Chapter 2

STUDY AND ANALYSIS OF REQUIREMENTS ENGINEERING APPROACHES

"The main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects."

-- Edsger W. Dijkstra

The critical factor in the success or failure of software engineering projects is requirements. Lacks of user inputs, incomplete requirements, changing requirements etc. are the major reasons why so many software projects fail [SOM 97b; NUS 00; DUS 02]. Also, the primary measure to success of a software system is the degree to which it meets the purpose for which it was intended. Broadly speaking, software systems' requirements engineering is the process of discovering that purpose.

Existing techniques to Requirements Engineering provide good support for identification and treatment of requirements. However, these requirements engineering approaches do not explicitly support, in an integrated fashion, all kinds of requirements, particularly crosscutting concerns. The result is that the implementation of these crosscutting concerns are spread among several modules, producing scattered and tangled representations that are invasive to implement, and therefore difficult to understand, maintain and evolve. AOSD aims at alleviating the problems of scattering and tangling by providing means for their systematic identification, separation, representation and composition.

The aim of this chapter is to set the context and background for this research work, and to help readers to understand some core key terms. It is organized into three sections. The first section illustrates traditional requirements engineering approaches and related terms. The following section explains aspect-oriented requirements engineering, and finally, the chapter finishes with a summary.
2.1 Requirements Engineering

Ambiguous, missing, or incomplete requirements contribute to the high failure rate of systems development projects [TSG 95]. If the requirements are not done right, the project will probably fail, no matter what is done in subsequent phases. This usually results from incomplete requirements gathering or inaccurate communication between the analysis and design phases. Improving these two aspects of the requirements engineering process thus is crucial to the overall improvement of software development project success.

Incomplete or defective requirements result in inaccurate product descriptions and erroneous cost and schedule estimates, which in turn deliver an unusable product. What is not asked for will not be produced, and what is asked for will be produced even if it's not needed. Table 2.1 shows the relative costs of fixing software requirements defects in different phases of the project. Whatever it may cost to do things right in the requirements phase, it will be 3 to 1000 times more costly to fix later if not done right.

Table 2.1: Relative Cost of Fixing Requirements Errors [GAU 89]

<table>
<thead>
<tr>
<th>Phases in Which Fixed</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
</tr>
<tr>
<td>Design</td>
<td>3 – 6</td>
</tr>
<tr>
<td>Coding</td>
<td>10</td>
</tr>
<tr>
<td>Development Testing</td>
<td>15 – 40</td>
</tr>
<tr>
<td>Acceptance Testing</td>
<td>30 – 70</td>
</tr>
<tr>
<td>Operation</td>
<td>40 – 1000</td>
</tr>
</tbody>
</table>

In order to obtain high-quality products along with higher productivity, it is required to carefully analyze, model, specify and manage system requirements. This would not only simplify system design and implementation but also reduce the number of defects that are identified later in the implementation stage. Requirements engineering is introduced to address such issues early in the development process. A well-
established requirement engineering process ensures that system requirements are properly elicited, analyzed, documented, verified and managed. This section presents an overview of requirements and requirements engineering.

2.1.1 Requirement

A lot of definitions exist in the literature for requirements. For example,

"A condition or capability needed by a user to solve a problem or achieve an objective" [IEE 90].

"A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents" [IEE 90].

A requirement is "a feature that the system must have or a constraint that it must satisfy to be accepted by the client" [BRU 07].

Requirements for the system are "the descriptions of the services and constraints ..." [SOM 97b].

Clearly, there is no universal definition of what a requirement is. There are, however, different kinds of requirements.

Wiegers [WIE 03] classifies requirements into following types:

i. *Business requirements* describe high-level goals of the organization or client who requests the software.

ii. *User requirements* describe required activities needed to achieve the users’ goals.

iii. *System requirements* describe the top-level product requirements that can include software and hardware.

iv. *Functional requirements* describe the software functionality from the developer’s perspective.

v. *Non-functional requirements* describe global properties that affect the whole, or a large part of, the system; they are quality characteristics such as performance, usability and efficiency.
Sommerville [SOM 04] classifies requirements into two broad categories as user requirements and system requirements. Furthermore, system requirements are classified as functional or non-functional or domain requirements. They are defined as follows:

i. “Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.”

ii. “Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, standards, etc.”

iii. “Domain requirements come from the application domain of the system and that reflect characteristics of that domain. They may be functional or non-functional.”

In the last few years, the requirements engineer community has been dedicating some attention to non-functional requirements that are defined as global requirements [CHU 99]. To this community, non-functional requirements are also quality attributes and constraints, and play a critical role during software development, since they are very difficult to test and they are usually evaluated subjectively.

Based on the classifications given above, we can synthesise by suggesting that there is a consensus around the existence of two broad classifications of requirements, functional and non-functional requirements. Functional requirements describe the interaction between the system and its environment independent of its implementation. The environment includes the user and any other system that interacts with the system. Non-functional requirements describe aspects of the systems that are not directly related to the functional behaviors of the system. It includes a broad variety of requirements that apply to many different aspects of the system, such as usability, reliability, performance (i.e. response time, throughput, accuracy, availability etc.), and supportability. We also call these requirements as quality requirements.
2.1.2 Requirements Engineering

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended. Broadly speaking, software systems requirements engineering (RE) is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation.

Typical textbook definitions of engineering refer to the creation of cost-effective solutions to practical problems by applying scientific knowledge. Therefore, the use of the term engineering in RE serves as a reminder that RE is an important part of an engineering process, being the part concerned with anchoring development activities to a real-world problem, so that the appropriateness and cost-effectiveness of the solution can then be analyzed. It also refers to the idea that specifications themselves need to be engineered, and RE represents a series of engineering decisions that lead from recognition of a problem to be solved to a detailed specification of that problem.

RE is often regarded as a front-end activity in the software systems development process. This is generally true, although it is usually also the case that requirements change during development and evolve after a system has been in operation for some time. Therefore, RE plays an important role in the management of change in software development. Nevertheless, the bulk of the effort of RE does occur early in the lifetime of a project, motivated by the evidence that requirements errors, such as misunderstood or omitted requirements, are more expensive to fix later in project lifecycles [NAK 91].

Zave defines requirements engineering as:

"Requirements engineering is the branch of software engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families" [ZAV 1997].

Wiegers defines requirements engineering as:
"Requirements engineering is a discipline based on requirements development and requirements management. Requirements development is related to elicitation, analysis, specification and validation. The goals of these activities are gathering, evaluating and documenting the requirements for the software. Requirements management goal is to establish and maintain an agreement with the customer on the requirements for the software" [WIE 03].

Thayer et al. defines it as:

"Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analysing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system" [THA 97].

Roger S. Pressman defines requirements engineering as:

"Requirements engineering helps software engineers to better understand the problem they will work to solve. It encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end-users will interact with the software. Requirements engineering begins with inception – a task that defines the scope and the nature of the problem to be solved. It moves onward to elicitation – a task that helps the customer to define what is required, and then elaboration – where basic requirements are refined and modified. As the customer defines the problem, negotiation occurs what are the priorities, what is essential? Finally, the problem is specified and reviewed or validated to ensure that the customers’ and your understanding of the problem coincide. At the same time, requirements management activities are needed to guarantee trace and changes of requirements" [PRE 05].

The requirements engineering process can be described in four generic activities, as depicted in Figure 2.1:

i. Requirements elicitation and analysis
ii. Requirements specification
iii. Requirements validation
iv. Requirements management
Figure 2.1: Requirements Engineering Process, taken from [SOM 04]

Requirements elicitation and analysis activity is divided into two sub-activities as requirements elicitation and analysis. During requirements elicitation, requirements engineers might use information-gathering techniques to obtain and capture derived requirements that are a logical consequence of what the users and clients requested. This activity results in a description of the system in terms of actors and use cases. During analysis, developers aim to produce system model that is correct, complete, consistent, and unambiguous.

Requirements specification activity is to obtain the descriptions and representations of all gathered requirements. In the context of software systems, the term specification means different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. The requirements engineer might use standard templates, such as proposed by [JAC 92; SOM 97a], graphical models, such as proposed by [JAC 92; VAN 01], mathematical equations and prototypes [JAC 92; VAN 01]. They argue that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the
best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments [PRE 05].

The requirements validation activity is to ensure that the "requirements and models elicited provide an accurate account of stakeholder requirements" [NUS 00]. It examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product. To achieve this, the requirements engineer can use review techniques, such as program inspections [GIL 93]. Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions.

Conflict management, including identification and resolution of conflicts between requirements or stakeholders’, can be included in the requirements validation activity [KAR 97; BOE 98; VAN 98; ALV 03].

Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds. It is to manage the requirements of the project’s products and product components and to identify inconsistencies among requirements, project’s plans, and work products. The project takes appropriate steps to ensure that the agreed-on set of requirements are managed to support the planning and execution needs of the project. When a project receives requirements from an approved requirements provider, the requirements are reviewed with the requirements provider to resolve issues and prevent misunderstanding before the requirements are incorporated into the project’s plans. Once the requirements provider and the requirements receiver reach an agreement, commitment to the requirements is obtained from the project participants. The project manages changes to the requirements as they evolve and identifies any inconsistencies that occur among the plans, work products, and requirements. Part of the management of requirements is to document requirements changes and rationale and to maintain bidirectional traceability between source requirements and all product and product component requirements.
2.1.3 Traditional Requirements Engineering Approaches

I. Use Case Approaches

Use case approach for requirements elicitation was firstly introduced by Jacobson, which has emerged as the most popular means of eliciting system stakeholders or actors' needs and capture requirements in industry [JAC 92]. Use cases are general sequences of events that describe all the possible actions between an actor and the system for a given piece of functionality. An actor is an external entity, a person, another system or anything that interacts with the system under study. Actors initiate a use case to access system functionality [BRU 05]. The use case can then initiate another use case to gather more information from actors. The exchange of information between actors and use cases is said to communicate.

The first step for requirement elicitation using this approach is the identification of actors. Actors are role abstractions and do not necessarily map to persons directly because the same person can fill different roles at different time. However, they access considerably different functionality. Hence, two such different roles are modelled as two different actors. It is quite easy to identify the actors when the system is deployed into an existing organization such as a company. Most of the actors in such situations exist before the system is developed as they correspond to roles in that organization.

Once the actors are identified, the next activity for requirements elicitation is to determine the functionality that will be accessible to each actor. There are several techniques to achieve this, from free text to some formal technique. Scenarios are, perhaps, the most common and simpler technique. A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor. Scenarios are not intended to replace use cases, as they focus on specific instances and concrete events. However, they enhance requirements elicitation by providing a tool that is understandable to users and clients.

Developers' emphasis during identification of actors and scenarios is to understand the application domain. Once the actors and scenarios are identified, the next activity is to formalize the identified scenarios into use cases.
A use case specifies all possible scenarios for a given piece of functionality, where a scenario is an instance of that use case. It represents a complete flow of events through the system i.e. describes a series of related interactions that result from its initiation. Generalizing scenarios and identifying the high-level use cases that the system must support enables developers to define the scope of the system. Writing use cases is a craft. An analyst learns to write better use cases with experience. Consequently, different analysts tend to develop different types, which can make it difficult to produce a consistent requirement specification. To address the issue of ensuring consistent requirements specification, analysts may agree to consult a use case writing guide from [COC 01].

The next activity is to identify relationships among actors and use cases. We use include relationship to minimize redundancy among use cases by factoring out common behaviours between use cases. So, instead of describing such behaviours repeatedly, we only need to define them once. The extend relationship allows us to separate exceptional and common flow of events. In other words, to add behaviour to a base use case at a set of extension points without changing the base use case. The added behaviour is specified in the extending use case which will be inserted into the base use case. The set of use cases, actors and relationships among them composes the use case model e.g. depicted use case model as figure 2.2.

This approach becomes a fundamental element of the Unified Modelling Language (UML) notation [JAC 98]. UML is a general-purpose modelling language that resulted from the unification of Object Modelling Technique (OMT) [RUM 91], Booch [BOO 94], and Object-Oriented Software Engineering (OOSE) [JAC 92]. It includes a graphical notation used to create an abstract model of a system that drive the software development process and was proposed by Object Management Group (OMG). In UML, actors are represented as stick figures and use cases are represented by ellipses. Lines indicate which actors are involved in which use cases. Figure 2.2 illustrates the notation used for use case models, as proposed by the UML language. The example is based on the First Responder Interactive Emergency Navigational Database (FRIEND).
It is a common practice to document a use case fully through a template, where primary and alternative scenarios are added as fundamental pieces of the document. There are a number of schemes, however, but typical sections are:

- **Use case name** provides a unique identifier for the use case and should be a verb phrase denoting what the actor is trying to accomplish.

- **Participating actors** often focusing on who initiates each use case, enables developers to clarify the roles of different users.

- **Flow of events** enables developers and clients to discuss the interaction between actors and system which results in many decisions about the
boundary of the system, that is, about deciding which actions are accomplished by the actor and which are accomplished by the system.

- **Entry conditions** are used to convey any conditions that must be true when a user initiates a use case.

- **Exit conditions** summarize the state of affairs after the scenario is complete.

- **Quality requirements** enable developers to elicit non-functional requirements.

<table>
<thead>
<tr>
<th>Use case name</th>
<th>PurchaseOneWayTicket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actors</td>
<td>Initiated by Traveler</td>
</tr>
<tr>
<td>Flow of events</td>
<td>1. The Traveler selects the zone in which the destination station is located.</td>
</tr>
<tr>
<td></td>
<td>2. The TicketDistributor displays the price of the ticket.</td>
</tr>
<tr>
<td></td>
<td>3. The Traveler inserts an amount of money that is at least as much as the price of the ticket.</td>
</tr>
<tr>
<td></td>
<td>4. The TicketDistributor issues the specified ticket to the Traveler and returns any change.</td>
</tr>
<tr>
<td>Entry condition</td>
<td>The Traveler stands in front of the TicketDistributor, which may be located at the station of origin or at other station.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>The Traveler holds a valid ticket and any excess change.</td>
</tr>
<tr>
<td>Quality requirements</td>
<td>If the transaction is not completed after one minute of inactivity, the TicketDistributor returns all inserted change.</td>
</tr>
</tbody>
</table>

**Figure 2.3: Example of a Use Case, taken from [BRU 05]**

22
Use case approaches have a lot of benefits, a few as follows:

- It is generally regarded as an excellent technique for elicitation the functional requirements of a system.
- It can improve the system robustness by capturing many alternatives.
- It enhances requirements elicitation by providing scenarios that are understandable to users and clients.
- It enables developers to define the scope of the system by generalizing scenarios and identifying the high-level use cases that the system must support.
- Use cases can be recorded using UML and maintained within widely available computer-aided software engineering tools.

Despite many benefits the use case approaches have several limitations as follows:

- Use cases generally regarded as an excellent technique for elicitation the functional requirements of a system but they are not well suited to capture non-functional requirements.
- Use case approaches lacks on conflicts resolution.

II. Goal-oriented Approaches

Goals have long been recognized to be essential components involved in the requirements engineering (RE) process. A goal is an objective the system under consideration should achieve [VAN 01]. Goal formulations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones, and bounded by the subject matter [JAC 95; ZAV 97a]. Goals may be formulated at different levels of abstraction, ranging from high-level, strategic concerns to low-level, technical concerns. Goals also cover different types of concerns such as functional concerns associated with the services to be provided, and non-functional concerns associated with quality of service e.g. safety, security, accuracy, performance, and so forth.

Mylopoulos et al. [MYL 92] proposed a comprehensive framework for representing and using non-functional requirements during the development process, which consists of five major components as:
i. A set of goals for representing non-functional requirements, design decisions, and arguments in support of or against other goals. The space of goals includes three mutually exclusive classes, namely, non-functional requirements goals (NFR goals), satisfying goals, and argumentation goals. Non-functional Requirements Goals range over the different categories of requirements, including accuracy, security, development, operating or hardware costs, and performance. Satisfying Goals range over different categories of design decisions that might be adopted in order to satisfy one or more non-functional requirements goals. The parameters associated with each sort, again, depend on the nature of the corresponding satisfying goal. Argumentation Goals always have the sort Claim, with subsorts Formal Claim, and Informal Claim, representing formally or informally stated evidence or counter-evidence for other goals or goal refinements.

ii. A set of link types for relating goals or goal relationships (hereafter links) to other goals. Links may relate a parent goal to one or several of its offspring. In fact, links may also be used to relate other links to argumentation goals, to indicate that an argument offers positive or negative support for a particular refinement of a goal. Thus, links too need to be satisfied either through a formal refinement process or through arguments provided by the designer.

iii. A set of generic methods for refining goals into other goals may be refined by the designer, who is then responsible for satisfying not only the goal’s offspring but also the refinement itself represented as a link. Alternatively, the framework provides goal refinement methods (methods for short) which represent generic procedures for refining a goal into one or more offspring. There are three types of goal refinement methods, corresponding to the three types of goals introduced earlier which are as goal decomposition methods, goal satisfying methods, and argumentations methods.

iv. A collection of correlation rules for inferring potential interactions among goals. The non-functional requirements set down for a particular system may be contradictory. For instance, having built-in procedures for validating or auditing the data managed by the information system in general requires additional manpower thereby interfering with the operating cost requirement,
Operating Cost [manpower]. Guidance is needed by the designer in discovering such implicit relationships and in selecting the satisfying goals that best meet a set of given NFR goals. This is achieved either through external input by the designer herself or through the representation of generic interactions between goals through correlation rules.

v. And finally, a labelling procedure which determines the degree to which any given non-functional requirement is being addressed by a set of design decisions. Labeling procedure determines the status of each node on the partially constructed goal graph through the assignment of a label. A link or node of the graph labeled satisfied (S) if it is satisfiable and not deniable, denied (D) if it is deniable but not satisfiable, conflicting (C) if it is both satisfiable and deniable, and undetermined (U) if it is neither.

This framework consists of five basic components which provide for the representation of non-functional requirements in terms of interrelated goals. Such goals can be refined through refinement methods and can be evaluated in order to determine the degree to which a set of non-functional requirements is supported by a particular design.

There are a lot of types of goals based on several classification exist in literature. They used to define heuristics for goal acquisition, goal refinement, requirements derivation, and semi-formal consistency/completeness checking [DAR 93; SUT 93; ANT 98; CHU 00; ANT01], or to retrieve goal specifications in the context of specification reuse [MAS 97].

The first classification often made in the literature is between functional goals and non-functional goals. Functional goals underlie services that the system is expected to deliver whereas non-functional goals refer to expected system qualities such as security, safety, performance, usability, flexibility, customizability, interoperability, and so forth [KEL 90]. Functional goals can be specialized as satisfaction goals that concerned with satisfying agent requests; and information goals that concerned with keeping such agents informed about object states [DAR 93]. Non-functional goals can be specialized in a similar way. For example, accuracy goals are non-functional goals requiring the state of software objects to accurately reflect the state of the corresponding monitored/controlled objects in the environment [MYL 92; DAR 93] -
such goals are often overlooked in the RE process; their violation may be responsible for major failures [VAN 00a]. Performance goals are specialized into time and space performance goals, the former being specialized into response time and throughput goals [NIX 93]. Security goals are specialized into confidentiality, integrity and availability goals [AMO 94]; the latter can be specialized in turn until reaching domain-specific security goals. A rich taxonomy for non-functional goals can be found in [CHU 99].

The second classification often made in the literature is between soft goals which represent non-functional requirements; and hard goals whose satisfaction can be established through verification techniques [DAR 93; DAR96]. The soft goals are systematically decomposed into more specific soft goals or sub-goals for comparing alternative goal refinements and choosing one that contributes the “best” to them.

The second classification often made in the literature is based on types of temporal behaviour prescribed by the goal [DAR 93]. There are three types of such goals as Achieve goals, maintain goals, and optimize goals. Archive goals generate system behaviours require some target property to be eventually satisfied in some future state. Maintain goals restrict behaviours, in that they require some target property to be permanently satisfied in every future state. Optimize goals compare behaviours to favour those which better ensure some soft target property.

The third classification is proposed by [SUT 93] according to desired system states, that is, positive, negative, alternative, feedback, or exception-repair; and to goal level, which are policy level, functional level, domain level.

Another classification proposed by [ANT 94] is between objective goals and adverbial goals. Objective goals refer to objects in the system whereas adverbial goals refer to ways of achieving objective goals.

Besides their types, goals can also be essentially characterized by their attributes such as name, specification, and priority. Other goal attributes that have been proposed include goal utility and feasibility [ROB 89].

Further, goal links are used to relate goals with each other and with other elements of requirements models. Such links form the basis for defining goal structures.
The Goal-based approaches focus on modeling variability from the perspective of the problem domain. They effectively capture the alternative ways by which the stakeholders achieve their goals, where top-level goals represent very high-level functional (called hard goal) and non-functional (called soft goal) requirements. These alternative ways are represented and refined with the AND/OR-decompositions of the goal graphs, as shown in Figure 2.4.

Figure 2.4: AND / OR Goal Decomposition Graph, taken from [LIA 06]

The Goal-based Variability Acquisition and Analysis approach [LIA 06] distinguishes a set of possible variability types in goal acquisition. Based on this set, the approach aims to achieve completeness in goal variability modeling also taking into account the context of the problem under modeling. [LIA 06] uses Filmore's concept of frame elements [FIL 68], which states that each verb in a language has a set of associated semantic slots or cases. By looking into a linguistic data for the given problem, this approach collects the set of elements that appear in each of the slots for each verb. It also looks at the adjuncts, such as location, temporal duration, etc. for each verb. In addition, this approach also models the "background variability" for agents, location and objects, thus capturing the properties of a specific context of a problem. This information is then used to construct a "goal-frame" for each goal for the given problem space. Where the relevant verbs with all their associated semantic slots and adjuncts are
represented in an OR-decomposition goal tree as alternatives for achieving a given goal.

Why goals are so important in the requirements engineering process [VAN 01]?

i. The main objective of requirements engineering process is to achieve requirements completeness. Goals provide a precise criterion for sufficient completeness of a requirements specification; the specification is complete with respect to a set of goals if all the goals can be proved to be achieved from the specification and the properties known about the domain considered [YUE 87].

ii. Another important objective of requirements engineering process is to avoid irrelevant requirements. Goals provide a precise criterion for requirements pertinence; a requirement is pertinent with respect to a set of goals in the domain considered if its specification is used in the proof of one goal at least [YUE 87].

iii. Explaining requirements to stakeholders is another important issue. Goals provide the rationale for requirements change, so a requirement appears because of some underlying goal which provides a base for it. This is motivated by the more dynamic business and organizational environments, where systems are increasingly used to fundamentally change business processes.

iv. Goals refinement provide a natural mechanism for structuring complex requirements documents for increased readability and provide traceability links from high-level strategic objectives to low-level requirements.

v. Goals provide techniques to manage conflicts among multiple viewpoints. Goals have been recognized to provide the roots for detecting conflicts among requirements and for resolving them eventually.

vi. Requirements engineers are faced with many alternatives to be considered during the requirements elaboration process. Our extensive experience revealed that alternative goal refinements provide the right level of abstraction at which decision makers can be involved for validating choices being made or
suggesting other alternatives overlooked so far. Alternative goal refinements allow alternative system proposals to be explored [VAN 00b].

vii. Last but not least, goals drive the identification of requirements to support them; they have been shown to be among the basic driving forces, together with scenarios, for a systematic requirements elaboration process.

The Non-Functional Requirements Framework, NFR, proposed by [CHU 00], i* framework, proposed by [YUE 97], and Knowledge Acquisition in Automated Specification, KAOS, proposed by [DAR 93] are representative and complementary goal-based approaches.

III. Viewpoint-oriented Approaches

Sommerville et al. [SOM 97a] proposed a multi-perspective approach to requirements engineering called viewpoint-oriented requirements engineering. The approach distinguishes that all system requirements information cannot be elicited by single perspective rather they are properly elicited and organized by considering the system from multiple perspectives or viewpoints. A viewpoint is an encapsulation of partial information about a system's requirements which may be used to structure the process of requirements elicitation and to structure the requirements specification. An analyst collects all relevant information for a system from different perspectives or viewpoints.

A system analyst collects and analyses the requirements for a system from a number of different sources and recognises that all these sources have different, often equally valid perspectives. Information from different viewpoints must be integrated to form the final system specification. Recognising these perspectives and reconciling differences between them is essential if the analysis is to be valid. Viewpoint-oriented approaches are simply a means of formalising this intuitive multi-perspective analysis.

The main arguments in favour of a viewpoint-oriented approach to requirements engineering are [SOM 97a]:

i. Viewpoints organise system requirements from different types of users and stakeholders.
ii. Viewpoints are used to collect and classify different types of information, for example, information about the application domain, information about the system's environment and engineering information about the system's development.

iii. Viewpoints may be used as a means for structuring the process of requirements elicitation.

iv. Viewpoints may be used to identify conflicts between different requirements.

In the viewpoint-based methods which have been developed, two different kinds of viewpoint have been proposed [SOM 97a]:

i. *Viewpoints associated with system stakeholders.* Informally, a system stakeholder is anyone who is, directly or indirectly, affected by the existence of a system. Hence stakeholders may be end-users of a system, managers of organisations where systems are installed, other human and computer-based systems in an organisation, external entities who have some kind of interest in the system (e.g., regulatory bodies, customers of an organisation which has installed the system) and engineers involved in the design, development and maintenance of the system.

ii. *Viewpoints associated with organisational and domain knowledge.* Organisational and domain knowledge is knowledge which constrains the system requirements. The constraints may be physical (e.g., network performance), organisational (e.g., incompatible hardware used in different divisions of a company), human (e.g., average operator error rate) or may reflect local, national or international laws, regulations and standards. This type of viewpoint cannot be associated with a single class of stakeholder but includes information collected from many different sources (people, documents, other systems, etc.).

A lot of models exist in literature for viewpoints-oriented approaches to requirements engineering, but most of them adapt to the general model shown as figure 2.5.
The phases of this generic requirements engineering process are:

1. **Requirements elicitation.** Requirements engineers consult various different sources to understand the problem and the application domain. They might use information-gathering techniques to obtain and capture derived requirements that are a logical consequence of what the users and clients requested. These requirements may not be complete and may be expressed in a vague and unstructured way.

2. **Requirements analysis.** The requirements collected during the elicitation phase are integrated and analysed. Here, the developers aim to produce a system model that is correct, complete, consistent, and unambiguous. Usually, this results in the identification of missing requirements, inconsistencies and requirements conflicts.

3. **Requirements negotiation.** The system stakeholders negotiate to agree on a set of requirements for the system. Generally, there are a greater number of desirable requirements than can be implemented so decisions have to be made at this stage about leaving out requirements and modifying requirements to result in a lower-cost system.

4. **Requirements specification.** The set of agreed requirements is documented. The output from this process may be a user requirements document, a system specification or both of these. A user requirements document is usually a natural language document where the system requirements are set out in a
form understandable by customers and end-users of the system. A system specification is a more detailed description of what services the system should provide and the constraints on its development and operation.

PREView (Process and requirements engineering viewpoints) is a viewpoint-oriented approach which adapts existing requirements engineering process to improve the processes of requirements discovery, analysis and negotiation rather than system specification. This approach uses the concept of viewpoints, where a viewpoint represents an entity that encapsulates some but not all information about a system's requirements [SOM 97a]. All information about the system is gathered by integrating the requirements derived from different viewpoints.

A PREView viewpoint includes the following information:

i. The viewpoint name may reflect a role in the organisation or a part of the system or process to which the analysis is restricted. This is used for identifying, referring and reflecting the focus of the viewpoint.

ii. The viewpoint focus is an explicit statement of the perspective adopted by that viewpoint. The focus can be a statement of the parts of the problem, the system or the process with which the viewpoint is associated, a statement of the role of the viewpoint sources, a statement of an organisational function such as management, health and safety, engineering or, perhaps, a mixture of these. Focus is the defining characteristic of a viewpoint.

iii. The viewpoint concerns are an innovative feature of PREView and have been introduced so that the requirements can be explicitly linked to organisational goals and priorities. Concerns correspond to high-level strategic objectives for the system. They are used to ensure that the requirements for the system are consistent with the business goals of the procuring organisation.

iv. The viewpoint sources are an explicit record of where the information associated with a viewpoint has come from. Viewpoint sources are not simply individuals or roles in an organisation. They may also include manuals of operating procedure, international, national or organisational standards, domain knowledge, experience data such as incident descriptions, other requirements placed on the system etc.
v. The viewpoint requirements are the set of requirements arising from analysis of the system from the viewpoint’s focus. The requirements may be expressed in terms of system functionality, user needs or constraints arising from application domain or organisational considerations.

vi. The viewpoint history records changes to the viewpoint as an aid to traceability. It includes changes to the focus, the sources and the requirements encapsulated in the viewpoint.

The PREView process consists of four main steps as illustrated in figure 2.6:

i. Requirements Discovery
ii. Requirements Analysis
iii. Requirements Negotiation
iv. Requirements Definition

The process starts with requirements discovery, which is broken down into four activities namely identify concerns, elaborate concerns as external requirements and questions, identify viewpoints, and discovery of the requirements for each viewpoint. The next step is requirements analysis, which is to identify those requirements which are inconsistent with concern questions, external requirements, or other viewpoints’ requirements. The objective is to discover internal viewpoint conflicts and external inconsistencies where requirements from different viewpoints are in conflict. To accomplish this, PREView uses decision tables, based on Quality Function Deployment proposed by Zultner, to cross-check requirements against overlapping, conflicting, or being independent of external requirements [ZUL 92]. The result of this activity is the input to the requirements negotiation activity where the inconsistencies should be resolved. So, the inputs to the requirements negotiation step are the sets of conflicting and overlapping requirements. PREView does not prescribe how conflicts are resolved or how overlapping requirements are rationalised as this will necessitate trade-offs and negotiation. Normally, we expect the people, who are viewpoint sources to meet together, discuss the requirements and agree on priorities. The viewpoint concern, focus and source information associated with each requirement should be used to inform the negotiation process and provide a context for resolution. The results of the process will typically be a set of changed
requirements. These changes should be fed back to the requirements discovery phase and should be recorded in the relevant viewpoints' histories.

Figure 2.6: The PREView Process taken, from [SOM 97a]

The following practical problems which make viewpoint-oriented approaches difficult to use for non-trivial projects:

- Inflexible viewpoint models. If the viewpoint model is too restrictive in its definition of a viewpoint, it will not encompass all of the possible stakeholder and domain viewpoints which may be required. Many requirements problems are human, social and organisational problems. Viewpoints need to be able to reflect these positions and not just technical expressions of system requirements.
o **Fixed notations for requirements definition.** Requirements sources often will not have time to express requirements in anything but their normal working notations. They are unlikely to be able to translate these easily into some different modelling notation. Automated or semi-automated conflict analysis of requirements is, in our view, impractical.

o **Limited support for requirements evolution.** Not only is the system's organisational, economic and political environment changing as the requirements are developed, the better understanding of the system which emerges during the RE process causes requirements to evolve. Viewpoint-oriented approaches must recognise this and must not, for example, require consistency at all times.

o **Limited support for requirements negotiation.** The process of establishing a final set of requirements for a system normally involves stakeholders negotiating changes and compromises between conflicting requirements. Some means of discovering conflicting and overlapping requirements is helpful here. However, automated conflict resolution may be counter-productive as it does not recognise the non-technical factors which influence the requirements negotiation activity.

o **No industrial-strength tool support.** Viewpoint-oriented approaches tend to generate a large amount of information which must be managed and this obviously requires tool support of some kind. This tool support must be available on platforms used by application developers, must be of good quality, must be compatible with other tools which have already been purchased and must be reasonably cheap. It is extremely difficult to meet these support requirements for any new method because the costs and risks of tool development are so high.

o **No recognition of the problems of non-functional requirements.** In some applications, non-functional requirements are more critical than functional requirements. For example, in many control systems, there is an inflexible requirement to maintain the safety of the system whereas the functional capabilities of the control system are usually negotiable. Requirements engineering methods don't handle non-functional requirements very well,
especially when these are ‘system-level’ requirements rather than requirements associated with a particular function or class of functions.

- **Incompatibility with other software engineering methods.** Organisations which might benefit from the use of viewpoints for requirements engineering generally have existing, defined, design processes. Any requirements engineering method must be compatible with existing design methods.

### 2.1.4 Flaws Encountered In Traditional Requirements Engineering Approaches

A lot of traditional (non-AO) requirements engineering approaches exist in literature such as discussed in section 2.1.3. All these approaches identify and treat the requirements in good manners. But, they all fail in handling the crosscutting concerns clearly. The major motivation factor in the invention of AORE approaches is to remove some of the flaws encountered in traditional requirements engineering approaches which are discussed here as:

These approaches suffer from *tyranny of dominant decomposition* symptom i.e. they are modularized in only one way by considering only one type of concerns such as use cases, viewpoints and goals at a time. And the other kinds of concerns that do not align with that modularization result as scattered and tangled modules. For example, PREview has focused on non-functional concerns. On the other hand, Use Cases have focused on functional requirements. In contract, Aspect-Oriented approaches, such as CORE treat all types of concerns equally and consistently. Thus, the first flaw encountered that AORE addressed is the equal treatment of all types of concerns of importance simultaneously.

Some traditional approaches have identified non-functional requirements as crosscutting requirements. But, they do not consider functional requirements as so. Also, crosscutting requirements are not modularized separately. In contract, Aspect-Oriented approaches, such as CORE have considered this issue. Thus, the second flaw encountered that AORE addressed is to identify and characterize the crosscutting influence for both functional and non-functional requirements and modularize them separately as aspects.

Mostly, all the traditional approaches lacks on composition mechanism. AORE provides composition as primary issue and handles it using joinpoint model and
composition semantics. The joinpoint model exposes structured points through which requirements can be composed. The composition semantics provide systematic meaning to the composition. Thus, the third flaw encountered in traditional requirements engineering approaches that AORE addressed is lacking the mechanism for requirement composability.

2.2 Aspect-Oriented Requirements Engineering (AORE)

2.2.1 Separation of Concerns and Crosscutting Concerns

A concern is any matter of interest in a software system [SUT 02]. Many kinds of concerns may be relevant to different software engineers in different roles, to achieving different goals, or at different stages of the software lifecycle [TAR 01]. For example, the prevalent concern in object-oriented software engineering is the class, which encapsulates data concerns. Feature concerns, like printing, persistence, and display capabilities, are also common, as are concerns like aspects, roles, variants, viewpoints, product lines, and configurations. The artifacts of a software engineering process such as requirements specifications, designs, architectures, code, test cases, etc., are also common concerns in software engineering, and a given concern may span multiple artifacts. An appropriate separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote traceability within and across artifacts and throughout the software lifecycle; facilitate reuse, non-invasive adaptation and customization, and evolution; and simplify component integration.

Separation of Concerns (SoC) is at the core of software engineering. This term was introduced by Edsger Dijkstra, to refer the ability of identifying, encapsulating and manipulating parts of software that are crucial to a particular goal or purpose in his book “A Discipline of Programming”. The following exert explains the concept of separation of concerns [DIJ 76]:

“To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one’s subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The others aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is
what I mean by "focusing one's attention upon a certain aspect"; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. I usually refer to it as "a separation of concerns", because one tries to deal with the difficulties, the obligations, the desires and the constraints one by one. When this can be achieved successfully, we have more or less partitioned the reasoning that had to be done – and this partitioning may find its reflection in the resulting partitioning of the program into "modules" – but I would like to point out that this partitioning of the reasoning to be done is only the result, and not the purpose.

The basic idea behind SOC is to handle one property of a system at a time. In other words, a complex problem that is hard to understand should be divided into a series of smaller problems; those are less complex and easier to handle by the designer. These smaller problems may then be designed one at a time by different designers and finally integrated to solve the big problem. In its most general form, SOC refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose. Concerns are the primary motivation for organizing and decomposing software into smaller, more manageable and comprehensible parts, each of which addresses one or more concerns.

Significant advances have been made since early days of software engineering, with technologies such as structured programming, object-oriented programming (OOP), Booch object-oriented design, etc. As systems become larger and more complex, however, the scale and complexity of the SOC problem continues to grow. With that in mind, we are working on a new generation of SOC technology, which is aimed at overcoming the resulting problems, including increased complexity, low comprehensibility, low reuse, high impact of change and difficult software evolution and integration. A key goal of this new area of advanced separation of concerns is to overcome the tyranny of the dominant decomposition. Traditional software development paradigms cannot apply multiple decompositions simultaneously – only one dominant decomposition structure can be chosen at a time. No matter how well a system is decomposed, not all concerns can be well modularized. Only a portion of concerns is separated properly, always at the cost of other concerns. This is known as
the *Tyranny of Dominant Decomposition* [TAR 01]. The concerns that do not fit well in the dominant decomposition structure are spread over the system and tangled with other concerns. These concerns are collectively referred to as *Crosscutting Concerns*. Due to their vary nature, crosscutting concerns are responsible for *scattering*, which means the implementation of a concern is spread over several program modules; and *tangling* which means a program module implements multiple concerns. Typical examples of crosscutting concerns are non-functional requirements, such as security, fault tolerance, persistency. However, crosscutting concerns can also be functional requirements, such as auditing, or validation [MOR 02; RAS 06b].

![Code Scattering and Code Tangling](image)

**Figure 2.7: Code Scattering and Code Tangling**

There are four systematic activities for separation of concerns as [DIJ 76]:

i. **Identifying concerns** is the first activity for separation of concerns. The functional and non-functional requirements are elicited by applying one of the requirements engineering approaches. This analysis is performed by conducting surveys, interviewing users, and by analysing user manuals.

ii. **Separating concerns** is the second activity for separation of concerns, in which, crosscutting concerns and the core concerns are separated from the identified system concerns.

iii. **Representing concerns** is the third activity for separation of concern which focuses to represent the identified concerns in use case, goal graphs, feature lists, graphical representations, templates, or as XML specifications.
iv. *Composing concerns* is the last activity that merges the core concerns and crosscutting concerns. The composition is preceded by conflicts resolution between the concerns.

### 2.2.2 Advanced Separation of Concerns (ASoC)

Significant advances have been made over the last forty years, with techniques such as structured programming, Object-Oriented programming, Booch Object-Oriented design, etc. As systems become larger and more complex, however, the scale and complexity of the separation of concerns problem continues to grow. With that in mind, we are working on a new generation of SOC technology, which is aimed at overcoming the resulting problems, including increased complexity, low comprehensibility, low reuse, high impact of change and difficult software evolution and integration. A key goal of this new area of *advanced separation of concerns (ASoC)* is to overcome the tyranny of the dominant decomposition, from which many modern artefact notations suffer? They allow separation of only certain kinds of concerns (e.g., data in object-oriented approaches). Other important kinds of concerns overlap, interact with, and cut across the dominant modules, and cannot be encapsulated effectively. Instead, the software pertaining to other concerns is scattered across many modules, and tangled with the software pertaining to other concerns. ASoC approaches specifically address these key characteristics of large-scale, complex software. The area is young and poses many open research issues and practical challenges.

### 2.2.3 A Brief Introduction to Aspect-Orientation Software Development (AOSD)

A lot of techniques exist in literature which follows these fundamental principles of software engineering. Some success in this direction has been achieved. But, still complete separation of concerns is not achieved even in today's most popular programming paradigm like OOP. In these traditional techniques, some concerns may be easily encapsulated with their building blocks such as classes, modules, procedures etc. But, same is not possible for another. They are non-modular and spanning over multiple classes, modules, or procedures in a software system and are therefore called crosscutting concerns. Typical examples are persistence, logging, exception handling, synchronization, auditing, security etc. Due to their vary nature, crosscutting concerns are responsible for scattering, which means the implementation of a concern is spread
over several program modules; and tangling which means a program module implements multiple concerns.

**Aspect-Oriented Software Development (AOSD)** [ELR 01a] is another step towards achieving improved modularity and aims at alleviating the problems of scattering and tangling. It aims at addressing crosscutting concerns by providing means for their systematic identification, separation, representation and composition [RAS 03]. Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. It later uses composition mechanism to weave them with other core modules at loading time, compilation time, or run-time [BAN 06]. This results in better support for modularization hence reducing development, maintenance and evolution costs.

AOSD was first introduced at programming level, where aspects are handled in code. **Aspect-Oriented Programming (AOP)** is a paradigm that supports two fundamental goals:

i. Allow for the separation of concerns as appropriate for a host language.

ii. Provide a mechanism for the description of concerns that crosscut other components.

AOP isn’t meant to replace OOP or other object-based methodologies. Instead, it supports the separation of concerns, typically using classes, and provides a way to separate crosscutting concerns.

Many Aspect-Oriented Programming (AOP) approaches have been proposed such as AspectJ, AspectC, AspectC++, JBoss AOP, JAsCo, HyperJ, adaptive programming, and composition filters. Without causing any intrusive changes to the entire code base, AOP technology helps modularize the implementation, and helps reduce dependencies between modules [OLA 02].

AOP principles supported by such tools address the challenges of crosscutting concerns, which pure OO methods do not. According to [MIK 05] all these tools are built on similar principles, which are Advice, Aspect, Joinpoint and Pointcut. Using pointcuts and advice, an aspect weaver brings aspects and components together. An advice defines the code that is defined on these joinpoints.
i. **Join point**: Join points are well-defined points during execution where you can apply crosscutting code. They denote a position to give advice in an aspect. Different points in the code where aspects can be woven e.g., class, methods, structures etc.

ii. **Pointcut**: This is the term given to the point of execution in the application at which crosscutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method. Table 2.2 shows some of the more commonly used pointcuts in AspectJ.

```
<table>
<thead>
<tr>
<th>Pointcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call(signature)</td>
<td>Select join points whenever the specified method or constructor is called</td>
</tr>
<tr>
<td>execute(signature)</td>
<td>Select join points whenever the specified method or constructor is executed</td>
</tr>
<tr>
<td>get(signature)</td>
<td>Select join points whenever the specified field is read</td>
</tr>
<tr>
<td>set(signature)</td>
<td>Select join points whenever the specified field is written to</td>
</tr>
<tr>
<td>handler(type-pattern)</td>
<td>Select join points whenever the exception handler associated with the throwable type-pattern is executed</td>
</tr>
</tbody>
</table>
```

iii. **Advice**: This is the code that is applied to, or that crosscuts the existing code. There are three choices when advice is executed (a) before - advice code is executed before the original code. It can be used to read/modify parameter values, (b) after - advice code is executed after a particular control flow or original code is executed. It can be used read/modify return values. And (c) around - advice body is executed instead of control flow.

iv. **Aspect**: When you use AspectJ, the Java programming language is extended with the definitions of aspects. You define an aspect in much the same way that you define a class. An aspect lets you bundle together Pointcuts, Advice, and Inter-type declarations. Like Java classes, aspects can have fields and
methods (both static and non-static). You can create abstract aspects, just like classes. You can also extend classes, or aspects, to create new aspects.

Table 2.3: Advice Types

<table>
<thead>
<tr>
<th>Advice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>before()</td>
<td>Execute your code before the selected join point(s)</td>
</tr>
<tr>
<td>after()</td>
<td>Execute your code after the selected join point(s)</td>
</tr>
<tr>
<td>around()</td>
<td>Execute your code at the location of the join points, letting you wrap or skip the execution of the join point(s) if desired</td>
</tr>
</tbody>
</table>

While using AOP, we can write aspect code in aspect files and in most of the cases we do not have to modify primary concern or main business logic classes. This makes the code flexible, extensible and less error prone.

A lot of work also has been carried out at the design level mainly through extensions to the UML meta-model e.g. [CLA 01; SUZ 99]. Research on the use of aspects at the requirements engineering stage is still immature. Handling crosscutting concerns in the early stages of software development is beneficial rather than handling them in later stages of software development because it not only makes the design simpler, but also helps to reduce the cost and defects that occur in the later stages of development.

**Aspect-Oriented Requirements Engineering (AORE)** [YYU 04; BAN 04] has emerged as a new way to modularize and reason about crosscutting concerns during requirements engineering. AORE extends the notion of separation of concerns in RE (e.g., viewpoints, use cases, goals, etc.) with that of requirements-level aspects. Such aspects modularize requirements that affect and constrain other requirements. Requirements pertaining to these concerns are often (fully or partially) scattered in the statements of other requirements. By explicitly modularizing crosscutting concerns at the requirements-level, AORE makes it possible to reason about such concerns in isolation from early on in the software lifecycle. By providing support for composition of requirements-level aspects with other concerns in the system, AORE facilitates analysis and understanding of the influences and constraints exerted by the former on the latter. The composition also makes it possible to identify potential trade-offs among requirements-level aspects, i.e., when two or more aspects influence or constrain the same set of requirements. This makes it
possible to resolve such trade-offs early on, for instance, by weakening the requirements of one aspect with reference to the other after negotiations with the stakeholders. This early understanding of aspect trade-offs plays a significant role in shaping the system architecture [MOR 05].

Identification of aspectual requirements is, however, a non-trivial task. Firstly, as is the case for identifying relevant concerns using any RE technique, one often has to mine for aspects in large volumes of input documents. Documents, such as interview transcripts, are frequently imprecise, full of apparent contradictions and missing essential information. Secondly, parts of aspectual concerns can often be scattered across a document or even across documents making their identification difficult. This is further compounded by factors such as the occurrence of similar, often incomplete requirements in several places, mutual influence of requirements, difference of language used to express it or similar requirements and implicit requirement implication. Furthermore, once identified, the aspects need to be examined for such issues as dependencies, conflicts, and their resolution and then structured (along with other requirements) into a Requirements Specification document.

2.3 Summary

In this chapter, we briefly explained some basic concepts used in this research work. These concepts include requirements engineering, separation of concerns, advanced separation of concerns, aspect-oriented software development, and aspect-oriented requirements engineering. Further, we also discussed some traditional requirements engineering approaches, e.g., use cases, goals, and viewpoints. These approaches provide good support for identification and treatment of most kinds of requirements; but they do not explicitly support all kinds of requirements [JAC 92; SOM 97a; VAN 01]. However, these requirements approaches do not explicitly support well broadly-scoped requirements, such as crosscutting concerns, and do not explicitly support their composition. Moreover, they all suffer from the “tyranny of the dominant decomposition”.

Aspect-oriented Software Development (AOSD) [ACM 01] is another step towards achieving improved modularity during software development. It focuses on crosscutting concerns by providing means for their systematic identification,
separation, representation and composition [RAS 03]. It encapsulates crosscutting concerns in separate modules, known as aspects. It later uses composition mechanism to weave them with other core modules at loading time, compilation time, or run-time [BAN 06].

Aspect-Oriented Requirements Engineering (AORE) [RAS 03] is an early phase in AOSD that supports separation of crosscutting concerns at requirements level. It not only aims to provide improved separation of crosscutting concerns during requirements engineering, but also to provide a better means to identify and manage conflicts arising due to tangled representations of crosscutting concerns.

Over the last few years, several research efforts have been devoted for handling crosscutting concerns at the early phases of software development especially at requirements level. These efforts are meaningless unless all the crosscutting concerns are properly identified. Many approaches only consider non-functional concerns as crosscutting concerns. However, crosscutting concerns may also be functional, such as auditing, or validation [MOR 02; RAS 06]. In this research work, we have proposed a systematic approach that will be discussed in chapter 4 to identify both the functional and non-functional crosscutting concerns during requirements engineering along with its application on a case study. Our approach supports the identification of both the functional and non-functional concerns as crosscutting concerns.