Tool Support for Requirements Engineering

Jacob L. Cybulski
Department of Information Systems
University of Melbourne
j.cybulski@dis.unimelb.edu.au

Abstract

This paper discusses the field of requirements engineering. First, the paper sets the developmental context for the discussion of the requirements engineering process. It shows the impact of the process on various phases of the software development life cycle and discusses the process variations in different development models. Subsequently, requirements engineering activities are listed, described and summarised. The paper then draws the reader's attention to the significance of capturing, recording and maintaining informal user requirements. Such requirements are normally expressed in natural language, they are collected in the earliest stages of software development, and they are usually incomplete, ambiguous and inconsistent. Still, they are commonly used as the basis for project negotiation, discussion and later validation. While acknowledging the need for informal requirements, the paper also stresses the importance of mapping them into more formal specifications and designs. The paper then identifies the main problems in the processing of informal requirements text. It then reviews different methods and techniques for requirements management. It assesses these techniques for their ability to assist requirements engineers in the task of collecting informal user requirements, their organisation, analysis and formalisation. The paper concludes by proposing an approach to dealing with software requirements based on the principle of requirements reuse.

1.1 Introduction

Development of large software products often starts with *requirements engineering*, a phase in the process of software construction which attempts to obtain a complete, consistent and unambiguous specification of user requirements [27, 42, 104]. The main objective of this activity is to collect, formalise, analyse and validate requirements so that they could be used in further systems development, i.e. its design and implementation. It is common to make a distinction between *user* and *system requirements* [130, p 256, 199, p 39]:

user requirements or *stakeholder requirements*	What the user requires of the software or system as a whole. User requirements are informal, they are written by the user or taken down by a system analyst in consultation with the user.
system requirements or *requirements specifications*	What is required of the system as a whole, either hardware, software, or both. System requirements are developed by engineers, as a refinement of user requirements, by translating them into engineering terms.

For large systems, user requirements are collected from many *system stakeholders*, i.e. people who will be affected by the system and who have either direct or indirect influence on its requirements [129, p 10]. The stakeholder groups may include end-users, managers responsible for the system use and its deployment, customers of the organisation providing services with the aid of the system, system administrators monitoring and supporting the system, system providers and developers, financiers, regulators and certification authorities, etc.

A documented collection of system requirements, the *system requirements specification (SRS)* document, is a statement of consensus between the suppliers and customers of a software system. It constitutes the basis for the subsequent software design, and provides a reference point for any

future validation of the constructed software. As the quality of requirements has pivotal role in reaching the quality of the final software product [202], thus, it is important that the requirements specification meets its some well-recognised quality standard. Therefore it should identify only the information that is necessary and actually useable in the development of the software project. All such information should be expressed in unambiguous and consistent terms and be complete and verifiable. Individual requirements should be prioritised to allow scheduling of all development tasks and should the user requirements change, the specification must be easily modifiable. Any software development products must also be traceable back to the original requirements statements.

In the early days of software development (1960 till late-1980s), the tasks of requirements processing, which were commonly associated with systems analysis and design, were perceived as narrowly focussed on systems modelling [57]. The activities leading to the acquisition of requirements and the activities associated with their analysis and integration, were not regarded as sufficiently important to warrant any special mention in the mainstream methodologies or the textbooks of the day. At the same time, however, the process of requirements processing was known to be high risk, labour and skill intensive, and very costly.

Only in the recent years (late 1980 till now), requirements processing has been given special attention in the professional literature, at software engineering conferences, and in undergraduate courses. The field of requirements engineering was established, and its development process has been studied to better understand its impact on the rest of the development life-cycle.

Presently, requirements engineering is conducted incrementally and iteratively, and is commonly structured into several phases [99], e.g. elaboration of needs and objectives, requirements acquisition and modelling, generation and evaluation of alternatives, and finally requirements validation. Elicitation of information from users, as manifested in the first two phases of the requirements engineering cycle, is of special interest to this project. This activity in its own right is quite complex and may involve identification of information sources, information gathering, its rationalisation, prioritisation, and its integration into a cohesive requirements document [42].

Requirements engineering has a very unique role in software development. It commonly begins the development life cycle, it is highly focussed, its impact is strong and definite. Its main goals are to set the scope of a software project, to define its objectives and to establish some of its quality criteria. At the same time, however, its influence can be felt throughout the life cycle, i.e. in software specification, design, implementation and testing. Depending on the development process model, requirements engineering can vary tremendously. Such a model can affect requirements work-products, it can affect different stakeholders, it can even move requirements engineering to different points of time in the software life cycle. Nevertheless, the main ingredients of the requirements engineering process stay invariant, i.e. the main deliverables of the process and the activities undertaken as part thereof. Hence, this section discusses the role of the requirements engineering process, its various models, and the activities typically performed in its phases.

1.1.1 Requirements vs. Life Cycle Activities

All of the development activities can be classified into a number of development phases concerned with [21] (see Table 1) project inception, definition of a system (analysis and specification), its design (architectural and detailed) and its production (implementation and testing).[1] Requirements engineering is commonly associated with systems analysis and specification, in practice, however, its activities and its work-products may be spread across the life-cycle. Requirements modelling, analysis and generation of alternatives - the main tasks in requirements engineering - are normally performed during system analysis. At the same time,

[1] This may be followed by system acceptance, operation and evaluation.

	Life-Cycle Activities			
	Inception / Management	**Analysis / Specification**	**Architect. Design / Detailed Design**	**Implementation / Testing**
Person Responsible	business manager / project manager	business analyst / system analyst	software architect / software designer	implementer / tester
Goals	project proposal project plan	requirements specification	software design	working programs
Sample Tasks	project inception feasibility study resource management configuration management domain and reuse management quality assurance	elaboration of needs and objectives requirements acquisition requirements modelling generation and evaluation of alternatives requirements validation	designing software architecture designing data model and flow designing user interfaces designing program logic and control designing tests tracking design decisions	data manipulation program coding program testing integration installation delivery
Sample Work-Products	project proposals feasibility reports management reports project charts and tables domain model enterprise model	pictures, drawings and photographs informal requirements texts data dictionaries diagrams and charts formal specifications prototypes	diagrams and charts decision tables pseudo-code and PDL specification formal designs	source code files and databases
→	**Requirements Engineering**		←	

Table 1: Requirements engineering vs. other life-cycle activities

requirements modelling may involve the elements of functional decomposition and may, thus, lay the foundation for software architecture design. Requirements analysis may focus on the system workflow, data flow or entity relationships, hence, resulting in partial design work-products, e.g. a functional model or a data model. Elaboration of needs and objectives, as well as requirements acquisition, frequently start in the initial stage of project inception, e.g. business planning, feasibility study, domain and enterprise analysis. Requirements validation, on the other hand, may start in system analysis with a model validation, but it can also involve prototype implementation, it can then move into system integration testing and be finally concluded during the system delivery. It is, therefore, quite difficult to place requirements engineering in time and it is equally problematic to determine the work-products created in its course.

In the following sections, we'll look at the development phases affected by the requirements process and we'll consider development artefacts involved in the process.[2] (Refer Table 1.)

1.1.1.1 Inception Phase / Project Management

The inception phase of software development deals with business activities leading to the project definition. In its course, it results in various management documents (reports, budgets and plans)

[2] The following sections are based on the concepts found in various textbooks on software engineering, e.g. by Pressman [175], Sommerville [197], Martin [150], and those found in systems analysis and design, e.g. by Hoffer, George and Valacic [98] or by Whitten and Bentley [221].

and various analysis documents (cost-benefit, break-even or achievability). However, the most valuable of the inception phase work-products are the project proposal and the feasibility study. The first describes in business terms the aims and the scope of the system to be developed - the two vital ingredients of any requirements specification. Feasibility studies, on the other hand, especially those looking at the technical and operational aspects of the system, have a primary impact on the later generation and evaluation of requirements alternatives, their analysis and validation.

Other activities that affect requirements engineering across development life cycle include those commonly associated with project management, i.e. communication of requirements between developers and clients, configuring and tracking dependencies between requirements and design artefacts, managing project resources, constructing and using reusable models of domain knowledge, requirements and designs, or tracking design rationale.

1.1.1.2 Analysis and Specificatön

The analysis phase starts with a detailed analysis of business needs, as included in business documents. Such a statement of needs will subsequently guide all analysis activities leading to gathering, organizing, evaluating and ranking, elaborating and clarifying user requirements. The analysis phase results in a large collection of user requirements outlining the problem and its scope and context from the user perspective. User requirements are most often derived from interviews and observation, hence they are frequently expressed in natural language using problem domain terminology [103]. On the other hand user requirements can also come in a variety of other "natural" forms [228], e.g. legacy system documents, pictures, drawings, photographs, and working prototypes. Whether in text or other media form, all such "natural" documents are commonly informal and unstructured, ambiguous and redundant, inconsistent and incomplete [159].

The specification phase focuses on the formalization and modelling of informal user requirements into a requirements specification document defining system data, functions, non-functional requirements and design constraints. Requirements are commonly specified with the use of requirements templates, data dictionaries, charts and diagrams [e.g. see review in 222]. Less common methods of requirements specification are still in their research phase, e.g. the use of formal specification languages [90], executable specifications [192] or requirements visualisation [164]. As such specifications are to be used later by developers to design a working and deliverable system, demand for their structure, formality, correctness, cohesion and completeness is far greater than is the case with the informal user requirements documents. Nevertheless, due to the fact that users need to validate requirements specification documents, requirements still need to be expressed using problem domain terms and a relatively informal and easy to understand notation. Whenever a formal notation is in use to encode requirements specifications, e.g. work flow diagrams, use case or sequence diagrams, they may have to be either explained to the user in user terms, or to be maintained simultaneously with an informal statement of requirements.

1.1.1.3 Architectural and Detailed Design Phase

The architectural and detailed design phases facilitate definition of system structure, data and process design, integration and test plans, and a definition of individual program units. Some practitioners argue that elements of architectural design should be considered as early as requirements specification [155]. It is, however, more commonly assumed that all design activities, including the system architecture, are independent of individual analysis and specification tasks and instead they stem from information contained in the previously prepared requirements document. Regardless of the design *modus operandi*, requirements and designs are considered to be tightly coupled, to the extent that every element of system design, whether of a high or low abstraction, should be traceable back to some part, or parts, of the requirements

specification. All design documents should be constructed using an appropriate formal design notation understandable and usable by developers, e.g.

- structured techniques may call upon the use of structure charts, data flow diagrams, state transition diagrams, Nassi-Shneiderman charts or pseudocode;
- object-oriented techniques may utilise collaboration diagrams, class diagrams, state charts, component and deployment diagrams.

Formal design notations describe the solution to the user problem, hence, design documents necessarily use terminology drawn from the solution domain, i.e. modules, data structures, processes and functions, objects and classes, their relationships and states.

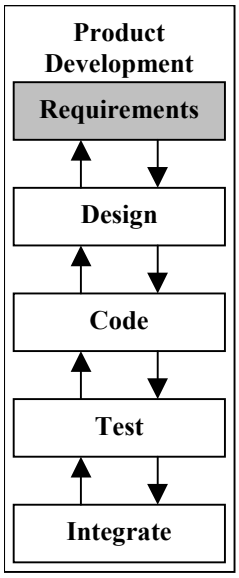

Figure 2: Waterfall model

1.1.1.4 Implementation and Testing Phase

This phase aims at translating software design into reliable program code thatworks in accordance with user requirements. Although this phase seems to be far removed from the requirements engineering stage of software development, the final product's behaviour needs to be tested against user requirements that were laid down in the specification document. Hence, there exists a causal link between these two developmental phases.

Although requirements engineering commonly initiates the software development process, requirements engineering work-products have a tremendous impact on the entire software development life-cycle and its supporting activities. It is, hence, necessary to structure the development process to explicitly take into account requirements engineering activities and their deliverables.

1.1.2 Requirements vs. Software Development Process

All development activities are commonly organised into a cohesive system of phases, steps, checks and contingencies. Together these elements form a software development process, which implies and governs the life cycle of the software product (this itself is the subject of extensive studies). Requirements engineering is normally performed at the beginning of this cycle. It is considered the main part of systems analysis. It is usually preceded by the project inception and then followed by software architectural and detailed design, implementation and testing (Refer back to Table 1). Depending on the adopted software development process, however, requirements engineering may be conducted in a variety of different ways. Its goals and deliverables may differ, its activities and methods disparate, and they may be carried out in different phases of the software life cycle [62]. Consider the following process models and their impact on requirements engineering.

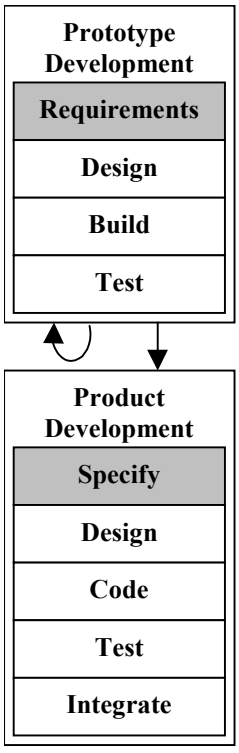

Figure 1: Prototyping model

1.1.2.1 Waterfall Model

In the *waterfall model* (see Figure 2), requirements engineering is completed before any further development task is undertaken. Requirements specification is commonly large and the task of requirements collection is monolithic, time-consuming and expensive. Specifications are considered to be a legally binding contract between customers and developers. The subsequent development process is rigidly determined by the collected

requirements. After the initial approval of formalised requirements, their changes are common, recording and maintaining the record of these alteration is usually difficult and the cost of manual tracking of changes is elevated.

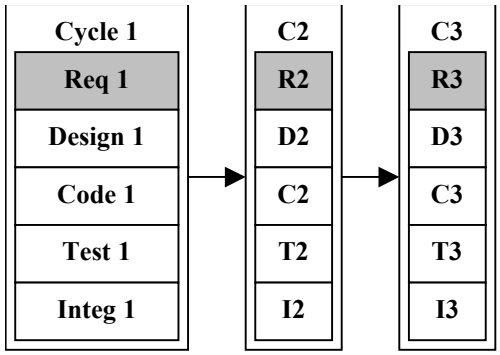

Figure 3: Evolutionary model

1.1.2.2 Prototyping Model

In the *prototyping model* (see Figure 1), on the other hand, the preliminary working model of software is developed with a view to explore and validate requirements, which only then are formally specified. Prototyping is commonly conducted in collaboration with end users, so that continuing user feedback warrants requirements to closely reflect their needs. As the prototype evolves, the requirements also evolve in small incremental steps, hence facilitating improved understanding of individual requirements statements, limiting the impact of requirements on the overall software functionality, and allowing better management of requirements complexity.

1.1.2.3 Evolutionary Model

In the *evolutionary process model* (see Figure 3), requirements engineering is done cyclically throughout the entire development process, with each increment in requirements specification leading to an increment in the system design, implementation, testing, integration and delivery. As in rapid prototyping, users have early opportunity to validate their requirements and to correct their misinterpretations at each cycle of the product delivery. Also at each development cycle, requirements engineers can focus on a limited set of well-defined user needs, hence, development can be controlled more effectively, complexity can be reduced and quality improved. On the other hand, in view of the cyclical nature of the development process, management of software requirements needs to be closely linked with change and configuration management of the entire project, its components and deliverables.

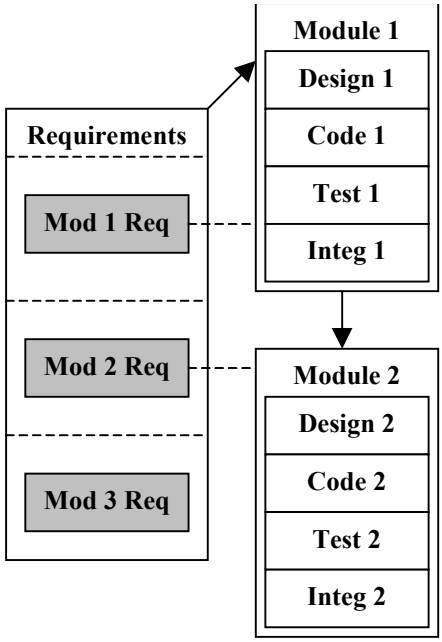

Figure 4: Incremental model

1.1.2.4 Incremental Model

In the *incremental model* (see Figure 4), requirements are collected and specified at the very beginning of the development process, they are subsequently segmented into modules or sub-systems to describe distinct software deliverables, which are then produced in a series of independent development increments (possibly overlapping in time).

The model enforces modular approach to software development early in the life cycle, i.e. during the process of requirements engineering. In some cases, this need for producing modular requirements may increase the complexity of requirements specification. In the long term, however, it also has an effect of separating designer's and implementer's concerns, thus, significantly reducing development complexity. In practice, incremental delivery of system components over a long period of time usually also leads to requirements changes throughout the development life-cycle, thus, making the process somewhat similar to that of previously discussed Evolutionary development.

1.1.2.5 Spiral Model

In the *spiral model* [22] software evolves through the numerous versions, each having its own cycle of developmental events. In such a model, requirements engineering is also iterative, incremental and inter-woven with prototyping, planning and risk assessment. Continuous quality appraisal, even at the earliest stages of requirement development, forces developers to take measures advancing requirements verification and validation, and leading to the provision of mechanisms to improve the requirements engineering process.

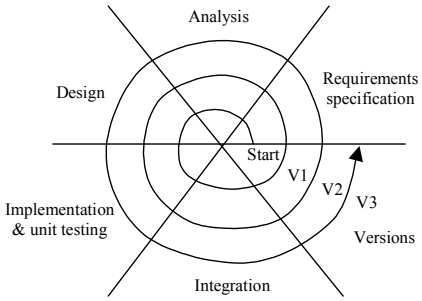

Figure 5: Spiral model (simplified)

1.1.2.6 The Impact of Process Models on Requirements Engineering

Considering the five different process models discussed so far, it is evident that requirements engineering is not a task that could be easily identified, characterised or described. Firstly, it is not possible to firmly allocate a time interval to its processing. In the traditional waterfall model, requirements engineering is located at the very beginning of the development process. However, the remaining models spread their requirements engineering activities along the entire life cycle. Secondly, requirements engineering cannot be viewed as a single cohesive block of activities. In the prototyping model, it is split into two tasks, but in the evolutionary, incremental and spiral models, it is fragmented into many smaller processes. The possibility of mixing, shuffling, splitting and merging of requirements engineering activities in software development can, thus, lead to many variations of the requirements engineering process.

What transpires, therefore, is that the very nature of the requirements engineering task is not in its temporal inter-relationship with other phases of the development life cycle, but rather in the intra-relationships of the activities that make up the requirements engineering process. We shall, hence, study these activities in the next section.

1.1.3 Requirements Engineering Activities

Despite large research efforts undertaken to date, requirements engineering activities are generally not well understood. Indeed the early process models, such as the waterfall model, postulated that requirements engineering could be performed in a single, monolithic step. As it can be seen from the previous discussion, the more recent models suggest that requirements engineering should be conducted iteratively and be structured into several inter-connected activities. These activities typically include elaboration of needs and objectives, requirements acquistions, requirements specification and modelling, generation and evaluation of alternatives, and requirements validation [99]. This cycle of requirements engineering events should also include two more activities, i.e. the initial planning phase and the completion phase, both equally important to the requirements engineering process, and both seldom mentioned as being part of that process. Let us discuss the requirements engineering activities in more detail in the following sections.

1.1.3.1 Setting the Requirements Management Plan

For the requirements engineering to be effective, its process must be well managed, its activities to be planned, the resulting products to be well structured, and the required skills and knowledge of participating developers anticipated and developed. The first phase in the requirements engineering process should, thus, be to define the framework for requirements processing and management. Such a framework can be defined as:

> "a systematic approach to identifying, organising, communicating and managing the changing requirements of a software application" [58].

Good requirements management practices provide more control over the project budget and schedule, improve the software quality, and ensure customer satisfaction [199, p 217]. The principal concerns of requirements management are [199, p 216]:

1. Managing requirements changes;
2. Managing various types of relationships between captured requirements;
3. Managing dependencies between requirements documents and other types of life-cycle work-products;

Other concerns include those related to the process and repository management, i.e.[3]

4. Managing requirements repository;
5. Managing customer-developer communication;
6. Managing customers' involvement, expectations, commitment and satisfaction;
7. Managing developers' readiness, expectations, skills and knowledge;
8. Managing requirements compliance with business objectives;
9. Managing compliance with standards, guidelines and procedures;
10. Managing the requirements process improvement;

Let us briefly describe each of these points.

Requirements Changes, Relationships and Dependencies. There are many different reasons for changing requirements, e.g. business needs may have been misunderstood, interview transcripts or observation logs may have been misinterpreted, or some business circumstances may have altered. As a result, existing requirements may need to be reworded, new requirements need be added, or some would have to be deleted. All changes together with the information about the change instigator and the reasons for the change have to be carefully recorded. Such details can be subsequently used to trace any development artefact back to the original statement of requirement or the business need that was behind its existence. At the same time, it should also be possible to find similar or related requirements, and to trace requirements forward to determine their refinement right down to software designs and code.

Requirements repository. Keeping track of a growing collection of constantly changing requirements is a complex and laborious task. Furthermore, requirements engineers must ensure that such a collection documents the viewpoints of the project stakeholders, it is cohesive and unambiguous, it has no conflicts or omissions. Of course, at the initial stages of any large software project, none of these conditions will hold - clients will not agree on the cohesive set of requirements, collected requirements will be ambiguous and in conflict and for the great part of the project duration will be mostly incomplete. Therefore, having a special-purpose and well-organised repository of software requirements is necessary to ensure the project success. Such a repository must be able to track requirements changes, be able to maintain different versions of changing requirements, should distinguish requirements characteristics, detect their inter- and intra-relationships, identify conflicts and have mechanisms to determine their completeness.

Customer-developer communication. As the software project evolves, the communication between customers and developers also undergoes significant changes as well. At the project inception, user requirements are conveyed to developers in a very informal fashion, perhaps as an unstructured list of business needs and customer's wishes, perhaps as legacy documents, memos, tenders and other business documents. Once requirements have been organised and structured into

---

[3] These concerns are recognised as important and suggested by the author of this paper.

a software requirements specification, they can be passed back to the customer for validation. Such specification text is later used as a source of information to produce diagrammatical or formal specifications that could be used by developers engaged in the software design and implementation. Subsequently, formal representation of software requirements may have to be communicated back to the customers for further validation. An effective requirements management system must be able to support different types of customer-developer communication, each at different levels of abstraction, structuredness and formality. Different forms of the same information need to be tightly related and be navigable to facilitate requirements traceability.

Customers Expectations. Different project stakeholders will have distinct levels of involvement in and commitment to the requirements engineering process. Each of the stakeholders will also have a different role in the project. Some will become the source of requirements information, some will supervise the requirements engineering process, some must be informed of the project status, and some will conduct requirements validation. Requirements management systems must be able to distinguish between different types of project stakeholders and to provide requirements services to these stakeholders in an appropriate form to meet their expectation. The system should support planning of requirements templates, entering new and modifying existing requirements, structuring and relating requirements, browsing, viewing, reviewing and validating requirements information at different level of their abstraction and formality. Also as the process is conducted incrementally, the system must also be able to report the progress of the requirements process in a piecemeal fashion to maintain the customer satisfaction.

Developers Expectations. Requirements are collected for the purpose of their subsequent implementation into a working computer system. Hence, the process of requirements management must serve not only the client but also the developer community.[4] Developers have to be able to restructure existing requirements to suit their development purpose. They will also be responsible for requirements analysis, correcting bad spellings and grammar, determining inconstancies, ambiguities, conflict and omissions. They will have to annotate problematic requirements, reconcile their conflicts, elaborate incomplete portions of requirements documents, formalise informal requirements statements into a more rigid expression of requirements specification. Again, as it is the case with the client stakeholders, the developer community is also greatly diversified. Different developer stakeholders will have different needs for requirements management and reporting, hence, the requirements management system must recognise these needs and provide information in the form most suitable for this type of requirements user.

Compliance with Business Objectives. As part of the requirements engineering framework the details of the requirements engineering process should be defined in the context of the organisation hosting the project. Such a process must take into consideration standards available to the organisation, its business processes and procedures, its strategic plans and long-term management objectives, available resources, its information and technological infrastructure. Currently used standards for the process quality and the software development life cycle will have impact on the requirements templates to be used in the software development project. structure, form and semantics of requirements engineering documents to be used in the project, and assessment of skills needed to conduct various activities and the allocation of development staff to perform them.

Compliance with Standards and Guidelines. The ultimate goal of the requirements engineering process is the production of a software requirements specification (SRS). SRS is commonly preceded by various other forms of requirements documentation used to capture and analyse the informal communication with users [228], these documents include interview transcripts, questionnaires, minutes, memos and the informal statement of user requirements. SRS, on the other hand, describes the future system properties in the more formal terms.

---

[4] The majority of requirements management systems support exclusively the developer only.

Although, such a document is commonly written in natural language, it is also frequently structured quite rigidly in compliance with one of many requirements specification standards [104, 160], recently also defined as part of the standards for development process life-cycle [105, 213]. To clarify individual requirements or their groups, the document commonly includes elements of the system model expressed in diagrammatic, tabular or formal notations. Numerous annotations and references to other types of documents provide further explanation of requirements. SRS is subsequently used as the means of communication between the clients and developers of the software system, and as a contract binding both parties to delivery and acceptance of the specific functionality and quality of the proposed software system.

1. Introduction
   - 1.1 Background
   - 1.2 Purpose
   - 1.3 Scope
   - 1.4 Definitions, Acronyms, and Abbreviations
   - 1.5 Document Overview
2. General Characteristics
   - 2.2 Product Perspective
   - 2.3 Product Functions
   - 2.4 User Characteristics
   - 2.5 Assumptions and Dependencies
3. Specific Requirements
   - 3.1 Functional Requirements
     - 3.1.1 Requirement 1
       - 3.1.1.1 Introduction
       - 3.1.1.2 Inputs
       - 3.1.1.3 Processing
       - 3.1.1.4 Outputs
     - 3.1.2 Requirement 2
     - …
   - 3.2 External Interface Requirements
     - 3.2.1 User Interfaces
     - 3.2.2 Hardware Interfaces
     - 3.2.3 Software Interfaces
     - 3.2.4 Communication Interfaces
   - 3.3 Performance Requirements
   - 3.4 Design Constraints
     - 3.4.1 Standards Compliance
     - 3.4.2 Hardware Limitations
     - …
   - 3.5 Attributes
     - 3.5.1 Security
     - 3.5.2 Safety
     - 3.5.3 Correctness
     - 3.5.4 Availability
     - 3.5.5 Reliability
     - 3.5.6 Efficiency
     - 3.5.7 Integrity
     - 3.5.8 Usability
     - 3.5.9 Maintainability
     - 3.5.10 Testability
     - 3.5.11 Flexibility
     - 3.5.12 Portability
     - 3.5.13 Reusability
     - 3.5.14 Interoperability
     - 3.5.15 Other Factors
   - 3.6 Other Requirements
     - 3.6.1 Database
     - 3.6.2 Operations
     - 3.6.3 Site Adaptation
     - …

Figure 6: Sample IEEE SRS template

Requirements template structures the SRS to include all the requirement types necessary for a given project or demanded by a specific development methodology. Such a structure is normally given by the document outline and by numerous guidelines on writing the SRS using the outline. The task of designing a good SRS template is very complex and time consuming, it is, hence, more common to adapt an existing requirements template, perhaps one of many that come with the requirements standard ued by the development organisation (see Figure 6). Many popular standards, e.g. those recommended by IEEE and DoD [104, 160], provide templates which are appropriate for either structured or object-oriented development, templates which focus either on the system function, its data, objects or their methods. Each template is also frequently accompanied by the guidelines and examples of their most effective use [141, 224].

Requirements identified in the process of requirements engineering, and documented in the SRS document, can be classified into four major classes [27]:

- *functional requirements* which specify a function that a system or a system component (i.e. software) must be capable of performing;

- *non-functional requirements* which state the characteristics of the system to be achieved that are not related to its functionality, i.e. its performance, reliability, security, maintainability, availability, accuracy, error-handling, capacity, types of users, changes to be accommodated, level of training support, etc.;
- *inverse requirements* which describe constraints on the system expressed in terms of what the system will not be able to do, e.g. in relation to software safety or security requirements; and
- *design and implementation constraints* which state the boundary conditions on how the required software is to be constructed and implemented.

Requirements Process Improvement. Finally, the process of requirements engineering itself, needs to be managed, monitored, measured, and reviewed to assure its high quality in the face of changing business and development conditions. The process may have to be adapted to the needs of individual customers. It may have to be altered to meet the standards and procedures already adopted by the client organisation. It may need to respond to the technological innovation, such as introduction of CASE or project management software. Finally, it may have to take into consideration changes to resource allocation, economic factors or organisational strategy.

1.1.3.2 Elaboration of Needs and Objectives

Once set in motion, the requirements engineering process starts with elaboration of needs and objectives for the software system as defined in terms of organisational context and the strategic management goals [219, ch 3 and 8]. Business problems and opportunities are identified and defined, project scope is outlined and constrained imposed on the proposed project. At this point, it is also common to determine the client initial need, though such needs are still subject to further deliberation and negotiation, as the feasibility study may be required to assess the viability of the proposed project.

1.1.3.3 Requirements Acquisition

Requirements acquisition aims at complete characterisation of the problem by eliciting all the necessary concepts from users and domain experts [42]. These concepts will provide building blocks for expressing the problem and specifying the software system. The first step towards acquisition of system and software requirements is in the identification of potential information sources, i.e. people, books and reports, memos, procedures, manuals, etc. Once the source of information has been determined, information gathering may commence. This is normally followed by information structuring, rationalisation, annotation and prioritisation.

There are many different methods of gathering, structuring, analysing, and interpreting requirements information. Goguen and Linde [84] summarise several such approaches to gathering requirements, i.e. via:

- introspection,
- interviews (questionnaire, open-ended, focus and application development groups, and discussion) [also in 96],
- protocol analysis, and
- discourse analysis.

Group techniques, simultaneously involving many project stakeholders is a particularly effective way of collecting information, assuring consistency, avoiding conflicts and achieving completeness of resulting requirements. Sometimes, such group techniques are termed as idea generation meetings [81, ch 10, 11, 157, pp 256-264], these include:

- brainstorming,
- mapping, brain-drawing and right-braining,

- nominal group technique,
- concensus decision making, and
- JAD [229], which is an especially successful approach to consensus decision making. Its success has been elevated by its adoption in Rapid Application Development [150] methodology, where it became the centre-post of the entire approach.

Computer-supported meetings have recently become a popular option for many organisations [212]. Simulation and prototyping is another possible way of collecting information from users [157, pp 281ff]. Methodologies based on the Personal Construct Psychology [194] have also resulted in several practical system for knowledge elicitation, e.g. WebGrid [77] and EnquireWithin [205] based on repertory grids, jKSImapper [71] and KMap [76] based on concept maps. Knowledge elicitation systems, used commonly in the early expert systems application, still find their way into requirements acquisition as well, systems such as KADS, AQUINAS, MORE, KBAM, LAPS or MEDCAT have been reported by Boose [23].

The process of eliciting information from customers, as manifested in the first two requirements engineering activities (elaboration of needs and objectives and requirements acquisition), may be further broken up into a number of distinct tasks [42]:

- source identification, i.e. determining *who* are the information providers and *where* is the additional information that could help clarifying requirements;
- information gathering through the use of multi-disciplinary views expressing *what* is to be built;
- rationalisation of collected information to determine *why* certain facts have been specified;
- prioritisation of all requirements to determine *when* they have to be addressed in relation to each other; and finally,
- integration of requirements with a view to combine different view points.

1.1.3.4 Requirements Specification and Modelling

Requirements modelling is necessary to organise and record all the collected information in a precise and understandable document referred to as a system specification. The specification is written using an appropriate formal or semi-formal notation, and it clearly describes all the system services.

In general, there are many different types of specification documents written at various stages of software development, by different project stakeholders, and at different level of abstraction and detail. Some of these documents are appropriate for specifying aspects of the problem domain, others aim at providing a description of the problem or its solution. Each type of specification documents, however, serves the common purpose though, to explicitly state the needs of a certain group of project stakeholders, whether those in the client or development community. In general, regardless of the notation used in their expression, their level of abstractness, or their role in the development process, the following requirements need to be identified:

- Behavioural requirements describing the required system function;
- Non-behavioural requirements specifying the system performance, accessibility, reliability, usability, etc.;
- Design constraints that consider any design issues imposed by the client on the system that have to be considered during requirements specification;

1.1.3.5 Generation and Evaluation of Alternatives

In the process of generation and evaluation of alternatives, the system specification is used to generate alternative software specifications subsequently handed over to designers for the construction of a software system [222, ch 5]. Such alternatives may consider varying project goals, they may anticipate the need for future changes or variations, they may provide alternatives project evaluation and give a range of possible selections for the system function, use and deployment.

1.1.3.6 Requirements Verification and Validation

During requirements validation the specification documents are analysed for their compliance with the original client requirements, the existence of inconsistencies, ambiguities or omissions.

When compared with the rest of the development life cycle, the cost and duration of the requirements engineering activities is very small - only 6% of the total [154, p 60]. At the same time, the number of defects introduced during this developmental phase is incredibly high - it accounts for 56% of all software errors [54, p 3]. Leaving requirements errors undetected until later development stages is incalculable - compared with early detection, the cost increases up to 25 times [153, p 26]. This is particularly irresponsible when the removal of requirements errors at their collection is not only possible but also practical. Verification and validation of software requirements is, hence, one of the most important activities in the process of software development. Methods, techniques and tools that contribute to more effective removal of software defects at the time of requirements engineering increase the overall software quality and dramatically reduce development costs.

There are many methods and techniques to assist requirements engineers to facilitate the validation process. These techniques may include active critiquing of collected requirements and their form [147], use of ambiguity metrics, technical reviews, measuring satisfaction, constructing test cases, studying existing products and reconciling requirements conflicts [81].

Such approaches can detect many of the common encountered problems in requirements documents [101], e.g.

- Bad assumptions;
- Inclusion of design details;
- Describing operations instead of requirements;
- Using incorrect terms;
- Using incorrect sentence structure or bad grammar;
- Missing requirements;
- Over-specifying, etc.

When dealing with requirements ambiguities, it is often useful to adopt a model of design that is based on the principle of ambiguity removal [81, ch. 19]. The authors of the method, Gause and Weinberg, suggest that the level of ambiguity in the project indicates the amount of design work still remained to be completed. To deal with development ambiguity Gause and Weinberg propose to measure it and then monitor one of its indicators with a view to refine the development processes and the work-products they produce. The authors suggest that a possible approach to judging levels of ambiguity is to investigate diversity of interpretation. This could be achieved by polling a group of professional systems designers and asking them to estimate the value of a single measurable entity associated with the development product.

Experimental data presented by Gause and Weinberg suggests that such a simple technique as an ambiguity poll provides a good measure of the requirements ambiguity and can also be used as the basis for estimating the project and resource cost.

1.1.3.7 Ending Requirements Engineering Process

The final activity in the process of requirements engineering is the completion phase. Sometimes the final termination of the process may be very difficult, as the developers or the clients may be tempted to aim for perfection, they may wish to renegotiate the project goals, extend the project scope, add the system functionality and improve it usability. It has been suggested that all stakeholders should develop courage to be inadequate [81, ch 25] from the project outset, and from the beginning set the targets for the system function and its quality.

Requirements engineering activities aim at the acquisition, analysis, modelling, verification and validation of system requirements. As the requirements processing can occur throughout the development life-cycle, special care must be taken when planning and monitoring of the requirements engineering process. The nature of the investigated problem, organisational context and the profile of the user community will play a central role in this planning activity. One of the factors commonly emphasises as the most important in the engineering process is the effective communication between the client and developer community, which is the main subject of the next section.

1.2 Communication of Requirements

As the elicitation process precedes all of requirements engineering phases, its effectiveness and efficiency is deemed to be of pivotal importance to the whole cycle. There are many factors that influence the efficacy of the requirements elicitation process, though, factors regarded as decisive to the requirements elicitation success include [45].

- *Successful communication of requirements* - the main source of communication problems amongst project stakeholders are terminological differences, introduction of ambiguities through the use of natural language, lack of formality and rigour, and the inherent complexity of requirements [42]. Christel and Kang [42] report that 56% of errors in installed systems were due to poor communication between various stakeholders and that these were the most expensive errors to correct using up to 82% of development time.
- *Ability to trace requirements* - traceability is considered difficult because of the complexity and frequency of software revisions. It was also reported that majority of currently used techniques for requirements capture and their representation focus on information modelling using formal notations, thus, ignoring the form the requirements were initially communicated in. In the process the original requirements are obscured and the full traceability is, hence, sacrificed [228]. This inability to locate and access the source of original requirements is one of the most commonly cited problems of requirements engineering [87].
- *Well defined elicitation process* - elicitation processes are hard to control due to inadequately defined business processes, organisational procedures and the poorly understood relationships between people involved, their responsibilities and the tasks performed. Streamlining and integrating elicitation steps can be accomplished by building tools to improve the capture and organisation of information in meetings, assisting in the negotiation process, and support in linking, traceability, and evolution of requirements. Few of the early commercially available software tools [185] addressed the issue of requirements elicitation at all. Only some of the more recent requirements management systems are giving due consideration to the need of representing and organisation of informal requirements CASSETS, CORE, Cradle/SEE, RDD, RTM, RequisitePRO, SLATE or Xtie RT [116] (discussed in section 1.5).

1.2.1 Formal vs. Informal Requirements

This research was, therefore, undertaken to address the issue of requirements communication, traceability and process understanding by means of tools supporting informal requirements texts.

Some of the above-mentioned problems and complexities are commonly blamed on the use of "informal" notations to record software requirements. However, in spite of recent achievements in the area of formal specification methods and CASE, the communication barriers still separate developers using precise notation and users preferring the use of natural language. The two viewpoints clearly provide a bone of contention between a number of factors (see Table 2).

Table 2: The formal and the informal views of requirements

Informal	Formal
Requirements are normally elicited from user's informal descriptions in natural language.	Requirement are subsequently specified in an appropriate formal notation by developers.
Clients have incomplete and inconsistent perception of their needs.	Developers tend to close the specification in a harmonious, rigid and comprehensive form.
With the increased understanding of the problem area, clients change their requirements.	Maintenance of formal requirements is high. Hence, developers prefer to freeze their requirements once in a cohesive and complete form.
Clients usually have low-level of computer literacy and thus prefer expressing their requirements using concepts drawn from the problem area.	Developers tend to be unfamiliar with business terminology and prefer to express requirements in computer terms.
Client organisation commonly has a number of different stakeholders in the project, all having multi-faceted and contradictory views of requirements.	Formal and semi-formal requirements notations used by developers are usually limited to express a single view of customer requirements.
The requirements validation process against clients' needs is more effective when requirements are expressed in the informally.	Requirements verification for their consistency, correctness and completeness is more effective when requirements have formal semantics.

1.2.2 Problems with Informal Requirements

The above-mentioned problems and complexities, evidently, find their source in the human factors of the requirements engineering process and the informal nature of pre-specification form of requirements documents, i.e. the media used to record them, their loose structure and inappropriate notation.

- *Media.* Initial requirements come in variety of media types, to include free text, graphics, image, video and animation, speech, sound, and other sign systems [162]. They are rarely in a computer-readable form, and when such a computer representation is available, its structure and rigour is hardly ever yielding to further automation, formalisation and integration.
- *Organisation.* Documents produced at such early stages of software development frequently lack formal structure and logical organisation, they may include minutes of meetings, logs of email discussions, interview transcripts, statements of work, requests for proposals, operational concept documents, technical notes, manuals, and technological surveys [228].
- *Notation.* Early requirements are commonly recorded in natural language, the notation severely criticised as suffering from noise due to redundancy, silence due to omission, over-specification due to solution details, contradiction due to incompatibility, ambiguity due to multiple interpretations, forward reference to concepts introduced later, or wishful thinking due to the lack of realistic validation [159].

Requirements specification documents, being the basis of further development, are commonly called for to be in a precise, technical and symbolic notation. A host of commercially accepted diagramming techniques are currently in use for just this purpose [185]. Formal specification languages and methods, based on the sound mathematical foundation, have a further advantage over the informal approaches to capturing software requirements. Such formal tools have been successfully used in Ada development [136, e.g. Clear], software requirements specification [90, e.g. CML and Telos, CIM, GIST, Taxis/RML, ERAE and KAOS], and domain analysis [201, e.g. CIP, OBJ and Z]. Formal description of entire problem domains also allows for the creation of formal development frameworks, which may lead to additional cost and productivity gains [80].

We believe that all these problems find their source in the lack of the formality of textual requirements documents. We suggest therefore that the problems could be addressed when we could effectively bridge the formality gap that exists between informal requirements and their formal requirements specifications.

1.2.3 Benefits of Informal Requirements

Unfortunately, the issue of requirements capture is not as simple as the notational convenience, rigidity of the elicitation process, or the completeness and cohesion of the collected requirements. Informal methods of expressing requirements may actually bring many (sometimes unexpected) benefits to the entire process of requirements engineering.

As some of the practising requirements engineers and researchers note, informal notations are frequently the only form of requirements expression acceptable to the end users [57, 103]. While it is desirable to specify requirements formally at some stage of requirements engineering, they rarely can be expected to learn and use effectively complex formalisms [45]. Also, since the early stages of requirements acquisition are commonly the subject of negotiation between software stakeholders, thus, initial user requirements is necessarily vague, ambiguous, incomplete and inconsistent, in such a situation, formal methods can only be successful when used after the consensus on the requirements has been reached [42].

More research should be conducted to provide requirements engineers with some tools and guiding principles of improving natural-language documents without the introduction of unnecessarily formal models [103]. Alan Davis, the author and prominet Industry spokesman, makes this profound observation:

> *"When I wrote Software Requirements almost a decade ago, the world thought that requirements management was equivalent to model building. This has proven incorrect. Requirement management is about people, about communications, and about attempting to understand before being understood. I guess I hold little hope in the near term for researchers in requirements management, who continue to solve the simple (that is, technical) parts of hard problems. Over my 20+ years of requirements management consulting, I have seen a strong movement away from having no requirements documents and from formal requirements modelling, and toward communicating with customers, writing requirements in natural language, and storing requirements in a database where they can be annotated, traced, sorted and filtered. I have more hope for practitioners."* [57].

In view of the need for the effective communication between the customers and developers, it is recognised that collecting informal software requirements in natural language will remain in use for the foreseeable future. During the 1989 Workshop on Requirements Engineering and Rapid Prototyping (held by the U.S. Army Communications-Electronics Command Center), the participants debated this very issue. In their conclusions [45, p 41], they stated that the specification language should serve the requirements engineering process rather than driving it. They noted that whilst it is desirable to use formal expression of requirements in the later stages of their processing, it is also critical that the actual users define the requirements. As it is not realistic to expect the users to learn complex formalisms, hence, the workshop participants pointed out, customers should use the notation they are most familiar with, i.e. natural language.

Natural language will, hence, be used as the primary means of verbal communication between stakeholders, and will also be used as the means of recording requirements in textual requirements specification documents.

1.3 Dealing with Requirements / Specification Gap

The dichotomy between user requirements and their specifications fosters a serious conflict of informality and rigour in the capture and specification of software requirements. Resolving this conflict has been a theme of a number of research projects that resulted in several different solutions to the problem (See section 1.4). However, in spite many technological breakthroughs in bridging the gap between the formal and informal in requirements engineering, Alan Davis, a prominent spokesman for the software engineering industry is highly sceptical about the technical advances to date, and more hopeful about the solutions that put emphasis on the human aspect of the requirements engineering problems [57]. In our view, the problem of requirements formality can be stated succinctly as follows :-

The specification vehicle should be informal enough to allow an untrained customer to understand what system functions will be delivered upon the system completion, and sufficiently formal to allow a system designer to have unambiguous statement of customer requirements that can be implemented and validated in its widest sense.

The following section explores and discusses both technological and people-oriented methods of requirements management in some more detail.

1.4 Requirements Engineering Methods

There are many different methods of conducting requirements engineering activities (see Figure 7). Some of these methods are applicable to identifying business requirements, others are concerned with only certain area of business, the majority of approaches, however, are designed to deal with the computer systems and their software components - the main focus of this research and this section in particular.

Traditionally, requirements processing focused on the issue of a system function and its data. The more recent approaches also emphasise other concerns related to requirements management. They investigate the role of one-to-one and group communication, stakeholder viewpoints, use of scenarios, reliance on decision support and issue tracking, knowledge management and finally the overall requirements engineering framework.

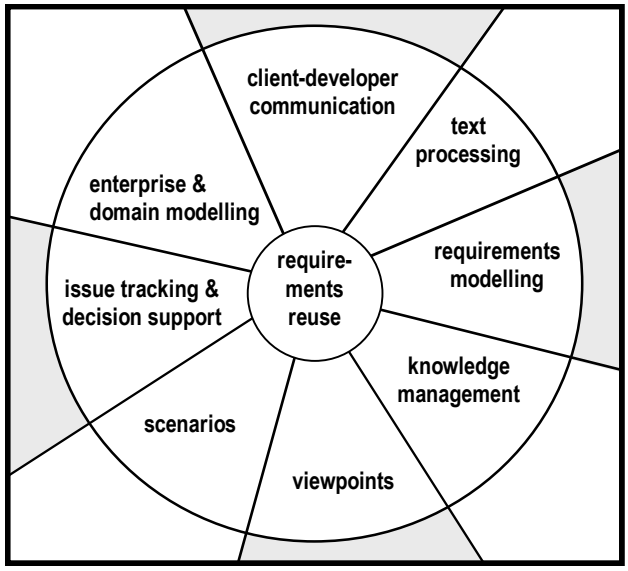

Figure 7: Bridging the formal and informal in software requirements

1.4.1 Management of Verbal Communication

The most commonly practised approach to reconciling differing views of developers and customers is to rely on requirements engineers to use their communication,

analytical and technical skills used during requirements collection, when analysing requirements, and then mapping them into more rigorous specification documents. In the process, requirements engineers single-handedly identify inaccurate and imprecise requirements, determine and resolve some of the inconsistencies, reconcile conflicting stakeholder views, and highlight some of the missing requirements. Completed specification is then carefully studied and validated by customers. Any detected problems are subsequently eradicated by iterating the process of requirements capture, analysis and validation to the full satisfaction of all participating parties.

The sheer effort of communicating and interpreting requirements could be greatly simplified with the use of a system prototype or a demonstration of its single aspect, either function, performance or user interface related. The system's working model could (should) be developed and validated with the active participation of its prospective users [214]. Prototypes improve the understanding of the problem by both the users and developers, they permit exploration of possible system behaviour, and reduce development risks by allowing developers to early focus on the feasibility analysis, identification and elimination of candidate requirements [103].

In a similar manner, customer-developer communication can be significantly improved by utilising visualisation tools, which are capable of modelling and representing problems in a graphical form [6, 38, 117, 164]. The main value of such tools is in the environment in which the process of identifying and describing the needs and concerns could be achieved by means of direct and intuitive manipulation of visually familiar, i.e. real-life like, concepts and objects [145]. Visualisation approaches to software development use quite creative technologies drawn from animation [109], multimedia [172, 228], theatre-like scenarios and visual demonstrations [6]. When combined with prototyping or generative software, visualisation tools empower the end users to contribute to the system development across its entire life-cycle, from requirements, through design, to its testing.

In some traditional methodologies, analysts confront their customers with a mental simulation of the system execution by "walking" the customers "through" the main functions of the "paper" system. Walkthroughs allow customers to gain better understanding of the planned system functionality and to validate requirements even without system prototypes or mock-up execution. The consensus on the system requirements can also be reached more readily [100]. In general, involving customers in joint development of all aspects of requirements planning and the subsequent software design benefits a software project immensely [43, 133, 150, 229].

The systematic client-developer collaboration on software requirements planning and the subsequent design is possible with the use of RAD (Rapid Application Development) meetings [150]. Known also as JAD (Joint Application Development) [229], the meetings provide a venue for mediated co-operation and negotiation between representatives of both client and developer communities. Such meetings provide an opportunity of instantaneous feedback, query and correction and force all participants of the requirements engineering process to find a common line of communication. Due to the complexity of technical or organisational contents of software requirements, whether communicated on one-to-one basis or in teams, they may require employment of special-purpose tools acting as communication agents [83], group support systems [170], or issue tracking systems [47]. Such tool-assisted "mediators" monitor and report user and developer opinions and expectations, facilitate collection and prioritisation of issues and concerns, and provide a platform for constructive negotiation and collaboration, and elimination of tension and conflict in the project.

1.4.2 Text Management

Tremendous amount of research effort was spent on improving written communication in software development. In this area of research, many favour development of early life-cycle documents in plain natural language. The earliest applications of natural language processing systems were in automatic programming from English descriptions [11, 13, 95] and in analysis of problem statements [1, 17]. Later, systems and methods were created to help processing

diagram annotation [18] and finding abstractions in problem descriptions [4, 86, 144]. More recent work includes automatic generation of formal specifications [211], requirements definition and specification reuse [165, 166, 186]. Outside software development area, natural language processing also found many useful applications, e.g. teaching, banking, navy, news filtration, or patent searching [66].

Use of controlled natural language, i.e. language of a limited vocabulary and simplified grammar, has certain advantages over the richness of natural language expression [125], e.g. it reduces ambiguity, limits the number of words used in technical writing, it emphasises good writing techniques, and it simplifies sentences and paragraph structure. Examples of commercial applications of controlled language include VLSI design [52], workcard control [40], authoring [158], writing technical documents [189, 227], controlling cross-border police communication [110], or producing customer and technical services [88].

With the use of a controlled natural language, requirements statements could be readily analysed, structured, indexed and classified [4, 13, 41, 53], or translated into formal, possibly even executable, statements of software specification [75, 165, 192, 230].

As an option differing from the natural language encoding of requirements, some researchers suggest the use of a formal notation for the requirements capture [90, 159, 188, 225]. Contrary to the common belief, the proponents of formal methods, provide ample evidence that the end users can in fact be trained and become proficient in the use of such specification formalism [25, 93]. Others offer techniques, methods and tools that allow the users to validate requirements by casting their formal specification into natural language [67, 113, 187]. Yet another alternative is to maintain a detailed record of the refinement and elaboration process leading from the informal documents into their more rigid form [36, 186]. Such a process may preserve the links between informal and formal concepts for the traceability sake, thus, creating a complete, cohesive and navigable requirements specification system [59, 118, 228].

Management of textual documents is a demanding process in its own right. Here, people suggest the use of standards and methodologies [19], version control and configuration management [209], or document visualisation and management techniques [33, 70, 78, 79, 102]. Hypertext and hyper-navigation between the life-cycle documents plays an especially important role in document management [10, 32, 35, 46, 92, 168]. In particular, it proved useful in managing [5, 34, 64, 179] and acquisition [156, 218] of knowledge – so useful in the process of requirements processing. Elements of hypertext were incorporated in various software design tools and CASE [20, 51, 60, 79, 124, 180], systems tracking design deliberations [47], requirements discussion and evolution [173]. Experimental systems have also been developed to use hypertext in support of requirements engineering meetings [120, 228] and of managing requirements documents and their related processes [118].

1.4.3 Model Management

During requirements acquisition or their analysis and further elaboration, it is commonly recommended to create a formal or semi-formal model of the future system, its data, structure and behaviour [56, p 215ff]. Such a model is a concise and complete description and an abstract representation of the system [107, pp 120ff]. Davis [56] and Wieringa [222] provide a comprehensive review of various modelling techniques, the artefacts they produce, and the methodologies that define development processes governing the construction of these artefacts.

The area of requirements modelling has been thoroughly discussed elsewhere in several software engineering or systems analysis and design textbooks [98, 175, 197, 221]. Such popular texts already provide detailed description of techniques useful in specifying systems function, data flows and structures, control, states or decisions, as used in both structured and

object-oriented methodologies. Hence, these modelling techniques and artefacts shall not be reviewed in this paper.

Requirements models, formalised in either a diagrammatical or logical terms, are useful in the analysis of the system behaviour, checking the system consistency, its correctness or completeness.

1.4.4 Viewpoints and Their Management

Software requirements are commonly collected from several project stakeholders. Each stakeholder has a limited view of the entire software project, and each of the stakeholders may have requirements conflicting with those of others. The majority of requirements engineering methods is capable of capturing and modelling a single requirements viewpoint. Such approaches ignore the existence of differing viewpoints and necessarily rely on the analyst's skills to identify possible conflicts that may exist between the multiple parties interested in the software project and to subsequently negotiate a single common view of system requirements. An alternative solution to the problem of multiple viewpoints in software development is offered by a small number of the viewpoint-oriented methods of requirements engineering [129, Ch 7]. Under the viewpoint-oriented regime of requirements processing, multiple user views, whether incomplete or conflicting, can co-exist in the software life-cycle, and hence, be gathered, modelled, analysed, integrated and reconciled to ensure the satisfaction of all participating parties, and to warrant the consistency of the resulting system.

The first attempt at the explicit viewpoint modelling was the CORE (COntrolled Requirements Expression) method, developed for British Aerospace by System Designers [163]. CORE prescribes a number of methodological steps:

1. The analyst, together with the customer, usually starts by establishing the system boundary, which is then considered a global project viewpoint.
2. At each iteration, viewpoints are decomposed into a number of sub-viewpoints, which initially begin with the viewpoints held by the project stakeholders, i.e. the system owner, users, operators, suppliers, maintenance people, etc. The hierarchy of all viewpoints is commonly represented in a *viewpoint structure chart*.
3. Viewpoints need to be checked for consistency within each decomposition level and with the levels above. Such analysis is conducted with the use of a *tabular collection table*. For each viewpoint, the table identifies its associated actions, the data used for these actions, and the source and destination of the data. The source and destinations are checked for consistency and completeness across all viewpoints.
4. All viewpoints are finally combined into a cohesive and consistent framework of requirements. This is commonly done with the use of data flow diagrams, which summarise all viewpoint actions and the flows of data between them.

Various diagrammatic notations and tools [e.g. SAFRA - 163] have been proposed to visualise and automate the CORE process.

Finkelstein *et al.* [68, 171] and Jackson et al. [128, 198] extended and formalised CORE into VOSE and VORD methods respectively. As in CORE, the two methods structure viewpoints hierarchically and produce collection-like tables specifying the processing actions. At the same time they add viewpoint representational, process and specification information. Each viewpoint represents some role of a domain participant in the system behaviour - this way the viewpoints are distributed among (or attributed to) individuals responsible for specific tasks, working in a particular environment and having unique knowledge. Allocation of viewpoints to domain participants or agents also allows structuring of viewpoints according to the agent taxonomy. They add rules capable of automatic checking of the viewpoint representation for some signs of their potential incorrectness or inconsistency. Leite [134] also proposed the model extension

with the capability to attest to the model validity. Mannon, Keepence and Harper [148] put forward an idea in their VODRD method that viewpoints can be used to more effectively facilitate reuse of requirements through the domain model.

Other less sophisticated techniques are used in practice to record and analyse user viewpoints. These include a simple tabular technique used as part of the Andersen Consulting Foundation Methods [8], i.e. *user context table* and *user voice table*. User context table is used for to sort out observations made by an analyst, and to understand what the business is doing currently, who is responsible for its completion and why, and to generate new, proposed business needs for review by the user. User voice table is used to sort out the statements made by the user by functional analyst, to determine business needs of each user, to understand the semantics of user statements, to compare and reconcile these needs.

1.4.5 Scenario Management

When eliciting requirements from the end users, it's been observed that they find it easier to relate to real-life examples of system use rather than to abstract descriptions of the system functions [129, p 64]. Such examples can be given in terms of scenarios, which describe specific cases of user interaction with the system. Because of the specificity of scenarios, they cannot be considered as a representation of requirements themselves, however, they are considered as useful in better understanding of such requirements. There are many different forms of scenarios, e.g. action table scenarios [223, 226], use case scenarios [108], inquiry cycle scenarios [173], etc. Although the approaches to scenarios have significant differences, they also share many similarities in their structure and contents [129, p 64], e.g. they include :-

1) Description of the situation before the system entered a particular scenario;
2) A typical flow of events in the scenario;
3) Exceptions to the normal flow of events;
4) Information about activities performed concurrently to the scenario;
5) Description of the situation after the scenario was completed.

Weidenhaup, Pohl, Jarke and Haumer [217] conducted a detailed comparison of different scenario-based methods. They found that in practice scenarios have many different uses, i.e. scenarios :-

- make abstract models more concrete,
- in some cases replace abstract models all together;
- enforce interdisciplinary learning;
- complement prototype-based development;
- reduce developmental complexity;
- facilitate partial agreement and consistency;
- cross-reference glossary information;
- help in better understanding of static models.

One of the most influential work on scenarios-based requirements processing was presented by Colin Potts, Kenji Takahashi and Annie Anton [173]. In their Inquiry Cycle model, scenarios are used to document requirements. They represent sequence of actions and they are structured into a hierarchy of scenario classes. Common actions shared between scenarios are defined via episodes, which are defined in terms of fine-grain actions. The most important element of the Inquiry Cycle model, is its ability to use scenarios as the basis for requirements discussion, to raise questions

about fragments of requirements,[5] to answer these questions, and to instigate requests for requirements changes. Potts, Takahashi and Anton developed a hypertext tool that manages both scenarios and discussions, the tools allow users and developers to navigate and find requirements, to find change requests and their rationale, trace requirements, and finally to plan and monitor the analysis process

Although, scenarios are well suite the process of requirements elicitation, there still more research that needs to be conducted to answer, e.g. into scenario integration with other techniques, their management, traceabiliy and process guidance [217].

1.4.6 Issue and Decision Management

Requirements elicitation and analysis are two very complex processes, the processes which require very involved cycle of reasoning and negotiation with the user, and the processes which call for frequent group decision making. Tracking the issues of prime concern to the user and keeping a record of decisions made in the process frequently requires the support of computer-aided methods and software tools.

IBIS (Issue-Based Information System), developed by Group Decision Support Systems [91], is a general-purpose approach to assist management of any organisation in conducting successful meetings. Its main philosophy is to detect and deal with the so-called "wicked" problems [48]. The problems are considered "wicked" when they do not yield to the traditional "scientific" approach to problem solving, which is to gather and analyse the data, and then formulate and implement the problem solution. Such problems lead to many meetings, constant arguments and confrontation, lack of focus, creation of many new sub-problems, involving increasing number of stakeholders, all of which do not seem to lead to any viable solution. To cope with this situation, Conklin suggested an approach that avoids confrontation but turning every serious argument and disagreement into an issue to be further investigated, confronted and resolved. IBIS helps management to keep track of such issues, their discussions and their solutions. The method is supported by the graphical tool, QuestMap,[6] which allows developing issue maps, and which helps resolving wicked problems by visual interaction with such maps. Interestingly, one of the first application of IBIS method was in assisting software development teams to controls their design deliberations, the primary focus of requirements specification and refinement. The application was further supported with gIBIS software [47].

Issue tracking and discussion is effectively a kind of decision making. An alternative approach to dealing with such decision making process, especially during requirements engineering, is to utilise special-purpose decision support tools, such as GroupSystems from Ventana.[7] GroupSystems is a suite of team-based decision software tools helping groups, such as requirements planning teams, to reach decisions. The suite consists of standard tools, survey and alternative analysis and activity modelling tools. The system allows you to create and maintain the roster of people, make notes, illustrate discussion points, or annotate an image via a whiteboard, distribute reference materials for group viewing such as a report, agenda, or spreadsheet file, gauging the opinions of a group, etc.

Automated group decision systems provide many advantages over the traditional (non-automated) methods of decision making [170], e.g. increased group participation and synergy, automation of record keeping, better structure of the meeting, higher satisfaction, more knowledge and better quality decision making. The main disadvantage is slow communication and not all decision tasks are amenable to the tools available at the meeting. In case of requirements

---

5 Such as what-is, how-to, who, what-kinds-of, what-relationship, what-if and follow-on questions.

6 More information about QuestMap can be found at URL: http://www.gdss.com/OM.htm.

7 See GroupSystems web site at URL: http://www.groupsystems.com.

management, it seems that automation and support of problem solving and issue deliberation is an important factor that needs to be considered in the planning of the requirements engineering process and in the design of tools to support it.

1.4.7 Knowledge Management

Some of the software providing assistance in requirements and design employ knowledge acquisition, representation, and expert systems techniques to extract customer's knowledge into a sophisticated knowledge base [24, 114, 126, 177, 181, 231]. Such a knowledge base can subsequently be reasoned about, processed, evaluated and formalised into a set of deliverable requirements specifications.

Tamai [206] believes that the main application of knowledge-based system to software development should be centred around requirements processing, i.e.

- application of knowledge-based system to modelling business domains via analysis of business procedures and manuals;
- problem analysis guidance system by building a knowledge-based system from above mentioned texts; and,
- guidance system for re-using, software components at the earliest development phases.

He also believes that most of the future effort should be put into:

- assisting requirement analysis and design; and,
- knowledge acquisition throughout the development.

Lowry [139, 140] sets a number of additional goals for broader knowledge-based software engineering (KBSE), i.e.

- to formalise artefacts of software development and the software engineering activities that produce these artefacts (e.g. formal specs allow stating requirement precisely and unambiguously);
- to use knowledge representation technology to record, organise, and retrieve the knowledge behind the design decisions that result in a software system (this will aid debugging and maintenance);
- to produce knowledge-based assistants to synthesise and validate source code from formal specifications (this will improve maintenance by replying synthesis process over modified specs);
- to produce knowledge-based assistants to develop and validate specifications (helps in resolving conflicts in user requirements, refine incomplete or informal requirements, and use domain knowledge in developing system designs); and,
- to produce knowledge-based assistants to manage large software projects.

Lowry also believes that a knowledge-based specification acquisition system should include the following components:

- domain models which serve as knowledge bases for expert systems that assist users in specification acquisition, it may also specify some information needed for program synthesis, domain models should represent static (ER) and dynamic aspects of a problem domain and its integrity constraints;
- requirements and specifications usually based on domain models, portions of previously defined specs could be reused, assistance in domain knowledge could be provided, and some systems could actually generate formal, consistent specs from vague and informal ones;

- design specifications may be constructed under the supervision of an expert system which supports use of domain knowledge, reuse of existing design knowledge, intelligent retrieval of designs and their modification, explain evolving specs, etc.

In the past there have been many attempts to improve the software development process with the use of intelligent systems, whether it be knowledge bases, expert systems or knowledge acquisition techniques. Table 3 summarises the most important approaches grouped into four categories, i.e.

1) those which use knowledge acquisition techniques to improve the process of communication with clients;
2) those which rely on the rich representation of knowledge that allows reasoning about information collected from domain experts;
3) knowledge-based software engineering which allows knowledge-intensive support of software development life-cycle; and finally,
4) those which focus on knowledge-based requirements engineering.

Table 3: Knowledge-Based Approaches to Requirements and Software Development

	Issue	References
Knowledge Acquisition (KA)	Review of knowledge acquisition	McGraw & Harbison-Briggs [157]
	Knowledge acquisition approach to design (of VLSI)	Stefik & Conway [203]
	Assistant to construct a computational domain model	Shoen [190]
	Acquisition of software design knowledge	Seppanen & Heikkinen [193]
	Review of knowledge elicitation techniques	Cordingley [49]
	Precision KA for Expert Systems	Naughton [167]
	KA by matching new with old requirements	Bennett[16]
	KA for requirements specification	Kinoshita [126]
	Process of design acquisition	Gero [82]
	KA for extending domain models for designs	Klinker, et al. [127]
Knowledge Representation	Overview of knowledge representation	Brachman & Levesque [26]
	Review of Expert Systems	Buchanan & Smith [30]
	Expert system applications to software development	Frenkel [74]
	Use of Conceptual Graphs in systems development	Sowa & Way [200]
	Design as a knowledge-based state-space search	Steier [204]
	Automatic programming with domain-knowledge	Barstow [14]
	Knowledge-based programming and design	Waters, et al. [85, 183, 215, 216]
	Domain mapping in design	Chandrasekaran [37]
	Knowledge-based software information system	Devanbu, et al. [61]
	Active assistance in domain modelling	Schoen, et al. [191]
Knowledge-Based Software Engineering	The impact of domain knowledge on design processes	Adelson & Soloway [2]
	Use of knowledge bases in software development	Ambras & O'Day [7]
	Impact of AI on software development	Brastow, Balzer, et. al. [12, 15]
	Knowledge-based design	Bradshaw & Young [28]
	Knowledge-based systems factory	Eliot & Scacchi [65]
	KB communication in SE	Fisher & Schneider [69]
	Knowledge bases in software development	Harandi [94]
	CASE integration via conceptual repository	Jarke [109]
	Automatic program analysis	Johnson & Soloway [111]

Table 3: Knowledge-Based Approaches to Requirements and Software Development

	Issue	References
	Intelligence in software development	Kaiser, et al. [119]
	Knowledge bases for software re-engineering	Kozaczynski & Ning [131]
	Review of knowledge-based software engineering	McCartney & Lowry [139, 140, 152]
	Knowledge-based CASE	Puncello, et al. [177]
	Overview of AI applications in software engineering	Rich & Waters [184]
	Knowledge engineering in software development	Tamai [206]
	Expert systems vs. software engineering	Taniak & Yun [207]
	KB systems in software development life-cycle	White [220]
Knowledge-Based Requirements Processing	Domain knowledge and requirement traceability	Adhami, et al. [3]
	KB requirement specification and prototyping	Chen & Chou [39]
	Capturing world knowledge in requirements	Greenspan, et al. [89, 90]
	ES for extracting requirements from user scenarios	Hobbs & Gorman [97]
	Knowledge-based requirement specification	Johnson, et al. [112-114]
	Formal specifications with knowledge-based assistance	Ladkin, et al. [132]
	Knowledge-based requirements specification	Zeroual, et al. [137, 143, 169, 232]
	Knowledge-based support for design rationale	Ramesh & Dhar [178]
	Requirements apprentice	Reubenstein & Waters [181, 182]
	KB specification for expert systems	Slagle, et al. [196]
	Use of KB in specification transformation	Tsai & Ridge [210]
	Deduction in requirement processing	Zeroual [231]

Knowledge-based specification methodologies use a combination of conventional, reuse-based, experimental and evolutionary processes. The specifications produced by knowledge-based systems can be validated by their symbolic evaluation, paraphrasing the formal specifications into natural language, or static analysis of specifications for their consistency and completeness [139]. Specifications also need to be maintained, with the greatest emphasis having been put on tracking design decisions and maintaining systems dependencies between domain knowledge, requirements and design decisions.

1.4.8 Enterprise, Domain and Product Line Management

The requirements engineering process is usually set in the context of a larger business framework, its organisation and its processes. In a business system, requirements either aim to improve some business process, or they have to comply with certain business practices, or they address a specific business issue that affects the future development, operation, or utilisation of the system.

In a typical business scenario, it is the business information strategy that defines the organisational information infrastructure that sets the information technology policy, and lays the rules for any system to be developed in the organisation. A number of information system development methodologies, such as IE Information Strategy Planning (ISP) - [149], [222, Ch 6], [221], Information Systems Work and Analysis of Changes (ISAC) - [222, Ch 5], or Organisational Information Requirements Analysis (OIRA) - [219], address the issue of requirements engineering business context. They also specify various methods and techniques useful in the integration of business activities leading to the identification of business needs with the subsequent requirements elicitation and analysis.

Domain knowledge is another factor that influences the development of software requirements [176]. Domain engineering aims at the construction of a complete model of a

problem domain. A typical business system may simultaneously span several such domains. Subsequently, a domain model will allow requirements documents to utilise some of the pre-classified and formalised domain objects, their descriptions, properties, behaviour and their relationship, thus, reducing the possibility of miscommunication, incompleteness and inconsistency. DARE [73], FODA [121], FORM [123] and ODM [195] are four most prominent domain modelling methodologies, proven to reduce the effort of building system requirements and improving their quality due to the significant reuse of domain knowledge, domain objects and the associated software systems previously developed to service these domain objects. Similar gains in the requirements effort can also be observed in the reuse of the product-line information [44], also useful in the phase of requirements definition.

1.4.9 Requirements Reuse

Finally, the systematic reuse of requirements and domain-related information in a wide-spectrum artefact reuse [142] can accomplish several important upstream reuse objectives, e.g. selection of domain objects and associated software components during requirements specification, ablity to critique and analyse user requirements, detection of requirements incompleteness and filling in missing requirements details, and selection of an appropriate refinement of requirements into more detailed specifications and designs. Detailed discussion of this topic, however, is discussed elsewhere.

1.5 Requirements Processing Tools

The tools regarded as providing the most comprehensive services for requirements processing are requirements management systems, i.e. systems capable of providing an integrated environment for capturing, organising, communicating and managing the changing requirements of a software application [58]. In 1998, INCOSE (International Council on Systems Engineering) produced a comparison, but not a review, of the most important contenders to the title of requirements management systems [106]. The INCOSE report covered the following systems (in alphabetic order): Caliber, CORE, Cradle REQ, DOORS, icCONCEPT RTM, RDD-100, RDT, RequisitePRO, SLATE REquire, TOFS, Vital Link and XTieRT. For completeness reasons, we briefly describe the capability of each of the systems and we provide a further reference to the vendor information in the footnotes.

CALIBER (Technology Builders) is an Internet-based requirements management tool. It allows automatic importing and parsing of requirements from other documents (e.g. Microsoft Word), hierarchical organisation of these requirements, attaching external references (such as documents or their parts), classification and searching of requirements by their attributes, keeping track of requirements changes and extensive reporting.[8]

CORE (Vitech Corp.) is a system that supports team-based development of requirements. The system captures requirements using formal specification language RSL. It also supports several different diagramming techniques (ER, hierarchy, FFBD and N2). CORE helps requirements engineers in the identification and resolution of issues, interfaces and risks. It allows linking external documents with its models, it helps verifying formal specifications and provides traceability.[9]

Cradle REQ (3SL) is a component of an integrated project data management system Cradle. Cradle draws a clear distinction between source documents external to the project (produced by clients) and requirements produced under the control of the requirements management system. It views requirements management to consist of source document management, requirements capture and requirements engineering in the context of a formal (and traceable) configuration

---

[8] Description and demonstration are available at URL: http://www.tbi.com/products/caliber.html.

[9] Description and software are available at URL: http://www.vtcorp.com/.

management process. With Cradle REQ, requirements statements can be captured in a semi-automatic way, they can be cross-referenced, indexed, classified either automatically or by the analyst, they can be checked for completeness and for consistent terminology, searched and traced. Additional information can be linked to requirements in a form of rationale, notes and dependencies. Requirements information is never lost as it is never isolated from its original context of a source document. Cradle also provides facilities for modelling systems using variety of notations, i.e. functional, behavioural, architectural and object-oriented.[10]

DOORS (Quality Systems and Software) is an object-oriented requirements management system. It's been designed to manage, configure, search, retrieve and link requirements and design components. DOORS can interface with many existing textual and diagramming tools. Its main role is to manage, trace and navigate links between many types of data, in many forms, which is relevant to the requirements process. The system also features the production of inter-linked and cross-referenced requirement reports.[11]

icCONCEPT RTM (Integrated Chipware, Inc.) emphasises reusability of requirements from one successful project to another. It allows entering requirements using one of many standard word-processing software and relating these requirements to other life-cycle work-products that can be produced with one of the supported CASE and development tools. The system provides assistance not only with managing requirements but also with the entire projects based on the repository of requirements that must be traced, assigned, verified, allocated resources, and of which status needs be continually reported. Visual change management and cross-referencing allows easy management of requirements collections.[12]

RDD-100 (Ascent Logic Corporation) is essentially a repository of development information that could be stored, managed and retrieved during the development process. RDD-100 provides facilities to represent information about project data flows, data content, control flows, concurrency, resources usage, layers of recovery, and reliability. The system is capable of tracing all of the requirements from their informal statement down to the test derived from these requirements. RDD-100 allows requirements to be extended, refined and linked, and in the process their consistency and redundancy checked. The system is also capable of analysing the system feasibility based on resources and costs and considering the risks of system failures. The system reporting and query capability supports a variety of standard and custom formats.[13]

RDT (GEC Marconi Systems) was designed to capture the design process and manage the system requirements. The system looks at the requirements management process from the time of producing a project tender. Hence the system assist developers in the formulation of tender specifications and requirements, subsequent formulation of tender responses against specifications, traceability and management of requirements throughout the development cycle of a contract, definition of test procedures and their allocation to requirements, etc. RDT features automatic capture of requirements from existing text documents, their parsing for document structure, requirements paragraphs and some of their attributes. As many other similar systems, RDT also assist in the management of documentation and drawings associated with a contract and with producing various requirements reports.[14]

RequisitePRO (Rational) is a marketed as groupware for requirements management. The system tailors with the suite of other Rational products, in particular Rational Rose (CASE tool) and Rational SODA (documentation generator). It allows creation of multimedia requirements documents with Microsoft Word, structuring recorded requirements in an integrated database, and

---

10 Further information can be found at URL: http://www.threesl.com/.

11 Description and demo are available at URL: http://www.qssinc.com/products/doors/index.html.

12 Further infomration can be found at URL: http://www.chipware.com/.

13 Description is available at URL: http://www.alc.com/products/sys-eng/rdd100.shtml.

14 Description is available at URL: http://www.ausnet.net.au/gecm/.

sharing document repository between many software projects. Requirements can be allocated various user-defined attributes, can be sorted, filtered, searched and browsed by these attributes, can be placed under a version control and be monitored. The system also offers numerous templates for producing documents in accordance with several requirements standards.[15]

SLATE REquire (TD Technologies, Inc.) requirements management system offers a unique approach to requirements tracing, which is not document-based but instead it is oriented towards the processing of individual requirement statements and their components. The system allows requirements to be attributed, prioritised and assigned. With SLATE, developers can import, identify (based on keywords and regular expressions), create, modify and analyse collections of requirements or only those requirements assigned for analysis. SLATE allows to optionally enforce a development process and to trace requirements throughout its life-cycle. Requirements analysis can utilise pre-defined and ad-hoc reports, symbol and trace tables, and information available from requirements reviews. Standard templates are used for the production of requirements documents. SLATE can also drive requirements stored in its repository into other downstream tools, such as SLATE Architect, several CASE tools or other third-party development environments.[16]

TOFS (Tofs) is a system tightly integrated with Microsoft Office, which assist the analyst in the production of requirements documents and their diagrams. TOFS overlooks this process and structures requirements into a dependency tree with externally attached text, pictures and diagrams. Each requirement can be fully attributed and the progress of its processing can be tracked and reported. Requirements development process can be formalised and then enforced with Odel - a design language intended for description of complex systems. Tofs also offers a requirements/test matrix to visualise connectivity between requirements objects and test cases, it also allows entering test results and comments, which could assist in requirements verification.[17]

Vital Link (Compliance Automation, Inc.) provides a hypertext view of requirements documents that could be linked, annotated and navigated. Individual requirements can be described with attributes related to their allocation, verification, rationale and other user-defined attributes. Vital Link allows managing and reporting all of the document linkage information. A particular feature of Vital Link is its ability to enforce regulatory compliance of produced requirements documents. Regulations can be maintained, reviewed, linked and traced across all types of requirements information. Similarly to RDT, Vital Link provides specialised facilities to deal with the preparation and monitoring of project proposals.[18]

XTieRT (Teledyne Brown Engineering) supports automatic identification and extraction of requirements from text documents. This is accomplished by detecting the use of keywords such as "shall", "will", "should" or "must". The requirements can subsequently be described in terms of their attributes, they can be grouped and organised into a hierarchy, which can then be viewed, filtered, searched and traced. XTieRT also generates several different requirements management reports.[19]

Table 4 is an extract of the summary comparison of the above-mentioned tools [106]). The extract comprises only the earliest stages of requirements capture and analysis - the focus of this paper. The table shows that the majority of commercially available systems have capabilities for requirements identification (by parsing and keyword detection) and classification (automatic or semi-automatic), visualisation (in text and graphics), allocation (for further processing),

---

15 Description and software available from URL: http://www.rational.com/products/reqpro/index.jtmpl.

16 Description and demonstration are available at URL: http://www.tdtech.com/.

17 Description is available at URL: http://www.toolforsystems.com/.

18 Description and interactive demos can be found at URL: http://www.complianceautomation.com/.

19 Description and a demo are available at URL: http://www.tbe.com/products/xtie/.

enrichment (by annotation and inter-linking with other documents) and traceability (which includes partial verification and cross-referencing).

Table 4: Comparison of major requirements management tools (fragment)

INCOSE Survey Response Summaries for Requirements Management Tools (8/3/98). Legend: ● **- Feature fully supported** ◑ **- Feature partially supported** **- No support for the feature**	**Caliber RM 1.1**	**CORE 2.0**	**DOORS 4.0**	**RDD-100 4.1.1**	**RDT**	**RequisitePro 2.0**	**RTM 3.x**	**SLATE 4.1**	**Tofs 98**	**Vital Link**	**XTie-RT**
1. Capturing Requirements/Identification											
1.1. Input document enrichment/analysis	●	●	●	●	●	●	●	◑		●	
1.1.1. Input document change/comparison analysis	●	●	◑	●	◑	●	●	●		◑	
1.2. Automatic parsing of requirements		●	●	●	●	◑	●	●		●	●
1.3. Interactive/semi-automatic requirement identification	●	●	●	●	●	◑	●	◑	◑	●	●
1.4. Manual requirement identification	●	●	●	●	●	●	●	●	●	●	●
1.5. Batch mode operation		●	●	●		●	●	◑	◑		●
1.5.1. Batch-mode document/source-link update	●	●	◑	◑	◑	●	●	◑		◑	
1.6. Requirement classification	●	●	●	●	●	●	●	●	●	●	●
2. Capturing system element structure											
2.1. Graphically capture systems structure	●	●	●	●	●	●	●	●	●		●
2.2. Textual capture of systems structure	●	●	●	●	●	●	●	●	●	●	●
3. Requirements flowdown											
3.1. Requirements derivation	●	●	●	●	●	●	●	●	◑	●	●
3.2. Allocation of performance requirements	●	●	●	●	●	●	◑	●	●	◑	●
3.3. Requirement linking to system elements	●	●	●	●	●	●	●	●	●	●	●
3.4. Requirement annotation	●	●	●	●	●	●	●	●	●	●	●
4. Traceability analysis											
4.1. Identify inconsistencies	●	●	●	●	●	●	●	●	●	●	●
4.2. Visibility of links from source to implementation	●	●	●	●	●	●	●	●	●	●	●
4.3. Verification of requirement	●	●	●	●	●	●	●	●	●	●	●
4.4. Performance verification from system elements	●	●	●	●	●		●	●	◑		

There are many other requirements processing tools that do not fully cover the cycle of requirements management, but which address only some of its aspects. Some are of commercial quality, e.g. systems supporting formal requirements specification IFAD VDM[20] or LARCH[21]. Others, although significant in size and features, only investigate the issues of research importance, e.g. PROTEUS - the system facilitating analysis of requirements change through the Truth Maintenance System and allowing automatic capture of requirements in restricted natural language [31].

In addition to these requirements tools, Stephen Andriole [9] identifies several task-specific tools that can be used in support of the requirements engineering process (subset in Table 5). He classifies these tools into three major groups, i.e.

---

[20] Description of IFAD VDM can be found at URL: http://www.ifad.dk/.

[21] Web links to LARCH-related products can be found in: http://www.sds.lcs.mit.edu/Larch/index.html.

- *Process reengineering tools*, which allow modelling and analysis of business processes that need to be improved with the introduction of the software systems;
- *Requirements analysis tools*, which assist the analyst in capturing, organisation and analysis of requirements information; and finally,
- *Prototyping and modelling tools*, which can be used in the production of both formal and executable model of the system and its behaviour.

Some of the tools reported by Andriole concentrate on issues such as decision support, simulation, modelling, graphing and visualisation or reporting. Others are experimental prototypes and concept demonstrators, employing innovative requirements processing methods, to be primarily treated as of research value only. Among the listed tools we can also find development environments and CASE tools.

1.6 Integrating Approaches for Requirements Processing

The work reported in this paper favours yet another approach to dealing with requirements expression. We believe that both informal and formal approaches to requirements engineering can bring certain benefits to the overall process, thus, it seems logical that they both should be integrated into one cohesive formula as also advocated by Balzer [11, 13], Abbott [1], Barstow [14], Johnson [112], Hsia [103] and Burns [31]. We admit the need to capture software requirements in the form and media most appropriate for the problem domain and convenient to the user community. We also advocate that such collections of mostly informal requirements should be analysed, refined and ultimately formalised, so that other conventional methods were possible in the ensuing development process. Where we may differ to many other researchers is our conviction that the best way of improving the cycle of requirements engineering, and elicitation in particular, is to :-

Process Reengineering	Requirements Analysis	Prototyping and Modelling
Fundation Vista	Acta Advantage	Allegro Common Lisp
Intelligent Workbench	Brainstormer	Anatool
Micro Saint	CASEware Modeller	BPwin
ModelPro	Company Ladder	Business Design Facility
Process Charter	Criterium Decision Plus	C-Scape
Re-engineer-iT	Ctrl-C	Cinemation
Revengg	Decision Analysis	Clarion Personal Developer
SIMprocess	Decision Analyst	Crystal Expert
SMARTsystem	Decision Map	Designer
SoftTest	Decision Pad	DYNAMO
WisdomWorks	DeltaGraph	EXSYS
Workflow*BPR	Descartes	Extend
	Design Generator	Hyper Animator
	EXPERT/CIO	ICpak
	Genesis	Instant Replay
	IdeaFisher	Ithink
	IdeaTree	Joshua
	INSPIRATION	MacroMind Director
	MindLink	MedModeller
	Model-C	Summit Process
	QFD Capture	SuperCard
	RTrace	ToolBook
	TopDown	ToolBuilder
	TreeAge	WITNESS
	VIG	@RISK
	VIMDA	

Table 5: Early tools useful in requirements engineering

- process informal documents with a view to identify an opportunity to reuse previously formalised requirements specifications; and to this end,
- provide requirements engineers with such software reuse tools which could assist them in the process of analysis, refinement and formalisation of informal requirements while; while at the same time,
- utilising the skills, knowledge and experience of the people intimately involved in the process.

The concept of reusing system and software requirement documents, their specifications and associated development processes is not new. The recent studies clearly show that early reuse benefits the entire development process and the quality of the final product [50]. The importance of such early reuse can be best characterised by the following remarks.

- Conventional requirements analysis techniques do not provide methods of reusing the results of previous analyses. Analysts are, thus, assumed to start with a blank slate for each new problem, and as a result, they are doomed to recreate previous mistakes [115].
- Many organisations aim at the introduction of systematic software reuse concerned primarily with the reuse of higher level life-cycle artefacts, such as requirements, designs, and subsystems [72, 151].
- Reusing requirements and specifications, rather then designs or code, provides cost and quality benefits, as well as, developmental assistance earlier in the software life-cycle [146].

- CASE-assisted reuse of requirements and high level designs early in the design process results in the reuse of subsequent lifecycle products [174].
- To effectively utilise available resources in the software development (i.e. software artefacts, techniques, methods, tools and human expertise), the resources that are applicable for the development of a target system must be identified early in the life-cycle, i.e. system requirements analysis phase [122].

Others support these claims, voicing the need to reuse large-scale artefacts going beyond design components and including entire design frameworks and domain resources [135]. Bubenko et. al. [29] further propose to combine design and reuse libraries to accommodate development processes capable of reusing conceptual schemas to support the process of requirements engineering. Morel and Faget [161] aim at extending this approach even to the entire software life-cycle.

1.7 Summary

In this paper we reviewed the mainstream of requirements engineering practice and research. We defined the main terms and concepts of the requirements engineering field, we investigated its needs, methods and its frameworks.

In our analysis we observed that requirements engineering initiates the development process, i.e. project inception, its analysis and specification. At the same time, we noted that requirements engineering work-products have their impact not only on the development phases immediately following it, but rather on the entire software life-cycle. In conclusion, we suggested that it is, hence, necessary to structure the development process to explicitly take into account requirements engineering activities and their deliverables.

We subsequently reviewed the main development process models. We discovered that in different models, requirements engineering activities could be fragmented, mixed, combined and scattered throughout the software life-cycle in many different ways. Hence, we believe, that the nature of requirements engineering is not in its temporal inter-relationship with other phases of the development life cycle, but rather in the intra-relationships of the activities that make up the requirements engineering process.

We identified a number of such activities as of special importance to requirements engineering, i.e. setting the requirements management plan, elaboration of needs and objectives, requirements acquisition, requirements specification and modelling, generation and evaluation of alternatives, requirements verification and validation, and finally ending the requirements engineering process. By looking at each of these activities in detail we determined that the factor commonly emphasised as important in the their process is effective communication between the client and developer communities.

We then discussed the issues of formal and informal communication of requirements. We considered the recording media, organisation of requirements documents and the notation used requirements capture. We considered pros and cons of notational formality. We deliberated the dichotomy of needs for requirements communication versus their formalisation. In conclusion we noted the inevitability of using informal approaches to expressing user requirements, e.g. the use of natural language text, and the formal ones in specifying requirements to be subsequently used in software development. We stated that the ideal specification vehicle should be informal enough to allow an untrained customer to understand what system functions will be delivered upon the system completion, and sufficiently formal to allow a system designer to have unambiguous statement of customer requirements that can be implemented and validated.

By means of an example we identified commonly encountered problems in informal, natural language requirements texts, which we believe all could be resolved by improving the analyst's ability to assess requirements similarity and their cross-referencing.

We then summarised the technologies that could assist in dealing with the problem of bridging the formality gap between user and system requirements. Here, we considered a number of different methods that could address the issues of managing client-developer communication, text, model and knowledge management, handling multiple viewpoints, capturing user scenarios, assisting in issue and decision tracking, extending requirements management to enterprise, domain and product-line, etc. The final point lead to the suggestion that reuse can also be helpful in bridging the requirements formality gap.

The paper concludes with a review of several requirements management tools that could provide a complete environment for handling requirements over their life-cycle. It also listed a number of task-specific requirements processing tools. Although by doing this, we hinted at many different ways of automating the requirements engineering process, we also emphasised that people are still the best judges of requirements ambiguity, integrity and completeness, they are still the best communicators and negotiators. Since, in the current state of technology, analysts cannot be replaced by automatic requirements engineering products, we suggested the need to develop support tools capable of incorporating human expertise in the requirements engineering cycle.

1.8 Bibliography

Over the years, the area of requirements engineering has been well established and vigorously researched. There exist several comprehensive reviews of requirements engineering theory and practice. One such review can be found in the book by Gause and Weinberg [81], "Exploring Requirements: Quality Before Design", which is regarded as the seminal work in requirements engineering. The book explores requirements processing as applicable to any development, whether it is software or some other type of commodity. A different view of the field is given by Jackson [107], who wrote a dictionary-like compendium of software requirements engineering concepts, methods and practices, all illustrated with insightful examples and personal reflection. Davis [55, 56] comprehensively reviews many techniques for modelling requirements functions, objects and their states. Loucopoulos and Karakostas [138] in turn offer a reassessment of system requirements processing with a comprehensive description of elicitation, modelling and validation techniques. Wieringa [222] details various requirements management frameworks and strategies useful to practising requirements engineers. Kovits [130] provides a highly practical guide to writing quality requirements documents. Thayer and Dorfman [63, 208] present a two-volume collection of publications on the topic of system and software requirements which includes research papers, standards, guidelines and examples. International symposia, conferences and workshops in the area of requirements engineering have now become regular events, e.g. International Symposium on Requirements Engineering (RE), International Conference on Requirements Engineering (ICRE) and Requirements Engineering Workshop at OOPSLA. Journals and newsletters also provide popular reading material to requirements engineers, e.g. Requirements Engineering Journal (Springer-Verlag), Requirements Engineering Newsletter (mailing list) and Requirenautics Quarterly (of British Computer Society).

1. Abbott, R.J. (1983): *Program design by informal English descriptions.* Communications of the ACM. **26**(11): p. 882-894.

2. Adelson, B. and E. Soloway (1985): *The role of domain experience in software design.* IEEE Trans. Soft. Eng. **11**(11): p. 1351-1360.

3. Adhami, E., R. Pyburn, and R.E.M. Champion (1989): *A knowledge-based approach to computer aided requirement engineering*, in *Software Engineering Environments: Research and Practice*, K.H. Bennett, Editor. Ellis Horwood Ltd: Chichester, West Sussex, England. p. 187-202.

4. Aguilera, C. and D.M. Berry (1990): *The use of a repeated phrase finder in requirements extraction.* Journal of Systems and Software. **13**(3): p. 209-230.

5. Akscyn, R.M., D.L. McCracken, and E.A. Yoder (1988): *KMS: A distributed hypermedia system for managing knowledge in organisations.* Communications of the ACM. **31**(7): p. 820 -835.

6. Ambler, A.L. and M.M. Burnett (1990): *Influence of visual technology on the evolution of language environments*, in *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, Editor. IEEE Computer Society Press: Los Alamitos, California. p. 19-32.

7. Ambras, J. and V. O'Day (1988): *Microscope : A knowledge-based programming environment.* IEEE Software: p. 50-59.

8. Andersen Consulting (1994): *FOUNDATION Methods Version 2.0*, Manual V 9.0, Andersen Consulting, Arthur Andersen & Co.

9. Andriole, S.J. (1996): *Managing Systems Requirements: Methods, Tools and Cases*. New York, NY: McGraw-Hill.

10. Balasubramanian, V. (1993): *Hypermedia Issues and Applications: State-of-the-Art Review*, , Graduate School of Management, Rutgers University, Newark, N.J.

11. Balzer, R. (1985): *15 year perspective on automatic programming.* IEEE Trans. on Soft. Eng. **SE**(11): p. 1257-1268.

12. Balzer, R., R. Fikes, M. Fox, J. McDermott, and E. Soloway (1990): *AI and software engineering will the twain ever meet?* in *8th National Conference on Artificial Intelligence*: MIT Press, p. 1123-1141.

13. Balzer, R.M., N. Goldman, and D. Wile (1978): *Informality in program specifications.* IEEE Trans. on Software Eng. **SE**(4): p. 94-103.

14. Barstow, D. (1984): *A perspective on automatic programming.* The AI Magazine. **5**(1): p. 5-28.

15. Barstow, D.R., H.E. Shrobe, and E. Sandewall, eds. (1986): *Interactive Programming Environments*. . McGraw-Hill Book Co.: New York, NY.

16. Bennett, J.S. (1985): *ROGET: A knowledge-based consultant for acquiring the conceptual structures of a diagnostic expert system.* Journal of Automated Reasoning. **1**: p. 49-74.

17. Berry, D.M., N.M. Yavne, and M. Yavne (1987): *Application of program design language tools to Abbott's method of program design by informal natural language descriptions.* Journal of Systems and Software. **7**: p. 221-247.

18. Biebow, B. and S. Szulman (1989): *Enrichment of semantic network for requirements expressed in natural language*. in *Information Processing'89*. San Francisco, California: North-Holland, p. 693-698.

19. Bielawski, L. and J. Boyle (1997): *Electronic Document Management : A User Centered Approach for Creating, Distributing, and Managing Online Publications*: Prentice-Hall.

20. Bigelow, J. (1988): *Hypertext and CASE.* IEEE Software: p. 23-27.

21. Birrell, N.D. and M.A. Ould (1985): *A Practical Handbook of Software Development*. Cambridge: Cambridge University Press.

22. Boehm, B.W. (1986): *A spiral model of software development and enhancement.* Software Engineering Notes. **11**(4): p. 22-32.

23. Boose, J.H. (1993): *A survey of knowledge acquistion techniques and tools*, in *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems*, B.G. Buchanan and D.C. Wilkins, Editors. Morgan Kaufmann Pubs: San Mateo, CA. p. 39-56.

24. Borgida, A., S. Greenspan, and J. Mylopoulos (1985): *Knowledge representation as the basis for requirements specifications.* IEEE Computer: p. 82-90.

25. Bowen, J.P. and M.G. Hinchey (1994): *Seven More Myths of Formal Methods*, PRG-TR-7-94, Oxford University Computing Laboratory.

26. Brachman, R.J. and H.J. Levesque, eds. (1985): *Readings in Knowledge Representation*. . Morgan Kaufmann Pub. Inc.: Los Altos, California.

27. Brackett, J.W. (1990): *Software Requirements*, SEI Curriculum Module SEI-CM-19l.2, Carnegie Mellon University, Software Engineering Institute.

28. Bradshaw, J.A. and R.M. Young (1991): *Evaluating design using knowledge of purpose and knowledge of structure.* IEEE Expert. **6**(2): p. 33-40.

29. Bubenko, J., C. Rolland, P. Loucopoulos, and V. DeAntonellis (1994): *Facilitating "Fuzzy to Formal" requirements modelling*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 154-157.

30. Buchanan, B.G. and R.G. Smith (1989): *Fundamentals of Expert Systems*, in *The Handbook of Artificial Intelligence*, A.B.P.R. Cohen and E.A. Feigenbaum, Editors. Addison-Wesley Publishing Company, Inc.: Readings, Massachusetts. p. 149-192.

31. Burns, A., D. Duffy, C. MacNish, J. McDeremid, and M. Osborne (1995): *An Integrated Framework for Analysing Changing Requirements*, PROTEUS Deliverable 3.2, Department of Computer Science, University of York.

32. Bush, V. (1945): *As we may think.* Atlantic Monthly. **176**: p. 101-108.

33. Carando, P. (1989): *Shadow: Fusing hypertext with AI.* IEEE Expert. **4**(4): p. 65-78.

34. Carenini, G., M. Ponzi, and O. Stock (1990): *Combining natural language and hypermedia as new means for information access*. in *Proc. Fifth European Conference on Cognitive Ergonomics Proceedings (ECCE-5)*. Urbino, Italy: European Association of Cognitive Ergonomics Free Univ, Amsterdam, Netherlands, p. 315-23.

35. Carlson, D.A. and S. Ram (1990): *HyperIntelligence: the next frontier.* Communications of the ACM. **33**(3): p. 311-321.

36. Castano, S. and V. De Antonellis (1994): *The F3 Reuse Environment for Requirements Engineering.* ACM SIGSOFT Software Engineering Notes. **19**(3): p. 62 -65.

37. Chandrasekaran, B. (1990): *Design problem solving: A task analysis.* The AI Magazine. **11**(4): p. 59-71.

38. Chang, S.-K. (1990): *Visual languages: a tutorial and survey*, in *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, Editor. IEEE Computer Society Press: Los Alamitos, California. p. 7 -17.

39. Chen, P.-M. and C.-R. Chou (1988): *The requirement model in a knowledge-based rapid prototyping system*. in *12th Annual International Computer Software & Applications Conference*: Computer Society Press of the IEEE, p. 418-426.

40. Chervak, S.G., C.G. Drury, and J.P. Ouellette (1996): *Field Evaluation of Simplified English for Aircraft Workcards*. in *10th FAA/AAM Meeting on Human Factors in Aviation Maintenance and Inspection*. Alexandria, Virginia: FAA Office of Aviation Medicine, p. http://hfskyway.com/hfami/mtng10/drury.htm.

41. Chin, D.N., K. Takea, and I. Miyamoto (1989): *Using natural language and stereotypical knowledge for acquisition of software models*. in *Proc. IEEE International Workshop on Tools for Artificial Intelligence*. Fairfax, VA, USA: IEEE, IEEE Service Center, Piscataway, NJ, USA, p. 290-295.

42. Christel, M.G. and K.C. Kang (1992): *Issues in Requirements Elicitation*, CMU/SEI-92-TR -12, Software Engineering Institute, Carnegie Mellon University.

43. Clavadetscher, C. (1998): *User involvement key to success.* IEEE Software. **15**(2): p. 30,32.

44. Clements, P.C. (1997): *Report of the "Reuse and Product Lines" Working Group of WISR8*, Web Report http://www.cis.ohio-state.edu/~weide/WISR8/wisr8-summary/product-long.html, WISR-8: Columbus, Ohio.

45. Cmu/Sei (1991): *Requirement Engineering and Analysis*, Technical Report CMU/SEI-91-TR -30, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

46. Conklin, J. (1987): *Hypertext: An Introduction and Survey.* IEEE Computer: p. 17-40.

47. Conklin, J. and M.L. Begeman (1987): *gIBIS: A hypertext tool for team design deliberation*. in *Hypertext'87*. Chapel Hill, NC: ACM, p. 247-251.

48. Conklin, J.E. and W. Weil (1998): *Wicked Problems: Naming the Pain in Organizations*, Web Report http://www.gdss.com/wicked.htm, Group Decision Support Systems: Washington D.C.

49. Cordingley, E.S. (1989): *Knowledge elicitation techniques for knowledge-based systems*, in *Knowledge Elicitation: Principles, Techniques and Applications*, D. Diaper, Editor. Ellis Horwood Limited: Chichester. p. . 90-265.

50. Cybulski, J.L. (1996): *Introduction to software reuse*, Research Report 96/4, The University of Melbourne, Department of Information Systems: Melbourne.

51. Cybulski, J.L. and K. Reed (1992): *A hypertext-based software engineering environment.* IEEE Software. **9**(2): p. 62-68.

52. Cyre, W. (1989): *Toward synthesis from English descriptions*. in *26th ACM/IEEE Design Automation Conference*. Las Vegas, USA: ACM Press, p. 742-745.

53. Dankel, D.D., M.S. Schmalz, and K.S. Nielsen (1994): *Understanding natural language software specifications*. in *Fourteen International Avignon Conference, AI'94*. Paris, France: Ec-2, p. .

54. Davies, C.G. and P.J. Layzell (1993): *The Jackson Approach to System Development*. Sweden: Chartwell-Bratt.

55. Davis, A.M. (1990): *Software Requirements: Analysis and Specification*. Englewood Cliffs, New Jersey: Prentice Hall.

56. Davis, A.M. (1993): *Software Requirements: Objects, Functions and States*. 2nd ed. Upper Saddle River, New Jersey: Prentice Hall.

57. Davis, A.M. (1998): *Predictions and farewells.* IEEE Software. **15**(4): p. 6 -9.

58. Davis, A.M. and D.A. Leffingwell (1996): *Using Requirements Management to Speed Delivery of Higher Quality Applications*, White Paper URL http://www.rational.com/products/reqpro/prodinfo/whitepapers/, Rational Software Corp.: Boulder, CO.

59. DeBellis, M. (1990): *The Concept Demonstration Rapid Prototype System*. in *Fifth Annual Knowedge-Based Software Assistant Conference*. Syracuse, NY, p. .

60. Delisle, N. and M. Shwartz (1986): *Neptune: A hypertext system for CAD applications*. in *ACM SIGMOD Int. Conf. on Management of Data*. Washington, D.C., p. 132-143.

61. Devanbu, P., R.J. Brachman, P.G. Selfridge, and B.W. Ballard (1991): *LaSSIE: A knowledge-based software information System.* Communications of ACM. **34**(5): p. 34-49.

62. Dorfman, M. (1997): *Requirements Engineering*, in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 7-22.

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

64. Duncan, E.B. (1989): *A concept-map thesaurus as a knowledge-based hypertext interface to a bibliographic database*. in *Informatics 10: Prospects for Intelligent Retrieval*. Cambridge, England: Aslib, The Association for Information Management, p. 43-52.

65. Eliot, L.B. and W. Scacchi (1986): *Towards a knowledge-based system factory: issues and implementations.* IEEE Expert. **1**(4): p. 51-58.

66. Engelien, B. and M. Ronnie (1991): *Natural Language Markets: Commercial Strategies*. London, England: Ovum Ltd.

67. Feather, M.S. (1989): *Reuse in the context of a transformation-based methodology*, in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. p. 337-359.

68. Finkelstein, A., D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh (1994): *Inconsistency Handling In Multi-Perspective Specifications.* IEEE Transactions on Software Engineering. **20**(8): p. 569-578.

69. Fisher, G. and M. Schneider (1984): *Knowledge-based communication processes in software engineering*. in *Proc. ICSE-7*, p. .

70. Fletton, N.T. (1990): *A hypertext approach to browsing and documenting software*, in *Hypertext: State of the Art*, R. McAleese and C. Green, Editors. Intellect Ltd: Oxford, England. p. 193-204.

71. Flores-Méndez, R.A. (1997): *JAVA Concept Maps for the Learning Web*. in *Ed-Media'97*. Calgary, Canada: Association for the Advancement of Computing in Education, p. .

72. Frakes, W. and S. Isoda (1994): *Sucess factors of systematic reuse.* IEEE Software. **11**(5): p. 15-19

73. Frakes, W., R. Prieto-Diaz, and C. Fox (1998): *DARE: domain analysis and reuse environment.* Annals of Software Engineering. **5**: p. 125-141.

74. Frenkel, K.A. (1985): *Toward automating the software-development cycle.* Communications of the ACM. **28**(6): p. 578-589.

75. Fuchs, N.E., H.F. Hofmann, and R. Schwitter (1994): *Specifying Logic Programs in Controlled Natural Language*, 94.17, Department of Computer Science, University of Zurich.

76. Gaines, B.R. and M.L.G. Shaw (1994): *Concept Maps as Hypermedia Components* Technical Report http://ksi.cpsc.ucalgary.ca/articles/ConceptMaps/, Knowledge Science Institute, University of Calgary: Calgary, Alberta, Canada.

77. Gaines, B.R. and M.L.G. Shaw (1996): *WebGrid: Knowledge Modeling and Inference through the World Wide Web*. in *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada: Knowledge Science Institute, University of Calgary, p. http://ksi.cpsc.ucalgary.ca/KAW/KAW96/gaines/KMD.html.

78. Garg, P.K. and W. Scacchi (1987): *On designing intelligent hypertext systems for information management in software engineering.* in *Hypertext '87*. North Carolina, USA: The Association for Computing Machinery, p. 409-432.

79. Garg, P.K. and W. Scacchi (1989): *ISHYS: Designing an intelligent software hypertext system.* IEEE Expert. **4**(3): p. 52-63.

80. Garlan, D. (1990): *The role of formal reusable frameworks*. in *ACM SIGSOFT, International Workshop on Formal Methods in Software Development*. Napa, California: ACM Press, p. 42-44.

81. Gause, D.C. and G.M. Weinberg (1989): *Exploring Requirements: Quality Before Design*. New York, NY: Dorset House Publishing.

82. Gero, J.S. (1990): *Design prototypes: A knowledge representation schema for design.* The AI Magazine. **11**(4): p. 26-36.

83. Girgensohn, A., D.F. Redmiles, and F.M. Shipman, III (1994): *Agent-based support for communication between developers and users in software design*. in *KBSE'94, The Ninth Knowledge-Based Software Engineering Conference*. Monterey, California: IEEE Computer Society Press, p. 22-29.

84. Goguen, J.A. and C. Linde (1993): *Techniques for requirements elicitation*. in *IEEE International Symposium on Requirements Engineering*. San Diego, CA: IEEE Computer Society Press, p. .

85. Goldberg, A.T. (1986): *Knowledge-based programming: a survey of program design and construction techniques.* IEEE Transactions on Software Engineering. **SE**(12): p. 752-768.

86. Goldin, L. and D.M. Berry (1994): *AbstFinder, a prototype abstraction finder for natural language text for use in requirement elicitation: design, methodologies, and evaluation*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 84-93.

87. Gotel, O.C.Z. and A.C.W. Finkelstein (1994): *An analysis of the requirements traceability problem*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 94-101.

88. Goyvaerts, P. (1996): *Controlled English, Curse or Blessing? - A User's Perspective*. in *1st Int. Workshop on Controlled Language Applications, CLAW'96*. Katholieke Universiteit Leuven, Leuven, Belgium, p. 137-142.

89. Greenspan, S., J. Mylopoulos, and A. Borgida (1982): *Capturing more world knowledge in the requirements specification*. in *6th International Conference on Software Engineering*: The Institute of Electrical and Electronics Engineers, p. 231-240.

90. Greenspan, S., J. Mylopoulos, and A. Borgida (1994): *On formal requirements modeling languages: RML revisited*. in *16th International Conference on Software Engineering*. Sorrento, Italy: IEEE Computer Society Press, p. 135-147.

91. Group Decision Support Systems (1998): *The IBIs Manual: A Short Course in IBIS Methodology*, Web Report http://www.gdss.com/IBIS.htm, Group Decison Support Systems: Washington D.C.

92. Halasz, F.G. (1988): *Reflections on notecards: seven issues for the next generation of hypermedia systems.* Communications of the ACM. **31**(7): p. 836-852.

93. Hall, A. (1990): *Seven Myths of Formal Methods.* IEEE Software: p. 11-19.

94. Harandi, M.T. (1986): *Applying knowledge-based techniques to software development.* Perspectives in Computing(Spring).

95. Heidorn, G.E. (1976): *Automatic programming through natural language dialogue: A survey.* IBM J. Res. Develop. **20**(4): p. 302-313.

96. Hickman, L. and C. Longman (1994): *CASE Method: Business Interviewing*. Wokingham, England: Addison-Wesley Pub. Co.

97. Hobbs, R.W. and T.P. Gorman (1986): *Diogenes: an expert system for extraction of data system requirements from user scenarios*. in *Expert Systems in Government Symposium*. McLean, Virginia, USA: IEEE Computer Society Press, p. 418-422.

98. Hoffer, J., J.F. George, and J.S. Valacich (1999): *Modern Systems Analysis and Design*. Readings, MA: Addison-Wesley.

99. Hofmann, H.F. (1993): *Requirement Engineering: A Survey of Methods and Tools*, 93.05, Institut fŸr Informatik der Universitat ZŸrich.

100. Hollocker, C.P. (1990): *A review process mix*, in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 485-491.

101. Hooks, I. (1993): *Writing good requirements: A requirements working group information report.* in *Third Int. Symp. of the NCOSE*: INCOSE, p. .

102. Horowitz, E. and R.C. Williamson (1986): *SODOS: a software documentation support environment - its use*, in *Computer-Aided Software Engineering (CASE)*, E. Chikofsky, Editor. IEEE Computer Society Press: Los Alamitos, California, USA. p. 76-87.

103. Hsia, P., A. Davis, and D. Kung (1993): *Status Report: Requirements Engineering.* IEEE Software. **10**(6): p. 75-79.

104. IEEE (1984): *Guide to Software Requirement Specifications*, Std 830-1984, The Institute of Electrical and Electronics Engineers, Inc.

105. IEEE/EIA (1998): *Information Technology - Software Life Cycle Processes*, IEEE/EIA Standard 12207, IEEE.

106. INCOSE (1998): *Tools Survey: Requirements Management Tools*, Web Report http://www.incose.org/tools/tooltax.html, International Council on Systems Engineering.

107. Jackson, M. (1995): *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Wokingham, England: Addison-Wesley.

108. Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard (1992): *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley Longman.

109. Jarke, M. (1992): *Strategies for integrating CASE environments.* IEEE Software: p. 54-61.

110. Johnson, E. (1996): *LinguaNet (TM) - Controlling Police Communication*. in *1st Int. Workshop on Controlled Language Applications, CLAW'96*. Katholieke Universiteit Leuven, Leuven, Belgium, p. 115-123.

111. Johnson, L. and E. Soloway (1985): *PROUST: Knowledge-based program understanding.* IEEE Trans. on Soft. Eng. **10**(3): p. 267 -275.

112. Johnson, W.L., K.M. Benner, and D.R. Harris (1993): *Developing formal specifications from informal requirements.* IEEE Expert. **8**(4): p. 82-90.

113. Johnson, W.L. and M.S. Feather (1991): *Using evolution transformation to construct specifications*, in *Automatic Software Design*, M.R. Lowry and R.D. McCartney, Editors. AAAI Press / The MIT Press: Menlo Park, California. p. 65-91.

114. Johnson, W.L., M.S. Feather, and D.R. Harris (1991): *The KBSA requirements/specification facet: ARIES*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 48-56.

115. Johnson, W.L. and D.R. Harris (1991): *Sharing and reuse of requirements knowledge*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 57-66.

116. Jones, D.A., D.M. York, J.F. Nallon, J. Simpson, and I.R.W. Group (1995): *Factors Influencing Requirement Management Toolset Selection*. in *Fifth Annual Symposium of the National Council on Systems Engineering*: International Council on Systems Engineering, p. http://www.incose.org/lib/rmtools.html.

117. Kado, M., M. Hirakawa, and T. Ichikawa (1992): *HI-VISUAL for hierarchical development of large programs*. in *IEEE Workshop on Visual Languages*. Seattle, Washington: IEEE Computer Society, p. 48-54.

118. Kaindl, H. (1993): *The missing link in requirements engineering.* ACM SIGSOFT Software Engineering Notes. **18**(2): p. 30-39.

119. Kaiser, G.E., P.H. Feller, and S.S. Popovich (1988): *Intelligent assistance for software development and maintenance.* IEEE Software: p. 40-49.

120. Kaiya, H., M. Saeki, and K. Ochimizu (1995): *Design of a hyper media tool to support requirements elicitation meetings*. in *Seventh International Workshop on Computer-Aided Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society Press, Los Alamitos, California, p. 250-259.

121. Kang, K., S. Cohen, J. Hess, W. Novak, and S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie-Mello University.

122. Kang, K.C., S. Cohen, R. Holibaugh, J. Perry, and A.S. Peterson (1992): *A Reuse-Based Software Development Methodology*, Technical Report CMU/SEI-92-SR -4, Software Engineering Institute.

123. Kang, K.C., S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh (1998): *FORM: a feature -oriented reuse method with domain-specific reference architectures.* Annals of Software Engineering. **5**: p. 143-168.

124. Kerola, P. and H. Oinas-Kukkonen (1993): *Hypertext System as an Intermediary Agent in CASE Environments*, Series A 16, University of Oulu, Department of Information Processing Science.

125. Kincaid, J.P., M. Thomas, K. Strain, I. Couret, and K. Bryden (1990): *Controlled English for international technical communication*. in *Human Factors Society 34th Annual Meeting*. Orlando, Florida, p. 815-819.

126. Kinoshita, T. (1989): *A knowledge acquisition model with applications for requirements specification and definition.* SIGART Newsletter(108): p. 166-168.

127. Klinker, G., S. Benetet, and J. McDermott (1988): *Knowledge acquisition for evaluation systems.* International Journal of Man-Machine Studies. **29**(6): p. 715-732.

128. Kotonya, G. and I. Sommerville (1996): *Requirements Engineering with viewpoints.* Software Engineering. **1**(11): p. 5-18.

129. Kotonya, G. and I. Sommerville (1998): *Requirements Engineering: Processes and Techniques*. Chichester, England: Wiley.

130. Kovits, B.L. (1999): *Practical Software Requirements: A Manual of Content and Style*. Greenwitch, CT: Manning Publications Co.

131. Kozaczynski, W. and J.Q. Ning (1989): *SRE: a knowledge-based environment for large-scale software re-engineering activities*. in *11th International Conference on Software Engineering*. Pittsburgh, Pennsylvania, USA: IEEE Computer Society Press, p. 113-122.

132. Ladkin, P.B., L.Z. Markosian, and A. Sterrett (1988): *System development by domain-specific synthesis*. in *3rd International Conference on Applications of Artificial Intelligence*. Stanford University, USA, p. .

133. Lawrence, B. (1998): *Designers must do the modelling.* IEEE Software. **15**(2): p. 31,33.

134. Leite, J.C.P. and P.A. Freeman (1991): *Requirements validation through viewpoint resolution.* IEEE Transactions of Software Engineering. **12**(12): p. 1253-1269.

135. Li, H. (1993): *Reuse-in-the-large: modeling, specification and management*, in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, P.-D. Ruben and B.F. William, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 56-65.

136. Litvintchouk, S.D. and A.S. Matsumoto (1989): *Design of ADA systems yielding reusable components: an approach using structured algebraic specification*, in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. p. 227-245.

137. Loucopoulos, P. and R.E.M. Champion (1989): *Knowledge-based support for requirement engineering.* Information and Software Technology. **31**(3).

138. Loucopoulos, P. and V. Karakostas (1995): *System Requirements Engineering*. London, UK: McGraw-Hill.

139. Lowry, M. and R. Duran (1989): *Knowledge-based software engineering*, in *The Handbook of Artificial Intelligence*, A. Barr, P.R. Cohen, and E.A. Feigenbaum, Editors. Addison-Wesley Publishing Company, Inc.: Readings, Massachusetts. p. 241-322.

140. Lowry, M.R. and R.D. McCartney (1991): *Automatic software design*. Menlo Park, CA: AAITP Press.

141. Loy, P.H. and J.M. Mitchell (1990): *Software requirements specification: the AUTOTELLER automatic teller system*, in *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, M. Dorfman and R.H. Thayer, Editors. IEEE Computer Society Press: Los Alamitos, California, USA. p. 439-456.

142. Lubars, M.D. (1988): *Wide-spectrum support for software reusability*, in *Software Reuse: Emerging Technology*, W. Tracz, Editor. Computer Society Press: Washington, D.C. p. 275-281.

143. Lubars, M.D. and M.T. Harandi (1986): *Intelligent support for software specification and design.* IEEE Expert. **1**(4): p. 33-41.

144. Maarek, Y.S. and D.M. Berry (1989): *The use of lexical affinities in requirements extraction*. in *5th International Workshop on Software Specification and Design*: IEEE Computer Society Press, p. 196-202.

145. Madsen, K.H. (1994): *A guide to metaphorical design.* Communications of the ACM. **37**(12): p. 57-62.

146. Maiden, N. and A. Sutcliffe (1989): *The abuse or re-use: why cognitive aspects of software re-usability are important*, in *Software Re-use, Ultecht 1989*, L. Dusink and P. Hall, Editors. Springer-Verlag: London, U.K. p. 109-113.

147. Maiden, N.A.M. and A.G. Sutcliffe (1994): *Requirements critiquing using domain abstractions*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 184-193.

148. Mannon, M., B. Keepence, and D. Harper (1998): *Using viewpoints to define domain requirements.* IEEE Software. **15**(1): p. 95-102.

149. Martin, J. (1990): *Information Engineering*. Vol. II. Englewood Cliffs, New Jersey: Prentice Hall.

150. Martin, J. (1991): *Rapid Application Development*. Singapore: Maxwell MacMillan International.

151. Matsumoto, Y. (1989): *Some experiences in promoting reusable software: presentation in higher abstract levels*, in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. p. 157-185.

152. McCartney, R.D. (1991): *Knowledge-Based Software Engineering: Where we are and where we are going*, in *Automatic Software Design*, M.R. Lowry and R.D. McCartney, Editors. AAAI Press / The MIT Press: Menlo Park, California. p. xvii-xxxi.

153. McConnell, S. (1993): *Code Complete*. Redmond, Washington: Microsoft Press.

154. McConnell, S. (1998): *Software Project: Survival Guide*. Redmond, Washington: Microsoft Press.

155. McDermid, J.A. (1994): *Requirements analysis: Orthodoxy, fundamentalism and heresy*, in *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J.A. Goguen, Editors. Academic Press: London. p. 17-40.

156. McGraw, K.L. (1990): *HyperKAT: a tool to manage and document knowledge acquisition*, in *Readings in Knowledge Acquisition: Current Practices and Trends*, K.L. McGraw and C.R. Westphal, Editors. Ellis Horwood Ltd: West Sussex, England. p. 164-181.

157. McGraw, K.L. and K. Harbison-Briggs (1989): *Knowledge Acquisition: Principles and Guidelines*. London, UK: Prentice-Hall International.

158. Means, L. and K. Godden (1996): *The Controlled Automotive Service Language (CASL) Project*. in *1st Int. Workshop on Controlled Language Applications, CLAW'96*. Katholieke Universiteit Leuven, Leuven, Belgium, p. 106-114.

159. Meyer, B. (1985): *On formalism in specifications.* IEEE Software: p. 6-26.

160. MIL-STD-498 (1996): *MIL-STD-498 Overview and Tailoring Guidebook*, Report MIL-STD-498, Joint Logistics Commanders, Joint Policy Coordinating Group on Computer Resources Management: Washington, DC.

161. Morel, J.-M. and J. Faget (1993): *The REBOOT environment*, in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, P.-D. Ruben and B.F. William, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 80-88.

162. Morris, S.J. and A.C.W. Finkelstein (1994): *Development of Multiple Media Documents*, , Imperial College, Department of Computing.

163. Mullery, G.P. (1979): *CORE - a method for controlled requirements specification*. in *Fourth International Conference on Software Engineering*: IEEE, p. 126-135.

164. Myer, B.A. (1990): *Visual programming, programming by example, and program visualization: a taxonomy*, in *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, Editor. IEEE Computer Society Press: Los Alamitos, California. p. 33-40.

165. Naka, T. (1987): *Pseudo Japanese specification tool.* Faset. **1**: p. 29-32.

166. Nanduri, S. and S. Rugaber (1995): *Requirements validation via automatic natural language parsing.* Journal of Management Information Systems. **12**(2): p. 9-19.

167. Naughton, M.J. (1989): *A specific knowledge acquisition methodology for expert systems development: a brief look at precision knowledge acquisition*, in *Structuring Expert Systems - Domain, Design, and Development*, J. Liebowitz and D.A.D. Salvo, Editors. Yourdon Press: New Jersey, USA. p. 25-52.

168. Nielsen, J. (1990): *The art of navigating through hypertext.* Communications of the ACM. **33**(3): p. 296 -310.

169. Niskier, C., T. Maibaum, and D. Schwabe (1989): *A look through PRISMA: towards pluralistic knowledge-based environments for software specification acquisition*. in *5th International Workshop on Software Specifications and Design*: IEEE Computer Society Press, p. 128-136.

170. Nunamaker, J.F.J., R.O. Briggs, and D.D. Mittleman (1995): *Electronic Meeting Systems: Ten Years of Lessons Learned*, in *Groupware: Technology and Applications*, D. Coleman, and Khanna, R., Editor. Prentice-Hall. p. 149-193.

171. Nuseibeh, B., J. Kramer, and A. Finkelstein (1994): *A framework for expressing the relationships between multiple views in requirements specification.* Trans. on Software Engineering. **20**(10): p. 760-773.

172. Palmer, J.D. and N.A. Fields (1992): *An integrated environment for requirements engineering.* IEEE Software: p. 80-85.

173. Potts, C., K. Takahashi, and A.I. Ant—n (1994): *Inquiry-based requirements analysis.* IEEE Software. **11**(2): p. 21-32.

174. Poulin, J. (1993): *Integrated support for software reuse in computer-aided software engineering (CASE).* ACM SIGSOFT Software Engineering Notes. **18**(4): p. 75 -82.

175. Pressman, R.S. (1992): *Software Engineering: A Practitioner's Approach*. 3 ed. New York, N.Y.: McGraw-Hill, Inc.

176. Prieto-Diaz, R. and G. Arango, eds. (1991): *Domain Analysis and Software Systems Modeling*. . IEEE Computer Society Press: Los Alamitos, California.

177. Puncello, P.P., P. Torrigiani, F. Pietri, R. Burlon, B. Cardile, and M. Conti (1988): *ASPIS: a knowledge-based CASE environment.* IEEE Software: p. 58-65.

178. Ramesh, B. and V. Dhar (1992): *Supporting systems development using knowledge captured during requirements engineering.* Report to appear in IEEE Transactions on Software Engineering: p. 1-25.

179. Raymond, D.R. and F.W.M. Tompa (1988): *Hypertext and The Oxford English Dictionary*. Communications of the ACM. **31**(7): p. 871-879.

180. Reed, K. (1988): *The original technical program for the Amdahl Australian Intelligent Tools Program*, Technical Report 1, La Trobe Univeristy, CS&CE, AAITP: Bundoora.

181. Reubenstein, H.B. and R.C. Waters (1989): *The requirements apprentice: An initial scenario*. in *5th International Workshop on Software Specifications and Design*: IEEE Computer Society Press, p. 211-218.

182. Reubenstein, H.B. and R.C. Waters (1991): *The Requirements Apprentice: Automated assistance for requirement acquisition.* IEEE Trans. on Soft. Eng. **17**(3).

183. Rich, C. and H. Shrobe (1984): *Initial report on the Programmer's Apprentice*, in *Interactive Programming Environments*, D. Barstow, H. Shrobe, and E. Sandewall, Editors. McGraw-Hill: New York, NY. p. 443-463.

184. Rich, C. and R.C. Waters, eds. (1986): *Artificial Intelligence and Software Engineering*. . Morgan Kaufmann Pub. Inc.: Los Altos, California.

185. Rock-Evans, R. (1989): *CASE Analyst Workbenches: A Detailed Product Evaluation*. London, England: Ovum Ltd.

186. Saeki, M., H. Horai, and H. Enomoto (1989): *Software development process from natural language specifications*. in *11th International Conference on Software Egineering*Pittsburgh, Pennsylvania: IEEE Computer Press, p. 64 - 73.

187. Salek, A., P.G. Sorenson, J.P. Tremblay, and J.M. Punshon (1994): *The REVIEW system: From formal specifications to natural language*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 220-229.

188. Sannella, D. (1993): *A survey of formal software development methods*, in *Software Engineering: A European Perspective*, R.H. Thayer and A.D. McGettrick, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 281-297.

189. Schachtl, S. (1996): *Requirements for Controlled German in Industrial Applications*. in *1st Int. Workshop on Controlled Language Applications, CLAW'96*. Katholieke Universiteit Leuven, Leuven, Belgium, p. 143-149.

190. Schoen, E. (1991): *Active assistance for domain modeling*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York: IEEE Computer Society Press, p. 26-35.

191. Schoen, E., R.G. Smith, and B.G. Buchanan (1988): *Design of knowledge-based systems with a Knowledge-Based Assistant.* IEEE Transactions on Software Engineering. **14**(12): p. 1771-1791.

192. Schwitter, R. and N.E. Fuchs (1996): *Attempto - from specifications in controlled natural language towards executable specifications*. in *GI EMISA Workshop, Natürlichsprachlicher Entwurf von Informationssystemen*. Tutzing, Germany, p. 163-177.

193. Seppanen, V. and M. Heikkinen (1991): *Real-life experience from the acquisition of software design knowledge*. in *11th International Conference Expert Systems & their Applications*. Avignon, France: Ec2, p. 111-121.

194. Shaw, M.L.G. and B.R. Gaines (1989): *Comparing Conceptual Structures: Consensus, Conflict, Correspondence and Contrast*, Technical Report http://ksi.cpsc.ucalgary.ca/articles/KBS/COCO/, Knowledge Science Institute, Department of Computer Science, University of Calgary: Calgary, Alberta, Canada.

195. Simos, M., D. Creps, C. Klinger, L. Levine, and D. Allemang (1996): *Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook Version 2.0*, Informal Technical Report STARS-VC-A025/001/00, also available on line: <URL: http://direct.asset.com/wsrd/product.asp?pf_id=ASSET_A_1176>, Lockheed Martin Tactical Defense Systems: M**a**assas, VA.

196. Slagle, J.R., D.A. Gardiner, and K. Han (1990): *Knowledge specification of an expert system.* IEEE Expert. **5**(4): p. 29-38.

197. Sommerville, I. (1992): *Software Engineering*. 4 ed. Wokingham, England: Addison-Wesley Pub. Co.

198. Sommerville, I. and P. Sawyer (1977): *Viewpoints: principle, problems and a practical approach to Requirements Engineering*, Technical Report CSEG/15/1997, Lancaster University, Computing Department: Lancaster.

199. Sommerville, I. and P. Sawyer (1997): *Requirements Engineering: A Good Practice Guide*. Chichester, England: Wiley.

200. Sowa, J.F. and E.C. Way (1986): *Implementing a semantic interpreter using conceptual graphs.* IBM J. Res. Develop. **30**(1): p. 57-69.

201. Srinivas, Y.V. (1991): *Algebraic specification for domains*, in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 90-124.

202. Standish Group (1995): *The CHAOS Report*, Web Report http://www.standishgroup.com/chaos.html, Standish Group International, Inc.

203. Stefik, M. and L. Conway (1982): *Towards the principled engineering of knowledge.* The AI Magazine. **3**(3): p. 4-16.

204. Steier, D. (1991): *Automating algorithm design within a general architecture for intelligence*, in *Automatic Software Design*, M.R. Lowry and R.D. McCartney, Editors. AAAI Press / The MIT Press: Menlo Park, California. p. 577-602.

205. Stewart, V. (1997): *Business Applications of Repertory Grid*, WWW http://www.enquirewithin.co.nz/business.htm, Enquire Within: Wellington, New Zealand.

206. Tamai, T. (1989): *Applying the knowledge engineering approach to software development*, in *Japanese Perspectives in Software Engineering*, Y. Matsumoto and Y. Ohno, Editors. Addison-Wesley Publishing Company: Singapore. p. 207-227.

207. Taniak, M.M. and Y.Y. Yun (1988): *Interaction between Expert Systems and Software Engineering.* IEEE Expert(Winter).

208. Thayer, R.H. and M. Dorfman, eds. (1997): *System and Software Requirements Engineering*. 2nd ed. . IEEE Computer Society Press: Los Alamitos, California.

209. Tichy, W. (1995): *Configuration Management (Trends in Software, No 2)*: John Wiley and Sons.

210. Tsai, J.J.P. and J.C. Ridge (1988): *Intelligent support for specifications transformation.* IEEE Software: p. 28-35.

211. Vadera, S. and F. Meziane (1994): *From English to formal specifications.* The Computer Journal. **37**(9): p. 753-763.

212. Ventana (1998): *Group System Overview*, WWW http://www.ventana.com/html/overview.html, Ventana: Tucson, Arizona.

213. V-Model-97 (1997): *Development Standard for IT Systems of the Federal Republic of Germany*, General Directive 250/1, BWB IT I 5: Koblenz, Germany.

214. Vonk, R. (1989): *Prototyping: The Effective Use of CASE Technology*. Englewood Cliffs, N.J.: Prentice-Hall Int.

215. Waters, R. (1984): *The Programmer's Apprentice: Knowledge base program editing*, in *Interactive Programming Environments*, D. Barstow, H. Shrobe, and E. Sandewall, Editors. McGraw-Hill. p. 464-486.

216. Waters, R.C. and Y.M. Tan (1991): *Toward a design apprentice: Supporting reuse and evolution in software design*. ACM Software Engineering Notes. **16**(2): p. 33 -44.

217. Weidenhaupt, K., K. Pohl, M. Jarke, and P. Haumer (1998): *Scenarios in system development: current practice.* IEEE Software. **15**(2): p. 34-45.

218. Wells, T.L. (1989): *Hypertext as a means for knowledge acqusition.* SIGART Newsletter(108): p. 136-138.

219. Wetherbe, J.C. and N.P. Vitalari (1994): *Systems Analysis and Design: Best Practices*. St. Paul, Minneapolis: West Pub. Co.

220. White, D.A. (1991): *The knowledge-based software assistant: a program summary*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 2-6.

221. Whitten, J.L. and L.D. Bentley (1998): *Systems Analysis and Design Methods*. Boston, MA: Irwin / McGraw-Hill.

222. Wieringa, R.J. (1996): *Requirements Engineering: Frameworks for Understanding*. Chichester, UK: John Wiley & Sons.

223. Wilkinson, N.M. (1995): *Using CRC Cards*: SIGS.

224. Wilkinson, R.T. (1990): *Software requirements specification for the Commodity Paging System*, in *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, M. Dorfman and R.H. Thayer, Editors. IEEE Computer Society Press: Los Alamitos, California. p. 457-477.

225. Wing, J.M. (1988): *A study of 12 specifications of the library problem.* IEEE Software: p. 66-76.

226. Wirfs-Brock, R.J., B. Wilkerson, and L. Wiener (1990): *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall.

227. Wojcik, R.H. and H. Holmback (1996): *Getting a Controlled Language Off the Ground at Boeing*. in *1st Int. Workshop on Controlled Language Applications, CLAW'96*. Katholieke Universiteit Leuven, Leuven, Belgium, p. 22-31.

228. Wood, D.P., M.G. Christel, and S.M. Stevens (1994): *A multimedia approach to requirements capture and modeling*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 53-56.

229. Wood, J. and D. Silver (1989): *Joint Application Design*. New York: John Wiley & Sons.

230. Yonezaki, N. (1989): *Natural language interface for requirements specification*, in *Japanese Perspectives in Software Engineering*, Y. Matsumoto and Y. Ohno, Editors. Addison-Wesley Publishing Company: Singapore. p. 41-76.

231. Zeroual, K. (1989): *Reasoning on requirement specifications: a deductive approach*. in *13th Annual International Computer Software & Applications Conference*: IEEE Computer Society Press, p. 650-657.

232. Zeroual, K. (1991): *KBRAS: a knowledge-based requirements acquisition system*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 38-47.