CHAPTER 2

CURRENT TRENDS IN REQUIREMENTS ENGINEERING AND SOFTWARE ADAPTABILITY

2.1 INTRODUCTION

Requirements Engineering has a direct bearing on the quality of the resultant software. The scope of Requirements Engineering covers quality and quality of service attributes of software such as adaptability, reliability, performance, etc. which are non-functional requirements. In this chapter, the nature and scope of the Requirements Engineering process are examined with special attention on the quality attribute of software adaptability. The basic processes to generate requirements and the primary driver for creation and maintenance of software are reviewed. The Requirements Engineering process is made more complex by the fact that a set of requirements is never fixed but keeps getting modified with change in stakeholders, with changing needs and with changing operating environments. The introduction of a new or changed requirement into a set of stable requirements may also introduce inconsistency or even conflict between requirements. Approaches to resolve these are examined for applicability to software adaptability. The role of adaptability in making software long lasting emerges. The dimensions of software adaptability are assessed with the intention of identifying key features to make software adaptable.

The software development process in general and Requirements Engineering in particular, have certain qualitative imprecise / uncertain
characteristics such as descriptions, specifications, choosing between multiple alternatives, etc., which make Fuzzy logic suitable for use in the domain. The applicability of Fuzzy logic to this research work has been introduced in this chapter.

2.2 REQUIREMENTS ENGINEERING PROCESS

The Requirements Engineering process is directed towards defining two aspects of software: the direct functionalities expected by users and the constraints under which these functionalities have to be delivered. Nuseibeh and Easterbrook (2001) defined a broad scope for Requirements Engineering: it is concerned with the real-world goals of a software system, its functions and constraints. Additionally, it is also concerned with specifications of software behavior and their evolution. Sommerville (2004) highlighted the twin dimensions of Requirements Engineering, namely, determining the direct requirements (functionalities) expected by users and determining the constraints under which the software should be developed and perform, such as resource constraints and performance targets. A key characteristic of requirements is that they originate from stakeholders whose priorities may change and with this, their requirements. Also, requirements changes are specified in the domain of the stakeholder whereas its implementation is to be done on the software.

There are a number of basic steps in the Requirements Engineering process. Researchers have combined and reconfigured these steps in different ways and grouped them into a different number of phases. Sometimes a preparation phase, also called the ‘Context and Groundwork’ phase is given as the first activity. Jarke and Pohl (1994) identified three phases in the Requirements Engineering process – elicitation, expression, and validation. Davis (1995) described two distinct Requirements Engineering phases,
referred to as problem analysis and product description. Similarly, Sawyer et al (1996) proposed a four-phased generic model, consisting of discovery, analysis, negotiation, and specification. Nuseibeh and Easterbrook (2000) have defined a five-phase process consisting of Requirements Elicitation, Modeling and Analysis, Requirements Communication, Agreement and Evolution. Other classifications of the Requirement Engineering activities were given by Krasner (1989) and Leite (1998). Thayer (1997) divided Requirements Engineering into five phases, namely, Elicitation, Analysis, Specification, Verification, and Management. The five-phase model, is perhaps the most widely-accepted. In this model, the requirements generation process starts by eliciting user goals and needs. Elicited information is then analyzed and modeled to describe the requirements, which is then formally documented. Management is a continuous process to maintain the integrity and consistency throughout all requirements generation phases.

Pohl’s (1994) innovative perspective on the Requirements Engineering process, helps understanding the complexity of the process and its boundaries. Pohl visualized the Requirements Engineering process as a journey in a three-dimensional space in which the three axes are Specification, Representation and Agreement as shown in Figure 2.1. The Specification dimension represents the degree of understanding of the requirements among the stakeholders. From the initial state of opaqueness, understanding of the requirements moves towards completeness. The second dimension is Representation in which progress is made from the initial state of informality to a final state where all understanding is documented formally. The third dimension is Agreement. Initially, each stakeholder has his own views of what the system should do and how it should be done. From the initial state, Requirements Engineering moves the process forward towards the final position when there is complete agreement among all stakeholders.
Pohl’s perspective on Requirements Engineering helps understand how a stable condition is upset when a stakeholder introduces a new requirement or changes an existing one. Both the Specification and Agreement dimensions of Pohl’s framework are dependant on the stakeholder. The end points along the two axes are dependant on a fixed set of stakeholders and requirements. Introduction of a new stakeholder or new requirement or changed requirement upsets this balance after which the Requirements Engineering process will need to be redone. The third dimension in Pohl’s framework is the Representation dimension, which documents the understanding between stakeholders – an understanding that may not be valid anymore with the change in stakeholders. This constant change in requirements results in change management becoming a more-or-less constant and permanent activity.

Figure 2.1  Pohl’s Requirement Engineering process within the three dimensions
2.3  RESOLVING INCONSISTENT AND CONFLICTING REQUIREMENTS

Another feature of requirements is a consequence of the inherent characteristic of software engineering. It has been pointed out by Jackson (1995) that the software development process makes abundant use of descriptions in models, specifications, designs, user guides, test plans, etc. An inherent characteristic of descriptions is that they are often subjective, ambiguous and prone to different interpretations. Added to this is the fact that a lot of requirements are informal and are often vague and imprecise and they keep changing over time.

Consequently, requirements may be inconsistent or even conflict with one another. They can interact through some common dependency such as a shared resource or through interfering with the actions of each other. Two requirements are said to be inconsistent with or conflict with each other, if the satisfaction of one requirement affects the satisfaction of the other.

Nuseibeh et al (1994) used the term inconsistency to denote any situation in which a set of descriptions does not obey some relationship that should hold between them. Inconsistencies and conflicts in requirements are inevitable parts of the software development processes and even in the most well-defined, managed and optimized development process, system requirements are often uncertain or contradictory, or alternative design solutions exist, and errors in implementation arise.

Requirements inconsistencies and conflicts have both positive and negative aspects. Nuseibeh identified the general concept of conflict as a key driver for group communications, productive work and systems development.
Neuman (1995) commented that while the satisfaction of a single requirement is difficult enough, the simultaneous and continued satisfaction of diverse and possibly conflicting requirements is typically much more difficult.

Robinson et al (1999) have listed a number of reasons behind requirements inconsistencies and conflicts, namely,

- Complex requirements arising from complexity in both domain and software which makes it difficult to understand exactly what has been specified or how components interact,
- Implicit requirements, which by their very unstated nature, are potentially full of conflicts,
- Conflicting stakeholder requirements when different requirements of the various stakeholders cannot be mutually achieved – a problem compounded by the fact that stakeholders change over time, and
- Managing stakeholders’ requests and their expectations of change.

They have termed the entire activity of detection and resolution of requirements inconsistency and conflict as Requirements Interaction Management (RIM). RIM is a two stage process in which the first is detection of inconsistencies and conflicts and the second stage is the resolution of these.

Methods of resolution of inconsistent / conflicting requirements have relevance to software adaptability. When resolving inconsistent / conflicting requirements, trade-off criteria are used to prioritize between
imprecise alternatives. Similarly, in software adaptation, when there is a change in requirement, the best from an available set of adaptive responses has to be chosen using appropriate criteria. The approaches and techniques used to resolve requirements inconsistencies and conflicts are useful in software adaptability.

The various techniques to detect inconsistent / conflicting requirements have been categorized by Robinson et al (1999) as follows:

- Domain techniques that identify interactions at the requirement level,
- Theorem proving techniques in which correctness and completeness can be verified in specifications described as deterministic state machines,
- Scenario Analysis techniques in which a particular aspect of system behavior is examined, and,
- Model Checking techniques that involve operational exploration of state-based models, and execution monitoring to track the runtime behavior of a system and note its deviation from expectations.

In software adaptability, software is required to cope with a changed or new requirement which may be in conflict with, or inconsistent with, an existing requirement. The scenario analysis and model checking techniques which relate to runtime behavior of software, has a great degree of relevance to software adaptability. The others categories are more applicable in the earlier Requirements Engineering phase.
The various approaches to resolving inconsistent / conflicting requirements have also been categorized by Robinson et al (1999) as follows:

- In the relaxation method, conditions that cause inconsistent/conflicting are relaxed by generalization or abstracting requirements to a higher level.

- The refinement approach consists of specialization, compromise and restructuring. In specialization, inconsistent / conflicting requirements are partially satisfied. This approach can be applied where such requirements can be decomposed into specialized sub-requirements, some of which can be satisfied. In the compromise approach, the conflict may be over a value that exists within a domain of values in which case a substitute value is found. In the restructuring approach, methods are employed to change the conflict context by changing the object of conflict and / or the conflicting requirement. This can reduce constraining relationships and promote greater resolution.

- In failure recovery, attempts are made to restructure the context and avoid conflict by either restructuring the precondition of the conflict or re-planning the requirement so as to settle for achieving a subordinate goal.

- In the postponement approach, postponing resolution of a conflict may help because the postponed conflict may get resolved by resolution of other conflicts.

- Abandoning a requirement is a severe action in which a conflicting requirement is simply dropped.
With reference to software adaptability, the inconsistent / conflicting situation that may arise when software has to cope with a new or changed requirement and the methods described above will be applicable.

In Requirements Engineering, different approaches have been used to arrive at a comprehensive set of requirements. An examination of these approaches identifies the boundaries of Requirements Engineering.

A scenario-based method was proposed by Letier et al (2005). Scenarios differ depending on the context. In the context of requirements, scenarios describe sequences of interactions between agents of a system composed of software agents, human agents, and hardware devices. The objective is to provide an early analysis of such scenarios and inconsistencies and conflicts.

The Viewpoint-based approach of Silva (2002) acknowledged the fact of multiple stakeholders – a situation that will need management of conflicts and inconsistencies. Requirements Engineering also involves entities like domain descriptions, interface specifications, etc., which again, can be categorized and interpreted from different viewpoints. Discrepancies between viewpoints are identified and resolved for completeness of requirements.

The Goal-oriented Requirements Engineering method of Letier and Lamsweerde (2004) uses goals for driving the Requirements Engineering process. These goals can be either functional or non-functional; can be either high-level or very specific. The higher the level of a goal is, the more understandable they are to the variety of stakeholders and the more permanent they are likely to be. Goals are also likely to reveal conflicts among stakeholders and consequently between requirements at an early stage,
thereby enabling conflict-resolution methods to be deployed early in the development process.

The Software Architecture Oriented Requirements Engineering method proposed by Hong Mei (2000) focuses on the process aspect of Requirement Engineering. The main objective in this method is to introduce concepts and principles of software architecture into requirements analysis and requirements specification, promote requirement reuse, traceability between requirement specifications and system design, and consistency in the whole software development process. However, implicit in this approach is the assumptions that requirements do not change.

The approaches of Letier et al (2005), Letier and Lamsweerde (2004), Silva (2002) give due importance to stakeholder involvement in the Requirements Engineering process but the entry of a new stakeholder at a late or post-development stage, or, a late change in requirement may upset the results. The goal-oriented methods of Letier and Lamsweerde (2004) attempt to draw up a set of complete requirements by focusing on high-level goals. Requirements are implementation of goals and goals being at a higher level are likely to be more permanent and have a broader agreement from multiple and varied stakeholders. This approach minimizes disagreements amongst stakeholders. But goal agreement at the highest level may break down at subsidiary levels. Another difficulty with this approach was foreseen by Potts (1997), namely that by abstracting away from the immediate problem, the subtleties, special cases, interpretations and concrete features of the context of use are reduced in importance in the process of capturing requirements.

A number of mathematical, logic-based techniques and formal methods have been used in Requirements Engineering.
Gervasi and Zowghi (2005) commended the use of formal methods for Requirements Engineering and for identifying and analyzing inconsistency in requirements from multiple stakeholders. Broy (2001) advocated intensive use of formal methods by mathematically modeling the entire Software Engineering process, including Requirements Engineering and then applying mathematical techniques to the model. The Software Cost Reduction method proposed by Parnas (1983), is a tabular method to ensure that systems satisfy a set of critical properties such as security, safety, fault-tolerance, survivability, etc. This method is useful in the development of embedded systems and such critical high-availability software.

Wen et al (2004) used a formal component state representation method called ‘Behavior Trees’ to determine the impact of changes in the functional requirements and the implementation of these changes. The Use Case Maps (UCMs) method, proposed by Van der Poll et al (2003), used a scenario-based notation for describing in a high-level way, the link between the organizational structure of a complex system and the related behavior to validate scenarios and mediate communications between key stakeholders. Jilani et al (2001) defined the concept of distance between specifications, such as Functional distance which measures to what extent two specifications act alike and Structural distance which measures to what extent two specifications look alike. The Win-win method of Lamsweerde and Letier (1998) is an iterative process having three steps which ends when a final set of goals are arrived at after inconsistency and conflict resolution.

A logic and logic programming based technique of Nuseibeh (1996) analyzes event-based requirements specifications in terms of a system’s reaction to events and safety properties. The Problem Frames method of Jackson (1995) provides a systematic approach to the decomposition of the problem followed by solving of the sub-problems.
Hunter and Nuseibeh (1998) used classical logic mathematics to represent partial specifications and determine inconsistencies. The Blackboard method of Hayes-Roth (1983) seeks to develop solutions by coordinating the knowledge of distributed stakeholders in a shared blackboard space.

The Viewpoints method of Finkelstein et al (1993) has use in large systems where there are, typically, multiple stakeholders who have multiple expectations and multiple perspectives of the system promoting distributed development of specifications. In this method, inconsistency and conflict resolution is done by combining Viewpoints framework and logic-based approach. In Deficiency-Driven Requirements Analysis, the failure of a design to satisfy a system requirement or a constraint is characterized as a deficiency which is to be rectified. The result of these operations is the emergence of a new set of specifications, which can then be analyzed for deficiencies. The cycle is continued till all requirements and environmental constraints are satisfied.

Formal techniques are very useful in the pre-program development stages of software development, for detection and resolution of requirements inconsistencies and conflicts and for vetting the requirements of large and complex systems which is best done through automation. They are also very useful when there is an abundance of resources, or when the problem can be defined in sharp mathematical terms. In Requirements Engineering, formal methods when applied to a set of frozen requirements help to identify inconsistencies and conflicts.

Formalizing requirements requires proper training and experience. The effort in formalizing requirements is considerable but necessary in large or complex or critical systems.
Formal methods require practitioners to have a high level of training and this factor inhibits their understanding by end users and most of the other stakeholders. Gervasi and Zowghi (2005) reported difficulty in using formal methods when discussing specifications with stakeholders in terms of understanding and communication. After a study to compare formal with classical methods, Widmaier (2000) reported that while a formal approach produced better results than classical software engineering, the advantage was not significant.

It is observed that considerable work has been done in the area of resolving inconsistent/conflicting requirements for quite some time and there are many products that aim at generating a comprehensive set of requirements. Each of these products has its own strengths which make it applicable to a particular domain or type of software. The fact that no perfect solution has been found suggests that there are limits to Requirements Engineering processes.

The approach of Mylopoulos et al (1992) has relevance to software adaptability. The Requirements Engineering process was described by Mylopoulos et al as choosing the best among many alternative design options that emerge in the Requirements Engineering process. Similarly, in software adaptability, choices have to be made between possible multiple adaptive responses to a changed or new requirement.

Requirements originate from stakeholders and, Requirements Engineering processes strive to capture all current and possible future requirements.

However, developing software to meet all the requirements thrown up by the Requirements Engineering process will call for providing for even
uncommon / rare functionalities and result in inefficient code with adverse impact on performance and software maintenance. Besides, there is always the possibility of a new requirement or one that is suddenly discovered. Software adaptability, on the other hand, offers a better solution as it makes the software lean and flexible by matching the software’s behavior to the changing requirements. Software adaptability can be said to complement effective Requirements Engineering processes.

2.4 ADAPTATION DESIGN CONSIDERATIONS

A number of studies have addressed the issue of adaptability and various approaches suggested.

Lapouchnian et al (2006) have provided a high-level view of adaptation and listed three ways to make software adaptable, namely:

- Providing it with possible adaptive functionalities,
- Providing it with planning capabilities so that it can delegate tasks to external agents, and
- Providing it with an evolutionary learning capability.

Subramanian and Chung (2002) have identified the following three features that an adaptive software system should have:

- The ability to recognize the change in its environment,
- The ability to determine the change to be made to the system according to this change in environment, and
- The ability to effect the change in order to generate the new system after adaptation.
They have categorized adaptation techniques as based on architecture, component, code, genetic algorithm and dynamic adaptation.

Some other approaches are reviewed below:

Koutsoukos et al (2001) suggested use of coordination contracts in an object-oriented environment. A coordination contract is a set of rules and constraints on the interaction between any two objects and the contract superposes its behavior on the interaction of the two objects between which the contract is valid. Adaptation is achieved by changing the contract and not the objects.

Yau and Karim (2000) proposed a real-time component factory for developing adaptable components for distributed systems. The real-time component factory develops components that can be customized for task priority and exception handling policy, by providing specific interfaces for customizing these services.

Ayed et al (2000) developed mathematical models for program adaptation based on incrementation, merging, modification and composition. Using these models any program adaptation along these methods can be mathematically achieved.

de Lemos (2000) described a connector-based adaptation process. The connectors, called co-operative action describes collaboration between classes. In order to achieve adaptation, the collaborations between classes are changed on the fly, without changing the classes.

Oreizy et al (1999) have commended architecture based techniques over run-time software evolution, because they provide several unique
benefits over previous approaches. These benefits include a common representation for describing software systems and managing runtime change, separation of computation from communication, and encapsulation of change application policies and scope within connectors. In their approach, the adaptation is divided into two sub-functions. The adaptation component (or manager) takes high-level decisions which are implemented by the evolution manager. The process is iterative.

Bosch (1999) proposed superimposition for adaptation of components. In superimposition, different behavior can be superimposed on an object to change its original behavior in a manner transparent to the object's clients.

Highsmith (1999) proposed an adaptive software development methodology. This methodology has three steps, namely speculation, in which a general idea of where to go in building the software system is obtained, collaboration, in which there is shared development of software and learning from experience.

McIlhaga et al (1998) have addressed the mechanics for adaptation and listed design criteria for making software adaptable.

Jarzabek and Hitz (1998) suggested domain-specific software architecture (DSSA) to be used for software evolution. Any architecture of a software system is to be treated as an instance of the DSSA and evolution is achieved by creating another instance of the DSSA with the needed modifications.
Keller and Holzle (1998) suggested an adaptable interface for each component. The code in the adaptable interface can be changed as required to achieve the needed adaptation without making any changes to the component.

Keller and Schauer (1998) performed binary component adaptation for changing Java classes on the fly.

Liu (1997) proposed an iterative evolutionary process in which requirements analysis and design are treated as successive evolutions of specifications. In a series of iterative changes, informal requirements are attempted to be formalized. Successive iterative evolutions move the specifications towards completeness.

The technique of Peterson et al (1997) called for developers to anticipate portions of the program likely to change during runtime, and structure the program around functions that encapsulate such changes and hence make module-level runtime changes. However change impact can happen at multiple points within the application source code. This technique is difficult to implement for large systems.

Gratch and DeJong (1996) recommended machine learning as a way to adapt general solutions. Using different learning algorithms, solutions for specific situations can be developed.

Gupta et al (1996) proposed a code-based technique in which the target program is analyzed to locate the program control points at which all variables affected by a change are guaranteed to be redefined before use, thereby addressing the issue of state change of the application. Adaptation conditions are put in at these points. They concede that in the general case locating all such control points is difficult and approximate techniques based
on source code data-flow analysis and developer knowledge are required. Scaling up this approach to manage change in large systems written in complex programming languages is still an open research problem.

Spears et al (1993) proposed use of genetic algorithms to deal with adaptation. Here the evolution of individual components occurs via the processes of recombination and mutation. Then the suitable components for the environment are selected.

Gorlick and Razouk (1991) proposed the ‘Weaves’ approach, which consisted of small software components that perform well-defined functions and are called ‘tools’. Tools communicate with each other asynchronously through transport services. Tool fragments communicate asynchronously by passing object references (pointers). If any problem occurs during communication, the tool fragment initiating the communication is notified and may retry the operation at its discretion. This enables the runtime reconfiguration of a ‘Weave’ an arbitrary network of tool fragments connected together by transport services, without disturbing the flow of objects.

Kramer and Magee (1990) visualized the subject application as a collection of processing nodes with links that enable communication between these nodes. When a runtime change is required, a ‘reconfiguration manager’ which has the information, moves those processing nodes that will be affected to an inactive state in which its state will be frozen. The reconfiguration manager is responsible for making decisions regarding the change application policy and its scope. Developers are required to have a detailed knowledge of the application to implement this method.
Notikin and Griswold (1988) provided for runtime changes by allowing users to define and execute new commands during the execution of the system. The application source code is in two parts, one consisting of the base application and the other achieved through an extension programming language. The base code is usually unchanged.

The approaches of Koutsoukos et al (2001), Yau and Karim (2001), Ayed (2000), De Lemos (2000), Bosch (1999), Peterson et al (1997), Gupta (1996), Spears (1993) address specific scenarios or program-level adaptability considerations. This implies that software needs to be created or engineered in a certain way for them to become adaptable. However, by following this approach, many existing, running programs may get excluded. Nevertheless, their analysis helps in crystallization into a set of guidelines for implementation on the subject software. Oreizy et al (1999) have recommended that adaptation be split into two component sub-functions, one that will take high-level decisions implemented through the second. McIlhaga et al (1998) have listed a number of design considerations for adaptation which have to be divided between the framework and the target software.

Based on a survey of various approaches to adaptation, the following key features / functionalities have been identified:

- An ability to sense and accept a change in requirement which will be the trigger for considering adaptive action.
- Set of adaptive responses based on an assessment of the changed requirement.
- A planning capability provided by an adaptive algorithm that will select from the set of adaptive responses to respond to a requirement change. The complexity of this algorithm can
vary depending on the criticality of usage and the framework should allow such variations.

- The ability to implement the adaptive changes onto the target software. This module is required to interface between the framework and the target program. This cannot be a standard component of the architecture because of the need to accommodate diverse target programs.

- An evolutionary capability that will enable the adaptation algorithm to improve the adaptive responses with increasing usage. This functionality is an extension of the planning capability mentioned previously.

- Provision to include users into the adaptation loop. For some systems such as safety-critical ones, it may be necessary that users participate in the adaptation process.

- User interface to facilitate stakeholders, who may be domain experts, to participate using the language of their domain.

- A capability to vary the frequency of adaptation depending on the subject application. In some applications, it may be necessary to implement adaptive actions immediately, whereas in others adaptation can be implemented at opportune moments with less adaptive overheads.

- Ability to monitor the behavior of the software so that changes can be sensed well in time and appropriate feedback can be given.

- Data collecting ability to journalize adaptation events, such as change requests, adaptive responses, new behavior, etc. This
will provide the evolutionary capability that will enable the adaptation planner to improve its responses. The frequency of data collection will depend on the subject application. Some may require very frequent collection of large data for a quick adaptive response, while in less critical applications it may sufficient to collect data at intervals.

2.5 Fuzzy Logic in Software Engineering

As mentioned previously, Requirements Engineering is a human-centered process, which deals with stakeholders’ preferences and attitudes besides their requirements. The qualitative and imprecise nature of some requirements prevents precise categorization, leading to inconsistencies and conflicts. This issue, which arises from the stakeholders’ problem domain, needs to be addressed in the solution domain. Requirements may only be stated approximately or expressed heuristically. Lee and Yen (1994) introduced the concept of a soft functional requirement which allows an elasticity to be introduced into the system model, which may more naturally reflect the competing requirements within a problem domain.

A software application is a complex entity whose behavior is the result of various sub-processes within itself as well as its operating environment and difficult to predict with exactness. Requirements Engineering makes extensive use of descriptions to specify software. Stakeholders who are more comfortable with descriptions need to deal with systems, which operate with discrete variables.

All these factors make Requirements Engineering suitable for deployment of Fuzzy set theory, enabling decision-making in an environment of descriptions and uncertain and unpredictable behavior.
A requirement can be either fully or partially satisfied. The degree of satisfaction that a requirement, R, derives from a component, C, can be represented as:

\[ \text{Sat}_R : C \rightarrow [0,1] \]

The descriptions and linguistic nuances that are inherent in requirements specifications make requirements satisfaction amenable to being expressed as fuzzy function, such as:

\[ \text{Fuzzy Sat}_R : \mu B(C), \text{ where the requirement to be satisfied for component } C \text{ is } R. \]

Liu and Yen (1996) have used fuzzy logic to analyze the issue of inconsistency and conflict in requirements. Fuzzy functions are used to represent the elasticity of imprecise requirements and relationships between requirements analyzed using these functions. The resolution of inconsistent / conflicting requirements has been explained.

In this research work, a fuzzy based algorithm is used in the proposed architecture to plan adaptive changes by choosing between available responses in the uncertain and imprecise conditions when a requirement changes.

2.6 CONCLUSION

Requirements Engineering processes are directed towards identifying requirements both current and future. Developing software to satisfy all requirements, including remote contingencies, will result in software that is unnecessarily loaded with rarely or infrequently used functionalities, thereby unnecessarily degrading its performance and requiring
more effort in maintenance and upgrade. On the other hand, software adaptability enables software to be flexible and with a changing behavior pattern that matches changing requirements. This policy ensures that software will run at maximum efficiency and maximum utility to users. Software adaptability starts where Requirements Engineering ends and Requirements Engineering needs to be complemented with Software Adaptability Engineering.

Software adaptability is the result of some key features and functionalities. These are to be present in the architecture and the subject software so that the software can be inserted into the framework and become adaptable.

In this chapter the features that enable adaptivity are identified. To enable software to become adaptable, changes may be required in the subject software under consideration. In the next chapter, these features and functionalities are elaborated into components that constitute the architectural framework.

Key components that deliver the features are given. Fuzzy logic has been found to be very useful in software engineering and is introduced in this chapter. Fuzzy logic has certain features that make it suitable for use as an adaptation algorithm. It deals with uncertainties and imprecision and has the capability to handle qualitative inputs from stakeholders. This capability will be of use in adaptive situations where adaptive responses have to be fashioned to cope with uncertainties and imprecision of requirements changes.