

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 <u>radar messages</u> from a <u>short-range radar</u> . The <u>scan-period</u> of the <u>radar</u> is 4 seconds. The <u>frequency</u> is 2.6-2.7 Ghz. The <u>pulse-repetition interval frequency</u> is 1040 Hz. The <u>number of tracks</u> shall be for 200 aircraft. The <u>band-rate</u> is 2400. The <u>message-size</u> is 104 bits/message. The system shall begin tracking aircraft that are within 2 miles of the controlled area. <u>Track initiation</u> 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 scenarios 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 $G1, G2, \dots, Gn$. The meaning of the *AND* decomposition is that in order for objective *Ob* to be attained, *all* goals $G1, G2, \dots, Gn$ must be attained. The other kind of objective decomposition is an *OR* decomposition. If objective *Ob* is or-decomposed into goals $G1, G2, \dots, Gn$, then for objective *Ob* to be attained it is sufficient that *any* of $G1, g2, \dots, Gn$ 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.

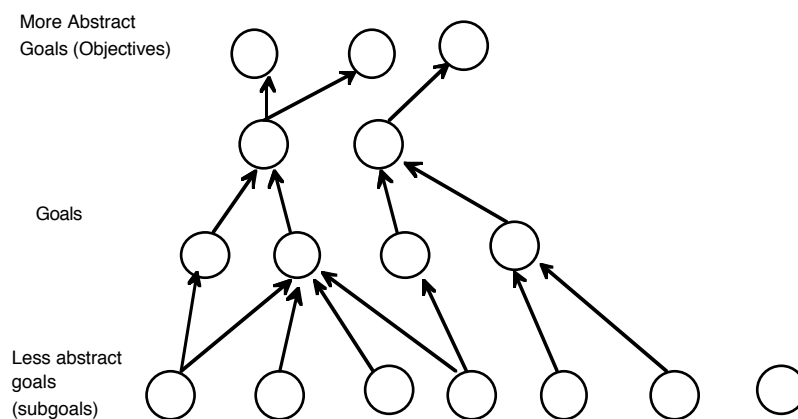


Figure 3.2. Goal Hierarchies.

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 scenarios 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.

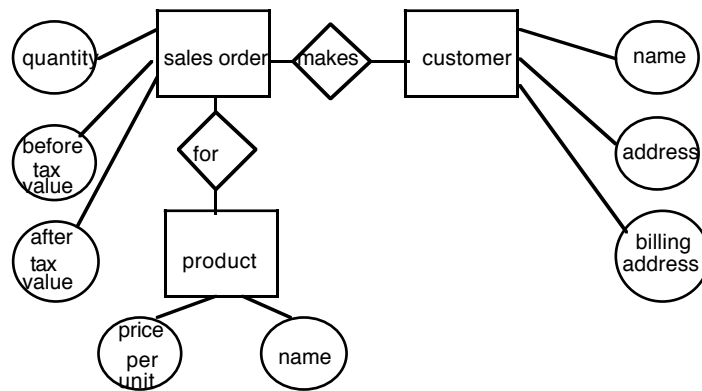


Figure 3.4: An entity-relationship model of a sales form contents.

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 thumb'.

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_salary* 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.

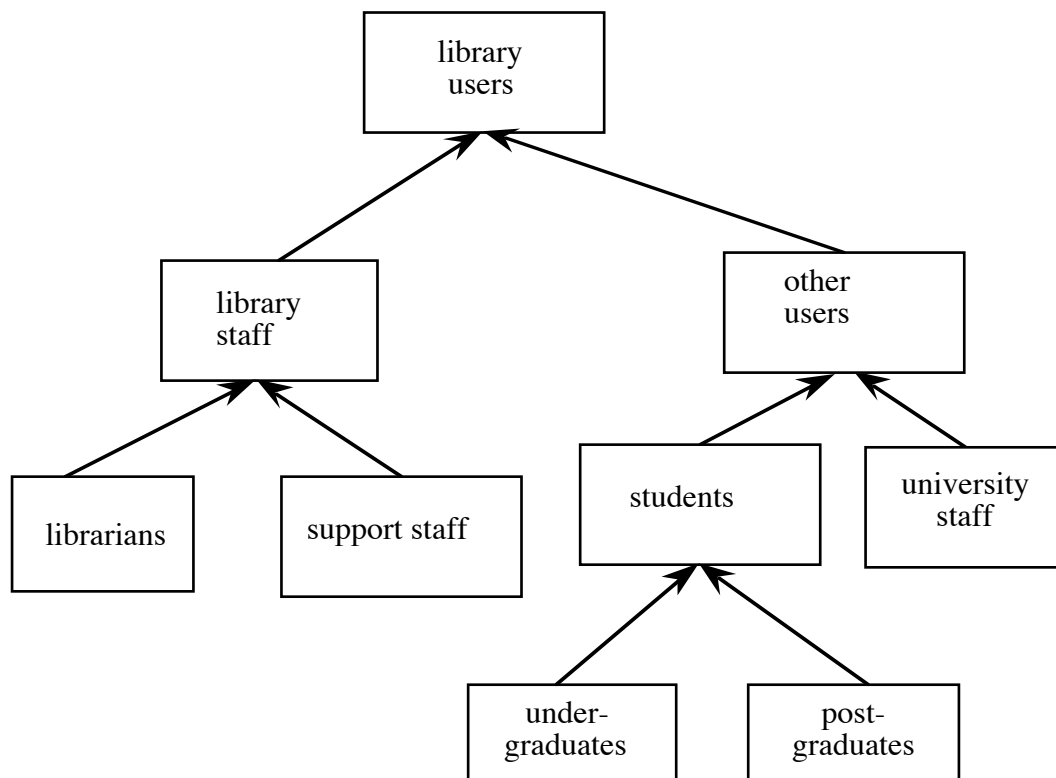


Figure 3.5: Hierarchy of library users

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 validity 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 ethnography 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 scenarios 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 World 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 Jirotko 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 Methodology 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 in 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*. Software 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. Sølvsberg (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. Jirotko 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 Design 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 into The 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.