ifi

Institut für Informatik der Universität Zürich

Requirements Engineering

A Survey of Methods and Tools

Hubert F. Hofmann

Nr. 93.05 **März 93**

Author's address:

Mag. Hubert F. Hofmann
Institute for Informatics
University of Zurich
Winterthurerstr. 190
CH-8057 Zurich

hofmann@ifi.unizh.ch

Table of Contents

Part III
Existing Methods

Preface

> The hardest single part of building a software system is deciding what to build. [...] No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later. [Brooks 1987, p.17]

This report is primarily aimed at people with some background in software engineering, although it should be valuable to anyone who wishes to understand requirements engineering.

We characterise a representative set of requirements engineering methods by outlining their goals, assumptions, processes, and tools. The characterisation offers a starting point to select and customise a requirements engineering method for a software project. Customisation is necessary, because a detailed description of how to do requirements engineering ("cookbook") can be appropriate in one situation, but can lead to an inappropriate software specification in another.

Thus, it is important to relate RE methods and tools to the experience and the development philosophy of the people involved in the software project. This promotes the understanding of the applied methods and helps to avoid adherence to the "one-best" method constantly applied without any reflection.

This report consists of three parts. In part I we define the terms requirements and requirements engineering, point out the importance of requirements engineering, and discuss some issues in the practical application of requirements engineering.

Part I lays the basis for part II in which we state those aspects of requirements engineering that we find essential. We identify fundamental principles of requirements engineering, show their relations, and outline essential ingredients, activities and techniques.

In part III we survey, from four different points of view, a selection of existing methods for requirements engineering. The majority of methods originates from software engineering, while IBIS, SQD, and SSM come from organisational science.

Acknowledgements

I would like to thank Norbert E. Fuchs for inspiring discussions, for critical comments and constructive suggestions on this report, and for his support of this work in general.

I would also like to thank Peter Baumann and Franz Lehner for discussing this manuscript and suggesting numerous improvements. Further thanks for suggestions and criticism go to my colleagues Thomas Abdallah, Brigitta Bailer, Thomas Barothy, Ralph Holbein, and Patrick Scheidegger.

This research is partially supported by the Swiss National Science Foundation under Contract 21-32746.91.

Part I

Definitions and Properties of Requirements Engineering

This chapter begins with some background information on software engineering and outlines the activities involved in software development. Then we set the stage for understanding requirements engineering by answering the question “What do you mean by requirements and requirements engineering?” To quote Humpty Dumpty, I can make a word mean whatever I want, but to communicate with you I'd better make sure you know what I want it to mean. The final part of this chapter considers the importance of requirements engineering and reflects on its practical application.

1 Software Engineering

The goal of software development is to create software systems that meet the needs of the clients, i.e. customers and users. A customer is an individual or an organisation who wants a software system to be built and bears its costs. The term user denotes an individual or an organisation who works with the software system.

Software development has to assure both the quality of the software system and the quality of its development process. A quality development process supports software evolution by defining its activities and their dependencies and by documenting their rationale. As software systems undergo change, a quality development process is essential to maintain and enhance the quality of the software system. The field of research dealing with methods, techniques, and tools for developing quality software is called software engineering [Naur & Randell 1969].

The engineering approach to software development — that replaced software development ”as an art“ — led to the view that software development resembles other engineering processes. Thus, models for software development were and are still derived from other engineering disciplines.

Figure 1 provides a simplified version of a development process model that consists of several well-defined phases and takes into account the possibility that phases recur to elaborate or revise client needs and requirements. Each phase results in a document that forms the starting point for the next phase. This process model, or variants of it, is often called the software life-cycle.

The phase *System and Software Requirements* establishes the goals of the project, investigates the client needs and the application domain, and specifies requirements.

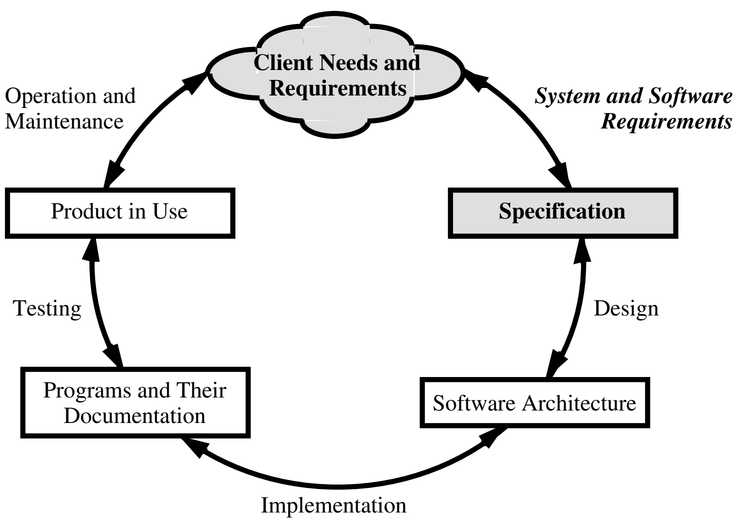

Figure 1. Software Development Process Model

The client needs are the “wish-list” of the client for the system to be developed. This phase concludes with a specification of the software system. The activities and documents of this phase are the subject of this report.

Design evolves from the software specification. It determines the architecture of the system, i.e. its built up from components and sub-components. Design results also in the specification of the underlying logical data model, in communication mechanisms of system components, and in the specification of algorithms and data structures.

Implementation transforms the software architecture into a set of system components, which are efficiently executable on the selected hardware. In other words, the implementation phase produces an operational software system based on concepts defined during previous phases.

Testing. The goal of testing is to assure that the implementation meets its requirements. It includes quality assurance procedures, walkthroughs and inspections, static analysis, symbolic execution, formal verification, unit testing, and system and acceptance testing.

Operation and Maintenance. Once the software system has been accepted by the client, installed and put into practical use, it enters the maintenance phase. Errors that were not discovered in earlier phases are corrected, and the system is enhanced as new requirements are formulated.

Early development models prescribed a strictly sequential process with clearly separated phases. However, software development is complex and dynamic, depending on the application domain and the qualification of the project team. This dependency cannot readily be described using a simple model. Thus, experience with a strictly sequential development process revealed its deficiency [Brooks 1975; McCracken & Jackson 1982].

Detailed software process models are still the subject of research. Like many researchers, we believe that exploratory and operational approaches to software development are most promising to build and to maintain quality software systems.

Exploratory software development advanced from the development of artificial intelligence applications (e.g. knowledge systems) where it is often difficult to establish detailed system and software requirements [Mostow 1985]. In this process model one implements software to meet initial requirements and exposes it to the users' working environment. Due to additional insight and experience with the system, more adequate requirements are formulated. Reformulation of requirements and modifications of software lead to iterative development until an acceptable system is built.

Operational software development is an evolutionary approach, where software gradually evolves from its specification [Zave 1984; Fuchs 1993]. This process model entails the formulation of executable or operational specifications, which are exercised, validated, and revised, to fulfil the requirements. An operational specification is a formal model of the software system and can (partially) generate its behaviour. Preserving external behaviour, specifications are transformed into formalisms used in the working environment to yield the intended software system.

2 Requirements

The field of software engineering constantly evolves and many terms do not have unique definitions. Of the many definitions of requirement, we present the definition of the IEEE that is widely used and closely reflects our understanding of the term.

The IEEE software engineering glossary [IEEE Std.610.12-1990] defines requirement as: *(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2).*

Requirements can be divided into functional and non-functional requirements. However, the boundaries between these two are less precise than one would wish.

Functional requirements define the functions that a system or system component must perform. They describe transformations the system or one of its components performs on inputs to produce outputs. For example, the requirements for a money dispenser have to answer questions like: What input is necessary to withdraw money? What have I to do to get a receipt?

Non-functional requirements are constraints, e.g. performance (time and space constraints, accuracy), user interfaces, reliability (e.g. robustness, availability of physical equipment), maintainability, security, portability, standards, economical and political aspects. Example non-functional requirements are: Under what conditions can we change the personal code of our card? How long should the money dispenser wait for user input?

Non-functional requirements are not just an “add-on” to functional requirements. Both are equally important to specify the software system. Moreover, non-functional requirements are likely, in due course, to become functional ones for some part of the system. For example, a requirement for reliability can be translated into some function for error reporting.

3 Requirements Engineering

The process of acquiring, refining and checking client needs for a software system is called requirements engineering [IEEE Std.830-1984]. The goal of requirements engineering (RE) is a correct and complete specification of software requirements.

In this section, we define the term “requirements engineering”. Instead of adding a new definition to the numerous existing ones, we discuss three definitions of Boehm, the STARTS guide and Leite.

Boehm [Boehm 1979] defines RE *as the discipline for developing a complete, consistent, unambiguous specification — which can serve as a basis for common agreement among all parties concerned — describing what the software product will do (but not how it will do it; this is to be done in the design specification).*

In the STARTS guide [STARTS Guide 1987] RE comprises those *processes by which the purchaser's*[1] *statements of intention and requirement, written or spoken, are transformed into a precise, unambiguous, consistent and complete specification of system behaviour including functions, interfaces, performance and constraints.*

For Leite [Leite 1987] requirements engineering is defined as *a process in which “what is to be done” is elicited and modeled. This process has to deal with different viewpoints, and it uses a combination of methods, tools, and actors. The product of this process is a model, from which a requirements document is produced.*

Boehm defines RE as a process that involves close collaboration between client and developer. The requirements specification is the communication media used in RE to reach an agreement about the intended software system. Boehm also emphasises the importance of avoiding design decisions when defining requirements. Thus, he draws a sharp distinction between what one is trying to achieve and how one is designing the software system to satisfy the requirements.

Though this is a desirable objective, in practice the boundary between specification and design is fuzzy. This holds both for functional and non-functional requirements. It seems to be virtually impossible to define a behaviour without at least a minimum of an idea how this behaviour should be achieved [Swartout & Balzer 1982]. Furthermore, the specification represents an understanding of the client's requirements at one point in time based on some level of experience. However, software evolves during both its development and operational use. As RE has to ensure the definition of a “realistic” software system, it seems artificial to separate RE from other concerns of software development at any cost.

[1] Purchaser means the same as client, that is, the user and customer.

The STARTS Guide and Leite focus on RE as the systematic process of investigating client needs for the software system. The investigation is based on understanding the perceived problem for which the software system is proposed.

A major difficulty in problem understanding is that experts and novices accomplish their problems differently cf. [Dreyfus & Dreyfus 1986]. In RE we have the special situation that the client is an expert in the application domain, and the requirements analyst is an expert in software engineering, and each may be a novice in the domain of the other.

4 Importance of Requirements Engineering

Many studies have shown that the later wrong development decisions are detected, the more expensive they are to repair cf. [Boehm 1989]. This is especially true for decisions taken during RE, which is the very beginning of software development.

RE activities provide in many ways the foundation for a software system. RE forces clients to consider their requirements carefully and to review them within the context of the problem. This process aims at a correct and complete specification of the software system.

RE records and refines requirements, improves the transparency of the software system, and enhances communication among clients and developers. Unless the requirements specification represents an informed consensus of clients and developers, the project is not likely to succeed.

RE enables test plan development used for validation and contractual compliance. The requirements specification is the standard against which design and implementation are checked for correctness and completeness.

RE supports project management by providing a starting point for subsequent control and management activities, e.g. estimates of cost, time and resources needed. Moreover, a requirements specification can be used to define the constraints for future changes and to support maintenance.

If RE activities are insufficient, users will lose confidence in the development team and in RE. This loss of trust will affect not only the current software project, but also the clients' long-term attitude toward software development and personnel [Zuboff 1989].

5 Issues in Requirements Engineering

Issues in RE concern business objectives and working environment, clients and their views, communication barriers and tacit knowledge, system evolution and integration, and documenting requirements. This list is not complete, but it enumerates the most important issues.

5.1 Business Objectives and Working Environment

Although business objectives are often stated in very general terms, they are used to decompose work into tasks. In such a situation RE usually focuses on task descriptions and on analysing similar software systems [Greenspan 1991]. This information provide the basis for specifying the software system to be built. However, the concentration on (abstract) tasks often leads to software systems that do not fit into the current or the planned working environment.

The software system will change the working environment, but it is hard to anticipate actual effects on the organisation. Change does not only occur when a software system is introduced and put into practical use, but also when the environment changes (new insights into problems, new machines being installed, etc.). The need for change is exemplified by the enormous costs of software maintenance. Although there are many reasons for the difficulty of maintaining software, lack of thorough attention to RE is a major one.

One reason for inadequate software systems is that the initial specification often is also the final one, thus hindering the communication and learning process of the people involved, which might lead to more suitable software systems.

5.2 Clients and Their Views

Large software systems usually have a diverse client community. Different clients have different priorities and different requirements have different degrees of importance for them. Also customers and users are rarely the same people. Customers may impose requirements because of organisational and budgetary constraints, which are likely to conflict with user requirements. Taking different and potentially conflicting views into account, avoids a narrow frame of reference and helps to build an appropriate picture of the problem.

Respecting different point of views can also have negative consequences. Multiple views can cause redundant data, inconsistency, and ambiguity [Finkelstein et al. 1992].

Also the size and complexity of requirements can give rise to misunderstandings, can make it difficult to focus on one aspect at a time and to see relations between requirements. There are also causes problems when the specified system is impossible to visualise from the specification.

5.3 Communication Barriers and Tacit Knowledge

RE depends upon intensive interaction and communication between clients and requirements analysts. However, there are interaction problems that cannot be solved by communication alone.

People know more than they can ever tell. When this "tacit knowledge" concerns a proposed software system, we are often at a loss how to bring forth this knowledge. The problem is more than just a vagueness of the client concerning requirements, because tacit knowledge may not be describable by the client. It is an open question whether elements of our tacit knowledge are inaccessible in principle [Wittgenstein 1960], or just hard to verbalise, and remain either consciously or unconsciously suppressed [Varela et al. 1992].

Exploratory and operational approaches help to overcome communication barriers, because they offer the chance to gain experience with the proposed software system.

5.4 System Evolution and Integration

Few software systems are built form scratch. In practice many projects extend an existing software system. Thus, the requirements analysts must understand these systems, which are usually an integration of components from multiple vendors.

Therefore, the emphasis shifts from designing and implementing a system to choosing and combining existing software components. Because existing components impose design constraints, it is even more important to map between requirements and software design. Immediate mapping from requirements to software code leads to fault fixing directly in the code. This will cause the software system to diverge from the requirements specification and the requirements specification will no longer reflect the needs of the client.

5.5 Documenting Requirements

RE documents are large. They may be several hundred or thousand pages long, each page containing many detailed items, each of which can have profound effect on the rest of the software system.

People have problems to understand documents of this size, even if they read it carefully. The problem is one of conceptualisation. It is almost impossible to read a specification document of several hundred pages and be able to conceptualise the system. Once a document gets to a certain size and complexity, it is impossible for one person to keep the complete picture in his or her head. This leads to problems and errors that are not seen until after the software system has been built.

Part II

Essentials of Requirements Engineering

In this chapter we describe our view of RE essentials. We identify fundamental ingredients of requirements engineering, show their relations, and outline essential activities and techniques. Flexibility is a characteristic for RE and this chapter offers not a “cookbook” to be followed every time.

1 Ingredients

This section outlines the fundamental ingredients of RE from our point of view. These are the perception of the problem for which a software system is proposed, the RE process, and the requirements specification, which is the essential document during RE — and during the whole software development.

1.1 Perception of the Problem

RE starts when a problem has been perceived that offers some potential for a software system and when it has been decided to tackle the problem this way.

Many definitions of the term problem can be found in literature, some of which are highly formalised [Mitroff & Mason 1981]. However, we can think of a problem as the

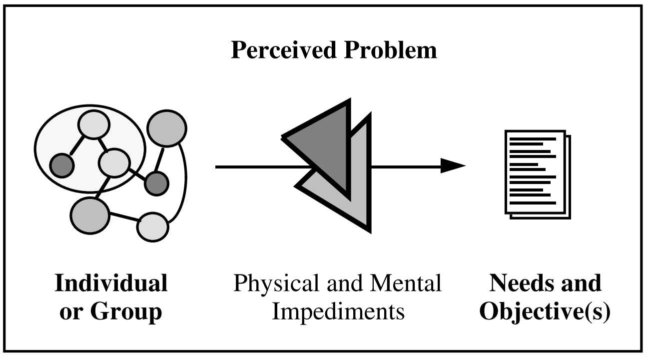

Figure 2. Perceived Problem

difference between what is and what ought to be. Problems are impediments that an individual or a group perceive when trying to achieve certain needs and goals (e.g. design software systems that meet the requirements of the client). In this endeavour an individual or group has alternative courses of action available and the choice made can have a significant effect on the problem solving process. Complete knowledge about a problem situation occurs in artificial situations only. In reality there are some doubts what the actual problem is, which solutions are available, and which one should be selected.

Problems can be described in various ways. One way considers the availability of strategies to solve perceived problems [Simon 1973]. Simon defines structured and ill-structured problems. A problem is structured if a problem solving strategy can be specified and handed over to an individual or group to solve the problem without additional help. Thus, the application of a problem solving strategy requires little knowledge of the problem to be solved. To the contrary, an ill-structured problem lacks definitive formulation and requires knowledge of the specific application domain.

Models offer another way to describe problems. A model is composed of objects and relations presented as a (formal) description of the problem. Models are problem specific and require domain knowledge to be applied appropriately.

Both the problem solving and the model-oriented aspects are of great importance in RE. Therefore, the essential difficulty is how to arrive at a set of concepts to express models of the problem situation, and how to find problem solving strategies.

1.2 Requirements Engineering Process

In this section, we briefly outline the activities and documents of the requirements engineering process and show relations between them.

The RE process has been divided in several ways cf. [Partsch 1991; Gause & Weinberg 1991; Martin 1988]. For our purposes it is convenient to partition RE into the activities: elaborating needs and objectives, requirements acquisition, requirements modeling, generating and evaluating alternatives, and validation. In Figure 3 these activities are depicted as transitions between the documents of the RE process.

The RE process begins with an understanding of the perceived problem. RE defines the boundaries of the proposed system by *elaborating needs and objectives*. The objectives for the software system are usually defined by (strategic) management, whereas the needs or the "wish-list" of the client for the system are subject to debate. A study is carried out to show the feasibility of the project and a preliminary project design is established.

The activities elaborating needs and objectives and *requirements acquisition* are interrelated. By requirements acquisition we mean that an individual or group identifies

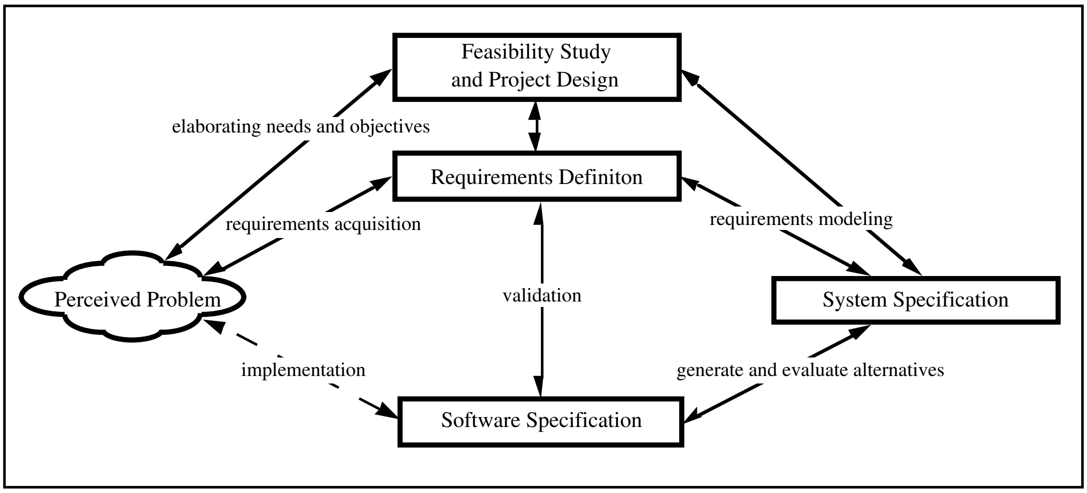

Figure 3. The Process of Requirements Engineering

or creates concepts of the problem situation. These concepts provide the building blocks for expressing the problem and specifying the software system. In other words, concepts provide the vocabulary to describe the perceived problem in detail. The documentation of this activity is called requirements definition.

The purpose of *requirements modeling* is to describe the problem situation using semi-formal and formal notations. Such a description is called system specification. It is a precise and understandable document that sets out the system services.

The system specification is used to generate alternative software specifications. Unlike the system specification, the software specification is primarily intended for designers rather than for users or management. The activity resulting in the software specification is called *generation and evaluation of alternatives*.

The requirements specification, i.e. the combination of system and software specification, must correctly and completely fulfil the client requirements. The process to ensure compliance is called *validation*. The implementation of a partial software specification influences the perception of the problem and can induce a new RE process.

1.3 Cyclic Requirements Engineering

It is difficult to specify the correct and complete software specification at the beginning of RE. The “right first time” principle will only work, if at all, in simple situations. Depending on the problem structure, the experience of the people involved, available methods and tools more than one RE cycle can be necessary to specify the software system. The RE process continues in recurrent cycles of exploring the perceived problem, proposing alternative specifications, and validating software specification (Figure 4).

A cyclic process allows the client to receive early feedback on the analyst's interpretation of requirements and to correct misunderstandings as early as possible. The RE process is pursued at different levels of detail to progressively prepare detailed specifications. Abstraction, partitioning, and projection are three techniques to support the different levels of detail in RE activities and to keep them in balance.

Abstraction. Abstraction concentrates on structural relations among general and specific entities. The use of abstraction leads to a vertical decomposition of considered entities, which allows to suppress details and to concentrate on essential properties.

Partitioning. Partitioning describes the problem in terms of its parts. It makes systems modular and allows to concentrate on one entity at a time. Partitioning leads to horizontal decomposition.

Projection. Projection allows different viewpoints. A projection presents the entire system, but with respect to only a subset of its properties.

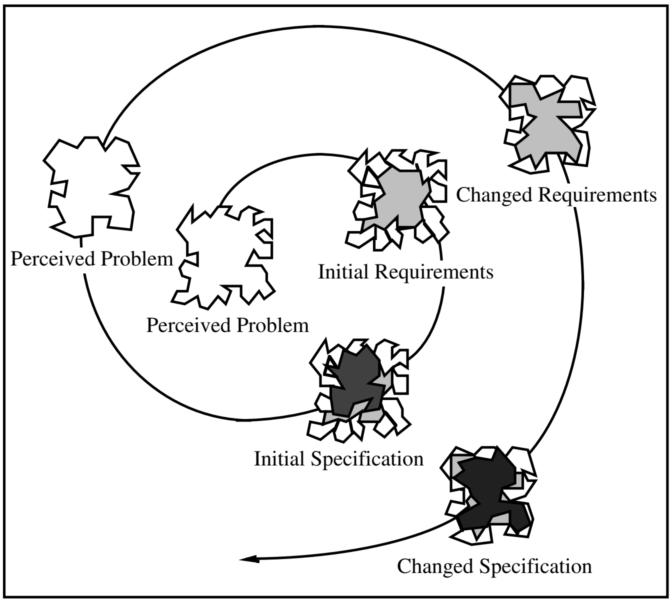

Figure 4. Cyclic Requirements Engineering

In RE one must carefully decide which problems to investigate to which degree of detail, i.e. one has to find the correct balance between problems and their solutions. Failure to find this balance can lead to insufficiently investigated problems, or to elaborate solutions of non-problems. The correct balance is project-specific, and once found it is subject to revisions[2]. Other projects and other situations may require to find a new balance.

1.4 Requirements Specification

The requirements specification is a combination of system and software specification. It is a document for which the domain concepts identified or created during requirements acquisition provide the building blocks. These concepts are often stated in an informal way in the requirements definition.

Ideally, the requirements specification is a reference containing everything necessary to develop software that is acceptable to the client. To be useful as reference throughout software development, the requirements specification must always be kept up-to-date.

1.4.1 Specification Attributes

Recent RE literature agrees on the attributes of a specification cf. [Thayer & Dorfman 1990]. Although these attributes are generally understood, there meaning must

2 “Man braucht nicht immer denselben Standpunkt zu vertreten, denn niemand kann einen hindern, klüger zu werden.” Konrad Adenauer

be defined by the software project. Moreover, the people involved in the software project have to state clearly why and how they decided on a particular combination of essential specification attributes.

Correct. A specification is correct if each specification statement corresponds to client requirements or is derived from them.

Complete. A specification is complete if it includes all functional and non-functional requirements, and conforms to any applicable standard.

Consistent. Consistent means that no specification statement is in conflict with any other in the specification. Examples for inconsistency are conflicting terms, concepts, and consequences.

Unambiguous. A specification is unambiguous if the specification statements have only one interpretation.

Understandable. Formal notations make specifications less ambiguous and more suitable for validation, but their understanding often requires special training. Paraphrasing and tools that translate the formal notation into an (almost) equivalent prose can increase understandability.

Modifiable. A specification should be written in a form that makes it easy to consider new requirements or to modify existing ones.

Annotated. The purpose of annotations is to provide guidance to the development team. Two types of annotations that are of most help are relative necessity and relative stability of each specification statement. Annotations can be also used to trace the history of specification statements.

Achieving all the preceding properties in a specification is difficult. While eliminating ambiguity makes the specification more understandable, an attempt to be complete lets the document become large and difficult to read. Thus, the only conclusion is that there is no perfect software specification.

1.4.2 Specification Structure

Since the requirements specification must serve a variety of people, it is important that it is structured so that it can be understood and analysed. There are various suggestions how to organise the requirements specification, e.g. [Sommerville 1992].

In the following, we provide an outline for the document. This outline can be easily adapted to comply with different standards.

Introduction. The introduction describes the need for the software system and places the system in context, briefly describing its functions and presenting a rationale for it.

Environment Characteristics. This chapter describes how the system fits into the business or strategic objectives of the organisation commissioning the software. It sets out the system model, showing the relations between the system and its environment.

System Evolution. This chapter describes the fundamental assumptions on which the system is based and outlines anticipated changes due to changing user needs, hardware evolution, etc.

Functional Requirements. The services provided for the client should be described in this section. The connections to the non-functional requirements should be stated explicitly.

Non-functional Requirements. The non-functional requirements concern constraints of the system (performance, user interface, reliability, maintainability, portability, etc.). They should be related to the functional requirements.

Glossary. The glossary defines the technical terms used in the document.

Sources. This chapter identifies the persons who contributed to the requirements.

Index.

2 Activities

Several activities are essential for the process of RE: discovering, understanding, and describing informally the client's requirements; constructing a conceptual model that integrates and consolidates different client views; specifying a software system to meet the requirements in a formal (executable) language; and validating that specification with respect to the requirements.

2.1 Elaborating Needs and Objectives

Before developing a software system management should have established an information system strategy. Information system strategies are understood as the entirety of measures defining the global and long-term development of the information infrastructure in the organisation. They lay out the objectives of the organisation concerning information systems.

Thus, requirements are rarely specified from scratch. The elaboration includes an analysis of business objectives and their relations to the project [Mittermeir et al. 1990]. It is necessary to understand the system currently doing the work, envision future working environments, and clarify the way in which software is used to support these activities.

Moreover, the context in which the software system is built delimits the sources from which the requirements are acquired. This activity identifies the people contributing to the software project and those who are affected by the software systems. Their needs, i.e. their "wish-list" for the software system to be built, are subject to debate and are discussed during RE. However, this does not imply that the information necessary for specifying an adequate software system are available.

Therefore, elaborating needs and objectives performs a feasibility study and project design. The feasibility is used to decide whether to proceed with the development project or to stop it when the costs for developing the system outweigh its benefits. The feasibility study also tries to discover existing solutions to the problem.

Project design assigns team members and defines how the development process will be organised. Project design during RE also offers a guideline for further development activities and for the allocation of organisational resources. However, it does not determine the software project in every detail, which would lead to a rigid and inflexible development process.

A statement of the problem context and the boundaries for the software system are a major result of the elaboration of needs and objectives. For well structured problems this information can already exist or be readily available.

2.2 Requirements Acquisition and Modeling

Preliminary acquisition focuses on existing material, e.g. the documented objectives and the "wish-list" of the client or reports about similar systems. Natural language is either the primary means of expression or, in the case of numbers and diagrams, the means of explaining primary data. Keywords are identified and associated with fragments in texts that help to describe them, much like entries in an encyclopaedia. The typical outcome of this analysis are glossaries and data dictionaries.

Clients and requirements analysts use these results when investigating the proposed software system and its connection to the environment. They attempt to impose a global structure on acquired data to create a model of the problem in terms of taxonomic hierarchies, causal networks, tables, flow diagrams, or whatever organisation is convenient for modeling the problem.

The results are documented in the form of conceptual models, which describe static and dynamic aspects of the problem, respectively the application domain. Static structures of concepts are represented by networks of objects, which encapsulate attributes [Rumbaugh et al. 1991; Dodd 1990]. Dynamic structures describe exclusive services, e.g. the interface with its environment.

The conceptual model selects features of the application domain that are of interest to the client, but ignores details and relations that can be important for another purpose. This is an important point in specifying and designing software systems. As one wants to reuse specifications, and as the perception of problems will change over time, conceptual models and their implementation must change, too.

Often semi-formal and informal representations are used as a communication device between clients and requirements analysts. They represent requirements as comments in natural language, e.g. Hypertext [Conklin 1987], or pictures, e.g. SA and data-flow diagrams [DeMarco 1979].

The activity can be made more precise, if the requirements analyst takes a formal approach and builds conceptual models using one of the formalisms that have been proposed in recent years, e.g. Conceptual Graphs [Sowa 1984], Petri-Nets [Peterson 1977], Z [Diller 1990].

Formal representations of requirements are unambiguous. They have been mostly used to model functional behaviour. Furthermore, they can often be transformed into machine-executable form.

However, each formalism has its strengths and limitations. Therefore, it is doubtful whether any single formalism can deal with all aspects of RE. Thus, we suggest to combine formal, semi-formal and informal representation of requirements. We view these representations as complementary in the process of specifying the software system.

Although this multi-lingual approach may seem burdensome, it avoids the exclusion of those who do not know the notation, and it eases communication. During acquisition and modeling requirements are typically ambiguous, incomplete, and inconsistent. Moreover, their expression includes different notations (e.g. text, graphic, formulas) and various levels of detail from concrete examples to general properties. Since software specifications are derived from these statements, the acquisition and modeling process must allow different levels of detail and style, and has to cope with ambiguity, incompleteness, and inconsistency [Fuchs 1993].

2.3 Generating and Evaluating Alternatives

The generation and evaluation of alternative specifications avoids the adherence to a “satisficing” alternative [Simon 1970] that “will do” and can be implemented with minimum effort. This can be appropriate, but should be adopted consciously than by default.

Since the generation of alternative specifications is an act of creativity, it is essential to document the rationale for each alternative, e.g. how an alternative relates to needs and objectives.

Accurate timing of evaluation activities is difficult. The sooner the assessment, the less likely one wastes resources, but the less one knows about client needs and the less reliable are the estimates. Hence, recurrent evaluations are necessary.

An important point for evaluating requirements is their priorisation. Priorisation effects RE significantly. High priority requirements are addressed first, subsequent requirements changes are defined and examined, before low priority requirements are considered. This can result in cost and time savings when changes of requirements during the software project are necessary.

The evaluation of alternatives also considers the consequences of using the software system to be developed. This avoids the selection of an alternative specification that might be inexpensive to implement, but for which the training costs of the resulting software system might be enormous.

No matter how many software specifications are assessed, evaluation is more than choosing the technically best alternative. Evaluation could also result in abandoning the project altogether.

2.4 Validation

Validation is often confused with verification. Verification means the check that specifications describe the system right. Whereas validation means the check that specifications describe the right system, i.e. specifications meet the client requirements.

Validation is a recurrent activity. Once a set of requirements is specified, it is validated to meet the client requirements. If the validation process is inadequate, misunderstandings and errors are propagated to design and implementation, and expensive modifications of the software system are required to correct them.

Usually an acceptable level of validity is achieved through testing, which is a confirmation technique ensuring that the software specification is following pre-established criteria. Testing often means inspections, reviews, walkthroughs, and static techniques cf. [Freedman & Weinberg 1990].

We believe that these techniques are not enough to reach an acceptable level of validity. Reading definitions and specifications, clients must picture the system in operation and imagine how that system fits into the working environment. This type of analysis is, at best, difficult for clients to perform. Hence, we suggest to perform validation by using the specification as an executable prototype.

Executable specifications can contribute enormously to validation, because they allow users to experiment with stable portions of the software system and thus give them the necessary touch-and-feel experience [Fuchs 1992]. So users can participate in the formulation of the specifications and in their immediate validation. The involvement of users and the immediate reflection on consequences of specifications are of utmost importance, because they contribute to the adequacy of software, to the reduction of costs, and to the adherence to the schedules.

3 Techniques

From the various techniques available for RE (creativity techniques, observation, etc.) we discuss interviews and questionnaires, the survey of documents and existing systems, the analysis of the working practice, and the experimentation with prototypes.

Although their combination is common practice, the appropriate choice of RE techniques depends on the development project. For example, when users have a clear understanding of requirements and the proposed system is similar to the software systems in use, one can usually rely on interviews and on the survey of documents and existing systems to acquire requirements. This deriving technique is most successful for standard software.

When the users are unfamiliar with the features of the proposed system or when the existing system is inadequate, one needs to rely more on analysing the working practice and on prototyping. The analysing technique is most useful when the working environment is altered to perform new or extended tasks, which are supported or solved by a software system.

Prototyping is an effective, even necessary technique, if the user interface is critical to the system's success, or when it is not possible to visualise the requirements of the system. Although this technique often means to generate new prototypes, existing prototypes of an earlier system can be very helpful in specifying the new software system.

3.1 Interviews and Questionnaires

Interviewing is widely used for acquiring information in the development of software systems [Burton et al. 1988]. It is especially helpful to get an initial understanding of the problem, respectively the application domain, and to complete the requirements.

Several techniques for structuring interviews have been suggested. Examples are the *tutorial interview*, in which experts give an introduction to the application domain, and the *focused interview*, in which the requirements analyst prepares interview topics, but not precise questions. Both techniques are appropriate for early RE sessions, because one is free to focus on aspects appearing during a session. In the *structured interview*, the requirements analyst covers a list of topics, not necessarily in a rigid order, but following a structured plan. For the efficient application of this technique the interviewer must be well-informed about the domain. The *teachback interview* is a technique in which a person or group describes a problem solving activity to the requirements analyst, who then teaches it back. *Card sorting* is another example of a structuring technique. A person is asked to group a (randomly distributed) set of cards, each labelled with a concept of the application domain, according to criteria of his or her choice. Yet another example is the use of *repertory grids* to classify concepts and objects of the application domain. Repertory grids help to judge the relation between domain concepts or objects by rating their difference.

Many interviewing techniques can be criticised for imposing their structure upon the RE process and hence the application domain. Moreover, they often require an analyst experienced both in the application domain and with development techniques. A qualification that is difficult to satisfy.

Questionnaires provide an alternative to interviews. A questionnaire is sent to the client and replies are analysed by the requirements analyst. Questionnaires are useful in gathering numerical data from the client or when one wants to get back discrete values, i.e. by using structured questions that include a range of replies and require the responder to tick one of them.

However, there are a number of disadvantages in using questionnaires. First of all, they are mostly suited to closed questions and are not suitable for open questions. Secondly, questionnaires are usually not answered completely and often express some current concern rather than long-term concerns.

3.2 Survey of Documents and Existing Systems

The protocols resulting form interviews, questionnaires or other existing documents are analysed for recurring patterns, i.e. general events and statements [Ericsson & Simon 1984]. For example, *protocol analysis* requires that a person performs the task while describing and interpreting problem solving activities (*think-aloud protocol*). The sessions are usually tape-recorded, transcribed and analysed later. There is some debate how far the protocol analysis interferes with the execution of the task and how far it is an accurate description of what the interviewed person is doing.

The *derivation from existing systems* assumes that the requirements of the software system can be derived by studying a system for the same type of organisation or for the same type of application. For example, money dispenser systems require similar inputs (card number, personal code, etc.), processing (e.g. calculating account), and outputs (money, receipt, etc.). By identifying these similarities, the existing software system can be used to create an appropriate model for the proposed system. Such a model allows to state explicitly the difference from the existing system; e.g. the money dispenser lacks output in foreign currency which is required by the proposed system. Recognising such differences helps to derive expanded requirements for the proposed system. Moreover, such differences are often crucial for the allocation of organisational resources to a particular software project: What is new about the software system you propose?

The derivation from existing systems is especially helpful if it is not possible to specify requirements in co-operation with the users. A reason to apply this strategy might be, when personnel fluctuation has eroded the base of user experience and knowledge about the application domain.

However, the examination of existing systems can only partially support the acquisition of requirements. Although this strategy can provide enough guidance to specify standard software, it is usually not enough to provide the user with software systems that are individually adequate.

3.3 Analysis of Working Practice

In the analysis of the working practice the requirements analyst has to understand the organisational objectives *and* the needs of the client. A starting point is to study inputs, processes, and outputs of the working practice and examine the relations between them.

Focusing on processes, inputs, and outputs is especially effective if the considered process is well structured. However, to focus on information processing primarily neglects that information is an interpretation of data and that the requirements analysts' interpretation thereof is not necessarily the interpretation of the client.

Moreover, this kind of analysis does not show aspects of responsibility, significance of informal communication, etc. Formal analysis emphasises details of the application domain or resource issues, but the nature of the working practice remains hidden from the requirements analyst.

Therefore, the analysis of the working practice must also diagnose existing jobs. It shows the impacts of changes in the working environment with respect to the software system to be built, cf. [Ulich 1992]. For example, the Job Diagnostic Survey [Hackman & Oldham 1975] provides measures of job dimensions (e.g. skill variety, task identity, autonomy, feedback from the job itself), individual psychological states resulting from these dimensions, and affective reactions of employees to the job and work setting (e.g.

high internal work motivation, high quality work performance, low absenteeism and turnover). The Job Diagnostic Survey proposes that positive personal and work outcomes are obtained when three critical psychological states are present for the employee, i.e. experienced meaningfulness of the work, experienced responsibility for outcomes of the work, and knowledge of the actual results of the work activities.

The analysis of the working practice focuses on information processing and on social aspects. Recently, the importance of social aspects for RE, and for software development in general, has been emphasised by several authors e.g. [Floyd et al. 1989; Greenbaum & Kyng 1991]. Greenbaum and Kyng start with the envisionment of future work situations allowing the user to experience the effect of software systems on their working practice.

3.4 Experimentation with Prototypes

Prototyping is the process of building an (operational) model that allows clients and requirements analysts to understand problems by experiencing them first hand, and to use this experience to guide the RE process [Budde et al. 1992]. Prototyping supports the stepwise development of the requirements specification and allows dynamic representation of system behaviour.

Prototyping acquires requirements through repeated iterations of prototype building and clients suggesting modifications and additional features. These iterations continue until both clients and analysts believe they have specified the requirements for the software system.

Prototyping is most commonly used to develop interactive systems with the user interface as the critical component. It enables earlier more effective dialogue between clients and requirements analysts focusing their discussion on real behaviour and on actual client experiences in exercising the prototype. The replacement of words by actions can give the client a better understanding of what is being developed. Moreover, it helps to expose unstated assumptions and triggers some of requirement changes early in software development.

Nevertheless, prototyping does not explicitly support the structuring of requirements. Systems whose requirements have been determined primarily through prototyping are often more difficult to integrate with other software systems and exhibit a less coherent design [Bischofberger & Pomberger 1992]. Therefore, prototyping must be complemented by structuring principles and careful co-ordination of the prototyping process is necessary. For example, explicit cost, effort, and time limits for experimenting with prototypes and procedures for their documentation.

Part III

Existing Methods

Many methods for RE have been developed until now. Several of them will be introduced here with a brief description of their goals, assumptions, RE process, and available tools, if any.

1 Characterisation and Evaluation

With increasing interest in RE, comparisons and critiques of available methods, techniques and tools have appeared in the literature, e.g. [Byrd et al. 1992; Davis 1988; Roman 1985; Taggart & Tharp 1977; Yadav et al. 1988]. Many of these papers propose characterisation and evaluation criteria for methods. While we agree with many of the published criteria — and will rely on them — we feel that they are not entirely appropriate.

Most important, there is often no clear distinction between the characterisation of methods, that is, describing what they provide, and their evaluation, that is, saying how well they support RE. We believe that this distinction is essential. To put forward our argument we outline four particular evaluation techniques [Olle et al. 1983; Fowler et al. 1990].

One technique to evaluate RE methods is to use a taxonomy of features for identifying a class of methods. The taxonomy is usually generated from some notion of an ideal method. The ideal or "one-best" method addresses all desirable features against which the surveyed methods are measured. In practice, however, methods differ significantly, because there is not or cannot be the "one best" method applicable to every problem.

The development of "yardsticks" tries to avoid the notion of an ideal method. Yardsticks are generated empirically by analysing features of existing methods to work out which factors have been considered useful in the past. This technique focuses on the internal validity of RE methods and thus is concerned with its consistency and completeness. However, it is not always possible to generate yardsticks from the methods that one is attempting to evaluate. Sometimes, simply because comparable methods do not exist in a given area.

Case studies are another technique to evaluate RE methods. They provide an in-depth analysis of how the examined method is used. Results of the analysis can be compared with the claims of the method. An essential prerequisite for the realisation of case studies is that evaluators are well trained in using the method. This is often difficult to decide. Moreover, the reliability of a method can be difficult to measure, because the same case is not repeatable, and case studies rarely involve comparisons.

Another strategy is to survey those who apply the method. This means that the value of RE methods is assessed from the analysts' point of view. However, the difficulty lies in the separation of methodical factors from contextual ones. If a RE method is evaluated as inadequate for a particular application domain this can be due to the method itself, to low motivation for its use or acceptance, to poorly trained requirements analysts, etc.

Evaluation and comparison of RE methods are extremely difficult, since there are no objective criteria. The evaluation depends upon several factors, e.g. the level of integration between methods, the suitability for specific situations, experience with the methods, skills and expertise required in using them. Because of these dependencies, the (long-term) consequences and benefits of methods are extremely difficult to evaluate and therefore difficult to generalise.

Nevertheless, for the purpose of this survey we will develop in the next section our own set of criteria. We use them to characterise selected methods.

2 Characterisation Categories

Methods range from process-oriented to data-oriented methods. Some methods focus more on the viewpoints necessary to specify the software system, whereas other methods concentrate on the resulting view — the software specification. Figure 5 shows a qualitative assessment of surveyed methods.

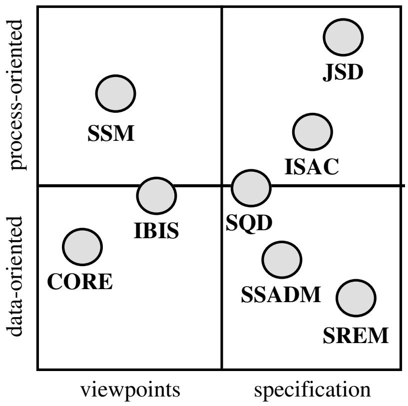

Figure 5. Overview of Methods

To characterise and to compare representative RE methods, we have developed four criteria: (1) *Goals of Method*; (2) *Assumptions of Method*; (3) *Requirements Engineering Process*; and (4) available *Tools*. These four criteria offer a sufficiently comprehensive perspective for RE methods.

The purpose of the following sections is not to outline all RE methods and tools in an encyclopaedic way, but to give a sufficient flavour of a representative selection of them. This information should help to decide whether the consideration of a method is worthwhile for the software project.

The following synopses are necessarily simplifications and thus authoritative references and suppliers are indispensable to learn how to use the methods appropriately.

3 COntrolled Requirements Expression

CORE is a requirements analysis and specification method that was developed in 1979 and work on CORE is still continuing.

References. Mullery 1979 and 1985; System Designers 1985.

Evaluation Studies. Stokes 1991; Finkelstein et al. 1992.

Goals of Method. CORE is a prescriptive, top-down method. It identifies the clients and their viewpoints, and describes dynamic aspects of the software system with the primary aim to determine functional requirements.

Assumptions of Method. CORE assumes that different viewpoints are necessary to specify requirements. Example viewpoints are environment view (e.g. user interfaces), process view (e.g. project schedules, information flows), and reliability view (e.g. failure conditions).

CORE defines responsibilities for clients and requirements analysts (viewpoint authorities) and structures their communication. For example, the customer is responsible to resolve conflicting views. In CORE the rationale for each requirement rests with the clients and is not documented in the RE process.

The requirements analyst is responsible for the documentation of requirements specifications, which express necessary viewpoints and ensure their successful combination into a coherent specification of the software system.

Requirements Engineering Process. CORE proceeds iteratively, first defining the problem, then decomposing it into viewpoints, and acquiring information about each viewpoint. This recurrent process leads to a hierarchy of viewpoints. The RE process comprises viewpoint structuring, tabular collection, data structuring, single and combined viewpoint modeling, and constraint analysis.

Viewpoint structuring delimits the application domain and partitions it in (two to five) viewpoints, e.g. view of customer, maintenance and user organisation. Viewpoints are the set of interests to be supported or influenced by the proposed system. The customer decides for each viewpoint whether it is to interact with the system or whether it causes one of the other viewpoints to interact with the system (indirect viewpoint). In the analysis output and input of indirect viewpoints are considered, but their actions are not.

The method is to interview responsible users and technical experts, to prepare a tabular collection form, a set of CORE diagrams at several levels of detail, and a single viewpoint model.

The tabular collection form defines for each viewpoint the input, its actions, the output, and their sources and destinations. The purpose of data structuring is to analyse the output of each viewpoint to detect its hierarchic and derivation dependencies imposed by the environment. Inputs and actions are usually not considered for data structuring. CORE diagrams are used to show the results of data structuring.

A single viewpoint model investigates a viewpoint in isolation. It is essentially a graphical representation that combines CORE diagrams and the tabular collection form. For their coherent integration single viewpoint models include additional data flows.

Combined viewpoint modeling attempts to integrate isolated viewpoints into one representation ensuring consistency and completeness. The activity results in a set of CORE diagrams showing how the various viewpoints fit together. Once the individual viewpoints have been reconciled completely, the analyst interviews users to acquire and assess some aspects of non-functional requirements.

Tools. Commercial support for the technique is available in the UK from Systems Designers Plc. British Aerospace has combined CORE with PSL/PSA in their SAFRA (Semi Automated Functional Requirements Analysis) project.

4 Information Systems Work and Analysis of Changes

ISAC was developed by a research group at the Royal Institute of Technology and at the University of Stockholm, Sweden.

References. Lundeberg, Goldkuhl and Nilsson 1981.

Evaluation Studies. Taggart & Tharp 1977; Winter, Scheschonk and Gier 1986; Olerup 1991.

Goals of Method. ISAC explores the needs and problems of the client and investigates their rationale. It applies a step-by-step approach to specify manual routines and software systems, where they can contribute to organisational performance.

Assumptions of Method. The theory of information systems as suggested by Langefors provides the basis for ISAC [Langefors 1973]. In the theory of information systems, requirements are derived from goals. Goals refer to desirable behaviours and states of the organisation.

Though ISAC is thoroughly grounded in the Langeforsian theory, it advocates user participation as an important prerequisite for specifying software systems. Moreover, RE is regarded as a communication and learning process.

The role of the requirements analyst is that of a catalyst, i.e. a person who gives support when needed, instead of an expert who knows best and proposes results.

Requirements Engineering Process. The RE process comprises the activities change analysis, activity study, and information analysis.

Change analysis starts with problem analysis resulting in a list of clients and a first statement of their needs and problems. Activities in the application domain are documented as activity graphs (A-graphs) and property tables. These are the basis for the

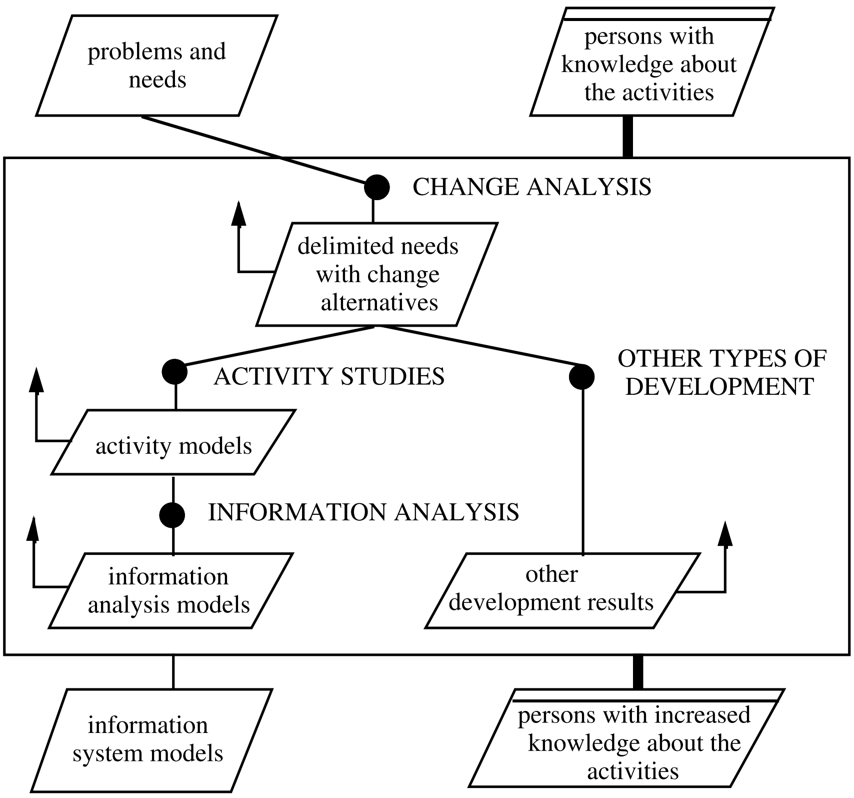

Figure 6. ISAC Process

generation and evaluation of change alternatives, each described in A-graphs and property tables.

If change analysis results in the demand for a software system, then the activity study is carried out. The activity study focuses on problems and needs of the client and not on existing technical aids. It delimits the software system in the activities of the organisation and defines its inputs and outputs. The results of this activity are documented in detailed A-graphs and property tables.

The information analysis elaborates the outputs of the proposed software system. Then it studies the processes necessary to arrive at these outputs. Information analysis starts with the construction of information precedence graphs (I-graphs) from the corresponding A-graphs. After that, precedence analysis is performed alternatively to component analysis. Component analysis is the study of the contents and structure of different information. Precedence analysis is documented among other things in form of I-graphs and the results of component analysis are shown with component relation graphs (C-graphs).

Tools. Courses in ISAC have been given in Sweden since 1969. A tool for the method is ISAC-GRAPH, a diagram editor running on MS-DOS and Unix.

5 Issue-Based Information Systems

IBIS was developed by Horst Rittel at the University of California, Berkeley and is further investigated at MCC and Corporate Memory Systems.

References. Kunz and Rittel 1970; Conklin and Begeman 1988; Rein and Ellis 1991.

Evaluation Studies. Yakemovic and Conklin 1990; Shum 1991.

Goals of Method. IBIS is a structuring method which organises and tracks the rationale underlying requirements. The method helps to point out where alternatives have not been considered and where specifications are strong or weak, based on supporting and objecting arguments.

Assumptions of Method. IBIS assumes that RE is fundamentally a conversation among clients and requirements analysts in which they pool their respective knowledge and viewpoints to specify the software system. The method acquires requirements by

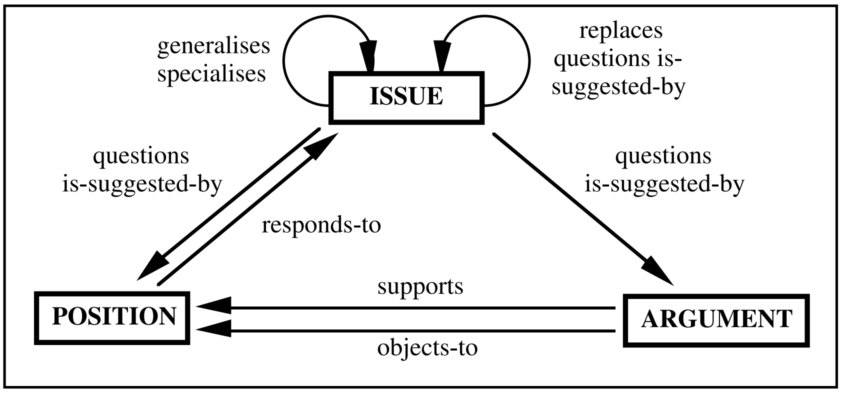

Figure 7. Rhetorical Moves in IBIS

keeping track of issues being discussed, propositions on these issues, and arguments in support of a position or object to it. Any problem, concern, or question can be an issue and may require discussion (if not agreement) to proceed with RE. IBIS offers no support for resolving issues and reaching consensus, e.g. for choosing among the various positions of an issue, as this is the responsibility of the people involved.

The issue-position-argument framework helps to focus on critical parts of the problem and to improve detection of incompleteness or inconsistency. It also makes assumptions and definitions more explicit.

Requirements Engineering Process. IBIS focuses on the articulation of key issues in RE. Each issue can have many positions. A position is a statement or assertion that resolves the issue. Each of an issue's positions, in turn, can have one or more arguments that either support that position or are object to it. Thus, each separate issue is the root of a (possibly empty) tree, with the successors of the issue being positions and the successors of the positions being arguments.

There are nine linking types in IBIS. For example, a position responds-to an issue, and this is the only place the responds-to link can be used. Arguments must be linked to their positions with either supports or objects-to links. Issues generalise or specialise other issues and can question or be-suggested-by other issues, positions, and arguments.

A typical IBIS discussion begins with someone posting an issue node containing a question such as “How should we do X?” That person can post a position node proposing one way to do X and can also post argument nodes supporting that position. Another client can post a competing position responding to the issue and providing a set of arguments. Others post additional positions or arguments which support or object to any of the positions. In addition, new issues, which are raised by the discussion, may be posted and linked into the nodes that most directly suggested them. Figure 7 shows a state transition diagram specifying legal moves within IBIS.

IBIS offers no stopping rule, nor has IBIS a way of registering that an issue has been resolved by agreement. Rather, the goal of the discussion is to understand each others' proposals and to persuade others of one's viewpoint.

Tools. gIBIS (graphical IBIS) is a SUN workstation-based tool which provides a graphical interface for building and browsing IBIS networks. rIBIS (real IBIS) is a hypertext system, which allows a distributed set of users to simultaneously browse and edit multiple views on a hypertext network. It is written in C and runs under Unix. CM/1 is a graphical groupware tool based on IBIS that runs on PC/Windows and Unix.

6 Jackson System Development

The Jackson System Development Method was developed by Michael Jackson and John Cameron in the mid 1970s.

References. Cameron 1986 and 1989; Jackson 1983.

Evaluation Studies. Potts 1989; Floyd 1986.

Goals of Method. JSD comprises software development starting from problem analysis and ending with code. The method does not subscribe to traditional life cycle models. Instead of top-down, it develops functional specification middle-out from an initial set of requirements.

Assumptions of Method. Based on Jackson Structured Programming (JSP), JSD applies JSP principles to RE. JSD is based on the concept of communicating sequential processes [Hoare 1978].

JSD describes the application domain as a distributed network of sequential processes. Processes are the means by which objects of the application domain are represented. They have local data, can communicate via message passing, and have read-only access to each other’s data. States of a process are viewed as homomorphic to states of the object being modelled.

JSD allows process hierarchies to represent different views of the same problem. The network of sequential processes is called the JSD specification, which is a principally executable version of the required software system.

The requirements analyst specifies the processes upon which system functions are to be based, determines the required functions and inserts them into model processes, and transforms the resulting specification into one realisable in the target environment.

Requirements Engineering Process. JSD follows five steps to specify requirements: entity-action and entity structuring, initial modeling, function step, information function step, and system timing.

The first two steps are concerned with creating a model of the application domain, the others with the specification of functions to be added to the model.

In entity-action and entity structuring, an abstract description of the application domain is created in terms of actions and entities. Attention focuses on the application domain and not so much on the intended software system. Entities and their actions are documented and constraints on time-ordering of actions are expressed as structure diagrams.

Initial modeling produces a model of the application domain in terms of communicating processes that correspond to actions in the structure diagrams. The definition of connections between model processes and the application domain is necessary to adapt to changes in the application domain.

The function step inserts the functions into the model as (executable) operations that produce required outputs. Functions generate either output messages for external users (information functions) or actions to be fed into the model (interactive functions).

In the preceding steps of JSD, a network of processes has been created. The purpose of system timing is to specify constraints on the implementation. The two issues considered are speed of execution and relative scheduling. Speed of execution is defined in terms of elapsed time between occurrence of an event and completion of all processing that can be scheduled because of that event. Relative scheduling is defined in terms of how far one process may advance about another when neither is blocked.

Tools. JSD support tools include (graphical) editors, code generators and JSP-Cobol, a pre-processor for suspend-and-resume mechanisms and testing aids.

7 Software Quality Deployment

Since 1972 Quality Deployment has been widely practised in Japan. In software development[3] it is used since the early 1980s.

References. Akao 1990; Sullivan 1986; Hauser and Clausing 1988.

Evaluation Studies. Bossert 1991; Zultner 1992.

Goals of Method. SQD is a quality system for client satisfaction. It investigates needs and requirements of the client with the aim to both improve the software system and its development process.

Assumptions of Method. A major assumption is that customer needs are more important than the value judgement of requirements analysts. The method is driven by “the voice of the customer.”

SQD uses a series of matrices to deploy customer and quality requirements. These deployments are a structured way of dealing with a special concern at a detailed level, and provide a basis for linking these concerns into an integrated framework. SQD identifies, explicitly states, and prioritises requirements.

Customer deployment precedes the specification of requirements in quality deployment. It identifies the customer, and explores and specifies high-value customer requirements (demanded quality). The customer requirements are translated and deployed into technical requirements (quality characteristics) in the so-called House of Quality matrix.

However, the method offers no “stopping” condition for the RE process. It is the responsibility of the people who apply the method to decide when the requirements are completely specified.

Requirements Engineering Process. Following a four phase approach, the RE process comprises the client characteristics/client segments matrix, the client

---

[3] SQD is mainly applied in Japan and the US. However, SQD is fundamentally different in US and Japanese companies. In Japanese companies, the customer's voice drives all activities, while in many US companies, it is the executive's voice or the analyst's voice that prevails. That is, the Japanese put more effort into designing quality at the product development stage, while US companies put a greater emphasis on problem solving.

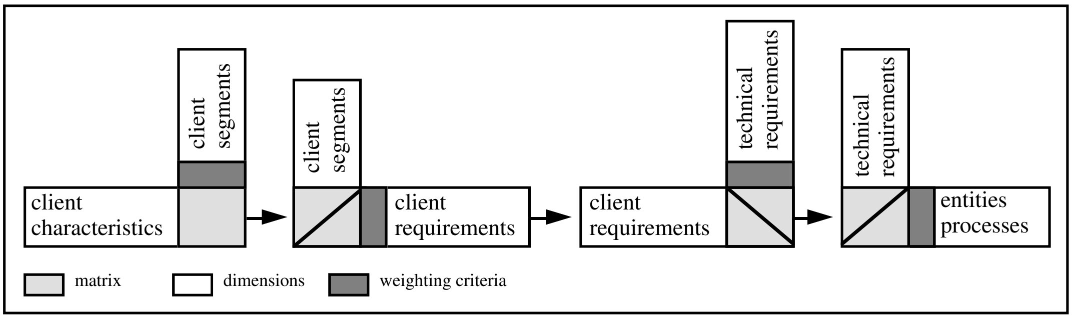

Figure 8. Software Quality Deployment

segments/client requirements matrix, the client requirements/technical requirements matrix, and technical requirements/entities-and-processes matrix.

First, customers and users are identified and classified in so-called client segments. Moreover, the problem is outlined and the development horizon is defined. Then primary needs of the client are documented and refined. Information about needs can come from various sources, e.g. customer opinion surveys, market research data.

The requirements are then used in the client segments/client requirements matrix to understand which segments value which requirements and to what extent. This matrix indicates whether defined requirements cover client needs and depicts conflicts between needs and requirements. The final priority is then deployed to the client requirements/technical requirements matrix.

The client requirements/technical requirements matrix is the House of Quality matrix. The software characteristics and capabilities necessary to support the client requirements must be determined. Customer importance ratings and strength and weakness analyses of similar software systems are also used to define them.

Technical requirements/entities-and-processes matrix maps the technical requirements into the software specification. For example, entities from entity-relationship diagrams and processes from data-flow diagrams are examined to see to which technical requirements they contribute to. This shows which entities and processes are essential to the customers. The software specification provides the basis for selecting those requirements, which are deployed through the implementation.

8 Soft Systems Methodology

SSM was developed in the 1970s at the University of Lancester and work on it is continuing.

References. Checkland 1981 and 1989; Wilson 1990.

Evaluation Studies. Benyon and Skidmore 1987; Checkland and Scholes 1990.

Goals of Method. SSM views RE as a recurrent learning cycle that aims at improving the problem situation. It applies systems thinking to explore the problem situation and to specify the software systems.

Assumptions of Method. SSM assumes that different views of the application domain exist and that these views must be considered in RE. The method compares perceptions of the problem with models of its application domain. Initiated by this comparison, assumptions about the problem are drawn, challenged and tested. SSM is based on the assumption that RE proceeds via debate and that it is intrinsically participative.

Requirements Engineering Process. Initially, the requirements analyst enters the situation considered problematic and identifies the clients for the software system to be built. A first investigation of the problem situation is carried out and documented (Figure 9).

Next root definitions are formulated. They specify the purpose of the system in terms of transformation processes. In constructing root definitions one has to consider the mnemonic CATWOE, where the C stands for customer as the victim or beneficiary of T, the transformation process. A denotes the actors who perform the activities of the system, whereas O is the owner who can stop these activities, and E are the environmental constraints. W describes the world view or Weltanschauung when formulating root definitions.

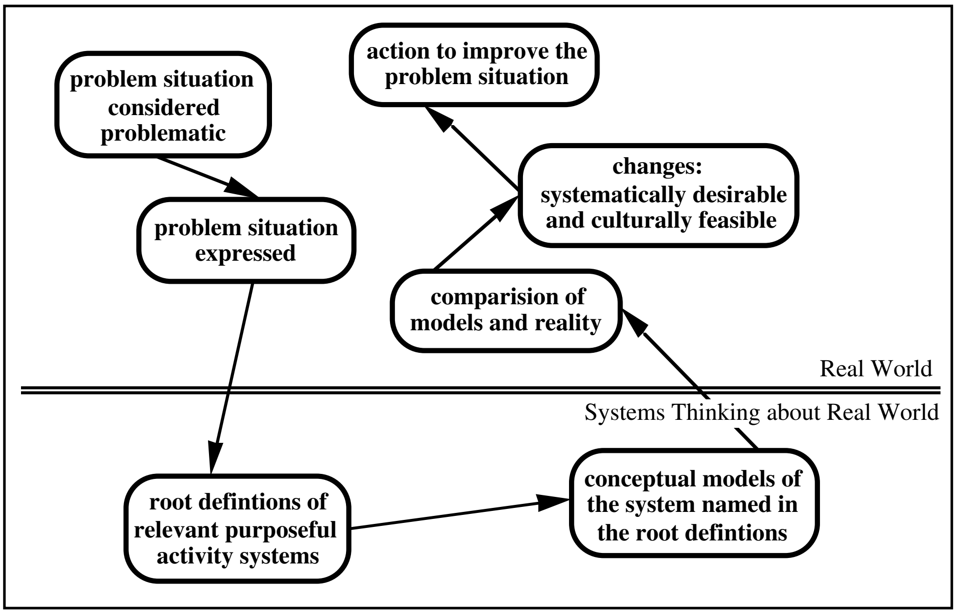

Figure 9. Soft Systems Methodology

Conceptual modeling assembles the verbs describing system activities and structures them according to logical dependencies to achieve transformation processes named in the root definition. Resulting models are enhanced by monitoring and control sub-systems to (ideally) allow the system to adapt to changing environments. The monitoring and control sub-systems are described by efficacy (it works), efficiency (minimum use of resources), and effectiveness (the right thing to do).

The fifth stage of SSM compares conceptual models with perceived reality using informal discussion, formal questioning, scenario writing based on operating the model, and trying to model the problem in the same structure as the conceptual model.

The comparison stage uses the differences between models and reality to discuss changes which are systematically desirable and culturally feasible concerning the world views specified in root definitions. Examples for systematically desirable changes are instituting mechanisms for assessing effectiveness, making sure resources are appropriate, ensuring that logical dependencies in the model are reflected in actions of the problem situation, etc. Then the cycle of SSM is completed by implementing these changes.

9 Software Requirements Engineering Methodology

In the late 1970s, SREM was developed by TRW under contract to the US Army Ballistic Missile Defence Advance Center.

References. Alford 1977 and 1990; Bell, Bixler and Dyer 1977.

Evaluation Studies. Davis 1988; Scheffer, Stone and Rzepka 1985.

Goals of Method. SREM[4] is a formal, systematic method progressing from system requirements to functional software requirements by taking performance requirements into account. It aims at explicitly and consistently documented functional requirements that are testable and traceable.

Assumptions of Method. SREM views any system as a finite-state machine and specifies its inputs, outputs, states and functions (transformations). Input and output are conceived as messages across subsystems.

---

[4] An enhancement to SREM, called Systems Requirements Engineering Methodology, allows to specify system requirements as time functions and their decomposition and allocation to software components.

The method is determined by its tools (Figure 10). It includes an executable, pseudo-English Requirements Statement Language (RSL) and a Requirements Engineering Validation System (REVS) to automatically process and translate RSL specifications to data base statements.

RSL is based on the four primitives elements, relationships, attributes and structures of requirements. REVS is the tools part of SREM and offers automated consistency and completeness analysis, information extraction and document production with graphical descriptions, and simulator generation. It uses a relational database (called the Abstract System Semantic Model) to capture requirements processed from RSL.

Requirements Engineering Process. First, all input and output messages of the problem situation are identified. Then the requirements analyst constructs for each input a requirements network (R-net). R-nets are a graphical notation for RSL to state input-output transformations. Input to R-net triggers transformations and events inside the R-net, computation of its output and triggers other R-nets. Transformations (functions) in an R-net are called ALPHA.

R-Nets are represented in RSL and then translated into an abstract form stored in the REVS relational database. REVS provides tools for examining the database to check the static consistency and completeness of the requirements.

Then each R-net is specified in more detail leading to a decomposition of the system. The requirements are refined until all processing required to satisfy the system requirements has been incorporated. Validation points at which performance requirements must be met are also included in the R-nets. The simulation of selected functions is

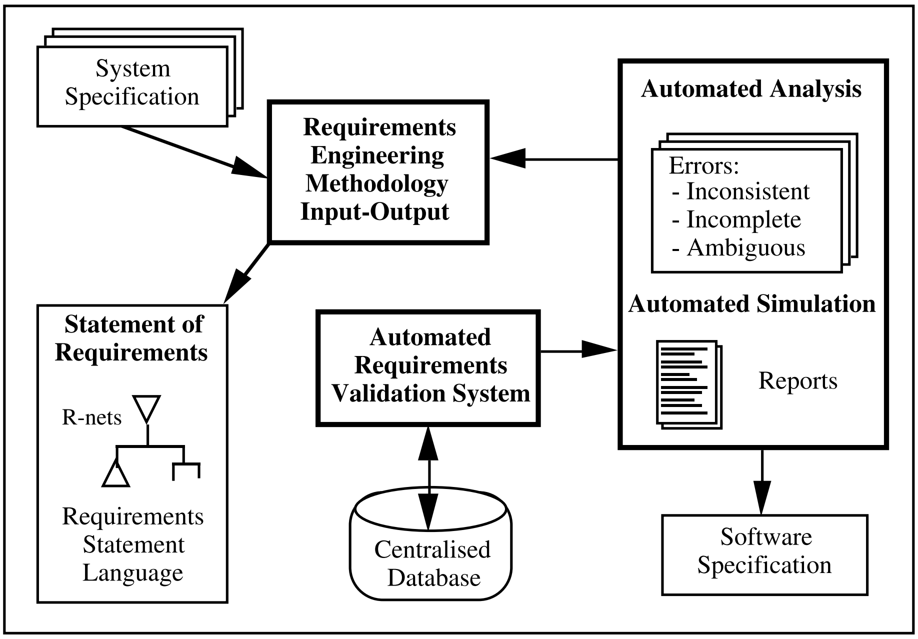

Figure 10. Software Requirements Engineering Methodology

used to check their correctness.

The next step defines traceable and testable performance requirements for the proposed software system. To specify performance requirements, the processing path can be defined as a sequence of validation points with path conditions. This allows to specify, for example, response-time requirements (minimum, maximum) and accuracy requirements (recorded data).

Finally, SREM uses prototypes for critical algorithms to simulate performance characteristics and therefore to demonstrate the feasibility of the software system.

Tools. The methodology is supported by Requirements Engineering and Validation System (REVS), an automated tool system.

10 Structured Systems And Design Methodology

SSADM was originally developed for the UK Central Government by Learmonth and Burchett Management Systems and the Government's computing advisory board.

References. Ashworth & Goodland 1990; Malcolm 1992; Robinson 1990.

Evaluation Studies. Ashworth 1989; Färberböck, Gutzwiller, Heym 1991.

Goals of Method. SSADM aims at the production of administrative data processing systems. It concentrates on specifying functional requirements, but also provides some guidance to non-functional requirements.

Assumptions of Method. SSADM views software systems as models of the problem situation. Software projects concentrate on three models, or views of data, for specifying the software system. These (intermediate) models are the logical data model (LDM), data flow models (DFMs) and entity life histories (ELHs), which are transformed into the software system.

LDM and DFMs represent the static structure of the software system, whereas the behavioural aspects are represented by ELHs. The LDM partitions the problem situation into a collection of entities, which represent the objects being modelled and their relations. DFMs result from document flow analysis of the working environment, its structuring into a hierarchy of data flow diagrams (DFDs), and the abstraction of

DFMs. The abstraction removes implementation details from DFMs to reveal the structure of the software system.

For EHLs each entity is examined to see which events affect it during its time in the system and in what possible sequence. EHLs add a dynamic dimension to the static model of data analysis.

Requirements Engineering Process. The RE process comprises the stages investigation of working environment, business system options, and definition of requirements.

The investigation of the working environment is the stage where the scope and size of the project are agreed and plans are drawn up for the RE process. Also an initial analysis of the problem is carried out, detailed task allocation of project members takes place, interview plans are set up, and an initial requirements definition is developed representing users needs. Moreover, DFMs are used to represent the working environment. A LDM is constructed producing a logical DFM of the working environment.

In the business systems options stage, LDM and DFM are used together with a requirements definitions to generate (typically up to three) business system options that cover the requirements. The options are associated with different cost/benefit metrics and are prioritised on their basis.

The selected business option is developed into a detailed requirements specification. The logical data model is extended to deliver the functionality required by the selected business system option, and so is the logical DFM. The logical DFM is then used to derive software functions.

Input-output structures of functions are subjected to relational data analysis and the results are used to enhance the logical data model. They are also used to derive prototypes for specified functions. The events identified in the functions and structural components of the logical data model are then used to build ELHs and effect correspondence diagrams to define system behaviour. Enquiry access paths are produced to define the production of system outputs.

Tools. There are various tools. For example, Excelerator and Aut2 are PC-based tools with graphical modelling capability integrated with a repository stored on the PC disk. Screen layouts and data definitions created with Excelerator can be exported to several code generators (e.g. TELON). The Excelerator also provides project management support.

Discussion and Conclusions

We have briefly characterised a representative set of methods. Representative in the sense, that this set shows the variety of approaches to RE. The methods range from process-oriented and data-oriented ones to methods that focus on viewpoints and such that focus on the specification.

Despite its importance, researchers have given RE less attention than design and implementation. Many research challenges persist. For example, the integration of non-functional and functional requirements, the assessment of alternative specifications, the annotation of specifications to document their rationale, the adequate formalisation of specifications, etc.

As any single approach to RE will have varying success across different projects, since not all problems yield in the same way to the same approach, we believe that it is unlikely that the “one best” method exists. The chase for the perfect method is somewhat illusory, because different methods offer different views of the problem situation.

As an alternative to the “one-best-way” approach, we presented the essentials of RE from our point of view. RE is a collaborative activity involving customers, users, requirements analysts, and designers. The RE process is dependent not only on the diversity of the problems perceived, but also on the diversity and experience levels of involved people.

Viewing RE as communication-intensive, co-operative work that constitutes the roots of software development, leads to a research approach that emphasises the bridging of communication gaps. We believe that RE needs a (broader) theoretical foundation to incorporate

- social factors, e.g. which groups are involved and what are their experience
- problem-specific factors, e.g. domain structure and available standards
- organisational factors, e.g. time and cost budgets
- design factors, e.g. user interface.

To improve the process of RE we have to include the possibility for frequent two-way transformation between informal and formal aspects of RE. Tools are indispensable for combining extended requirements representations (e.g. with exceptions and multiple conflicting viewpoints), semi-formal graphs, and informal hypermedia representations. We have to ensure compatibility, on the formal side with emerging formal specification methods and on the informal side with established practice.

References

Definitions and Properties of Requirements Engineering

Boehm, B.W. (1979): “Guidelines for Verifying and Validating Software Requirements and Design Specifications”. In: Samet, P.A. (ed.): IFIP 1979, North-Holland: Amsterdam. 711-719.

Boehm, B.W. (1989): “Software Risk Management”. IEEE Computer Society Press: Washington.

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

Brooks, F.P. (1987): “No Silver Bullet: Essence and Accidents of Software Engineering”. IEEE Computer, vol. 20, no. 4, 10-19.

Dreyfus, H.L.; Dreyfus, S.E. (1986): “Mind over Machines”. The Free Press: New York.

Finkelstein, A.; Kramer, J.; Nuseibeh, B.; Finkelstein, L.; Goedicke, M. (1992): “Viewpoints: A Framework for Integration Multiple Perspectives in System Development”. International Journal of Software Engineering and Knowledge Engineering, vol. 1, no. 2, 31-58.

Fuchs, N.E. (1993): “Compositional Software Development”. Technical Report, Institute for Informatics, University of Zurich, Zurich.

Greenspan, S. (1991): “The Scruffy Side of Requirements Engineering”. In: Van Lamsweerde, A.; Fugetta, A. (eds.): Third European Software Engineering Conference, Springer: Milan. 492-494.

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

IEEE Std.830-1984 (1984): “IEEE Guide to Software Requirements Specification”. The Institute of Electrical and Electronics Engineers: New York.

Leite, J.C. (1987): “A Survey on Requirements Analysis”. Technical Report RTP-071, Department of Information and Computer Science, University of California at Irvine.

McCracken, D.D.; Jackson, M.A. (1982): “Life Cycle Concepts Considered Harmful”. ACM Software Engineering Notes, vol. 7, no. 2, 28-32.

Mostow, J. (1985): “What is AI? And What Does It have to Do with Software Engineering?”. IEEE Transactions of Software Engineering, vol. SE-11, no. 11, 1253-1256.

Naur, P.; Randell, B. (eds.) (1969): “Software Engineering: Report on a Conference”. NATO Scientific Affairs Division: Garmisch.

STARTS Guide (1987): “Requirements Definition and Design”. In: The STARTS Guide: A guide to methods and software tools for the construction of large real-time systems, NCC Publications: Manchester. 177-223.

Swartout, W.; Balzer, R. (1982): “On the Inevitable Interwining of Specification and Implementation”. Communication of the ACM, vol. 25, no. 7, 438-440.

Varela, F.J.; Thompson, E.; Rosch, E. (1992): “The Embodied Mind: Cognitive Science and Human Experience”. MIT Press: Cambridge.

Wittgenstein, L. (1960): “Schriften” (in German). Suhrkamp: Frankfurt.

Zave, P. (1984): “The Operational versus the Conventional Approach to Software Development”. Communications of the ACM, vol. 27, no. 2, 104-118.

Zuboff, S. (1989): “In the Age of the Smart Machine”. Basic Books: New York.

Essentials of Requirements Engineering

Bischofberger, W.; Pomberger, G. (1992): “Prototyping-Oriented Software Development - Concepts and Tools”. Springer: Berlin.

Budde, R.; Kautz, K.; Kuhlenkamp, K.; Mathiassen, L.; Züllighoven, H. (1992): “Prototyping: An Approach to Evolutionary System Development”. Springer: Berlin.

Burton, A.M.; Shadbolt, N.R.; Rugg, G.; Hedgecock, A.P. (1990): “The Efficacy of Elicitation Techniques: A Comparison across Domains and Levels of Expertise”. Knowledge Acquisition, vol. 2, no. 2, 167-178.

Conklin, J. (1987): “Hypertext: An Introduction and Survey”. IEEE Computer, vol. 20, no. 9, 17-41.

DeMarco, T. (1979): “Structured Analysis and System Specification”. Prentice Hall: Englewood Cliffs.

Diller, A. (1990): “Z: An Introduction to Formal Methods”. Wiley: Chichester.

Dodd, T. (1990): “Prolog — A Logical Approach”. Oxford Science Publications: Oxford.

Ericsson, K.A.; Simon, H.A. (1984): “Protocol Analysis: Verbal Reports as Data”. MIT Press: Cambridge.

Floyd, C.; Reisin, F.-M.; Schmidt, G. (1989): “STEPS to Software Development with Users”. In: Ghezzi, C.; McDermid, J.A. (eds.): European Software Engineering Conference, Springer: Berlin. 48-64.

Freedman, D.P.; Weinberg, G.M. (1990): “Handbook of Walkthroughs, Inspections, and Technical Reviews”. Dorset House: New York.

Fuchs, N.E. (1992): “Specifications are (preferably) executable”. Software Engineering Journal, vol. 7, no. 5, 323-334.

Gause, D.C.; Weinberg, G.M. (1991): “Exploring Requirements: Quality Before Design”. Dorset House: New York.

Greenbaum, J.; Kyng, M. (1991): “Design at Work”. Erlbaum: Hillsdale.

Hackman, J.R.; Oldham, G.R. (1975): “Development of the Job Diagnostic Survey”. Journal of Applied Psychology, vol. 60, no. 2, 159-170.

Martin, C.F. (1988): “User-Centerd Requirements Analysis”. Prentice Hall: Englewood Cliffs.

Mitroff, I.I.; Mason, R.O. (1981): “Creating a Dialectical Social Science”. Reidel: Dordrecht.

Mittermeir, R.T.; Roussopoulos, N.; Yeh, R.T.; Ng, P.A. (1990): “An Integrated Approach to Requirements Analysis”. In: Ng, P.A.; Yeh, R.T. (eds.): Modern Software Engineering: Foundations and Current Perspectives, Van Nostrand Reinhold: New York. 119-164.

Partsch, H.A. (1991): “Requirements Engineering” (in German). Oldenbourg: Munich.

Peterson, J.L. (1977): “Petri Nets”. Computing Surveys, vol. 9, no. 3, 223-252.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. (1991): “Object-Oriented Modeling and Design”. Prentice Hall: Englewood Cliffs.

Simon, H. (1970): “The Sciences of the Artificial”. MIT Press: Cambridge.

Simon, H. (1973): “The structure of Ill-Structured Problems”. Artificial Intelligence, vol. 4, no. 2, 181-201.

Sommerville, I. (1992): “Software Engineering”. Addison-Wesley: Wokingham.

Sowa, J.F. (1984): “Conceptual Structures: Information Processing in Mind and Machine”. Addison Wesley: Reading.

Thayer, R.H.; Dorfman, M. (1990): “System and Software Requirements Engineering”. IEEE Computer Society Press: Los Alamos.

Ulich, E. (1992): “Arbeitspsychologie” (in German). Verlag der Fachvereine: Zurich.

Existing Methods

Akao, Y. (1990): “Quality Function Deployment: Integrating Customer Requirements into Product Design”. Productivity Press: Cambridge.

Alford, M. (1977): “A Requirements Engineering Methodology for Real-Time Processing Requirements”. IEEE Transactions on Software Engineering, vol. SE-3, no. 1, 60-69.

Alford, M. (1990): “Software Requirements Engineering Methodology (SREM) At the Age of Eleven - Requirements Driven Design”. In: Ng, P.A.; Yeh, R.T. (eds.): Modern Software Engineering: Foundations and Current Perspectives, Van Nostrand Reinhold: New York. 351-377.

Ashworth, C.; Goodland, M. (1990): “SSADM: A Practical Approach”. McGraw-Hill: London.

Ashworth, C.M. (1989): “Using SSADM to Specify Requirements”. In: Professional Software Engineering Group C1 (ed.): Colloquium on Requirements Capture and Specification for Critical Systems, The Institution of Electrical Engineers: London.

Bell, T.E.; Bixler, D.C.; Dyer, M.E. (1977): “An Extendable Approach to Computer-Aided Software Requirements Engineering”. IEEE Transactions on Software Engineering, vol. SE-3, no. 1, 49-60.

Benyon, D.; Skidmore, S. (1987): “Towards a Tool Kit for the Systems Analyst”. The Computer Journal, vol. 30, no. 1, 2-7.

Bossert, J.L. (1991): “Quality Function Deployment: A Practitioner's Approach”. ASQC Quality Press: Milwaukee.

Byrd, T.A.; Cossick, K.L.; Zmud, R.W. (1992): “A Synthesis of Research on Requirements Analysis and Knowledge Acquisition Techniques”. MIS Quarterly, vol. 16, no. 1, 117-138.

Cameron, J.R. (1986): “An Overview of JSD”. IEEE Transactions on Software Engineering, vol. 12, no. 2, 222-240.

Cameron, J.R. (1989): “JSP and JSD: The Jackson Approach to Software Development”. IEEE Computer Society Press: Washington.

Checkland, P.B. (1981): “Systems Thinking, Systems Practice”. John Wiley & Sons: Chichester.

Checkland, P.B. (1989): “Soft Systems Methodology”. In: Rosenhead, J. (ed.): Rational Analysis for a Problematic World, John Wiley & Sons: Chichester. 71-100.

Checkland, P.B.; Scholes, J. (1990): “Soft Systems Methodology in Action”. John Wiley & Sons: Chichester.

Conklin, J.E.; Begeman, M.L. (1988): “gIBIS: A Hypertext Tool for Exploratory Policy Discussion”. ACM Transactions on Office Information Systems, vol. 6, no. 4, 303-331.

Davis, A.M. (1988): “A Comparison of Techniques for the Specification of External System Behavior”. Communication of the ACM, vol. 31, no. 9, 1098-1115.

Färberböck, H.; Gutzwiller, T.; Heym, M. (1991): “Ein Vergleich von Requirements Engineering Methoden auf Metamodell-Basis” (in German). In: Trimm, M. (ed.): Requirements Engineering '91, Springer: Marburg. 40-66.

Finkelstein, L.; Kramer, J.; Finkelstein, L.; Nuseibeh, B. (1992): “Using Software Specification Methods for Measurement Instrument Systems”. Measurement Journal, vol. 10, no. 2, 79-86.

Floyd, C. (1986): “A Comparative Evaluation of System Development Methods”. In: Olle, T.W.; Sol, H.G.; Verrijn-Stuart, M. (eds.): Information Systems Design Methodologies: Improving the Practice, Elsevier Science Publisher: Amsterdam. 19-54.

Fowler, C.J.; Kirby, M.A.; Macaulay, L.A. (1990): “Historical Analysis: A Method for Evaluating Requirements Capture Methodologies”. Interacting with Computers, vol. 2, no. 2, 190-204.

Hauser, J.R.; Clausing, D. (1988): “The House of Quality”. Harvard Business Review, vol. 66, no. 3, 63-73.

Hoare, C.A.R. (1978): “Communicating Sequential Processes”. Communications of the ACM, vol. 21, no. 8, 666-677.

Jackson, M.A. (1983): “Systems Development”. Prentice Hall: Englewood Cliffs.

Klein, H.K.; Hirschheim, R.; Nissen, H.-E. (1991): “A Pluralist Perspective of the Information Systems Research Area”. In: Nissen, H.-E.; Klein, H.K.; Hirschheim, R. (eds.): Information Systems Research: Contemporary Approaches and Emergent Traditions, Elsevier Science Publishers: Amsterdam. 1-20.

Kunz, W.; Rittel, H. (1970): “Issues as Elements of Information Systems”. Working Paper No. 131, Institute of Urban and Regional Development, University of California, Berkeley.

Langefors, B. (1973): “Theoretical Analysis of Information Systems”. Studentliteratur: Lund.

Lundeberg, M.; Goldkuhl, G.; Nilsson, A. (1981): “Information Systems Development: A Systematic Approach”. Prentice Hall: Englewood Cliffs.

Malcolm, E. (1992): “SSADM Version 4: A User's Guide”. McGraw-Hill: London.

Mullery, G.P. (1979): “CORE — A Method for Controlled Requirements Specification”. Fourth International Conference on Software Engineering, The Institution of Electrical Engineers: London. 126-135.

Mullery, G.P. (1985): “Acquisition-Environment”. In: Alford, M.W.; Ansort, J.P.; Hommel, G.; Lamport, B.; Liskov, B.; Mullery, J.; Schneider, F.B. (eds.): Distributed Systems: Methods and Tools for Specification, Springer: Berlin. 45-130.

Olerup, A. (1991): “Design Approaches: A Comparative Study of Information System Design and Architectural Design”. The Computer Journal, vol. 34, no. 3, 215-224.

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

Petri, C.A. (1979): “Ansätze zur Organisationstheorie rechnergestützter Informationssysteme” (In German). Oldenbourg: Munich.

Potts, C. (1989): “Requirements Analysis, Domain Knowledge, and Design”. Technical Report STP-313-88, Microelectronics and Computer Technology.

Rein, G.L.; Ellis, C.A. (1991): “rIBIS: A Real-Time Group Hypertext System”. International Journal of Man-Machine Studies, vol. 34, no. 3, 349-367.

Robinson, K. (1990): “Aut2 User Manual: A Workbench Designed by Analysts for Analysts”. Model Systems Ltd.: London.

Roman, G.-C. (1985): “A Taxonomy of Current Issues in Requirements Engineering”. IEEE Computer, vol. 18, no. 4, 14-22.

Ross, D. (1977): “Structured Analysis (SA): A Language for Communicating Ideas”. IEEE Transactions on Software Engineering, vol. 3, no. 1, 16-34.

Scheffer, P.A.; Stone, A.H.; Rzepka, W.E. (1985): “A Case Study of SREM”. IEEE Computer, vol. 18, no. 4, 47-54.

Shum, S. (1991): “Cognitive Dimensions of Design Rationale”. Technical Report EPC-91-114, Rank Xerox EuroPARC, Cambridge.

Stokes, D.A. (1991): “Improving Requirements Analysis: Providing Support for an Extended CORE Method”. DPhil Thesis, University of York, York.

Sullivan, L.P. (1986): “Quality Function Deployment”. Quality Progress, no. 6, 39-50.

Systems Designers (1985): “CORE: The Method”. Systems Designers Scientific: Camberley.

Taggart, W.M.; Tharp, M.O. (1977): “A Survey of Information Requirements Analysis Techniques”. Computing Surveys, vol. 9, no. 4, 273-290.

Wilson, B. (1990): “Systems: Concepts, Methodologies and Applications”. John Wiley & Sons: Chichester.

Winter, C.; Scheschonk, G.; Gier, K. (1986): “ISAC - ein System zur Unterstützung der Systembeschreibung und der Systemanalyse” (in German). Handbuch der Modernen Datenverarbeitung, no. 130, 97-106.

Yadav, S.B.; Bravoco, R.R.; Chatfield, A.T.; Rajukmar, T.M. (1988): “Comparison of Analysis Techniques for Information Requirements Determination”. Communications of the ACM, vol. 31, no. 9, 1090-1097.

Yakemovic, K.C.; Conklin, J.E. (1990): “Report on a Development Project Use of an Issue-Based Information System”. In: Bikson, T.K. (ed.): Conference on Computer-Supported Cooperative Work, ACM: Los Angeles. 105-118.

Zultner, R.E. (1992): “QFD for Software”. American Programmer: Princeton.