CHAPTER 2

LITERATURE SURVEY

2.1 PREAMBLE

This chapter presents the important inferences derived from the literature survey. An extensive study was made on the issues such as software architecture and its features, architecture analysis methods, design patterns and innovative patterns (proposed patterns) and their impact. The various models like SAAM, ATAM etc used at different levels were analyzed. The papers are grouped and discussed under different categories like software architecture and its features, architecture analysis methods and architectural analysis, design and patterns.

2.2 SURVEY PAPERS RELATED TO SOFTWARE ARCHITECTURE AND ITS FEATURES

This section presents some papers related to software architecture, importance and features of software architecture etc.

2.2.1 The Past, Present and Future of Software Architecture

This paper addresses the latest thinking about creating, capturing and using software architecture throughout a software system, its life cycle is a main issue. This paper also emphasize methods, techniques, tools and software engineering principles that support organizations taking an architecture-centric approach to software development (Kruchten et al 2006).
During the past decades, software architecture was used mostly in the sense of system architecture (meaning a computer system’s physical structure), then it grew to fill the gap between the system and the software. Compared to the past, now many rich ADLs are used. Along with ADL the following platforms such as J2EE, .NET, Symbian, Web Sphere and Scripting languages like Python and Perl are used (Bosch 2004; Kitchenham and Peeger 2002).

2.2.2 Analysis of Architecture Evaluation Data

The view of this paper is described below.

The output of software architecture evaluation is analyzed and the goal of the analysis is to find patterns in the important quality attributes, and the risk themes identified in the evaluations. The major results are the categorization of the risk themes, the observation that twice as many risk themes are risks of omission as are risks of commission, the failure to find the relationship between the risk theme and business, mission goals of the system, a failure to find a correlation between the domain of a system being evaluated and the important quality attribute for that system, and a wide diversity of names used for various quality attributes (Len Bass et al 2008; Clements et al 2002; Kazman et al 1999).

An affinity diagram (to discover a meaningful group of ideas from a raw list) is used to group the risk themes into categories. The 3 groups of risk themes are Architecture, Process and Organization (BAPO model). Architecture is further refined into runtime qualities such as availability, performance, security and development time qualities such as modifiability and integration. A process group contains process and tool, requirements, allocation, and documentation. An organization group contains big picture, unrecognized needs, product lines, awareness, scope, and coordination (Cortellessa et al 2005).
In addition to the categorization based on the affinity diagram, the risk themes are divided into categories based on their type. These are Risk of commission, Risk of omission and neither. The results achieved thus far, however, contain applications for practitioners, researchers and those performing architectural evaluation (Robert Nord et al 2008).

2.2.3 A General Model of a Software Architecture Design Derived from Five Industrial Approaches

The view of this paper is described below.

The five industrial software architecture methods such as Attribute–Driven Design, Siemens’4 views, RUP’s 4+1 views, Business architecture process and organization and Architectural separation of concerns provide the fundamentals of architecture design. By comparing the five industrial software architecture design methods such as ADD, S4V, RUP 4+1, BAPO, and ASC, the commonalities are extracted. By using the general approach, a comparison made across the five methods to find similarities and differences. The great variation in terminology and description it is found that the five approaches have a lot in common and match more or less the ideal pattern introduced. This ideal pattern is the basis for an evaluation grid (Herrmann 2000; Yakoub et al 2004).

After analyzing these methods, a general model is formed. The model contains the following activities, architectural analysis, architectural synthesis, architectural evaluation and artifacts such as architectural concerns, context, ASR, candidate architectural solutions, validated architecture and backlog. From this model, commonalities (common theme that deals with quality requirements and multiple views) and differences such as the intent of the design approach, emphasis on which evolutionary prototype, driving forces to shape the architecture, architectural scope and process scope between the contributing methods are identified (Hofmeister et al 2005).
Finally, the important part of this general model is the inclusion of analysis and evaluation activities as part of architectural design (Philippe Kruchten et al 2006).

2.2.4 Assessing the Influence on Processes when Evolving the Software Architecture

The view of this paper is described below.

Changes in a business objective may raise changes in the software development life cycle or architecture. This change may influence the other. This paper describes these relationships and proposes a method for assessing the influence that a proposed architectural change can have on the process. This method is used to identify the need for changes in the processes to be able to utilize the advantages made possible due to the architectural evolution.

The case study supports the proposal that a structured method to assess the impact on the process when changing the architecture of a system helps to reduce the risks and to facilitate the envisioned business benefits. This also identifies other types of changes. Central to this method is the use of scenarios and process reference models. Combining the solution to the process requirements into strategies gives a possibility for stakeholders to easily understand the implications of the different decisions.

Future work includes detailing the description in the method, adding details on how each step should be performed, and also giving additional examples. Additional details need to be added regarding scalability and resource needs, for using the model in different types of organizations (Stig Larsson et al 2007).
2.2.5 Software Design and Architecture

This paper explores the design of a software which has been the focus of software engineering research and shows why design will remain a principal focus. The elements of software design, both process and products are discussed. Design is an activity that is engaged in by the stakeholders, acting throughout the life cycle and making a set of key choices which constitute the application’s architecture (Bachmann et al 2005).

Directions for design research are outlined in this paper including: (a) drawing lessons, inspiration, and techniques from design fields outside computer science, (b) emphasizing the design of application “character” as well as the application’s structure, and (c) expanding the notion of software to encompass the design of additional kinds of intangible complex artifacts.

Design and architecture, as described, comprise a broad field and arguably sits at the very core of software engineering. The absence of progress in any one of the areas, impedes the progress in the other. This is broadly based on the whole set of sub topics which constitute a grand challenge for software engineering and an appropriate, critical focus for software engineering research (Taylor et al 2007).

2.2.6 Software Architecture Reliability Analysis Using Failure Scenarios

The view of this paper is described below.

The size and complexity of software in embedded systems becomes a primary threat to its reliability. Several conventional reliability engineering techniques have addressed failures in hardware components, and usually assume the availability of a running system. In order to address this problem,
a Software Architecture Reliability Analysis Approach (SARAH) is proposed early at the software architecture design level and before a system is implemented. SARAH benefits from mature conventional reliability engineering techniques and Scenario-based software architecture analysis. Besides this, SARAH provides another distinguishing property by focusing on user-perceived reliability unlike the conventional reliability analysis techniques. It manifests itself in the prioritization and analysis of failure scenarios, based on a user perception model (Bedir Tekinerdogan et al 2007).

SARAH defines the notation of the failure scenario model that is based on the Failure Modes and Effect Analysis method (FMEA) in the reliable engineering domain. This model is used to derive the Fault Tree Sets (FTS). The FTS provides a severe analysis for the overall software architecture and the individual architectural elements. For example, SARAH is applied in industrial cases in which Digital TV architecture is introduced.

In FMEA, the five attributes of a failure scenario are identified as failure id, related component, failure cause, failure mode and effect. SARAH offers a failure analysis report that can be utilized to identify architectural tactics for improving the reliability of the software architecture.

2.2.7 Software Change Impacts on Evolving Perspective

The view of this paper is described below.

As software engineering practice evolves to respond to demands for distributed applications on heterogeneous platforms, software change is increasingly influenced by middleware and components. Inter-operability dependency relationships now point to more relevant impacts of software change and necessarily drive the analysis.
Software changes to software systems that incorporate middleware components like web services, and exposes these systems and the organizations they serve to unforeseen ripple effects that frequently result in failures. Current software change impact analysis models have not adequately addressed this trend. Moreover, as software systems grow in size and complexity, the dependency webs of information extend beyond most software engineer’s ability to comprehend them (Bannerman 2009).

This paper examines the preliminary research for extending current software change impact analysis to incorporate inter-operability dependency relationships for addressing distributed applications and explores three dimensional (3D) visualization techniques for more effective navigation of software changes (Bohner et al 2002).

### 2.2.8 Recommended Best Industrial Practice for Software Architecture Evaluation

The view of this paper is described below.

Architectural decisions have a great impact on the consequent quality of software systems. As a result, it is important to evaluate how software architecture meets its quality demands. Though much focus has been placed on modeling and describing the software architecture as a design artifact, this paper found that relatively little is known about the current experience with software architecture evaluation (Lung et al 1997; Clements et al 2002).

The details of the results of two workshops to address this issue on software architecture evaluation, held at the Software Engineering Institute (SEI) on November 9-10, 1995 and May 9-10, 1996 are presented. The purpose of the workshops was to determine the state of industrial practice in
the evaluation of software architectures with respect to a set of desired quality attributes, and to uncover recommendations for best practices.

2.3 SURVEY PAPERS RELATED TO ARCHITECTURE ANALYSIS METHODS

This section discusses some papers related to software architecture analysis methods, its characteristics, the comparison of all methods based on various criteria etc.

2.3.1 A Survey of Software Architecture Analysis Methods

The view of this paper is described below.

The main purpose of the architecture evaluation of a software system is to identify risks and quality requirements. The taxonomy of formally defined orthogonal properties of Software Architecture (TOPSA) has three dimensions: like abstraction level, dynamism and aggregation level. Evaluation techniques at the architectural level are questioning and measuring. Questioning techniques generate qualitative questions. Measuring techniques suggest quantitative measurements. Most of the architecture analysis methods use scenarios.

Analysis methods are Software Architecture analysis method (SAAM), SAAM founded on Complex Scenarios (SAAMCS), Extending SAAM by Integration in the Domain (ESAAMI), Software Architecture Analysis method for Evolution and Reusability (SAAMER), the Architecture Trade-Off Analysis Method (ATAM), Scenario based Architecture Reengineering (SBAR), Architecture Level Prediction of Software Maintenance (ALPSM), and A Software Architecture Evaluation Model (SAEM) (Colquitt and Leaney 2002; Molter 1999).
The framework elements for the comparison of Analysis Methods include Specific goal, evaluation techniques used, quality attributes, SA description, stakeholders involvement, Method’s activities, Reusability of an existing knowledge base, and Method validation. All these methods use scenarios except ATAM, SBAR, and SAEM. Quality attributes are Modifiability, Flexibility, Reusability and quality model. The activities of the methods differ in complexity and aggregation level. By discussing evaluation methods, it is found that both qualitative and quantitative progress is observed at ATAM, because it focuses on finding the trade-off points. By considering the reusability of an existing knowledge base ATAM uses ABAS, and ESAAMI proposes packages of analysis templates. The selection of a suitable method depends on how well each comparison element fits into the context of the problem. The problems involved are scenarios and naming quality attributes (Dobrica et al. 2002; Duenas et al. 1998).

2.3.2 The Architecture Tradeoff Analysis Method

The view of this paper is described below.

The Architecture Tradeoff Analysis Method (ATAM), is a structure technique for understanding the tradeoffs inherent in the architectures of software-intensive systems. This method was developed to provide a principal way to evaluate the software architecture’s fitness with respect to multiple competing quality attributes: modifiability, security, performance, availability, and so on (Bate 2008).

These attributes interact; and often comes at the price of worsening one or more of the others as is shown in this paper, and the method helps us to reason about architectural decisions that affect quality attribute interactions. The ATAM is a spiral model of design: one of postulation candidate
architectures followed by analysis and risk mitigations, leading to refined architectures.

The ATAM was motivated by a desire to make relational choices among competing architectures, based upon well-documented analyses of system attributes, concentrating on the identification of tradeoff points. The ATAM also serves as a vehicle for the early clarification of requirements. As a result of performing an architecture tradeoff analysis, the understanding of the system’s ability is enhanced to meet its requirements (www.sei.cmu.edu/architecture/ata_method.html) (Rick Kazman et al 1996).

2.3.3 Expanding the View on Complexity within the Architecture Trade-Off Analysis Method

The view of this paper is described below.

The ATAM process is presented in terms of a people and system dimension. Research background and Methodology involves Research situation, ATAM, Action Research and interpreting the outcomes. The Architecture based Trade-off Analysis Method is used throughout the life cycle. The steps involved in Phase one are, the introduction of the method to participants, presenting the business case and solution architecture, identification of architectural approaches, generation of a quality attribute tree and analysis of architectural approaches. Phase two steps deal with decision key to specific quality, multiple qualities and risks (www.sei.cmu.edu/architecture/ata_method.html) (David Colquitt et al 2007).

Action Research is to perform both roles and also accepting the change. The method adopted for AR is the Susman Action Research Method. People dimension to complexity includes an individual’s capabilities, the influence of beliefs and roles on participants, based on the diversity of
language and differing world views. Systems dimension to complexity involves the Concomitant nature of the problem and solution, nature of understanding, development of usage aspects, and environmental influences. The impact of the situational complexity of the ATAM process falls into three categories, like the difficulty to associate and understand the business strategic and quality system perspectives, potential mismatch in communications, exhaustive understanding of requirements, ensure customer satisfaction.

The complexity involved in the design of an ATAM process includes covering broader issues within business drivers, dependency of system qualities on objectives, stable understanding of business needs, and modeling system goals to system qualities. The conceptual nature of architectures challenges the design process. The process itself needs to adapt in order to provide the social framework. Enhancement is proposed through a greater focus on the content aspects of the system (Colquitt and Leaney 2002).

2.3.4 Application of an Evaluation Framework for Analyzing the ATAM

The view of this paper is described below.

For all disciplines and fields, evaluation is a critical process. In software engineering, to develop and analyze an evaluation method, a framework of six basic components can be applied. The components are Target, Evaluation Criteria, YardStick, Data Gathering Techniques, Synthesis Techniques and Evaluation Process. This framework is based on Software and non-Software disciplines. This paper proposes some of the analyses of theoretical and methodological evaluation concepts.
This framework is used to analyze the complete definition of the Architecture Trade off Analysis Method (ATAM) and provide suggestions for improvement. The target of ATAM is architecture and during the evaluation, a description of the following views, such as functional, module, process and hardware of the architecture is given. An evaluation criterion is based on the characterization of the quality attributes of ATAM. Yardstick is a set of scenarios which is associated with the evaluation criteria. Data Gathering techniques are applied in ATAM to obtain information about the target by analyzing the architectural approaches. Synthesis techniques are closely related to the data gathering technique to obtain the output of ATAM. The evaluation process consists of four phases, preparation and partnership, initial evaluation, complete evaluation and follows up (Kazman et al 2001).

As a result, this framework is used to identify the set of characteristics and improvements in an analysis of ATAM. The identified characteristics are, that an evaluation team analyzes each quality attribute, there are no predefined criteria and a yardstick can not be applied because it is a set of scenarios that provides strong collaboration of stakeholders during the evaluation process. Other suggestions described are the explicit description of the target, explicit development of data gathering and a synthesis technique to provide a more detailed definition of the evaluation process and relationship among the components, and the development of basic architectural styles (ABAS) to be used as a yardstick (Marta Lopez 2002).

2.3.5 Formally Analyzing Software Architectural Specification Using SAM

The view of this paper is described below.

The Software Architecture Model (SAM) is a general software architecture development framework based on two complementary
formalisms—Petrinets and temporal logic, and it is a graphical formal software architecture description model. Petrinets are used to visualize the structure, and model the behavior of software architecture, and temporal logic is used to specify the required properties of software architecture. SAM is used for visualizing, specifying and analyzing software architectures. SAM must support precise specification and formal analysis. SAM must be expressive, flexible, understandable and usable.

In SAM, software architecture is visualized by a hierarchical set of boxes with ports connected by directed arcs. These boxes are called compositions. Each composition may contain other compositions. The bottom–level compositions are either components or connectors. The behavior of a component or a connector is explicitly defined using a Petrinet. An example of a Petrinet model is the dinning philosopher problem. In these problems there are three places (Thinking, Chopstick and Eating) and two transitions (Pickup and Put down). Depending on the given Petri net models, different temporal logics are used. Some temporal logics are values, state variables, states, rigid variables, rigid functions, predicate symbols, state functions, predicates, transitions, temporal formulas, etc.

Analyzing SAM specification using symbol model checking. Modeling the alternative bit protocol, specifying alternating bit protocol (ABP). Analyzing SAM specification of the ABP using symbol model checking. The software model checker (SMC) system, translating a SAM specification into an SMC program. SAC provides a unique hierarchical framework to tailor the dual formalism for software architecture specification while other works aimed to provide a general specification methodology. SAM also provides symbolic model checking software specification in general, and model checking software architecture specification in specific (Xudong He et al 2002).
2.3.6 A Software Architecture Approach for Structuring Autonomic Systems

The view of this paper is described below.

Autonomic systems manage themselves when high-level objectives are given to their administrators. They utilize feedback from their own execution and their environment is self-adapt in order to satisfy their goals. An important consideration for such systems is a structure which is conducive to self-management. This paper presents a structuring methodology for autonomic systems which explicitly models self-adaptation while separating functionality from evolution (Kazman and Bass 2002).

This paper has presented a structuring methodology based on the concepts of producers and evolvers as part of a unified framework for building autonomic systems. The novelty of this approach is the combination of feedback and change mechanisms within a Calculus based strongly typed executable architecture description language and a development methodology designed for change.

This paper also suggests that it is beneficial in developing evolvable systems. One of the avenues for further research is the integration of constraints into the evolution framework. At present it is able to evolve both architectures and behaviors as they execute. Further work also needs to be done regarding the implications of that ability for autonomic systems (Dharini Balasubramaniam et al 2005).
2.3.7 Scenario-Based Analysis of Software Architecture

The view of this paper is described below.

Software architecture is one of the most important tools for designing and understanding a software system, whether that system is in the preliminary design, active deployment, or maintenance. Scenarios are important tools for exercising architecture to gain information about a system's fitness with respect to a set of desired quality attributes (Rick Kazman et al 1996; Krutchen 1995).

A structured method for scenario-based architectural analysis, called SAAM, is discussed. SAAM uses scenarios to analyze architectures with respect to achieving quality attributes. Finally, lessons and morals are presented, drawn from the growing body of experience in applying scenario-based architectural analysis techniques.

Architectural analysis helps to improve communication among development team members, and between team members and "outsiders" (upper-level managers, clients, users). Part of this improved communication is accrued simply by choosing a common syntactic and semantic notation for architectural representation. A much larger part of the improved communication, however, arises because scenario-based software architecture analysis helps to focus high-level and global software design discussions on specific problem areas. This analysis motivates development teams to critically evaluate and discuss the architectural alternatives of their system early in the life cycle.
2.4 SURVEY PAPERS RELATED TO ARCHITETURAL ANALYSIS, DESIGN AND PATTERNS

This section discusses about some papers related to architectural analysis, design and patterns and their impact on software architecture etc.

2.4.1 Architecture-Level Modifiability Analysis (ALMA)

The view of this paper is described below.

In order to reduce the cost, the organization addresses the modifiability during the system’s development. To address this modifiability, Scenario-based analysis methods are used. Finally, the Architecture-Level modifiability analysis (ALMA) is introduced from the combination of modifiability and Scenario-based analysis methods. This ALMA addresses a unified Architecture-level analysis method that focuses on modifiability, distinguishes multiple analysis goals, has explicit assumptions and provides repeatable techniques for performing the various steps (Lassing et al 2002).

The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specifications. The software architecture of a program or computer system is the structure or structures of the system, which comprises the software components, the externally visible properties of those components and the relationships among them. In order to analyze the software architecture, we should find the properties and quality requirements that are implemented and predicted by the system.

ALMA has the following five steps to perform maintenance predictions, such as Set goal, Describe software architecture, Elicit scenarios, evaluate scenarios and interpret the results. Concerning the prediction, a prediction model based on the estimated change volume and productivity
ratios is presented. The result of the prediction is an effort estimate currently lacks a frame of reference (PerOlof Bengtsson et al 2002).

2.4.2 Design Patterns: Between Programming and Software Design

The view of this paper is described below.

In computer science, the two areas, programming and software are usually separated. The design patterns and software design which tries to bridge the gap between these two areas. Design patterns are on the border of programming and software engineering that can be covered from both sides either as an advanced programming course or as an application of software design and micro architectures (Eden and Kazman 2003; Frank Buschmann et al 1996).

Both approaches can be refined so as to bridge the gap between programming and designing. This paper suggests that taking up the design patterns first and then software design, helps bridging the above mentioned facts from a programmer's perspective. We could see how patterns are applied in a complete system and gain a feeling about what is a “good” design (Christopher Denzler et al 2008; Etzkorn et al 1999; Larman 2004).

2.4.3 Aspect-Oriented Analysis for Architectural Design

This paper proposes an aspect-oriented analysis method, in which it is possible to capture the commonality and differences between softwares, considering their requirements on quality attributes.

In order to design a software architecture that is robust towards evolution, to analyze the commonality and differences among potential software that can be developed, and design a software architecture so as to accommodate the commonality and differences (Ferrari et al 2009).
When the architectural design is analyzed, it is important to consider not only the functional aspects of the software, but also aspects relating to quality attributes such as performance and reliability. Because, software architecture can be different if the requirements on quality attributes is different and if functional requirements are exactly the same.

This paper shows how to examine the requirements from each aspect, how to merge them, and how to determine the direction of the architectural design. This method is based on experience in which a simple layered architecture caused problems on quality attributes, such as performance and memory size (Tomoji Kishi et al 2002; Kroes 2002).

2.4.4 Finding your Innovation Sweet Spot

The view of this paper is described below.

A Systematic process is based on five innovation patterns. Powerful patterns are Subtraction, Multiplication, Division, Task Unification and Attribute dependency change pattern

The approach followed in the subtraction pattern is “Instead of trying to improve a product by adding components, remove undesirable components but to avoid drifting too far from the task, look for the replacement”. Philips consumer electronics used the subtraction pattern and came up with an idea of removing the local display and all control buttons on the DVD player. After testing the idea, the company found that it could get by with just one button, and the remaining operations moved to a graphical user interface (remote control).

The multiplication pattern represents a different approach: Instead of removing an element, it is better to make one or more copies of an existing
product. An example of this type is the Gillette double bladed razor. Instead of adding an extra blade for more shaving, add it at a slightly different angle.

The approach taken by the Division pattern is, dividing the existing product into its components. Division can take a number of forms: Physical division (a product is cut along the physical line), functional division (components with different functions are separated), and preserving division (a product is divided in such a way that each part preserves the characteristics).

The Task Unification Pattern approach unifies two tasks in a single component. An example of this pattern is the defrosting filament in an automobile windshield.

Attribute Dependency Change involves the dependent relationships that exist between the attributes of a product and its immediate environment. Example: Take two standard pairs of eye glasses; there is no dependent relationship between the color of the lens and external lighting conditions. By creating a dependent relationship, a lens that changes color when exposed to sunlight is used, thus eliminating the need to buy a separate pair of glasses for sunny days. Attribute dependency change is applicable in a variety of situations but it is difficult to apply.

Guidelines are used for choosing the right tool. For example, highly complex products start with subtraction. If the aim is to control cost, then task unification is tried, which encourages an efficient use of resources, or subtraction which eliminates costly product components. Patterns are used in connection with one another. A method that focuses on the product (what is essential? what can be rearranged, removed) can enhance the company’s current idea generation methods (Jacob Golden Berg et al 2003). (Based on this paper only the innovative patterns have been proposed).
2.4.5 A Survey of Architecture Design Rationale

The view of this paper is described below.

There is a problem about the consequence of not documenting the design rationale. In order to address this problem, this paper reports a survey of practitioners to probe their perception of the value of design rationale and how they use and document the background knowledge related to their design decisions. In order to achieve this, this paper develops a conceptual framework and associative tools to facilitate the capture and use of the design rationale (Bratthall et al 2000).

The findings of survey of practitioners who have had experience in architecture design and how design rationale are used, documented and perceived by designers and architects provide the following objectives: understanding design constraints, strength and weakness from architect perceptions, to determine frequency of documenting and reasoning with different elements of design rationale and reason for not documenting design rationale, to identify the potential challenges and opportunities for improving the documentation of design rationale (Erich Gamma et al 1995; Garcia 2009).

Some of the methods and tools for documenting the design rationale are, to apply organization standards and templates (word, excel, visio, power point), UML tools, IBM GS methodology, to document architecture decisions using the formal method and notation, internally developed tools, QMS design template documents, requirements traceability matrix and architecture tool CORE. Practitioners view design rationale as important but there is a lack of methodology and tool support. To address these issues identify technical and socio-technical factors that influence design decisions (Tang and Han 2005).
2.4.6 The Essential Components for Software Architecture Design and Analysis

The view of this paper is described below.

Architecture analysis and design methods such as ATAM, QAM, ADD and CBAM have enjoyed modest success in recent years and are being adopted by many companies as part of their standard software development processes. They are used in the software lifecycle, as a means of understanding business goals and stakeholder concerns, mapping these onto an architectural representation, and assessing the risks associated with this mapping. These methods have evolved a set of shared component techniques. How these techniques can be combined in countless ways to create needs-specific methods will be shown. The generality of these techniques by describing a new architecture improvement method called APTIA (Analytic Principles and Tools for the Improvement of Architectures) will be demonstrable. APTIA almost entirely reuses pre-existing techniques but in a new combination, with new goals and results. Lastly, APTIA’s use in improving the architecture of commercial information systems is exemplified (Bondi 2009; Dorst and Dijkhuis 1995).

What are the consequences of precisely knowing the differences between the terms architecture, design and implementation? Among others, these distinctions facilitate-

- Determining what constitutes a uniform program, e.g., a collection of modules that satisfies the same architectural specifications (Paul Clements et al 2003);
- Determining what information goes into architecture documents and what goes into design documents;
Determining what to examine and what to not examine in an architectural evaluation or design walkthrough (Rick Kazman 2005; Eriksson and Penker 2000).

### 2.4.7 Model-Driven Approach to Software Architecture Design

The view of this paper is described below.

Software Architecture (SA) allows for an early assessment and design for the quality attributes of a software system, and it plays a critical role in current software development. Thus, it is extremely difficult to build complete and appropriate software architecture, even though it is recognized as a fundamental artifact. To address this difficulty it is necessary to define an architecture design method that enables the systematic and assisted construction of the SA of Enterprise Applications, taking into account major quality attributes involved in this family of systems (Lundberg et al 1999; Flood 1987).

The architecture is treated as a mega-model (a model composed of related models) and the application of design decisions is encoded in terms of model transformations. In this context, the architecture design activity can be seen as a large model transformation which obtains, from an initially empty architecture, the complete system architecture. This large transformation is composed of a sequence of smaller sub-transformations, each encapsulating the application of a design decision, i.e. the resolution of a particular architectural concern. Then, the set of sub-transformations available to the architect can be regarded as the definition of a family of large transformations, i.e. as all the possible ways to design the complete architecture from scratch (Avgeriou and Describing 2003).
Then, the application of Model-Driven Engineering techniques not only favors the evolution of the approach, but also increases its power. Using MDA not only favors modularization and reuse, but also organizes and systematizes the architect’s task.

From the literature survey analysis we can easily understand the views of software architecture and the models or methods which are used to analyze the software architecture, the impact of quality attributes and how to handle these to come out with a good architecture (Erman Coskun and Martha Grabowski 2005; Kiears and Polsen 1999).

2.5 SUMMARY AND CONCLUSION

This kind of a literature survey is helpful in understanding the importance of software architecture, its features, analysis and design level problems and solutions, role of patterns and their inter dependency with other patterns and the impact of various analysis methods to evaluate a particular architecture. In this chapter an extensive literature survey is presented. The various methods for the software architecture analysis, their complexity, and various models used, the impact of patterns and the related details are explained. The comparison and performance analysis of various software architecture analysis methods are discussed in the subsequent chapter.