CHAPTER 1

INTRODUCTION

1.1  PREAMBLE

This chapter provides a comprehensive introduction to software architecture analysis. It presents the basic concepts and features of software architecture, quality attributes and the various software architecture analysis methods.

1.2  SOFTWARE ENGINEERING

Software engineering is emerging as a promising discipline for developers of software. It is organized and clearly articulated as a new concern for the software system and has evolved over time as a natural evolution of design abstractions. Engineers have always been on the hunt for better ways to comprehend their software and contemporary ways to build sophisticated, larger software systems (Mary Shaw 1999; Mary Shaw and Paul Clements 2006; Brooks 1987).

The word “engineering” is used to describe following activities:

- Creating cost effective solutions
- Solving practical problems
- Applying scientific knowledge to solve critical problems
- Building things-Engineering emphasizes the solutions, which are usually tangible artifacts
- Service of command-Developing technology and expertise
Traditionally, engineering has emerged from an ad hoc practice in two stages. First, management and manufacturing techniques enable routine production. Later, the problems of production stimulate the development of supporting science (Mary Shaw 1999).

1.2.1 Evolution of Software Engineering as a Discipline

The following issues were prominent in the evolution of software engineering discipline:

- Specifications (abstract models)
- Software structure (bundling representation with algorithms)
- Language issues (modules, scope, user-defined types)
- Information hiding (protecting the integrity of information not in the specification)
- Integrity constraints (invariants of data structures)
- Rules for compositions (declarations) (Mary Shaw 1999; Balsamo Arlan et al 2004)

1.3 WHAT IS SOFTWARE ARCHITECTURE?

Software Architecture involves the description of elements from which systems are built, interactions among elements, patterns that guide their composition and constraints (Avison 1999; Garlan and Shaw 1994).

1.3.1 Architecture

From the late 1980s, the word “Architecture” was used mostly in the sense of system architecture (meaning a computer system’s physical structure) or sometimes in the narrower sense of the family of a given computer’s instruction set (Kruchten et al 2006).
1.3.2 Software Architecture

A Software Architecture system is a composition of components; it contains a Global control structure of the system, specifies the protocols for communication; Synchronization and data access; Assignment of functionality to design elements; Composition of design elements; physical distribution; scaling and performance; dimensions of evolution, and selection of design alternatives. This is the software architecture level of design (Garland and Anthony 2003).

Software Architecture is defined as,

- The profession of designing software
- The process of software design
- The conceptual structure of the software
- The actual structure of the code

The Software architecture of a program or computing system is the structure or functions of the system, which comprise the elements, the externally visible properties of those elements and the relationships among them (Bass Len et al 2003).

Software architecture involves

- The structure and organization by which modern system components and subsystems interact to form systems, and
- The properties of systems that can best be designed and analyzed at the system level (Philippe Kruchten et al 2006).

The architecture of a software system defines the system in terms of computational components and interaction among those components.
Components are elements such as clients and servers, databases, filters and layers in a hierarchical system. Interaction among the components at this level of design can be simple and familiar such as procedure call and shared variable access. This can also be complex semantically, such as client-server protocols, database-accessing protocols, asynchronous event multicast and piped streams (Mary Shaw 1999).

Research in Software Architecture (SA) description addresses the different perspectives one could have of the architecture. In general, it is the responsibility of the architect to decide which view to use to describe the Software Architecture (SA) (Bass Len et al 2003; Shaw and Garlan 2008).

1.3.3 The Status of Software Architecture

The architectural design of a system has always been the deciding parameter of the success of the system, for there are many useful architectural paradigms (such as pipelines, layered system, client-server organization, etc.); they are typically understood only in an idiomatic way and applied in an ad hoc fashion.

Software architecture has begun to develop as an important field of study for software engineering practitioners and researchers. Architectural issues are being addressed by work in areas such as,

- Module interface language
- Domain-specific architectures
- Software reuse, codification of organizational patterns for software
- Architectural description languages
The first category addresses the problem of architectural characterization by providing new architectural description languages.

The second category addresses the codification of architectural expertise. Work in this area is concerned with cataloging and rationalizing the variety of architectural principles and patterns that engineers have developed through software practice.

The third category addresses the frameworks for specific domains. Frameworks can be easily instantiated to produce new products in the domain (Ali-Babar et al 2006).

The fourth category is addressing the formal underpinning for architecture. New notations are developed as the practice of architectural design is better understood (Mary Shaw 2002). In the subsequent section we will discuss about the various quality attributes in detail.

1.4 QUALITY ATTRIBUTES

The principle of architecture evaluation of the software system is to assure its quality. One of the most important issues relating to the software development is quality, though it is not a new idea in the software field. The concept of software architecture (SA) has emerged as the appropriate level for dealing with software quality. This is because the scientific and industrial communities have recognized that Software Architecture (SA) sets the boundaries for the software qualities of the resulting system. It is recognized that it is not possible to measure the quality attributes of the final system based on the SA design (Colquitt and Leaney 2007). This would imply that the detailed design and implementation, represents a strict projection of the architecture. Future work is needed to develop systematic ways of bridging the quality requirements of software systems with their architecture.
Developers of critical systems are responsible for identifying the requirements of the application, for developing software that implements the requirements, and for allocating appropriate resources (processors and communication networks). It is not enough to merely satisfy the functional requirements. Critical systems, in general, must satisfy security, safety, dependability, performance, and other similar requirements as well.

Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability) (Mario Barbacci et al 1995; Klein et al 1999; Reed and Jacobs 1993).

The various quality attributes are addressed in the following section in detail.

1.4.1 Interoperability

Interoperability refers to the ability of a collection of communicating entities to share specific information and operate on it according to an agreed-upon operational semantics (Brownsword 2004). Increased interoperability is the most prominent benefit of SOA (Service Oriented Architecture), especially when we consider Web services technology (McGovern 2003). Distributed systems have been developed using various languages and platforms that vary from portable devices to mainframes. They have used technologies such as the Common Object Request Broker Architecture (CORBA), Remote Method Invocation (RMI), Distributed Component Object Model (DCOM), Remote Procedure Call (RPC), and sockets for communication. However, until the advent of Web services, there was no standard communication protocol or data format that could be used effectively by systems using different technologies to interoperate on a worldwide scale (Liam O’Brien et al 2005; Chidamber and Kemerer 1994).
However, the promise of cross-vendor and cross-platform interoperability in Web services begins to fall short when services start to use features beyond the basic Web Service Definition Language (WSDL) and Simple Object Access Protocol (SOAP) standards. Over the last few years, a myriad web services standards (e.g., Business Process Execution Language [BPEL], and WS-Security) has emerged from a number of standard bodies. Web services development platforms do not implement the same standards and the same versions; so, interoperability may not be as seamless in practice as it is in theory.

1.4.2 Reliability

Reliability is the ability of a system to keep operating over time (Clements 2002). Several aspects of reliability are important within an SOA, particularly the reliability of the messages that are exchanged between the application and the services, and the reliability of the services themselves. Applications developed by different organizations may have different reliability requirements for the same set of services. And an application that operates in different environments may have different reliability requirements in each one (Liam O’Brien et al 2005; Iannino 1994; Jackson 1988).

1.4.2.1 Message reliability

Services are often made available over a network with possibly unreliable communication channels. Connections break and messages fail to get delivered or are delivered more than once or in the wrong sequence. Although techniques for ensuring the reliable delivery of messages are reasonably well understood and available in some messaging middleware products today, messaging reliability is still a problem. If reliability is addressed by service developers who incorporate reliability techniques directly into the services and application, there is no guarantee that they will
make consistent choices about what approach to adopt. The outcome might not guarantee end-to-end reliable messaging. Even in cases in which the application developers defer dealing with reliable messaging to messaging middleware, different middleware products from different vendors do not necessarily offer a consistent approach to dealing with the problem. The use of middleware from different vendors might preclude reliable message exchange between applications and services that use different message-oriented middleware (Weerawarana 2005; Kanoun Kaaniche and Laprie 1997).

1.4.2.2 Service reliability

Service reliability means the service operates correctly and either does not fail or reports any failure to the service user. Service reliability also means making sure that the service is obtained from a reliable provider so that a level of trust in the service’s accuracy and reliability can be established. Issues that have to be dealt with, include managing the transactional context; for example, dealing with failures or some form of compensation if the service fails. In some cases, brokers or intermediaries may link the service users and providers. As a result, these issues may be handled by third parties.

1.4.3 Availability

Availability is the degree to which a system or component is operational and accessible when required for use. Availability of services both from the user’s and provider’s perspectives is a concern for the success of an SOA. From the service user’s perspective, if the system relies on a set of services being available in order to meet its functional requirements and one of those services becomes unavailable (even transiently), it could have dire consequences on the success of the system. From the service provider’s perspective, in order for the services to be used (for which the provider may receive compensation), they must be available when needed. Otherwise, the
provider’s finances and reputation could be impacted (especially if compensation has to be paid when the services are not available).

1.4.4 Usability

Usability is a measure of the quality of a user’s experience in interacting with information or services. To provide a more usable system, service providers should consider several things that derive from the distributed and service nature of SOA: data granularity, services to support usability, and disconnected operation (Liam O’Brien et al 2005; Karlsson et al 1995).

1.4.5 Scalability

Scalability indicates the capability of a system to increase performance under an increased load when resources (typically hardware) are added. A system whose performance improves after adding hardware, proportional to the capacity added, is said to be a scalable system.

There are two kinds of scaling:

- Vertical scaling means to add resources to a single node in a system, such as adding memory or a bigger hard drive to a computer.

- Horizontal scaling means to add more nodes to a system, such as adding a new computer to a clustered software application (Colquitt and Leaney 2007).
1.4.6 Security

Security, in this context, deals with enforcing that users or programs can only perform actions that they are allowed to perform, so that, for example, nobody can read or manipulate data that he or she should not have access to.

1.4.7 Performance

Performance deals with the speed of response and the throughput (from the location where the data is stored to the requester) of a storage system (Mario Barbacci et al 1995).

1.4.8 Extensibility

Extensibility is the ease with which the services’ capabilities can be extended without affecting other services or parts of the system. Extensibility for architecture today (in particular, an SOA) is important because the business environment in which a software system lives is continually changing and evolving. These changes in the environment will mean changes in the software system, service users, and service providers and the messages exchanged among them. Extending an SOA means making changes that include extending (Lundberg et al 1999).

- The architecture to add additional services. SOAs allow for the easy addition of new services through loose coupling and the use of various Web standards. Services can be created and published by the providers and discovered by service users. Service users must update their application code to incorporate these new services.
Existing services without changing the interfaces. Because services are loosely coupled, adding new capabilities to them that do not require a change in the service interface can be done without affecting other services. However, an application may require changes if these new capabilities were already incorporated into the application (i.e., the functionality for these capabilities was either included in the application or handled by additional services). Identifying the services’ capabilities when they are first designed and implemented is very important because later, changes may cause problems within the service users’ applications.

Existing services with changes to interfaces. Adding new capabilities to a service—ones that require changes to the service interface—may have a major impact on the success of an SOA. Usually, an application learns about a service’s interface by reading information provided by the directory provider, and the interface may change over time. The service users’ application must be able to handle any changes to the interface (Lundberg et al 1999).

A major obstacle to extensibility is the interface message. If interface messages are not extensible, users and providers will be locked into one particular version of the interface to a service. Moreover, messages must be written in a format, structure, and vocabulary understood by all parties. Limiting the vocabulary and structure of messages is a necessity for any efficient communication. The more restricted a message is, the easier it is to understand, although it comes at the expense of reduced extensibility. Restriction and extensibility are deeply entwined. Both are needed, and increasing one comes at the expense of reducing the other. Tradeoffs between them are necessary to achieve the right balance (Liam O’Brien et al 2005).
1.4.9 Adaptability

Adaptability means the ease with which a system may be changed to fit changed requirements. Adaptability for a business means, it can adapt quickly to new opportunities and potential competitive threats, which implies that the application development and maintenance groups within the business can quickly change the existing systems. The use of an SOA approach brings various benefits to the ability to adapt by allowing the following:

- Services can be built and deployed using the principles of location and transport independence and declarative policy. As a result, service users can dynamically discover and negotiate the method to be used for binding and the behavior to be exhibited for interacting with a service. If the service needs to adapt, this discovery and binding should be automated and not require a change in the application. It is the role of the application development group to build and maintain the abstraction layer between services and the underlying technology. It is the role of the architect to build applications that are resilient to infrastructure and organizational policy changes (Bruce 2004).

- Business processes that are modeled using services can be adapted, and those services can be combined in new and different ways. Additional services can be added, or adapted services can be swapped where they are needed. What will require changes is the underlying application using these services. In an SOA, in which most of the system’s functionality is contained in the services, this task should not be a major one, as compared to a system in which the majority of the underlying applications make up the business process, a large amount of which will have to change (Lassing et al 1999).
Services are being developed that must operate on different platforms, in different computing environments (including development and testing environments), using different combinations of sensors, multiple diverse communication protocols, human-computer interaction (HCI), and applications, and pursuing different missions. These services must be “configurable” to the environment in which they will reside—a significant adaptation challenge that requires “spiral” development with incremental deliveries to particular platforms, interoperability between different platforms, and backward compatibility to multiple previous releases (Lundberg L et al 1999).

To achieve adaptability, the services will need to be managed and monitored properly as a single cohesive solution, and the interaction between the service and the underlying infrastructure will have to be managed. Proper measurements of capacity, performance, and availability are required to support this management function (Liam O’Brien et al 2005).

### 1.4.10 Modifiability

Modifiability is the ability to make changes to a system quickly and cost-effectively (Clements and Northrop 2002). SOA promotes loose coupling between service consumers and providers. Services are self-contained, modular, and accessed via cohesive interfaces. These characteristics contribute to the creation of loosely coupled SOAs where there are few, well-known dependencies between services. That fact tends to reduce the cost of modifying the implementation of services, thus increasing the system’s modifiability. However, if service interfaces need to be changed, the change may create problems because once service interfaces are published and used by applications, it can be difficult to identify who is using a service and what impact changing its interface will have (Liam O’Brien et al 2005). Table 1.1 depicts the snapshots of all quality attributes discussed in the earlier section (Lassing et al 2001).
Table 1.1 Quality attributes

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interoperability</td>
<td>It is the property referring to the ability of diverse systems (or its components) with different specifications to work together.</td>
</tr>
<tr>
<td>Reliability</td>
<td>The ability of the system to keep operating satisfactorily over its specified life time. Reliability is usually measured by mean time to failure which defines the average time between failures.</td>
</tr>
<tr>
<td>Usability</td>
<td>The measure of a user's ability to utilize a system effectively. The ease with which a user can learn to operate, prepares inputs for, and interpret outputs of a system or component. A measure of how well users can take advantage of some system functionality. Usability is different from utility, which is a measure of whether that functionality does what is needed or not.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The ability to maintain or improve performance while system demand increases.</td>
</tr>
<tr>
<td>Security</td>
<td>A measure of the system's ability to resist unauthorized attempts of usage and denial of service, while still providing its services to legitimate users. Security is categorized in terms of the types of threats that might be made to the system.</td>
</tr>
<tr>
<td>Performance</td>
<td>Performance represents the overall responsiveness of the system which could be measured in terms of the time required to respond to stimuli (events) or the number of events processed in some interval of time. the number of transactions per unit time, or the amount of time that it takes to complete a transaction with the system.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>The ease with which a system or its components can be modified for use in applications or environments, other than those for which it was dedicated.</td>
</tr>
</tbody>
</table>
Table 1.1 (Continued)

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptability</td>
<td>The ability of a system to be both flexible and undergo change rapidly.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>It is the ability to make changes to a system quickly and cost effectively</td>
</tr>
</tbody>
</table>

In the subsequent section we will discuss about the various software architecture analysis methods.

### 1.5 SOFTWARE ARCHITECTURE ANALYSIS METHODS

The purpose of the software architecture evaluation method is to analyze the SA to identify potential risks and verify that the quality requirements have been addressed in the design. In addition, it helps to know the important properties of software, even before it is developed and to predict the downstream effects of the software.

Therefore, it has been decided to study such methods in order to cover as many particular points of view of objective reflections as possible, to be derived from the general goal. SA is considered the first product in an architecture based development process.

There are eight methods available to analyze software architecture.

- **SAAM** Software Architecture Analysis Method
- **SAAMCS** SAAM Founded on Complex Scenarios
- **ESAAMI** Extending SAAM by Integration in the Domain
- **SAAMER** Software Architecture Analysis Method for Evolution and Reusability
The various architecture analysis methods are discussed below.

### 1.5.1 Software Architecture Analysis Method (SAAM)

SAAM appeared in 1993, corresponding with the trend for a better understanding of general architectural concepts, as a foundation for proof that a software system meets more than just functional requirements. Thus, in the early stage of a system's development, the correction of architectural mistakes detected by the analysis is still possible without causing excessively high costs (Colquitt and Leaney 2002; Gamma et al 1995; Mary Shaw and Clements 2006).

The architecture analysis methods are described based on their Goals, Evaluation techniques, Quality attributes, Stakeholders' involvement etc.

**Specific goals:** A SAAM's goal is to verify basic architectural assumptions and principles against the documents, describing the desired properties of an application.

**Evaluation technique:** During the analysis, it is determined whether a scenario requires modifications to the architecture. Scenarios that require no modifications are called direct and those that require modifications are called indirect.
The **quality attributes**: The fundamental characteristic of this method is the concretization of any quality attribute in the form of scenarios.

The **stakeholders' involvement**: SAAM harmonizes the various interests of the stakeholder groups, thus setting up a common understanding of the SA as a base for later decisions.

**Reusability of the existing knowledge base**: SAAM does not consider this issue (Dobrica and Niemela 2002).

**SA description**: The method is applied to a final version of the SA but prior to the detailed design. The description of the SA should be in a form that is easily understandable by all stakeholders, as shown in Figure 1.1.

![Software architecture diagram](image-url)

**Figure 1.1  Software architecture**

In the next section we will discuss about SAAMCS.
1.5.2 SAAM Founded on Complex Scenarios (SAAMCS)

SAAMCS considers that the complexity of scenarios is the most important factor for risk assessment (Colquitt and Leaney 2002; Kazman et al 1996). The SAAMCS contributions for extending SAAM are, on one hand, directed to the way of looking for the scenarios and, on the other, to where their impact is evaluated, as shown in Figure 1.2.

![Diagram of SAAMCS inputs and activities]

**Figure 1.2 Inputs and activities of SAAMCS**

**Specific goal:** Risk assessment represents the only goal of SAAMCS.

**Evaluation technique:** SAAMCS is looking for scenarios that are possibly complex to realize. Based on the initiator of the scenario, SA description, and version conflicts, a list of classes of scenarios that are complicated to implement is provided.
The quality attributes: Flexibility represents the quality attribute analyzed by SAAMCS.

The stakeholders' involvement: The method appreciates stakeholders' involvement and identifies the important role of the initiator of a scenario. The initiator is the organizational unit that has most interest in the implementation of that scenario. SAAMCS is applied to the final version of the architecture, which is described in sufficient detail. In this method, the idea of the systems within a domain, not being isolated but instead integrated within an environment, is advanced (Dobrica et al 2002). In the next section we will discuss about ESAAMI.

1.5.3 Extending SAAM by Integration in the Domain (ESAAMI)

ESAAMI is a combination of analytical and reuse concepts and is achieved by integrating the SAAM in the domain-specific and reuse-based development process. ESAAMI is similar to SAAM with records to the evaluation technique, the quality attributes, the stakeholder's involvement and SA description.

Specific goal: SAAM applied in an architectural-centric development process, considers only the problem description, requirements statement and architecture description, as shown in Figure 1.3.

SA description: A reusable SA to be deployed in the new system is selected in the first step of ESAAMI. It has to be ensured that SA provides an adequate basis for the system to meet its requirements (Colquitt and Leaney 2002).
Reusability of the existing knowledge base. ESAAMI proposes packages of analysis templates which represent the essential features of the domain (Dobrica et al 2002).

In the next section we will discuss about SAAMER.

1.5.4 Software Architecture Analysis Method for Evolution and Reusability (SAAMER)

SAAMER better suggests how system could support each of the quality objectives or the risk levels for evolution or how to reuse it. SAAMER provides a framework of activities that are useful for the analysis process. This framework consists of four activities like gathering information about stakeholders, modeling usable artifacts, analysis and evaluation.

Specific goal: From the point of view of two particular quality attributes, evolution, and reusability, SAAM is extended to SAAMER (Colquitt and Leaney 2002; Iannino 1994; Kazman et al 2000).
**The evaluation technique:** Scenarios are the main drivers for evaluating various areas of SA. They describe an important functionality that the system must support, or recognize where it may need to be changed over time.

**The quality attributes:** Evolution and reusability are considered. Evolution integrates new quality objectives (maintainability and modifiability) obtained from domain experts (IEEE Standard 1061-1992).

**Stakeholders' involvement** is similar to that of SAAM. Additionally, two kinds of sources of information namely, the required changes and the domain experts' experiences, are considered.

**SA description:** SAAMER considers the following architectural views as critical: static, map, dynamic, and resource (Dobrica et al 2002).

In the next section we will discuss about ATAM.

1.5.5 **The Architecture Trade-Off Analysis Method (ATAM)**

ATAM has grown out of the work on architectural analysis of individual quality attributes: modifiability, performance, availability, and security as shown in Figures 1.4 and 1.5.

It was considered a spiral model of designs in 1998. This method is divided into four main area of activities. These are the gathering of scenarios and requirements, architectural vies and scenario realization, attribute model building and analysis and trade-offs (Kazman et al 2000).
2. Collect Requirements/Constraints/Environment

3. Describe Architectural Views

4. Realise Scenarios

5. Attribute – Specific Analyses (best individual Theoretical models)

6. Identify Sensitivities

7. Identify Trade-offs

PHASE I
Scenarios and requirements gathering

PHASE II
Architectural views and Scenario realization

PHASE III
Attribute model building and analyses

PHASE IV
Trade-offs

Figure 1.4 ATAM phases

Figure 1.5 ATAM activities that consider scenarios
Specific goals: The objective of ATAM is to provide a principal way of understanding an SA’s capability with respect to multiple competing quality attributes (Colquitt and Leaney 2002; Gina Kingston 2000).

SA description: The space of architecture is constrained by legacy systems, interoperability, and failure of the previous projects.

The quality attributes: Multiple competing quality attributes are analyzed by ATAM (Dobrica et al 2002). SBAR is discussed in the next section.

1.5.6 Scenario-Based Architecture Reengineering (SBAR)

The contribution of this method is not only in the architecture design but also in the scenario-based evaluation of the software qualities of a detailed architecture of a system (Colquitt and Leaney 2002; Schmidt; Kazman et al 2000), as shown in Figure 1.6.
Specific goal: SBAR estimates the potential of the designed architecture to reach the software quality requirements.

The included evaluation techniques: Four different techniques for assessing quality attributes are identified: scenarios, simulation, mathematical modeling, and experience-based reasoning.

Stakeholders' involvement: SBAR does not require the involvement of many stakeholders. The evaluator is the designer of the SA.

SA description: The Peculiarity of this method is that, for assessing the architecture of the existing system, the system itself can be used. SBAR uses a detailed design of SA (Dobrica et al 2002). In the next section we will discuss about ALPSM.

1.5.7 Architecture Level Prediction of Software Maintenance (ALPSM)

It involves in the process of the identification of categories of maintenance tasks, synthesis scenarios, estimation of the size of all elements, scripting the scenarios and calculation of the predicted maintenance effort.

Specific goal: ALPSM analyzes the maintainability of a software system by looking at the impact of scenarios at the SA level (Colquitt and Leaney 2002).

The evaluation technique: ALPSM defines a maintenance profile, like a set of change scenarios representing perfective and adaptive maintenance tasks.
Stakeholder's involvement: Only the designer is involved in the method activities.

SA description: ALPSM is applied to the final version of SA.

Method's activities: The method has a number of inputs: the requirements statement, the description of the architecture, expertise from software engineers and possibly historical maintenance data, as shown in Figure 1.7.

Figure 1.7 ALPSM inputs and result

The evaluation technique: SAEM tries to define the metrics of quality based on the goal-question-metric (GQM) technique (Dobrica et al 2002). SAEM is discussed in the next section.
1.5.8 A Software Architecture Evaluation Model (SAEM)

A quality model based on standard software quality assessment process is chosen and a conceptual framework that relates quality requirements, metrics and internal attributes of the SA and the final system is proposed (Colquitt and Leaney 2002).

Specific goal of the method: SAEM establishes the basis for the SA quality evaluation and prediction of the final system quality.

The evaluation technique: SAEM tries to define the metrics of quality based on the goal-question-metric (GQM) technique.

The reusability of the existing knowledge base SBAR does not consider this issue. SBAR has been validated for a measurement software system (Dobrica et al 2002).

1.6 SUMMARY AND CONCLUSION

This chapter threw light on the salient aspects of the basic concepts of software architecture and its relevance as a separate discipline. Detailed explanation was presented about architecture, quality attributes, and important analysis methods for software architecture. In the subsequent chapter we will discuss about the impact of software architecture and analysis methods using the literature survey.