

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.

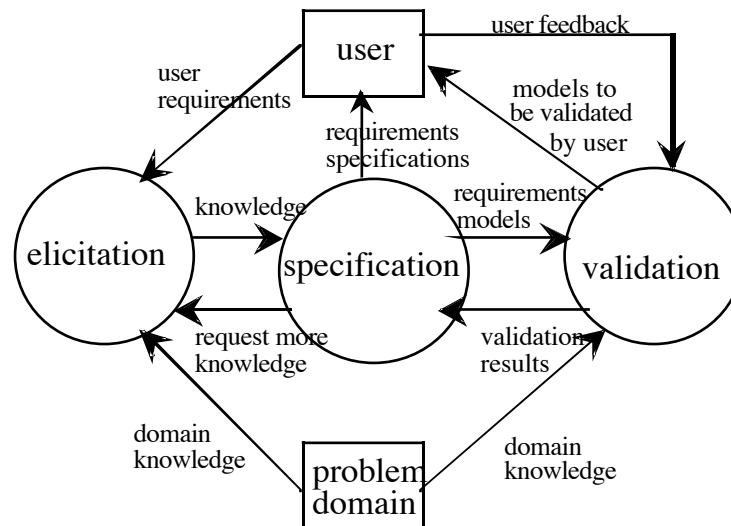


Figure 2.1: A Framework for Requirements Engineering Processes

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.

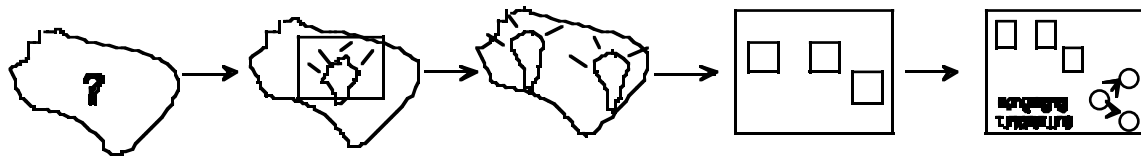


Figure 2.2: A succession of models is created in Requirements Engineering.

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 identification*, [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.

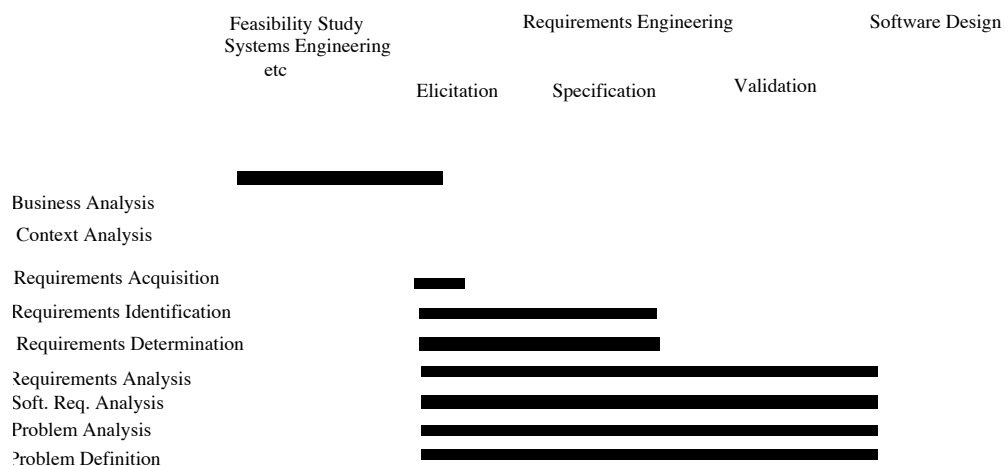


Figure 2.3 Coverage of Requirements Engineering activities by different approaches

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 assess 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 assess 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.

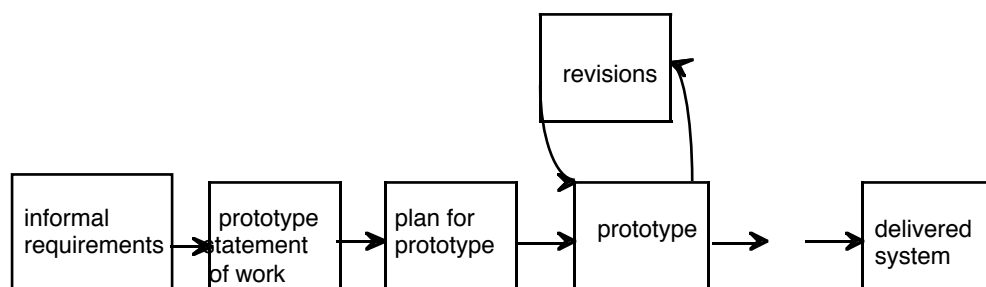


Figure 2.4: Schematic representation of the prototype model.

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.

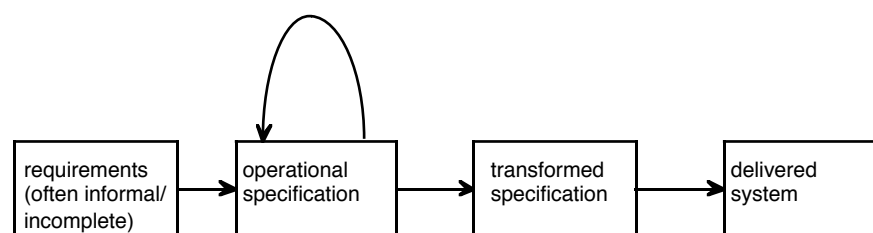


Figure 2.5: The operational paradigm

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.

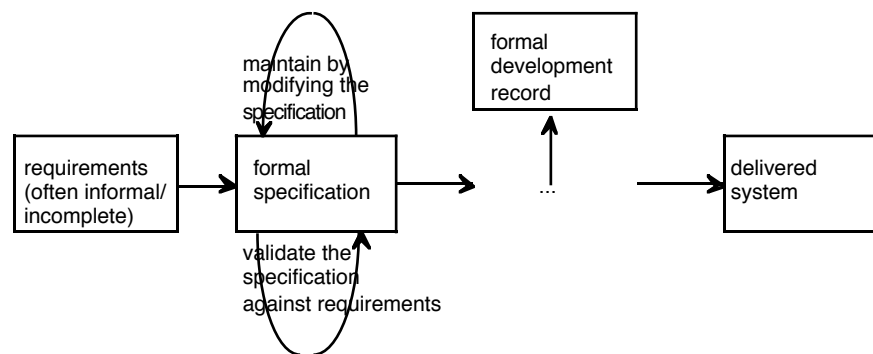


Figure 2.6: The transformational model

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.

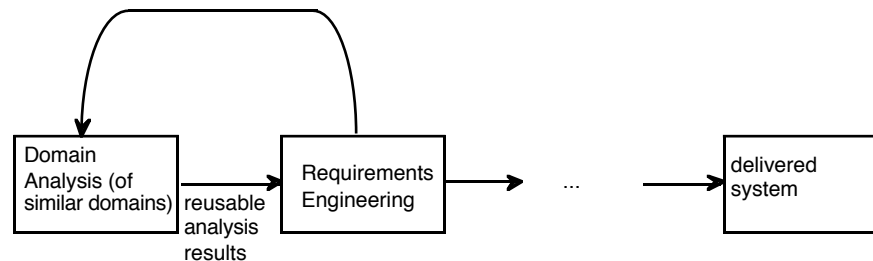


Figure 2.7: The Domain Analysis Approach

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 satisfies 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.

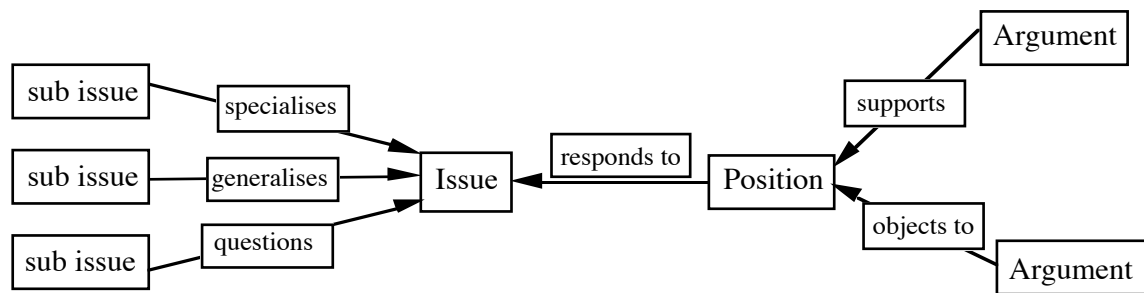


Figure 2.8: The IBIS model

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.