable and reusable software system elements. This has led to the evolution of a new approach, called as component -based development (CBD) that is meant to be the independent and uses the concept of reusability in the application development. CBD supports three concepts 1) software systems development using existing software components, 2) development of components as independent and reusable entities and 3) system expansion is done by customizing and replacing the components.

Software development may have not matured into an engineering discipline, if it has not exercised techniques to use reusable assets for developing software products in regular manner. CBS acquires adaptability by isolating the stable parts of the systems (i.e. components) from the specification of their composition. Component based development has shown considerable success in various domains and accepted in software organizations as a new effective development paradigm in recent years [22]. CBD technology supports the mechanism of increasing the reliability of the entire system and reducing its development cost. Fast and in time delivery, high quality solutions, efficient reusability, flexibility, higher productivity and easy scalability are main advantages of the CBD paradigm [23]. It follows “the ‘buy, don’t build’ philosophy” [24] and shifted the attention from programming to composing software systems the same way that early subroutines excused the programmer from thinking about details [25]. The fundamental objective of the CBD is to develop composite software architecture by reusing smaller, manageable, software elements [26]. It uses component as a fundamental building unit to depict a modular unit of systems [27]. Higher productivity is achieved by using standard components and from the very beginning CBSD encountered the issue of acquiring a concrete and common definition of a component.

2.4.1 Component
The component based architecture or software system is composed with defining the individual interconnecting component. Each of the component is defined by relative properties, methods and events. To define a component, there are different answers with different persons, many people thought a component is a software unit with contractually specified interfaces and its interdependent context. A component can be independently deployed and used by other users.
Szyperski [3] gave a most commonly used definition of component as “it is unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed autonomously and is subject to composition by third party. In spite of its general use this definition was accepted by those, who do not support contractually specified interfaces.

Heineman et al. [26] provided a more general definition of component as “software component is software element that conforms to the component model and can be deployed and composed independently without adapting composition standard”. Currently, the use of components is considered to reduce the development cost because component technology produces more functionality with the limited resource. When system is developed using components, new issues such as variant explosion, dynamic configuration and scalability must be dealt with. These mentioned issues are addressed by CBSD discipline. It facilitates models, methods and guidelines for the software communities of the CBD. Components are viewed as families of routines that are constructed on the basis of well-defined principles so that these families fit together as building blocks.

Figure 2.2: Component based waterfall software development lifecycle [27]

In the last few years, the CBD approach has become very popular. It deals with the entire lifecycle of component-based products that has been focused on technologies related to the design and implementation of software components and systems built from software
components. A CBD approach requires certain changes in the development and life cycle processes. A component-based approach cannot be fully utilized if the development processes and even the development organizations are not adopted according to basic principles of CBSD. The life cycle of CBSD is similar to the standard waterfall development model with increments and iteration depicted in figure 2.2 depicts, when incorporated into the conventional waterfall process and presents comprehensive details of various phases of the CBD.

2.5 CHALLENGES OF CBSD

2.5.1 Dynamic Updating of Components
The component based software system (CBSS) requires the continuous updating in terms of new component inclusion, removal of component etc. The analysis of these updated systems, updated components is required in terms of interfacing and functional behaviour of the system. The capability of a software system can be modified at run time by applying the architectural or the implementation updating. During the updation, the dependencies are constructed and the operational interconnections are re-validated. This kind of updating is performed to optimize the performance of a particular component or software module. The error rectification can also be achieved through component updation [28].

2.5.2 Non-Technical Issues
The CBSS suffers from various non-technical issues including the social, educational, legal, economical and legal. The handover of the software product to other developers for using in their software as an integrated part is one of the fear. To design the software system, the skilled developers are required who can understand the scope of the programmers. The licensing, legal liabilities are required to setup before distributing and using these components in a software system [29].

2.5.3 Component Certification
Before using the components, it is required to verify the trustworthiness of these components. If the component is taken from open environment or from third party developer, more care is required before utilizing the component as part of the software system. The domain specific certification, check can be applied along with various test to validate the component reliability. If the component is developed by the own team, even then the domain specific validation check is required to perform before usage [30].
2.5.4 Long Term Management

The component based software system and sub-system can be developed by setting up a long term plan and development model. Various questions, problem and the associated solutions are highlighted by considering all the possible complexities of the system. The components are defined under the condition of administrative, organization and domain specific validations. The component updating is performed while analysing the legal issues and the integration issues in composite form. The risk and error level observations are considered at the earliest stage of component development so that while utilizing it as the integral part no such issues will occur in the software system [31].

2.6 NEED FOR MIGRATION OF OOS TO CBS

The need of re-engineering legacy system is basically driven by three factors [32]:

- Expansion of the system’s functionality;
- Improved maintainability of the system using latest tools and techniques; and
- Reduction of operational costs by increasing reuse potential of the system.

The demand for re-engineering has largely been increased as legacy systems become outdated concerning their architecture, the platforms on which they run, and their maintainability. Software re-engineering is an important activity for recovering and reusing the useful legacy systems assets because it helps to extract critical operations and reusable components of the legacy systems [32]. It establishes a maintainable base for future software evolution and reduces high software maintenance cost. Re-engineering of OO legacy systems enhances the overall quality of software systems by: 1) improving the usability of these systems 2) making them reusable 3) preparing legacy systems for future change and 4) conforming and making adaptable to new application areas.

The system is decomposed into a set of components to achieve the desired level of adaptability. “Components are binary units of independent production, acquisition, and deployment those interact to form a functioning system” [3]. CBD aims at constructing and designing a system using a predefined set of software components explicitly created for reuse [33]. Source code is reused in the form of objects in OOSD and developers reuse objects through different mechanism such as inheritance and polymorphism. This concept is similar in the CBD but instead of objects whole software component is reused. Different rationales are discussed by
researcher in literature, why software systems made up of components are better than OOS while dealing with changing business and technological needs.

- Component based paradigm facilitates construction and packaging of robust, scalable and adaptable components helping software developers to build software systems quickly and efficiently. This promotes software reuse and in turn increases productivity [34].
- Components are developed independently and loaded dynamically as compare to concept of classes in OOSD where classes are linked together. Ping et. al [4] also discussed that component is having higher independence, integrity and self-description as compared to object.
- Generally frameworks developed for CBD are black box in nature i.e., a framework can be reused without modifying their source code and can be scaled through composition as compare to object oriented framework.
- Reuse of components reduces the development and maintenance cost of new developed systems in case of CBD, while object oriented programs cannot be reused efficiently even if they contain reusable functions [35]. Sinha et al. [36] provided an empirical study where IT professional’s perception about the ease of reuse in business system is considered and professional perceived components as much easier to reuse than objects.
- Object Oriented classes are mutually dependent that creates difficulty in reusing of parts of existing OO programs which are composed of classes, so it becomes a need to transform a part of an OO program into independent components [36].
- Frameworks developed for CBD improve the flexibility and prepare legacy systems for future change [37].
- Technologies in distributed systems are more and more based on the use of component based paradigm [38]. Shivani et al. [39] discussed that object technology had not shown substantial development in distributed system, while component technologies is assumed to be more suitable for development of distributed system due to its granularity and reusability.
- Components are being considered more as coarse –grained as compared to objects and facilitate a high level of abstraction [40].
• Complexity and monolithic size of OO legacy systems leads to large and complicated collection of object and classes, which are hard to handle and understand. These systems can be improved by some advanced means of structuring and describing them.

• One major advantage of a component is its plug and play feature that enables the easy composition, selection and adaptation of components rather implementing software systems from scratch [41]. Dr. Pami Bahsoon, School of Computer Science, University of Birmingham endorsed the concept of CBD with the fact that it originated from the failure of object oriented development (OOD) to facilitate the effective reuse because single object classes are too detailed and specific, while components are abstract and can work as standalone service providers.

• OOS are complex because they contain large collection of classes with different dependencies, while components provides modelling elements that suites to high level abstraction and organized structure of complex software [3].

The goal and motivation of re-engineering OO legacy systems into component technology is to unwarp the software system into subsystems, which can be tested, delivered and marketed independently [42]. The maintenance of legacy systems is a daunting task for software industries. Almost 80% of budget of software development is used for maintaining the systems after their implementation. Insigma Technologies Co. provided various IT services in legacy engineering and re-engineered large number of financial legacy systems. One of its clients mentioned that “re-engineering their project compared to the cost of system replacement saved millions of dollars”. Therefore, the lifetime of a legacy system can be increased by re-engineering and it is the part of a software modernization process. But before a legacy system can be reengineered, the system must be reverse engineered. Reverse engineering is the prerequisite for legacy system’s renovation [14].

2.7 REVERSE ENGINEERING TECHNIQUES FOR LEGACY OOS
Research in reverse engineering of object oriented software has surfaced around 1999 after leading European companies, namely Daimler-Benz and Nokia started a research project named FAMOOS [42] to investigate tools and techniques for dealing with object oriented legacy systems.
Table 2.1 Tools and Techniques for Reverse Engineering

<table>
<thead>
<tr>
<th>S. No</th>
<th>Year</th>
<th>Author</th>
<th>Methodology</th>
<th>Tool</th>
<th>Legacy language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1997</td>
<td>Huang et. al</td>
<td>Reverse Engineering</td>
<td>JBOORET</td>
<td>C++</td>
</tr>
<tr>
<td>2</td>
<td>1999</td>
<td>Michele Lanza</td>
<td>Reverse Engineering is combining graph and metrics</td>
<td>Code Crawler</td>
<td>C++</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>Tarja Systa</td>
<td>Static and Dynamic Reverse Engineering</td>
<td>SCED Prototype</td>
<td>Java</td>
</tr>
<tr>
<td>4</td>
<td>2001</td>
<td>Surendranath</td>
<td>UML Notation</td>
<td>Rational Rose</td>
<td>Java</td>
</tr>
<tr>
<td>5</td>
<td>2002</td>
<td>Rudolf et. al</td>
<td>Reverse Engineering Framework</td>
<td>Columbus</td>
<td>C++</td>
</tr>
<tr>
<td>6</td>
<td>2003</td>
<td>Daniele Talerico</td>
<td>Reverse Engineering using graphical grouping</td>
<td>Code Crawler</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>7</td>
<td>2003</td>
<td>Nidal B. Eshah</td>
<td>Reverse Engineering by combining metrics</td>
<td>VMCPP</td>
<td>C++</td>
</tr>
<tr>
<td>8</td>
<td>2003</td>
<td>Michele Lanza</td>
<td>Polymetric View</td>
<td>Code Crawler</td>
<td>C++</td>
</tr>
<tr>
<td>9</td>
<td>2005</td>
<td>Polo Tonella</td>
<td>Object flow graph</td>
<td>Algorithm</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>10</td>
<td>2005</td>
<td>Wei et. al</td>
<td>Design pattern</td>
<td>DPVK</td>
<td>C++, Java</td>
</tr>
<tr>
<td>11</td>
<td>2006</td>
<td>Maximo et. al</td>
<td>Metamodel and conversion algorithm</td>
<td>-</td>
<td>C++</td>
</tr>
<tr>
<td>12</td>
<td>2006</td>
<td>Chiara et. al</td>
<td>UML Models</td>
<td>-</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>13</td>
<td>2007</td>
<td>Xinyi et. al</td>
<td>Hybrid Model</td>
<td>Swagkit</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>14</td>
<td>2008</td>
<td>Liliana Favre</td>
<td>MDA based Framework</td>
<td>-</td>
<td>Java</td>
</tr>
<tr>
<td>15</td>
<td>2009</td>
<td>Dan Cosma</td>
<td>Distributable features based approach</td>
<td>M SiDe</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>16</td>
<td>2010</td>
<td>Holger et. al</td>
<td>Rigi Environment</td>
<td>Rigiedit</td>
<td>C, C++ and COBOL</td>
</tr>
<tr>
<td>17</td>
<td>2011</td>
<td>Pereira et.al</td>
<td>MDA based Approach</td>
<td>Algorithm</td>
<td>Java</td>
</tr>
<tr>
<td>18</td>
<td>2012</td>
<td>Salman et al.</td>
<td>UML (Extraction of State Transition Diagram)</td>
<td>-</td>
<td>C++</td>
</tr>
<tr>
<td>19</td>
<td>2012</td>
<td>Thakore et. al</td>
<td>Software Quality Metrics</td>
<td>SRET</td>
<td>Java</td>
</tr>
<tr>
<td>20</td>
<td>2013</td>
<td>Mrinal et al.</td>
<td>UML (Sequence Diagram)</td>
<td>GNU pic2plot</td>
<td>Java</td>
</tr>
<tr>
<td>21</td>
<td>2013</td>
<td>Abhijeet et al.</td>
<td>UML Diagram</td>
<td>Framework</td>
<td>Java</td>
</tr>
<tr>
<td>22</td>
<td>2014</td>
<td>Hugo et al.</td>
<td>Model Driven Reverse Engineering</td>
<td>Framework</td>
<td>Java</td>
</tr>
<tr>
<td>23</td>
<td>2015</td>
<td>Favre et al.</td>
<td>Architecture Driven Modernization (ADM)</td>
<td>Framework</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>24</td>
<td>2016</td>
<td>Alshanqiti et al.</td>
<td>Visual Contract Model</td>
<td>VCE Tool</td>
<td>Java</td>
</tr>
</tbody>
</table>
After this numerous researcher examined the potential advantages of reverse engineering the OOS and developed distinct methods and tools for the same summarized in table 2.1. Hong et al. [85] discussed the design and implementation of Jade Bird Object Oriented Reverse Engineering Tool (JBOORET). It is developed in Microsoft Visual C++ for reverse engineering C++ program. It extracts design information from C++ code through pairing and store information in the database. Michele Lanza [44] proposed an approach to reverse engineer object oriented systems by combining graphs and metrics. Lanza implemented a tool called as CodeCrawler, which graphically display source code and provide an interactive environment to the user. He combined software visualization and software metrics to reverse engineer OOSS. Paolo Tonella [45] discussed that the most reliable and accurate description of the actual behaviour of a software system lies in its source code, but all the answer cannot be answered by the static behaviour of the system, so we need to focus on the dynamic behaviour of the system also. Tarja [46] provided evidence of the importance of SCED, a prototype tool used for modelling the dynamic behaviour of object oriented systems because both static and dynamic information need to be extracted to understand the system. During the reverse engineering of an existing software, a parser and debugger are used for extracting the static and dynamic behaviour. This parsed information can be seen as a graph using Rigi, a reverse engineering environment.

The objective of reverse engineer is to cope with the complexity of the software system by constructing various models. A system can be visualized using multiple views. Surendranath [47] discussed the use of UML Model to produce the multiple views of a system. The practical tools play an important role in reverse engineering process. Petri et al [48] discussed the importance of case tools for reverse engineering and presented the state of art for OOS. Author discussed that since UML is adopted by industries as a standard for representing the design information of OOSD, its tools provide reliable reverse engineering functionality and the research prototypes facilitate the advanced analysis through algorithms.

Reverse engineering frameworks are also discussed in the literature. Rudolf et al. [49] had made an effort in this direction. They presented a framework named as Columbus. This tool is capable to analyse the large C++ projects and schema. Reverse engineering of OOS using grouping is also adopted by researchers. Grouping adds much benefit to the process of reverse engineering. Some of the benefits discussed by Daniele Talerico [50], in his master thesis are:
- It helps in understanding of the system and its design recovery.
- Addition of higher abstraction levels of the system.
- Helps to reduce the complexity by the scalability of grouping.

Nidal Bashir Eshah [51] extended a software product tool named VOO++ (Visualizing object oriented C++ files) which was developed by Mersa in 2000 by incorporating the software metrics into the VOO++ and the new developed tool was named as VMCPP (Visualizing and measuring C++ files). The tool was developed to visualize the software and its main objective was extraction of the software components and software attributes, and presenting them graphically to software developers. Making design patterns for reverse engineering object oriented systems also attracted attention in the past. Wei Wang et. al [52] introduced a tool DPVK (Design Pattern Verification tool kit) to detect patterns in Eiffel systems. DPVK examined both the static and dynamic behaviour of a system that is written in Eiffel (object oriented language). To understand structure and behaviour of a legacy system whose documentation is not available, Máximo et. al [53] proposed a metamodel, for reverse engineering of C++ code into sequence diagram. In metamodel the characteristics of the system such as entities, attributes and relationship are included. UML is widely used high level object oriented language for specification. Chiara et. al [54] proposed an approach for reverse engineering using UML as a modelling language of the Railway Logistic System.

Xinyi Dong et. al [55] proposed a hybrid model for reverse engineering of object oriented software. The developed model makes use of classes and entities at different level of granularity. The main objective of hybrid models is to avail software engineers a divide and conquer strategy to deal with complex, large systems. These models help maintainers by providing a higher level of abstraction since at higher level maintainers can focus on external properties and relationship without concerning about their internal implementation. Model driven architecture approach i.e MDA is also emerging in the last few years. Liliana Favre [56] discussed the MDA concept and focussed on reverse engineering of both platform dependent and platform independent model of OOS. The objective of MDA is to increase level of abstraction by incorporating the use of models. Models are the primary artifacts in the software development. MDA is a model driven technical framework to enhance the portability, interoperability and reusability using separation of concerns. Pereira et al. [57] discussed an MDA (Model Driven Architecture) approach to recover the use case diagrams from Java code.
Nowadays distributed software system has become important for the functioning and growth of civilization. Dan C. Cosma [58] analysed the distributed object oriented systems through the process of reverse engineering in his doctorate work. The proposed approach presents a set of integrated methodology for identifying the distributable features and classes. The most important part of any system is having the source code available for documentation [59]. But if the source code is not available due to some unavoidable reasons, then it poses several problems in understanding the system on software engineers.

Rigi environment also played an important role in analysing and documenting the large software system in late nineties. Rigi environment is a research tool that helps in reverse engineering of large software systems. Holdger et al. [60] in their article described Rigi’s main components and its functionality, and assess its effectiveness on reverse engineering. Rigi is used to model C, C++ and COBOL code. Some of the researcher also emphasizes the role of use case diagrams while reverse engineering the software systems. Salman Abdul et al. [11] presented a reverse engineering process for a C++ program to extract the state machine diagram. State diagrams are used to model the dynamic behaviour of the system by the software community. So, if the state transition diagrams are extracted from the legacy code, the dynamic behaviour of the software can be understood from the source code itself. In this work two strategies are presented to extract the state machine from the C++ code.

To assess the software qualitatively and quantitatively, software quality metrics play, an important role and quality metrics are not only measure the software statically but it also helps in assessing the behaviour, size and complexity of the object oriented software. Thakore et al. [61] proposed an automated and comprehensive tool which measures product metrics such coupling, cohesion etc. for object oriented programming. Mrinal Kranti et al. [62] emphasized on the dynamic view of the program to fully understand the system. They outlined a reverse engineering approach by extracting the sequence diagrams from Java code to analyse the dynamic nature of the system. The developed approach works only for simple object oriented programs.

Abhijeet et al. [63] focused on structural characteristics of a program during reverse engineering process. So, they extracted the class diagrams and activity diagrams from Java source code. With the emergence of Model Driven Engineering (MDE) in the last decade, its principle and techniques have been used as an effective reverse engineering solution.
Application of MDE in reverse engineering is called Model Driven Reverse Engineering (MDRE) and it is used to reverse engineer the legacy system in order to generate the model based views to facilitate the legacy system understanding and manipulation. Hugo et al. [64] discussed different MDRE practices and based on their experience in legacy system modernization, they introduced MoDisco: a generic, extensible and global MDRE approach and a framework implemented in eclipse plug-in. The objective of MoDisco is to provide the development of model based and model driven solution of legacy systems with the purpose of understanding. Favre et al. [65] discussed a technical framework architecture driven modernization (ADM) for information integration and tool interoperability and which is emerged as new object management group (OMG). Alshanqiti et al. [66] developed a visual contract extractor (VCE) tool that support reverse engineering activity of visual operations contracts from Java code.

2.8 SOFTWARE REUSABILITY
Reusability is one of the core software development process. As the technological improvement in the programming languages is done, the scope and awareness to the reusability is increased. The CBSD is one such development technology in which software systems are composed in the form of integrated component. Each of the components can be defined individually with relative properties, functions and events. These components can be used by other software systems to make them scalable. The reusability avoids developing or designing the particular module from scratch. Reusability is also a scientific process that also handles system level, process level and component level challenges. These challenges include maintaining the context of a component, in which it is defined. In most of the cases, the basic context or reference of existing component cannot be changed in raw form. Another challenge is to map the functionality of existing component with new functionality. A middle layer can be included to map the input output of existing and the new system [67].

Reusability provides an easy form of software development by utilizing existing software projects or components. A lot of the earlier models are available to utilize the existing projects. Hyun et al. [68] has defined an evaluation based framework for component utilization in a software system. Author identified the key attributes and focussed on theoretical soundness and acceptability of software components for a new software project. The discussed model covered various software aspects including modularity, portability and compatibility etc.
Different metrics are defined by the author to evaluate the reusability integration. Sampson et al. [69] has introduced a framework to learn the reusable objects with integrated activities. Author defined the method to process the elements of a component and its learning to the real software environment.

Etzkorn et al. [70] explored the characterization of reusability and its scope to improve the software system quality, productivity and cost factor. The structured software systems defined by object oriented language are more reusable than any unstructured or the procedural software system. Various quality metrics and measures are defined to quantify the reusability under various inclusive aspects including usability, maintainability and portability. The component level estimation was discussed in literature to characterize the quality factors and to enhance the product level quality features of a software system. The reusability can be applied at different level, including design level, structure level, code level and component level as shown in figure 2.3. The design level reusability [71] is applied to design a similar software system in the same application domain. The requirement set, feasibility study and the conceptual design constraints can be reused [72] in a new software system. This kind of reusability is applied from core level which reduces the efforts and cost of system study and analysis.

Figure 2.3: Different levels of reusability
The structure level reusability is another term which provides the artifact based reusability and where, software artifacts can be designed using UML diagrams. For already designed software system, these diagrams are either available or can be extracted using reverse engineering [71 - 75]. These diagrams are able to describe an individual module, component, process or the system. The updation to existing artifacts is easy with cost and time enhancement. Code level reusability is one more critical form which can be described by software maintenance, software updation, software reconfiguration or software reconstruction.

The components and object driven systems are having a significance to provide full or partial reusability [76-77]. The process includes the reusable component selection, complexity estimation and its integration in view of a system. Various researchers had given attention for quantifying the reusability at different levels. Reghu et al. [78] discussed a structure based study to explore each element of reusability. The complexity derivation, subject level description and design time integration are explored by the researchers using real time examples. Judith Barnard [79] proposed a metric to evaluate the potential reusability factor for O OSS. The metric measured the component connectivity strength, methods correctness and degree module interfacing effectively.

2.8.1 Reusability Assessment

2.8.1.1 Use Case Pattern

Use case patterns (UCPs) are generative pattern which is specific to the application domain and expressed through user viewpoint. The UCPs can present structure and design of a software system in a better and effective way. The pattern categories and the application domain specific taxonomy are defined for different application domains. UCPs are able to explore the features of each component, function, behaviour and the capability available in the software project for different application domain references and elements. The patterns are included in the catalogue with relationship characterization to define the process pattern. These UCPs also having the adaptability to reuse the components and the software system elements in a cost effective way. The continuous requirement, usage and the quality based aspects are defined with each pattern to improve the maintainability of component usage [80].

2.8.1.2 Architectural Constraint Map
The architectural constraints are defined in terms of parameter and signature specification. While accessing the component in a new software system, the constraint specific signature map is performed. The entity driven customized entities can be composed using these components and the architectural specifications. The constraint specific component map can be applied to design the new improved software architecture. The constraint level check is performed by Chouki et. al [81] to map each of the configuration vector.

2.8.1.3 Qualitative Factors
Charu et al. [82] estimated the reusability based on different qualitative factors including modularity, maintainability, flexibility, adaptability and interface complexity. Author first identified relation qualitative features and later on processed them under soft computing methods of fuzzy logic and neural network methods. These membership functions are applied based on the parameter map to predict the reusability on the software system. The sensitivity analysis on two metric model is applied to identify the strength of new developed software product. Monga et al. [83] also worked on the qualitative attributes such as understandability, flexibility, maintainability, size, independence and portability to estimate the level of reusability. The association level analysis between existing and a new software product is discussed to quantify the software system development. Gupta et al. [84] mentioned potential features of a software system under specification of quality features including reusability, maintainability and testability. The software component usage and a change analysis are defined by the author to evaluate the degree of new software system composition.

2.8.1.4 Metrics Based Estimation
Software metrics are best described as the measurement components, units, processes or the activities to analyse a software system under different aspects. These are the imperative factor to measure software quality. Software metrics [85] are defined through various quality attributes to measure various aspects, process and features of software system development. Singh et al. [86] performed an evaluation of new and existing software using six CK metrics. The fault proneness in new software system is analysed using MOOD and QMOOD model. The extension observation in a software system is identified using number of child components (NOC). The reuse indicator used the complexity relation evaluation between existing and a new software system. The reusability prediction and complexity metric specification with regression algorithm is provided to estimate the reusability in additive regression form.
The resembling in the software systems under complexity metric feature and extension behaviour is also discussed in literature to measure the black box evaluation on the complexity metrics. Vinobha et al. [87] applied the inheritance metrics to measure the reusability for aspect oriented software system. The design time evolution of software systems under attribute, quality and reusability vectors is defined to define a new software system. The extension to the existing functionality and reuse of these features in new software system can be conducted to generate new software system artifacts. The concrete aspects at multiple level can be measured under the effect of mathematical formulation. Various inheritance specific metrics, including DIT (Depth of Inheritance Tree), NOC (Number of Children) are used by the author to identify the degree of reusability. Roatru et al. [88] presented the mathematical model based software metrics to characterize the reusability adaptability to control a software system observation and the feature adaptive with alternative flow analysis. The functional evaluation and measure of quality components is discussed in this research. The composability, portability and modularity in a new software system is analysed to recognize the degree of reusability.

The metrics are the integrated processes to measure the response and outcome of a particular process or stage. These are able to quantify the reusability of a particular component, process or system respective to different features. The complexity, size, reusability of software product or process can be evaluated through these metrics. The attributes that describe the definition of software metrics include lower quality attributes which includes coupling, cohesion etc. The high quality attributes measure applied in a generic way in different software development technology includes reusability and maintainability. The single quality can be defined to assess the featured aspect of a software system or the component. The outcome driven mapping can be implied to measure the quality of the software systems. While defining the software metrics, some of the discussed constraints are as follows [89]:

### 2.8.1.4.1 Relevance

The relevancy of the software metrics to the application, process and domain is required to be mapped. The systematic mapping method must be adopted under consideration of systematic mapping. The relevancy is able to categorize the metrics based on the technology, usage, environment and the required criteria.

### 2.8.1.4.2 Environment
The environment in which the metrics or measures are adopted also defines the context on which the measures are implied. The metrics are adopted to multiple scenarios and adopted in different contexts. The selection of qualitative and quantitative measures also depends on the environment in which they are implied. The accuracy level and the validation requirements are also decided based on the environment. The catalogue control metrics can be defined for each category to evaluate the relative process or the component.

2.8.1.4.3 License Type
The open source metrics or the maintainability assessment can be regulated on a proprietary software system to take the decision about the reliability of the particular product, feature of the version. The source specific metrics can be defined to evaluate the quality of product component. The decision rules are also defined to control and measure the product features. The open source product is required to analyse under risk vector and types of licensing so that the extended software product can be analysed and measured.

2.8.1.4.4 Internal Attributes
The quality attributes to verify the quality of a software product can be measured under various architectural constraints. These constraints include size, complexity, coupling, cohesion and the architectural features. While deciding the implied metrics, these are the primary features which are generated, evaluated for the process, component or software product.

2.8.1.4.5 External Attributes
These are the sub-characteristics measured through associated metrics. These metrics include stability, changeability, scalability, usability, adaptability, verifiability, adaptability, fault proneness etc. The context driven attributes can be measured to validate the quality at product and process level.

2.8.1.4.6 Other Methods
Rodriguez et. al [90] evaluated the reusability of CBS under dependency and property level estimation. The relational estimation of existing and a modified software product is performed under cost and property driven estimation. The static and dynamic analysis of the structure of software components is provided based on properties, procedures and conditions. The similarity, compatibility and multiplicity feature observation are defined to improve the features and reliability of the software systems. The semantic evaluation and configuration
observation of a software system is also considered to compute the functionality and accuracy in comparison. Efat et. al [91] used the feature prioritization method to customize the requirement for a new software system development and to estimate the degree of reusability in such software system. The quality of a new software system is evaluated under reference of five different aspects. The component quality estimation is considered respective to the organizational perspective. The developer specific composability analysis is performed to enhance the degree of reusability. The framework level composition is used to improve the adaptability of software components.

Amin et. al [92] defined the conceptual model for measuring the reusability during evolution. The assessment of reusability is done based on the potential perspective of reusability. The maintainability index complexity evaluation is enabled for package level evolutionary analysis using common language and tools. Maintainability evaluation and reusability evaluation are also discussed to identify the component level changes in a software system. Guojie et. al [93] proposed the component design model for structure level reusability. The interactive interface observation is provided to generate the extended software system. The environment specific, component specific and application specific extensions are defined by the author with component level interfacing. The effort level expandability to the normalized computation is given to configure the content level reusability in the software system. Yu et. al [94] performed a finite element analysis to improve the reusability. The systematic observation of the software system under assumption, computational modelling and theoretical measure is facilitated to recognize the requirement and variabilities. The maintainability specific domain engineering is discussed to improve the requirement specific observation and to analyse the variability of software systems.

Marshall et. al [95] measured the readiness level as the feature to identify the potential reusability in a software system. The process driven analysis with precision improvement is observed to evaluate the readiness in a new system. The region specific analysis and feedback driven observation for a software system is highlighted in this paper. The concept of characterization and support of software reuse is discussed which reduces various ambiguities in new software development. Burgin et. al [76] defined a metric based theory to explore the concepts associated to software system reusability. The properties, formulation are done in terms of quality characteristics, such as operability, customisability, clarity, learnability, attractively, helpfulness, understandability and explicitness. The reusability metrics associated
with software quality constraints are defined by the user. The context driven evaluation formulates the extensive metrics level derivation for the software systems.

2.8.2 Benefits of Reusability
The reusability is the key aspect of software development in which a new software system can be developed by utilizing the existing components. The function and structural map of existing component reduces the efforts and enhances the reliability of a new software system. Some of the benefits of software products reusability are:

2.8.2.1 Reduced Maintenance Time
As the used software components are fully tested, the overall maintenance of a software system in which these components are reused can be significantly reduced. The qualified components are having lesser integrated errors or faults at the functional level and structural level. The efforts and time in the maintenance of a software system may also be reduced. The reusability improves the functionality along with reduction of errors in the software systems [96].

2.8.2.2 Reduced Development Time
The inclusion of ready-made component or software module or code segment reduces the time to develop that code or a component. The time required to design the module, utilize the resource and to manage the activity is also reduced. The reusability increases the productivity at design time. The component specific development reduces the user efforts in the software system development. Each developed module needs not to be designed from scratch, instead utilizing an existing component can improve the reliability and productivity of a software system [2, 97].

2.8.2.3 Quality Improvement
The software systems developed using reuse of the existing components will have lesser errors or problems. The reusability is able to improve the simplicity, efficiency and reliability of the software systems [98].

2.9 ASSESSMENT OF SOFTWARE QUALITY
Software quality is a feature which is recognized as the excellence of a particular component, process or the software system. The quality can be measured as degree of accuracy, user
satisfaction and task completion. The quality characterization can be obtained by applying various mathematical and statistical viewpoints. To improve the quality of a software system, the defect healing is required for the software projects. Various object oriented and procedural metrics exist those are able to find the fault as compared to traditional complexity and size specific metrics. The faults can be identified at the code level or dynamically at run time. The post release faults can be identified during the beta testing or alpha testing. Most of CK metrics are validated for the fault identification and evaluation [99].

Peng et al. [100] validated a metric set for software fault prediction of different scenarios. Author defined the metric subset along with the investigation guidelines, training data and classifier specifications. These predictors are validated as the performance predictor using statistical measures. Zivkovic et al. [101] evaluated the acknowledged project efforts and identified them as interchangeable metric sets. Various descriptive statistics and measures are applied to various open source projects to validate the significance of presented quality metrics. Qsmundson et al. [102] developed a metric to measure the quality in software management. The analytical observation is presented by the author for four main aspects called requirement management, planning management, risk management and people management. The QMM (Quality Management Metric) is considered to be effective for a project manager to take the administrative decisions and to improve the performance of software management. Barkmann et al. [103] validated various quality metrics with statistical significance for open source software products. The metrics are applied independently and collectively to identify the criticality existence in the software system.

Oliveria et al. [104] identified various quality metrics for embedded software projects and categorized them as traditional metrics, product based metrics, process based metrics and object oriented metrics. Various physical concerns for embedded systems are also combined while highlighting the quality behaviour. The physical features including performance, power, weight, memory energy are considered as the attributes while evaluating a software product and its usability to the other software systems. Alexan et al. [105] used the code specific weak links to identify the integration between the external metrics. The correlation measures are defined to measure the quality in composite form. Various features and measures are also implied to estimate the software quality.
As the software system consists of large set of software metrics and measures, various optimization methods can be applied to reduce the complexity of these measures. One such dimension reduction method using the clustering approach is provided by Turan et al. [106]. Author used the clustering adaptive similarity measure to identify the most effective set of metrics. The difference specific measure is identified using the backward feature selection method. The dimension reduction is also applied to enhance the simplicity in the processing metric specification and it improves the performance of the software product by working on most appropriate metric set. PSO (Particle Swarm Optimization) is used as another optimization method by Ali et al. [107] for metric parameter selection. The optimization score based on Fmeasure, odd ratio and power features are used in combination to optimize the effect of software metrics.

The project level quality metrics are discussed by Huang et al. [108] to control the behaviour of software development and practices. The project metrics, quality management with knowledge distribution is measured by the author. The quality metrics are evaluated over the software development life cycle. The product level metrics are evaluated under complexity and performance criteria. The structural parameters of a software system are analysed under complex relationship estimation and the criteria specifications. Then the probabilistic measure is applied to efforts and product features to identify the criticality and quality of software products [109]. Zeman et al. [110] identified the usability metrics as the quality vector for new software generated from existing component. The performance and complexity measures are highlighted by the author. Shrivastava et al. [111] implemented a case study to measure the refactoring of a software system. The software reusability evaluation using hybrid metrics and features is performed in this work.

2.10 SUMMARY
This chapter presents the related concepts for re-engineering of legacy systems, reverse engineering, CBD, revolution of OO legacy systems, software metrics, reusability and software quality. Legacy software systems are regularly reengineered in order to contribute new functionalities or adapt them with new technologies. The majority of the re-engineering techniques are suggested for OOS and orientation of migration is towards CBS because early adopters of object oriented paradigm discussed that benefits of the objects have been hard to realize and these systems have become legacy now. The main drivers of re-engineering are
changing technology and high maintenance cost. The major challenges faced during process of re-engineering are lack of expertise, adaptability and lack of documentation. CBD paradigm has been discussed in research to improve and supports these aspects. Research shows that reengineered legacy systems are having proven technology over original legacy systems and there are potential benefits of CBS over OOS. It is highlighted in the literature that while migration is performed, assessment of reusability and in turn quality of software systems is also needs to be monitored. The study specifies the options of re-engineering, reverse engineering and ontology extraction for migration of OO legacy systems to CBS with reusability evaluation using different sets of metrics.