Introduction to Research Work

With increasing importance placed on standard quality assurance methodologies by large companies and government organizations, many software companies have implemented rigorous quality assurance (QA) processes to ensure that these standards are met. The use of standard QA methodologies cuts maintenance costs, increases reliability, and reduces cycle time for new distributions. In the crucial time of cost cutting, cost of quality assurance technique should also be major factor. To make software quality assurance (SQA) more cost-effective, the focus is on reproducible and automated techniques. The research work deals with the existing methodologies used as QA techniques and then evaluate mathematically them as per the cost and then it talks about generic algorithm for Quality Assurance. An SPER Model, which is based on SEI CMM standard, is recommended in the study that can generically deal with problems associated with existing QA techniques. In particular, we focus on configuration management, quality control, and testing as they are very much associated with QA of Real Time Complex System.

This chapter is an introduction to design patterns and the motivation behind detecting design patterns in software. We explain the research contribution and common vocabulary used throughout the thesis.

Design patterns, acting as recurring solutions to common problems, offer significant benefits such as avoiding unnecessary complexity, and promoting code reuse, maintainability and extensibility. There are several code based design pattern detection approaches which currently exist. However, these approaches have the drawback that the design pattern analysis is performed at implementation level and not at design level. This is at much later stage in the software development life-cycle, because of which good design decisions can not be incorporated at early stage of software development. That is why design based pattern recognition addressed as a part of our approach is more suitable for forward engineering activities.

As a part of our approach, we propose, the quality of software architecture can be analyzed on the basis of conformance to the design patterns. Patterns are often used in software architecture designs to implement simple concept and have proven their value over time. Design pattern symbolizes good arrangement of data and control as a part of software structure. Therefore it is useful to check the presence of a pattern in software architecture to assess the quality of the design. The quality can be quantified by the degree of conformance software architecture has
with a set of already defined design patterns. In our two pronged approach, this is the first factor which we have considered for quality analysis.

Cohesion in Class design reduces the need to edit multiple classes when making changes to application logic. A fundamental goal of OO design is to place the behaviour (methods) as close to the data they operate on (attributes) as possible, so that changes are less likely to propagate across multiple classes. Coupling indicates the level of interaction between several software components. Afferent (Ca) and Efferent Coupling (Ce) count respectively the number of ingoing and outgoing links between software entities, which are identified by Chidamber and Kemerer as indicators of their modularity and reusability. As a matter of fact, the more a software entity is autonomous, the more it is easy to separate it from its original application; therefore it is reusable. The software structures depicting design pattern should for a cohesive unit serving towards a single purpose. Such software structures are also independent of the outside functionalities are should prove to be self-sufficient to execute the required task. In other words, there should be as less dependencies as possible between the elements of a design pattern and the ‘outside’ elements leading to minimum amount of coupling. Meaning the number of links between the ‘inside’ and ‘outside’ elements should be low, the degree of interaction between these elements should be low and the knowledge of the states of the ‘outside’ elements for the ‘inside’ should be minimum. As a second part of our two pronged approach, the second metric of quality assessment is the cohesion and coupling properties of the detected design patterns.

Some approaches have talked about the methodologies of the analyzing the software quality by assessing a cohesion and coupling metrics at a class or a package level [18]. But, our approach is a novice one to connect the design patterns identification and software metrics to accesses the software design quality. We have designed a tool, DPI (Design Pattern Identifier) to analyze both these parameters, viz pattern conformance and coupling index for quality analysis feedback for software structures at design time.

For measuring software quality, majority of approaches focus on metric calculation based on code, which comes very late in the software development life cycle. If the calculated Object Oriented metrics are poor at this stage lot of efforts are needed to correct them. Therefore it is better to calculate the Pattern Software Metrics at Design Phase to assure Software Quality well before the Implementation starts. This paper describes how Design Patterns can be identified from UML class diagrams and their quality also can be assessed at the same time i.e. at Design
phase. This paper describes a win-win situation both for quality assurance persons as well as maintenance persons as their work is supplemented by our tool, DPI.

1.1 Statement of the problem
Software design patterns abstract reusable object-oriented software design. Each pattern solves design problems that occur in every day software development. The detection of design patterns during the process of software reverse engineering can provide a better and faster understanding of the software system. The first time the term design pattern was related to software development was in the book Design Patterns - Elements of Reusable Object-Oriented Software written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (14). The group of authors is also known as the Gang-of-Four or GoF. They discuss object-oriented design techniques that are based on their experience as software developers. They introduce 23 design patterns using a consistent format of information for each pattern to provide a uniform structure. Using the intent, trade-offs and graphical notations for design patterns, software engineers can decide which design pattern solves their design problems. It also makes it easier to discuss design problems and solutions with colleagues by using a common vocabulary. It is also useful documenting which design patterns have been used in a software so that other developers will get a better overview of the software without having to read the source code in detail. This thesis is concerned with the problem of detecting software design patterns. We present an approach that will detect software design patterns using their static structure - as described in class diagrams - as well as their dynamic behaviour.

The results can be used to verify the implementation of design patterns that were specified before the implementation phase. Having this additional information can be crucial during software maintenance. If well-designed software is poorly documented then this good design might be broken by a different developer that needs to add more functionality. These changes might introduce new bugs and problems and lead to degradation of the software. Therefore, it is important to document these design choices to improve the understanding of the software.

1.2 Definition of terms
This section provides a list of definitions that will be used throughout this thesis. These definitions are meant to help the reader and establish a common vocabulary. The terms will be
explained in more detail when they are first introduced; therefore it is not necessary to memorize them.

- **Structural analysis** - examination of class design without executing the program; provides understanding of the code structure.
- **Behavioural analysis** – inspecting the code and checking the sequence of method calls, provides details about the behaviour of the program.
- **Semantic analysis** – retrieving the pattern information based on naming conventions.
- **Role (also participant)** - class that has a specific characteristic/function in the context of a design pattern.
- **Intent** - encapsulates the idea and objective of a design pattern.
- **Candidate instances** - possible candidates for a design pattern. Each candidate instance is a set of classes. These classes possibly match the roles from a design pattern.
- **XMI** - XML Metadata Interchange Standard format
- **UML** - Unified Modelling Language

### 1.3 Research contribution

The purpose of this thesis is to present a forward as well as reverse engineering approach that will detect software design patterns in UML model for forward engineering and from Java source code as a part of reverse engineering. Our approach uses structural, behavioural and semantic analysis. We introduce behavioural and semantic analysis that removes false positives from our structural analysis results. We are interested in assessing the quality of the software design by checking whether it conforms to design pattern and calculating package software metrics. Based on these two parameters the quality of the software system can be analysed. We provide a tool that implements our approach.

### 1.4 Feasibility Study

Feasibility study determines whether the new solution proposed can be mapped to real world. Class Diagram: This will be the input given to the DPI and matching patterns to it will be identified as a match. UML can be used to provide structural details of the Class Diagram. As UML has now become the De-Facto Standard of the Modelling World, Class Diagram Designed using UML can be provide as an input to my system.