ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/233819114

System Requirements Engineering

Book · January 1995

CITATIONS
417

READS
8,604

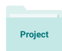
2 authors, including:

Peri Loucopoulos
The University of Manchester

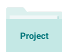
282 PUBLICATIONS **3,752** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Rubric View project



i-Doha View project

All content following this page was uploaded by Peri Loucopoulos on 31 January 2015.

The user has requested enhancement of the downloaded file.

PREFACE

While a commonly accepted concise definition of the term 'Requirements Engineering' is yet to be defined, it is widely agreed that

> Requirements Engineering deals with activities which attempt to understand the exact needs of the users of the software system to be developed and to translate such needs into precise and unambiguous statements which will subsequently be used in the development of the system.

Requirements Engineering is becoming the key issue for the development of software systems that meet the expectations of their customers and users, are delivered on time and developed within budget.

Since the mid 1970's when Requirements Engineering was established as a distinct field of investigation and practice, a great deal of progress has been made in the methods, techniques and tools used within this important phase of software development. Despite this however, a significant gap exists in terms of theories and technology between on one hand the activities pertaining to the specification of requirements and on the other hand those activities concerned with the design and implementation of software systems. Today, requirements are still, in many cases, collected, analysed and translated to software thanks to excessive informal interaction between users and developers, trial and error, the ingenuity of a few individuals, often with failures which are more spectacular than the success stories!

In contrast to other areas of software development, research and practice in Requirements Engineering are fragmented. Although there is a vast literature covering individual facets of the

area, such as descriptions of tools, methods or techniques, each contribution falls into one of two categories: either it represents a prescriptive approach to requirements, normally as part of a development method of a wider scope than just requirements or it deals with a narrow set of issues from a particular philosophical or technological viewpoint. Furthermore, the coverage of the area tends to pay more attention to specification languages issues while, issues such as for example, understanding organisational aspects and their influence on software requirements or understanding requirements in terms of system properties, are virtually ignored.

This book is our response to these shortcomings. In our involvement in the field of Requirements Engineering as teachers, researchers and practitioners, we have very often, felt the need for reference material which would meet a number of objectives such as:

- providing a discussion of the issues, models, techniques and tools applicable to the field of Requirements Engineering within a general framework applicable to many different viewpoints

- covering both practical experience as well as research efforts in the area

- avoiding 'cookbook' solutions which so often describe the style adopted by many methods or tools

We believe that the book that emerged addresses all these objectives. Since the book appears in a ‘Software Engineering’ series, the discussion of the various topics in the book is influenced by the target of the Requirements Engineering in the form of software systems. An attempt however is made to relate requirements for such systems to the organisational and social settings within which they are intended to operate. The book is intended to serve the needs of different audiences in a balanced way by:

- supplying teachers and students of Information Systems/Software Engineering with material which can provide the basis for a stand-alone course on Requirements Engineering or as part of a more broad Systems Analysis/ Software Engineering course

- providing practitioners and researchers with state-of-the-art material on techniques, methods and tools for the elicitation, representation and validation of requirements.

Despite our research involvement in the field, we have opted for a rather detached stance on the issue called best practices in software Requirements Engineering. Without any intention to nominate the 'best' requirements engineering method, technique or tool, we attempted to make an uncompromising statement of facts based on the latest and most authoritative views on the subject as they appear in textbooks, journal publications, reports on standards and conference proceedings and which are of concern to the community of practitioners and researchers working in this area. We deliberately avoided including cookbook recommendations of requirements 'solutions' preferring instead an integrated treatment of the requirements *issues*, to avoid disorientating the reader in a maze of sometimes misleading, often contradicting, approaches. We believe that practitioners of such a complicated task as Requirements Engineering should first equip themselves with a thorough understanding of the best concepts and theories, then become exposed to a number of tools and techniques and finally formulate their own opinion about what works and what does not in the areas they practice. Also, for those who seek to advance Requirements Engineering beyond the current state-of-the-art, we hope that this book gives insights into the most important and fruitful paths to the unparalleled challenge posed by system requirements.

Establishing requirements for a software-intensive system involves two intellectual activities, analysis and specification. The former requires conceptual analysis of the needs of customer and user needs, their goals and assumptions whereas the latter is concerned with descriptions of the system behaviour and constraints placed on the system and its development by its environment. These activities are carried out in a social setting involving the requirements engineer, the builder of the system, the customer who commissions the system, the user who will eventually interact with the system and the personnel who will finally introduce the system in the enterprise.

The material in this book is presented from a system engineering perspective while recognising that the contextual setting of requirements engineering is a social one.
The book is organised around a framework which captures the pivotal aspects of Requirements Engineering, i.e. processes, models and tools.

Chapter 1 provides all the essential background and terminological knowledge required for the understanding of the material in succeeding chapters. Requirements Engineering is viewed from different perspectives, i.e. business, Software Engineering, and even from a cognitive perspective which describes the behaviour of requirements analysts.

Chapter 2 seeks to shed light into the confusion caused by different (and sometimes contradicting) terminology used for describing the same concepts within Requirements Engineering, by suggesting a framework for Requirements Engineering activities. The

framework views Requirements Engineering as a combination of three concurrent interacting processes which correspond to the three major concerns of eliciting knowledge related to a problem domain, ensuring the validity of such knowledge and specifying the problem in a formal way. The three succeeding chapters are devoted to these three topics.

In Chapter 3 the first of the Requirements Engineering processes, namely requirements elicitation is examined from the perspectives of concepts methods and tools. First the conceptual foundations of elicitation as a process in its own right are established, followed by a detailed discussion of approaches to elicitation, which range from traditional Systems Analysis techniques such as user interviews to the latest methods employed in disciplines such as Ethnomethodology and Knowledge Engineering.

Chapter 4 deals with another concern of Requirements Engineering, namely the development of conceptual models which specify the desired behaviour of the software system and the properties that the system must exhibit. Modelling principles and techniques are introduced which ensure that all the relevant information and concerns can be captured in a conceptual model. A requirements specification is viewed as a composite of three components: enterprise requirements, functional requirements and non-functional requirements. In this sense this chapter takes a wider, and in our opinion a more appropriate view of requirements specifications, than the traditional view of concentrating almost exclusively on functional requirements.

The three-fold view of the Requirements Engineering as elicitation, specification and validation is completed in chapter 5, where the process of requirements validation is covered. In a similar manner to preceding chapters, this chapter discusses the difficulties inherent in obtaining the users agreement on what constitutes a valid description of their problem, and presents methods, techniques and tools which attempt to overcome such difficulties. Following a discussion on the importance of validation within Requirements Engineering, this chapter introduces validation techniques such as prototyping, animation, and expert system approaches.

Chapter 6 focuses on the 'tools' aspect of the 'Concept-Method-Tool' view of Requirements Engineering. This chapter gives a historical overview of the role of Computer Aided Software Engineering (CASE) in Requirements Engineering. The multiple classifications of CASE technologies in this chapter aim to guide the reader into establishing criteria for selecting, integrating and using CASE tools for Requirements Engineering.

Requirements engineering is a discipline which addresses issues within both spheres of ‘business’ and ‘software systems’ and importantly it is concerned with the relationship

between the two. The need for rapid response to changing business environments, the employment of new approaches to organisational restructuring and the enabling influence of computers and communications lead us to believe that requirements engineering is an essential discipline of study and enquiry which brings systems engineering concerns closer to problems experienced in organisational settings. Requirements engineering is about addressing the problems associated with *business* goals, plans, process etc. and *systems* to be developed or to be evolved to achieve them.

In this book we have striven to cover a range of issues of importance to requirements analysis and specification in a non-prescriptive manner. To this end we have opted for a wide coverage of the subject concentrating on a discussion of the issues and current approaches to the problems being experienced in requirements engineering.

The book is aimed at students of undergraduate and postgraduate programmes with a substantial component of system development subject matter. The book assumes that the reader has already knowledge of system development techniques for either data-intensive or real-time systems.

Acknowledgements

We wish to express our thanks to our colleagues who worked with us in exploring the exciting field of Requirements Engineering. In particular we wish to thank Janis Bubenko, Matthias Jarke, John Mylopoulos, Barbara Pernici, Colette Rolland, Arne Sølvberg, Alistair Sutcliffe, Babis Theodoulidis with whom we had many informative and stimulating discussions on this topic.

We would also like to thank Vangelio Kavakli, Vana Konsta, Nikos Loucopoulos and Roger Smith for their help with case studies material. As always, Janet Houshmand has provided much appreciated administrative support. Finally, we would like to thank Rupert Knight and his colleagues at McGraw-Hill for their help and collaboration in the production of this book.

Pericles Loucopoulos
Bill Karakostas

CHAPTER 1

Introduction

Introduction

Information systems are increasingly becoming an integral part of our everyday lives to the extent that the welfare of individuals, competitiveness of business concerns and effectiveness of public institutions often depend upon the correct and efficient functioning of these systems. The success of these systems, often referred to as socio-technical systems[1], depends to a large extent in their ability to meet the needs and expectations of their *customer* (i.e. the individual, group or organisation that commissions the development of the system) as well as their *user* (i.e. the individual, group or organisation that will work with the system itself). It is therefore, the task of the *supplier* of the system (i.e. the system developer, or service provider) to deliver a solution that meets the expected level of functionality and ensures successful 'integration' of the technical system in the organisational setting.

It has long been established that the effectiveness and flexibility of a system are inexorably related to the correct understanding of the needs of the system's customers or users. There is a

---

1 i.e. systems that involve computer-based components interacting with people and other technical system components in an organisational setting.

key component therefore, of any development process which plays a central role in this process namely the *requirements specification.* The process of developing a requirements specification has been called *Requirements Engineering* [COMPUTER 1985; TSE 1977].

As a discipline Requirements Engineering is still evolving with a diversity of approaches being proposed and a lively debate going on. Therefore, it is neither possible nor appropriate to be prescriptive about *the* approach that one might adopt in developing a requirements specification. It is however important to discuss some of the concerns underpinning the field of Requirements Engineering and highlight some of the major issues of current investigation and practice before proceeding with a discussion on approaches to eliciting, representing and validating requirements.

To this end, this chapter first examines the term ‘requirements’ from two relevant perspectives, the organisational perspective and the software perspective. Section 1.1 briefly discusses the issue of requirements from an organisational perspective and in particular from the need of organisations to transform their functioning by using an information system as a facilitator to such a transformation. Furthermore, understanding the organisational setting is crucial to developing a more complete understanding of requirements for information systems. Information systems and their formal descriptions exist for some reason - they serve some strategic, tactical and operational objectives of the enterprise. Indeed the development of an information system impacts on the functioning and social organisation of the enterprise itself and therefore it is important to establish at the outset the significance that requirements have in an organisation context.

Section 1.2 considers requirements from a software engineering perspective and highlights the place of requirements in the software development lifecycle. This is the traditional view of examining requirements, considering that the task of developing a requirements specification precedes other development activities such as design and implementation.

To complete the discussion on requirements, section 1.3 examines the characteristics of the Requirements Engineering process itself. The processes involved in Requirements Engineering, as observed in controlled experiments and by analysis of industrial practices are discussed in this section. This gives an insight to the nature of problems that are specific to the task of requirements specification and analysis.

1.1 Requirements

A definition of requirements in [IEEE-Std.'610' 1990] is given as:

(1) A condition or capacity needed by a user to solve a problem or achieve an objective.
(2) 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.
(3) A documented representation of a condition or capability as in (1) or (2).

Although, this definition has been given with software systems in mind, it is general enough to apply to non-software specific situations.

In general, requirements fall into two broad categories: *market-driven* and *customer-specific*. These two broad categories of requirements have different characteristics and are often treated differently within a development process. Some of the key differences reported in an extensive study of requirements engineering projects ([Lubars, Potts, et al 1993]) are shown in figure 1.1.

Market-Driven Projects	Customer-Specific Projects
• Requirements are sketchy and informal.	• Requirements are voluminous and more 'formal'.
• Use of techniques from manufacturers rather than Software Engineering, e.g. QFD.	• Use techniques from Software Engineering.
• Specification is in the form of a marketing presentation.	• Specification may be in hundreds of pages of documentation.
• Not readily identifiable 'customer'. Developers tend to have less experience in application domain.	• Make use of domain expertise. Developers have in-depth knowledge of domain (even sometimes surpassing that of the customers).
• Projects rely on consultants for advice on desirable features.	• Projects rely on in-house personnel.
• Less structured approaches adopted. Task force used in 'brainstorming' sessions.	• Structured approach following a particular approach.

Figure 1.1: Comparisons Between Market-Driven and Customer-Specific Projects

The primary concern of this book is with 'customer-specific' requirements i.e. requirements for systems that will need to operate within a well identifiable organisational context, although, it should be stressed that some of the issues discussed in the book are equally applicable to 'market-driven' requirements for example, specification of the functionality of the intended product or identifying the enterprise objectives that motivate the development of a 'product'.

1.1.1 Requirements - An Organisational Perspective

The usage of information systems has evolved from the automation of structured processes to applications that introduce change into fundamental business procedures. In increased level of sophistication, the contribution of information systems to organisations can be examined in terms of:

- *Automating* production by reducing the cost of the processes that make up production.
- *Informing* decision makers through the exploitation of automated processes.
- *Transforming* the organisation in a way that management and business processes make the best use of information technology by strategically aligning the information systems to the objectives of the organisation and its personnel.

In order for organisations to meet the challenges and opportunities presented by information systems in the automating, informing and in particular in the transforming stages the following are regarded as a general set of prerequisites [Morton 1991]:

- Clear definition of business purpose and vision of what the organisation should become. This vision should be visible and understood by the organisation itself.
- Alignment of corporate strategy and information systems development.

In other words, the development of information systems is not simply about designing database structures and algorithms but also about understanding the needs of individuals and other stakeholders within the enterprise and ensuring that the system meets user requirements and business strategy. There is therefore, a natural relationship between the 'enterprise' domain and those of 'system' and 'system user' domains as shown in figure 1.2.

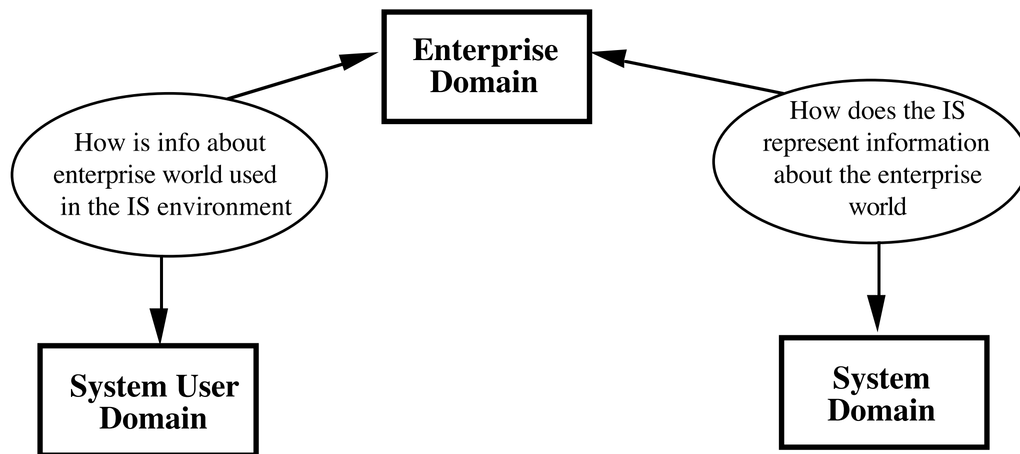


Figure 1.2: Relating Information Systems to Organisations

For example, by examining the objectives of the organisation, a rationale is established not only about the organisation itself but also about the infrastructure that supports or will support in the future the enterprise and the way that the system will fit the organisation and used by different end-user communities [Bubenko and Wangler 1993]. From an organisational perspective therefore, determination of requirements involves a number of interrelated tasks, shown in figure 1.3, that address management, social and information system concerns which need to be considered in a co-operative way in order to develop systems that fit their intended purpose.

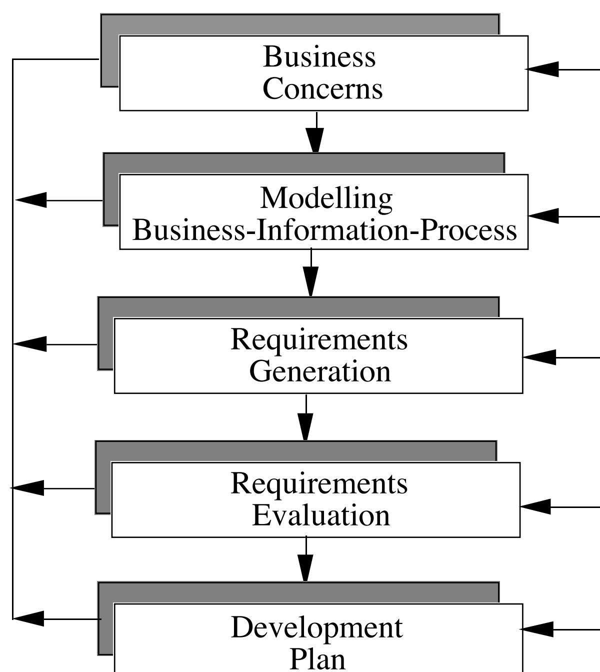


Figure 1.3: Requirements Specification from an Enterprise Perspective

Information systems are entering a new phase moving beyond the traditional automation of routine organisational processes and towards the assisting of critical tactical and strategic enterprise processes. Development of such systems needs to concentrate on organisational aspects, delivering systems that are closer to the culture of organisations and wishes of individuals. Specifying requirements in this context directly address issues such as:

> *Improving change management* by explicitly identifying and specifying those aspects of enterprises that are liable to change and by developing information systems that go beyond the automation of existing processes.
>
> *Providing integration of views within an enterprise* by adopting an approach which encourages a co-operative generation and assessment of requirements.
>
> *Relating information systems to business strategy* by facilitating the modelling of business goals and their realisation in information systems structure.

Many organisations perceive that a major challenge in the future will be to lead their organisations through the transformations necessary for a sustainable growth in a globally competitive environment. Competitiveness means increased quality, innovation and responsiveness to change. Business success is today critically dependent on the ability of the enterprises to link their information systems - their development and use - more closely to the business development process and such a success can only be achieved if infrastructure systems truly meet the needs and expectations as articulated within an organisational framework.

Crucially therefore, requirements engineering is about establishing the 'connection' between the need for some change within an organisational framework and the technology that could bring about such a change. In other words, requirements engineering can be considered as a way of managing change. This involves:

- an understanding at a conceptual level of the current status
- a definition of the change in terms of the transition from the 'old' conceptual situation to a 'new' target conceptual situation
- the implementation of the change in terms of the new components of the system and

- the integration of this new implementation in the environment which contained some legacy system.

1.1.2 Requirements - A Software Perspective

Interest about the role of requirements for software systems development can be traced to the early days of Software Engineering with the realisation that errors in the requirements definition stage resulted in costly maintenance of software systems at best and total rejection at worst [Bell and Thayer 1976]. As a consequence, Requirements Engineering was established as a sub field of Software Engineering with the task of developing models, techniques and tools that addressed this particular area. Since these early days the field has expanded in scope and point of view and there are many strands of investigation and practice nowadays that go beyond the strict confines of software construction.

Historically, the term Software Engineering was introduced when it became apparent that an engineering approach to software construction was needed. It was subsequently thought that if software development is to be approached in an engineering manner, then the same should apply to individual stages within it. As a result the term Requirements Engineering was adopted to describe an engineering approach to early stages of software construction.

The IEEE Glossary of Software-Engineering Terms gives the following definition of Software Engineering:

> Software Engineering is a systematic approach to the development, operation, maintenance, and retirement of software

The above term implies the existence of a *life-cycle* view of software. According to this view, software generally undergoes the phases of development, operation, maintenance and retirement. Each of these phases can be seen from a dynamic viewpoint as a *process*. The term *process* which will be used throughout this book is defined according to the International Standards Organisation (ISO) as follows:

> a unique, finite course of events defined by its purpose or by its effect, achieved under given conditions

Software development, therefore, is a process which has as a purpose the development of a complete software system. Within software development there are events which correspond to the start of individual (sub) processes. Thus, requirements determination is a sub process within software development. Other processes within software development include design, coding and testing. Each of these processes has a unique purpose which, however, contributes to the overall purpose of developing the software. The purpose of design for example is, according to [IEEE-Std.'729' 1983]

> ...defining the software architecture (structure), components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements

The view of software development as a sequence of processes is not dissimilar to other areas of engineering endeavour e.g. manufacturing. Manufacturing of a product involves a number of stages with processes such as product design, production planning, production monitoring and so on. Each manufacturing process has a purpose which contributes to the overall purpose i.e. the manufacturing of a complete product.

Manufacturing however is done in a systematic way. More specifically, there are three key elements-*methods, tools* and *procedures,* that enable the control of the manufacturing process and the development of a quality product.

- a *method* is a prescription of steps that need to be employed in order to achieve a specified result

- a *procedure* is a sequence of *actions* that must be performed

- a *tool* finally is the machinery used to perform some action as part of a procedure.

It is not difficult to draw analogies between software engineering and other engineering activities such as manufacturing. Similar to any other engineering discipline, software engineering uses methods, tools and procedures in a systematic way in order to arrive at the desired result which is the development of a complete software product.

Models for software development have been and are still continuing to be developed. With particular reference to the processes involved in requirements engineering, chapter 2 discusses a number of such process models.

There are many different models of software development. However, the majority of these different process models recognise the existence of components shown in figure 1.4 (note that no process model is shown in figure 1.4 but rather a set of deliverables - their derivation depends on the process model advocated by a particular method):

The *Requirements Specification* details the concerns of customers and users of the system, including the functionality of the system and the constraints that must be satisfied (both system and organisational constraints).

The *System Specification* is concerned with the definition of the system boundary and the information used in the interaction of the system and its environment. This specification represents a 'black box' view.

The *Architectural Design* represents a high-level view of the system's internal design.

The *Detailed Design* represents a decomposition of the system and concentrates on the details of individual components.

The *Implementation* is concerned with the software components that finally realise the original user requirements.

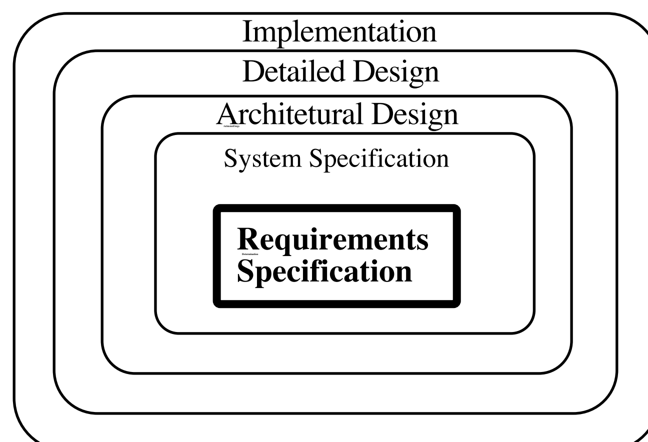


Figure 1.4: A General View of Components in Software Development

The model shown in figure 1.4 recognises that a software development process consists of a number of distinct activities each activity yielding a particular category of 'product'.

Developing a system constitutes a *design* activity. A design process typically involves [Dasgupta 1991]: (a) a set of requirements to be met by some artefact, (b) the output of the process is some design, (c) the goal of the designer is to produce a design such that if an implementation of this design were to materialise then the artefact would satisfy the requirements and (d) the designer has no knowledge of any design that satisfies the requirements. These properties of the design process are common to all information systems development approaches[2].

The following characteristics are generally recognised in the software development process:

- the software development process involves the generation of a number of different models
- the software development process can be viewed as a series of steps
- these steps are goal driven and can be regarded as *transitions between representations*, preserving the semantic content, as refinements on these representations are applied.

The exact nature of the activities, the products and the transformations depends on the chosen process model.

Software is not different from other human artefacts in the sense that it is made in order to fulfil a perceived need or to solve a specific problem. If one considers some typical areas of today's world where software is being used, it becomes obvious that software is not an ultimate end *per se*, but rather a means to an end. Software is the *enabling* technology, i.e. the technology which helps people in their problem solving tasks, tasks which can be as straightforward as writing a letter on a word processor or as complicated as flying the Space Shuttle. Since software is being used in all sorts of conceivable applications by all kinds of people, more often than not, the need for a software solution is experienced by people who are not software expert themselves. Naturally, non software experts cannot build the software they want themselves, at least in the vast majority of cases and therefore their need for a software solution must be catered for by the experts, the software developers. It is this seemingly straightforward situation, but in fact laden with many problems and difficulties, where a software user requires

2 Although the degree to which different methods adhere to these four aspects may vary considerably.

the services of a software provider which gave birth to terms like 'software crisis' and to the emergence of Requirements Engineering as a discipline.

At first, having a non-expert requiring the services of an expert seems like a totally natural situation which is often encountered in every day life. To those who have been involved with software either as users or as builders, the situation regarding the provision of software that satisfies some requirement, where this requirement is part of a larger set of organisational needs and expectations, would be recognised as sufficiently different to everyday needs for a number of reasons:

- Because software is not made of any physical substance it cannot be described in standard physical descriptions such as material, colour, dimensions etc.; it is rather described in terms of the often intangible characteristics of the situations, tasks and environments in which people use it.

- Software users best understand and describe their own work; software experts are more familiar with their domain-software.

It becomes apparent now that the seemingly straightforward task of understanding one's requirements and translating them into a software solution is a far cry from being as such, i.e. straightforward. The requirements part of software development is what is termed a *hard* problem, and yet one which we can ill-afford to leave unsolved. One obvious reason why requirements are important is because they set the criteria for the acceptance, success and usefulness of the software that is to be built. As discussed already, software on its own has no particular value; it is only when used as an enabler to other activities that it acquires a value. The best engineered piece of software is therefore worthless to someone if the software cannot be utilised to address their problem solving needs.

In addition, developing software is an activity which taxes a scarce resource, namely human software developers. Statistics show that while our ability to produce software will be increasing at a rate of 4% a year this is outstripped by the demand for new software which is increasing with an annual rate of 12%. In contrast the rate of increase in the number of qualified software developers is only 4% a year. It is obvious that we cannot afford the development of useless software and the waste of scarce resources such as time, people and money. A systematic approach to software requirements is needed, which will ensure the understanding of the user requirements and the production of useful software in a cost

effective way. Such an approach has to follow an *engineering* approach i.e. to apply proven methods, techniques and tools in a well described fashion.

1.2 Requirements Specification

Documenting requirements for software construction is an activity which results in what has been traditionally termed *requirements specification.* According to [Rzepka and Ohno 1985] a requirements specification represents both a model of what is needed and a statement of the problem under consideration. There is a wide variety of ways of expressing a requirements specification, ranging from informal natural language text to more formal graphical and mathematical notations. The structure of the specification itself varies according to different standards and practices [DOD-STD-2167A 1988; IEEE-Std.'830' 1984; NCC 1987].

At this point it is important to address the question "what is the purpose of a requirements specification?". There are a number of reasons for striving to develop a requirements specification. First, it provides a focal point for the process of trying to correctly understand the needs of the customer and user of the intended system. In other words it is the target of a systematic approach with the specification itself relying on the use of some 'language' for representing the contents of the specification. Second, the specification can and should be the means by which a potentially large and diverse population of requirements stakeholders and requirements analysts communicate. The specification itself can be used for clarifying a situation about the intended system or its environment i.e. the organisational context. Third, the specification may be part of contractual arrangements, a situation that may become especially relevant when an organisation wishes to procure a system from some vendor rather than develop it 'in house'. Fourth, the specification can be used for evaluating the final product and could play a leading role in any acceptance tests agreed between system consumer and supplier. Irrespective of its intended use the need for developing a requirements specification which at the very least expresses the problem in hand is well accepted by practitioners.

The traditional view of a requirements specification is that of a *functional* specification i.e. a definition of the desired service of the intended system. For example requirements for a system that handles a customer transaction in an airline ticket reservation system would need to address issues such as "`what is the information required by the system in order to issue a ticket and what results will the transaction processing function`

`will yield?`". This view of specification is indeed prominent in many contemporary information systems methods[3].

A functional requirements specification is concerned, as the name implies, with the description of the fundamental functions of the software components that make up the system. One is interested in defining the transformations which the system components should perform on inputs in order to produce some output. Functions are therefore, specified in terms of *inputs*, *processing* and *outputs*. A dynamic view of a system's functionality would need to consider aspects such as *control* (e.g. sequencing and parallelism), *timing* of functions (e.g. starting and finishing), as well as the behaviour of the system in terms of *exceptional* situations. Inevitably, since functions deal with a variety of data formats, data will also need to be defined and form part of the functional specification. Data can correspond to inputs and outputs to functions, stored data and transient data. The specification of 'real-world' *entities* and their *relationships*, particularly for data-centred systems, need to be defined at appropriate levels of abstraction and related to descriptions of system functions.

The emergence of software development methods during the late 1970's and 1980's gave prominence to the importance of functional specifications. A variety of specification languages[4] have been developed and extensively used for industrial and commercial projects. However, this over-reliance on functional specifications has been criticised as having a number of undesirable side effects. One concern is the amount of detail with which both requirements holders and requirements analysts have to deal. It has been argued c.f. [McDermid 1994] that when a functional specification becomes the focal point of requirements analysis then one makes a decision on the boundary of the system before any understanding is gained of the real needs of the requirements holders. In other words, functional specifications tend to deflect attention from other important aspects. For example, it is at least as important as defining a functional specification to consider issues such as the objectives of the system itself and the relationship of these to organisational objectives or specifying other desirable properties of the system (e.g. performance, security, usability, etc.) and constraints on its development (e.g. use of a particular toolset, economic constraints etc.). It is only then that one can adequately understand the reasoning behind the needs and aspirations of requirements holders. Furthermore, such a wider view of a requirements specification could accommodate situations requiring resolution of conflict of requirements at an organisational level, deciding to give

---

3 For a discussion on functional requirements models as found in contemporary methods the interested reader may refer to [COMPUTER 1985; Olle, Sol, et al 1983; Olle, Sol, et al 1984; Olle, Sol, et al 1986].

4 There exist for example a number of formalisms that are ideally suited to this task.

priorities to the stated requirements, or evaluating alternative scenarios for the satisfying of the stated requirements.

Another area of concern is the relationship between a requirements specification and a system architecture. It has long been accepted that a requirements specification should define the 'what' i.e. a description of the problem in hand and not the 'how' i.e. the way that the problem is to be solved. It has been argued in the past that a requirement specification should just say enough about the problem and nothing else. In this sense the requirements engineering process is a front-end to a series of other activities within the domain of software development. There are a number of factors however, that make a distinction between the 'what' and the 'how' difficult to achieve and in many situations possibly even inappropriate. There is much anecdotal evidence that in practice there are many projects in which some understanding of the system architecture is required in order to be able to articulate, represent and evaluate requirements that by their very nature address the solution space, albeit at an architectural rather than a detailed design or software implementation level. For example, a requirement on some system characteristic (e.g. "`the display screen of an air traffic control system should be capable of handling up to 100 tracks`") or a requirement on some general architectural consideration (e.g. "`the system must conform with existing practice for client-server organisation`" or "`the system must conform to some communication standard`") are all important requirements that cannot be ignored until the detailed design stage since by their very nature impose constraints on the design itself. Furthermore, many intended systems have to be considered in the framework of other legacy systems and developers very frequently have little choice on infrastructure components.

The example statements above can be thought as architectural requirements i.e. requirements which inevitably are imposed on the solution by the customer or user of the system. These statements may be considered as qualifiers on some 'real' organisational problem (e.g. "`the need for air traffic controllers to visualise air traffic scenarios`") but there would be little justification in this example to exclude this type of requirement from the specification. In the context of Requirements Engineering, the relationship between the 'what' and the 'how' is nowadays not as clear-cut as traditionally thought. A number of authors advocate that a requirements specification needs to go beyond the description of functionality and performance issues and that by including architectural issues during requirements provides an early opportunity to consider important parameters, tangible ones such as cost, as well as intangible ones such as acceptability of the system, in the introduction or evolution of a system [Garlan 1994; Jackson 1994; McDermid 1994; Mead 1994; Reubenstein 1994; Shekaran 1994].

The purpose of building a software system is to be found outside the system itself, in the *enterprise*[5], i.e. the context in which the system will function. Requirements of customers need to be represented in a specification in terms of the explicitly stated (in the specification) observed phenomena about the enterprise itself [Bubenko, Rolland, et al 1994; Bubenko and Wangler 1993; Greenspan, Mylopoulos, et al 1994; Jackson 1994; Jackson and Zave 1993; Loucopoulos and Katsouli 1992; Loucopoulos, McBrien, et al 1991; Nellborn, Bubenko, et al 1992; Yu and Mylopoulos 1994; Yu 1993].

A broader view therefore, of requirements specification is one that goes beyond the description of system functionalities. A functional specification should be one view supplemented, or even motivated by two other perspectives. First, an understanding should be gained of the domain within which the intended system will be firmly embedded. An explicit definition of the enterprise within which the system will eventually operate is a fundamental prerequisite to the development of a common understanding of the requirements holders, system customers, system users and system developers about the problem in hand. The emerging consensus within the Requirements Engineering community is that a requirements specification should include not only software specifications but also any kind of information describing 'real world' phenomena. Second, an understanding should be gained of the constraints that can be placed on the system, its environment or its development, known as *nonfunctional* requirements (NFRs) (e.g. security, availability, portability, usability, performance, etc.). For example the requirement that the "`airline booking system must respond within 15 seconds`" would be a non-functional requirement that needs to be considered by the system designer who would have to make a number of design choices in order to meet this constraint[6].

The term 'requirements specification' is used throughout this book from the broader perspective to refer to a description of requirements in the enterprise expressed in terms of the phenomena that are common to the enterprise and system domains. Descriptions of the enterprise are independent of any behaviour of any system whereas descriptions of the system refer to properties that the system must provide. This view is depicted in figure 1.5, where a

---

5 Synonyms of this are the terms 'application domain' and 'environment'.

6 The distinction between functional and non functional requirements is not often clear and some authors prefer to avoid this distinction. It has been rightly argued, that some requirements after originally being classified as non functional they become, in due course, functional requirements themselves. For example, in an air traffic control system, the need to be able to handle some upper limit of aircraft tracks would be originally expressed as a NFR but this eventually will have to be handled by a system function. However, from a methodological perspective the distinction is useful in delienating different areas of concern during requirements analysis and therefore, NFRs are considered from a distinct viewpoint in this book.

requirements specification is shown to constitute an interrelated set of descriptions in three domains namely, the enterprise, functional and non-functional domains.

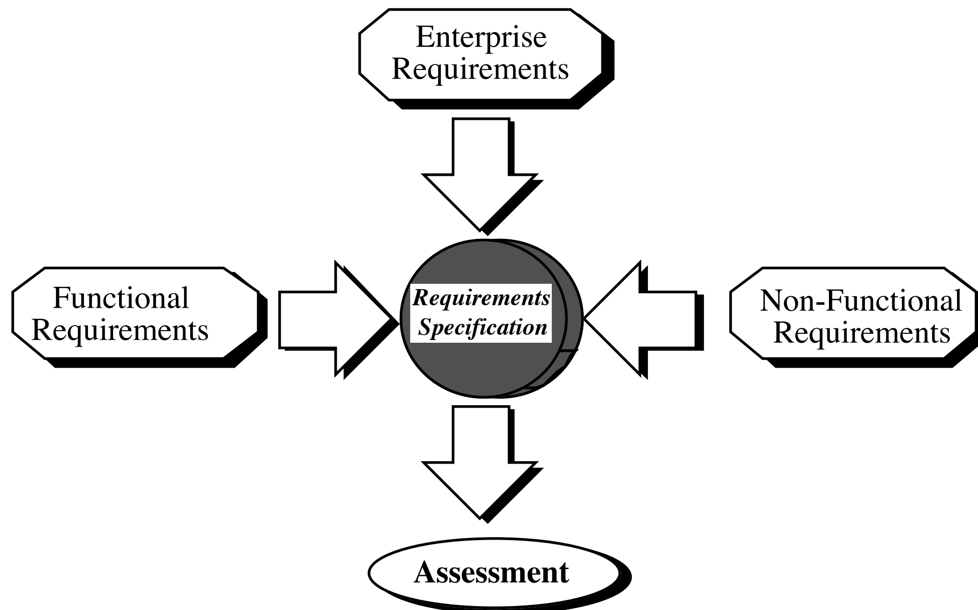


Figure 1.5: Conceptual Framework for Requirements Specifications

1.3 Requirements Engineering

The term *requirements engineering* (RE) can be defined as "`the systematic process of developing requirements through an iterative co-operative process of analysing the problem, documenting the resulting observations in a variety of representation formats and checking the accuracy of the understanding gained`".

This definition reflects the view that requirements specification involves an interplay of concerns between representation, social and cognitive aspects [Pohl 1993]. Issues of representation range from informal descriptions such as natural language expressions and hypertext to formal conceptual modelling languages. In the social domain, consideration is given to the complex social process in which the communication and co-operative interaction between the stakeholders of the requirements determines the quality of the final product. Issues in the cognitive domain concern different orientations of models in terms of understanding the process itself and validating the requirements.

Requirements Engineering consists of the knowledge elicitation, representation and validation cycle. The success of the requirements engineering process often depends on the ability *to*

proceed from informal, fuzzy individual statements of requirements to a formal specification that is understood and agreed by all stakeholders. However, the process is far from deterministic and straight forward. A requirements specification cannot be developed in a simple, linear fashion but a cyclic approach which gradually yields an involving specification seems to be more appropriate.

The transition from *informal to formal* requirements constitutes a conceptualisation activity within which a developer might make use of domain knowledge partly expressed in descriptions of the enterprise, and partly in existing requirements specifications. Reflecting back *from formal to informal* requirements is a process of validation which may take a number of different forms including prototyping, and explanation as to the decisions made in producing a requirements specification.

It is a truism that there is no unique or standard way for specifying requirements and although a number of authors have argued that specifications need to conform to certain characteristics such as completeness, correctness, unambiguity, understandability, modifiability and consistency [Dorfman and Thayer 1990], many of these qualities are hard to achieve. For example, it is very hard to test for completeness since there is no other 'model' against which the specification can be tested; it is only through repeated validation cycles that one can gain some confidence of completeness. The process is typically situation, context, and issue dependent, i.e. the kind of specification that is being developed during a development step depends on the development issues and questions experienced in previous steps.

Whilst, most of the skills required to develop software are primarily technical in nature this is not sufficient for the task of eliciting, specifying and validating requirements. The need to understand the underlying skills for the task of systems requirements analysis and their relationship to successful job performance has involved the use of other disciplines, apart from Computer Science, in attempting to define the way that systems analysts perform certain tasks. For example, the task of systems requirements analysis can be viewed as a type of *problem solving* [Newel and Simon 1972]. Under this perspective, requirements analysis is the reasoning process which attempts to understand the requirements of a problem domain in order to synthesise a solution for a system which will satisfy the needs of its users. During this thought process, an analyst may use clues, goals, strategies, heuristics, hypotheses, information and knowledge which has been acquired from different sources, i.e. the problem domain as well as from the analyst's own experience.

Examination of various Requirements Engineering projects has provided insights into the problem itself. For example, the following observations have been made about requirements and requirements analysis [Vitalari and Dickson 1983] [Button and Sharrock 1994]:

- Analysis problems, at their inception, have ill-defined boundaries, structure, and a sufficient degree of uncertainty about the nature and make-up of the solution.

- Requirements are found in organisational contexts, with associated conflicts on expectations and demands about some intentional system.

- The solutions to analysis problems are artificial. That is, they are designed and hence many potential solutions exist for any one problem. Ending the process of requirements specification is a matter of practical necessity.

- Analysis problems are dynamic. That is, they change while they are being solved because of their organisational context and the multiple participants involved in the definition and specification process.

- Solutions to analysis problems require interdisciplinary knowledge and skill.

- The knowledge base of the systems analyst is continually evolving and the analyst must be ready to incorporate changes in the technology and to participate with users in different ways.

- The process of analysis itself, is primarily cognitive in nature, requiring the analyst to structure an abstract problem, process diverse information, and develop a logical and internally consistent set of specifications. All the other skills such as interpersonal interaction and organisational skill facilitate this cognitive process.

A number of empirical studies have also examined the way that the Requirements Engineering process is carried out within the software engineering domain. A study of software engineering practices, in many different application areas, reports that "accurate problem domain knowledge is critical to the success of a project" and "requirements volatility causes major difficulties during development" [Curtis, Krasner, et al 1988].

The uncertainty inherent in trying to 'discover' and document requirements is also problematic. A review of the state of practice in Requirements Engineering [Lubars, Potts, et al 1993] revealed that "although most informants were able to describe the nature of requirements specification that they produced, they were unable to describe the *process* by which they arrived at these specifications". Also, "in customer-specific projects, changes to the requirements occurred due to changes in the environment whereas in market-driven projects competing products and insights to the market affected requirements. The importance of tracking the effects of changes to requirements through designs and implementation was recognised by most organisations".

By observing the way that requirements analysts carry out their work and by comparing experienced analysts to novices the following characteristics emerged [Curtis, Krasner, et al 1988; Fickas 1987; Sutcliffe 1990; Sutcliffe and Maiden 1990; Sutcliffe and Maiden 1989; Vitalari and Dickson 1983].

- Analysts use information from the environment to classify problems and relate them to previous experience. Experienced analysts begin by establishing a set of context questions and then proceed by considering alternatives. Much of the contextual information depends on previous knowledge about the application domain and the analogies that an analyst will establish are based on such knowledge.

- Expert analysts tend to start solving a problem by forming a mental model of the problem at an abstract level. This model is then refined, by a progression of transformations, into a concrete model, as further information is obtained.

- Hypotheses are developed as to the nature of a solution, as information is collected. Experienced analysts uses hypothetical examples to capture more facts from a requirements holder. Such examples are also used to clarify some previously acquired information about the object system.

- Developers almost always summarise in order to verify their findings. It has been observed that during a typical user-analyst session the analyst will summarise two or three times and each time the summarisation will trigger a new set of questions.

A requirements specification is likely to change many times before proceeding to design and as discussed already, a specification needs to be subjected to evaluation in order to gain

confidence as to its validity. This cyclic approach of acquisition-representation-evaluation involves a succession of propositions which are increasingly closer to end users' perceptions about the target system.

A requirement for an improved system begins with an initial hypothesis which is vague and requires further elaboration in order to produce the desired result. In this sense, requirements analysis involves the generation of hypotheses, and subjecting these hypotheses to a process of disconfirmation [Aguero and Dasgupta 1987; Hooton, Aguero, et al 1988]. These hypotheses are generated and evaluated against attributes of the intended system which it is anticipated that it will meet a desired state in the enterprise. Hypothesis formulation is based on the vision or belief that the participants in the analysis process may hold about the intended system, its role in the enterprise, its effects on organisation practices, its effects on personal and group status and so on. In short the participants define a set of system characteristics which in their opinion will lead to an improved situation. Hypotheses evaluation is based on the idea that every hypothesis undergoes criticism and is thoroughly examined for its validity i.e. looking for evidence to disconfirm the hypothesis. This implies that a specification describing objects, processes business rules, agents, IS components and so on, represents a set of propositions that are considered to be true until proven otherwise.

Summary

Requirements Engineering is the activity that transforms the needs and wishes of customers and potential users of computerised systems, usually incomplete, and expressed in informal terms, into complete, precise, and consistent specifications, preferably written in formal notations.

This activity is arguably the most crucial activity in system development, if only because errors made in the early requirements specification phases are the most costly to repair once the system has been implemented. It is also a most delicate one, as it requires heavy involvement of requirements stakeholders, and a consensus between them, as well as between requirements stakeholders and system designers, who have quite different backgrounds and concerns.

Requirements engineering is a systematic process, normally carried out within a broader spectrum of development activities, that attempts to discover, capture and document the requirements of many different stakeholders, those that commission the project as well as those that will eventually use the product. The input to this process is usually an informal, ill-defined 'wish-list' whereas the output should be a specification which is formal enough to be analysed

and agreed upon by requirements stakeholders and developers alike. Such a specification should have three orientation:

- a view of key aspects of the enterprise defining the objectives for the target system
- a view of the required functionality of the system and
- a view of the properties that must be exhibited by the system and its environment for it to meet the enterprise objectives.

The above issues have been examined from a number of different disciplines and indeed different 'philosophical' standpoints.

Areas of concern, which ultimately influence the approach adopted in deriving a requirements specification, fall into two broad categories:

- issues of functionality and service to be provided by the intended system and
- issues of suitability of the intended system in an organisational context.

Traditionally, these two classes of concern have given rise to two broad classes of approaches, the so call 'hard' and 'soft' methods. This rather unhelpful distinction has tended to polarise the way that one considers requirements determination with the effect that important facets of requirements go missing from specifications. However, recent realisation that requirements for socio-technical systems span the areas of concern of both 'hard' and 'soft' methods has resulted in the investigation of techniques that go beyond the traditional divide c.f. [Jirotka and Goguen 1994; Petrie 1992]. Emerging techniques within the field of Requirements Engineering advocate a balanced view between technical and organisational considerations.

References

Aguero, U. and Dasgupta, S. (1987) *A Plausibility-Driven Approach to Computer Architecture Design*, Communications of the ACM, Vol. 30, No. 11, 1987, pp. 922-931.

Bell, T.E. and Thayer, T.A. (1976) *Software Requirements: Are they Really a Problem*, 2nd International Conference on Software Engineering, 1976, pp. 61-68.

Bubenko, J., Rolland, C., Loucopoulos, P. and de Antonellis, V. (1994) *Facilitating "Fuzzy to Formal" Requirements Modelling*, IEEE International Conference on Requirements Engineering, 1994.

Bubenko, J.A. and Wangler, B. (1993) *Objectives Driven Capture of Business Rules and Information Systems Requirements*, IEEE Conference on Systems, Man and Cybernetics, 1993.

Button, G. and Sharrock, W. (1994) *Occasioned practices in the Work of Software Engineers*, in 'Requirements Engineering: Social and Technical Issues', M. Jirotka and J. A. Goguen (ed.), Academic Press, London, pp. 217-240.

COMPUTER (1985) *Special Issue on Requirements Engineering*, IEEE Computer, 1985.

Curtis, B., Krasner, H. and Iscoe, N. (1988) *A Field Study of the Software Design Process for Large Systems*, CACM, Vol. 31, No. 11, 1988, pp. 1268 ff.

Dasgupta, S. (1991) Design Theory and Computer Science, Cambridge University Press, Cambridge, UK, 1991.

DOD-STD-2167A (1988) *Defense System Software Development*, Department of Defence, February, 29, 1988, 1988.

Dorfman, M. and Thayer, R.H. (1990) (ed.) Standards, Guidelines, and Examples on System and Software Requirements Engineering, IEEE Computer Society Press, Los Alamitos, California.

Fickas, S. (1987) *Automating the Analysis Process: An Example*, 4th International Workshop on Software Specification and Design, Moterey, USA, 1987.

Garlan, D. (1994) *The Role of Software Architecture in Requirements Engineering - Position Statement*, The 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994, pp. 240.

Greenspan, S., Mylopoulos, J. and Borgida, A. (1994) *On Formal Requirements Modeling Languages: RML Revisited*, 16th International Conference on Software Engineering (ICSE-94), IEEE Computer Science Press, 1994, pp. 135-148.

Hooton, A., Aguero, U. and Dasgupta, S. (1988) *An Exercise in Plausibility-Driven Design*, IEEE Computer, Vol. 27, No. 7, 1988, pp. 21-33.

IEEE-Std.'610' (1990) *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, New York, Std.610.12-1990, 1990.

IEEE-Std.'729' (1983) *IEEE Standard 729. Glossary of Software Engineering Terminolgy*, The Institute of Electrical and Electronics Engineers, New York, 1983.

IEEE-Std.'830' (1984) *IEEE Guide to Software Requirements Specifications*, The Institute of Electrical and Electronics Engineers, New York, ANSI/IEEE Std 830-1984, 1984.

Jackson, M. (1994) *The Role of Software Architecture in Requirements Engineering - Position Statement*, The 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994, pp. 241.

Jackson, M. and Zave, P. (1993) *Domain Descriptions*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, San Diego, California, 1993, pp. 56-64.

Jirotka, M. and Goguen, J. (1994) (ed.) Requirements Engineering: Social and Technical Issues, Computers and People Series, B. R. Gaines and A. Monk (ed.), Academic Press, London.

Loucopoulos, P. and Katsouli, E. (1992) *Modelling Business Rules in an Office Environment*, ACM SIGOIS, No. August, 1992.

Loucopoulos, P., McBrien, P., Schumacker, F., Theodoulidis, B., Kopanas, V. and Wangler, B. (1991) *Integrating Database Technology, Rule-Based Systems and Temporal Reasonig for Effective Information Systems: the TEMPORA Paradigm*, Journal of Information Systems, Vol. 1, No. 2, 1991, pp. 129-152.

Lubars, M., Potts, C. and Richter, C. (1993) *A Review of the State of the Practice in Requirements Modelling*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, San Diego, California, 1993, pp. 2-14.

McDermid, J.A. (1994) *Requirements Analysis: Orthodoxy, Fundamentalism and Heresy*, in 'Requirements Engineering: Social and Technical Issues', M. Jirotka and J. A. Goguen (ed.), Academic Press, London, pp. 17-40.

Mead, N.R. (1994) *The Role of Software Architecture in Requirements Engineering - Position Statement*, The 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994, pp. 242.

Morton, M.S. (1991) The Corporation of the 1990s: Information Technology and Organisational Transformation, Oxford University Press, Oxford, 1991.

NCC (1987) The STARTS Guide. A guide to methods and software tools for the construction of large real-time systems, National Computing Centre Ltd, Manchester, UK, 1987.

Nellborn, C., Bubenko, J. and Gustafsson, M. (1992) *Enterprise Modelling - the Key to Capturing Requirements for Information Systems*, SISU, F3 Project Internal Report, 1992.

Newel, R. and Simon, H. (1972) Human Problem Solving, Prentice-Hall, 1972.

Olle, T.W., Sol, H.G. and Tully, C.J. (1983) (ed.) Information Systems Design Methodologies : A Feature Analysis, North Holland, Amsterdam.

Olle, T.W., Sol, H.G. and Verrijn-Stuart, A.A. (1984) (ed.) Information System Design Methodologies: A Comparative Review, North Holland, Amsterdam.

Olle, T.W., Sol, H.G. and Verrijn-Stuart, A.A. (1986) (ed.) Information Systems Design Methodologies : Improving the Practice, Elsevier Science Publishers B.V (North Holland), Amsterdam.

Petrie, C.J. (1992) (ed.) Proceedings of the 1st Conference on 'Enterprise Integration Modeling', Scientific and Engineering Computation Series, MIT Press, Cambridge, Massachusetts & London, UK.

Pohl, K. (1993) *The Three Dimensions of Requirements Engineering*, 5th International Conference on Advanced Information Systems Engineering (CAiSE•93), C. Rolland, F. Bodart and C. Cauvet (ed.), Springer-Verlag, Paris, France, 1993, pp. 275-292.

Reubenstein (1994) *The Role of Software Architecture in Requirements Engineering - Position Statement*, The 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994, pp. 244.

Rzepka, W. and Ohno, Y. (1985) *Requirements Engineering Environments: Software Tools for Modelling User Needs*, IEEE Computer, No. April, 1985, 1985.

Shekaran, M.C. (1994) *The Role of Software Architecture in Requirements Engineering - Position Statement*, The 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994, pp. 245.

Sutcliffe, A. (1990) *Human Factors in Information Systems: a Research Agenda and some Experience*, Human Factors in Information Systems Aanlysis and Design, A. Finkelstein, M. J. Tauber and R. Traunmuller (ed.), North-Holland, Scherding, Austria, 1990, pp. 5-23.

Sutcliffe, A. and Maiden, N. (1990) *Analysing the analyst: Requirements for the next generation of CASE tools*, in 'The Next generation of CASE-tools', S. Brinkemper and G. wijers (ed.), SERC, Noordwijkerhout, The Netherlands, pp. C2-1 - 9.

Sutcliffe, A.G. and Maiden, N.A.M. (1989) *Analysing the Analyst: Cognitive Models in Software Engineering*, , 1989.

TSE (1977) *Special Issue on Requirements Engineering*, IEEE Transactions on Software Engineering, 1977.

Vitalari, N.P. and Dickson, G.W. (1983) *Problem Solving for Effective Systems Analysis: An Experimental Exploration*, Communications of the ACM, Vol. 26, No. 11, 1983.

Yu, E. and Mylopoulos, J. (1994) *Understanding 'Why" in Software Process Modeling, Analysis and Design*, 16th International Conference on Software Engineering, Sorrento, Italy, 1994.

Yu, E.S.K. (1993) *Modelling Organizations for Information Systems Requirements Engineering*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, San Diego, California, 1993, pp. 34-41.

CHAPTER 2

Processes in Requirements Engineering

Introduction

Chapter 1 introduced the major deliverables of the Requirements Engineering process, as being *problem domain* model, *functional* requirements model and *non-functional* requirements model. However, what has not been mentioned so far is *how* Requirements Engineering derives the above deliverables. This Chapter introduces a *Framework* for describing the dynamics ('how') as opposed to the deliverables ('what') of Requirements Engineering. By integrating current proposals for Requirements Engineering processes, this Chapter arrives at a description of the process in terms of three major interacting, concurrent (sub)processes, namely *requirements elicitation, requirements specification* and requirements *validation* (Sections 2.2-2.4). These processes are discussed with respect to

their place in the Requirements Engineering lifecycle, their products, and the ways of interacting with each other. The requirements process model presented in this Chapter is compared with other models found in the literature. Additionally, this Chapter discusses the relative roles and importance of the elicitation, specification and validation processes in the context of major software development paradigms such as the waterfall model, prototyping and others.

2.1 A Framework for Describing Requirements Engineering Processes

Contemporary software methods do not prescribe a formal Requirements Engineering process. The majority of them (with a few exceptions, e.g. the framework for software specification described in [Rombach, 1990]) focus on the deliverables of the process rather than on the process itself. Such apparent inability or unwillingness to provide a formal description of Requirements Engineering probably explains the proliferation of requirements models.

Requirements engineering is easier described by its products than its processes. However, there is a need for a point of reference, as a basis for understanding, evaluating and comparing different proposals in the Requirements Engineering area.

A framework for describing Requirements Engineering processes can be constructed by considering three fundamental concerns of Requirements Engineering:

- the concern of *understanding* a problem ('what the problem is')
- the concern of formally *describing* a problem
- the concern of attaining an *agreement* on the nature of the problem.

Each of the above concerns implies that some activities must take place in order to provide answers, and in doing this some resources must be used. For example, in order to understand a problem, relevant information about it (a resource) must be at the hands of the problem solver. In turn, if that information is not already available to the problem solver then it must be obtained (an activity). In the same way, the relevant information must be validated in order to ensure its accuracy, consistency and relevance (another activity).

The current literature on software requirements, classifies activities such as the above under various terms i.e. as 'acquisition', 'elicitation', 'analysis', 'specification', 'validation' etc. However, such terms have multiple interpretations in software development methods with regard to their scopes, objectives, inputs and deliverables. In order to avoid adding to the existing confusion, this Chapter uses the minimum amount of terminology to describe Requirements Engineering processes. The processes proposed here (namely elicitation, specification and validation) correspond to the three fundamental concerns of Requirements Engineering, namely understanding, describing and agreeing on a problem.

Sections 2.2-2.4 discuss further the fundamental Requirements Engineering processes. Each process is described in terms of the following

- the *purpose* of the process
- the *input* to the process and its origins
- the *activities* which take place during the process and techniques used
- the *final* and *intermediate* deliverables
- the *interaction* with other Requirements Engineering processes

In section 2.5, existing models for requirements processes are classified and comparisons between them and the framework are made. The purpose of introducing this Framework is to provide the reader with an integrated (but not simplistic) view of the Requirements Engineering process. Such view is not readily available in the relevant literature.

Figure 2.1 shows a schematic representation of the Framework.

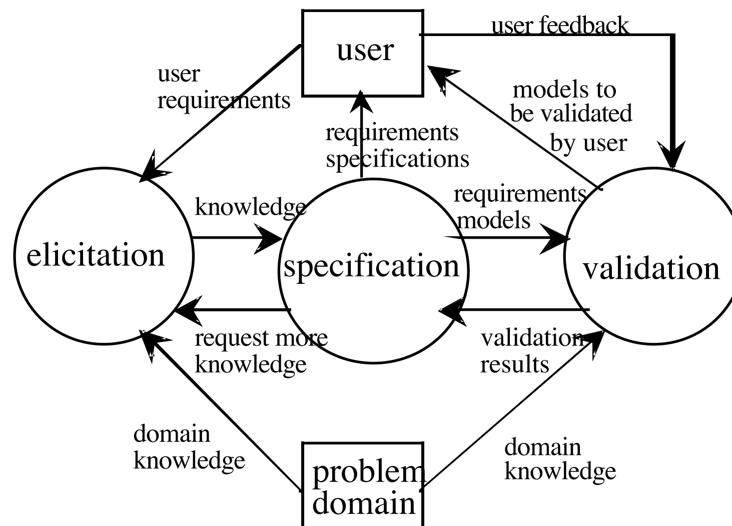


Figure 2.1: A Framework for Requirements Engineering Processes

2.2 Requirements elicitation

The importance of requirements elicitation cannot be easily overestimated. Even within methods which do not consider it as a separate process (i.e. by placing it under the name of 'analysis'), elicitation is the first activity that takes place and continues through the Requirements Engineering lifecycle. There are many reasons for this, with the most obvious one being:

when you have to solve somebody else's problem the first thing you have to do is finding out more about it.

2.2.1 Purpose of requirements elicitation

In the majority of cases, at the start of a software project, an analyst knows very little about the software problem to be solved. The only way to reverse this is by delving into *everything* that is relevant to the problem and, in a sense, becoming a problem owner. Software related problems, however, are usually complex enough to have the relevant knowledge about them distributed amongst many people, places and sources. Moreover, the knowledge is usually available in a variety of notations which range from sketches,

through natural language prose to formal models (e.g. mathematical models), to, even worse, some mental model in people's minds!

The purpose of requirements elicitation, therefore is to gain of knowledge relevant to the problem, which can be used to produce a formal specification of the software needed to solve the problem. It is not an exaggeration to say that at the end of the Requirements Engineering phase, the analyst should become an expert on the problem domain. If this does not happen, it will most likely mean that some important parameters of the problem were not considered (or not considered in the right way) and the software will not provide the best of the solutions to the users' problem (or even worse it will provide no solution at all).

2.2.2 Input to requirements elicitation

Whilst some approaches restrict the source of elicitation to be humans (the users), the approach taken in this book does not restrict the possible sources of domain knowledge. In reality, all of the following can be sources of domain knowledge

- domain experts
- literature about the domain
- existing software systems in that domain
- similar software applications in other domains
- national and international standards, which constrain the development of software in that domain
- other stakeholders in the larger system (e.g. an organisation) which will host the software system.

2.2.3 Activities and techniques of requirements elicitation

In requirements elicitation, the analyst is presented with the tasks of

- identifying all the sources of requirements knowledge
- acquiring the knowledge
- deciding on the relevance of the knowledge to the problem in hand
- understanding the significance of the elicited knowledge and its impact on the software requirements.

A variety of techniques for requirements elicitation exist today and a number of them is examined thoroughly in Chapter 3. Each technique has unique strengths and weaknesses and is more applicable to some types of problem than others. The most typically used techniques elicit the requirements from users through interviews, and through the creation of mock-up (prototype) software. More recent elicitation techniques attempt to *reuse* knowledge acquired in similar problem domains.

Elicitation is a labour intensive process, taking a large share of time and resources for software development. This is partly due to the fact that knowledge elicitation (particularly from humans) is inherently a difficult task.

2.2.4 Deliverables of requirements elicitation

The majority of the practised methods do not prescribe a formal outcome (model) for the elicitation process, as it is traditionally believed that the only formal outcome of Requirements Engineering should be the requirements specification model.

Experience, however, shows, that a better view of Requirements Engineering is as a *model creation* process. According to this view a succession of models is created throughout Requirements Engineering, starting with *conceptual* models and ending with the requirements specification model. The analyst starts formulating models of the problem domain in early stages of Requirements Engineering (elicitation). Such mental models

which contain domain dependent knowledge such as environmental factors, domain goals, policies, constraints etc., are usually formulated and exist in the analyst's head. As the analyst's understanding of the problem domain grows, these models become more refined and elaborated. Moreover, as the Requirements Engineering process progresses, the conceptual models become more software oriented than problem domain oriented (Figure 2.2). It is safe therefore to say that although elicitation does not (always) produce any formal models, it produces a succession of mental models of the problem domain which become more elaborate as the analyst's understanding of the domain increases.

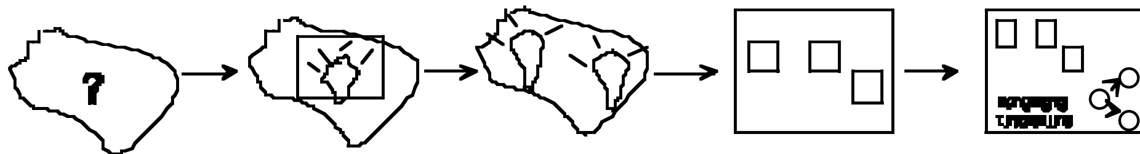


Figure 2.2: A succession of models is created in Requirements Engineering.

2.2.5 Interactions between elicitation and other processes

Elicitation can be considered as an ongoing process of Requirements Engineering. It can be viewed as providing the 'raw material' to other processes such as specification, needed for the production of a formal model. In this respect, elicitation occurs in parallel with specification and validation processes.

2.3 Requirements Specification

A specification can be viewed as a contract between users and software developers which defines the desired functional behaviour of the software artefact (and other properties of it such as performance, reliability etc.) without showing how such functionality is going to be achieved.

In arriving at such a ‘blueprint’ there is a need, as discussed in chapter 1, to include other types of description such as application domain (enterprise models) and nonfunctional models.

2.3.1 Purpose of requirements specification

The requirements specification process derives formal software requirements models to be used in subsequent stages of development. The purpose of producing a formal specification model is twofold:

- The specification model is used as an agreement between the software developers and the users on what constitutes the problem which must be solved by the software system
- the specification model is also a blueprint for the development of the software system.

2.3.2 Input to requirements specification

The process of specification requires as input knowledge about the problem domain. This knowledge is supplied by the elicitation process (Section 2.2). In most cases the input knowledge comes in a 'raw' format which must be converted to meaningful information in order to produce a formal specification model. Knowledge for example about an organisation's general policies must be interpreted with respect to how they affect the requirements for software systems. Other types of knowledge produced by the validation process is also employed in specification. Such knowledge states what is valid and what is not in the formal specification, and as such it acts as a force for changing the formal requirements model.

2.3.3 Activities and techniques in requirements specification

It is important to see specification as a complex process which requires feedback from the analyst to the user and vice-versa. The process uses and produces a variety of models including the final formal requirements specification. The process is *analytical* because all different kinds of knowledge that the analyst elicits from the problem domain must be examined and cross-related. Specification is also *synthetic* because the heterogeneous knowledge must be combined into producing a logical and coherent whole which is the requirements specification.

Requirements specification is described therefore by the following major activities:

- analysis and assimilation of requirements knowledge
- synthesis and organisation of the knowledge into a coherent and logical requirements model.

2.3.4 Deliverables of requirements specification process

The majority of Requirements Engineering approaches assume that the outcome of this process is the requirements specification model. It is more appropriate however to view the specification process as producing a variety of models which correspond to different views of the problem. In this respect, requirements specification produces:

- user-oriented models specifying the behaviour, non-functional characteristics etc. of the software which serve as a point of understanding between the analyst, the customer and the user
- developer-oriented models specifying functional and non-functional properties of the software system as well as constraints on resources, design constraints etc. which act as blueprints for further development stages.

All the above models correspond to what was termed in Chapter 1, enterprise models, as functional requirements models and nonfunctional requirements models. However, some requirements methods do not distinguish between different models for reasons of simplicity and effort. Such methods for example use the same functional specification model for all classes of requirements stakeholders. Nevertheless, this is not always appropriate since a specification notation which is perfectly clear for developers can be difficult to comprehend by users.

2.3.5 Interactions between requirements specification and other processes

Requirements specification is the central process of Requirements Engineering. Specification controls both the elicitation and validation processes as follows. During specification it may become apparent that more information about the problem is required. This will trigger the process of elicitation which will in turn supply the needed information. On the other hand, some change in the problem domain (e.g. change in some assumption, made about the domain) can trigger a change in the specification model. Thus elicitation can take place during the specification process. Similar interactions appear between specification and validation. Completion of some part of the specification model can cause the need for validation. For example, completing the specification of the user-interface may cause the need to validate the results in co-operation with the user. If the validation has negative results (i.e. the user is not satisfied with the proposed interface) then more analysis and specification must take place.

2.4 Requirements Validation

Requirements validation is an ongoing process of Requirements Engineering which aims to ensure that the right problem is being tackled at any time. In many Requirements Engineering methods, validation is not considered as a separate activity but instead is taken as a part of requirements specification. Nevertheless, it is important, for conceptual as well as practical reasons, to make the distinction between validation and other processes and activities of Requirements Engineering as it is argued in the remaining of this Section and again in Chapter 5.

2.4.1 Purpose of requirements validation

Requirements validation is defined as the process which certifies that the requirements model is consistent with customers' and users' intents. This view of validation is more general than those found in the literature because it treats validation as an ongoing process which proceeds in parallel with elicitation and specification. The need for validation appears at the moment a new piece of information is assimilated in the current requirements model (i.e. when the relevance, validity, consistency etc. of the new information must be

examined) and also when different pieces of information are integrated into a coherent whole. Validation, therefore is not applied only on the final formal requirements model, but also on all intermediate models produced, including the raw information received by the elicitation process. In this respect, validation encompasses activities such as *requirements communication.*

2.4.2 Input to requirements validation

Any requirements model (formal or informal) is subject to validation and thus provides an input to the process. Knowledge about the problem domain for example must be validated, i.e. checked for accuracy, consistency, relevance to the project etc. In a similar way, some part of the formal requirements model must be validated in parts and as a whole. For example, the specification of a mathematical routine must be checked for correctness. At the same time, the routine must be tested against the rest of the specification model, in order to make sure that it provides the results required by other parts of the specification.

2.4.3 Activities and techniques used in requirements validation

Validation is a process which requires interactions between analysts, customers of the intended system and users in the problem domain. This is similar to the scientific process of formulating a new theory (specification) and subsequently testing it by performing experiments (validation). In some occasions however, the analyst can test the validity of the requirements model without resorting to experimentation, e.g. by using common-sense. In a scientific sense, validation is therefore characterised by two principal types of activities:

- preparing the settings for an experiment
- performing the experiment and analysing its results

Chapter 5 contains an extensive discussion of techniques used for requirements validation. Some of the techniques require some preparation before the actual validation of the requirements, e.g. the development of a mock-up software system. This corresponds to the 'preparation for experiment' activities above. Other validation techniques involve extensive interaction between the analyst and the user through imaginary scenarios about the use of

the software system. These techniques correspond to the 'experiment and result analysis' type of activities mentioned above.

2.4.4 Deliverables of requirements validation

Validation delivers a requirements model which is consistent and in accordance with the users expectations. This does not mean that the model is in any sense correct. In the majority of the cases, validation yields a compromise between what was desired by the users and what is possible and feasible under the project constraints.

2.4.5 Interaction between validation and other Requirements Engineering processes

Validation is ever present in all stages of Requirements Engineering. The need for validation is triggered by the acquisition of new knowledge about the problem domain (elicitation), or by the formulation (even in part) of a requirements model (specification). Validation is also needed during the analysis and synthesis phases of requirements specification, since information analysed must be checked for correctness and synthesised requirements must be checked for logical consistency and coherence.

2.5 Other Requirements Process models and terminology used in the literature

Despite current attempts for its standardisation (see for example the IEEE Guide to Software Requirements Specification [IEEE, 1984]) a consensus on Requirements Engineering terminology has not yet been achieved. There are specific reasons for the proliferation of concepts and terminology used in Requirements Engineering for example:

- the field of Requirements Engineering is relatively young
- proprietary Requirements Engineering methods employ their own terminology which leads to a fragmentation
- Requirements Engineering consists of ill-structured, ill-defined processes which are hard to formally define.

This section compares other suggested requirements process models to the Framework introduced in this Chapter.

The definition of *elicitation* according to the Framework is slightly more general than others appearing in the literature (see, for example, [IEEE, 1984]) which define the deliverable of the elicitation process to be software requirements only. According to the Framework, the elicitation process produces not just software requirements but also all other kinds of domain knowledge which can be directly employed in analysing and specifying the software requirements. The Framework makes a distinction between requirements elicitation and other activities known as *context analysis*, *business analysis* etc. which attempt to establish the technical, economic and operational boundary conditions for the software development process. Also, according to the Framework, requirements elicitation commences after all the boundaries and development conditions have been established for the software project.

The Framework's definition of requirements elicitation partially overlaps with what is coined in the literature as *requirements identificatio*n, [Davis, 1982] [Martin, 1988] [Powers et al, 1984], *requirements determination* [Yadav et al, 1988], and *requirements acquisition*. *Requirements identification* and *determination* also partially overlap with the analytical phase of requirements specification.

In a similar manner, the analytic phase of requirements specification according to the Framework encompasses activities which the literature places under the headings of *software requirements analysis* [DoD, 1988], *requirements analysis* [Wasserman et al, 1986] [Pressman, 1987] *problem analysis* [Davies, 1990], *problem definition* [Roman et al, 1984], *requirements definition* [Berzins and Gray, 1985].

The analysis phase can also contain activities such as assessment of potential problems, classification of requirements, evaluation of feasibility and risks. The synthesis phase of specification contains activities such as

- *external product description* [Davis, 1990] which corresponds to the specification of the functionality(behaviour) of software,

- *requirements presentation* [IEEE, 1984] in which the results of requirements identification are portrayed and

- *software requirements specification* which includes complete documentation of what the software does externally (without regard to *how* it works internally) [Davis, 1990].

Also, terms such as *system design* [Roman et al, 1984] and *external design* [Wasserman et al, 1986] are (confusingly) used to describe the functional requirements specification.

Finally, validation according to the Framework encompasses activities such as *requirements communication* [IEEE Glossary, 1990] which is defined as the activity in which "`results of requirements definition are presented to diverse audiences for review and approval`".

Figure 2.3 shows the scope of the above process/activity requirements approaches with respect to processes of the Framework.

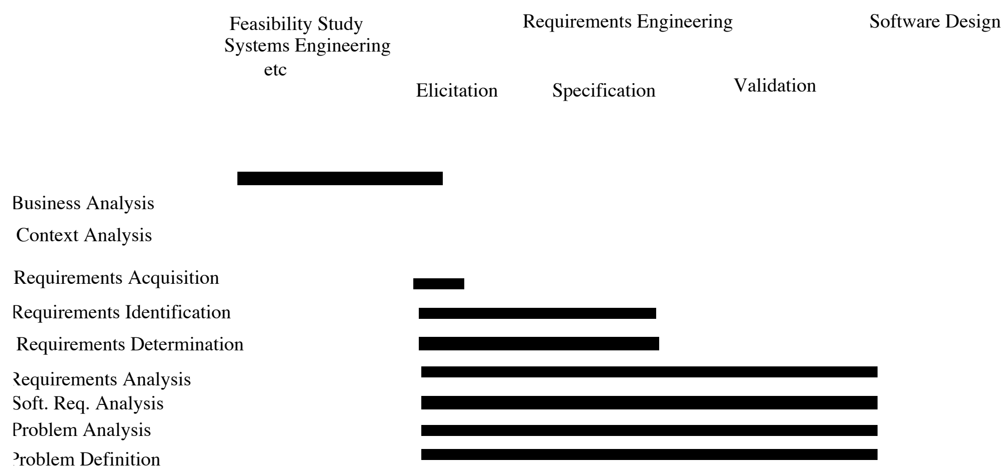


Figure 2.3 Coverage of Requirements Engineering activities by different approaches

2.6 Processes of Requirements Engineering in the context of software development models

This Section discusses the processes of Requirements Engineering, in the context of the following software development models

- the *waterfall* model
- the *spiral* model
- the *prototyping* model
- the *operational* model
- the *transformational* model
- the *knowledge-based* model
- the *Domain Analysis* model

At the end of this section, a comparison of the various software development models is made.

2.6.1 Requirements Engineering in the Waterfall Model

The philosophy which describes the 'waterfall model' [Royce, 1970] is that software development consists of a step-wise transformation from the problem-domain to the solution through a number of phases which are wholly satisfied before their successors begin. According to this philosophy, Requirements Engineering is the phase in which the software requirements are acquired, analysed, validated and a formal specification of them is produced. This phase is usually preceded by another one known as *Market Analysis* (also as *Business Analysis/Planning* or, in cases where software is part of a larger hybrid system, as *Systems Engineering*) which defines the context for the Requirements Engineering phase. Finally, Requirements Engineering is succeeded by the Design Phase which is concerned with specifying the software solution to the requirements specification. The popularity of the waterfall model lies in its principle that each of the phases are autonomous and can produce their deliverables using only the deliverables of their immediate predecessors, which guarantees that each phase can be completed and yield a specific outcome.

The waterfall model views Requirements Engineering as a *comprehension* phase which is succeeded by the *invention* (design) and *realisation* (coding) phases. Variations of the

waterfall model adopt slightly different organisations of the Requirements Engineering process. Also, the formality of the produced requirements model and the extent of tool-support varies with different waterfall-based approaches. All the variations, however suffer (to different extents) the consequences of the assumptions made by the waterfall model. More specifically, the assumption the waterfall model makes, namely that a phase only relies on the results of its previous one, is simplistic. In real life situations it is common that changes or new discoveries in a late phase of development (i.e. coding) can affect many earlier ones, including Requirements Engineering. In addition, changes in a phase after it has been completed can have a ripple-effect on changes in subsequent phases. For example changes in some business plan can have effect on the software requirements, on the design, on the code etc. In this respect, the waterfall model which is designed so as to resist change, becomes impractical in many situations. In the literature, the waterfall approach has been criticised for a variety of reasons, such as:

- lack of user involvement in the development after requirements specification has ended [McCracken and Jackson, 1981]
- inflexibility to accommodate prototypes [Alavi, 1984]
- unrealistic separation of specification from design [Jackson, 1982]
- inability to accommodate reused software [Castano and De Antonellis 1993; Fugini and Pernici 1992]
- maintainability problems [Balzer et al, 1983]
- complicated systems integration [Yeh and Ng, 1990].

In conclusion, the waterfall model takes a static viewpoint of Requirements Engineering by ignoring issues such as the inherently dynamic nature (volatility) of requirements and its impact on earlier and later phases of development.

2.6.2 Requirements Engineering in the Spiral Model of Software Development

The Spiral Model of Software Development [Boehm, 1988] recognises the iterative nature of development and the need to plan and asses risks at each iteration. According to this model, in each of the software development phases, the following activities must be performed:

- plan next phases
- determine objectives, alternatives, constraints
- evaluate alternatives, identify and resolve risks
- develop, verify next level product.

The Spiral Model introduces the additional subprocesses of Requirements Engineering, known as *requirements risk analysis* using techniques such as *simulation* and *prototyping* (both are discussed in Chapter 5) and *planning for design*. Such additions aim at reducing the risk of change at some subsequent stage. By evaluating the feasibility of the proposed requirements, for example, the approach reduces the risk of having to repeat Requirements Engineering once it has reached a stage (e.g. design) where it might be discovered that it is not feasible to produce the required software system. The Spiral model, however cannot cope with unforeseen changes during some stage of development and their impact on other phases. If for example, a new business objective occurs whilst the development has reached the coding stage, then unavoidably Requirements Engineering and design has to be repeated.

2.6.3 Requirements Engineering in the Prototyping Model

In the context of software development, *prototyping* [Floyd, 1984] is a technique which constructs and experiments with a mock-up version of the software system, in order to gain some understanding of the functionality and behaviour required from it.

It must be emphasised that prototyping is now a widely accepted technique which can be used in the context of any software development model. The *spiral* model of software

development, for example uses prototypes of the final software system in order to asses various risks associated with its development. Prototyping as a requirements validation technique is extensively discussed in Chapter 5. In the context of this Section, however, prototyping is viewed as the core activity in a software development methodology.

According to the prototyping model of development, processes of Requirements Engineering such as elicitation, analysis and specification are concerned with the planning, development and experimentation with a prototype. After an initial (and possibly incorrect and/or incomplete) understanding of users' needs, the prototype developers determine the objectives and scope of the prototype. It may be decided that a prototype has to be constructed for one (or more) of the following reasons

- to understand the requirements for the user interface (such requirements are in fact, difficult to specify using conventional means)
- to examine the feasibility of a proposed design approach
- to understand system performance issues.

After a number of necessary revisions the developers will usually feel that the prototype is satisfying all the objectives. At this point there are two options available to developers:

- either refine the prototype into a complete system or
- begin a conventional development process having benefited from developing the prototype.

If the first option is chosen, then it can be said that the prototype *is* the requirements specification of the system. In case the developers prefer the second option, then a formal requirements specification must be developed based on the enhanced understanding of the requirements, gained in the prototyping process. The choice between the first and second option must be made based on a number of key factors, namely

- how much functionality is already present in the prototype

- how robust, flexible and maintainable will the prototype-based production system be.

In conclusion, the prototyping development model views processes of Requirements Engineering such as elicitation, specification and validation as occurring in the context of developing a prototyping system. More specifically,

- elicitation of requirements is achieved by involving the user in experimental use of prototype

- analysis of requirements is done by analysing the structure and behaviour of the prototype

- formal specification coincides with prototype development (in case the prototype becomes the final system)

- validation is achieved by validating the prototype against the users intents.

The prototyping model has been criticised for hampering subsequent development stages when it is treated by the users as *the* solution, instead of what it actually is (i.e. a mock-up of the solution). It nevertheless presents a promising alternative to the waterfall model of development. The prototyping model is shown graphically in Figure 2.4.

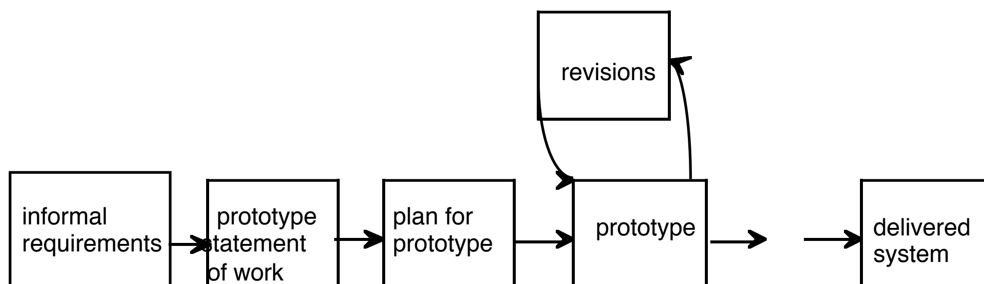


Figure 2.4: Schematic representation of the prototype model.

2.6.4 Requirements Engineering in the Operational Specification Model

The operational specification model challenges the assumption made by the waterfall model, namely that the requirements phase should be concerned with the “how” rather than with the “what” of development, (which should be the responsibility of the design phase). More precisely, an operational specification is a system model that can be evaluated or executed in order to generate the behaviour of the software system. The operational specification is constructed early in Requirements Engineering and its purpose is to analyse not only the required behaviour but also the required structure of the software system. The proponents of this approach claim that it is impossible to separate structure (the 'how') from behaviour (the 'what') when analysing a system [Zave, 1984] According to them, a major inadequacy of the waterfall model is that it leaves the designer with too many things to consider, specifically:

- how to decompose the problem domain functions into a succession of lower level functions

- how to introduce features such as information hiding and abstraction into the design

- how to take into account implementation issues, e.g. the feasibility of the design to be implemented in a specific software and hardware environment.

To tackle problems such the above, the operational approach advocates that decisions about the structuring of the domain problem should be made early in the development lifecycle. Although, such suggestion may cause the impression that an analyst using the operational model is forced to do design instead of analysis this is not necessarily the case.

The behaviour of each process would be specified using an operational specification language. The main characteristics of such language is

- the language is executable (either by compilation or interpretation, similar to languages such as *Pascal*, *Ada* etc.)

- the data and control structures of the language are independent of specific resource configuration or resource allocation strategies.

The above characteristics imply that although the language is executable it does not refer to any particular processor, memory or operating system organisation. Such decisions however belong to the design rather than to the Requirements Engineering phase.

In conclusion, the operational approach deliberately intertwines 'what' and 'how' considerations in an executable specification model. By doing so, the approach attempts to guarantee an early validation of the (executable) requirements model by the user as well as that the feasibility of the proposed solution. The operational approach considers Requirements Engineering processes such as elicitation, specification and validation as follows:

- the process of elicitation is carried at an initial stage, prior to the construction of the operational specification
- the specification process coincides with the construction of an executable specification model
- the validation process coincides with the exercise of the operational specification model.

The operational approach has been criticised that it deals with too many detailed technical issues too early. For example, concepts such as processes, asynchronous communication etc. used in the operational specification are regarded as difficult concepts to be fully understood by end-users. The diagram of Figure 2.5 shows the various stages of development using the operational model.

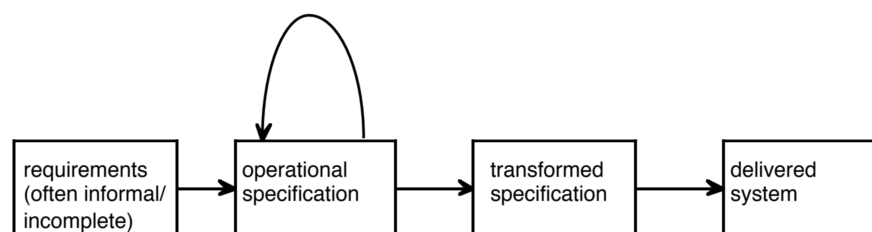


Figure 2.5: The operational paradigm

2.6.5 Requirements Engineering in the Transformational Model

The transformational approach to software development [Balzer et al, 1983] attempts to automate labour intensive stages of development such as design and implementation by using the concept of a *transformation*. A transformation is defined as a mapping from a more abstract object (such as a specification) to a less abstract one (such a design or piece of code). The transformational approach advocates the use of a series of transformations that change a specification into a concrete software system.

The transformational approach implies the need for a formal specification as the initial input, as automatic transformations can only be applied to a formal object (specification). The need for a formal specification, however, contradicts with the need for specifications which are comprehensible by the users. In general, the closer the specification is to the user understanding the harder (in time and effort) it becomes to apply the transformational process. Despite this problem, however, the transformational approach presents a promising new way of developing software for the following reasons.

- since the transformations are *correctness preserving* it is guaranteed that once the specifications are proved correct, the final system will also be correct. Thus the specification is the only object that needs to be validated

- the maintenance effort is significantly reduced since maintenance is now performed on the specification which is easier to understand and modify than code.

The altered code is produced by a two-step process. First changes are made to the series of transformations which produced the original code and which were recorded during development. Second, the modified set of transformations is 'replayed' in order to derive the new version of the software system.

The transformational systems which have so far been implemented vary as to the degree of the human participation to the transformation process which range from completely automatic transformations to manual selection of the appropriate ones by the analyst. The stages of the transformational model (shown in Figure 2.6) correspond to Requirements Engineering processes as follows:

- requirements elicitation is the phase which derives an initial informal and incomplete requirements model, prior to the commencing of transformational development

- requirements specification is the phase which produces the formal specification model

- requirements validation is the transformational phase where the formal model is validated by the user.

The transformational approach has been also criticised for its need to create and validate a formal specifications model as well as for the difficulty of automating the transformation process.

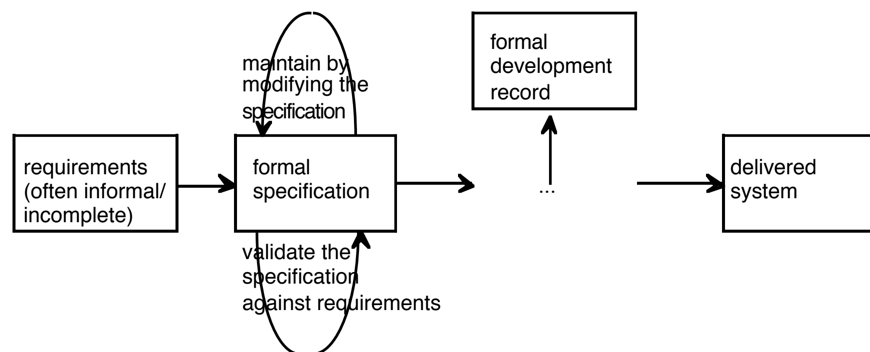


Figure 2.6: The transformational model

It can be noticed from the above discussion that the three software development models prototyping, operational and transformational model are not so dissimilar in principle. It is actually, true that all these models were proposed as a solution to the problem of inadequate end-user participation caused by the 'waterfall' mode of development. All three models therefore, propose the creation of an object (which is called prototype, operational specification and formal specification, respectively) early in the development life cycle, which can be used as a means of understanding and validating the user requirements.

2.6.6 Requirements Engineering in the Knowledge-Based model

This section briefly describes approaches to Requirements Engineering which are characterised by the use of intelligent software tools to perform (or support) some activity within Requirements Engineering. The term 'intelligent' implies that the tools incorporate a knowledge-base consisting of

- knowledge about how to perform some aspect of Requirements Engineering (e.g. elicitation, specification, validation) and/or

- knowledge about the characteristics of some problem domain (called *domain knowledge*) which can be employed in Requirements Engineering.

The knowledge-based paradigm does not necessarily imply the use of a software development model different from the ones discussed previously. Thus, there can be knowledge-based Requirements Engineering approaches which adopt any of the waterfall, prototyping, operational etc. models. Therefore, the major differences between knowledge-based and non-knowledge-based approaches exist in the degree of intelligent tool usage in a process within Requirements Engineering. For instance, requirements validation is conventionally performed by letting the user inspect the requirements model (which can be a piece of text, diagrams, prototype etc.). If, however, validation is performed by checking the requirements model for consistency against rules (which state when a model is consistent) stored in a knowledge-base, then validation becomes a knowledge-based approach.

Knowledge-based approaches to requirements validation are discussed in Chapter 5. More general, knowledge-based tools which assist the process of Requirements Engineering are examined in Chapter 6.

2.6.7 Requirements Engineering according to the Domain Analysis model

The Domain Analysis paradigm [Arango, 1988] is the only one introduced so far which departs from the assumption that Requirements Engineering (and software development in general) is a 'one-of' activity. More specifically, the paradigm realises the existence of similarity between applications belonging to the same problem domain and advocates that the analysis results from one application can be re-applied to the analysis of a similar one.

Domain Analysis has been viewed as an activity that takes place prior to Requirements Engineering [Hall, 1991]. While Requirements Engineering is concerned with analysing and specifying the problem of developing a software application, domain analysis has been concerned with identifying commonalties between different applications under the same domain. The deliverable of domain analysis is a set of objects relations and rules that are common in a problem domain and thus can be reused across different applications. For instance, software applications which deal with airline ticket reservations all consider a standard set of objects such as *passenger, flight, reservation, ticket* etc. Domain Analysis suggests that concepts such as the above can be abstracted and organised in libraries so that they can be reused 'off-the shelf' in future applications. In this respect, domain analysis radically changes our understanding of Requirements Engineering for the following reasons

- phases such as problem understanding which are traditionally considered in Requirements Engineering are reduced to 'selecting and retrieving the contents of the appropriate library which contains the analysis results of the domain under consideration'

- the effort for requirements elicitation is significantly reduced since to a large extent, elicitation has already be done as part of Domain Analysis

- requirements specification consists of selection of an appropriate component from the reusable analysis components library with possible adaptation if necessary

- finally, the need for validation is reduced since the library components have been already validated as part of Domain Analysis.

Domain Analysis is considered again in Chapter 3 for its impact on requirements elicitation. Figure 2.7 shows the impact on domain analysis on Requirements Engineering.

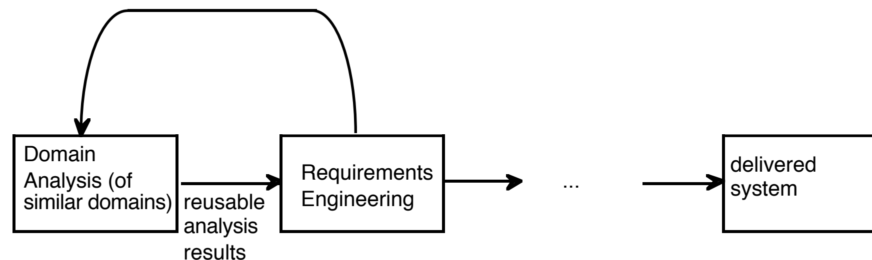


Figure 2.7: The Domain Analysis Approach

2.8 Managing the Requirements Engineering Processes and their Deliverables

The previous sections described Requirements Engineering as the set of intertwined processes of elicitation, specification and validation. It is important to emphasise that the deliverables of Requirements Engineering are in a state of *flux*, during this process, and may even remain so through subsequent stages of development such as design and coding. A formal software specification is the end-product of a large number of decisions, negotiations and assumptions made throughout the Requirements Engineering process, and, as such, a specification is as valid as the assumptions and decisions which underlie it. It is therefore important to be able to recreate the *rationale* behind some specification item in order to question its appropriateness and validity in the light of changed circumstances. However, this is not possible without assistance from a *rationale recording* process which runs throughout Requirements Engineering. Such process is beneficial for the following reasons:

- In an explicit form, the rationale behind a system requirement provides a communication mechanism amongst the members of the development team so that during later stages of development such as design and maintenance it is possible to understand what critical decisions were made during requirements

specification, what were the alternative options to a particular specification and why this particular one was selected over the other alternatives.

- The effort required to produce the rationale behind a specification forces Requirements Engineers to deliberate more carefully about their decisions. The process of deliberation can be assisted by explicitly showing all the arguments in favour or against a particular specification decision.

There are several possible ways of capturing the rationale of specifications in a model. Amongst them the most widely used is based on a model called *IBIS* (Issue-Based Information System). IBIS was developed in the 1970s for representing design and planning dialogues. Graphical versions of it such as *gIBIS* have been used for documenting the rationale behind software design decisions [Conklin and Begeman 1989].

Figure 2.8 shows the basic concepts of the IBIS model. An *issue* is the root of the model (e.g. a user need for which a software solution is required). There are also secondary issues (*sub*-issues) which modify the root issue in some way. A *position* is put forth as a potential resolution for the issue (i.e a *position* represents a software specification which satisifies the user need). There can be more than one alternative positions (specifications) associated with an issue. In turn, each position is related to arguments which support it and arguments which object to it. Such arguments might refer to technical, financial etc. criteria and constraints under which the merits of each alternative specification will be judged. Such repertoire of modelling constructs allows IBIS models to be used during requirements analysis and specification sessions as a means of recording the issues deliberated and the decisions made.

In conclusion, rationale capturing models such as IBIS can provide an effective answer to the problem of managing and evolving the products and by-products of the Requirements Engineering process.

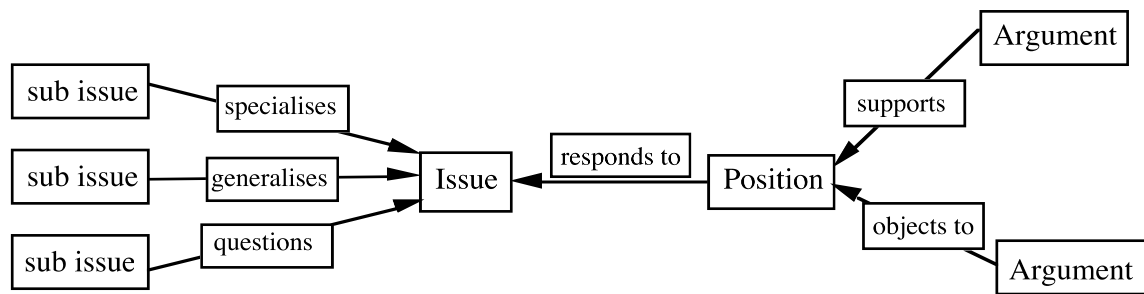


Figure 2.8: The IBIS model

Summary

Reviewers of the current Software Engineering literature arrive soon to the conclusion that today no consensus exists as to what constitutes Software Requirements Engineering, its scope, stages, aims and deliverables. The problem is caused partially by the fact that Requirements Engineering has only recently been acknowledged as a discipline of software development in its own right. The lack of consensus is also caused by the varying degrees of rigorour and formality by which Requirements Engineering is treated in various software development methods. Finally, the lack of consensus in defining Requirements Engineering lies in the fact that systems analysis is an inherently ill-defined and ill-structured process. For all these reasons, contemporary software development methods prefer to describe Requirements Engineering by its products rather by its processes, as proved by the plethora of modelling formalisms in use.

This Chapter, aimed at a look of Requirements Engineering from a dynamic perspective (i.e. looking at *how* Requirements Engineering is done) rather than a static one (i.e. what Requirements Engineering *produces*). In doing so, it abstracted from current proposals for organising Requirements Engineering into processes, stages, steps and so on, to produce a framework of three fundamental processes within Requirements Engineering, namely elicitation, specification and validation.

Requirements Elicitation was defined as the process of acquiring all the necessary knowledge which is used in the production of the formal requirements specification model.

Requirements Specification was defined as the process which receives as input the deliverables of requirements elicitation in order to create a formal model of the requirements (called the requirements specification model).

Requirements Validation, finally was defined as the process which attempts to certify that the produced formal requirements model satisfies the users needs.

The above processes were examined in the context of different software development paradigms, since different paradigms usually put more emphasis on one or another of the processes. In general, recent software paradigms such as prototyping and operational approach, pay more attention to validation of requirements by constructing a formal object (prototype, operational specification) with which the user can experiment , early in the development life cycle.

It can be said that all the software development models consider the major processes of Requirements Engineering, namely elicitation, specification and validation . It is also true that different approaches tend to underestimate or overemphasise one or more processes of Requirements Engineering. Traditionally, the less emphasised and considered process of Requirements Engineering has been *validation*. This however has proved to have catastrophic consequences for software projects. It is only natural therefore, that more recent proposed models such as the prototype model, the operational model etc. put more emphasis on requirements validation by the user. Also the latest approaches to Requirements Engineering attempt to rectify the fact that the process still remains the most labour intensive one in software development. The approaches taken towards Requirements Engineering automation belong to two broad categories. Approaches in the first category attempt to automate fairly trivial but nonetheless labour intensive tasks such as document preparation. The second category includes approaches which attempt to automate (fully or partially) activities in Requirements Engineering such as specification and validation by using support from knowledge-based software tools. This category contains also approaches which advocate the improvement of productivity in Requirements Engineering by reuse of results from previous analysis activities. Automation of Requirements Engineering processes will become the key issue in the software trends of the near future.

References

Alavi, M. (1984) An Assessment of the Prototyping Approach to Information Systems Development. Communications of the ACM, 27 (6).

Arango, G. (1988) Domain Engineering for Software Reuse. PhD thesis, Department of Computer Science, University of California, Irvine.

Balzer, R., Cheatham, T. E., Green, C. (1983) *Software Technology in the 1990's: Using a New Paradigm*. IEEE COMPUTER, November.

Balzer et al (1988). *RADC System/Software Requirements Engineering Testbed Research and Development Program.* Report TR-88-75, Rome Air Development Center, Griffiss Air Force Base, N.Y., June.

Berzins, V., and Gray, M. (1985) *Analysis and design in MSG 84: Formalising Functional Specifications*. IEEE Trans. on Software Engineering 11 (8) August.

Boehm, B. W. (1988) *A Spiral Model of Software Development and Enhancement*. IEEE COMPUTER Vol. 21 No 5, May .

Boehm, B. W. & Papaccio, P. N. (1988) *Understanding and Controlling Software Costs*. IEEE Trans. on Soft. Eng., Vol. 14, No. 10, Oct.

Castano, S. and De Antonellis, V. (1993) *Reuse of Conceptual Requirement Specifications*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, San Diego, California, 1993, pp. 121-124.

Conklin, J. and Begeman, M.L. (1989), *gIBIS:A tool for all reasons* Journal of the American Society for Information Science, March 1989.

Davis, G. B. **(1982)** *Strategies for Information Requirements Determination*. IBM Systems Journal, 21 (1).

Davis, A. M. (1990) Software Requirements Analysis & Specification. Prentice-Hall, NJ.

DoD (Department of Defence.) (1988) *Military Standard:Defense System Software Development*. *DOD-STD-2167A*. Washington, D.C., February.

Fickas, S. (1987) *Automating the Analysis Process: An Example*. Proc. 4th Int'l Workshop on Software Specification and Design, IEEE.

Floyd, C. (1984) *A Systematic look at prototyping*. In Approaches to Prototyping. Budde, R. ed., Springer-Verlag.

Fugini, M.G. and Pernici, B. (1992) *Specification Reuse*, in 'Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development', P. Loucopoulos &. R. Zicari (ed.), John Wiley & Sons, New York, pp. 535 - 548.

Hall, P. A. V. (1991) *Domain Analysis*. Proc. UNICOM Seminars, London, December.

IEEE (1984) IEEE Guide to Software Requirements Specifications. The Institute of Electrical and Electronics Engineers.IEEE/ANSI Standard 830-1984. New York, 1984.

IEEE (1990) IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers. Std. 610.12-1990.

Jackson, M. (1982) Systems Development. Prentice Hall.

Martin. C. (1988) User-Centered requirements Analysis. Englewood Cliffs. N.J.: Prentice-Hall.

McCracken, D. and Jackson, M. (1982) *Life Cycle concept considered harmful*. ACM SIGSOFT Soft. Eng. Notes, vol. 7, No. 2, April.

Powers, M, Adams, D and Mills, H. (1984) Computer Information Systems Development: Analysis and Design. Cincinnati:South-Western.

Pressman, R. (1988) Software Engineering: A Practitioner's approach, 2nd Ed. New York. McGraw-Hill.

Roman, G.-C. et al (1984) *A Total System Design Framework*. IEEE COMPUTER, 17 (5).

Rombach, H D (1990) *Software Specification: A Framework.* Curriculum Module SEI-CM-11-2.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January.

Royce, W. W. (1970) *Managing the Development of large software systems: Concepts and Techniques* In Proc. WESCON, August.

Wasserman, A. et al. (1986) *Developing Interactive Information Systems with the USer Software Engineering Methodology*. IEEE Trans. on Software Engineering 12, 2.

Yadav, S. et al. (1988) *Comparison of Analysis Techniques for Information Requirements determination.* Communications of the ACM 31, 9. September.

Yeh, R. & Ng P, A. (1990) *Software Requirements-A management perspective*. In System and Software Requirements Engineering. R. H. Thayer & M. Dorfman (eds) IEEE Computer Society Press.

Zave, P. (1984) *The Operational Versus the Conventional Approach to Software Development*. Communications of the ACM, Vol. 27, No. 2, February.

CHAPTER 5

Validating Requirements

Introduction

Chapter 4 discussed a number of modelling formalisms which can be used for creating a functional requirements model. The use of a single formalism or a combination of them, for example entities/relationships, rules, state transition diagrams etc., will result in a model of what the analyst perceives to be the user requirements for a software system. With careful checking, possibly with some help from an automated CASE tool (such as those discussed in Chapter 6), the analyst can become fairly convinced that the requirements model is a *correct* one, where the term *correct* means that the model will not contain illogical or self-contradicting definitions.

There is a problem with the above approach however. *A correct requirements model is not necessarily the right requirements model*. What has been thought to be the user's problem can sometimes be something totally irrelevant, i.e. a situation can occur where time and effort is spent analysing the *wrong problem*. Such a situation is illustrated in Figure 5.1.

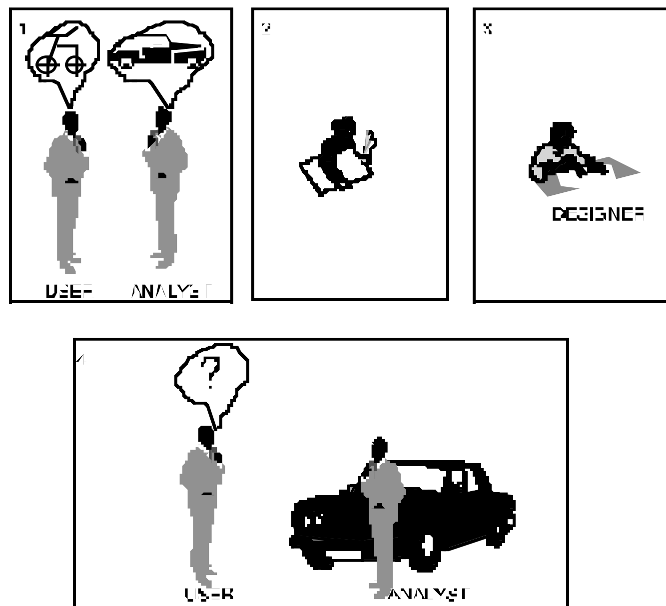


Figure 5.1: Solving the wrong problem.

Chapter 3 identified the main causes for such misconceptions to be:

- difficulty in eliciting the requirements from a user. (In Expert System terminology this is called the *knowledge acquisition bottleneck*)

- difficulty in establishing a common framework of understanding between the analyst and the user, i.e. *making them speak the same language.*

Obviously, wasting precious time and resources in solving the wrong problems is something ill-afforded. It is therefore important to make sure at a fairly early stage that the analysed problem is indeed the user's problem and that the resulting requirements model will be a faithful representation of the user's demand for a computer solution to his/her problem.

`Requirements validation is the process of certifying the requirements model for correctness against the user's intention`

As such requirements validation helps to *do the right thing* in contrast with the careful following of a modelling approach which helps in *doing the thing right.*

Section 5.1 provides more arguments in favour of an early validation of requirements which will avoid correcting expensive mistakes later on in the software life cycle. Section 2

gives guidelines on issues that should be considered when validating the requirements model. Section 4 introduces techniques for requirements validation, namely prototyping, simulation/animation, the use of natural language in validation, and finally expert system approaches.

Prototyping provides the users with a mock-up version of the software system as a means of acquiring their correct requirements.

Simulation and animation are techniques for 'bringing life' into the requirements model, taking it through different states and generally testing it with under different scenaria in order to see how well it copes in real-life like situations.

Natural language paraphrasing is a technique for translating the requirements model in the user's own language (i.e. natural language) in hope that this will make it easier for the user to judge the validity of the requirements.

Expert system approaches to validation introduce the concept of an *assistant* , i.e. a system containing knowledge about the problem domain which it uses in order to bring to the attention of the human analyst contradictions, omissions, unresolved issues etc. about the requirements model.

5.1 The need for requirements validation

This section stresses the importance of an early validation of the requirements model. Also, validation is distinguished from the more usually practised *verification*. It must be emphasised that in contrast to verification, validation cannot performed by the analyst and software tools alone; the user's active participation is always necessary.

The IEEE SET Glossary [IEEE, 1983] defines validation as the process of evaluating software at the end of the software development process to ensure compliance with software requirements. The same glossary defines verification as "... The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase...".

Some attention must be paid to the above definition. First, validation is defined as taking place only after the software has been developed. Second, verification is always carried

with respect to some requirements. The questions raised as a result of these definitions are as follows:

- is it sufficient to validate software only after it has been developed?
- against what can requirements be verified, i.e. what are the 'requirements for the requirements model'?

Statistics have proved the answer to question 1 to be 'no'. A price has to be paid be paid if software validation is left until late. The longer software validation is postponed for, the more expensive (in terms of testing, debugging and redevelopment) it becomes to correct mistakes . In a survey carried by Boehm in the 70s [Boehm, 1980] it was revealed that up to 50% of the bugs in a software systems were results of errors in the requirements. Moreover, the cost of fixing a bug which was a result of an error in requirements definition was up to 5 times the cost of errors in designing or coding. Actually, such findings are justifiable, considering that most software systems are developed today using a 'waterfall' lifecycle approach (see Chapter 2), and the reasons are as follows. Suppose that an error occurs during the requirements specification phase. This will result in some part of the requirements model to be erroneous and the rest part to be correct. As the development of software progresses, some part of the design will be based in part on the erroneous requirements and in part on the correct requirements. Such design *will not* be partially wrong; it will be completely wrong. Similarly, code which will be derived partially in erroneous and partially in correct design will be wrong. This 'snowball' effect can have catastrophic consequences. A single error in the requirements specification may cause the need to re-design, re-code and re-test a large part of the software system Therefore:

`Validating at the requirements specification phase can help to avoid fixing expensive bugs after the software has been developed.`

In a similar manner, the answer to the second question asked at the beginning of this discussion, ('is there some other model against which the requirements model can be verified?'), is, again, 'no'. There is no such a thing as the requirements' requirements. Whilst code can be verified against the design and design itself can be proved correct with respect to the requirements, there is no way we can formally verify the later. If such a thing as a 'requirements' requirements' model existed it would had been inside the users' heads. In reality however, users do not know what they want, neither (sometimes) what is best for them, nor (unless they are computer experts themselves!) what is feasible. This leads to

another conclusion:

```
It cannot be proved formally that a requirements model is correct.
```

What can, however be achieved instead is a well-justified *belief* that the solution specified in the requirements model is the right one for the user. This can only be achieved by the set of techniques known as requirements validation which can be now defined as

```
The processes which establish and justify our (and the user's) belief
that the requirements model specify a software solution which is right
for the user's needs.
```

5.2 Guidelines on what to validate in a requirements model

Requirements validation should be thought as more of a "debugging" activity rather than a "proving correct" one. The aim of requirements validation, is not to prove the requirements correct; but instead to identify and correct all the errors (inconsistencies, omissions, incorrect information), now rather than later on when the software will be designed and coded. In Chapter 3 validation was defined to be a process which proceeds in parallel with other processes of Requirements Engineering i.e. elicitation and specification. Indeed, validation is an 'ever present' activity which takes place every time a piece of requirement is acquired, analysed or integrated with the rest of the requirements model . Validation of requirements is also mainly an *unstructured* process. This means that it is not possible to apply an algorithmic procedure which will obtain a validated requirements model. However, there is a set of tasks that are performed during validation which apply equally well to all methodologies and formalisms used to construct the requirements model. The purpose of these tasks is to identify the existence of a number of desirable properties in the requirements model, namely:

- internal consistency
- non-ambiguity
- external consistency

- minimality
- completeness
- redundancy

The above properties of requirements models are discussed in the sequel.

Internal Consistency

A requirements model is *internally consistent* if no contradicting conclusions can be derived from it. Consider as an example a requirements model for a payroll application. If some requirement in the model states that "employees who earn less than 10000 should not be taxed" and at the same time a different requirement states that "for all employees tax should be deducted from their salaries" then the model is inconsistent. It must be noticed here that it is not necessary for both contradicting requirements to be explicit in the model; one (or both of them) may be implied from other requirements. This of course makes the validation task difficult since it is not always possible to check all the implications of the requirements model.

Faced with the problems of checking consistency, some researchers have suggested that a requirements model should be specified using *mathematical logic* [Dubois et al, 1987]. In logic we can view requirements as *logic sentences* which can be either *true* or *false*. Complex requirements statements can be constructed from more simple ones using logic connectors such as *and*, *or, not* and *implies*. An example of a complex requirement formed from two simple ones is as follows:

"a book is available for borrowing"
"a book is out of the library"
"a book is available for borrowing" *or* "a book is out of the library".

Being able to manipulate the requirements in a logical way when possible, has a number of advantages. A *theorem prover* is a program which can infer new logic sentences by combining existing ones, using rules of *inference*. A theorem prover applied to a logic requirements model can help the analyst discover certain types of contradictions.

Non-ambiguity

Non-ambiguity is another important characteristic for which a requirements model should

be checked. Non-ambiguity means that a requirement cannot be interpreted in more than one way. Unfortunately, ambiguities can be easily introduced in a requirements model, especially when natural language is used. The following example will show the potential disastrous effect that ambiguous requirements can have. Imagine a model of requirements for a software controlled furnace. One particular requirement states

> *when the furnace temperature reaches 200 ^{o}C or the environment temperature falls below 5 ^{o}C and the water valve is turned on then the oil valve must be turned on.*

The above requirements statement is ambiguous because it can be interpreted in two different ways, i.e. as

> *when the furnace temperature reaches 200 ^{o}C and the water valve is turned on, the oil valve should be turned on. When the environment temperature falls below 5 ^{o}C and the water valve is turned on then the oil valve should be turned on*
>
> or
>
> *when the furnace temperature reaches 200 ^{o}C the oil valve must be turned on. When the environment temperature falls below 5 ^{o}C and the water valve is turned on, the oil valve must be turned on.*

Whilst there seems to be little differences between the two interpretations, in fact the second one is wrong and if implemented in a software system could have devastating consequences. According to the second interpretation, the oil should start flowing in the furnace when its temperature reaches 200°C. This however, ignores the requirement that the furnace's temperature must be controlled by a cooling mechanism. According to the second interpretation, this cooling system would never come into operation, allowing a possible overheating and explosion of the furnace.

The above is only one possible example of the risks brought by ambiguous requirements. On of the most important jobs of the analyst, is therefore to eliminate ambiguities from the requirements model using his common sense, rules of logic, by clarifying ambiguous specifications in conjunction with the users.

External Consistency

External Consistency is the agreement between what is stated in the requirements model

and what is true in the problem domain. In a study by the Naval Research Laboratory, documented in [Basili and Weiss, 1981] it was revealed that 77% of all the requirement errors for the A-7E aircraft's flight program were non-clerical. Forty nine percent (49%) of these errors were incorrect facts. Every effort must be therefore put on ensuring that what is said in the requirements model is in accordance with the problem domain. Most of the facts appearing in the requirements model will come from interviewing the user's or studying the literature about the problem domain etc. (See Chapter 3). If the facts about the domain are volatile (i.e. change frequently-something that usually happens when for example the software is going to be embedded in a new piece of hardware, yet to be developed) then every attempt should be made to keep the facts in the requirements model up to date.

Minimality

Minimality is an important feature which should be present in our requirements document. The opposite of minimality is called *over specification* which is simply the tendency to include more in the requirements model that it is necessary. Usually, overspecification is an attempt to prescribe a design solution at the same time as we specify the requirement. This is, however, wrong because it constrains the possible solutions which the designers can choose from. Example of overspecifying the problem of a heating control system is given below.

When the furnace temperature exceeds 200 ^{o}C the master controller module should send a request of the type to the valve controller. The valve controller should use a first-in-first out queue to store the requests it receives from the master controller.

The above statement belongs more to a design rather than to a requirements document as it introduces concepts (modules, queues etc.) and solutions which are not essential to specify the heating control problem. In general, anything that should limit the designer's choices, unnecessarily, should not be included in the requirements module.

Completeness

A requirements model is complete when it does not omit essential information about the problem domain which could result into a system not meeting the user's needs. Completeness is a difficult thing to check in a requirements model, since there is no formal

procedure to do so. A requirements model contains goals to be achieved in a problem domain, as well as rules, facts and constraints that apply to the domain and to the software system that will be operating in the domain. Failure to capture any of these ingredients of a requirements model will have an impact on the correctness of the model.

- The omission of an objective, for example, will result in a model that will not represent all the user needs.

- Failure to capture a rule or fact of the modelled domain, will prohibit the software system from being a correct model of that domain.

- The absence of a constraint from the requirements model can lead to incorrect behaviour of the software system.

Some types of completeness checking can be performed fairly easily, and even be automated using a tool (See Chapter 6). These are usually checks for definition of all the names that appear in the requirements model. If, for example process *xyz* is mentioned in the requirements model, then the tool should be able to check whether the process is described somewhere in the model or not..

The most difficult kinds of completeness checking, i.e. checking for missing goals, facts or constraints can be assisted in a number of ways such as:

- by looking at systems with similar requirements to the one being analysed , we can identify "forgotten" requirements (goals).

- by assuming a hierarchical organisation of the requirements, we can identify requirements which do not have any corresponding "higher level" ones. If the justification for including some particular requirement is missing, then this missing justification could be some other, forgotten, requirement

A good, general way to check for completeness in the requirements model is to use *prototyping* (described in Section 5.4.1). By prototyping the requirements model, obvious omissions (as well of course as inconsistencies, ambiguities etc. can be found by the analyst and the participating users).

Redundancy

A requirement is redundant if it can also be obtained from some other part of the requirements model, or if it is simply some property or behaviour not wanted in the software system. Multiple definitions of requirements are not desirable features of a requirements model, since ideally, a requirement should be identifiable in one and only one place in the requirements model. Care, however, is required when a requirement is *implied* from other stated requirements. If a requirement is implied (in a logical sense) from other requirements and, at the same time, is stated explicitly, then one of the following conditions apply:

- the explicit statement of the requirement can be removed, as long as there is no fear that the implied requirement will be ignored

- the explicit requirement can remain in the document together with a reference to the requirements by which it is implied. If any of those requirements is subsequently removed, the analyst should re-examine the status of the implied requirement.

Deciding on whether a requirement is redundant or not is an activity which requires an understanding of the users' expectations from the system (sometimes called a "wish list"), the feasibility of the proposed requirements and also assigning priorities to requirements. A way to do so is assigning to each requirement a value from a range, starting from "absolutely essential" through "a nice thing to have in your system" down to "redundant".

5.3 Resources needed for requirements validation

It is fair to describe validation as the *quality assurance* process which can be applied to the requirements model. Quality is a sought after property of every computer artefact e.g. design, code, documentation etc. The need for quality brings forward two issues of practical importance, i.e. the time and resources spent on requirements validation and the quality criteria which can be applied to a requirements model.

In an ideal world, the validation process should take as long as it is required. The requirements model should be checked thoroughly by the analyst (in conjunction with the

users and software tools when necessary) for all the points we mentioned before (omissions, inconsistencies, violation of constraints etc.). The model's behaviour should be exhibited using techniques such as prototyping, animation etc.) and the model should be modified until it corresponds to the user's expectations. Additions, deletions or modifications of requirements should be checked for their impact on the rest of the requirements model.

In the real world, however, the situation is different. Software projects have to meet budgets and deadlines. Requirements validation takes up only a small percentage of the overall project lifecycle and it cannot possibly last forever. Moreover there is usually a tendency amongst the developers to get over with requirements specification in order to move to more 'challenging' issues such as design. Finally, the users do not always have all the time and willingness in the world to collaborate with the analysts in endless requirements validations (inspections, walkthroughs) sessions.

The analyst is thus coming under time and resource constraints to perform a daunting and critical job such as requirements validation. As assistants to the validation task, the analyst can employ a number of tools and practices, i.e:

- logical organisation of the requirements model which show the interdependencies amongst the requirements

- automated tools which help avoiding clerical errors and speed up some aspects of validation

- communication with the user in any opportunity using techniques such as simulation and prototyping

- bringing in experience of analysis of similar systems; reuse of previous requirements models.

Naturally, the duration and effort spent on requirements validation will vary with the modelled application. Also tools and techniques for validation may appear to be more suitable for some kinds of applications than for others. Nevertheless some sort of formal validation procedure is beneficial for all types of applications, since "getting the requirements right" is the single most important step towards a successful software project.

5.4 Techniques for validating requirements

5.4.1 Prototyping

Prototyping (Chapter 2) is the term used for describing the process of constructing and evaluating working models of a system, in order to learn about certain aspects of the required system and/or its potential solution. Prototyping is a common technique in engineering disciplines (e.g. aeronautics) where, a scaled-down model of the artefact (aeroplane, car etc.) under production is first created and used for experimentation in order to arrive at a production model.

In software engineering, prototyping (also called *rapid prototyping*) has been advocated as the paradigm of producing software quickly and cheaply as possible at some stage of the development. The actual uses of prototype vary, depending on the phase of the lifecycle in which it is used, as well as on the particular life-cycle model followed. The software prototype can be throw-away (used only for understanding and assessing solutions, performances, risks etc.) or it can be transformed into the final production system [Balzer et al, 1983]. In order for a model (of any artefact, including software) to be characterised as a prototype, the following must be attainable:

- it must be possible to obtain information about the behaviour and performance of the production system from the prototype. To provide such information, prototypes should be capable of being fully instrumented; the result of such instrumentation is that the execution of a prototype results in the generation of data from which needed information can be inferred.

- prototyping should be a *quick* process. In a well-supported environment, producing a working prototype should take significantly less time and effort than it would take to produce a full-scale artefact.

Use of Prototyping in Requirements Validation

In Chapter 2, the process of Requirements Engineering was said to have a lifecycle, consisting of three major phases, namely those of *elicitation, specification* and *validation*. Prototyping can assist in all three phases of Requirements Engineering. It has already been discussed (Chapter 3) how prototyping techniques can help in acquiring (eliciting) and

formalising the requirements. This Section is more concerned with the application of prototyping to validation, i.e. *to the process of certifying the requirements model for correctness against the user's intention.*

Since, by necessity, validation is a process in which users is very heavily involved, the prototype model must provide the user with meaningful information. There are two kinds of such information, namely *behavioural* and *structural*. A behavioural prototype models what the prototyped software system is supposed to do. In this respect, a behavioural prototype is a black-box model of the software system which exhibits responses to stimuli. It can be said that a behavioural prototype models the *functional requirements* (see Chapter 4) of the software. In contrast, a structural prototype models how the system being prototype will accomplish its black-box behaviour. A structural prototype is thus a 'glass-box' model which exhibits aspects of the internal structure and organisation of the system being prototyped, and in this respect it can be said that a structural prototype models the *non-functional requirements* (see Chapter 4) of software.

Techniques for prototyping

A requirements model can be created using a variety of different languages and formalisms as discussed in Chapter 4. However, in order to be used for prototyping, the modelling formalism (language) must exhibit a number of properties as follows.

- it must allow the inference of information 'hidden' in the requirements model

- it must have the notion of 'state' as well as 'state transition'.

The first property (inferencing) essentially means that not only someone must be able to look at what is stated in the requirements model, but also to examine the implications of such statements. One way to make this possible is to use some *logic* language together with an inference mechanism to create the prototype. A very suitable language for this purpose is *Prolog* and an extensive literature on prototyping using Prolog exists now [Budde, 1984]. The major advantage of Prolog is that (by virtue of being a programming language) allows the execution of the requirements specification. As a Very High Level Language (VHLL) Prolog permits incompleteness in the specifications and does not force the use of strong typing or procedural control structures. In addition, it is possible to transform the initial requirements prototype into an efficient production system, without having to leave Prolog.

Amongst the disadvantages of the Prolog (or more generally, Logic programming) approach to prototyping is that

- there is no graphical notation associated with it, and
- users do not always find logic specifications easy to understand

The first problem can be overcome by using some graphical formalism (e.g. an entity-relationship model) as the front-end and having Prolog translating E-R definitions to equivalent Prolog structures in the background.

The second problem can be overcome by paraphrasing Prolog definitions using some sort of 'IF THEN ELSE' rules and pseudo-English.

The following example (taken from a library specification problem) illustrates the use of Prolog in prototyping requirements specifications.

Part of the natural language requirements model for the library case is as follows:

> In a university library there are two kinds of users, namely staff and students. Staff can perform two kinds of activities, namely check out a book for a student who wants to borrow it and check-in a book returned by a student. Once a book is checked out it is not available for borrowing until it is checked-in again. A student cannot have more than ten books borrowed at any time.

The above requirements specification translated to Prolog looks as follows.

```
user(X) :- staff(X)or student(X).

check_out(Book, Borrower) :- student(Borrower)and not checked_out(Book) and
 not borrowing_limit_exceeded(Borrower) and assert(checked_out(Book))  and
update_borrowing_record(Borrower, borrowed).

check_in(Book, Borrower) :- student(Borrower) and book(Book) and retract(checked_out(Book)) and
update_borrowing_record(Borrower, returned).

update_borrowing_record(Borrower, borrowed) :- borrowing_record(Borrower, Number_of_Books) and
New_Number_of_Books is Number_of_Books + 1 and
retract(borrowing_record(Borrower, Number_of_Books) and
assert(borrowing_record(Borrower, New_Number_of_Books)).

update_borrowing_record(Borrower, returned) :- borrowing_record(Borrower, Number_of_Books) and
New_Number_of_Books is Number_of_Books - 1 and
retract(borrowing_record(Borrower, Number_of_Books) and
assert(borrowing_record(Borrower, New_Number_of_Books)).

borrowing_limit_exceeded(Borrower):- borrowing_record(Borrower, Number_of_Books) and
Number_of_Books = 10.
```

It can be noticed in the above Prolog requirements specification follows closely the English specification, making at the same time some additional statements which are usually taken for granted in the natural language specification (such as, for example, that if a person wants to check out something then that person must be a student and the thing to check out must be a book).

Despite looking like a conventional program, the above Prolog specification conforms to the quality criteria set for a requirements specification; it describes the problem domain, rather than the solution domain. As it is, the above specification says nothing about design decisions, e.g. decisions about the choice of data structures to hold the borrowing record of students. However, by virtue of its executability, the above specification provides the second necessary ingredient for prototyping, namely *states* and *state transition.*

In order to create a state in the above Prolog specification, a number of *facts* must be defined. Facts are supposed to represent a (hypothetical) state in the real library, as follows.

```
student(student1).
student(student2).
book(book1).
book(book2).
{stating that the individual students 'student1', 'student2', as well as the individual books 'book1', 'book2'
exist in the library}
checked_out(book2). {stating that book 'book2' has been checked out'}
borrowing_record(student1, 0).
borrowing_record(student2, 1). {stating the borrowing records of students}
```

The above library state can be transformed to another state by invoking one of the rules describing the activities of checking in and out in the library. If for example, rule 'check_in' is invoked as follows

```
check_in(student2, book2).
```

The new state of the library will be as follows.

```
student(student1).
student(student2).

book(book1).
book(book2).

borrowing_record(student1, 0).
borrowing_record(student2, 0).
```

In this new state, 'book2' is no longer borrowed and 'student2' borrowing record has been amended to show that the student has currently borrowed no books. In a similar way, by invoking other rules the present state can be transformed to a number of other allowable states.

The value of this approach lies on the fact that users are asked to validate something which is more close to their real experience than a static requirements document. Users are more capable of detecting anomalies (in terms of inconsistencies, omissions etc.) when they have a hands-on experience with a dynamic executable requirements model. As an example, consider the above library state in which 'student1' is trying to check-in 'book1' (i.e. a book which is not borrowed!). If the execution of such request (i.e. check_in(student1, book1)) is refused by Prolog then this is obviously a positive evidence for the quality of the specification, since a correct specification should not allow the creation of meaningless library states. On the other hand, the execution of the above request by Prolog would mean that there are problems with the specification, since obviously the later does not model the reality in a realistic manner. A possible remedy to this problem would be to redefine the rule describing the activity of 'checking-in'. A precondition could be added to the rule stating that in order for a book to be checked-in it must be currently in a 'checked-out' state, as follows.

```
check_in(Book, Borrower) :- student(Borrower) and book(Book)
and checked_out(Book) /*the precondition*/
 and retract(checked_out(Book)) and update_borrowing_record(Borrower, returned).
```

It is up to the analyst to devise a number of suitable test cases which can lead to the identification of flaws in the requirements model.

Apart from running test cases, another prototyping technique known as *static validation* can be used with Prolog requirements specifications. With static validation, the specification is analysed in order to check a number of possible flaws such as missing definitions. In the above case study for example, it could be discovered that the 'check in' activity can never take place because there is no corresponding rule for the 'check-out' activity defined. Static validation can save the user from going through too many test cases, but as dynamic validation is not a sufficient technique on its own. Here it must be noted that static analysis is also supported by many CASE tools for a variety of specification formalisms (data flows, Entity/Relationship models etc.). Such tools will be also discussed in Chapter 6.

In summary, prototyping is a technique whose importance in requirements validation cannot be easily dismissed. However, whether prototyping is achievable at all depends on the modelling formalism and supporting environment used. .

5.4.2 Animation

Animation is a technique which can be effectively brought into use for the validation of real-time systems. Real time systems have a common set of notions such as *processes*, *process synchronisation, messages* and *timers*. An important concern in validating real-time systems is that the requirements for concurrent and time-constrained activities are correctly captured. With the advent of new technology such as personal high-speed workstations with high resolution display devices, the graphical representation of concepts such as processes and the dynamic display of their behaviour has become possible.

Animation is a multiple graphical view of a process in action. In an animation environment, the analyst is given the ability to depict graphically all the major objects of the requirements model (processes, timers) and interact with them (using a mouse or pointing device) in terms of messages. For example, the user can change the state of a process from active to suspended and see the effects of that on the rest of the system. The objects have usually some way of graphically showing the state they are in (e.g. by using colour) and in addition numerical information is available on the screen. By animating the processes, the analyst can identify problems of performance and also detect situations such as deadlocks, starvation etc.

In the context of information systems development, a prototype specifications animation approach described in [Lalioti and Loucopoulos, 1994] aims to provide a visual environment for validating and symbolically executing conceptual specifications (in terms of entity, process and rule models) of an information system. The term *conceptual prototype* is used by this approach to emphasise the difference between executing the specifications and creating a prototype of the information system. In conceptual prototyping there is no need to make any design decisions prematurely. By animating the conceptual prototype, all the actors of the Requirements Engineering process can understand and inspect the behaviour of the system under development as early as possible in the Requirements Engineering process.

Visualisation has been applied successfully in programming environments in order to provide an indication of the behaviour of the program. In the context of conceptual specifications, visualisation involves the animation of the behaviour of a system and a visual interface reflecting the results of events upon the graphical - and where appropriate the textual - components of the specification.

The advantage of visualisation over prototyping is that design decisions will not have to be made prematurely during Requirements Engineering when things are still vague. A requirements specification is likely to change many times before proceeding to design and visualisation should help in deriving a succession of specification.

Experiences from the use of visual environments in programming tasks has encouraged researchers in Requirements Engineering to make use of similar techniques, normally referred to as animation techniques, in assisting the activity of validating conceptual specifications [Kramer and Ng 1988; Tsalgatidou 1988].

Animation of a specification is the process of providing an indication of the *dynamic* behaviour of the system by walking through a specification fragment in order to follow some scenario. Animation can be used to determine causal relationships embedded in the specification or simply as a means of browsing through the specification to ensure adequacy and accuracy by reflection of the specified behaviour back to the user.

5.4.3 Natural Language paraphrasing

The technique of natural language paraphrasing has being devised in order to tackle the problem caused by two conflicting concerns in Requirements Engineering, namely the concern of the analyst to develop a formal requirements model, and the users' need to communicate their requirements in their own terminology. Paraphrasing is a technique which compromises the two concerns by providing a 'user-friendly' version of a formal requirements model. Since paraphrasing sometimes summarises parts of the requirements model it can come to the aid of not only the user but the analyst as well, who can see the specification from a new perspective.

As with prototyping, paraphrasing does not need to be applied to the whole requirements model, as it can instead focus on those parts of the model which need clarification and reconsideration by the user. Naturally, paraphrasing should be an automated activity so that it does not occupy more valuable time than necessary. Such an automatic paraphrasing tool has been developed by a research project [Myers and Johnson, 1988] and is targeting specifications of the language *Gist*. A typical *Gist* specification together with its English paraphrase produced by the tool is shown in Figure 5.2.

Gist specification	*English paraphrase*
type driver	
subtype of person	All cars are mobile-objects. All drivers
type car	are people.
subtype of mobile-object	Each car can be in the state of running.
WITH{relation running().	
invariant in-motion() =>	
self: physical-object-location isa location	
PROCEDURE start[]	Start is a procedure of car.
DEFINITION	Start a car simultaneously
atomic {	(atomically) does the following. It asserts
insert in-motion().	that the car is in-motion. It updates the
update self: physical-object-location to a location	physical-object-location of the car
}.	to any location.

Figure 5.2: An English Paraphrase of the Car Specification.

5.4.4 Expert System Approaches

This category of approaches includes a number of automated (CASE) prototype tools which assist the validation of requirements. What justifies the characterisation of such tools as 'expert systems' is the knowledge of some aspect of Requirements Engineering process that they embody. This can be either *method knowledge*, i.e. knowledge of how to apply a method for Requirements Engineering (e.g. Structured Analysis- see [Yourdon, 1989]), or *domain knowledge*, i.e. knowledge about the domain which the software system is supposed to model (see Chapter 3). Expert approaches are still at the prototype status but they are certainly expected to have an impact on the functionality of the next generation of commercial CASE tools.

In general, there are three modes at which an expert system tool can act in the requirements validation process, namely

- the *expert*. At this mode the tool is expected to carry the validation process on its own
- the *assistant*. At this mode the tool is asked to assist the analyst validate some

aspect of the requirements, i.e. by checking for some specific kind of inconsistency in the model

- the *apprentice*. At this mode the tool is asked to perform the low-level clerical activities of validation, i.e. keeping track of all the names used in the requirements model, checking for multiple definitions and so on.

The current generation CASE tools function at the level of the apprentice. However, the next generation of CASE tools is expected to show performance at the level of the *assistant* (see also Chapter 6). Below is a scenario showing a typical interaction between a human analyst and a requirements validation assistant tool.

The following is the scenario for the analysis of the library system introduced in Section 5.4.2. In the session the human analyst text is in normal font, the tool's output is in **bold** and the comments are in italic.

The Analyst has created a first-cut mode of the problem using the Entity-Relationship analysis method and is asking the Assistant tool to perform certain types of validation

<ANALYST>:	Validate model LIBRARY
<ASSISTANT>:	**Select one of the following types of validation:** **Syntactic validation** **Consistency with Domain Knowledge**
<ANALYST>:	Syntactic validation
<ASSISTANT>:	**I notice that the cardinality of relationship BORROWS from STUDENT to BOOK has not been defined You can do one of the following:** **Define the cardinality of the relationship** **Consult KB "libraries" for typical cardinality of relationship BORROWS**

At this stage the analyst has the option to supply the missing cardinality or be advised by the library domain knowledge-base on what the cardinality should be like.

<ANALYST>:	Consult KB "libraries"
<ASSISTANT>:	**The typical cardinality of relationship BORROWS is 0 TO MANY. Action?**
<ANALYST>	…

Some of the typical prototype tools belonging to the category of expert assistants are *ASPIS* [Pietri et al, 1987] the *Requirements Apprentice* [Reubenstein and Waters, 1991] *KATE* [Fickas, 1987] and the *Analyst Assist* [Loucopoulos and Champion, 1988]. All these systems help not only in validating the requirements but also in tasks such as requirements acquisition, formal specification (see also Chapter 6).

Summary

This chapter tackled the important issue of validating the requirements model. Important

ideas discussed in this chapter include the following

- validation is a process whose importance cannot be easily dismissed. Errors that pass through the Requirements Engineering phase to design and coding can have catastrophic consequences on the software system

- validation of requirements can only be performed against the user's intention; thus the user's participation in the validation process is paramount

- requirements can never been proved correct in a formal manner. They can however be checked for qualities such as consistency minimality, non-redundancy etc.

- validation is a time-consuming process that can be greatly assisted by automated tools

- one of the biggest problems in requirements validation is getting the user understand the formal models created by the analyst. A number of techniques to overcome this problem are *prototyping*, *simulation/animation* and *paraphrasing*

- all the above techniques presuppose the use of suitable modelling languages and environments in which the requirements can come closer to the user's experience through execution, animation and simulation and thus be easier validated

- expert tools are expected to play a significant part in the near future in validating requirements by drawing upon method and domain knowledge.

References

Balzer, R., Cheatham, T. E., Green, C. (1983) *Software Technology in the 1990's: Using a New Paradigm*. IEEE COMPUTER, November.

Basili, V. and Weiss, D. (1981) *Evaluation of a Software Requirements Document by Analysis of Change Data*. In Fifth IEEE Int’l Conference on Software

Engineering, Washington D.C. IEEE Computer Society Press.

Boehm, B. W. (1980) Software Engineering Economics. Englewood Cliffs. N.J:Prentice4-Hall.

Budde, R. (1984) (ed). Approaches to Prototyping. Springer-Verlag, Berlin, 1984.

Dubois, E. & Hagelstein, J. (1987) *Reasoning on Formal Requirements: A Lift Control System*. Proc. 4th Int'l Workshop on Software Specification and Design, IEEE.

Fickas, S. (1987)*Automating the Analysis Process: An Example*. In Proc. 4th Int'l Workshop on Software Specification and Design, Monterey, CA. IEEE.

IEEE (1983) IEEE Standard 729. *Glossary of Software Engineering Terminology*. IEEE New York.

Kramer, J. and Ng, K. (1988) *Animation of Requirements Specification*, SPE, Vol. 18, No. 8, 1988, pp. 749-774.

Lalioti, V. & P. Loucopoulos (1994) *Visualisation of Conceptual Specifications*. Information Systems, Vol. 19, No. 3, pp. 291-309

Loucopoulos, P. & Champion, R. (1988) *A Knowledge-Based Approach to requirements Engineering Using Method and Domain Knowledge*. Journal of Knowledge-based Systems, June.

Myers, J. J. & Johnson, W. L. (1988) *Toward Specification Explanation: Issues and Lessons*. Proc. 3rd Annual Rome Air Development Center Knowledge-Based Software Assistant Conference, Utica, NY.

Pietri, F., Puncello, P. P. , Torrigiani, P., Casale, G., Innocenti, M. D., Ferrari, G., Pacini, G., Turini, F. (1987) *ASPIS: A knowledge-based environment for software development*. In ESPRIT '87: Achievements and Impact, North-Holland.

Reubenstein, H. B. & Waters, R. C. (1991) *The Requirements Apprentice: Automated Assistance for Requirements Acquisition*. IEEE Trans. on Software

Engineering, Vol. 17, No. 3.

Tsalgatidou, A. (1988) Dynamics of Information Systems Modelling and Verification, PhD thesis, UMIST, Manchester, UK, 1988.

Yourdon, E. (1989) *Modern Structured Analysis.* Prentice-Hall.

CHAPTER 3

Requirements Elicitation

Introduction

In chapter 2 Requirements Engineering was said to consist of three major processes namely *requirements elicitation, requirements formalisation* and *requirements validation.*

The first of these processes, requirements elicitation, is defined as

```
the process of acquiring (eliciting) all the relevant knowledge
needed to produce a requirements model of a problem domain
```

The above definition implies that requirements elicitation is all about 'understanding' some particular problem domain. Only after understanding the nature, features and boundaries of a problem can the analyst proceed with a formal statement of the problem (requirements

specification) and subsequently with its validation by the user (requirements validation). The following example, makes more apparent the need for a thorough understanding of the problem domain, before formal specification is attempted. The example is about the specification of a radar and tracker system for aviation.

> The system shall accept radar messages from a short-range radar. The scan-period of the radar is 4 seconds. The frequency is 2.6-2.7 Ghz. The pulse-repetition interval frequency is 1040 Hz. The number of tracks shall be for 200 aircraft. The band-rate is 2400. The message-size is 104 bits/message. The system shall begin tracking aircraft that are within 2 miles of the controlled area. Track initiation will occur after 6 seconds...

Even in such a small fraction of requirements like the above, the radar-specific terminology can be overwhelming for the analyst who is not familiar with the domain. Moreover, it is impossible for the unfamiliar analyst to test the above specification for things such as consistency and completeness. For instance, there is a possible conflict in the above requirements, between the 2 mile margin of controlled area and the distance of 4 miles that can be covered by an aircraft travelling at 600 mph for 24 seconds (which is the initiation time of the tracker). The necessity of understanding radar technology is beyond doubt in this example. This leads however to a different sort of question: *Where can such (domain) knowledge be found and how can it be elicited?*

In the case of the radar system above, an obvious solution is to have the knowledge supplied to the analyst by the radar (electronics) engineers who are developing the non-software components of the tracker system. They will be able to explain all the domain specific concepts to the analyst, who will in addition be expected to have a basic understanding of radar technology, as well as mathematical skills. Indeed, as real-life shows with the all the different specialisations of software engineers that exist today (e.g. commercial systems, telecommunications, real-time systems engineers etc.) it would probably take an analyst with significant experience in the field, to produce a trusted specification for the tracker system above.

Naturally, it is impossible for an analyst to acquire experience in more than a handful of different categories of applications in the span of a life-time. Moreover, there exist software applications which are 'one of a kind', i.e. knowledge about them cannot be acquired either from other similar ones or from textbooks. Nevertheless, for common or 'one-off' applications the task of the analyst is to

```
elicit the knowledge about some problem domain and to some extent
become an 'expert' about the domain
```

Coad and Yourdon [1991] argue for example that the involvement with the domain of the analyst developing a system for air traffic control must be so close as to result in nuances to be discovered which even the experts in air traffic control have not yet fully considered.

This Chapter is concerned with the problem of *domain knowledge transfer* from some source (i.e. human, book or any other type of source) to the analyst. Knowledge transfer is classed as a problem for the following reasons

- the knowledge is not always readily available in a form that can be used by the analyst and
- it is difficult for the analyst to elicit the knowledge from its source, especially when the source is a human 'expert'.

This Chapter discusses methods and techniques for eliciting knowledge from some problem domain. It starts with the discussion of the most obvious source of requirements, i.e. the application domain expert. Section 3.2 discusses approaches which view the requirements model as a set of goals that must be achieved, activities that must be performed to achieve the goals, and constraints which restrict the activities that can take place. The idea is intuitive, because in a way, the whole software system can be seen as serving a purpose within some larger system (e.g. an office, factory etc.). It is only natural then, that the functioning of the software system must be guided by goals, which are set by the host system. The principle of viewing requirements as goals has many different variations and has been even used for the modelling of nonfunctional requirements (Chapter 4).

Another requirements elicitation technique discussed is that of *scenario-based elicitation* (Section 3.3). Again, this technique belongs to the more broad category of *prototyping* techniques which are presented in Chapter 5. Under this technique, the users are participating in executing scenaria that mirror problem solving in real-life situations and in such way that their expertise (which constitutes part of the problem domain knowledge) is elicited.

Form Analysis (Section 3.4) is another elicitation technique, which concentrates on knowledge that can be extracted from the various documents (forms) used in the problem domain rather than from humans. This technique is effective in dealing with data intensive software applications. In contrast, natural language-based knowledge elicitation approaches (Section 3.5) rely not on formal documentation about the problem domain, but on more easily available natural language descriptions either in the form of text, or as direct input from the user.

Section 3.6 discusses a family of elicitation techniques which are based on the idea of reusing existing requirements specifications. This is based on the premise that:

> *There are commonalties between different applications belonging to the same category. Thus eliciting requirements from scratch each time we want to analyse a new application is like re-inventing the wheel. In many situations it is feasible (and very cost effective) to reuse requirements from similar old applications into new applications.*

In Section 3.7 a different view of requirements elicitation as a *social* process is presented. The basic premise of these approaches is that the problems of requirements elicitation cannot be solved in a purely technological way, because the social context is crucial in this phase, of the development process, much more so that subsequent phases such as design and programming.

Section 3.8 gives a comparative of two related disciplines (which have grown independently from each other), namely *requirements elicitation* and *knowledge elicitation.* Knowledge elicitation is the process of extracting knowledge from a human expert with the purpose of encoding it in a *knowledge-based expert system.* There appear to be knowledge elicitation techniques that can be applied to requirements elicitation (and vice-versa) and therefore of potential benefit to a requirements analysis professional.

3.1 Requirements Elicitation from Users

Elicitation of requirements from users working in the application domain is the most intuitive of the elicitation approaches since it is the users who should 'know what they

want' from the planned software system. In practice, however, elicitation from users presents difficulties for the following reasons:

- users may not have a clear idea of their requirements from the new software system
- users may find it difficult to describe their knowledge (expertise) about the problem domain
- the backgrounds and aims of the users and analysts differ; users employ their own domain-oriented terminology whilst analysts use a computer-oriented vocabulary
- users may dislike the idea of having to use a new (unknown) software system and thus be unwilling to participate in the elicitation process.

To overcome problems such as these, a number of techniques have been devised which enable the communication between the analyst and the user and thus the transfer of knowledge from the latter to the former.

The easiest interaction to conceive between analyst and user is called *open ended interview* [Graham and Jones, 1988]. The analyst simply allows the user to talk about his or her task. The lack of formality in the interview makes for a relaxed atmosphere which facilitates the flow of information from the user to the analyst. Open interviews are more appropriate for obtaining a global view of the problem domain and for eliciting general requirements. However, such techniques are inadequate for eliciting detail information requirements or for describing user tasks in detail. The reason for this is psychological as uncaused recall is often incomplete and unstructured. For the elicitation of more detailed requirements, therefore, methodical approaches described in the sequel are used. *Structured interviewing techniques* [Edwards, 1987] direct the user into specific issues of requirements which must be elicited. In structured interviewing techniques the analyst employs *closed*, *open*, *probing* and *leading* questions in order to overcome the elicitation problems discussed above. Using structured interviews, a great deal of information is acquired and used to:

- fill gaps in domain knowledge acquired so far

- resolve obstacles such as lack of consensus amongst the users
- achieve a better support of the environment.

Another technique used to overcome the problem of lack of consensus amongst the users is called *brainstorming collective decision-making approach* (BCDA) [Telem, 1988]. BCDA combines brainstorming and collective decision-making in order to help the analysts understand the problem domain. Brainstorming tackles the problem discussed above, i.e. the difficulty users experience in describing their own expertise. On the other hand, collective decision making reduces the problem of lack of consensus with respect to the goals, priorities etc. that different users set for the software system. In addition, BCDA has the positive effect that it helps users to understand information technology and analysts to understand organisational needs.

In summary, interviewing techniques are the most straightforward techniques for software elicitation. They require however special skills from the analyst since these techniques are most sensitive and delicate ones. These techniques also suffer from a number of problems such as the limited amount of time that users may be available for interviews, psychological difficulties in eliciting user expertise etc.

3.2 Objective and Goal Analysis

This category of requirements elicitation approaches is concerned with questions frequently occurring at the start of a software project, such as

"Why does this organisation need what its staff have expressed in their requirements statements?"

or

"Do they really want what they are stating?"

Questions like the above, re-enforce what has been emphasised in the beginning of this Chapter: "Only after understanding the nature, features and boundaries of a problem can

the analyst proceed with a formal statement of the problem...".The activity and goal approaches therefore,

- attempt to place the requirements (problem) in a wider context
- understand how the problem relates to ultimate problems and objectives of the larger system which will be hosting the software system, and
- in short, attempt to 'get the right requirements'.

Objective and goal analysis approaches are based on a set of key concepts such as *objectives*, *goals*, and *constraints* which will be defined in the sequel.

3.2.1 Concepts of Objective and Goal Analysis

Fundamental to the following discussion, is an understanding of the concept of *teleological view of systems*. According to the teleological view, a system (such as an organisation, machine, human etc.) has a set of goals which it seeks to attain. Thus, the teleological view attempts to explain a system's behaviour in terms of its goals. A *goal* is defined as a defined state of the system. Since a state is described in terms of the values of a number of parameters, a goal can be alternatively defined as a set of desired values for a number of parameters. For instance, if the system is a (profit making) organisation then one of its goals can be

```
To make profits of $1M in the next financial year.
```

Here, the goal parameter is "profits" and the desired value is "$1M".

Goals can vary in their degree of specificity (or else abstraction). In general, the more desired values are mentioned, the more specific the goal is. Thus, the goal

```
To make profits of $1M in the next financial year.
```

is more specific (less abstract) than the goal

```
To make profits  in the next financial year.
```

The varying degree of specificity (abstraction) in goals, has a lot to do with the hierarchical way most human-purpose systems are organised. In a large organisation, for example, there can be many levels of management. The job of the senior management (the executives) is to make decisions on the general strategy of the organisation. This however, makes their goals necessarily more abstract than the goals in lower levels of the decision hierarchy. If for example, the senior management decides that 'the organisation must be profitable in the next financial year', it is up to the middle management to specify how profitable the organisation will have to be and how this can be achieved. At the next level of seniority (operational management) the goals will be with regard to the tactics and procedures which will ensure the profitability of the organisation (Figure 3.1).

Figure 3.1: Levels of abstraction in an organisation's goals.

Sometimes, goals which are more abstract (vague) are called *objectives*. Objectives do not usually specify 'when', 'how much' or 'how'. An objective could, for instance, state 'The organisation must strive for profitability' without specifying how this profitability will be measured or when it must be attained. Usually happens, an objective is decomposed into a number of more specific ones (which are therefore goals). There are two different kinds of decomposition which can be applied to the objective. An objective *Ob* can be decomposed to a conjunctive set of goals *G1, G2, ..., Gn* . The meaning of the *AND* decomposition is that in order for objective *Ob* to be attained, *all* goals *G1, G2, ..., Gn* must be attained. The other kind of objective decomposition is an *OR* decomposition. If objective *Ob* is or-decomposed into goals *G1, G2, ..., Gn*, then for objective *Ob to be attained* it is sufficient that *any* of *G1, g2, .., Gn* is attained.

The following example shows both AND and OR decompositions. In order for objective

```
Increase profits
```

to be attained, any of the following goals must be attained

```
Increase sales, reduce production cost
```

In order for goal 'reduce production cost' to be attained, *all* of the following must be attained

```
reduce cost of raw material, reduce cost of machinery, increase
productivity, reduce staff cost
```

The above decomposition of objectives to goals can continue to many different levels of abstraction, creating a *goal hierarchy* (or according to some authors a *goal-subgoal* or *goal-means* hierarchy). Usually, the goals that appear at the lower levels of the hierarchy are called *subgoals* (or *means*) since they represent specific ways with which a goal can be attained. If for example, the goal 'increase productivity' can be achieved by 'install automatic production system XYZ' then the later can be called a subgoal, or means towards realising the former.

In many situations, a subgoal may be instrumental to more than one (super)goals, thus the goal hierarchy (actually a *lattice*) looks similar to Figure 3.2.

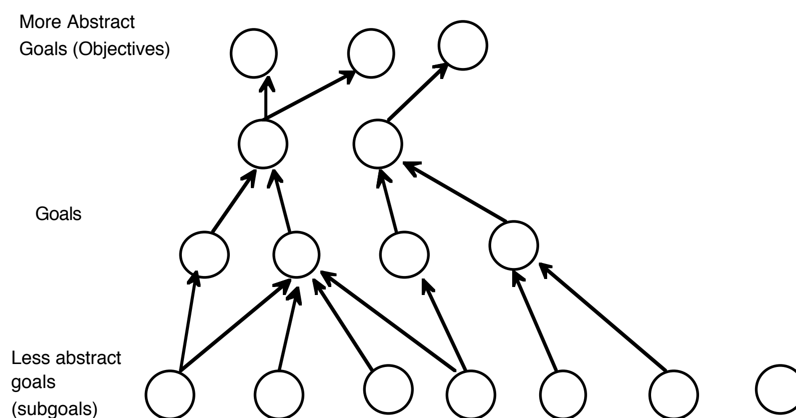


Figure 3.2. Goal Hierarchies.

Apart from the goal-subgoal relation (which is inter-level), there exist intra-level relations which must be considered when modelling a goal-subgoal hierarchy. Two goals appearing

in the same level of the hierarchy can be *mutually supportive* or *mutually conflicting*. Mutually supportive are those goals which affect positively the attainment of each other. Mutually conflicting goals affect negatively the attainment of each other. In contrast, the goals 'increase automation' and 'reduce investment in new machinery) conflict each other since automation implies the acquisition of new machinery.

Another concept occurring frequently in objective-goal analysis is that of a *constraint*. A constraint prohibits the full attainment of some objective/goal. Constraints may originate within the system (e.g. in an organisation, physical operations, personnel structure, finance etc. can act as constraints), from the environment (e.g. customers, competitors, laws, government regulations) and so on. When the system under discussion is software, then more constraints to its development can be limitations of the current technology, constraints imposed by the host system etc.

In summary, objective/goal analysis approaches view the problem domain as consisting of objectives, goals, subgoals(means) and objectives, organised into a goal-subgoal (ends-means) hierarchy. The use of the goal/subgoal hierarchy is discussed in the sequel.

3.2.2 Steps in Objective/Goal Analysis

The purpose of constructing the goal hierarchy in the Objective/Goal Analysis approach is first to identify the software requirements in the context of the problem domain, i.e. the larger system which will become the host of the software system, and second to map software requirements to (higher level) system objectives. Obviously, not the whole of the organisation's goal hierarchy will be relevant to the requirements for the software system. The first step therefore, is to select that portion of the goal hierarchy which is relevant to the software requirements specification. Such portion of goal/subgoal/constraint definitions will consist of the following:

- a hierarchy of objectives/goals/constraints which are directly relevant to the information processing system of the organisation and which will consist at the lower levels of

- means towards their realisation, i.e. requirements for the software system.

For instance, if the goal 'increase productivity', is refined through many levels of goals/subgoals to 'automate task XYZ', then the latter is an expression of a software requirement. The question that arises now is whether the above requirement is a valid and justifiable one. To answer such question, we must re-examine the goal hierarchy, paying particular emphasis to cases of conflicting goals. When cases of conflicting objectives/goals occur, some consensus must be reached about the goal structure and its refinement into subgoals (tasks). The ultimate goal of this exercise is to arrive at a consensus amongst the stakeholders (the parties who have interest in the software system under development). During this process all sets of alternatives should be evaluated. If for instance, the objective 'increase productivity' can be attained by either 'automate task XYZ' or 'automate task PQR', then the alternative which is less promising to meet the objective must be eliminated. Repeat of this exercise will arrive at a complete set of requirements which can be directly attributed to valid organisational objectives and which are also associated with organisational/environmental or technical constraints.

Further analysis of the requirement will yield all the detailed information needed for recording in the requirements specification model.

In summary, the steps of objective/goal analysis are as follows:

- analyse organisation and the external environment with which it interacts in terms of objectives, goals, constraints
- create goal-subgoal hierarchy consisting of organisational objectives, goals and constraints and their interrelationships (support, conflict, constraint)
- validate the model and achieve a consensus amongst the stakeholders about it
- identify the portion of the goal/subgoal hierarchy modelling the information processing part of the organisation
- eliminate cases of conflicts in the above model by negotiating/bargaining etc. with stakeholders
- select tasks (requirements) by eliminating alternatives.

Objective/goal analysis approaches tackle the problem of eliciting requirements successfully for the following reasons.

- the analysts have a clear understanding of the problem domain including what belongs to the software system and what belongs to the host system
- by placing the requirements problem in its wider context, the danger that users will be so overwhelmed by short term problems that they loose track of the long term objectives is reduced
- A number of potential solutions (which otherwise would be lost) can be considered and comparatively evaluated.

Goal-oriented analysis has been used in the contexts of Artificial Intelligence and Cognitive Science. The main proposals for applying the approach to requirements modelling appear in [Karakostas, 1990], and also in [Mittermier et al, 1990] and [Bubenko and Wangler 1993].

3.3 Scenario-Based Requirements Elicitation

Approaches under this category rely on the strength of scenarios as an (almost) universal form for the organisation and dissemination of experience. In the most general sense, a *scenario* is a story that illustrates how a perceived system will satisfy a user's needs. Scenarios are important instruments for creating social meaning and a shared sense of participation [Crowley, 1982], i.e. elements needed in a process such as requirements elicitation.

More specifically, during a requirements elicitation session, a scenario represents an idealised but detailed description of a specific instance of a human-computer interaction. Scenarios can use flexible media, close to the end-user's conceptualisation of the system, such as text, pictures or diagrams. Also they can be structured in various ways such as dialogues or narrative descriptions.

There is a close relation between scenarios and *prototypes*. Prototypes (which are discussed in detail in Chapter 5) are mock-up versions of the software system. The difference between scenarios and prototypes lies in the fact that the latter are more general than the former. A scenario deals with only one instance of human-computer interaction which is supposed to be typical for the expected use of the future software system. In contrast, a prototype mimics more than one instance and type of interaction between the user and the software system under development. This can be better explained in the example of Figure 3.3.

Consider a university library which has a computerised system for checking books in and out. A check-out scenario for a book is as follows. A student arrives at the assistance desk with a book to be checked out. The library assistant asks the student for his/her student card which contains the student's id. The assistant enters the id in the following screen • The assistant checks the response to see if the borrower's privileges are restricted for any reason. If not, the book's id is entered on the screen. • After the id is entered the book's title and the due date for the loan are displayed on the screen. • The assistant enters a 'Y' at the 'OK' prompt and at that point the volume is on loan to the student.

Figure 3.3: A University Library System

The above scenario is supposed to represent a fictitious but realistic human-computer interaction in the library system. Because it is realistic, the scenario allows the elicitation of expertise from the user. The library assistant for example, will be in a position to criticise the above scenario for its lack of realism, much more easily than it would have been with the case of a formal requirements model. The library assistant could for example recall that

> 'When I am checking-out a book for a student, I always check if that student has any overdue books, in which case I remind the student about it, by showing the book titles and due-back dates.'

The analyst understands such recalled experience as a missing requirements statement. More specifically, the analyst notes that

- books which are overdue (defining overdue as the due date being after 'today') must be flagged as such and

- all overdue books for a student must be displayed on the screen in a checkout session.

The analyst can proceed with other similar scenaria which will elicit more tacit knowledge from the library assistant and help towards completing and refining the requirements model. Other useful scenarios for instance are:

```
The student who wants to check out a book does not have his
student card with him. However the student is at the same time
checking in a book he borrowed previously. Can the information
from the previous checkout be used for the new checkout?

A student wants to check out a book which according to the records
has already been checked out.  What happens in such case?
```

It is obviously up to the analyst to select the most appropriate scenarios, bearing in mind that usually the user's time for participation is limited. Also, scenarios should be used to clarify issues and implications of a requirement when there is no other way to do so. If for instance, the library procedures are clearly written down there is no need to waste the user's time in long interaction sessions. Still, however, the scenarios technique is invaluable in cases where a large part of the requirements is concerned with the user interface. There is no better way of understanding the interaction requirements than giving the user a hands-on experience with the software!

In summary, the scenario techniques for requirements elicitation are based on the principle that users find it easier to transfer their expertise to the analyst through an active 'story telling' session, rather than through questionnaires and interviews. Together with prototyping techniques (discussed in Chapter 5), scenario techniques present a promising solution to the difficult problem of communication and transfer of expertise that usually exist between the analyst and the user. Scenario-based techniques for requirements elicitation are documented in [Hooper, and Hsia, 1982] [Holbrook, 1990] [Karakostas and Loucopoulos, 1989] [Loucopoulos and Karakostas, 1989]. Today, many CASE tools (see Chapter 6) provide the ability to develop a sequence of screen layouts along with a background of narrative illustrating their use.

3.4 Form Analysis

In contrast to scenario-based requirements elicitation approaches, form analysis approaches do not regard the user as the prime source of knowledge about the problem domain. They instead rely on a communication object very widely used in organisations, namely *forms*. A form is any structured collection of variables which are appropriately formatted to support data entry and retrieval. A form is a promising source of knowledge about a domain for the following reasons:

- it is a formal model and thus less ambiguous, and inconsistent than equivalent knowledge expressed in natural language
- a form is a data model, thus it can provide the basis for developing the structural component of a functional model
- important information about organisations is usually available in forms
- the acquisition of forms is easy since they are the most commonplace object in the organisation
- the instructions which normally accompany the forms provide an additional source of domain knowledge
- forms analysis can be easier automated than analysis of other sources of requirements knowledge such as text, drawings etc.

The most common use of forms is as an input to the process of constructing an entity-relationship [Chen, 1976] model. An entity-relationship model consists of the following modelling constructs

- entities, which are objects of interest in the problem domain
- relationships which are meaningful associations amongst entities, and
- attributes which are properties of entities.

The following example will clarify the concepts of form, entity, relationship and attribute.

A sales order form contains information about a sales transaction. The information contained in such forms are usually about

- the sale's order number
- the date of the sale
- the number of the corresponding customer order
- the name and address of the customer that raised the order.
- The name and address of the customer where the order is to be sent (billing address)
- The names, prices per unit, quantities and amounts of the products sold with the order
- The total (before tax), tax and total after tax value of the order.
- The 'id' of the salesperson that prepared the sales order.

In the above description of the information appearing in a sales order a number of concepts, such as 'order number', 'date of sale', 'customer' and so on, can be found. Some of these concepts can be used to model entities, relationships or attributes in an E-R model which describes the problem domain of sales orders. Mapping the concepts 'hidden' in a form to appropriate constructs of an E-R model is actually the task of the form-analysis approaches. In general, there are no clear cut rules which state what can correspond to an entity, relationship etc. Different approaches therefore apply their own tactics in order to overcome the lack of formal rules:

- Some approaches apply manual methods to extract the E-R constructs from a form, i.e. they rely on the analyst's judgement and experience

- others however automate the analysis process by using heuristic rules to match forms contents with entity-relationship constructs.

Naturally, automated approaches to form analysis are more appealing than manual ones since they reduce the analyst's overhead as well as the number of possible errors. One of the most well known automatic form analysis approaches [Choobineh et al, 1988] uses three kinds of heuristic rules, namely *entity identification rules*, *attribute attachment rules* and *relation identification rules.* The following are example of such rules:

Entity identification rule:

Any form field which is the source of another form field, whether of this form or another is a possible entity

Application of this rule to the sales order example will yield CUSTOMER ORDER as an entity since the value of the CUSTOMER ORDER# field comes from another form (not shown above) namely CUSTOMER ORDER

Attribute Attachment Rule:

Any field which has a small proximity factor to a field which is 'discovered' as an entity is probably an attribute of that entity. (The proximity factor of a field is defined as the difference between the position of the field and the position of another field that has been 'discovered' as an attribute of the entity). This rule captures the observation that the attributes of an entity appear physically close in a form.

Application of the above rule would yield for example that TAX is an attribute of the ORDER entity, after its physically close TOTAL BEFORE TAX has been found to be an attribute of the same entity.

Relationship Identification Rules

Relationship identification rules are more complicated than those for entity identification and attribute attachment. Applications of such rules (which are beyond the scope of this book) would yield for example that entity ORDER is related to entity SALESPERSON by relationship PREPARES

Application of rules such the above mentioned would result in the creation of an E-R model such as the one shown in Figure 3.4 The resulting schema could be checked automatically for consistency (e.g. for things like entities having the same name). Form contents which cannot be automatically analysed must be considered by the analyst who decides about their roles in the E-R model. Finally the user would be presented with the completed E-R model for the task of validating it.

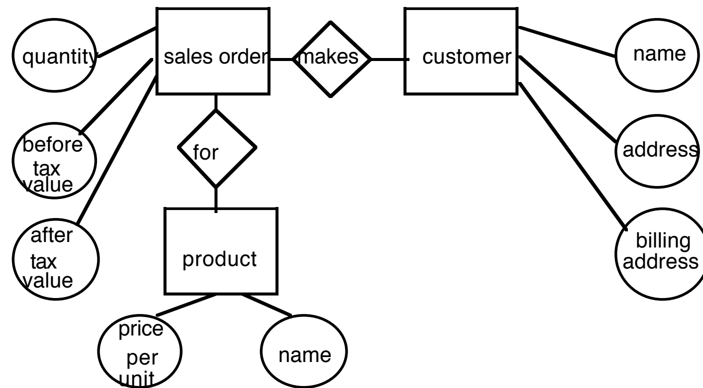


Figure 3.4: An entity-relationship model of a sales form contents.

In conclusion, forms are useful sources of problem domain knowledge which can be effectively used in the process of requirements elicitation. Although, form analysis approaches are limited to data intensive software applications their effectiveness in eliciting domain knowledge, in particular when they are used with an expert tool cannot be underestimated.

3.5 Natural Language Approaches

The elicitation techniques described so far are based on a diversity of problem domain knowledge sources such as users (either as groups or as individuals) and forms. It is true however, that for the majority of the domains the most common knowledge representation medium is natural language. The attractiveness of eliciting requirements from natural language (NL) descriptions lies in the fact that in most cases everything that is worth known about the problem domain can be stated (or is somewhere written) in NL. Thus NL elicitation approaches fall into two categories:

- approaches which directly interact with the user in NL in order to elicit the requirements from the user, and

- approaches which elicit the requirements from NL text.

Three things make requirements statement in NL attractive: *vocabulary, informality* and *syntax*. Indeed, the existent vocabulary of thousands of predefined words used to describe any possible concept makes natural language an efficient communications medium.

Informality (i.e. the possibility that a statement is ambiguous, incomplete, contradictory and/or inaccurate) is very important also. Whilst informality is not a desired feature of the final requirements specifications document, it is very useful in early phases of Requirements Engineering as a means of dealing with complexity. As a matter of fact, in everyday situations it is informality that comes with NL which allows us to communicate without been bogged down by detail. Syntax, finally is a useful feature of NL because it is familiar and thus requires no time for learning it.

Features of NL such the above have made the 'programming without programmers' a dream in the first decades of computing. Soon, however, it was realised that the promising idea of automatic generation of software from user requirements is not feasible in the vast majority of situations. Today, the focus of research is on powerful formal specification languages and the use of knowledge (see knowledge-based tools for Requirements Engineering in Chapter 6) rather than on the pursuit of the 'NL specification' dream. However, NL descriptions of the problem domain has been proved an efficient source from which knowledge can be elicited. The state-of-the-art research approaches today consider only descriptions in a subset of NL, from which they can derive a formal requirements model. Other manual approaches elicit the knowledge from NL text by applying a number of heuristics and 'rules of thump'.

The automated approaches to natural language analysis are facing the problem of the enormous richness and variety of expressions that can be stated in NL. Since unrestricted NL understanding is still an unresolved problem of Artificial Intelligence, such approaches necessarily restrict the acceptable input to only a small subset of NL. A NL approach called OICSI [Rolland and Proix, 1992] for example, uses so called *cases* of NL sentences. A *case* is a type of relationship that groups of words have with the verb in any clause of a sentence. There are nine major cases, considered by the OICSI approach, namely *OWNER, OWNED, ACTOR, TARGET, CONSTRAINED, CONSTRAINT, LOCALISATION, ACTION, OBJECT*. The meaning of these cases is exemplified by the following set of sentences

In the sentence

```
A subscriber is described by a name, an address and a number
```

`'subscriber'` is associated to the OWNER case, and `'name'`, `'address'`, `'number'` are associated to the OWNED case.

In the sentence

```
A subscriber borrows books
```

'subscriber' is associated to the ACTOR case and the OWNER case, while 'books' is associated to the OWNED case. The entire clause is associated to the ACTION case.

In the sentence

```
When a subscriber makes a request for a loan, the request is
accepted if a copy of the request book is available, else the
request is delayed.
```

the clause '`When a subscriber makes a request for a loan`' is associated to the LOCALISATION case. Inside this clause, the phrase '`request for a loan`' is associated to an OBJECT case. The clause 'if a copy of the requested book is available' is associated to the CONSTRAINT case. The clause '`the request is accepted`' is associated to the ACTION and the CONSTRAINED case. Inside this clause, the word '`request`' is associated to the TARGET case.

The above cases are mapped to the constructs of a requirements modelling formalism used by the REMORA methodology [Rolland and Richard, 1982], according to a set of mapping rules. The constructs used in REMORA are *entity*, *actions*, *events* and *constraints*. For example, cases of type OWNER, OWNED, ACTOR, TARGET, OBJECT are mapped to entities of the requirements model, cases of type LOCALISATION are mapped to events, and so on.

As a result, in the above sentences, 'subscriber' would be mapped to an entity type, the clause `'When a subscriber makes a request for a loan'` would be mapped to an event etc.

The OICSI environment provides also the facility of creating a *paraphrase* of the generated conceptual model which can be used for its validation (Validation and the paraphrasing technique are thoroughly discussed in Chapter 5).

Other automated requirements elicitation approaches which use NL as input include SECSI [Bouzegoub and Gardarin, 1986] and ACME [Kersten et al, 1986]. However, current automated approaches have limited applicability since they can accept only a small subset of NL as input and create requirements models only in a few formalisms.

Manual approaches to NL requirements elicitation, on the other hand, are more flexible because they rely on the superior NL understanding capabilities of humans.

Such approaches, analyse NL descriptions in order to identify constructs (verbs, nouns, adjectives etc.) which will map to constructs of a requirements modelling formalism, according to some rules. NL analysis is a technique favoured by a category of requirements specification approaches called *object-oriented* (Chapter 4). For the sake of this discussion it will suffice to say that object-oriented approaches consider the following constructs

- *objects* which are entities of interest appearing in the problem domain
- *attributes* of objects i.e. characteristic properties of objects, and
- *operations* which are actions performed or suffered by the objects.

A sample strategy for identifying objects, attributes and operation is given below. Similar (more elaborate) strategies are proposed by authors of object-oriented analysis methods such as [Booch, 1986]. The strategy is as follows.

- objects are determined by looking at the NL descriptions for nouns or noun clauses
- attributes of objects are identified by identifying all adjectives and then associating them with their respective objects (nouns)

- operations are determined by underlying all verbs, verb phrases and predicates and relating each operation to the appropriate object

To illustrate the use of the above rules, consider the NL description of the requirements for a radar and tracker system, first listed in the Introduction of this Chapter.

The system shall accept radar messages from a short-range radar. The scan-period of the radar is 4 seconds. The frequency is 2.6-2.7 Ghz. The pulse-repetition interval frequency is 1040 Hz. The number of tracks shall be for 200 aircraft. The band-rate is 2400. The message-size is 104 bits/message. The system shall begin tracking aircraft that are within 2 miles of the controlled area. Track initiation will occur after 6 seconds...

After scanning the above description for nouns, the following objects are identified:

```
(tracker) system
radar
aircraft
(controlled) area
```

The second step identifies attributes (properties) of the above object. Most of the attributes are identified by looking at nouns and clauses that qualify the above found objects. For example 'message' is a property of radar. 'short-range' is a qualification of radar, and is further analysed to 'range' (which is the radar's attribute) and 'short' which is the value of it. Other attribute-value pairs for the radar are:

```
(scan period, 4 seconds), (frequency, 2.6-2.7 Ghz), (pulse
repetition interval frequency, 1040Hz),
(number of tracks, 200), (band rate, 2400) (message size 104
bits/message) (track initiation 6 seconds).
```

The other object with properties identified in the above text is 'aircraft' which has the property '`distance from controlled area`'.

The third type of analysis of the above text attempts to identify operations suffered or performed by objects.

The operations identified by looking at action and event descriptions are as follows:

`'send'` performed by `'radar'` and suffered by `'system'`. The parameters of the operation `'send'` are contained in the attribute `'message'` of radar.

Similarly, the operations `'tracking'` and `'track initiation'` performed by `'system'` are identified.

It can be seen from the above example, that object oriented analysis of NL provides the analyst with a simple mechanism for representing key concepts in the problem domain. Because the approach uses heuristic rules it relies to some extent on the ability of the analyst to apply the rules effectively, as well as on his familiarity with the analysed domain. Usually, a number of iterations will be required before the analyst arrives at a stable initial set of objects, attributes and operations. Nevertheless the simplicity of the approach make its use as a first stage requirements elicitation technique worthwhile.

In summary, requirements elicitation from NL is a promising approach (because the majority of knowledge about a domain is expressed in NL) which however suffers from a number of limitations i.e.

1. the complexity of NL makes the development of tools which can analyse unrestricted NL descriptions, impossible; thus, today only small subsets of NL can be processed by automated tools

2. the ambiguity of NL makes it unsuitable as a means to express a formal requirements model; therefore, all NL requirements must at some stage be translated to some formal language.

3.6 Techniques for Reuse of Requirements

Under this heading are examined approaches to requirements elicitation which are based on the following intuitively appealing idea:

> *Requirements which have been already captured for some application can be reused in specifying another similar application.*

The above statement seems appealing for the following reasons

- since requirements elicitation is admittedly the most labour and time consuming part of software development, any reduction in the time and resources it uses can result in significant overall productivity improvement

- there is a significant degree of similarity in systems which belong to the same application area. As Jones [Jones, 1984] indicates, only 15% of the requirements for a new system are unique to the system; the rest 85% comes from the requirements of existing similar ones.

Despite being a promising idea, requirements reusability is faced with a number of practical questions of applicability. The first question relates to the fact that requirements documents for existing systems are not easily available. This applies in particular to older systems for which the requirements were rarely recorded on any other medium than paper, nor updated or revised. The second question lies in the apparent difficulty of checking the suitability (relevance, consistency etc.) of an old requirement in the context of the specifications for the new system. It is obvious therefore, that for the idea of requirements reusability to become reality, the following things must become possible:

- requirements for existing systems must be easily accessible

- there must be facilities for selecting an old requirement, testing its suitability in the context of the new requirements model and modifying it if necessary, and finally

- all the above must cost less than simply doing requirements elicitation from scratch.

The approaches which are going to be discussed in the sequel, attempt to bring solutions to the above prerequisites to the reusability of requirements. More specifically, requirements reuse approaches tackle the problem of selecting and adopting an existing requirement for reuse.

Amongst the approaches falling in the category of 'requirements reuse' are the following:

- *Domain analysis*. Domain analysis has been characterised as the precursor to requirements analysis. Domain analysis identifies objects, rules and constraints common amongst different (but similar) domains and formalises them. In this way, requirements elicitation can use the results of domain analysis and save a significant amount of effort

- *Reusable requirements libraries*. Many approaches have advocated the development and maintenance of a library of reusable requirements components. Reusable components can have a significant impact on the effectiveness of requirements elicitation

- *Reverse engineering*. Reverse engineering is a technique of obtaining higher level information (requirements specifications/designs) from lower level one such as code. The technique seems to be promising, since some part of the requirements for a new software system is usually captured in an existing older system.

The final area of techniques, *reverse engineering* tackles the problem of acquiring the requirements for existing systems from a different angle. Reverse Engineering, reconstitutes the requirements model of a software system from information available in sources such as design, code, documentation etc. The primary aim of reverse engineering is to make old applications easier maintainable by maintaining specifications instead of code. However, a by-product of reverse engineering is that it makes the requirements model available again. This model can be used to either re implement parts of the existing system or to provide the basis for a new system. In this respect, reverse engineering facilitates the task of requirements elicitation. The above mentioned categories of requirements reusability will now be discussed in more detail.

3.6.1 Reuse of Requirements Specifications

Under this heading come all the approaches which propose libraries of reusable requirements as well as techniques for reusing them. In accordance with the general trends for automation in software development , these approaches tend to automate activities such as selection and modification of a reusable requirement. It must be noticed that the approaches described below are still at an experimental stage. This however, does not

reduce the validity of the ideas which they demonstrate, i.e. that reuse of requirements is a technique which the analyst uses anyway, sometimes even subconsciously. Psychological experiments [Vitalari, 1983] have revealed that reuse of requirements from similar systems is a common strategy employed by experienced analysts when faced with the analysis of a new system. As a matter of fact, it is exactly the ability to reuse past analysis expertise that makes the difference between an experienced analyst and an inexperienced one.

The first approach to requirements reusability comes in the context of a long-term research project known as the Knowledge-Based Requirements Assistant (KBRA) [Balzer et al, 1988] which aims at producing an intelligent tool for requirements elicitation and analysis. The reusable requirements in the KBRA tool appear as *formulas*, which can be engineering equations of the form 'distance = rate * time', statistical tables or simulation-generated tables. Formulas are used to capture aspects such as constraints on the non-functional requirements of the system such as accuracy, resolution, processing and response time, coverage etc. Other uses of formulas include tracing of formula-derived requirements, critiquing requirements input and suggesting ways for completing partial descriptions of requirements.

Another approach under the KBRA project, the *Requirements Apprentice*, [Reubinstein and Waters, 1991] codifies the reusable requirements in a *cliché* library. The term cliché is used to refer to a concept which is common in a class of similar problem domains. The clichés are classified into the categories of *environment*, needs *and system.*

A fragment of a cliché library, containing clichés about information system (which maintains a data base of information) and *tracking system* (which follows the state of something in the environment),both special cases of *system.*

In a typical session with the RA, the analyst is able to give informal definitions of requirements, which the RA matches to clichés already stored in its knowledge base. For instance, assume that the analyst wants to specify requirements for a university library system. RA has an extensive knowledge about information systems, tracking systems and repositories but no knowledge about libraries or library information system. The analyst, can therefore define the word library in terms of a repository and specify that its state is the set of books contained in it, as follows:

```
(Define Library :Ako Repository
                    :Defaults (:Collection-Type Book))
```

Because RA does not have a definition for 'books', the analyst must explain what a book is in terms of another cliché called *Physical Object*, whilst at the same time defines *title*, *author* and *Isbn* to be properties of *book*.

```
(Define Book :Ako Physical Object
                    :Member-Roles (Title Author Isbn))
```

Continuing in a similar manner, the analyst starts to define the functionality of the library system which is then checked by RA for consistency and completeness (based on expectations set up by various clichés). Obviously, in this approach RA plays an important role in requirements elicitation by allowing the analyst to speak in terms of high-level concepts which are subsequently refined into a specific and formal requirements model.

The last approach discussed under the section of 'reusable requirements' employs *analogical reasoning* as the technique of reusing a specification [Sutcliffe and Maiden, 1992]. The power of analogical reasoning lies in its potential to retrieve knowledge from one domain and apply it to a different (but similar) domain. There are many approaches to analogical reasoning, which however have as common concept the development of an abstract knowledge structure which contains commonalities of the two domains.

For instance consider the domains of *theatre reservation system* and *university course administration system*. Although belonging to different areas, the two domains share a significant number of features (e.g. reservations, waiting lists, places). It is therefore possible to abstract from the two domains a *resource allocation system* involving a *resource* (seats, places) and *clients* (theatre-goers, students).

Analogical reuse of requirements involves three processes, namely *categorisation* of a new problem, *selection* of a candidate requirements model belonging to the same category and *customisation* of the selected analogous requirements to the new domain.

Requirements reuse by analogy is a frequently (albeit informally and sometimes subconsciously) employed technique. When supported by a tool, the technique can help to overcome the inherent difficulties encountered by inexperienced analysts. Similar to other

tool-supported reuse techniques, analogical reuse is hindered by the excessive resources that the construction of a reusable requirements knowledge base takes.

3.6.2 Domain Analysis for Requirements elicitation

Most of the reuse approaches, remain silent as to how the mass of reusable requirements will be initially acquired. Domain Analysis in contrast, aims at creating the infrastructure needed for reuse of requirements, namely at

- identifying categories of problem domains, i.e. of similar applications
- identify and formalise the concepts which are common amongst the different applications in the domain
- organise the concepts in libraries of reusable components and provide facilities for accessing them.

Domain Analysis is a term used to describe the systematic activity to identify, formalise and classify the knowledge in a problem domain. Jim Neighbors, one of the pioneers in the area defines domain analysis as the activity of identifying objects and operations of a class of similar systems in a particular problem domain as well as of needs and requirements for a collection of systems which seem 'similar' [Neighbors, 1984].

It can be deduced from the above definition, that the objectives of domain analysis and requirements analysis are the same, the difference being that domain analysis considers the requirements of more than one application. As mentioned in the introduction of this Chapter, what makes requirements elicitation difficult is the lack of understanding of the problem domain. In this respect, Domain Analysis comes as an aide to requirements elicitation in the sense that it provides all the knowledge required by the latter in a reusable format. Thus, under the Domain Analysis paradigm, requirements elicitation becomes a sequence of

- selection of reusable requirement, and
- possible adaptation of requirement for incorporation in the new requirements model.

It must be noted that Domain Analysis caters for all the phases of software development instead of just for requirements analysis. As suggested in [Prieto-Diaz and Arango, 1991] for example, "`..Domain Analysis is the process of identification, acquisition, and evolution of reusable information on a problem domain...`".

Domain Analysis needs a representation vehicle in order to convey the reusable knowledge from a domain. Requirements modelling formalisms (with more important one being the Object-Oriented Model) such as those discussed in Chapter 4 are usually adequate for more of the Domain Analysis. Domain Analysis requires in addition, a set of methods and tools for its application. As various researchers note, Domain Analysis is a difficult process requiring usually four months from an expert time to complete a first attempt at a domain. The cost of this initial investigation can be quickly amortised as the results of Domain Analysis increase the productivity and quality of the software projects on which they are applied. Major inputs to the Domain Analysis process are technical literature, existing system implementations, expert advice etc. Major outputs of Domain Analysis include a taxonomy of domain concepts, standards (e.g. for user interfaces), generic architectures of systems and domain-specific languages. A plethora of experts is also required to participate in domain analysis such as domain experts, analysts, librarians (which classify, update and distribute the reusable components) etc.

Domain Analysis is a young discipline, which nevertheless is capable of changing the conventional ways of developing software. So far it has been applied only to a small number of large scale projects, such as CAMP (common software components in a missile system) with considerable success [Hall, 1991]

3.6.3 Reverse Engineering

Reverse Engineering [Chickofsky and Cross, 1990] is the process of analysing a software system in order to

- identify its components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction

In the context of this Chapter only the type of reverse engineering which reconstructs the system requirements specifications from lower level information is considered. In this perspective, therefore, the outcome of reverse engineering is a requirements model which can be directly used in the elicitation of the requirements for the new system. Despite the preferred treatment of software development as an activity that starts from scratch, this is rarely the case. In the majority of cases, a new system is built as an 'extension', 'enhancement' etc. of an existing one, and even in some cases the new system is only a subsystem of the older one. The importance of reverse engineering in obtaining the requirements of the original system cannot therefore be underestimated.

In many situations, the task of Reverse Engineering is hindered by the loss of information which was originally created during software development. Information such as the justification for a particular specification, the rationale behind a design decision, the link between a requirement and the corresponding design etc., is rarely recorded during software development. Also subsequent (maintenance) changes to code are not reflected on the requirements document which thus becomes inconsistent with the actual running system. For all these reasons, reverse engineering a system to its requirements is a difficult, or sometimes impossible task.

Most existing automated approaches to reverse engineering rely on low level documentation (i.e. code) in order to recreate higher level documents such as designs. Moving from design to requirements, however remains an insurmountable obstacle in many situations, unless some necessary information about the software system becomes available. Recent experimental techniques (e.g. [Biggerstaff, 1988] [Karakostas, 1992]) succeed in recreating the requirements model of a system) by relying more on knowledge that can be found in the program code alone. Central to the success of these approaches is the concept of the problem domain model. A domain model contains representations of the major concepts that appear in the problem domain modelled by the software system. In addition to that, a domain model contains *development specific* knowledge, i.e. patterns which show how the concepts are *typically* transformed to design and coding constructs.

In the experimental system IRENE (which is an acronym for Intelligent Reverse Engineering Environment) described in [Karakostas, 1992], the domain knowledge base consists of

- concepts typically appearing in the domain. For instance, in a payroll application typical concepts include *tax, taxable-salary, tax-rate* etc.

- knowledge about relations between the concepts, e.g. that the *taxable_salar*y and the *tax-rate* determine the payable *tax*

- implementation knowledge, such as the knowledge that *tax* is implemented in COBOL as a *small integer* (between 0 and 100).

Based on the above types of knowledge, IRENE searches the program code in order to match portions of code with domain concepts. IRENE can for example verify that a domain concept is mentioned in the original requirements (e.g. that *tax-rate* is defined in the requirements for calculating *tax payable*). The system can also identify possible definitions of concepts which are 'hidden' in the code but not mentioned in the requirements document. For instance, the system can come across the data name TX-RELF used in the calculation of *tax payable*. By looking at the library of domain concepts, IRENE 'suspects' that TX-RELF corresponds to concept *tax-relief* which was not however mentioned in the requirements document! In this respect, IRENE not only reconstructs the true requirements for the software, but also corrects inconsistencies, outdated definitions etc. that might appear in the existing requirements document.

Knowledge-based systems like the above, have the potential to provide a truly automated solution to the reverse engineering of requirements. Since, however, reconstructing the original requirements is paramount to the elicitation of new requirements the analyst must use any existing documentation that can lead to (even a partial) reconstruction of the old requirements. Care must be taken, however, that the analyst is not relying too much ('anchor' his analysis) on the old requirements since they might describe things which are of no use (or even not true about) to the new system.

Despite its pitfalls, retrieving and reusing existing requirements for requirements elicitation, is a pragmatic technique with real importance. The degree of successful application of requirements reusability is determined by factors such as

- the availability, accessibility, testability and modifiability of the existing requirements

- the extent to which the new software system is similar to existing one(s).

As automation of activities like requirements analysis and specification becomes more widespread, and more software development related information is captured and stored in repositories (Chapter 6) we can expect in the near future a large increase in the popularity of requirements reusability techniques.

3.7 Task Analysis for Requirements Elicitation

Task Analysis is an effective method for eliciting user requirements, in particular those requirements concerned with human-computer interaction issues. The term 'Task Analysis' refers to a set of methods and techniques which analyse and describe the way users do their jobs in terms of

- activities they perform and how such activities are structured
- what knowledge is required for the performance of the activities.

Historically, Task Analysis has focused in describing in a very detailed manner the order with which people perform their activities, starting with *plans*, down to the level of basic tasks which cannot be further analysed [Diaper 1989a]. Hierarchical Task Analysis is a method which builds a hierarchy of tasks and sub-tasks and also plans describing in what order and under what conditions sub-tasks are performed. Figure 3.4 uses the library case study first discussed in Section 3.3 of this chapter to show the decomposition of the task *check out book.*

```
0      In order to check out a book
       1 get the borrower's library card
             1.1 check to see if the card is valid
             1.2 check the borrower's record to see if the number of
                   borrowed books allowed at any time has been exceeded
             1.1 get the borrower's name from the card
             1.2 get the card's number

       2  get the book from the borrower

       3  get a new (unused) check out card
             3.1 enter the current date on the check out card
             3.2  enter the borrower's name on the check out card
             3.3  enter the book's catalogue number on the record
             3.4   enter the due back date on the record
                   3.4.1  calculate the date the book is due back
                   3.4.2   write the due back date on the check out card
       4  stamp the book with the due back date
       5  hand the book back to the borrower
```

Figure 3.4: A plan for checking out a book

In the plan shown in figure 3.4 not all the sub-tasks need to be performed, nor necessarily in the order presented in it. For example, Task 2 ('get the book from the borrower') can be performed before Task 1 ('get the borrower's library card'). In addition, there are no clear cut rules as to the level where the decomposition of tasks into sub-tasks must terminate. Guidelines suggest, however, that the attempt to further analyse tasks which contain complex motor responses (physical actions) or internal decision making may result in incorrect identification of sub-tasks.

The use of methods and techniques in order to describe the knowledge required to perform a task is called *Knowledge-Based Analysis* [Diaper 1989b] and is complimentary to Hierarchical Task Analysis. *Knowledge-Based Analysis* creates models of objects, relations and events in the task domain and in this respect it is similar to functional modelling approaches (see Chapter 4). However, the aim of Knowledge-Based Analysis differs from that of modelling of functional requirements, in the sense that the former does not attempt to model entities which will be represented in the information system but considers instead physical entities. The example of Figure 3.5 shows a hierarchy of real people who use the library facilities. Note that this hierarchy may differ from the one that will be eventually modeled inside the library's computerized information system.

Task Analysis can provide a valuable input to the Requirements Elicitation process. However such input is principally one of clarifying and organizing the knowledge about the

problem domain. Task Analysis cannot yield requirements for the new system since it refers to the existing system, not the planned system and in addition it includes many elements which will not be part of the future software system. Nevertheless, Task Analysis can provide the basis for specifying the requirements for the new systems based on modifications, extensions and novel features which must be incorporated in the current system. Going back to the example of Figure 3.4, many of the listed tasks will continue to be performed in the new system, albeit in a new computerized form, some may disappear and others may need to be retaught completely.

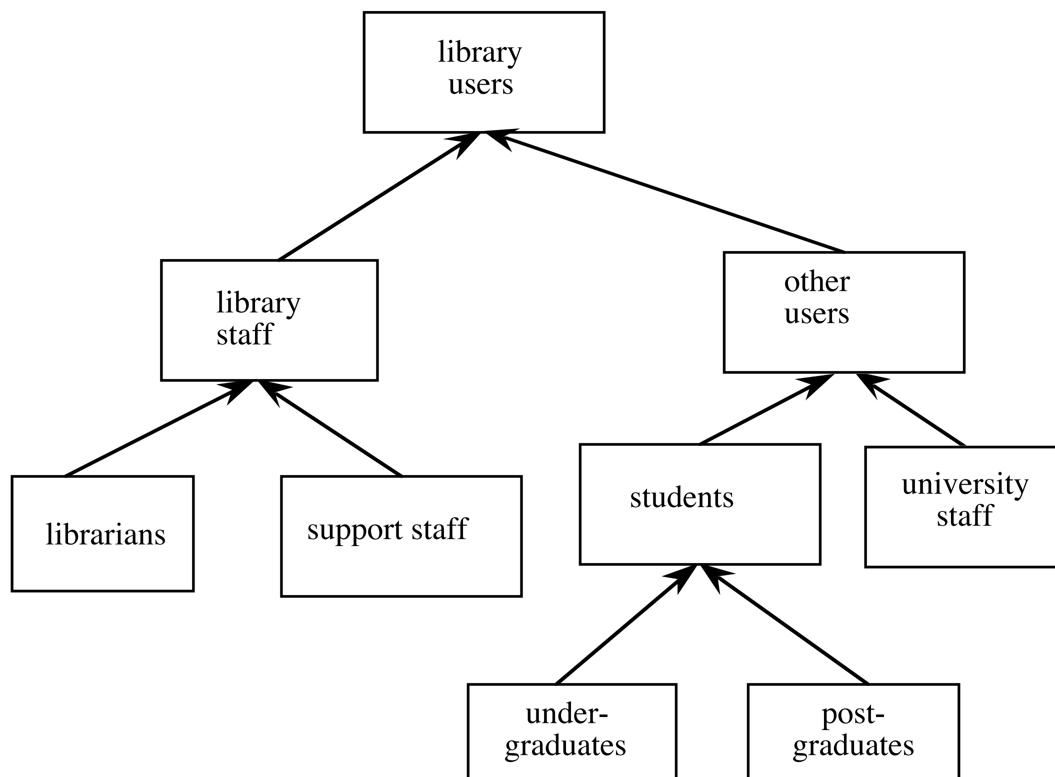


Figure 3.5: Hierarchy of library users

3.8 Requirements Elicitation as a Social Process

There is a body of work in Requirement Engineering which is based on the premise that requirements elicitation is not primarily a technical problem, but this process should be carried out within a social context. Some researchers claim that the lack of proper consideration of the social context in which the software will be used accounts for the

majority of failures i.e. for projects in which either no system can be built that satisfies the user requirements, or in which the developed system does not support the real user needs.

Socio-Technical approaches to Requirements Elicitation are based on the premise that the social issues of the system (organisation) which will host the software system are as important as the technical ones and that the two are inevitably interdependent. According to these approaches 'elicitation' is probably not an appropriate term since it assumes that requirements are 'out there' to be elicited. In reality, however, a user requirement is more an outcome of the interaction between the users and the requirements engineer, rather than some pre-existing concept in the users minds.

Elicitation, according to the socio-technical approaches cannot be practised at the level of individual users, in isolation from their environments and from their interactions with each other. As a consequence, requirements have meaning only within the context (time, place and situation) in which they were observed. This of course renders the classic requirements analysis techniques which attempt to isolate the requirements from their context, inadequate.

The argument in favour of adopting different techniques to those advocated by traditional systems analysis methods is that in traditional methods the user plays a passive role. A user is simply considered as the provider of information pertinent to the requirements elicitation process in hand. However, socio-technical approaches suggest that system design should be considered in a social and organisational context [Mumford and Weir 1979], that a more active participation of users is required and that much thought must be given to the constitution of the development team. The construction of the development team is guided by the realisation that technical experts and users provide different expertise and knowledge to the task of requirements elicitation and that ultimately they have vested interests in the solutions proposed. Three options are proposed in [Eason 1987] for the structure of a development team:

- Technical Centred Design. Customers are informed and consulted by technical experts throughout the development process.

- Joint Customer-Specialist Design. User representatives are involved at all stages of the development process.

- User-Centred Design. Technical experts provide a technical service to users and all users contribute to the design process.

It has been shown that each approach has its advantages and disadvantages. For example whilst the Technical Centred Design approach provides the basis for appropriate use of technical skills and is acceptable to the commissioning organisation, but, fails to take into consideration the need for involving users in the construction of a system that ultimately will change the working practices of the users themselves. The User-Centred Design approach seems to answer this criticism but, is regarded as being too inefficient on resources.

According to the socio-technical approach therefore, an important consideration for the successful interpretation of user needs is the optimum involvement of all stakeholders i.e. all those that have an interest in the change being considered, those that stand to gain from it and those who stand to lose [Mitroff 1980]. There are typically the following kinds of stakeholders in the requirements definition process [Macaulay 1993]:

- Those that have a financial interest in the system to be developed. Typically, these may be customers procuring the system or system component or marketeers concerned with some future product or the evolution of an existing product.

- Those who are responsible for the design and implementation of the system to be developed. These may be project managers, software engineers, telecommunication experts etc.

- Those responsible for the introduction of the system once the system has been developed for example training personnel, user managers etc.

- Those that have an interest in the use of the system. These could be frequent users, occasional users or even users affected by the system without necessarily being involved with its use.

These four classes of stakeholders provide the basis for organising a series of workshops during which requirements are explored in a co-operative fashion with the help of a facilitator [Macaulay 1994]. The benefits of these workshops lie very much in the face-to-

face exploration of current situations and future needs with the aim to developing a shared understanding of the issues involved. To develop this shared understanding one needs to use some 'language' for communicating individual views and for documenting agreed positions. There is a variety of such languages from natural language and informal definition of basic concepts e.g. [Macaulay 1994b] to more formal specification approaches e.g. [Bubenko, Rolland, et al 1994; Loucopoulos 1994; Nellborn, Bubenko, et al 1992].

In conjunction with paying attention to the concerns of development team organisation, the socio-technical approach has encouraged the move away from conventional elicitation techniques such as interviews and questionnaires in favour of ones which originate in social sciences and linguistics.

Social science methods such as *ethnography* are also suggested as promising techniques for understanding and eliciting the true user requirements [Goguen and Linde, 1993] [Sommerville et al, 1993]. Ethnography is a method developed and used by anthropologists for understanding social mechanisms in 'primitive' societies (e.g. tribes). The same method however can be applied to the analysis of the work practices within organisations.

Ethnomethodology is a branch of sociology which questions the validity of conventional sociological methods such as questionnaires and statistics, preferring instead methods which are based on the behaviour of the members of the user group. Through these observations, ethnomethodologists aim to understand the categories and methods used by the users for rendering their actions intelligible to others, instead of trying to impose their own methods and categories onto the users. Ethnomethodology therefore seems to provide an alternative to classical requirements elicitation, which promises to yield higher quality requirements than would have been the case when using traditional techniques.

The application of ethnography to the study of organisations entails the analyst spending a long period of time with the organisation and making detailed observations about its work practices. Subsequent analysis of the observations can reveal vital information about the organisation, which usually differs markedly from the one recorded in formal documents (manuals, handbooks) of the organisation. The advantage of the ethnography approach over conventional systems analysis lies on the fact that analysts are passive observers and do not try to impose their judgements on the practices which are observed.

In contrast to task analysis, ethnography is based on the premise that there is no such thing as context-free user task. Ethnography questions the valididity of task analysis as a mechanism for analysing user activities because of the reliance of task analysis to concentrate on individual tasks and the imposition of a kind of structuring that does not take into consideration the cooperative and interactive nature of activities in organisational settings. Ethnography is also different to traditional systems analysis methods in that there are no pre-conceptions about the application being studied and there is no judgement being offered on the practices being observed.

The usefulness of ethnography to requirements engineering is yet to be clearly defined. Results from empirical studies however, tend to support the notion that a social science perspective can be relevant particularly in settings involving interaction and cooperation [Sommerville, Bentley, et al 1994]. Practical experience has also shown that the use of ethography in requirements elicitation needs further elaboration and structuring, it is sometimes difficult to understand and time consuming to practice [Sommerville, Bentley, et al 1994] [Goguen 1994].

In reality, as the practitioners of the approach indicate [Goguen, 1994], Ethnomethodology can be difficult to understand and time consuming to practice. Moreover, there are no clear guidelines as to which of the results are useful in eliciting software requirements. Other approaches which have grown out of ethnomethodology such as 'conversation analysis' (which focuses on aspects of ordinary conversation such as timing, overlap, response etc.) and interaction analysis (which uses videoed user activities) might also prove a useful addition to the repertoire of requirements elicitation techniques.

In summary, Socio-Technical approaches to requirements analysis can prove to be an important supplement to more technical oriented Requirements Analysis techniques, since they provide valuable information about the users environment, i.e. activities, concepts and patterns of interactions.

3.9 Requirements elicitation vs. knowledge elicitation

This Section argues that the requirements engineer can no longer afford to be unaware of the developments in a field very much related to Requirements Engineering, namely Knowledge Engineering.

It has been proposed [Byrd et al, 1992] that a merged awareness of both Requirements Engineering and Knowledge Engineering research must take place resulting in an exchange of ideas, techniques and methods between the two disciplines.

In the Expert Systems literature Knowledge Elicitation (Acquisition) has been described as the transfer and transformation of problem-solving expertise from some knowledge source to a computer program [Hayes-Roth, 1984]. The knowledge acquired from the user comes usually in two forms, namely a declarative form (which consists of facts about concepts, their classifications and relationships) and a procedural form which contains information about where and how to apply the declarative knowledge.

During knowledge elicitation the practitioner is faced with similar problems to those we discussed earlier in this Chapter, regarding the elicitation of software requirements. The major difficulty in both Requirements Engineering and Knowledge Engineering is obtaining a good understanding of the problem domain. In both cases, understanding the domain is a problem because the major source of domain knowledge is the user. The problem of extracting knowledge from the user has been coined 'the knowledge acquisition bottleneck' in the Expert Systems literature. The following are some of the obstacles in extracting knowledge from the user.

- limitations of humans as information processors and problem solvers account to an extent for the Knowledge Engineering problems. Users find it, in general, difficult to recall and explain their actions and decisions when solve a problem.

- communication problems stemming from the fact that users and Knowledge Engineers use different languages. The user's language is specialised terminology about the problem domain, whilst the Knowledge Engineer uses technical jargon related to the design aspects of the Expert System.

- problems stemming from the need to deal with a number of users with sometimes conflicting experiences and needs.

In order to tackle the above problems Knowledge Engineering research has developed a number of techniques which fall into 5 broad categories. It must be noted that some of these techniques have close counterparts in Requirements Engineering, while others have not, but are nevertheless very applicable themselves. The knowledge elicitation techniques categories are:

- *observation techniques* (the user is observed while doing a specific task). Well known examples of observation techniques are *behaviour analysis* and *protocol analysis*

- *unstructured elicitation* techniques in which the user(s) participates in interviews, brainstorming sessions etc. Typical examples of this category are *teachback interview* and *open interview*

- *mapping techniques* are psychological techniques used to acquire conceptual knowledge from the user; *multidimensional scaling* and *variance analysis are examples of* this category

- *formal analysis techniques* are automated techniques which induce rules from data, analyse text etc. *Machine Rule Induction* is a typical example of this category.

- *structured elicitation techniques* in which the users are participating in a series of structured experiments from which knowledge is elicited. *Card Sort* is a typical example of this category.

Many requirements elicitation techniques discussed in this Chapter can be considered as belonging to one of the above categories. *Objective/goal analysis* (Section 3.1.1) for example, is a type of unstructured elicitation technique. Natural Language techniques can be considered as belonging to the *formal analysis* category. *Scenario-based elicitation* (Section 3.1.2) falls in the structured elicitation techniques category. Prototyping (discussed in Chapter 5) can be considered as another unstructured elicitation technique because of its user-participation nature.

The value of the comparison between requirements elicitation and knowledge elicitation lies in the fact that techniques from one category can be applied to the other and vice versa.

Depending on the type of the problem domain and the nature of the communication problem (e.g. user limitations, language problem etc.) analysts can improve their techniques repertoire by selecting and applying a suitable knowledge elicitation technique.

Summary

This Chapter was concerned with the phase of Requirements Engineering known as requirements elicitation. The essence of requirements elicitation, as the process of 'understanding of the problem domain' was highlighted throughout the Chapter.

There are different types of problems which make requirements elicitation a difficult task, not dissimilar to the 'knowledge acquisition bottleneck' which hinders its sister discipline Knowledge Elicitation. The major problems of knowledge elicitation is

> the difficulty of the analyst to acquire knowledge from the users or other sources and thus become himself an expert in that domain

The various types of elicitation techniques presented in this Chapter have different strong and weak points when dealing with the above problem.

- User interviews are straightforward to use but usually require careful preparation of the questionnaire if they are to be effective.
- Objectives/goal analysis techniques succeed in achieving a consensus amongst the different users on explicitly defining the primary problems (goals, objectives).
- Scenario-based techniques tackle the problem of limited memory and recall of expertise of the user by making the user participate in various scenaria regarding his interaction with the software system.
- Form-based analysis techniques bypass the user as a source of domain knowledge and focus instead on a rather plentiful source of knowledge in organisational environments, namely forms.

- Natural Language analysis approaches tackle the problem of language differences between user and analyst by carrying the elicitation process in the most convenient medium for the user which is of course natural language.

- Reuse-based approaches attempt to dispense with the necessity of doing elicitation from scratch, by providing a set of reusable requirements as a start point. Tool-supported reuse approaches store the reusable requirements in repositories and provide assistance with regard to their retrieval and adaptation. Domain Analysis aims at producing formal reusable models of requirements capturing commonalties between domains. Reverse Engineering reconstructs the requirements of existing system with the purpose of partially reusing them for the new system

- Social science approaches take into account the social rules and practices in the organisation, both at the personal and group level, in order to obtain insights about their real working practices and thus lead to definition of their real requirements.

References

Arango, G. (1989) *Domain Analysis From Art to Engineering Discipline*. Proc. Fifth Int'l Workshop on Software Specification and Design IEEE Computer Society Press.

Balzer et al (1988). *RADC System/Software Requirements Engineering Testbed Research and Development Program*. Report TR-88-75, Rome ir Development Center, Griffiss Air Force Base, N.Y., June.

Biggerstaff, T. J. (1988) *Design Recovery for Reuse and Maintenance*. MCC Technical Report STP-378-88.

Booch, G. (1986) *Object-oriented development*. IEEE Trans. on Software Engineering, Vol. SE-12, No. 2, February.

Booch, G. (1991) Object-oriented design, Benjamin-Cummings.

Bouzegoub, M. & Gardarin, G. (1986)*SECSI: an expert approach for data base design*. In Proc. of IFIP Wold COngress, Dublin, Sept.

Bubenko, J.A. and Wangler, B. (1993) *Objectives Driven Capture of Business Rules and Information Systems Requirements*, IEEE Conference on Systems, Man and Cybernetics, 1993.

Bubenko, J., Rolland, C., Loucopoulos, P. and de Antonellis, V. (1994) *Facilitating "Fuzzy to Formal" Requirements Modelling*, IEEE International Conference on Requirements Engineering, 1994.

Byrd, T. A., Cossick, K. L. & Zmud, R W. (1992) *A Synthesis of Research on Requirements analysis and Knowledge Acquisition Techniques*. IS Quarterly. March.

Chen, P. (1976) *The Entity-Relationship Model. Toward a Unified View of Data,* ACM Trans. on Database Systems.

Chicofsky, E. J. and Cross II J. H. (1990) *Reverse Engineering and Design Recovery: A taxonomy*. IEEE Software 7(1).

Choobineh, M., Mannino, J., Nunamaker, J., Konsynsky, B. (1988) *An Expert database System Based on Analysis of Forms*. IEEE Trans. on Software Engineering. Vol. 14, No. 2.

Coad P. and Yourdon, E. (1991) OOA-Object-Oriented Analysis, Prentice-Hall, Englewood Cliffs, NJ.

Crowley, D. J. (1982) Understanding Communication:The Signifying Web. New York: Gordon and Breach Science Publishers.

Diaper, D. (editor) (1989a) *Task Analysis for Human Computer Interaction*. Ellis Horwood, Chichester, 1989.

Diaper, D. (1989b) Task Analysis for Knowledge Descriptions (TAKD): the method and an example, In Diaper, D. (editor) *Task Analysis for Human Computer Interaction.* Ellis Horwood, Chichester, 1989.

Edwards, A. (1987) *Mining for Knowledge.* Accountancy, April.

Eason, K. (1987) Information Technology and Organisational Change, Taylor and Francis, 1987.

Goguen, J. (1994) *Requirements engineering as the reconciliation of social and technical issues*, in requirements Engineering Social and Technical Issues, M Jirotka and J Goguen (eds) Academic Press, 1994.

Goguen, J. A. & Linde, C. (1993) *Techniques for Requirements Elicitation.* IEEE Symposium on Requirements Engineering.

Graham, I. and Jones, P. L. (1988) Expert Systems: Knowledge Uncertainty and Decision. St. Edmundsbury Press Ltd., Bury St., Edmunds, Suffolk, England.

Hall, P.A.V. (1991) *Overview of Reverse Engineering and Reuse Research.* Department of Computing, Open University, Milton Keynes, England.

Hayes-Roth, F. D. (1984) *The Knowledge-Based Expert System: A Tutorial.* IEEE COMPUTER 17 (9) September.

Holbrook III , H. (1990) *A Scenario-Based Methopdology for Conducting Requirements Elicitation,* ACM Soft. Eng. Notes, Vol. 15 no 1, January.

Hooper, J. W. and Hsia, P. (1982) *Scenario-based Prototyping for Requirements Identification.* ACM SIGSOFT Software Engineering Notes, 7 (5).

Jones, T. C. (1984) *Reusability n Programming. A Survey of the State of the Art.* IEEE Trans. on Software Engineering. Vol SE-10, No. 9, September

Karakostas, V. (1990) *Modelling and Maintaining Software Systems at the Teleological Level.* Journal of Software Maintenance, Vol. 2.

Karakostas, V. (1992) *Intelligent Search & Acquisition of Business Knowledge from Programs* Journal of Software Maintenance, Vol. 3.

Karakostas, V. & Loucopoulos, P. (1989) *Constructing and Validating Conceptual Models of Office Information Systems: A Knowledge-based approach.* Proc. 3rd Conf. Putting into Practice Methods and tools as aids to design information systems, Nantes, France.

Kersten, M. L. , Weigand, H., Dignum, F. & Proom, J. (1986) *A Conceptual Modelling Expert System.* In Proc. 5th Int'l Conf. on the ER Approach. S. Spaccapietra (ed), Dijon, France.

Loucopoulos, P. & Karakostas, V. (1989) *Modelling and Validating Office Information Systems: An object and logic-oriented approach.* Sofware Engineering Journal, March.

Loucopoulos, P. (1994) *Extending Database Design Techniques to Incorporate Enterprise Requirements Evolution*, Baltic94, J. Bubenko, A. Caplinskas, J. Grundspenkis, H.-M. Haav and A. Sølvberg (ed.), Vilnius, Lithuania, 1994, pp. 8-23.

Macaulay, L. (1993) *Requirements Capture as a Cooperative Activity*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press, San Diego, California, 1993, pp. 174-181.

Macaulay, L.A. (1994) *Cooperative Requirements Capture: Control Room 2000*, in 'Requirements Engineering: Social and Technical Issues', M. Jirotka and J. A. Goguen (ed.), Academic Press, London, pp. 67-86.

Mittermeir, R. T., Rousopoulos, N., Yeh, T. & Ng, P. (1990) *An Integrated Approach to Requirements Analysis.* In Modern Software Engineering: Foundations and Current Perspectives (eds. P. A. Ng and R. T. Yeh). Van Nostrand Reinhold, New York, Oct.

Mitroff, I.I. (1980) Management Myth Information Systems Revisited: A Strategic Approach to Asking Nasty Questions About System Design, in 'The Human Side of Enterprise', N. Bjorn-Adnersen (ed.), North-Holland, Amsterdam.

Mumford, E. and Weir, M. (1979) Computer Systems in Work Design - the ETHICS Method, Associated Business Press, London, 1979.

Neighbors, J. M. (1984) *The DRACO approach to constructing software from reusable components*. IEEE Trans. on Software Engineering. Vol. SE-10, No. 5, September.

Nellborn, C., Bubenko, J. and Gustafsson, M. (1992) *Enterprise Modelling - the Key to Capturing Requirements for Information Systems*, SISU, F3 Project Internal Report, 1992.

Prieto-Diaz, R. & Arango, G. (1991) Domain Analysis and Software System Modelling, IEEE 199

Reubenstein, H. B. & Waters, R. C. (1991) *The Requirements Apprentice:Automated Assistance for Requirements Acquisition*. IEEE Trans. on Software Engineering, Vol. 17, No. 3.

Rolland, C. & Richard, C. (1982) *The REMORA methodology for Information Systems Dersign and Management*. In IFIP WG8.1 Working Conf. on Information Systems Design Methodologies: A Comparative Review. Olle T. W. et al . eds. North-Holland.

Rolland, C. & Proix, C. (1992). *A Natural Language Approach for Requirements Engineering*. In Proc. 4th Int'l Conference CAISE'92, P. Loucopoulos (ed) Springer-Verlag.

Sommerville, I., Bentley, R., Rodden, T. and Sawyer, P. (1994) *Cooperative Systems Design*, The Computer Journal, Vol. 37, No. 5, 1994, pp. 357-366.

Telem, M. (1988) *Information Requirements Specification I: Brainstorming Collective Decision Making Approach.* Information Processing and Management (24:5).

Vitalari, N. and Dickson, G. (1983) *Problem Solving for Effective Systems Analysis: An Experimental Exploration.* CACM, Vol. 26, No. 11, November.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. & Twidale, M. (1993) *Integrating Ethography intoThe Requirements Engineering Process.* IEEE Symposium on Requirements Engineering.

Sutcliffe, A. G. and Maiden, N. A. M. (1992) *Supporting Component Match for Software Reuse.* In Proc. 4th Int'l Conference CAISE'92, P. Loucopoulos (ed) Springer-Verlag.

CHAPTER 6

C.A.S.E. Technology

Introduction

The overall aim of CASE technology is to improve the productivity and quality of the resulting systems by assisting the developer throughout the different stages of the development process; from the acquisition of the functional and non-functional system requirements to the design and implementation of the system considering all the relevant technical and operational features.

CASE provides the software tools that support methodologies to employ in modelling all levels of an organisation. In this sense it is more appropriate to consider CASE in a wider context than just software production, that has normally been the case. Therefore, CASE may be described as software tools for enterprise support consisting of enterprise strategic planning, IS strategic planning, project planning, systems development, documentation and maintenance.

This chapter is organised as follows. Section 6.1 presents the possible advantages from the CASE technology and the need for CASE as part of system development and especially Requirements Engineering. Section 6.2 provides a classification of CASE technology. Distinctions are made between stand-alone CASE tools and integrated environments. Section 6.3 presents a generic architecture for CASE. Despite the plethora of CASE tools and supported methodologies in use today, the typical CASE architecture is built around the concept of a

repository used for storing the various types of models which are created during software development, which are accessed by a number of assorted tools such as editors, diagrammers etc. The same section therefore specifies the requirements for repository functionality as part of CASE for Requirements Engineering. Issues to consider when selecting and integrating CASE tool for requirements are discussed in section 6.4. Finally, section 6.5 is concerned with properties and futures of research-oriented CASE tools for Requirements Engineering. Many of such tools are based on the principle that the utilisation of problem domain and methodical knowledge stored in the tool can automate (at least to an extent) many of the human labour intensive tasks of Requirements Engineering.

In summary, this Chapter presents an overview of the role of CASE technology within Requirements Engineering, keeping the discussion as much as possible free from references to specific tools and technologies. CASE is an indispensable feature of any modern software development approach and will be even more so in the future as its potential applicability within the Requirements Engineering process increases.

6.1 The Need for Computer-Aided Software Development

Requirements Engineering is one of the software development phases for which CASE is particularly suitable. This fact was realised in the early days of CASE and it was soon put into practice in the form of research prototypes first and commercial tools soon after. There are of course good reasons why CASE is particularly applicable to Requirements Engineering: Requirements Engineering produces vast amounts of information (in both textual and graphical forms). Obviously this information must be managed, captured, stored, retrieved, disseminated and changed. However the manual capture, storage, manipulation etc. of requirements increases the risk of errors creeping into the process and into the final outcome. Such errors cam manifest themselves in various forms i.e. as out-dated, inconsistent or incomplete or erroneous requirements models.

Another serious problem of manual practice of Requirements Engineering lies with the difficulty to enforce certain software standards. As it happens, users and software engineers have their individual ways of communication in textual or pictorial form. This however can cause serious communications problems in the context of software development. User reports written in a variety of formats and styles can be a nightmare for those responsible for their editing and translation into technical descriptions.

CASE tackles the problems related to software requirements in the following ways:

- First, by providing automated management of all the requirements related information.

- Second by enforcing standards. CASE tools provide standard formats for inputting, retrieving or changing information. Such standards can be for example document formats for textual requirements, diagrammatic notations etc. The use of a uniform set of standards across the software development team ensures that problems such inconsistency and misinterpretation are eliminated or greatly reduced.

CASE also affects the speed with which a requirements model is produced and updated. This is important, as Requirements Engineering has to deal with two contradicting demands, namely to involve the user as little as possible in the process (since users are probably too busy doing other things to participate in software development) and at the same time obtain all the information required from the user including feedback on the specified requirements. In resolving this conflict, CASE is invaluable since it offers two primary facilities, namely easy communication with the user through graphical models and prototyping. CASE allows the quick construction of quality graphical models which are easily understood by the user as well as their equally quick modification. Prototyping is also valuable in putting the user through life-like interaction with the software system in order to understand the user's true requirements (see also Chapter 5).

Most of the arguments in favour of using CASE in Requirements Engineering apply equally to the use of systematic methods for Requirements Engineering. As a matter of fact, CASE started as nothing more than an attempt to automate paper-and-pen software methods which in turn provided the much needed standardisation of documents and models, procedures for requirements change etc. to software development. It is a truism therefore that CASE and software methodologies have been dependent on its other for their success. Methods require automation in order to be practical; CASE on the other hand is of little help unless used in a systematic way within the development process, i.e. within a *method*. In a 'chicken and egg' fashion therefore the structured methodologies and CASE are responsible for each other's fast spreading of popularity in the 80's and 90's.

However, as the state-of-the-art moves beyond the structured approaches and into object-oriented and knowledge-based paradigms, the importance of CASE as the enabling technology for Requirements Engineering is moving into new areas. Sections 6.3 and 6.5 discuss the new

role of CASE as a component of a Requirements Engineering approach, in both commercial and research-oriented settings.

6.2 Classification of CASE Technology

There exist several classifications of CASE in the literature. One of the first and most important ones classified CASE as *language-centred*, built around a programming language, *structure-centred* that were based on the idea of environment generation, *toolkit environments* that were primarily consisting of tools that supported the programming phase of the development and *method-based* which were centred around a specific methodology for developing software systems.

Another classification [Fuggetta 1993] is based on a framework consisting of three parts: *tools* that support only specific tasks in systems development, *workbenches* that support one or more activities and *environments* that support a large part of the software process. According to this classification, tools can be further classified into editing, programming, verification and validation, configuration management, metrics and measurement, and project management tools. Workbenches are classified depending on the activities that they support as: business planning and modelling, user interface development, programming, verification and validation, maintenance and reverse engineering, configuration management and project management workbenches. Finally, environments are classified as either toolkits which are integrated collections of products, language-centred, integrated, fourth generation environments or process-centred environments.

6.2.1 Upper and Lower-CASE

The most popular classification of CASE technology and tools is based on the distinction made between the early and late stages of systems development. Many of the current CASE tools deal with the management of the system specification only by supporting strategy, planning and the construction of the conceptual level of the enterprise model. These tools are often termed *upperCASE tools* because they assist the designer only at the early stages of system development and ignore the actual implementation of the system. The emphasis in upperCASE is to describe the mission, objectives, strategies, operational plans, resources, component parts etc. of the enterprise and provide automated support for defining the logical level of the business, its information needs and designing information systems to meet these needs.

UpperCASE tools support traditional diagrammatic languages like Entity Relationship Diagrams, Data Flow Diagrams, Structure Charts etc. providing mainly draw, store as well as documentation facilities. They support a limited degree of verification, validation and integration of the system specifications due to the inherent lack of formal foundations for the requirements modelling formalisms.

Other CASE tools deal with the application development itself with regard to the efficient generation of code. These are termed *lowerCASE tools* because they assist the developer at the stage of system generation and ignore the early stages of system requirements specification. The starting point of the system development with lowerCASE tools is the conceptual model of the information system. The conceptual modelling formalism is usually, based on formal foundations in order to allow for automatic mapping to executable specifications and efficient validation and verification of the system specification itself.

LowerCASE tools employ mapping algorithms to automatically transform formal specifications into an executable form. This includes, among others, transformation of specifications to relational database schemas, normalisation of database relations and SQL code generation. The majority of these tools facilitate rapid prototyping of specifications in terms of the functionality of the system. They do not support the development process itself but rather, they offer a powerful tool for making system design more effective and efficient.

The state of the art products of the CASE market nowadays claim that provide support for both the early stages as well as the implementation stages of information systems development. Clearly, from a users' perspective, this move towards *integrated CASE (ICASE)* is far more important [Gibson et al, 1989]. In this architecture, the repository plays a more active role in that all tools can interface and exchange data with it. A repository, holds data fields and definitions and ensures that data integrity is maintained throughout the development lifecycle. As a consequence, ICASE allow tools to work together relatively seamlessly and alleviates much of the stop-start nature of non-ICASE environments.

All the CASE environments mentioned above are often rigid and do not support the users' native methodology nor different methodologies. To avoid this, more flexible and customisable tools, called *CASE shells*, are emerging. They allow customisation of the CASE shell to a given methodology. Users are able to describe their methodology either through a set of meta-modelling editors or through a set of formal languages and tailor it to their specific requirements in order to create dedicated CASE tools. Many of the products and research prototypes of CASE shells move towards the support of different methodologies during the development of a single information system.

6.2.2 Integrated Software Development Environments

Central to the issue of CASE integration, is the concept of an Integrated Software Development Environment (*ISDE*). ISDE as the term implies, provides support for the coordination of all the different activities that take place during a software project. There are different types of support that an ISDE can provide. Typical examples of ISDE support include:

- automated coordination of development activities. For example, mechanisms may be provided which trigger the design activity at the end of the specification activity

- mechanisms for inter-activity communication. This means that data produced, for example by a specification tool can be filtered and subsequently transmitted to the design tool, to the project planning tool etc.

A major approach towards ISDEs is the European project *PCTE* (Portable Common Tool environment [Boudier et al, 1988]). The PCTE approach is similar to the idea of a modern operating system such as *UNIX* which offers a set of standard facilities such as text formatters, filters, process communications etc. which can provide the building blocks for creating sophisticated applications. In analogy, PCTE provides common building blocks to the developers of CASE tools which will run under PCTE. In turn, this can facilitate compatibility between the tools since they all use common facilities for storing and communicating their data.

6.3 A Generic CASE Architecture

6.3.1 Overview

Central to any CASE architecture is the concept of *data dictionary* or *repository* [Bruce et al, 1989; Burkhard, 1989; Martin, 1989b]. The role of the repository is to store all the logical and physical objects whose task is to provide control and integration in the development and maintenance of information systems. Essentially, the repository is the single point of definition in the software life cycle. In this role, the repository holds the metadata (data about data) which not only defines what and where the data is, but how it relates to other data and how the logical manipulation of data is mapped across physical structures like databases, file systems and, ultimately, physical objects such as networks and CPUs [McClure, 1988; Martin, 1989a]. In terms of the development life cycle, this means that any program or flow chart which is used in

the building of an application and any tools which are employed in its construction, must receive and enter data to and from the repository.

The repository provides true integration of specifications from the different tools because it permits sharing specifications rather than converting and passing them between tools. CASE tools connect directly with the repository for specification storage and retrieval. The repository uses these specifications to drive an application generator and to generate operating systems commands, database calls, communication commands and user documentation.

A generic architecture for CASE tools is shown in Figure 6.1

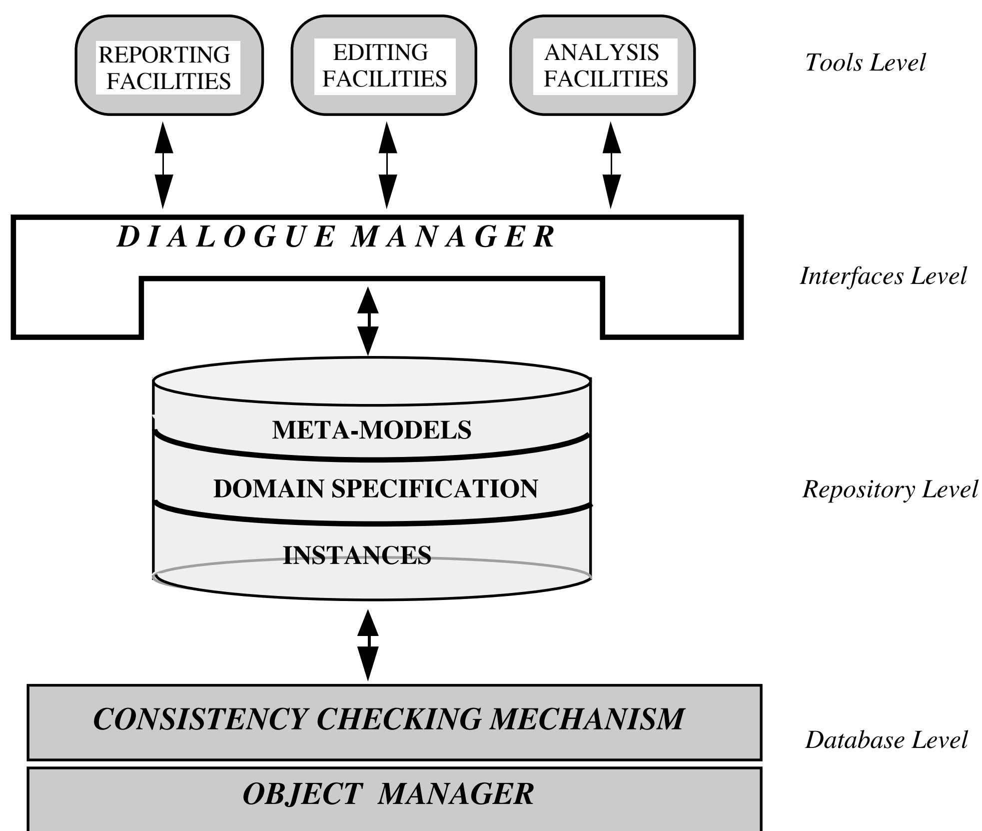


Figure 6.1: Architecture of a CASE tool

According to the architecture of Figure 6.1, a CASE tool consists of the following major components:

- *The repository*. This is usually a database or file system in which all the information about the current status of the development process is held. Depending on the particular phase of software development (i.e. analysis, design, testing etc.)

different types of data are recorded (e.g. diagrams, text, test data etc.) in a CASE repository. Note that a repository is frequently shared amongst different CASE tools. When all the tools are used within the same development phase (e.g. analysis) the repository must be capable of maintaining data about the individual projects that go on concurrently as well as about their inter-communication requirements. A repository that holds data from different development phases (e.g. analysis, design, etc.) must make them accessible from all tools which require them (i.e. analysis data must be accessible by the design tools, design data by the program generation tool etc.).

- *The assistant modules*. These can be considered as tools in their own right, responsible for performing some task within the particular development phase either in an entirely automatic fashion, or by assisting the user of the tool. In the requirements phase, assistant tools can, for example, automatically produce drawings (graphical models) from textual descriptions, check for consistency and completeness of the requirements models, propagate the changes in some of the models to the rest of the models (e.g. redrawing the corresponding graphical models when the user changes some textual descriptions etc.). In the following sections we will distinguish between assistant modules which are responsible for the most mundane/clerical tasks (such as screen drawing) and the ones which are capable of completely or in part undertaking intelligent tasks such as model validation, consistency maintenance etc.

- *The human-computer interface (HCI) component*. This component is responsible for handling the tool's communication with the user. The HCIs for CASE tools folow the general trends in user interface technology. There has been therefore a continuous evolution in the HCI technology employed by CASE tools from the early teletype style of interaction to the latest types of interaction using graphical user interfaces (GUIs) in multitasking environments.

- *The communications component*. The communications component is responsible for exchanging data with other CASE tools. This usually implies that the communications component receives data about the software project which was created in a previous development phase and transmits data for use by tools in subsequent development phases. Effective communication between CASE tools is a problem still to be solved in a satisfactory manner for a variety of reasons. The lack of widely acceptable standards between the CASE tool developers has

resulted in a plethora of proprietary and often mutually incompatible formats used by the tools for the internal storage of development information.

6.3.2 Architecture and Functionality of a Repository for Requirements Engineering

The task of a repository is to help manage Requirements Engineering data by offering a variety of services that promote data sharing, data integrity and convenient access. The repository can:

- help users logically associate the various products of the process (documentation, formal specifications etc.)
- keep track of users' annotations which contain explanations and assumptions
- manage different versions of requirements, and the associated documentation
- control different views of the system under development
- help the managerial side of a team development (e.g. estimate required development effort)
- maintain historical data about the decisions taken through the process of eliciting, specifying requirements.

A CASE environment places specific demands on the repository technology used, especially in the case of large-scale team projects. It must support simultaneous access by team members, editing, and authorship in a computer network. Different versions of requirements must coexist, where team members work independently and then merge their specifications back into the main project. Analysts must be allowed to build specific configurations and version trees, and subsequently merge versions together. In summary, a CASE repository supporting Requirements Engineering must exhibit functionality in terms of:

Complex Information Modelling. A repository for Requirements Engineering must be able to store large variable-length objects, such as documents and graphical specification models.

Integrity Constraints and Triggers. Due to the size and complexity of the requirements models, the maintenance of data consistency should be performed by enforcing constraints as the data evolves over time.

Transaction Management. Advanced transaction management features for collaborative Requirements Engineering must be provided. The classical notion of serializability of a transaction is no more adequate, as it significantly reduces concurrency and is largely unsuitable for Requirements Engineering environments at large.

Specification Updates. The process of Requirements Engineering is incremental by nature. It is therefore compulsory to provide the means for changing and updating freely the structure of a preliminary requirements model as modelled by the schema of the repository.

Data Sharing. One of the key issues in collaborative Requirements Engineering is the sharing of data between various analysts. As in client-server architectures, requirements data may be partitioned based on various criteria. However, data must be accessible by all.

Distribution and Cooperative Work. Effective communication protocols between analysts is essential, since they are often unaware of each others developments. This sometimes results in lack of coordination, reduced parallelism, a considerable waste of time and resources, and faulty specifications due to misinterpretations of data. Mechanisms to support the distribution of tools such as distribution of the repository itself or at least distributed access to a repository residing on a database server are needed. In the same spirit, tools for handling and controlling the distribution must also be provided.

Concurrency. The repository should offer the same services as current database systems with respect to the handling of multiple users.

Recovery. In case of hardware or software failures, the system should recover, i.e., bring itself back to some coherent state of the data.

6.3.3 Features of CASE Technology

This section discusses the properties of contemporary CASE tools used in the Requirements Engineering phase. The common characteristic of all such tools is that they are built in order to automate aspects of software methodologies. Most of these methodologies existed before the

introduction of CASE, albeit in a manual form. The origin of such methodologies is the paradigm of structured development which was initially applied to programming and subsequently to the design and analysis phases [Jackson, 1983] [Yourdon, 1989]. Structured methodologies are based on the principle of 'divide and conquer' i.e. on the stepwise decomposition of data and functional elements of a system to their parts. This approach ensures the efficient dealing of tasks of arbitrary size since the original task is progressively decomposed into smaller and easier to be dealt with sub-tasks. One serious drawbacks of the structured methodologies (especially when practised without tool support) is that they are usually cumbersome to apply, mainly because of the large amounts of information (documents, drawing etc.) they generate. The most important task of a CASE tool which supports a structured methodology therefore is to provide automated functions for the storage and manipulation of the information generated during software development.

The major application of CASE tools developed in the Seventies and early Eighties therefore, was to assist in the collection and manipulation of the voluminous information generated by the structured methodologies. The progress in the ability of CASE tools to store and manipulate project information, closely followed the progress in areas such as hardware (processor and graphics technologies), databases and computer communications.

Early CASE tools for Requirements Engineering support were handling requirements specifications, either as free text (in which case they were little more than word processors) or in a stylised form which allowed for some limited automatic processing of the requirements. Such automatic processing of requirements specifications usually included checking for *inconsistencies* (e.g. terms with multiple definitions) and *incompleteness* (i.e. terms which are used but not defined).

Advances in hardware technology such as high resolution graphics screens made possible the development of tools capable of manipulating graphical requirements models. The ability to manipulate graphical definitions has given an important boost to the acceptability of the CASE technology since the structured methodologies which the early CASE tools support, rely heavily on the use of graphical requirements models. Even in today's CASE the ability to draw and manipulate graphical specifications models on the screen remains the most heavily used and essential feature.

Technological progress, together with a more mature understanding of the Requirements Engineering (and more general of the software development) process, guided the incorporation of additional functionality in the CASE technology. Today's Requirements Engineering tools utilise the latest advances in processing, graphical user interfaces, database and communications

facilities in order to provide effective support to the most fundamental activities of Requirements Engineering. Whilst proprietary tools differ on a number of issues such as the methodology they support, whether they are standalone or parts of an integrated environment etc., they nevertheless share a number of features such as:

- Facilities for prototyping. The importance of prototyping in activities such as requirements acquisition and validation is now generally accepted. The increased use of prototyping as an important technique within Requirements Engineering influenced the CASE developers into incorporating facilities for prototyping in their tools. Basic prototyping facilities usually encountered in CASE tools include report generation and screen painting. More advanced prototyping facilities include animation and symbolic execution of the requirements models. Also limited code generation (which allows for a crude program to be quickly generated from the specifications) is a facility offered by the more sophisticated tools.

- Facilities for data management. Whilst, the early CASE tools used simple file structures for storing the requirements models, contemporary ones utilise database technology in order to offer facilities such as concurrent access to the data by many developers, version control etc.

- Inter-tool Communications facilities. As the relationship between Requirements Engineering and other development phases becomes better understood, so does the ability of requirements CASE tool to communicate with other CASE responsible for tasks such as project planning, configuration control, design etc.

- Graphical user interface. It allows the representation of requirements models using the graphical notations used by the methodology which the tool supports . Usually the tool proves facility for dynamic redrawing of the model on the screen, each time the user changes some part of it. This facility significantly reduces the time it would take to manually redraw the model from scratch.

- Data administration. Managing the data generated during Requirements Engineering entails tasks such as keeping lists of the various types of data, enforcing standards (e.g. that the names of the various concepts follow certain naming conventions), checking for inconsistencies and omissions (e.g. duplicate names).

- Utilities for requirements animation and prototyping. Such utilities may include screen painters, program generators etc. which support the requirements validation process (Chapter 5)

- Data communication facilities. This includes facilities for importing and exporting data to and from other tools. A Requirements Engineering tool usually exports data to design tools and to project planning tools which use such data to inform about the current progress with the project and to plan ahead).

6.4 Selecting, Integrating and Using CASE tools for Requirements Engineering

6.4.1 Desired features of Requirements Engineering CASE

Modern CASE technology integrates the different categories of CASE that were mentioned before into the concept of an *Integrated Software Development environment* (ISDE for short). Very often therefore, these days, the prospective buyer/user of CASE has to choose between different ISDE options rather than the stand-alone CASE which is gradually becoming a rarity. Assuming that the dilemma of choosing between the ISDE and standalone one is circumvented, a number of technical and organisational issues that must be considered in the CASE selection process arises, i.e.:

- Support for specific formalisms and methodologies. The vast majority of commercial CASE support only one or two of the mainstream software methodologies, e.g. Structure Analysis [Yourdon, 1989] Jackson System Development [Jackson, 1983] Structured Systems Analysis and Design Method (SSADM) [NCC, 1990], Information Engineering [Macdonald, 1986] etc. Usually, there is little flexibility for customisation of the models used by the tool to suit the individual users' need. One exception, to this is the class of Generic or Meta-CASE [Alderson, 1993] tools which allow the customisation of existing models or even the creation of entirely new ones to support in-house developed methods.

- Support for specific types of applications. Although, most CASE tools can be used with varying degree of success for any type of application (e.g. data intensive, real time, process control, expert system etc.), some formalisms and methods better accommodate the needs of specific application types. It is essential

therefore that, for example, CASE used for real time applications should provide support for formalisms such as StateCharts, Petri Nets etc., whilst data intensive CASE should be able to handle the entity-Relationship model or some of its variants.

- Repository capabilities. The majority of the modern CASE tools provide repository facilities similar to those discussed in Section 3. It is important however that a checklist is drawn by the prospective user which includes the following capabilities: Fully integration of data in the repository, database facilities, project management information, shareability of data amongst developers, automatic report generation and support for ad-hoc querying, import/export capabilities to other repositories, automatic analysis and control of changes in the data.

In the majority of real life software projects, even the most sophisticated CASE or ISDE tool on its own will fail to meet one hundred percent the demands of the requirements phase. One possible way to overcome this is by acquiring and using a set of different CASE with complimentary abilities, e.g. a diagramming tool combined with a prototyping tool and an executable formal specification language. An obvious problem in this approach is cost, since three times as many tools (compared with the one-tool option) will have to be purchased. Even if cost is not an obstacle, communication between the tools is also a potential source of difficulty. In many cases, special software (called *bridges*) will have to be written to allow the transferring of data between tools. Hopefully, this situation will gradually become a rarity as we move towards standard environments and repositories for CASE.

6.4.2 Integrating CASE tools

Integration and communication can be aimed at three levels.

The first level is that of *tools integration.* An example approach for tool integration is presented in Figure 6.2. The tools themselves are responsible for data structuring and control activities; in this way co-operation between tools is achieved through passing of streams of bytes. The advantage of this approach is that tools can be developed almost independently and generic file transfer utilities can be used for the communication between tools. There are however, many disadvantages with this approach such as difficulty in expansion, duplication of role and effort in tools, manual co-operation between tools rather than assistance in a team effort and loss of relationships between design data.

The second level is that of *data integration.* A set of data structures is agreed by all tools within an environment. A meta-schema is agreed upon by all tools before any development of these tolls commences. Any future expansion will need to conform to this meta-schema. The advantage of this approach is that most of the actions necessary for analysing, validating and converting data structures are no longer required within each tool. The disadvantage is that the way that the tools are used is not constrained or guided in any way. This can only be achieved by agreeing not only on the data structures but also on the *process* within which the tools will be used. The interest lies only in the management of information as a consistent whole and not how parts of the information are transformed or operated upon.

The third level is that of *method integration.* A set of data structures and the process model are agreed by all tools which then interact effectively in support of a defined process.

Data integration and method integration imply that the tools communicate by interacting at a level of abstraction higher than the operating system i.e. there is a project level interface which provides facilities and services for controlling the requirements specification process.

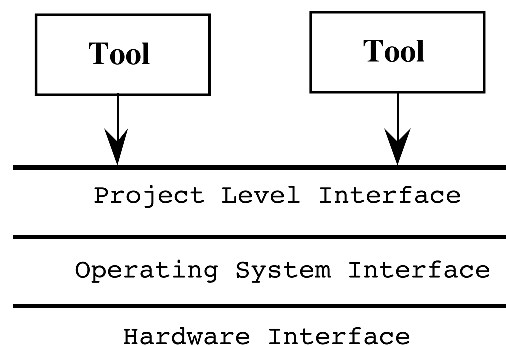


Figure 6.2: Tool Integration

6.5 Research CASE for Requirements Engineering

6.5.1 Introduction

In contrast to commercial CASE, research tools for Requirements Engineering do not fall easily into predefined categories. Generally, CASE research prototypes do not attempt to provide complete or integrated support to the tasks of Requirements Engineering, focusing instead on specific problems such as acquisition, specification, validation etc. Such tools have already been discussed in Chapters 3 and 6.

Research prototypes are usually concerned with proving the validity, feasibility or applicability of a certain paradigm for practising Requirements Engineering. A paradigm, in general, is a specific way of doing things (solving problems) and as such it cannot easily be proved true or false. Research paradigms for Requirements Engineering often materialise as 'tools' or 'environments' which are exclusively used in controlled experiments rather than real-life applications. Eventually, some of the research ideas find their way to real life practice by being incorporated in commercial tools and methodologies.

In recent years, the paradigm that has received the most attention in Requirements Engineering research has been *Knowledge-based Requirements Engineering*. The rationale behind viewing Requirements Engineering as a knowledge-based process has been discussed in many different occasions in previous chapters of this book. Viewing Requirements Engineering as a process which relies to a large extent to the availability of knowledge of various sorts, invokes a number of research issues:

- what types of knowledge is used in Requirements Engineering?
- how can such knowledge be formalised and represented within computers?
- how can the availability of this knowledge improve and sometimes automate the practising of Requirements Engineering?

Research addressing the above questions, has produced tools which can represent and store knowledge about the domain the software application belongs to (*Domain Knowledge*). This knowledge is used for purposes such as completing and validating the requirements model. Reusable requirements knowledge speeds up the overall process (since less interaction with the users is required) as well as improves the quality of the requirements model. Problems that are still to overcome in this approach are related with the difficulty of identifying suitable sources of reusable requirements knowledge and the cost of eliciting and formalising reusable requirements models.

Another source of knowledge which some research tools are based upon is method knowledge. It is fairly easy to automate the steps of a requirements method defined in an algorithmic way which has well defined inputs ad outputs. Structured methodologies fall into this category, i.e. some of their steps and deliverables can be applied mechanistically and are therefore suitable candidates for automation. Unfortunately, this cannot be said for the more unstructured processes such as elicitation and validation. Tools which attempt to (even partially) automate

such processes therefore, are equipped with knowledge and methods of reasoning which mimics human knowledge and reasoning. Such tools are frequently called *intelligent requirements assistants* [Anderson and Fickas, 1989] [Reubenstein and Waters 1991]. Again, before such tools acquire commercially applicable a number of important issues related to their ability to stand up to real life software systems requirements must be resolved. Figure 6.3 shows a generic architecture for an 'intelligent' Requirements Engineering CASE tool.

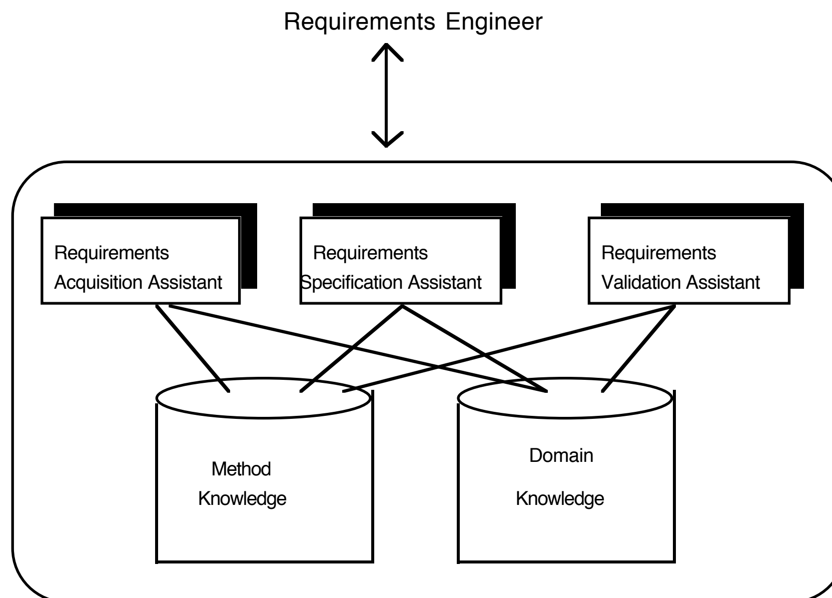


Figure 6.3: Architecture of an 'Intelligent' Requirements Engineering Tool

6.5.2 Intelligent CASE in Requirements Analysis and Specification

In general, these tools can be distinguished along two dimensions. The first dimension is concerned with tools supporting *building of the specification* whereas the second dimension is concerned with those supporting the *management of the specification and the building of the application out of a given system specification.*

The requirements for the next generation methods and CASE environments that are discussed in this section revolve around the activities of requirements capture and analysis, validation and management of the captured knowledge. These issues give rise to two lines of investigation regarding the requirements for the next generation development methods and CASE environments:

- Improved tools and techniques for assisting the *process of deriving a conceptual specification* of an enterprise.

- Improved tools and techniques for managing a conceptual specification and the building of the application out of a given system specification once such a specification has been developed.

These requirements give rise to a number of research issues discussed in the following sections that aim at providing intelligent support facilities during the process of conceptual specification design, conceptual specification management and the generation of the application itself.

6.5.3 CASE for Conceptual Specification Design

During the last decade, an intensive effort has been made by the industrial community as well as the research community to develop conceptual modelling formalisms that allow to describe Information System (IS) in high level terms, the so-called conceptual schema, and to reason upon this description. However, little effort has been paid to model the process by which one can reach the conceptual schema of an IS to be built. In other words, there exists a plethora of formalisms for the representation of the Requirements Engineering (RE) product whereas the number of techniques which deal with the RE process is very small (see also Chapter 2). As a consequence, CASE tools only concentrate on supporting the *population* of the repository in that they assist the requirements engineer to (1) enter the RE product in a diagrammatic form (2) store its contents in a repository and (3) document it. They do not help the requirements engineer in *constructing* the requirements product itself by supporting the transition process from an informal requirements description to a formal IS specification.

The upperCASE strand of research is strongly influenced by the application of Artificial Intelligence (AI) techniques. The purpose of applying AI techniques is to better understand and consequently, formalise the conceptualisation process. In contrast with the lowerCASE approach, emphasis is placed on the way that requirements are acquired and the way that these are transformed to populate the conceptual schema of the business and the information system.

The system development process is characterised as non-deterministic because of the difficulties in identifying the limits of the problem area, the scope and goals of the information system and the approach to conceptual schema definition. However, from a management point of view, developers wish to control the development process through the use of formal techniques and experimental knowledge.

Taking these two dimensions of the information system development process into account, it is clear that its automation cannot be based solely, on a pure algorithmic solution. This has been recognised by some researchers who developed CASE prototype toolsets based on an expert system approach (SECSI [Bouzeghoub, 1985], OICSI [Rolland, 1986; Cauvet, 1988]). In such an approach, both experimental and formal knowledge are represented in the knowledge base whereas the application domain knowledge is stored within the fact base. The development process is viewed as a knowledge based process which, progressively, through the application of the knowledge base rules on elements of the fact base, transforms the initial requirements into the final information system conceptual schema.

Some approaches to process modelling attempt to employ metamodelling formalisms and toolsets to explicitly represent the knowledge about the development method as well as the development product. The SOCRATES project [Wijers, 1991] follows this approach in that it aims at offering automated support of the information modelling processes by representing and manipulating the experienced practitioners' information modelling knowledge. This approach advocates a model independent architecture where the designer will be able to describe his method and the underlying modelling process using a number of formal languages.

Other approaches, attempt to automatically acquire the knowledge used in the analysis process. Whereas in most of the approaches the development process knowledge is provided by human experts, here it is deduced using automatic learning techniques. For instance, the INTRES tool [Pernici,1989] uses explanation based generalisation for specifying static properties of elements of well understood applications, based on examples of documents. In [Mannino, 1988], the approach is based on the strategy of learning from examples employed in a form definition system, they develop specific learning algorithms. The tool automatically induces the form properties and some functional dependencies. This work has been extended further in [Talldal, 1990] where the learning algorithms are optimised and the induced conceptual schema is expressed in an extended ER formalism.

Summary

This Chapter has been concerned with Computer Aided Software Engineering technology as applied to the Requirements Engineering phase. In the last two decades we witnessed a changing role of CASE in Requirements Engineering from simple clerical task handling to automating essential activities such as formal specification and validation.

Today's State-of-the-Art CASE tool acts as an editor and repository for the various models created during Requirements Engineering (discussed in Chapter 4) as well as an interface between Requirements Engineering and other software development phases. CASE technology particularly suits those Requirements Engineering methodologies which rely on graphical models, mainly because such models can be quickly drawn, manipulated and communicated (i.e. by using techniques such as prototyping, animation etc.) to the users.

The application of CASE technology to Requirements Engineering has been less effective when dealing with more 'hard' problems such as requirements elicitation and formal validation. Research however into intelligent CASE attempts to overcome the limitations of today's tools by utilising expert system technology as well as our improved understanding of the processes involved in Requirements Engineering.

An architecture for a Requirements Engineering CASE tool of the future is presented in Figure 6.4. Such a tool will be capable of performing tasks which are currently the responsibility of the human engineer. These will include:

- automatically completing the requirements model by utilising preexisting reusable requirements knowledge

- advising and supporting the human requirements engineer using domain and method-dependent knowledge and rules

- validating the requirements model using techniques such as natural language paraphrasing, symbolic execution etc.

- generating test cases directly from the requirements.

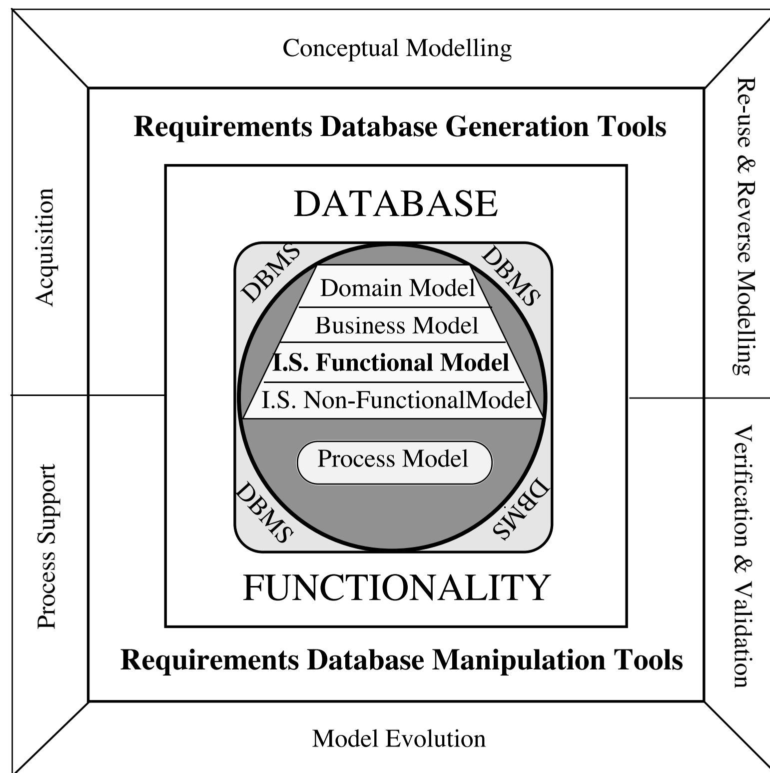


Figure 6.4: An Architecture for the Next Generation of CASE Tools

Such tools might be expected to lead to the CASE environments of the future that will support a very wide range (if not almost all) of stages in systems development with special attention paid to the specification of requirements. A CASE environment will then deal with the following functionality at the Requirements Engineering level:

- *Editing*. Inserting new information in the corporate knowledge base.
- *Browsing*. Viewing objects and sets of objects at various levels of detail.
- *Querying*. Editing and executing queries pertaining to the stored corporate knowledge base.
- *Reporting Facilities*. The reporting facilities range from simple access to data to more elaborated results used in decision support activities.
- *Analysis Facilities*. Analysis facilities will include scenario building and evaluation using different approaches including visualisation techniques.

References

Alderson, A. (1993) *Meta-CASE Technology*. IPSYS Software plc, Macclesfield, Cheshire, UK.

Anderson, J. & Fickas, S. A (1989) *Proposed Perspective Shift: Viewing Specification Design as a Planning Problem.* In Proc. 5th Int'l Workshop on Software Specification and Design, Pittsburgh, PA, IEEE.

Boudier, G., Gallo, F., Minot, R. & Thomas, M. J. (1988) *An Overview of PCTE and PCTE+*. In proc. 3rd ACM Symposium on Software Development Environments, October.

Bouzeghoub, M., Gardarin, G., Metais, E. (1985) *Database Design Tool: an Expert System Approach*, Proc. of the 11th VLDB Conference, Stockolm, August 1985.

Bruce, T. A., Fuller, J., Moriarty, T. (1989) *So You Want a Repository*, Database Programming and Design, May 1989.

Burkhard, D. L. (1989) *Implementing CASE Tools*, Journal of Systems Management, March 1989.

Cauvet, C., Rolland, C., Proix, C. (1988) *Information Systems Design: an Expert System Approach*, in Proc. of the Int. Conf. on Extending Database Technology, Venice, March 1988.

Fuggetta, A. (1993) *A Classification of CASE Technology*, IEEE Computer, Vol. 26, No. 12, 1993, pp. 25-38.

Gibson, M. L., Snyder, C. A. and Carr, H. H. (1989) *CASE: Claryfing Common Misconceptions,* Journal of Information Systems Management, 7(3), 1989.

Jackson, M.A. (1983) System Development, Prentice Hall.

Macdonald, I.G. (1986) *Information Engineering: An Improved, Automated Methodology for the Design of Data Sharing Systems*, in Information Systems Design

Methodologies: Improving the Practice, Olle,T.W et al (eds), IFIP TC8, North-Holland.

Mannino, M. V., Tseng, V. P. (1988) *Inferring Database Requirements from Examples in Forms*, 7th Int. Conf. on Entity-Relationship Approach, pp1-25, Rome, Italy, 1988.

Martin, J. (1989a) Information Engineering: Volume 1, Prentice Hall Inc., New Jersey, 1989.

Martin, J. (1989b) *I-CASE Encyclopedia Brings Consistency to IS*, PC Week, January 1989.

McClure, C. (1988) CASE is Software Automation, Prentice Hall Inc., New Jersey, 1988.

NCC (1990) National Computing Centre. SSADM Version 4 Manual, Manchester, UK.

Pernici, B., Vaccari, G., Villa, R. (1989) *INTRES : INTelligent REquirements Specification,* Proc IJCAI'89 Workshop on Automatic Software Design, Detroit, Michigan, USA, August 1989.

Reubenstein, H. B. & Waters, R. C. (1991) *The Requirements Apprentice:Automated Assistance for Requirements Acquisition.* IEEE Trans. on Software Engineering, Vol. 17, No. 3.

Rolland, C., Proix, C. (1986) *An Expert System Approach to Information System Design*, in IFIP World Congress 86, Dublin, 1986

Talldal B., Wangler B. (1990) *Extracting a Conceptual Model from Examples of Filled in Forms*, Proc. of Int. conf. COMAD, pp 327-350, N. Prakash (ed), New Delhi, India, Dec 1990.

Wijers, G. M., der Hofstede, A. H. M., van Oosterom, N. E. (1991) *Representation of Information Modelling Knowledge*, Proc of the 2nd Workshop on The Next Generation of CASE Tools, Trondheim, Norway, May 1991.

Yourdon, E. (1989) Modern Structured Analysis. Prentice-Hall.

View publication stats