Appears in P. Quintas (ed.) "Social Dimensions of Systems Engineering: People, processes, policies and software development". London: Ellis Horwood, pp144-164, 1993.

Negotiation and the Role of the Requirements Specification[1]

STEVE EASTERBROOK
School of Cognitive and Computing Sciences
University of Sussex,
Falmer, Brighton, BN1 9QH
<Easterbrook@uk.ac.susx.cogs>

The term *requirements engineering* describes the processes leading to the production of a requirements specification. Much of software engineering research takes the existence of this document for granted, concentrating instead on the downstream areas of software development. In this chapter, we argue that the problems of requirements engineering deserve greater study. To understand why this is so, we consider the role of the specification in the software engineering process, and describe issues which must be addressed during specification construction. The difficulties of requirements engineering come from many directions, including the sheer quantity of knowledge involved, the inherent uncertainty, and the need for negotiation where there are conflicting requirements. We conclude that a prescriptive framework to support negotiation of requirements is highly desirable, and describe a number of objectives for such a framework.

1. Introduction

The phase of the software engineering process that begins with an informal statement of need and produces a requirements specification is generally referred to as *requirements analysis*. Ideally, the resulting document should contain all the information about the requirements that might be needed during the design stage, although in practice, the clients' perceived needs will change during the lifecycle of the system. If the requirements aren't clearly defined, the result is uncertainty throughout the software life-cycle.

The importance of the requirements specification implies that a great deal of effort should be invested in the creation of such a document (Boehm, 1981). Creating the specification is far more than just analysis: it involves eliciting relevant knowledge; understanding the task and its social and organisational context; negotiating with the client over the scope, contents and language; resolving conflicting requirements; and synthesising appropriate structures for describing the result. Use of the term *Requirements Engineering* has been proposed to indicate the complexity of this process, and to convey the message that specifications need to be carefully constructed.

We will begin by examining the importance of requirements engineering in the software process, concentrating particularly on the roles played by the specification. We will then consider how specifications are constructed and validated, and discuss the difficulties. Finally, we propose that requirements engineering be seen as a process of negotiation, in which the importance of conflict is recognised and addressed.

2. The Role of Specifications

Specifications have a vital role to play in the software engineering process, as precise descriptions of needs. A specification provides a way to verify the correctness of the eventual design and implementation. If the specification is inappropriate, verification will be pointless.

[1]This paper was presented to the workshop on Policy Issues in Systems and Software Development, organised by the Science Policy Research Unit, University of Sussex, and held in Brighton, July 1991.

Information omitted from the specification will not be taken into account in the design process. Ambiguities in the specification lead to uncertainty throughout the process. Misunderstandings and errors in the specification will lead to designs which, while complying with the specification, do not properly satisfy the needs of the users.

We can identify three important roles which the requirements specification is expected to play. First and foremost, this specification forms a contract between the clients and the software developers. Secondly, it is the main communication channel between the developers and the clients. Thirdly, the specification establishes the commitment of the people who contribute to it.

The principle role of the specification is that it forms a contract between the clients and the software developers. This contract states what is expected of each party, and in particular sets out requirements and constraints for the deliverable software system. Balzer & Goldman (1979) describe three criteria by which a specification should be judged, namely: it must be clearly and unambiguously understandable by both parties; it must be testable that any implementation satisfies the specification, and that the specification meets the needs it is designed for; and it must be easy to modify, as the requirements will change over time. These criteria reflect the contractual nature of specifications.

The specification also acts as a channel of communication amongst the software team. As the main source of information about the clients' needs, it defines what will be common knowledge among the developers. Too often, the specification does not adequately fill this role, and a recent field study of behavioural aspects of software developers (Curtis, Krasner, & Iscoe, 1988) concluded that many software teams depend upon a single *exceptional designer*. Such designers are characterised as having a deep understanding of both the application domain and the design process. In such cases, this designer is a better source of information than the specification. However, the development team will not all have equal access to such a person, and so will be working with different amounts of knowledge. Clearly, it is preferable to express as much of this knowledge as possible in the specification, in order to ensure dissemination.

Because of its accessibility, the specification ought to facilitate co-operation between the various parties. However, there is evidence that it frequently fails to do this. The study described above suggested that exceptional designers are able "to integrate different, sometimes competing perspectives on the development process". In other words, the specification is only providing one perspective, and there are other important points of view which it has excluded. This leads to two major problems, namely: whole sets of knowledge and ideas are ignored by the specification; and the people concerned will lose faith in the specification. The software team can become fractured, as such people attempt to exert influence in other ways, and will become dependent on the existence of a team member with the background knowledge and communication skills required to resolve the problem.

These problems are indicative of a third role that a specification plays: it provides a way of securing the commitment of the contributors. As the specification is supposed to represent the users' needs, it affects the attitude of those users towards the development process and its products. For this reason the specification should be (and be seen to be) representative of the many people that contribute to it (Zave, 1982). Potential users are unlikely to co-operate in the development, nor accept the final system if they feel their views have not been taken into account. In the worst case, participants have to circumvent the specification in order to get an issue raised.

We have identified three major roles of the specification. The first two of these are well known: the specification is both a communication medium, and a yardstick by which design and implementation are judged. In order to fulfil these two roles, the specification must be unambiguously understandable, testable, and modifiable. The trend towards formal specification languages is an attempt to satisfy at least the first two of these criteria. The third role is as a token of commitment, and to meet this role, the specification should be representative.

We have referred throughout this discussion to specifications as though they are of a uniform type. In fact, several types of specification can be distinguished, which are derived in different ways and have different uses. For example, it is common to distinguish between requirements specifications and design specifications, where the former describes needs, and the latter describes how those needs are to be met. While this discussion is directed at requirements specifications, other types of specification may play similar roles within the portion of the lifecycle in which they are used.

2.1. Constructing Specifications

As the specification has several important roles to play, it needs to be carefully constructed. It will be read by a number of different people, with widely differing backgrounds, and so must be accessible to them. Presentation is therefore important. It is essential that the specification should answer the types of question that various groups of people are likely to ask of it. In other words, it should be easy to interrogate. Above all the specification should be regarded as a designed artefact, itself created to fill certain needs.

Throughout the life of a project, pressures will arise for changes to the specification. These have many causes, from understanding gained in attempting to satisfy the current specification, to changes in the environment of the system. Where such changes are incorporated, the specification should be modified accordingly, in order that it remain up-to-date. If the specification is not up-to-date it will fail in its communicative role. Specifications, therefore, must be easy to annotate and modify. Although some would argue that the contractual nature of specifications implies that they should not change, it is clearly preferable to negotiate a change in the specification than to deliver software which does not meet the evolving needs of the user organisation.

The specification’s role as a source of information, and the need to allow modifications, indicate that the notion of a requirements specification as a large printed document is too inflexible. Rather, the specification should be seen as a form of repository, and it should include some form of database or knowledge base. Work on knowledge-based systems has provided some useful ideas both for modelling requirements (e.g. Borgida, Greenspan & Mylopoulos (1985)) and for reasoning with the specification (e.g. Reubenstein (1990)). A knowledge based component can also facilitate access to and management of the body of information gathered during requirements engineering. Such a component can be used to interrogate the specification throughout the lifecycle.

2.2. Validating Specifications

The specification must be sufficiently precise to determine whether subsequent designs and implementations meet it, a process known as *verification*. If the specification is in a formal language, the verification process can be mathematically rigorous (Bjorner, 1987), and to a certain degree, automated.

However, verification does not ensure that the specification is correct. The specification must be *validated* with regard to the actual, evolving needs of the client. This process is particularly difficult, as there is no definitive statement of those needs: the requirements specification *is* the first precise description of those needs. Because of this, validation cannot be formalised, and must remain a subjective human activity, requiring input and discussion by the originators of the needs (Blum, 1985).

Validation can only proceed if the participants can relate the specification to their needs, and is only successful if the specification is relevant to those needs. An important facility for validation is traceability (Alford, 1977). If components of the specification can be traced back to the original statements that inspired them, then the participants can assess the relevance more readily. Also, if a statement has been misinterpreted, this can be traced through to those parts of the specification which are based on the error and these can be modified.

Validation is an important part of requirements engineering. As it requires the originators' participation, it is likely to be considerably smoother if those people have participated throughout the requirements process. Such an involvement means that the requirements can be validated as they evolve, rather than when they have been refined into a specification.

2.3. Design Capture and Rationale

Often the ability to trace a component of the specification back to an originating statement is not enough to understand its purpose. For this, the process that led to the current specification needs to be recorded. Given that specifications are designed artefacts, then recording the derivation is a form of design capture. The design history must record the decisions made and their rationales in order to be of use.

Decision capture can be problematic, as rationales tend to be idiosyncratic (Kaplan, 1989). Analysts are experts in their jobs, and may have difficulty explaining their actions to others who need to understand the specification. Furthermore, explanations are usually tailored to a particular audience. If the analyst is recording a rationale, it is not clear to whom it should be aimed. Also, understanding decisions involves making the goals of the participants clear, which is difficult as these goals are often unconscious, and involve many implicit assumptions. There needs to be a way to prompt for these goals, and to encourage all participants to think about the decisions involved.

While it is unlikely that requirements engineering can be automated, some degree of automation might be introduced for recording the process. Where interactive tools are used, the operations can be recorded, automatically, as a basis for the attachment of rationales, using a *machine-in-the-loop* (Green, Luckham, Balzer, Cheatham, & Rich, 1983). To a certain extent, such tools can prompt for the rationales underlying the operations being carried out, and automatically record these. The entire process history represents a documentation of the requirements engineering process, and should be stored with the specification as a supplementary source of information. However, storing and manipulating this documentation is a huge knowledge management task, from which the analyst should be freed.

2.4. Exploration and Replay

While software engineering aims to produce higher quality software, it is not always possible to get it right first time (Brooks, 1975). All models of the software lifecycle allow for a degree of feedback and revision. The causes are well known: no-one has perfect foresight to predict what they will want in the future; clients are not even certain about what they want now, nor what is possible; and the consequences of particular requirements cannot be foreseen (Swartout & Balzer, 1982). Furthermore, the introduction of a new system itself generates new requirements.

All these problems suggest that some form of exploration is desirable. For the later stages of software engineering, exploratory programming has been proposed. The specification process, on the other hand is naturally an exploratory process, in which the participants explore their requirements. Such an exploration is essential, because analysts will be unsure of the clients needs, and the clients will be unsure of what is possible. Once an initial specification is produced, clients will want to explore how it relates to their needs before accepting it.

A useful tool for exploration is the ability to re-trace steps, undoing previous actions. This allows the participants to explore the consequences of a particular development without having to commit themselves to it. In a large project, however, this can be problematic since the action to be undone may have been originated by someone else, on a previous date. The facility therefore depends on the accurate recording of rationales and the ability to trace dependencies.

A related facility to the undo operation is the ability to replay parts of the process. This can allow re-use of previous, similar systems, by replaying their development, making changes where necessary. It can also simplify program evolution, as alterations can be made to the specification, and the development process replayed to generate a new implementation. Again, this depends on the capture of rationales and tracing of dependencies.

3. Difficulties

Specification construction is a difficult task. It is of a type of problem that has been termed *wicked:* it is ill-structured and open-ended, and the knowledge available is incomplete (Rittel & Webber, 1973). Most importantly, there is no notion of a finished specification, and the only criteria for stopping is some form of satisfaction.

Five major areas of difficulty can be identified as contributing to the problems associated with the formulation of requirements. The knowledge sources are diverse and vary greatly in the quality of information and exposition; the actual knowledge takes many different forms; the contents of the specification need to be negotiated; conflicts need to be resolved; and the knowledge itself is uncertain and unstable. The problems are compounded by the nature of the software engineering process itself, in that the sheer size of the process makes effective communication vital, for which a solid common understanding of the problem is needed. Furthermore, the resulting specification must be (to some degree) consistent and complete.

3.1. Requirements Formulation

Before the specification can be constructed, the necessary groundwork must be laid, which involves knowledge elicitation and the formulation of the requirements. Both of these are difficult tasks. Elicitation of knowledge forms a major part of the process, whether or not it will eventually form a knowledge base. There are a number of well known problems in elicitation (Gaines, 1987), caused by the tenuous nature of knowledge, the difficulties people have in articulating it, and the element of irrelevant or misleading statements. Like knowledge engineering, requirements engineering involves the extraction and representation of information through some form of interaction with the domain experts, in this case the clients. Both have the same set of techniques available for extracting the information, including various types of interview, observing people in action (and subsequent debriefing), tutoring, and case analysis. Most of the information gathered in this way is needed throughout the lifecycle of the software.

Finkelstein & Finkelstein (1983) describe the processes involved in requirements formulation. There are three basic systematic methods: the use of check lists; lateral (or divergent) idea generation; and formal specification languages. Decomposition and abstraction are important parts of the process, removing barriers to innovation. The primitive concepts used in design come from a number of sources, including: existing designs, analogy, convergent deduction, and divergent thinking[2].

Specification construction can be seen as an evolutionary process. The incremental steps either add more detail, or clarify existing parts by introducing exceptional case behaviour or retracting incorrect or over-simplified statements (Feather, 1987). We have also noted that specifications are designed artefacts, and hence the specification process involves design. Dubois & Hagelstein (1987) point out that requirements engineering differs from typical software design tasks in that the latter involve artefacts such as programs, and are done exclusively by

[2] See also Carter *et. al.* [1984] for a particularly graphic and entertaining account of requirements formulation

specialists, while requirements engineering involves real world concepts, and requires extensive communication with non-experts.

3.2. Nature of the Knowledge Sources

There are many people involved with the development of a large software system, both in its design and use. Whilst these people can be broadly classed into groups, such as users (often of several types), management and software developers, these groupings will not enforce conformity of the individual members. Each person is a potential source of knowledge, and might contribute a unique insight. Additionally, people are not the only sources of knowledge: valuable information may be found in manuals, memos, and other documents.

Much of the knowledge is difficult to elicit because the subjects need to untangle it, and the analyst needs to know how to organise it. For example, it is hard to disentangle the requirements from the goals and perspectives (and associated opinions and biases) of the people involved. It is usual for example to restrict the specification to what is desired rather than how it is to be achieved, to distinguish between functional and non-functional requirements, and to distinguish the system model from its environment. People who are not conversant with analysis techniques will be unable to make such distinctions, and so cannot disentangle the various contributions they are making. They will certainly be unlikely to offer information in a form that neatly fits with the organisational structure the analyst is developing.

Each user has a specific idea about what they want from the resulting system and how it will help (or hinder) them, and so are most concerned to influence its design. These individual goals colour their perspective of the requirements, and cause them to introduce systematic bias into their responses. Furthermore, organisations have goals and policies which may inhibit their members, again biasing their responses. This contrasts with knowledge elicitation for expert systems, where the expert is unlikely to have any particular requirements concerning the resulting system, and so can provide more objective responses. Unfortunately, knowledge acquisition research has shown that even disinterested experts are not free from bias (Meyer & Booker, 1989), and it is likely that most of these forms of bias also crop up in requirements elicitation.

The analysts themselves are also sources of knowledge, providing a great deal of experience about requirements for similar systems, and knowledge about the likely effects of particular decisions. There is inevitably a temptation for the analyst to impose his or her own preferences in the specification, and studies have shown that convincing the clients that a chosen solution is suitable is the largest part of the analyst’s job (Fickas, Collins, & Olivier, 1987). Even where the analyst has experience of the domain in question, that knowledge will be of a subtly different flavour to that of the client, and there will be differences in the detailed requirements of the particular situation. The analyst must be careful, therefore, to blend his or her knowledge with others’ contributions.

It is often difficult to compare the knowledge offered by different subjects, because their experience varies, both in type and level. Kolodner (1984) points out that experience changes the way a person reasons, and that the key difference between experts and novices lies not so much in the level of knowledge, but the ability to apply knowledge more effectively. Dreyfus & Dreyfus (1986) describe the process of evolution from novice to expert, and suggest that novices use simple heuristic rules, while experts internalise their knowledge. This implies that domain experts are less able to explain their behaviour than novices. Also, people have a tendency to tailor their explanations to the hearer, so for example a layman would receive a superficial answer, while an expert would receive a detailed response (Compton & Jansen, 1989). All these factors combine to make it very difficult to compare knowledge elicited from people with different levels of experience.

Another reason knowledge is difficult to compare is that subjects use different assumptions, and these assumptions are not always obvious. While some sources will articulate what others have tacitly assumed, it is not always clear when this has happened, nor is it always obvious when people have made assumptions that others would not agree with. As there are a large number of assumptions accompanying any communication act, ranging from assumptions that the hearer understands the language of discourse, to very specific assumptions about the domain, it is impossible to identify all of them.

3.3. Nature of the Knowledge Involved

Many areas of knowledge need to be taken into account. In a large-scale project, the information required includes knowledge about software engineering methodologies and the target machine's architecture, as well as application domain knowledge. Furthermore, for maintenance purposes, knowledge about the target software and the design history are also needed. Much of this knowledge is specific to a particular project; for example, knowledge about the needs which gave rise to the current study, knowledge about typical user behaviour, knowledge about performance requirements of the various components of the system, and knowledge about the software development process (its goals, its progress, its restrictions, etc.). In his excellent survey of the use of AI in software engineering, Barstow (1987) describes the roles played by these types of knowledge and argues that AI techniques could greatly assist with the handling of this knowledge.

The analyst will, as a matter of course, use many different notations to capture the wide variety of information (Burton & Shadbolt, 1987), as no single notation has sufficient expressive power for the many different types of knowledge (Sloman, 1985). Even natural language is frequently supplemented with diagrams and other devices. Where a single specification language is to be used, the analysts will still have to handle other notations during elicitation, because people attempt to explain their contributions in idiosyncratic ways. Indexing and cross-referencing multiple notations is a difficult knowledge management task in itself, for which extensive support is desirable.

As well as facts about the domain, relevant policies and preferences need to be considered, and these are difficult to represent. They cannot be represented as specific facts, nor as verifiable goals. Rather they are heuristic-like guidelines that restrict and channel the design process (Anderson & Fickas, 1989). The need to choose between the diversity of methodologies available for software engineering strengthens the role of institutional policy and individual preference in the decision process.

Finally, the sheer volume of knowledge compounds these problems, and makes the management of the knowledge difficult. Whilst consistency checking can be handled in a small knowledge base, as the size increases such checking rapidly succumbs to the combinatorial explosion. When a huge amount of information is involved, consistency checking is prohibitively difficult. In requirements elicitation, inconsistencies occur frequently, usually indicating a conflict between the interested parties. In such cases, the conflict represents the need for an explicit decision by the analyst, which should not be taken until all the appropriate information has been gathered. since as far as possible, all alternatives must be captured and accommodated, the project timetable must allow the analyst to delay these decisions as far as is necessary.

There is currently a paucity of computer support available to the analyst to manage this knowledge. This requires meta knowledge: knowledge about how to manage knowledge. Most existing software tools are geared to producing manual notations to represent the information. This means that most of the meta knowledge needed is held in the software practitioners' heads: only a small proportion is explicitly stored in the documentation. This increases problems of communication across the software team. If the analyst never formally records such meta knowledge, it can only be passed on by word of mouth to others involved with the software life-cycle.

3.4. Negotiation

Because of the diversity of sources and types of knowledge which input to the requirements specification, there will be many differences of opinion. Requirements engineering can be seen as resolution of the expression of various constraints and goals of the people involved, and integration of these into a single consistent specification. Conventional analysis techniques do not address the resolution process directly, and so it usually begins in the analyst’s head. Part of the analyst’s skill involves the juggling of competing requirements and negotiating with the client.

Each participant brings a number of preconceptions or biases into the specification process, which are adapted as the process proceeds. One of the reasons for this adaptability is that clients are not always sure of what they want or what is feasible. The analyst’s expertise can be used to guide them into understanding their needs better, as part of the analysis process.

However, too often the analyst takes this as an opportunity to impose his or her own solution. This arises from the view of analysis as a process of translating user intent into formal or semi-formal documents. The usual approach is to learn about the client, analyse the data, and then make suggestions. Presenting these suggestions to the client takes a significant amount of time, as the client needs to be convinced that the analyst’s interpretation is right (Fickas, et al., 1987). As analysts naturally have preconceptions of the requirements, these will be used as a basis of their understanding of the problem. Information gathered during elicitation is used to construct a description which reflects these preconceptions. Unfortunately, there is a danger that this will commit the specification to one particular understanding of the requirements, and information which conflicts with the current description will be discarded.

We prefer to treat the requirements process as a form of negotiation amongst clients and analysts, where the analyst and clients share their knowledge, and enter into a mutual process of making suggestions, and critiquing each other's suggestions. Unfortunately little computer support is available for synthesising a solution from the various inputs, nor for resolving the conflict inherent in the process.

If negotiation is used as a basis for requirements engineering, there will be points at which explicit resolution of conflict becomes necessary. Such conflicts arise for a number of reasons, varying from misunderstandings to differences of values. In many cases it is not obvious what the cause of the conflict is, nor how to resolve it. Our view of requirements engineering as negotiation emphasises the role of conflict, as a catalyst for discussion and innovation. In section 4 we examine the process of conflict resolution in more detail.

3.5. Uncertainty

One of the main aims of software engineering has been to formalise as much of the development process as possible, in order to reduce the arbitrary nature of software development, and to introduce automation. This formalisation facilitates verification, which has been advocated as a means of improving reliability. However, a formal verification can only be used to demonstrate that an implementation fulfils the formal requirements specification; in other words that no errors creep in during the design process.

Because of the informal nature of the business environment in which the eventual system must operate, there will always be a degree of uncertainty in the requirements (Balzer, Goldman, & Wile, 1978), (Lehman, 1990). In particular, the initial step in requirements engineering is an informal statement of need, and so there can never be a guarantee that a formal specification describes exactly what is required.

One symptom of this uncertainty is that the specification gets altered once implementation is underway. The fact that the process of designing a specification cannot be fully separated from the implementation has already been noted. Swartout & Balzer (Swartout & Balzer, 1982) identify two main reasons for alterations to the specification once the implementation is underway. The first of these involves physical limitations arising from implementation decisions. The second reason is the most interesting, and is put down to lack of foresight in the specification. The implementation may yield new insights into the requirements of a task, and as such the entire software engineering process can be seen as one of prototype refinement (Giddings, 1984).

Another barrier to formalisation of requirements engineering is the need for negotiation. Studies of conflict resolution show that the most successful methods require a degree of creative input (Fisher & Ury, 1981). Formal approaches to conflict resolution have a tendency to produce compromise solutions which do not properly satisfy any participant's needs (Luce & Raiffa, 1957).

The inherent uncertainty and the need for creativity means that requirements engineering can never be fully formalised. However, there is plenty of scope for prescriptive methods and tools to support the process. Cunningham *et. al.* (1985) identify a number of dangers that proposed specification models have suffered from. These include: lack of a method; difficulty of grafting methods onto existing procedures; stultification of creativity; and univocality, as few methods support elicitation from many sources or their consolidation into a consistent specification. Formal methods which do address these problems could provide a powerful framework for requirements engineering.

4. Conflict Resolution

Conflict can be thought of as interference in one party's activities, needs or goals caused by the activity of another party (Easterbrook, Beck, Goodlet, Plowman, Sharples, & Wood, 1993). In software engineering, the specification encapsulates the needs of the participants as a set of requirements. If two parties have opposing requirements, then any attempt to represent these requirements in the specification will give rise to conflict, as each would exclude the other. On the other hand, if one set of requirements is ignored completely, then a potential conflict has been suppressed. Depending on the actions of the 'injured' party, the conflict may resurface later.

The same principle holds for descriptions of the world ("domain descriptions"). In the case of requirements elicitation, part of the information elicited is a description of the system as it is at present. This includes both activities that may eventually be subsumed by the system, and the environment with which the eventual system must interact. These descriptions are rarely objective; opposing views which might not be expressed directly will manifest themselves as differences in these descriptions. Hence, conflicts will frequently be expressed as discrepancies between the viewpoint descriptions.

4.1. Sources of Conflict

It has been demonstrated that conflict, as defined above, is common in group interactions (Robbins, 1989). We can therefore assume that any application domain involving more than one person will be subject to typical group conflicts. While it might be argued that a design process with a single goal, perceived in the same way by all participants, might be free of conflict, few real design processes are of this nature. This immediately suggests two possible sources of conflict in a real-world design process: conflict between the various participants' perceptions of the domain, and conflict between the many goals of a design.

The extent of conflict in software engineering has recently been revealed by a major field study of software projects (Curtis, et al., 1988). Focusing on the behavioural aspects of software design, this study identified three major problem areas: the thin spread of application domain knowledge; fluctuating and conflicting requirements; and breakdowns in communication and co-ordination. Each of these problem areas is a source of conflict, and each depends crucially on communication between participants as a basis for any solution. A good conflict resolution approach necessarily emphasises communication between parties.

Conflicting and fluctuating requirements have many causes, from change in the organisational setting and business milieu, to the fact that the software will be used by different people with different goals and different needs. Handling constant change in requirements (which has been termed *requirements maintenance* (Finkelstein, Goedicke, Kramer, & Niskier, 1989)) requires an evolutionary approach that must be based on accurate capture of rationales and process information.

Unless the application domain with which the software deals is free of conflict, then the resulting software must incorporate this conflict. For small programs, the domain can be restricted until the conflict is excluded. For any large scale software, this is not practical. When the application knowledge is spread over many people, there is likely to be much disagreement between them, and fitting together the many contributions will inevitably lead to inconsistency.

Even if a domain appears to be free of conflict, quite often there will be areas in which there are different ways of looking at things. While such perspectives may not be fundamentally incompatible, they are likely to appear inconsistent, and so lead to conflict. Even if participants are describing essentially the same concepts, the style in which these are described may vary: even formal notation schemes allow enough variation in style so that there may be many different ways of saying the same thing.

Other sources of conflict include: conflicts between suggested solution components; conflicts between stated constraints; conflicts between perceived needs; conflicts in resource usage; and discrepancies between evaluations of priority.

4.2. Consequences of Suppressing Conflict

Existing software process models generally ignore conflict. This can lead to a number of problems. Where conflicts do occur, they are likely to get suppressed, because there is no means of expressing them within the framework. It is possible that these conflicts will remain suppressed, leading to dissatisfaction with the specification and the process that led to it. Often, a single perspective will be adopted as the basis for the specification at the cost of any alternative perspectives. In the process, useful ideas associated with the rejected perspectives will be discarded, along with the goodwill of their originators.

If these conflicts are eventually resolved, the resolution must be carried out outside the framework of the method and consequently is likely to be carried out at an inappropriate time, using an undesirable means. In addition, resolution thus achieved is untraceable, making rationales invalid, and the process irreproducible. Suppression of conflict will have serious effects on the remainder of the software development process. In the worst case, suppressed conflicts may lead to the breakdown of the requirements process, or the withdrawal of participants. Failure to recognise conflict between the perspectives of the participants will cause confusion during the requirements phase, which will then continue throughout the lifecycle. The participants’ understanding of the specification will differ, leading to further misunderstandings during design and implementation.

Research into group behaviour indicates that conflict can produce higher quality solutions (Brown, 1988). Certainly, exploration of the areas where participants descriptions differ can lead to a much better understanding of the domain (Easterbrook, 1991b). This is a strong argument for conflict to be carefully managed in the software process, with participants encouraged to express divergent views. This will ensure that the resulting system does not reflect just one point of view, and does not ignore concerns which interfere with the dominant concern.

In software design, effective collaboration is essential. It is vital that there be no losers from any conflict in the specification process, as the commitment of all participants must be maintained. Hence, encouragement of conflict must be matched with resolution methods which strive to satisfy all parties. An integrative approach should be adopted, to ensure that when divergent views arise they are incorporated into the process. The ultimate goal of the requirements process should be to produce a specification which represents all concerns.

4.3. Role of Communication

The need to maintain collaboration implies that any model for conflict resolution in requirements engineering must be based on collaborative modes of interaction. The two key collaborative methods for conflict resolution are (integrative) negotiation and education (Deutsch, 1973). Both of these emphasise communication between participants, and both greatly ease conflicts based on communication problems.

Communication between participants has an important role in conflict resolution. As Robbins (Robbins, 1974) notes, increased communication leads to decreased conflict up to a certain level, but that too much communication can lead to increased conflict. A possible explanation is that a certain amount of communication allows participants to discover commonalities, iron out perceived conflicts, and correct misunderstandings. However, a high level of communication highlights the details on which participants do disagree. Such conflicts are likely to be well-founded, and should not be discouraged. However, arguing over trivial details can be counter-productive, and so a balance must be struck between encouraging communication and devoting appropriate amounts of effort to resolution of particular differences.

Comparison of descriptions derived from different sources forms an important part of the process of eliminating errors. This can be facilitated by separating the elicitation of knowledge descriptions from the comparison activities. In the comparison process, participants may then compare their own descriptions with those elicited from others. By discussing areas of divergence, underlying assumptions are revealed, as are any omissions in the descriptions.

5. Framework

We have discussed the importance of requirements engineering, and the particular problems relating to the process. Clearly, a model of requirements engineering is needed, which allows the analyst to overcome the difficulties. In this section we present a set of objectives for such a model, and for tools to support it. Note that we are not proposing automation. Case studies of analysts at work (e.g. Fickas et al. (1987), Adelson & Soloway (1985)) have revealed that a broad range of skills are employed by analysts. It is unlikely that the full range of these skills can be automated.

5.1. Framework

Rather than a rigid formal process, the analyst needs a framework which can guide his or her expertise. This framework must support the creative input and interpretative skills of the analyst. Finkelstein & Fuks (1989) suggest that such a framework should be: flexible; empirical (in that the model maps onto the results of observational studies); enactable; co-operative; and that it should be able to handle conflict.

However, in order to facilitate management of the process and provide a support environment, a degree of structuring must be introduced. An incremental, evolutionary approach is needed. The individual steps which build the descriptions arise out of the dialogues between participants, and a model that is overly-prescriptive will severely limit the scope of these dialogues, possibly causing vital steps to be missed. Therefore, any automated support or formalisation must account for, and indeed encourage dialogue as an exploration of the current state of the specification. A result is that specification comes to be seen as a conversational activity.

In order to allow the participants to control the process, the model must allow any order of discussion. In other words, it cannot be guaranteed that needed information will be provided immediately. Rich et al. (1987) discuss the inevitable informality of human communication, listing abbreviation, ambiguity, poor ordering, incompleteness, and contradiction as key features. These features represent an essential part of the human thought process, as a means of dealing with complexity. People present ideas in the order they occur, not in an order which is convenient to the hearer. In particular, the human mind is adept at ignoring inconvenient consequences of particular statements with the intention of clarifying them later. Studies of designers have shown that they frequently make notes to themselves to return later to a particular item (Littman, 1987)

Finally, the model must allow the participants to delay the resolution of conflicts and the making of decisions. Requirements engineering is primarily an exploratory process, involving the gathering and formulation of knowledge. It is vital therefore, that it does not become overly-restricted by premature decisions. A framework for the process should encourage participants to gather all the relevant knowledge and explore all the issues before making a decision (Easterbrook, 1991b).

5.2. Support Environment

In addition to a model for the specification process, we need to consider what kind of support is needed. Automated tools should form an environment in which the knowledge collected can be organised, manipulated and interrogated. We have characterised the specification as a knowledge base, which implies that techniques from knowledge-based systems research can be applied (Barstow, 1987). If all the knowledge is collected into an on-line knowledge base, it can remain accessible for the remainder of the lifecycle (Harrison, 1987). This includes not just the knowledge about the domain, but the documentation of the process itself.

We therefore envisage an environment which comprises a knowledge base containing all the gathered information, an inference engine which defines the operations which might be carried out on the knowledge base, and a set of tools which assist in the formulation, refinement and presentation of the knowledge. This environment is based on the typical architecture of a knowledge based system (Boose, 1986), and has been applied to requirements engineering elsewhere (e.g. Reubenstein & Waters (1989)). The form of the inference engine will depend on the notation(s) used within the knowledge base. As the knowledge base represents at any point the current state of the specification, reasoning within the knowledge base allows participants to test the specification.

The knowledge base will be continually added to, and hence any reasoning is non-monotonic in that new knowledge may invalidate previous conclusions. The incremental refinement of descriptions inevitably involves adding details such as exceptional case behaviour, to fix problems which occur when descriptions are tested. Detailed tracing and recording of dependencies throughout the knowledge base is therefore desirable.

As a specification evolves, it will frequently become inconsistent, and at all times will be (to some degree) incomplete (Yue, 1987). At times there will be temporary inconsistencies, over-generalisations, and over-simplifications. However, participants will need to manipulate the specification as it evolves, as part of the exploratory process, and so the reasoning mechanisms must cope with inconsistency and incompleteness. Areas of conflict, and places where more details are needed can often be detected automatically, but the need to allow commitments to be delayed means that participants might choose not to resolve these immediately.

Finally, the entire process should be documented automatically. We have stressed the importance of capturing the design history, together with rationales. However, to do so requires a lot of extra effort from the participants. They are unlikely to be persuaded to make this effort unless a great deal of the recording process is automatic. If the series of actions made using automatic tools is recorded, this can form a framework to which rationales can be attached.

5.3. Tools

The last section described the general nature of the environment needed to support requirements engineering. There are a number of areas in which tool support can be of particular help.

One major task in which tool support can help is in guiding participants towards those areas which need more discussion. We noted above that automatic detection of conflicts and identification of missing information should be possible. Given enough background knowledge about the domain, it should also be possible to provide a degree of knowledge based critiquing (Fickas & Nagarajan, 1988), to supplement the manual critiquing process. Maarek & Berry (1989) note that automating the detailed clerical work of checking specifications is an ideal way to supplement the human activity.

The critiquing process will lead to more knowledge being gathered. Tools can assist with the incremental integration of this new information within the existing knowledge base. The translation of natural language utterances into the appropriate notation is unlikely to be automated, but again, clerical assistance can be given. Feedback can be given regarding the effect of the new knowledge, for example by tracing the effects of new cases. Where the new information was prompted by problems in the existing knowledge base, the system can keep track of which parts have been resolved, how they were resolved, and what problems still remain.

One advantage of automatically tracing the process is that the context of statements can be more readily accessed. Contextual information provides important clues for interpretation and again for validation. If the dialogues are recorded and held as a part of the knowledge base, then tools to access these transcripts can be provided.

Conflict is an important part of the specification process, and tools to help identify and resolve conflicts are needed. Whilst resolving conflict is essentially a human activity, a range of options needs to be created and explored. Assistance with developing and reasoning with the options can be provided. The system should also ensure that all relevant views are represented in the resolution process.

Finally, tools are needed for presenting the knowledge back to the participants. This includes assistance with building initial descriptions from the participant’s comments, and assistance with demonstrating to the participants the current state of the specification. Several techniques are useful for this, including animation of the specification (Kramer, Ng, Potts, & Whitehead, 1987) and summarization (Fickas, 1987).

6. Summary

This chapter has examined the importance and the difficulties of requirements engineering, and the difficulties, concluding that a model is needed to support the process. Requirements engineering is important because it is concerned with the production of specifications, which play a pivotal role in a software engineering project. The specification acts as a communication medium amongst the software team, and as a yardstick by which results of the later stages of development will be judged. The specification should contain all the information about the requirements needed during the remainder of the software lifecycle. It must be unambiguous, testable, and modifiable. It must be representative of the many people whose needs it refers to. Above all, it should be a precise description of the requirements.

Specifications need to be carefully constructed so that they are useful and usable. In particular, we have suggested that they be treated as designed artefacts, and careful consideration given to the role they must fill. The design process that creates specifications needs to be recorded for validation, to allow traceability, and this design history must include rationale. As there is an inevitable amount of uncertainty in requirements, exploratory approaches must be supported.

There are a number of difficulties in requirements engineering. A very large amount of knowledge needs to be captured, covering a range of areas. A range of sources need to be consulted to elicit this knowledge, including people with different backgrounds and different perspectives, together with reference to various texts and similar media. The knowledge might be represented in a number of different ways, and may be entangled with personal preferences and biases. Specification involves negotiation between the many participants, and conflict resolution, where there are competing needs and constraints.

All these issues point toward the need for a model of the requirements engineering process. We presented a number of objectives for such a model. The model should provide support for the interactions between analyst and client, and encourage them as explorations of the current state of the specification. It is vital that during this process the participants should be in control: the method must only guide, rather than force, the order of discussion.

Computerised support can assist this process in two main ways: documenting the information already gained, and guiding the discussions to areas which need more exploration. The support should form a knowledge management environment capable of accommodating knowledge from many conflicting sources, able to reason when inconsistencies aren't resolved immediately, and which can guide and document the process throughout.

7. Conclusion

Specification is a negotiation activity between a group of people consisting of analysts, developers, users, administrators and managers. Involving all these groups of people in all stages of the specification process will help ensure their commitment to the project, and provide a more representative specification. Furthermore, such participation will facilitate validation as the participants will already be familiar with the contents of the specification.

Currently, participation to the extent advocated here is difficult, particularly because existing techniques do not provide any means to express and resolve conflicts. Without such support, conflicts become counter-productive.

A model of requirements engineering is needed which incorporates the exploratory nature of negotiation. Support tools based upon such a model would provide an environment in which the participants can describe their perspectives, and compare them to others'. The comparison process then provides a focus for resolution of conflicts, and elimination of errors and misunderstandings, resulting in a richer, more representative specification.

Acknowledgements

This work was carried out at Imperial College, London, and forms part of the author's PhD thesis [(Easterbrook, 1991a). I am indebted to Anthony Finkelstein, and many other colleagues as Imperial for useful discussions on the ideas presented here. The work was funded by the SERC, studentship number 87311891.

References

Adelson, B., & Soloway, E. (1985). The Role of Domain Experience in Software Design. IEEE Transactions on Software Engineering, SE-11(11), 1351-1360.

Alford, M. W. (1977). A Requirements Engineering Methodology for Real-Time Processing Requirements. IEEE Transactions on Software Engineering,, SE-3(1), 60-69.

Anderson, J. S., & Fickas, S. (1989). A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. In Proceedings, Fifth IEEE International Workshop on Software Specification and Design, . Pittsburg, Penn:

Balzer, R., & Goldman, N. (1979). Principles of Good Software Specification and their Implications for Specification Languages. In N. H. Gehani & A. D. McGettrick (Eds.), Software Specification Techniques Reading, MA: Addison Wesley.

Balzer, R., Goldman, N., & Wile, D. (1978). Informality in Program Specifications. IEEE Transactions on Software Engineering,, SE-4(2), 94-102.

Barstow, D. (1987). Artificial Intelligence and Software Engineering. In Proceedings, Ninth International Conference on Software Engineering (ICSE-87), (pp. 200).

Bjorner, D. (1987). On the Use of Formal Methods in Software Development. In Proceedings, Ninth International Conference on Software Engineering (ICSE-87), .

Blum, B. I. (1985). On How We Get Invalid Systems. In Proceedings, Third IEEE International Workshop on Software Specification and Design, (pp. 20-21). London:

Boehm, B. W. (1981). Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall.

Boose, J. H. (1986). Expertise Transfer for Expert System Design. Amsterdam: Elsevier.

Borgida, A., Greenspan, S., & Mylopoulos, J. (1985). Knowledge Representations as the Basis for Requirements Specifications. IEEE Computer, April 1985, 82-90.

Brooks, F. P. (1975). The Mythical Man-Month: Essays on Software Engineering. Reading, MA: Addison-Wesley.

Brown, R. (1988). Group Processes: Dynamics within and between Groups. Oxford: Basil Blackwell Ltd.

Burton, M., & Shadbolt, N. (1987). Knowledge Engineering No. Tech Report No 87-2-1). Dept of Psychology, University of Nottingham.

Compton, P., & Jansen, R. (1989). A Philosophical Basis for Knowledge Acquisition. In Proceedings, Third European Workshop on Knowledge Acquisition for Knowledge Based Systems (EKAW-89), . Paris:

Cunningham, R. J., Finkelstein, A. C. W., Goldsack, S., Maibaum, T. S. E., & Potts, C. (1985). Formal Requirements Specification - The FOREST Project. In Proceedings, Third IEEE International Workshop on Software Specification and Design, . London:

Curtis, B., Krasner, H., & Iscoe, N. (1988). A Field Study of the Software Design Process for Large Systems. Communications of the ACM, 31(11).

Deutsch, M. (1973). The Resolution of Conflict. New Haven: Yale University Press.

Dreyfus, H. L., & Dreyfus, S. E. (1986). Mind Over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. New York: Macmillan.

Dubois, E., & Hagelstein, J. (1987). Reasoning on Formal Requirements: A Lift Control System. In Proceedings, Fourth IEEE International Workshop on Software Specification and Design, . Monterey, CA:

Easterbrook, S. M. (1991a) Elicitation of Requirements from Multiple Perspectives. PhD, Imperial College, University of London.

Easterbrook, S. M. (1991b). Resolving Conflicts Between Domain Descriptions with Computer-Supported Negotiation. Knowledge Acquisition: An International Journal, 3, 255-289.

Easterbrook, S. M., Beck, E. E., Goodlet, J. S., Plowman, L., Sharples, M., & Wood, C. C. (1993). A Survey of Empirical Studies of Conflict. In S. M. Easterbrook (Eds.), CSCW: Co-operation or Conflict? (pp. 1-68). London: Springer-Verlag.

Feather, M. S. (1987). The Evolution of Composite System Specifications. In Proceedings, Fourth IEEE International Workshop on Software Specification and Design, . Monterey, CA.:

Fickas, S. (1987). Automating the Specification Process No. Technical Report No. CIS-TR-87-05). Dept of Computer and Information Science, University of Oregon, Eugene, OR.

Fickas, S., Collins, S., & Olivier, S. (1987). Problem Acquisition in Software Analysis: A Preliminary Study No. Technical Report No CIS-TR-87-04). Dept of Computer and Information Science, University of Oregon, Eugene, OR.

Fickas, S., & Nagarajan, P. (1988). Being Suspicious: Critiquing Problem Specifications. In Proceedings, Seventh AAAI National Conference on AI, (pp. 19-24).

Finkelstein, A. C. W., Goedicke, M., Kramer, J., & Niskier, C. (1989). ViewPoint Oriented Software Development: Methods and Viewpoints in Requirements Engineering. In Proceedings, Second Meteor Workshop on Methods for Formal Specification, . Springer-Verlag LNCS.

Finkelstein, A. C. W., & H., F. (1989). Multi-Party Specification. In Proceedings, Fifth IEEE International Workshop on Software Specification and Design, (pp. 185-195).

Finkelstein, L., & Finkelstein, A. C. W. (1983). Review of Design Methodology. IEE Proceedings, 130, Pt A(4).

Fisher, R., & Ury, W. (1981). Getting to Yes: Negotiating Agreement Without Giving in. London: Hutchinson.

Gaines, B. R. (1987). An Overview of Knowledge Acquisition and Transfer. In B. R. Gaines & J. H. Boose (Eds.), Knowledge Acquisition for Knowledge Based Systems, London: Academic Press.

Giddings, R. V. (1984). Accommodating Uncertainty in Software Design. Communications of the ACM,, 27(5), 428-434.

Green, C., Luckham, D., Balzer, R., Cheatham, T., & Rich, C. (1983). Report on a Knowledge-Based Software Assistant No. Tech. Report No RADC-TR-83-195). Rome Air Development Centre.

Harrison, W. (1987). RPDE3: A Framework for Integrating Tool Fragments. IEEE Software, Nov 1987.

Kaplan, S. M. (1989). COED: Conversation-Oriented Software Environments No. University of Illinois at Urbana-Champaign.

Kolodner, J. L. (1984). Towards Understanding of the Role of Experience in the Evolution from Novice to Expert. In Coombs (Eds.), Developments in Expert Systems London: Academic Press.

Kramer, J., Ng, K., Potts, C., & Whitehead, K. (1987). Tool Support for Requirements Analysis No. Tech. Report No DoC 87/3). Dept of Computing, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2AZ.

Lehman, M. M. (1990). Uncertainty in Computer Application is Certain (Software Engineering as a Control). Communications of the ACM, May 1990, 584-586.

Littman, D. C. (1987). Modeling Human Expertise in Knowledge Engineering: Some Preliminary Observations. In B. R. Gaines & J. H. Boose (Eds.), Knowledge Acquisition for Knowledge Based Systems London: Academic Press.

Luce, D. L., & Raiffa, H. (1957). Games and Decisions: Introduction and Critical Survey. New York: J. Wiley & Sons.

Maarek, Y. S., & Berry, D. M. (1989). The use of Lexical Affinities in Requirements Extraction. In Proceedings, Fifth IEEE International Workshop on Software Specification and Design, . Pittsburg, Penn:

Meyer, M. A., & Booker, J. M. (1989). A Practical Program for Handling Bias in Knowledge Acquisition. In Proceedings, Fourth AAAI Knowledge Acquisition For Knowledge-Based Systems Workshop, . Banff:

Reubenstein, H. B. (1990) Automated Acquisition of Evolving Informal Descriptions. Ph.D. Thesis, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Reubenstein, H. B., & Waters, R. C. (1989). The Requirements Apprentice: An Initial Scenario. In Proceedings, Fifth IEEE International Workshop on Software Specification and Design, . Pittsburg, Penn:

Rich, C., Waters, R. C., & Reubenstein, H. B. (1987). Towards a Requirements Apprentice. In Proceedings, Fourth IEEE International Workshop on Software Specification and Design, . Monterey, CA:

Rittel, H. W. J., & Webber, M. M. (1973). Dilemmas in a general theory of planning. Policy Science, 4, 155-169.

Robbins, S. P. (1974). Managing Organizational Conflict: A Nontraditional Approach. New Jersey: Prentice Hall.

Robbins, S. P. (1989). Organizational Behaviour: Concepts, Controversies, and Applications (Fourth Edition ed.). New Jersey: Prentice Hall.

Sloman, A. (1985). Why We Need Many Knowledge Representation Formalisms. In M. A. Bramer (Eds.), Research and Development in Expert Systems (Proceedings, 4th Technical Conference of the BCS specialist group on Expert Systems, 1984) Cambridge, UK: Cambridge University Press.

Swartout, W., & Balzer, R. (1982). On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM, 25(7), 438-440.

Yue, K. (1987). What Does It Mean To Say That a Specification Is Complete? In Proceedings, Fourth IEEE International Workshop on Software Specification and Design, . Monterey, CA.:

Zave, P. (1982). An Operational Approach to Requirements Specification for Embedded Systems. IEEE Transactions on Software Engineering,, SE-8(3), p250-269.