TOWARDS THE ELICITATION OF HIDDEN DOMAIN FACTORS FROM CLIENTS AND USERS DURING THE DESIGN OF SOFTWARE SYSTEMS

by

Wernher Rudolph Friedrich

submitted in accordance with the requirements

for the degree of

MASTER OF SCIENCE

in the subject

Computer Science

at the

University of South Africa

School of Computing

Supervisor: Professor J A van der Poll

November 2008

Abstract

This dissertation focuses on how requirements for a new software development system are elicited and what pitfalls could cause a software development project to fail if the said requirements are not captured correctly.

A number of existing requirements elicitation methods, namely: JAD (Joint Application Design), RAD (Rapid Application Development), a Formal Specifications Language (Z), Natural Language, UML (Unified Modelling Language) and Prototyping are covered. The aforementioned techniques are then integrated into existing software development life cycle models, such as the Waterfall model, Rapid Prototyping model, Build and Fix model, Spiral model, Incremental model and the V-Process model.

Differences in the domains (knowledge and experience of an environment) of a client and that of the software development team are highlighted and this is done diagrammatically using the language of Venn diagrams. The dissertation also refers to a case study highlighting a number of problems during the requirements elicitation process, amongst other the problem of tacit knowledge not surfacing during elicitation.

Two new requirements elicitation methodologies are proposed namely: the SRE (Solitary Requirements Elicitation) and the DDI (Developer Domain Interaction) methodology. These two methods could potentially be more time consuming than other existing requirements elicitation methods, but the benefits could outweigh the cost of their implementation, since the new proposed methods have the potential to further facilitate the successful completion of a software development project. Following the introduction of the new requirements elicitation methods, they are then applied to the aforementioned case study and highlight just how the hidden domain of the client may become more **visible**, because the software development team has gained a deeper understanding of the client's working environment. They have therefore increased their understanding of how the final product needs to function in order to fulfil the set out requirements correctly.

Towards the end of the dissertation a summary and a conclusion as well as future work that could be undertaken in this area are provided.

Keywords: Requirements Elicitation, JAD, RAD, Formal Method Z, Natural Language, UML, Prototyping, Waterfall model, Rapid Prototyping model, Build and Fix model, Spiral model, Incremental model, V-Process model, Venn diagrams, DDI, SRE, Hidden domain.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my family, especially my grandmother, mother and friends who continued to motivate and inspire me to reach my goal. Without their support this would have been so much more difficult. My supervisor, Professor J A van der Poll, must be thanked for his dedication to me. Without his guidance and insight I would not have been able to complete this dissertation.

I would also like to thank my wife Ané, for her support and dedication and continued motivation, as well as my mother and father-in-law for helping me stay focussed. A special thanks to my editor, Sylvia, she has been a pillar of strength.

Dedication

This dissertation is dedicated to my wife Ané and to my family, especially my grandmother.

Chapter 1
Introduction

This is a dissertation on how to facilitate the elicitation of tacit (hidden) domain knowledge from clients, so as potentially to be able to increase the success rate of a software development project. The reader may possibly wonder what is meant by tacit domain knowledge and this dissertation endeavours to describe what it is and how a domain containing tacit knowledge is created and just how difficult it may be to access knowledge in such a domain. If the factors that reside in the hidden domain of the client (the person(s) requiring the development) are elicited, a software project potentially has a better chance of success.

Below are two definitions of tacit knowledge from literature, essentially stating the same idea.

Definition of Tacit Knowledge

Tacit knowledge according to Blandford and Rugg [14, p78] is "*knowledge which is not accessible to introspection via any elicitation technique*".

Stapleton, Smith and Murphy [86, p164] refer to Baumard's [8] definition of tacit knowledge as *"a body of unspoken intelligence"*. They [86, p164] also refer to Polanyi's [73] definition of tacit knowledge as *"art hidden in the depth of the human soul"* and to his definition of tacit knowledge as "*knowing more than we can tell"*. This statement refers to the fact that individuals often know how to perform a task, but cannot easily explain to another person how to perform that specific task. Tacit knowledge is further defined by definition 1.12 in section 1.1.

In essence, therefore, tacit knowledge is domain specific knowledge residing in the mind of the client and which is not freely available to a software development team. It may furthermore not be easy to unveil such knowledge during a requirements elicitation process.

The reason for the potential increase in the success of a software project once a certain amount of tacit knowledge has been uncovered, is that if a critical success factor residing in the hidden domain is elicited, it may result in the changing of the complete project specification and design of the total solution. This could impact on other phases, such as the implementation and maintenance phases of the project. Correct capturing of requirements for a new software system is of vital importance and this dissertation proposes two new methodologies to aid a software development team in the elicitation of tacit domain knowledge.

It is not always easy for experts in a domain to answer questions or state their knowledge to non experts of the domain. Hudlicka [48, p4] states "… *neither experts nor the intended system users are always able to state their knowledge and system requirements succinctly in response to direct questions.*"

1.1 Definitions and denotations

In addition to the definition of tacit knowledge above, the following definitions and denotations are used throughout this dissertation.

General

- The word **client** normally indicates the person for whom the software is written. The word **user** is often used instead, but mostly indicates a person who will be interacting with the system directly. A **user** is, therefore, often employed by a **client**. The word **customer** when used, implies **client** and vice versa unless otherwise stated.

- The pronoun **he** when used, also implies **she** and vice versa unless otherwise stated.

- The word **him/his** when used, also implies **her**. Likewise, the word **himself** when used also implies **herself** and vice versa unless otherwise stated.

- The word **developer/programmer** when used, implies **software development team** and vice versa unless otherwise stated.

- The list of **references** in this dissertation is sorted alphabetically and these are also numbered.

Definitions

- **Def 1.1** **Architect:** The meaning of **architect**, according to The Concise Oxford Dictionary [88] is a "*designer of complex structure*". He considers issues such as front end and back end processing requirements, how the system will be running, what processing speeds are necessary for facilitating smooth operations in the client's business environment, as well as hardware and software requirements.

- **Def 1.2** **Business Analyst:** A person responsible for analysing the business requirements of a client by examining the details of the situation carefully in order to advise the client on how the new software product would enhance his business.

- **Def 1.3** **Domain** is defined in Longman [60] as "*an area of activity, interest, or knowledge, especially one that a particular person, organisation etc deals with*". A **domain** is therefore a sphere of activity or knowledge.

- **Def 1.4** **Knowledge** is defined in Longman [60] as "*the information, skills, and understanding that you have gained through learning or experience*".

- **Def 1.5** **Project** as defined by Longman [60] is "*a carefully planned piece of work to get information about something, to build something, to improve something*".
  Manager defined by Cambridge [24] is "*someone in control of an office, shop, team etc.*".
  Project Manager as used in this dissertation is therefore a person responsible for the project. He typically develops a project plan with resources and timelines and monitors progress keeping the client up to date on the progress of the project.

- **Def 1.6** A **programmer** according to Longman [60] is "*someone whose job is to write computer programs*".
  Longman [60] defines a **developer** as "*a person or an organisation that works on a new idea, product etc to make it successful: software developers*".
  For the purposes of this dissertation **programmer / developer** is therefore a person who physically develops the software and writes lines of code to create the product as per requirements.

- **Def 1.7** **Software Engineering:** The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software. Pressman [75].
  Software Engineer: A person who practises the above methodology. A software engineer, therefore, performs many of the activities mentioned by definitions 1.1 to 1.9.

- **Def 1.8** **Systems Analyst:** A person responsible for analysing the requirements of the system so the new product will be able to execute it with as few problems as possible, once implemented.

- **Def 1.9** **Tester:** Longman [60] defines tester as "*a person or piece of*

equipment that tests something". Therefore, a tester may also be another computer program. A person responsible for testing the new product that has been developed according to the specifications for the product.

- **Def 1.10 Traditional Tacit Knowledge:**
  As above, "*knowing more than you **can** tell*".

- **Def 1.11 Enhanced Tacit Knowledge:**
  Subconsciously knowing more than you are ***actively*** aware of.

- **Def 1.12 Tacit Knowledge ≡ Old Tacit Knowledge + Enhanced Tacit Knowledge.** Enhanced tacit knowledge is what will be referred to in this dissertation simply as *tacit knowledge*.

1.2 Background

The technological revolution has changed and continues to change the way people work and interact. Freeman and Louçã [38, p301] notice this: "*Even those who have disputed the revolutionary character of earlier waves of technical change often have little difficulty in accepting that a vast technological revolution is now taking place, based on the electronic computer, software, microelectronics, the Internet, and mobile telephones."*

New developments in technology are constantly changing and improving our working processes, enhancing our daily lives and creating more advanced systems. Technology has not only made our daily lives easier, but also more complex, because people want systems with more functionality than before. "*In short, the world can no longer be comprehended as a Simple Machine. It is a Complex, Highly Interconnected System.*" Mitroff and Mohrman [62, p70]. These requirements come from users who need systems to function better, faster and more efficiently. Technology has enhanced our productivity levels in the order of magnitude; people want more, because in the world of today, businesses, people and operations need to move faster.

When a product is bought or a service requested, technology is involved in most cases and will continue to be involved. Products in retail stores are bar coded and people are identified via identification numbers, which are kept in databases of different organisations. Devices used on a daily basis such as cell phones include technology and this continues to grow and evolve as technology advances.

Humans have put technology to good use and products are being manufactured better and faster. Choi and Samavedam [26, p99] write that "*RP (Rapid Prototyping) parts are used for visual inspection, form-fit analysis, ergonomic evaluation, masters for secondary manufacturing processes, etc., in various stages of product development*".

Mass production of products is controlled and quality inspected using technology. Software development and systems integration are the order of the day and many companies have started to realise that if they want their business to survive and remain competitive, they need to adapt and embrace technology and the impact it has on their business. *"Whereas some historians have cast doubts on the pervasiveness and the magnitude of the effects of earlier technological revolutions, such as the railways, few doubt the significance of the Information Technology Revolution and some, such as Castells, see it as ushering in a new type of economy and even a new civilization."* (Freeman and Louçã [38, p301])

Often systems that have been developed many years ago are still in use today. Mainframe systems used in banks are examples of this. Other developments where technology has an impact are in safety critical (life depending) systems such as medical systems, e.g. the diagnosis of life-threatening diseases. An example of an expert system for medical diagnoses is MYCIN (Jackson [52]), which was used to determine blood infections.

When a system needs to be built for a critical environment, the requirements may be daunting at first, but by following certain methodologies many of the requirements for the new system may be elicited. Some methods that have been used in a variety of situations include JAD (Joint Application Design) (Wood and Silver [93]), RAD (Rapid Application Development) (Schach [80]), Prototyping (Olsen [71]), Natural

Language, UML (Unified Modelling Language) (Bahrami [7]) and the use of a formal method such as Z (Potter, Sinclair and Till [74]).

In this work a brief survey in the form of a questionnaire was also undertaken to determine the widespread use of the above methods in a variety of companies in the retail and supply chain sectors of the economy. The results of the questionnaire appear in Appendix A.

We are experiencing a technology revolution. New, better and less expensive products are being manufactured all the time and systems can communicate with each other following standard protocols and integration rules. Freeman and Louçã [38]

Why is this happening and who is driving the technology revolution?

On the one hand vendors drive the technology revolution by continuously developing new products and releasing these into the marketplace, while users on the other hand drive the technology evolution since they require systems with more functionality to enable them to improve their daily lives.

1.3 User requirements

When a business wants to transform the way in which it operates and wishes to improve efficiency, it may decide to introduce more technology into its business operations. This is pointed out by Onuh [69, p220] when he writes: *"Economic and industrial communities worldwide will be subject to the increasing impact of competitive pressures resulting from the globalisation of markets and supply chains to supply these markets."* He is also of the opinion: *"This is an area in which rapid prototyping (RP) technology is assisting companies to remain competitive and be on the leading edge of product innovation and development."*

Introducing technology into a business does not only aid the business processes, but also reduces costs, which translates into higher efficiency rates and higher revenue. Freeman and Louçã [38, p306]: *"As in the earlier examples of steel production, oil*

refineries, and automobiles, the combination of technical and organizational innovations with the scaling up of production proved an extraordinarily powerful method of cost reduction and gave a big advantage to large firms."

Technology not only aids business and economic environments, but many other domains as well. These include research environments, where data capturing and statistics aid decision making; manufacturing environments, where the product quality can be greatly improved; critical environments, such as military systems; medical environments, where research on new disease treatments can be done faster and the results kept for future purposes and even in aviation, where passenger planes, as well as the quality of training of pilots have improved greatly. Freeman and Louçã [38, p306] point out the advantages: *"As in the earlier examples of steel production, oil refineries, and automobiles, the combination of technical and organizational innovations with the scaling up of production proved an extraordinarily powerful method of cost reduction and gave a big advantage to large firms".*

MYCIN (Jackson [52]), is an example of technology advancement in the medical domain, since a knowledge based expert system was developed to aid in the diagnosis of blood infections.

Before technology can be introduced into an environment, there may be many issues which need to be addressed first. Naturally a crucially important issue is the user together with his requirements for such a device or system. If user issues are not addressed adequately, the danger exists that when the system development is completed, the system may not fully cater for the environment in which the system needs to operate. Hudlicka [48, p4] refers to the following remark often made by users after a new system was developed: *"this isn't what we asked for"*. This stresses the point that the user's requirements are not always addressed correctly.

Naturally, comprehensive requirements elicitation is not the only factor upon which the success of a project depends. It is nevertheless an important factor. There may be other factors, not addressed in this dissertation, which may also cause a project to run into problems. A factor such as communication problems among team members as described later in section 2.4, may also be the cause of flaws in the product. Brooks

[22, p11] refers to this as the "*difficulty of communication among team members, which leads to product flaws, cost overruns, [and] schedule delays*".

1.4 Hidden domain

There are numerous domains in which people work and where technology is used today. Each of these domains has a variety of factors inherent to them and it may take a person working in a specific domain many years to become an expert in it. A medical doctor, for example, who has been trained as a neurosurgeon, has to study and do research over a number of years and needs to gain practical experience over time to become a specialist. A nuclear physicist also needs to do extended study and has to work on a variety of projects in science to be able to understand the dynamics and intricacies of the domain before he or she can be regarded as an expert. In essence the domain specialist accumulates a body of tacit knowledge over a long time.

The above is true of many domains. Be it economics, physics, mathematics, astrology, aviation or any other domain, it takes patience and time for a person to become a specialist in a certain environment. An expert according to Hoffmann, Shadbolt, Burton and Klein [47], referring to Mullin [64], is someone who obtained years of experience and has professionally been qualified via graduate degrees, training experiences, publication records, memberships of professional societies and so forth.

Naturally, if a system is required for a specific environment, the most critical requirements need to be met before such an environment can be automated or simulated in some form or another. These requirements are not always easy to obtain and various techniques have been developed such as JAD (Joint Application Design) Wood et al. [93], RAD (Rapid Application Development) Schach [80] and Prototyping Olsen [71]. These techniques have been developed to aid software development teams (developers) to understand the requirements correctly and capture the essence of a system in such a manner that when the specification for the system has been done, any misunderstanding of what the system is required to do is minimized or, an ideal scenario, completely eliminated.

The following example illustrates how a simple piece of tacit knowledge residing in the mind of a client remains hidden and may have an effect on the correct functioning or not of the system under construction.

Example 1.1

An insurance system needs to be able to give a customer a quote in the shortest time possible and if the customer is a repeat customer, the system needs to automatically identify such a customer from an existing database and give him an additional discount in premium. This will aid the insurance company not only in retaining the customer, but also to do further business with the customer and in doing so, increase the revenue stream of the company. The critical requirements for this system are that the system should calculate insurance premiums quickly and check if the customer is an existing customer. Should the system fail in any of these areas, the company could forfeit business, which translates into lower profits. If the client (person that is requesting the system development) does not point out that repeat customers are given further discounts, developers might not know about this critical factor (tacit knowledge – definition 1.1), since they are not part of the client's business environment and do not know how the client's business operates. This means the system will not be designed and developed in a manner that would allow for further discounts to repeat customers. Note that the client may neglect to mention the scenario of discounts for repeat customers simply because it may appear as being **such common knowledge that everybody is aware of it.**

Unfortunately it is at the point of non-elicited requirements that many software development teams experience problems which could lead to assumptions and incorrect understanding of requirements as stated by Dix [33, p46]: "*Other errors result from incorrect understanding, or model, of a situation or system"*.

The issue of tacit knowledge residing in the hidden domain of a client has been highlighted from literature as well as the preceding discussion. It therefore raises the research question below.

1.5 Research question

A client approaching a development team to produce a software system often neglects to raise valuable tacit knowledge gained through an in-depth knowledge, gained through many years of experience working in such a domain. The research question to be dealt with in this dissertation is to determine which mechanisms may be employed to facilitate the mining of much needed tacit knowledge to aid developers in capturing the requirements of the client.

In this dissertation an answer to the above research question is approached from the angle of enhancing one or more of the existing requirements elicitation techniques as well as introducing two new methodologies. To this end it is proposed that the mechanism of a JAD workshop be augmented and that a working relationship between a client and one or more developers be defined. Augmenting a JAD workshop takes on the form of a retreat for the workshop members, called a Solitary Requirements Elicitation (SRE) exercise, developed in chapter 6. A working relationship between the two parties is defined by allowing the developer to spend some time in the environment of the client. Such working relationship is defined by a Developer-Domain-Interaction (DDI) mechanism also described in chapter 6.

1.6 Hypothesis

The above approach leads to the following hypothesis:

Hypothesis:

> The SRE and DDI methodologies have the potential to reveal some of the tacit knowledge and hidden domain factors residing in the hidden domain of a client and thereby potentially increase the success rate of a software development project.

The research methodology followed in this dissertation is discussed next.

1.7 Research methodology

The writing of this dissertation was based on research methodology which included the following (Mouton [63]):

- A research idea.
- The formulation of a research problem.
- A research proposal.
- A research process.

The **research idea** of this dissertation was formed during practical working experience and observations in different software development and business environments. The idea came to mind when, after working on the user side of a financial software product, the writer of this dissertation moved on to the software development side of a similar financial software product. He became very familiar with the way in which the business domain functioned, as well as the way in which exceptions were addressed on a daily basis, while working on the user side. This resulted in a thorough understanding of requirements of both the client and technical domains, when he was employed on the software development side of the software product. This unique combination of domain integrations that took place allowed him to foresee problems in a software product without having developed any code. It was this chain of events that initiated the idea to investigate the elicitation of requirements by a software development team and the manner in which they would ascertain whether the requirements so elicited would solve the problem at hand.

The experience referred to above proved that current methodologies being used on different software development projects do not always result in a successful project, although it is not always the shortcomings of a methodology that is to blame. Other factors such as resource constraints may also contribute to project delays and failures.

In literature the problem of the elicitation of tacit knowledge in the hidden domain of a user forms part of the client-developer problem described above. Definitions of tacit knowledge were presented in the introduction to this chapter and discussions on tacit

knowledge are presented by, amongst others Polanyi [73], Hudlicka [48] and Blandford and Rugg [14].

The **research problem** is linked to the question of why current software development methodologies often do not lead to successful projects in different business environments. The Standish Group [85] Chaos Report regularly presents statistics regarding the failure rate of software projects. Another question to be investigated is that software does not always function as expected by the client.

A **research proposal** entitled **TOWARDS THE ELICITATION OF HIDDEN DOMAIN FACTORS FROM CLIENTS AND USERS DURING THE DESIGN OF SOFTWARE SYSTEMS** was then developed. The proposal was to research current requirements elicitation and software development methodologies through a study of existing literature.

A questionnaire (attached as Appendix A) aimed at the retail sector industry was also developed, using an interview approach. (Kujala, Kauppinen, Lehtola and Kojo [56])

It was also proposed that, should it be possible to determine why current software development methods fail, enhanced or new requirements elicitation techniques or software development methodologies with the potential to alleviate the shortcomings of existing techniques and methodologies, would be suggested.

A **research process** followed, incorporating the following:

- A literature review of previous as well as current requirement elicitation techniques and software development methodologies.
- The drawing up of a questionnaire and distributing it electronically to a target business domain.
- A case study was extracted from literature and an illustrative scenario created to study various elicitation techniques.

The writing of this dissertation, including all the research done - amongst others the introduction of two new methodologies (refer to sections 1.5 and 1.6) – followed.

1.8 Referencing style

The referencing style used in this dissertation is a blend between a numeric and an alphabetic style, allowing for a reference to be found via a number as well as an author name. This style stems from the Natbib style of *LaTeX*. Further details may be found at http: **//www.ctan.org/tex-archive/help/Catalogue/entries/natbib.html**.

1.9 Layout of dissertation

Chapter 2 which follows contains a description of the problem of correct requirements elicitation for a system before the development team embarks on any specification, design and development phases.

A number of existing requirement elicitation techniques used for software development is described in chapter 3. Each one of these techniques is covered in more detail. Their strengths and weaknesses are also considered.

Software development life cycle models are presented in chapter 4, focussing on the unique characteristics of each model. Among these are the Waterfall, Incremental (including Agile), Rapid Prototyping, Spiral and V-Process models.

In chapter 5 a case study extracted from literature is presented as a vehicle to illustrate the problem of incorrect requirements elicitation. An illustrative scenario is created to show how existing requirements elicitation techniques do not always succeed in gaining access to the tacit knowledge in hidden domains.

Two new methodologies, namely, SRE (Solitary Requirements Elicitation) and DDI (Developer Domain Interaction) are discussed chapter 6. These are then used to describe the process of correct requirements elicitation. If the methodologies are

applied as comprehensively as possible, especially in complex environments, the success rate of a software development project may be enhanced.

Chapter 7 ends the dissertation by revisiting the hypothesis and research question, providing a summary, giving a brief description of the SRE and DDI methodologies, referring to possible future work and reaching a conclusion.

A list of references is provided after the final chapter.
The following appendices are attached.

- Appendix A contains the questionnaire results from a retail sector of the economy.

- Appendix B contains Z schemas which relate to the case study in chapter 5.

- Appendix C is the refereed paper that was published at INSITE 2007, held in Lubljana, Slovenia.

Figure 1.1 illustrates the logical flow of this dissertation. Each block displaying the major points covered in each chapter.

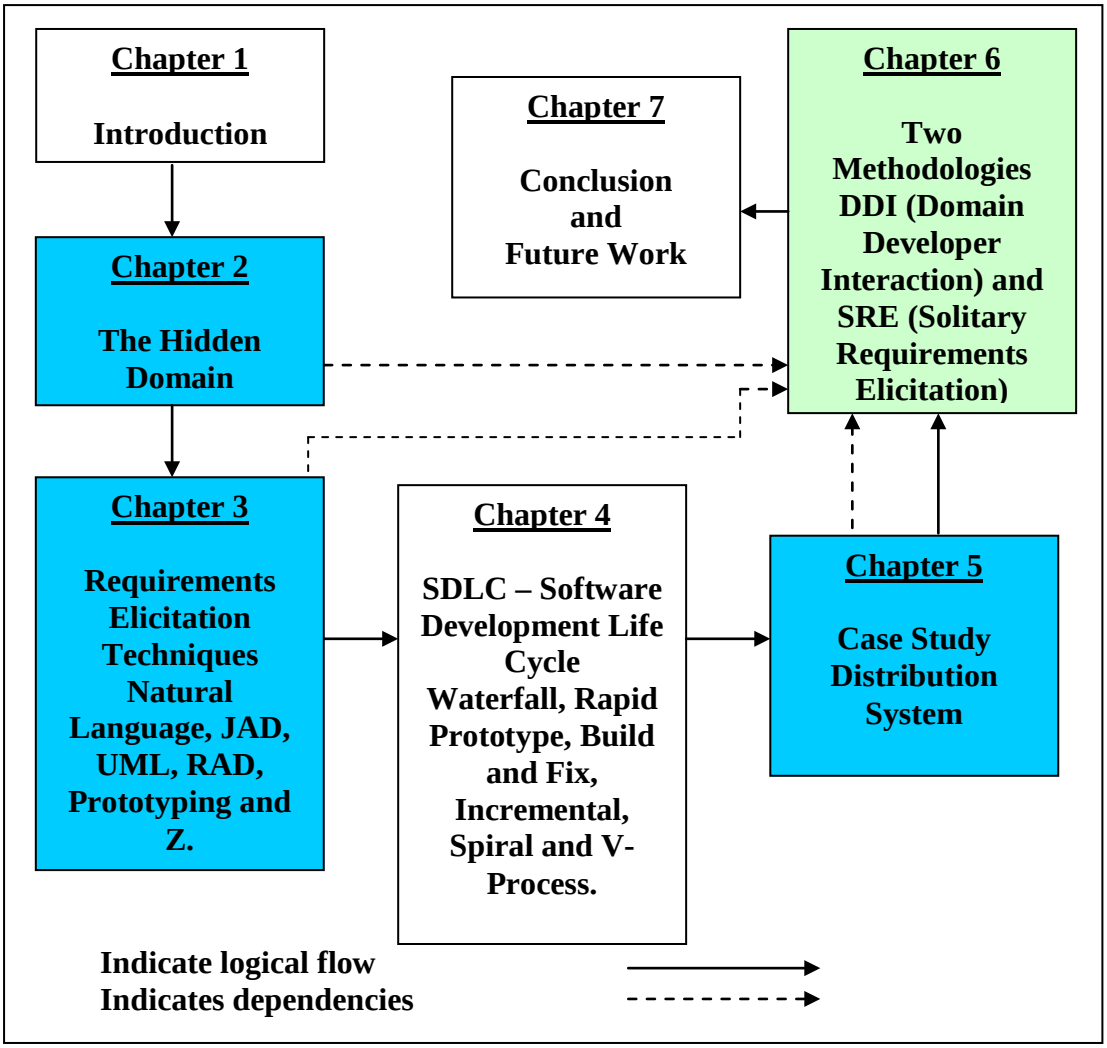


Figure 1.1 Dissertation outline.

Solid arrows in figure 1.1 indicate the logical flow of chapters including descriptions of different methodologies and techniques used to develop software. Dashed arrow indicates dependencies. The hidden domain described in chapter 2 may be elicited via the new methods dealt with in chapter 6. The contents of chapter 5 will be influenced by the hidden factors elicited. The requirements elicitation techniques described in chapter 3 may be influenced by the application of the new methods described in chapter 6.

Chapter 2
The Hidden Domain

2.1 Analysis of hidden domain

In chapter 1 the problem of gaining access to tacit knowledge in the hidden domain of the client was introduced. It was argued that gaining such access has the potential to increase the success rate of a project, because the requirements for a system could then potentially be better understood than would have been the case if sufficient access to the hidden domain had not been gained.

The problem of gaining access to the hidden domain is further elaborated upon in this chapter. The language of mathematical Venn diagrams is used to illustrate the concept of domains visually. Visual interpretation of domains may facilitate an understanding of the way in which different domains integrate and how a hidden domain is created.

This chapter also introduces different organisational frameworks that exist in software development domains, as well as the dynamics within these frameworks. The communication gap between a client and a software development team is described and an introduction to the SDLC (Software Development Life Cycle) is given.

2.2 Creation of hidden domain

A hidden domain of a client is created through experience, which is what is gained over time by becoming an expert in a certain domain. (Refer to section 1.4). When a person, such as a scientist, establishes a new finding or creates a new theory after many years of investigation, study and tests, he would be considered to be a specialist, because he is an expert in his known domain.

Hudlicka [48, p4] states: "*In other words, neither experts nor the intended system users are always able to state their knowledge and system requirements succinctly in*

response to direct questions." This statement relates directly to the fact that domain experts often find it difficult to express or articulate what they know to another individual who does not have the same background or knowledge.

Related to this phenomenon is the claim made by the father of the concept **tacit knowledge**, Polanyi [73, p466] when he wrote: "*…knowing of more than you can tell.*". These quotations hint at a hidden domain which could be depicted as in figure 2.1.

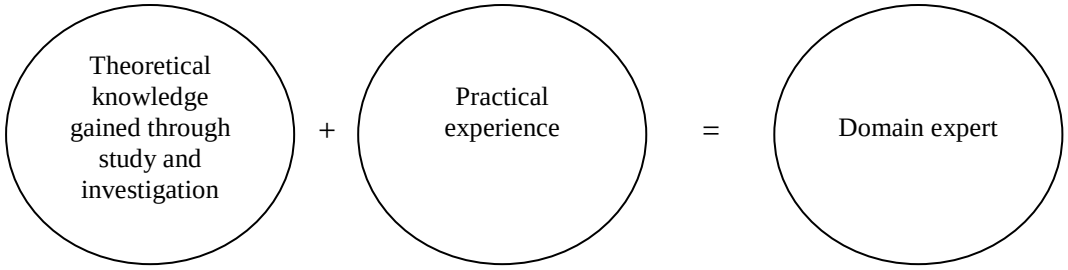


Figure 2.1 Graphic representation of domain expert.

Venn diagrams are used as a vehicle to model the concept of domains and demonstrate how they integrate visually. Venn diagrams are also used later in the chapter to highlight the differences between the domains of a software development team and the client.

2.3 Introduction to Venn diagrams

Venn diagrams are traditionally used to add visual clarity to the results of operations in mathematical set theory (Enderton [35], Labuschagne [57]). In what follows below, the well-known set-theoretic definitions of union $(\cup)$ and intersection $(\cap)$ are described. The reason for describing these two operations is simply because the operations normally involving domains are those of union and intersection.

Examples of set-theoretic union and intersection are given below. There are many more definitions and operations relating to sets,and the reader is referred to Labuschagne [57] for more information.

Example 2.1

Consider the following definitions for sets X, Y and Z:

$X = \{x \mid 1 \leq x \leq 6\}$, which is the set of positive integers less than or equal to six.
$Y = \{x \mid 4 \leq x \leq 9\}$, which is the set of positive integers between four and nine inclusive.
$Z = \{x \mid 1 \leq x \leq 4\}$, which is a set of positive integers less than or equal to four.

In list notation the above sets are given by:

X = {1, 2, 3, 4, 5, 6}, Y = {4, 5, 6, 7, 8, 9} and Z = {1, 2, 3, 4}.

The above three sets are illustrated via Venn diagrams in figures 2.2a, 2.2b and 2.2c respectively.

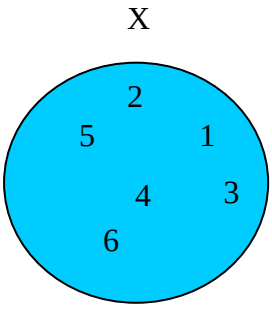


Figure 2.2a, Set X.

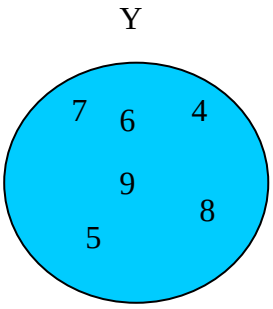


Figure 2.2b, Set Y.

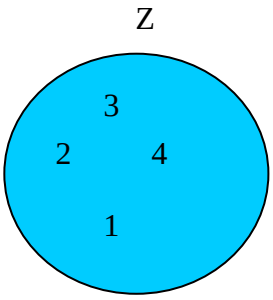


Figure 2.2c, Set Z.

Combining (taking the union of) two sets X and Y, one would obtain a new set with all the elements in X or in Y or in both.

$X \cup Y = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Elements 4, 5 and 6 are in both X and Y. The elements 1, 2, 3 belong to X only, while the elements 7, 8, 9 are only in set Y. The order of the elements in a set is unimportant and neither do they repeat. The union, i.e. $X \cup Y$, consists of all the elements, i.e. the whole of figure 2.2d.

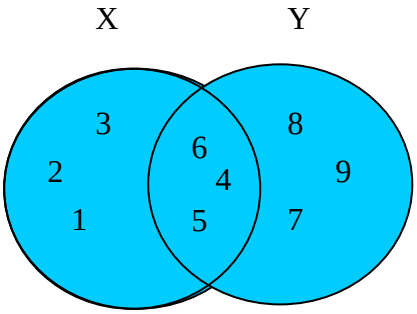


Figure 2.2d Venn diagram illustrating X ∪ Y.

Combining three sets, X, Y and Z, one would obtain a new set with all of the elements of X, Y and Z:

X ∪ Y ∪ Z = {1, 2, 3, 4, 5, 6, 7, 8, 9}.

Again elements 1, 2, 3, 4, 5 and 6 are in the union of X and Y. The elements do not repeat.
Elements 5 and 6 are in both X and Y.
Element 4 are in all three sets, X, Y and Z.
Elements 1, 2, 3 are only in X and Z.
Elements 7, 8, 9 are only in set Y.

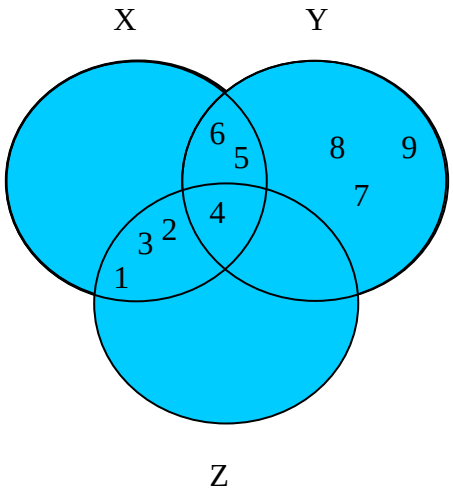


Figure 2.2e Venn diagram illustrating X ∪ Y ∪ Z.

It is also possible to determine which elements belong to both sets. The operation to calculate this is set-theoretic intersection. Therefore, the elements belonging to both X and Y is indicated by $X \cap Y$.

For $X = \{1, 2, 3, 4, 5, 6\}$ and $Y = \{4, 5, 6, 7, 8, 9\}$, one has $X \cap Y = \{4, 5, 6\}$. This set is indicated by elements 4, 5 and 6 in figure 2.2f.

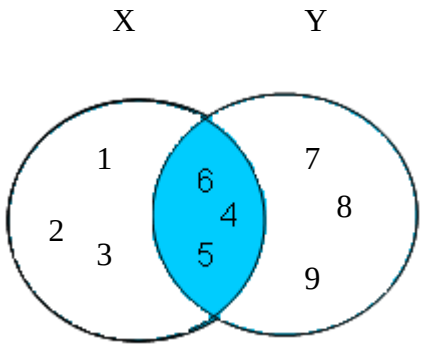


Figure 2.2f Venn diagram illustrating $X \cap Y$.

It is also possible to determine which elements occur in all three sets. To do this one would write, $X \cap Y \cap Z$ for all three sets.

As before, if $X = \{1, 2, 3, 4, 5, 6\}$, $Y = \{4, 5, 6, 7, 8, 9\}$ and $Z = \{1, 2, 3, 4\}$ then $X \cap Y \cap Z = \{4\}$. In Venn diagram notation the intersection is the area where all three circles overlap, i.e. element 4 in figure 2.2g.

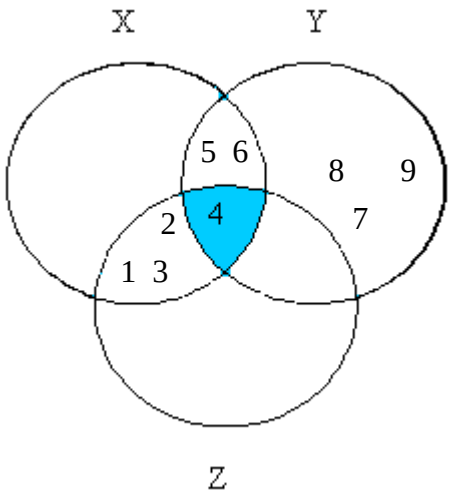


Figure 2.2g Venn diagram illustrating $X \cap Y \cap Z$.

There are two further set-theoretic operations which sometimes appear in specification work and these are the **arbitrary** union and intersection. (Enderton [35]). An arbitrary union is a distributed operation that takes the union of any number of sets. In this dissertation an arbitrary union is indicated by (∪).

For example: W = {{1,2,3}, {3,4}, {3,6,7}}, i.e. W is a set containing a number of other sets (3 in this case). The arbitrary union of W, indicated by (∪) is obtained by taking the union of the three inner sets. Therefore ∪W = {1,2,3} ∪ {3,4} ∪ {3,6,7} = {1,2,3,4,6,7}. There is a corresponding operation for taking a distributed intersection, indicated by (∩), of any number of sets. For the set W above ∩W = {1,2,3} ∩ {3,4} ∩ {3,6,7} = {3}. The SRE methodology defined in chapter 6 makes use of an arbitrary union.

The next section provides a description of the way in which software development teams function and how communication channels among members of a software development team integrate in a software development environment. The different roles of members in a typical software development team were defined in chapter 1.

2.4 Functioning of software development teams

In many software development companies the working environment is similar to the following:

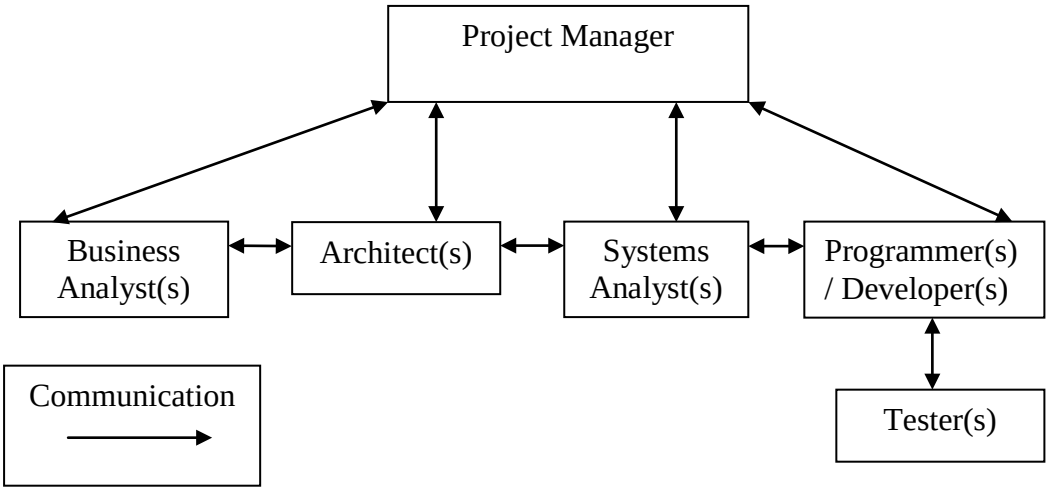


Figure 2.3 Typical software development team.

The double arrows in figure 2.3 indicate two-way communication channels between the relevant parties. For example, an architect has a two-way communication channel with each of the business analyst, the project manager and the systems analyst. Note that there may be more than one individual in each of these positions, hence the plural options in figure 2.3.

In many situations the same individual performs more than one role, for instance the architect may also perform the role of the business analyst as well as that of the systems analyst. Ko [55, p7] states that "*the information requirements gatherers (who were also developers) were understanding the process more than any one individual in the organization*".

Figure 2.3 displays the many different roles that may exist in a software development organisation, either internally or externally. There are also many communication channels that may exist between each of the roles (indicated by arrows) and if the requirements for a product are not clearly stipulated each person often interprets requirements in his own manner. Dix [33, p46], stipulates: "*Other errors result from incorrect understanding, or model, of a situation or system. People build their own theories to understand the causal behaviour of systems."* This claim by Dix relates to how software development teams could understand requirements for a system incorrectly.

2.5 Understanding requirements of a new system

When a client approaches a software development organisation, he meets with as many of the role players as displayed in figure 2.3 as possible. The client already has requirements for a product and usually cannot purchase suitable software which has already been developed. Very often then, the software product that the client needs has to be developed for his specific business requirements.

There are various methods which may be employed by a development team to elicit the software requirements from a client. These include:

- Natural Language, (Bowen [21]).
- JAD (Joint Application Design) Workshops, (Wood [93]).
- UML (Unified Modelling Language), (Jalloul [53]).
- RAD (Rapid Application Development) Tools, (Schach [80]).
- Prototyping, (Olsen [71]).
- Formal Method e.g. Z, (Lightfoot [58]).

The above techniques are introduced below and are covered in more detail in chapter 3.

2.5.1 Natural Language

A consultation is a process in which the development team analyses the client's business requirements through questionnaires and personal interviews with the client and his personnel. Users (people tasked to actually work with the system) of the new system, may provide insight into how their daily job routine functions are carried out. The individual driving this process would be a business analyst. (Refer to figure 2.3), because it is the responsibility of such an individual to determine the exact business functionality and requirements of the client. (Refer to definition 1.2). The business analyst would then provide input into how the new system would enhance the client's business environment, focussing on cost and efficiency benefits. An example of the type of question that could be asked in such a scenario by using a questionnaire is:

"How would you aid an IT expert/Developer to understand your business environment and how would you test if this person does understand your business model?" (Questionnaire sent out by writer of this dissertation to various retail outlets, attached as Appendix A)

In some software development companies all team members will be involved during the consultative process. A team working in such a manner would gain valuable experience and insight into how the client operates his business and following such an approach could enhance the success of a project.

The consultative process, however, presents a weakness in that users and personnel in the service of the client are usually not technically inclined and might only be able to answer questions regarding their own working domain. Typically, team members would only get answers to questions they actually ask. If no questions are asked about extreme situations and the user does not highlight these scenarios to the team member, there is a potential that critical factors, i.e. much needed tacit knowledge, are not highlighted. These factors not highlighted will therefore not be incorporated when all the feedback is processed.

2.5.2 JAD

A JAD workshop is similar to a consultative process. The difference is that it is usually done with as many of the software development team members as possible (figure 2.3) and usually the client and only a few users (people who will actually work with the system) from the client's organisation, depending on the size of the client's organisation. In the JAD workshop brainstorming sessions in which ideas are highlighted and debated are held, getting input from all members present (Wood et al. [93]). The weakness of JAD is that the software development team can only include information offered by the client and the users during the session. Specific scenarios, if not mentioned by the users or not probed by the development team, could potentially not be provided for. Hence, important tacit knowledge may remain hidden.

2.5.3 UML

UML Use Case models are sometimes used in JAD sessions. While developers discuss scenarios with the client, they can be modelled using Use Case models, which are illustrations indicating what operations need to be followed by the system. An example of a Use Case model for a banking system in which a user deletes an account is given in figure 2.4. The weakness in the Use Case models of UML is similar to that of JAD. If a scenario is not highlighted, a Use Case model would not be developed and the scenario would thus not be catered for by the system. Again, therefore, tacit knowledge may remain hidden.

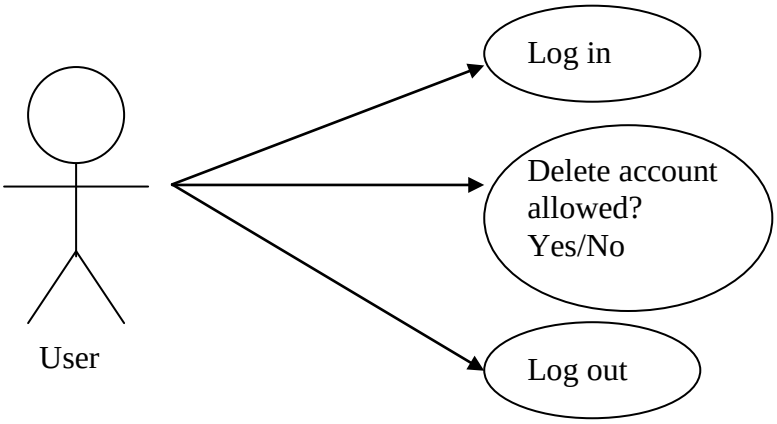


Figure 2.4 Example of Use Case model.

2.5.4 RAD

JAD workshops may also make use of RAD tools, which allow for greater visualisation of the product. RAD is mainly used to illustrate how the application would visually display to a user. (An example of a RAD user interface is given in figure 3.3, section 3.3.3.3). Users and developers are enabled to form a clearer picture as to how the product looks. Again, however, if the user does not delve into domain-specific knowledge or highlight specifics, provision would not be made for them.

Prototypes (discussed below and more comprehensively covered in chapter 3), are also used in JAD workshops and this may help to increase the understanding of the requirements for the product. Prototypes may be paper based, sketches on a board or even a RAD application. One of the main aims of a prototype is to facilitate in determining the requirements of the proposed system. When users view something tangible, they begin to realise that some specifics could potentially have been omitted and they could then raise those specifics to be included in the product. Only then would developers know about specific scenarios that need to be catered for. If users do not highlight these situations (i.e. examine their tacit knowledge), developers would remain unaware of them and the product would potentially not make provision for them.

2.5.5 Prototyping

Prototyping is a requirements elicitation method used to capture the essence of a product, without any added functionality or any extras like printing options. The aim of a prototype is to capture the core requirements of a system, via paper or electronic means and to aid in the understanding of those requirements. If the requirements are correctly captured and the client is satisfied that the system will functionally be able to do what is required, the complete system can be developed.

One of the risks of a prototype is that it can become the product itself (Schach [79]). This is generally known as an evolutionary prototype (Hughes and Cotterell [49]). The use of an evolutionary prototype may be problematic since factors such as flexibility and processing speed could have been compromised while the prototype was being developed. Should the prototype become the product, procedures such as quick enhancements to the system could be a problem, since the prototype was not developed to be flexible enough to cater for such requirements. Nevertheless, Hughes and Cotterell [49] present the evolutionary prototype as a valid development tool.

2.5.6 Formal Method Z

Formal methods like Z (Lightfoot [58], Potter et al. [74]) are methodologies using mathematical notation, which aid a software development team when a specification needs to be developed for a software system. An advantage of using a formal method is that the ambiguity of Natural Language specifications could be reduced to a minimum. Formal methods are therefore useful where precision and mathematical rigour is at stake. A disadvantage of formal methods is that not many software developers are familiar with mathematical formalism. Formal specifications can become quite complex and could be difficult to use. This sentiment is echoed by Hall [44, p3]: "*This is clearly a challenge: current formal notations are notoriously opaque, and formal methods tools are almost all hard to use.*"

A graphic layout of the way in which the requirements elicitation techniques described above may follow upon each other in different situations and scenarios and possibly be integrated into a larger methodology is illustrated in figure 2.5.

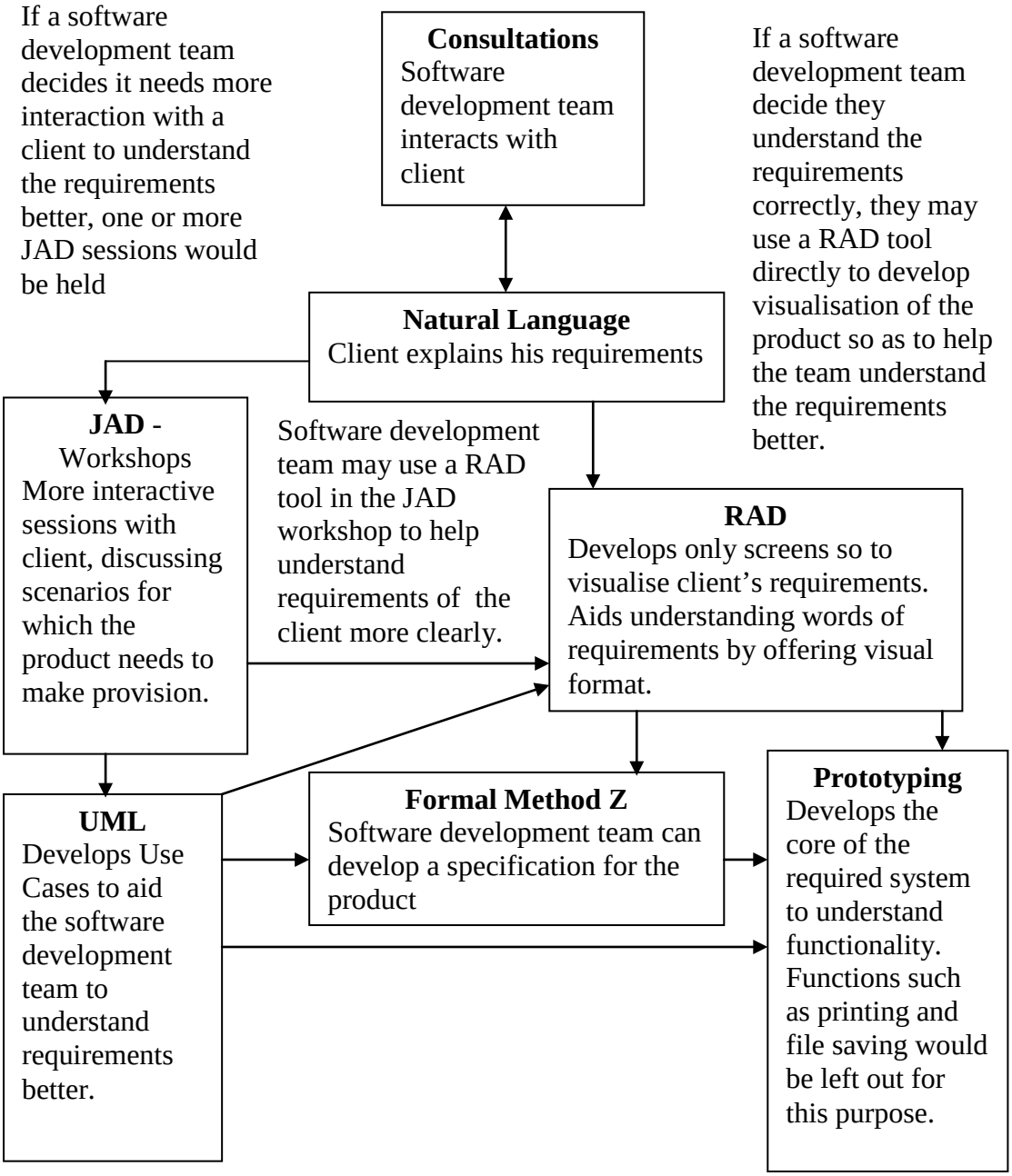


Figure 2.5 Integration of different requirement elicitation techniques.

In the following section the issue of a communication gap as related to the mining of tacit knowledge is discussed. This gap is a result of the differences in the understanding of requirements between software development teams and clients. These differences may be as a result of different terminological backgrounds, i.e. the existence of different knowledge domains not overlapping sufficiently.

2.6 Communication gap

Since clients and development teams often speak different terminological languages, there appears to be a communication gap between what software development teams understand and what clients need, but fail to ask for precisely. This phenomenon is articulated by Schach [79, p33] as:

"*I know that this is what I asked for, but it isn't really what I wanted.*"

The communication gap is created because users who are not technically literate may not understand technical questions and hence answer incorrectly. The same problem presents itself in the software development team, because the members do not work directly in the user's environment and hence might not always understand phrases and terms the user refers to when answering questions or highlighting information. Critical tacit knowledge in both the domains of the client and the development team, therefore, remains hidden.

In figure 2.6 the communication gap is illustrated. The double arrows indicate communication channels between the individuals in the software development team and the client and users. (Refer to figure 2.3, setting out the internal communication channels between the software development team.)

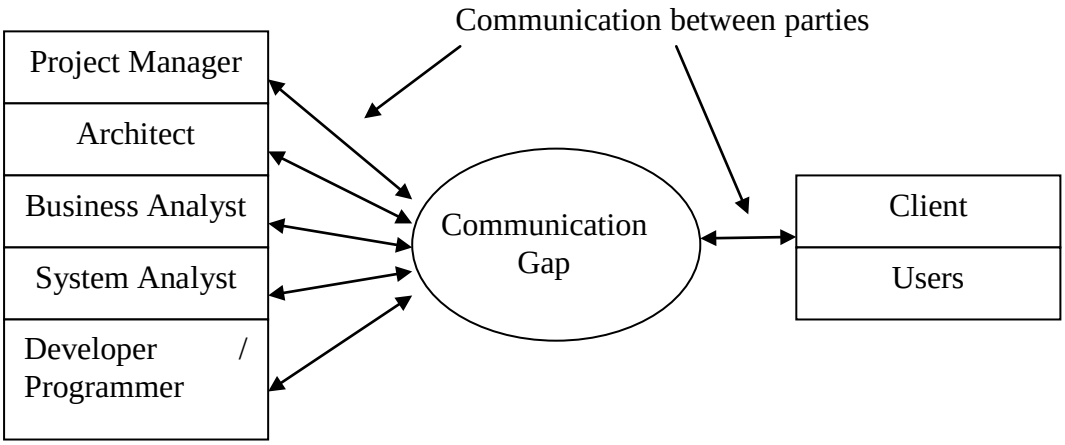


Figure 2.6 Communication gap among stakeholders.

The denotation **Client** in figure 2.6 refers to the person or company requesting the product. (Refer to the general definition in chapter 1). **User** in the figure refers to individuals who will actually work with the application, e.g. they may be employed by the client. The software development team may experience a communication gap with both the client and the users. Usually, users are interviewed after the client has been interviewed, because they will work with the system on a daily basis.

In any JAD session the client and the users may all be present or any combination of client and users may be present, all depending on what is required of the session. The aim of a JAD session is to gather as much information as possible to understand what needs to be developed. Note that although in this dissertation clients and users are normally different groupings, in practice clients and users may be the same individual(s), or may be different individual(s). It depends on whether the client will be paying for and using the system himself or whether he pays for the development but employees in his organisation will be using the system.

Next the roles fulfilled by each of the members of the development team in figure 2.6 are elaborated on in line with the definitions in section 1.1. The discussion presented below is taken mostly from Schach [78], Schach [79] and Schach [80].

The **project manager** provides a project plan and budgets for sufficient resources to complete the task at hand. One problem caused by many project managers is that, should there not be enough resources at any point in time during the project, they add more personnel to the project hoping that the project can still be completed on time. Adding additional resources to a project does not always enhance the speed of delivery; it usually slows down already productive members. New resources need to be brought up to date with the current status of the project and they also need to understand how they can add value before they can start to be productive. This learning curve through which new members of the development team need to go takes some time and could influence the already productive members negatively.

According to Sukhoo, Barnard, Eloff, van der Poll and Motah [87] effective communication is a soft skill which is very important for project managers to have, because if communication is not clear in a team environment it could cause

misunderstandings. These kinds of misunderstandings may be interpreted as another form of tacit knowledge not being brought to light.

The **architect's** role is usually that of having a more holistic view of the overall solution, looking at how the complete solution needs to be integrated to make both front end and back end processing possible for the client's business.

Some **business analysts** have some technical background, but cannot always assist in both the technical as well as the business requirements when analysing a client's business.

The main function of the **system analyst** is to look at the system requirements and how the system will need to perform to reach the desired output. The system analyst and architect usually work together very closely (or may even sometimes be the same individual) to help ensure that the performance of the system would be adequate for the environment in which it needs to operate.

Programmers/Developers are more technically orientated and usually develop software products according to specifications. These individuals are normally proficient in at least one programming language and are, therefore, sometimes referred to as coders of systems.

Together all the role players above form a software development team and one could refer to them collectively as **IT**. Client and users on the other side of the communications gap could be referred to as **business**. These denotations, namely, IT and business for a development team and client/users respectively are used in the discussion that follows next.

An international private company (BMC Software) did a research study highlighting the fact that many projects fail because of a communication gap. (Refer to figure 2.6) between business and IT, resulting in a misalignment of their combined goals. Hence the unveiling of tacit knowledge on both sides of the spectrum is neglected.

"A communication gap appears to be at the heart of the problem, with SA being among the worst offenders," says Brian Whittaker, UKMEA BMC MD. "Leadership is an issue because CEOs are not getting the message across to IT" (ITWEB [51]).

The study was done over a wide spectrum namely (BMC Software [15]):

- Manufacturing.
- Finance.
- Utilities.
- Retail/Distribution.
- Public Sector.
- Professional Services.

The BMC study showed that each of the above sectors followed the same pattern, that of information technology projects failing as a result of a communication gap that exists between business and IT.

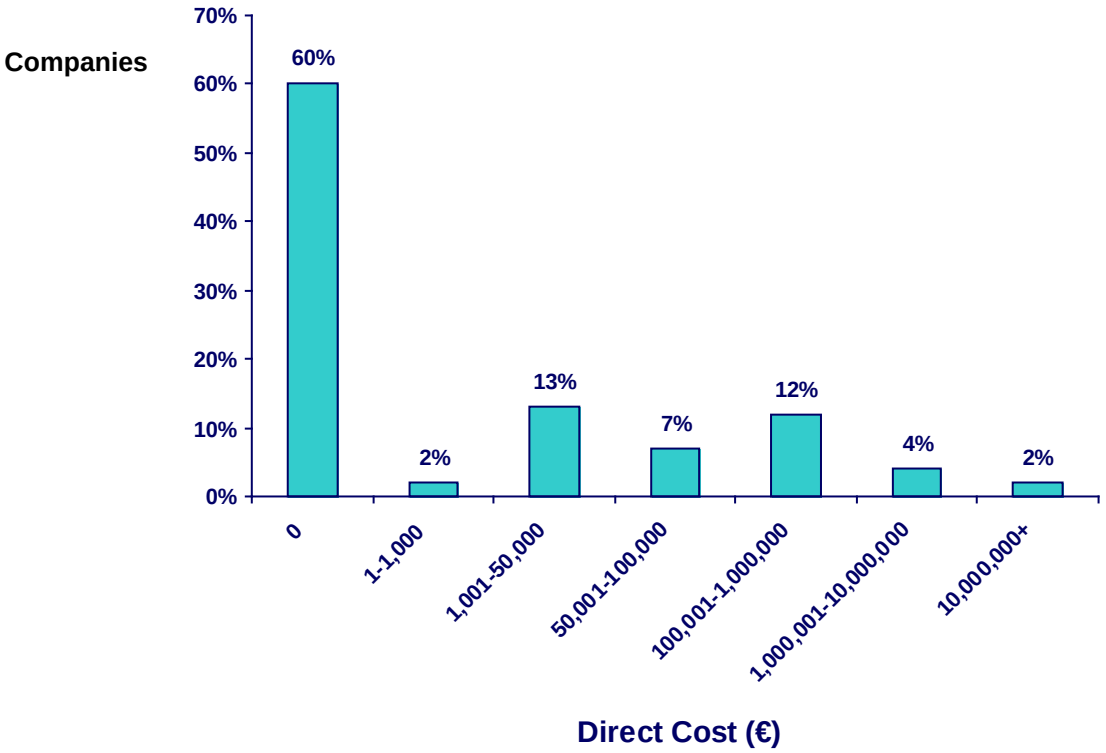


Figure 2.7 Total direct cost for information technology failures. (BMC Software)

Figure 2.7 indicates that although 60% of companies surveyed reported no losses, 12% of companies surveyed reported losses of between 100,000 and 1,000,000 Euros and 4% of companies in the survey reported losses of between 1,000,000 and 10,000,000 Euros as a direct result of information technology failures.

It is plausible that the above communication gap originated at least partially because of necessary tacit knowledge in the hidden domain of (mostly) the clients not being mined effectively by members of the development team. To this end two methodologies aimed at narrowing a communication gap are developed in chapter 6. The writer of this dissertation trusts that the successful application of these new techniques may result in fewer projects failing and could lead to savings for companies. The main focus of the new methodologies is to allow business and IT to move closer together and to allow both sides of the tacit-knowledge equation to start communicating in such a manner that each side may achieve its objectives. For business it could be how to enhance productivity, how to increase revenue and customer satisfaction. For IT the objective could be how to enable the business to achieve its objective by improving the functionality of the system and processing speed. Development time may be shortened through improved requirements elicitation procedures, leading to better customer retention and satisfaction.

In section 2.7 which follows next, an introduction to the Software Development Life Cycle (SDLC) is given, with emphasis on the first step in any SDLC methodology, namely requirements and analysis.

2.7 Introduction to SDLC

The software development life cycle (SDLC) is a documented process which traditionally consists of the following phases (Schach [79], Charette [25]):

- Requirements and Analysis.
- Specification.
- Design.
- Implementation.

- Testing (included in Verification and Validation).
- Maintenance.

The **requirements** of the proposed system such as which problem the system needs to solve or which business value would be introduced into an organisation by the system, are determined. Requirements analysis and elicitation – as stressed throughout this dissertation - is a crucial step in the SDLC.

The next step is that a **specification** outlining the functionality of the product should be developed. The development team creates the specification document from the requirements which they have already determined. The specification may take on many possible forms: a Natural Language version (often viewed as informal), a semi-formal version (e.g. UML) or a formal-methods version (e.g. Z).

Following the specification phase is the **design** of the product in the software development life cycle and this involves the development of the layout of the product and the physical design of how the software would be built. An example of this is a class diagram (refer figure 5.2).

Implementation follows upon design and the development team should decide how the coding for the product should be implemented, keeping in mind that the tasks need to be done in collaboration with each other. Each developer in the team may be assigned a certain section of the product to develop. This may enhance productivity and speed up delivery, but it does not mean developers should work in isolation from one another. Developers may still work together as a team, sharing knowledge and ideas, thereby enhancing a collective overlap of their individual tacit-knowledge domains.

Many software companies follow a model (refer to section 4.2.4 Extreme Programming) in which a developer works together with other developers and in close association with testers of the software product. The role of testers is to point out possible discrepancies in the product to the developers. In XP the developers and the testers may be the same individuals.

Testing is an extremely important activity in the software industry and is one aspect of a much wider topic often referred to as **verification** and **validation** (V&V) (Pressman [75]). Verification embodies a set of activities to determine whether a piece of software implements a particular function correctly or not. Validation is made up of another set of activities to determine whether a piece of software has been developed in line with the requirements of the client or not. V&V were originally clarified by Boehm [16, p37] as:

- Verification: "*Am I building the product right?*"
- Validation: "*Am I building the right product?*"

While the wider activities encompassed by V&V, e.g. formal technical reviews (Pressman [75]) are vitally important, it is the experience of the author of this dissertation that the majority of V&V activity in industry manifests itself in the form of software **testing.** The remainder of this section will therefore be devoted to testing procedures.

Testers of software ought to test the product according to a specifications document. If the product presents with any errors, testers will refer it back to developers for investigation. Developers will investigate and make sure that there is indeed a problem before any correction is made.

Software testers often make use of **white** and **black** box testing when testing a software program. White and black box testing in essence entail the following. (Discussion synthesised from: Black Box and White Box Testing compared [13]):

White box testing may be viewed as a **verification** procedure in the sense that it is concerned with testing the software product for correctness; it does not consider whether the complete specification has been implemented or not. **Black box** testing is concerned with testing the specification; it does not consider whether all parts of the implementation have been tested (i.e. verified). Therefore black box testing is testing software against the specification and is aimed at discovering **faults of omission**,

indicating that part of the specification has not been fulfilled. Hence black box testing is a **validation** exercise.

White box testing tests against implementation and is aimed at discovering **faults of commission**, indicating that part of the implementation is faulty. In order to test a software product comprehensively, both black and white box testing need to be done.

Beizer [10, p8] points out that black box testing is also known as behavioural testing and that white box testing is also called structural testing. Black box testing does not focus on the internal structure of the product but rather on the functional requirements of the product. White box testing focuses more on the structure, which requires access to the source code according to Beizer [10, p8]. He [p11] also highlights the point that black box testing is not the be all and end all test method, but that more than one technique should be used, i.e. further V&V procedures ought to be employed.

The final phase after testing is the **maintenance** phase in which the product is changed as the requirements of the client or the environment in which the product is functioning changes. This involves product updates. Every time a change is made to the product it needs to be tested (verified and validated) first before released into the live working environment.

The above discussion is but a brief overview of how the traditional SDLC functions. For more information the reader is referred to chapter 4 in which the SDLC is discussed in more detail and which also focuses on several models, including the following:

- The Waterfall model.
- The Spiral model.
- The Incremental model.
- The Build and Fix model.
- The Rapid Prototyping model.
- The V – Process model.

The example below illustrates the way in which the communication gap introduced above may have an impact on a software development project.

Example 2.1

A client approaches a software development organisation to develop a software program that should be able to diagnose blood infections. This is a condition which, if correctly identified, has the potential to be treated, provided that a method of treatment or medication exists for the particular disease causing the blood infection.

One of the requirements is that the system must prompt the user with questions and then, based on answers given by the user, be able to determine a diagnosis.

During the requirements elicitation process, be it via Consultations (Natural Language), JAD, RAD, Z, UML or Prototypes or a combination of these, the client raises a specific question about the shape of the organism causing the infection when viewed under a microscope.

What is the shape of the organism when viewed under the microscope?

1) Round
2) Elliptical
3) Oval
4) Star shaped

Developers include this requirement as one of the questions and then continue with the requirements elicitation process. This is exactly where there could potentially be an oversight without developers knowing. If the client does not reveal to them that if the organism is **round and red** a particular set of questions should be prompted, the result will be that the software product will only prompt the user if the organism is round and then continue with a different (standard) set of questions. This lack of tacit information could lead to a system which makes a completely wrong diagnosis. The result of such incomplete information could lead to a physician prescribing incorrect

medication, thus not destroying the infection in the patient's blood and this could lead to death.

The above is a simple example of how to partly diagnose Myelodysplastic Syndromes, where the shape and colour of cells viewed under a microscope are vitally important (American Cancer Society [3]).

If the domains of the software development team and that of the client differ substantially, it could result in a system that could be without substance. Unless the client reveals all hidden domain factors, and any tacit knowledge that has not yet been revealed, to the software development team, the system could be dysfunctional. Unfortunately the client does not always think of all scenarios as a software development team would, because he is usually not as technically inclined as members of the team would be. This may result in his thinking enough information has been elicited. This is when, after the product has been developed, the client sometimes utters the following words as stated by Schach [79, p33] "*I know that this is what I asked for, but it isn't really what I wanted*".

The reason for the statement above becomes clearer once the system has been commissioned. Only then clients become aware that the system is sometimes not fully functional in their daily working environment, because it does not always function according to common daily practices. This may cause users not to accept the system and not to use it, or to keep on asking for enhancements to the system in order to bring it more in line with daily practices.

In the following section the differences between the knowledge and skills domains of a client (an expert in a specific application domain) and a software development team (experts in software engineering – refer to definition 1.7) are revisited in terms of the Venn diagram notation presented in section 2.3.

2.8 Domain differences between client and software development team

When a new product needs to be developed the requirements for that system comes from the client and users who will actually use the system. Access to tacit knowledge in the hidden domain of the client is one of the barriers that software development teams need to address or they might end up with a product that is functionally incorrect for the target environment.

Venn diagrams introduced in section 2.3 are used to describe how domains between client and developers integrate and differ. Definition 1.6 indicates that the denotation **Developers** should be understood to mean the software development team, of which the business analyst, architect, systems analyst and programmer are all part. It is ultimately the programmer who will do the actual coding of the product, so it is very important that he understands what is involved. The programmer reads the specification and studies the design put forward by the systems analyst before coding a solution. Note that a development team may have various individuals, each with his function as described before or it could be one individual that performs most or all of the functions. The requirements for a system need to be understood correctly, be it by either a team or a single individual, to create a system that will function as envisaged by the client.

An example of how the total knowledge and skills domains of one client and two members of the software development team may overlap is shown in figure 2.8.

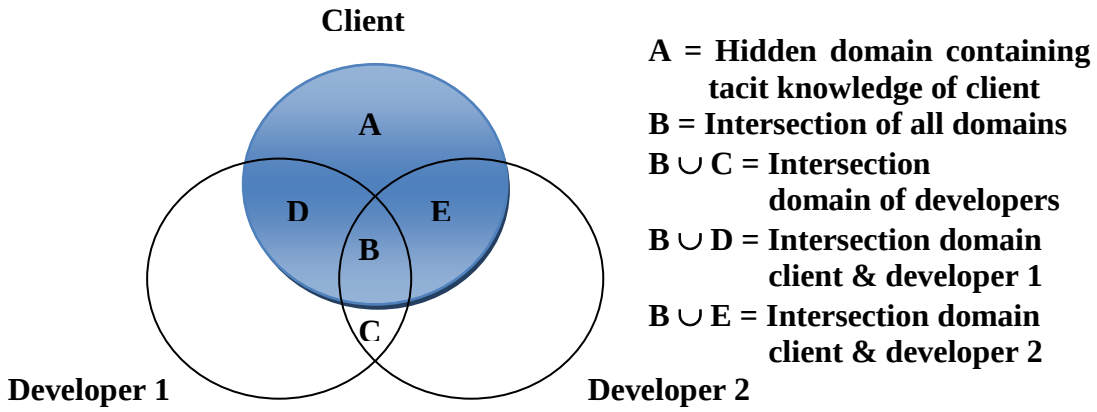


Figure 2.8 Venn diagram indicating different domains.

Figure 2.8 illustrates the domain (labelled **A**) that resides solely within the client. It is this domain that contains the tacit knowledge of the client and also the domain that needs to be understood better by a software development team so that the potential success rate of the project may be increased.

The above example is for two developers only. In practice it could be any number greater than or equal to one, because it can be more than one person fulfilling the software development function or it can be one person only. The problem of hidden domain elicitation may potentially be larger if more people are involved, because more domains come into play. Naturally, current requirements elicitation techniques attempt to integrate (overlap) the different domains as much as possible, i.e. increase the size of **B** and decrease the size of **A**.

Each diagram from figure 2.9a to 2.9c indicates that the smaller the hidden domain, the better the software development team may understand the domain and therefore the requirements the system needs to fulfil.

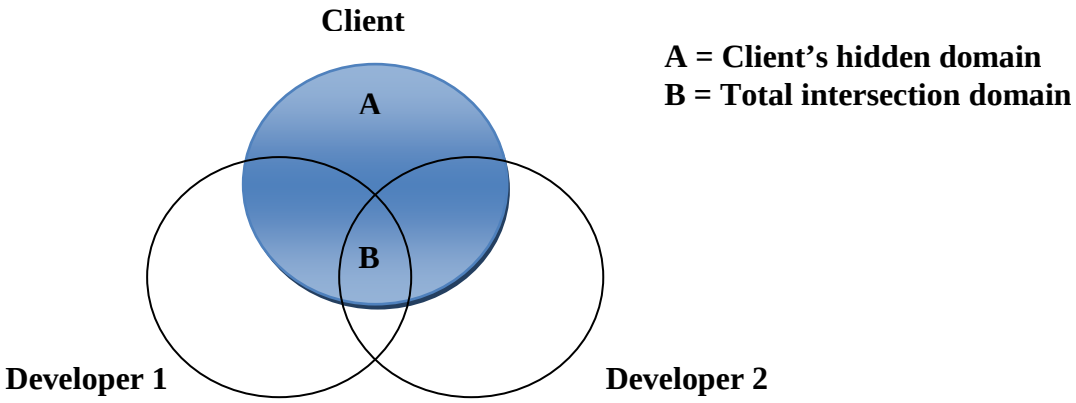


Figure 2.9a Intersection of domains of developer(s) and client.

If the knowledge domains of the two sides (client's domain and software development team domain) do not overlap sufficiently, the system might not take into account enough tacit knowledge and hidden factors as illustrated in figure 2.9a, area **A**.

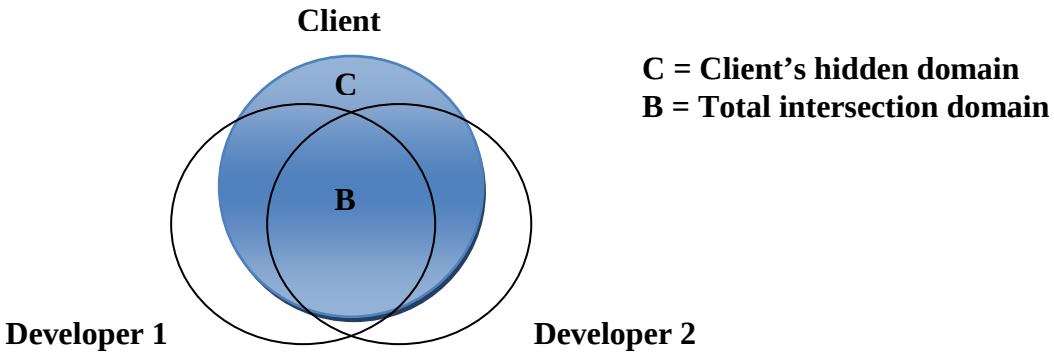


Figure 2.9b Increased intersection of domains of developer(s) and client.

If the relevant domains increasingly overlap as more tacit knowledge and hidden factors are elicited, better provision can be made for all. In figure 2.9b it is illustrated that area **C**, the hidden domain is much smaller than the corresponding hidden domain, **A** in figure 2.9a. Hidden domain factors which could still affect the core of a system could reside in the sub domain indicated by **C** in figure 2.9b.

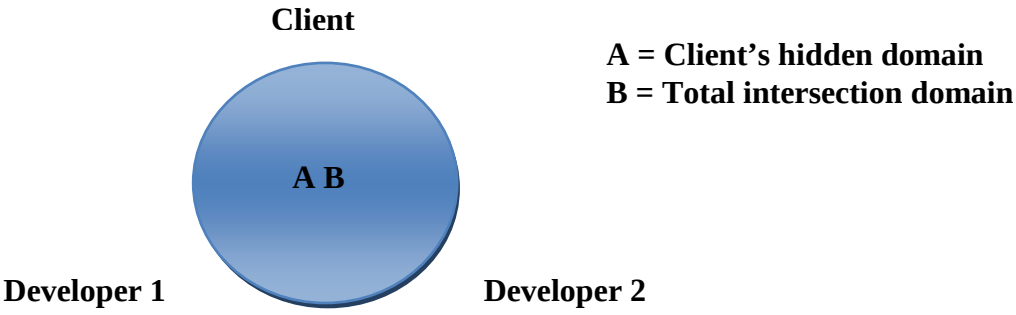


Figure 2.9c Total intersection of domains of developer(s) and client.

An ideal situation in which the client and the developers are the same person(s) is depicted in figure 2.9c (domain **A** = domain **B**). This would mean that a client develops a system for himself, and would therefore know of all the scenarios and exceptions that the system needs to cater for. However, it is recognised that this scenario is not always possible.

It is illustrated in figures 2.9a to 2.9c above that the more the domains are integrated (overlapped), the higher the potential of the software development team to understand

the full requirements of a client and the greater the chance of success for a software development project.

2.9 Summary

This chapter focussed on the hidden domain of the client and the fact that if an improved understanding of the hidden domain containing tacit knowledge of the client is gained, the requirements elicitation process during a software development project could be better than when the domain is not understood correctly. Venn diagrams were used to facilitate the presentation of such domains.

A generic software development environment was also described focussing on the roles of Project Managers, Architects, Business Analysts, System Analysts and Programmers/Developers. The dynamics of this environment were also covered.

The chapter touched on the SDLC, covering the process of how a software product starts its life cycle and the way in which the different stages function. Part of the SDLC process is requirements elicitation and the following methods to aid this process were highlighted: Natural Language, RAD, JAD, UML, Prototyping and Formal Method Z.

A communication gap which often exists between a client and a software development team was described. An introductory example was given to aid the understanding of the concept of the hidden domain of a client containing tacit knowledge and the effect it could have on the success of a software development project.

Venn diagrams were also utilised to aid the understanding of how different domains ought to integrate for the success of a software project.

The next chapter focuses on each of the requirement elicitation methodologies introduced in this chapter.

Chapter 3

Requirements Elicitation Techniques

In chapter 2 the hidden domain containing tacit knowledge of clients was described as well as how the domains between a client and a software development team may differ. An introduction to the SDLC was also presented as well as a number of requirements elicitation techniques namely, Natural Language, Prototyping, RAD, JAD, UML and Formal Method Z. The communication channels between different individuals in a software development team were referred to and a description of the communication gap between business and IT was given.

Chapter 3 contains a more comprehensive description of requirements elicitation techniques, as well as their strengths and weaknesses and examples are used to aid in the understanding and use of these techniques.

3.1 Existing requirements elicitation techniques

Requirements elicitation is a very important step in order to be able to gain the correct understanding of what the function of the software product should be. Various techniques have evolved over time. This chapter focuses on six techniques, namely:

- Natural Language.
- Prototyping.
- JAD (Joint Application Design).
- RAD (Rapid Application Development).
- UML (Unified Modelling Language).
- Formal Method Z.

The techniques mentioned above are used in the software development industry and each of these techniques will be described in more detail. In particular, different prototyping techniques are described as well as how these can be included when RAD or JAD is used. A basic overview of a part of UML, namely Use Case models, used

for requirements elicitation is given and an example is used to illustrate the effectiveness of UML. A Formal Specification Language Z is presented, describing its functionality and schemata.

3.2 Natural Language

Requirements for a system are often written in Natural Language. One of the draw backs is that it may be vague and ambiguous and developers could interpret the system wrongly.

Schach [79] describes a scenario in which a client approaches a development organisation with a certain requirement for a software product to be developed. He [p33] stipulates that: "*the client will outline the product as he/she* ***conceptualises*** [Bolded by writer of dissertation] *it. From the viewpoint of the developers the client's description of the desired product may be vague, unreasonable, contradictory or simply impossible to achieve. The task of the developers at this stage is to determine exactly what it is that the client needs and to find out from the client what constraints exist.*"

Schach refers to developers in his statement above and it should also be understood that developers could mean either the entire software development team or specific individuals in the team. Either the whole team or certain individuals may be responsible for requirements elicitation. In both cases domains need to integrate and increasingly overlap as much as possible for the tacit knowledge and hidden domain factors to surface. If this does not happen the result could be a system not functioning according to the requirements of the client, as developers could have conceptualised the system incorrectly. The previous point is also referred to by Brooks [22, p11]: "*We still make syntax errors, to be sure; but they are fuzz compared with the* ***conceptual errors*** [Bolded by writer of dissertation] *in most systems."*

The reason for the occurrence of conceptual errors is, the ambiguity that may be caused by Natural Language. If a client indicates that he needs a system to calculate insurance premiums and that the system should reward existing customers, a software

development team could conclude that all customers who are not new customers need to be rewarded. If the client does not stipulate clearly who should be rewarded and who not - for instance that individuals who have been customers for more than one year and have not claimed need to be rewarded - the software development team will not create the rewards of the system to cater for this stipulation. They will simply pay heed to the statement and create a system that rewards all customers.

Natural Language is an easier method of describing what the function of a system should be, but if specifications are not stipulated clearly, problems of ambiguity could arise and this could affect the core of a system negatively. In a renowned paper Meyer [61] gives a classical example of the inherent ambiguity of Natural Language specifications.

3.3 Prototyping

In this section the idea of first building a model of the proposed system which does not include all functionality is presented. This idea is embodied by the concept of a prototype.

3.3.1 Definition of prototype

"The word **prototype** comes from the Latin words "proto", meaning original and "typus", which means form or model." Prototype [76]. Longman [60] defines a prototype as: *"…the first form that a new design of a car, machine etc has, or a model of it used to test the design before it is produced."* Cambridge [24] defines a prototype as: *"…the first model or example of something new that can be developed or copied in the future a prototype for a new car."*

A prototype is, therefore, an original form or model which is usually built for demonstration purposes and which should indicate whether an idea or product is actually feasible or not.

3.3.2 Function of prototype

Prototypes are used for many projects covering many application areas and their main purpose is to indicate that an idea or concept is feasible and can actually be realised or implemented. Cooper [28], states that "*Prototyping is a manufacturing proof*". For example, if an architect designs a building, he often first builds a scale model of the proposed building to visualise the requirements of the client, as well as obtaining his approval before the project commences on a full scale. In information technology related projects, a prototype is built to capture the client's requirements and obtain his approval before the project continues.

In figure 3.1 (section 3.3.3.1) a layout for a new website is illustrated and the rough sketch is used to aid in the requirements elicitation process for the system. Should the client be satisfied, developers could build an electronic prototype such as the one in figure 3.3.

3.3.3 Different prototyping techniques

Prototypes and the different prototyping techniques one could use to assist the requirements elicitation process are described in this section.

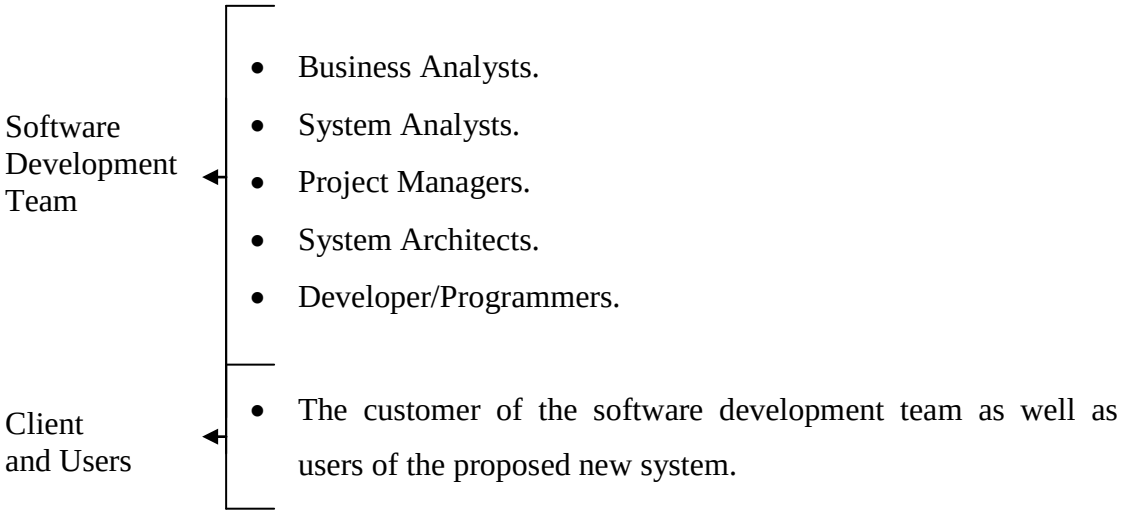
3.3.3.1 Prototyping with paper or cards

Prototyping with paper or cards is fairly simple, because the process follows that of the drawing of the proposed system or solution onto pieces of paper or cards. The method is not costly and should a particular piece of paper or card not meet certain requirements, it could either be improved or discarded and redrawn on a new piece of paper or card. An example of the use of this technique is presented in Snyder [83] and is discussed below.

Starting such a paper session requires that as many of the team members as possible from both the software development group as well as the client be involved. This is

important because it is during this first brain storming session (when a group of people meet in order to try to develop ideas and think of ways of solving problems, Longman [60]), that ideas are mentioned and captured as they go along, harvesting from experience and also previously implemented solutions.

Team members whose presence is critical during such a session include (refer to section 1.1):



For example, when using paper prototyping, a new web based system, including screens and the logic of how they need to flow from the viewpoint of users, may be designed in one session. While ideas and sketches are being made, the developers are already forming a picture in their own minds of what is involved and how they may need to proceed.

The picture shown in figure 3.1 below represents the layout of a web page on a piece of paper. This will form part of an entire web based transactional system used by the client's customers to order goods and services. From the picture in figure 3.1 one can form an idea of how the web page could be displayed and when viewing it, the client can support members of the software development team by informing them what else needs to be included and what needs to be excluded.

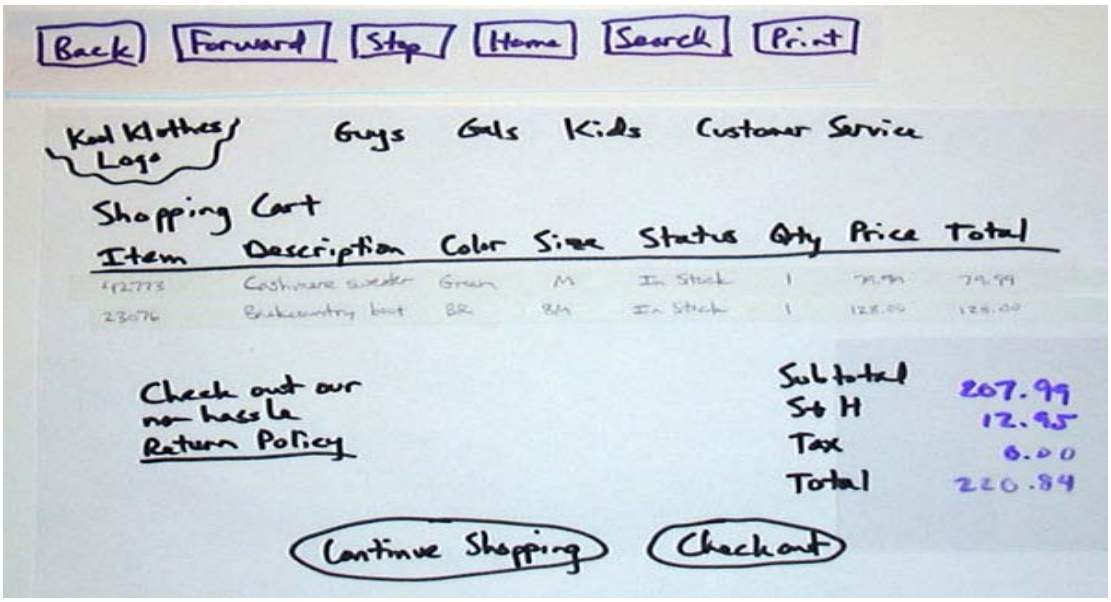


Figure 3.1 Shopping cart page from e-commerce site. (Snyder [83])

Note how the use of a picture as the one in figure 3.1 ties up with a very important design principle by Norman [67]. He contends that the user becomes creative in envisaging the possibility of certain ideas and methods when these are presented to him in visible form.

The use of paper prototypes has the following strengths (Snyder [83]):

- Focus is placed on concepts and terminology.
- Navigation and workflow are facilitated.
- Content issues are highlighted.
- Page layout is shown clearly.
- Missing functionality could be shown.

Some weaknesses of paper prototypes are (Snyder [83]):

- Technical feasibility is not shown.
- Non-functional requirements (e.g. response times) are not indicated.
- Colours and fonts are not shown.
- Scrolling of screens is not shown.

- Interaction (how the user would react to certain criteria) with the system is not known.

Additional benefits which may be realised by using a card or paper based prototyping method include:

- **Reduced cost**: If a piece of paper or a card does not meet certain requirements, the paper or card can easily be discarded.
- **Saved time**: The problem is addressed visually and this could enhance the understanding of the proposed system by both the client and the developers. It also facilitates a developer's grasp of the solution.
- **Focused team members**: Both the client and software development team members, start seeing what needs to be built and how all the pieces of the puzzle create a more comprehensive picture. This holistic view may enhance the understanding of the team members considerably.
- **Enhanced communication**: The client and the users of the proposed system could realise that certain issues which need to be included in the solution, have been overlooked.

There is, however, one more disadvantage associated with the use of a card or paper based prototypes: Despite all the visual aids built into this technique, the client and the users may still neglect to mention important tacit knowledge or other factors residing in their hidden domains.

3.3.3.2 Prototyping using 3D Whiteboards

Whiteboard prototyping addresses aspects very similar to cards and paper based techniques. Here too requirements are determined as one progresses through the process of requirements elicitation, deciding what the display of the system should be and how it needs to function. Ideally all team members mentioned above in section 3.3.3.1 need to attend while brainstorming is performed and system functionality is discussed and decided upon.

Figure 3.2a shows a 3-dimensional (3D) Whiteboard, executed while members of the software development team interact directly in designing a system. 3D Whiteboards are rather advanced compared to the ordinary magnetic Whiteboards (figure 3.2b) found in so many boardrooms. Both boards can speed up the process of determining user requirements, but the 3D board has a more visual appeal and is more user-interactive compared to an ordinary magnetic Whiteboard. Whiteboard markers are used to draw sketches of designs on the board for all participating members to see. The 3D board makes use of a computer to display objects discussed electronically. If any object needs to be changed, this can be done with relative ease and the team may have **what if** questions addressed. Such a question could, for example be: **What if the font is enlarged? Would it still be able to fit into the allocated space provided and if not, do we need to move any objects around or remove any objects on the current screen and create another screen with the objects removed from the previous screen?** This question addresses an issue of usability.

Gronbak, Gundersen, Mogensen and Orbak [43, p411] state the following regarding the use of a 3D board: "*The whiteboard can be used as an ordinary 2D whiteboard and seamlessly move into a 3D mono interface allowing users to place documents and objects in the background, in clusters, on top of each other, etc; creating more room for work while maintaining awareness of* ***collaborative manipulation of other relevant documents and objects***. [Bolded by writer of dissertation]"

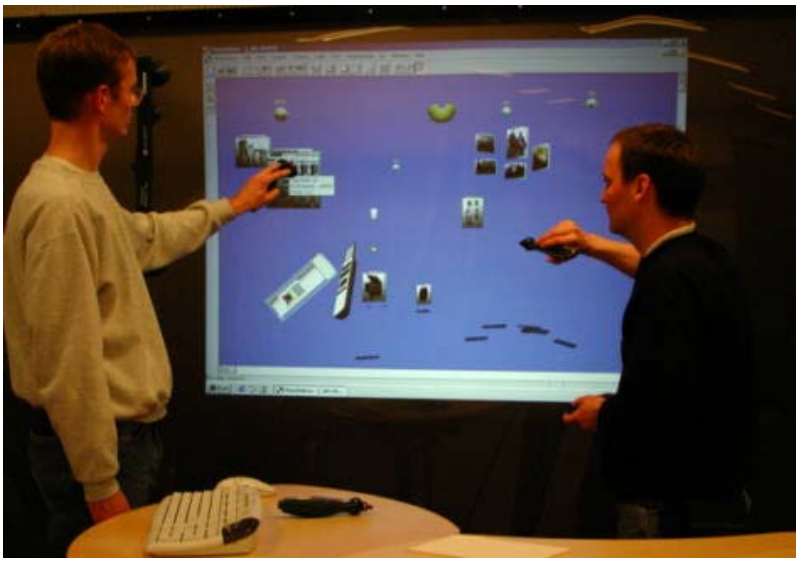


Figure 3.2a 3D Whiteboard. (Picture from Google, public domain)

Figure 3.2b Ordinary magnetic Whiteboard. (Picture from Google, public domain)

Should a scenario, after having been drawn on the board, not meet the approval of the stakeholders, the screen or scenario could either be amended or erased. This relative ease of changing scenarios facilitates requirements elicitation of the system. Visually this helps all team members (including client and users) to see an image of what it is they are discussing and it creates a better understanding of the system they are designing.

The benefits of paper or card prototyping techniques may also be realised by Whiteboard prototyping, as each technique has the same basic underline, namely, being able to draw rapidly and discard that which is not fulfilling the requirements of the client and the users.

Similar weaknesses, except one, are realised by paper, card and Whiteboard prototyping. This exception occurs when a 3D Whiteboard is used. With a 3D Whiteboard both the software development team and the client (and users) may view the display of the screens as well as the colours and fonts being used. One can use pens of different colours to draw on a magnetic Whiteboard to indicate colour differences, but the effect on a 3D Whiteboard is clearer and visually more appealing, because the electronic display (figure 3.2a) may be more natural to the client than a drawing would be on a magnetic Whiteboard (figure 3.2b).

The use of a 3D Whiteboard has a powerful **visual** advantage over paper, cards or magnetic Whiteboard prototyping techniques. This advantage creates a much improved global picture of the project and a clearer view of what the program would display, as well as how a user would be interacting with the system. Norman [67],

points out the advantages of visual representations in enabling the user of the program to explore the suggested possibilities, ideas and methods.

Therefore, the more visualisation a software development team can introduce during requirements elicitation, the better.

3.3.3.3 Prototyping with RAD (Rapid Application Development)

When using RAD methodology, the development team usually builds a Rapid Prototype, which is a smaller version of the program, focussing on the core functionality of the program (Hughes and Cotterell [49], Schach [80]). This means the client and users may view software screens, buttons and fields, without their having any actual underlying functionality, e.g. file saving capability. Nevertheless, these should still function as per the requirements of the client.

In some cases clients (and users) actually form part of the RAD process, but this usually happens during JAD (Joint Application Development) workshops, held between the client, users and the software development team (Wood [93]).
Figure 3.3 below is an example of a screen for an online seminar booking system. Once the system is up and running, customers may view the web page and make informed decisions on when and where they would like to attend a seminar. The customer can also reserve his place by using the system to make the booking. Requirements issued by the client and users who requested such a system to be built have been shown in a visual way, and this helps the client to form a better picture of what it is that he really wants from this product (Norman [67]).

Choi et al. [26, p100] state: "*Visualization of the part prior to physical fabrication will definitely enhance the designer's understanding of the part.*" This is a strength of Rapid Prototyping, because developers' understanding of the product they are building is enhanced. It is largely immaterial whether the process is taking place in a **manufacturing** or **software development** environment. If the product could be presented visually, understanding of the product would be enhanced.

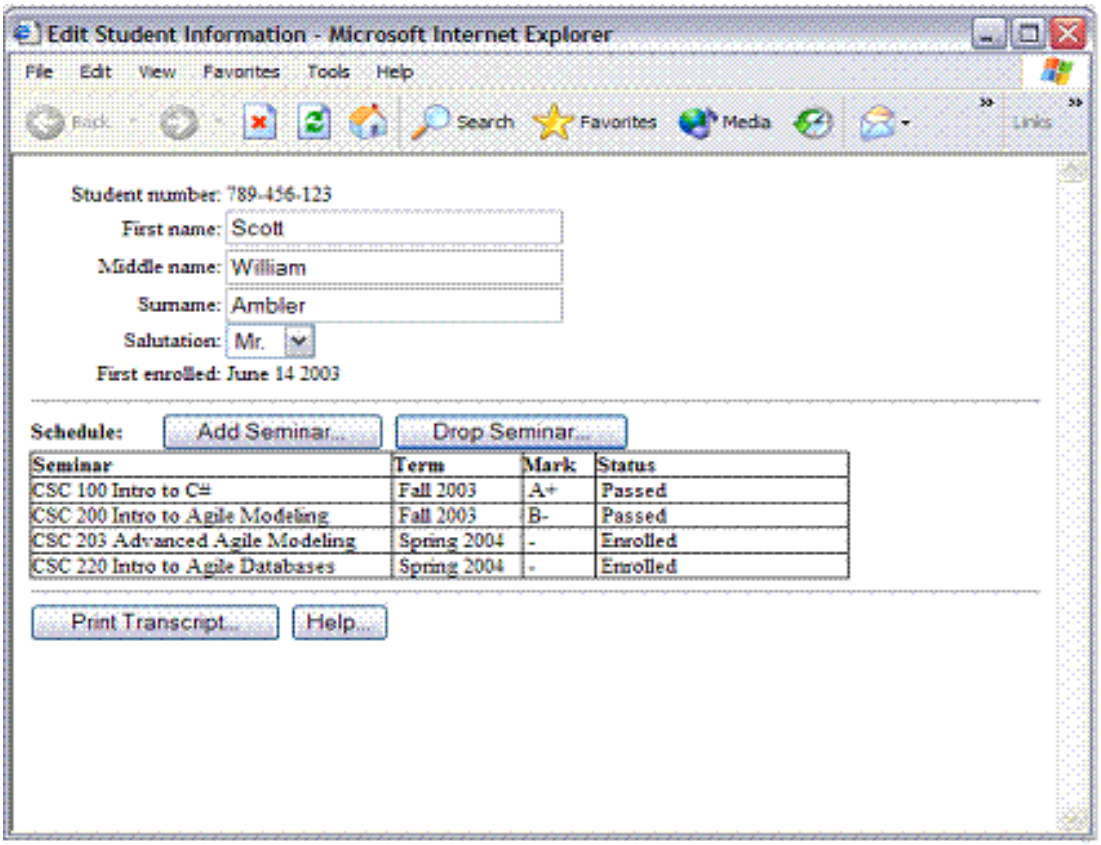


Figure 3.3 Example of user interface prototype. (HTML page from Google)

After a first version of a user interface has been displayed, the client may realise what else could be needed to support him in streamlining certain processes. At this stage a first review takes place at the next JAD session. The use of a RAD tool may, form part of a JAD workshop during which screen layouts and sequences are discussed and evaluated with respect to appearance and usability (Olsen [71]). Any other issues specific to the application to be developed may also be probed.

In using RAD, the approach is more formal than in the case of cards and paper prototyping. Documentation is drawn up, as would be the case with any of the techniques described to allow for further enhancements of the user interface and program requirements. Any other changes after the requirements documentation has been drawn up should be reflected in the documentation as the project proceeds through the normal software development life cycle phases, which will be discussed in more detail in chapter 4.

Strengths of RAD include (synthesised from Hughes & Cotterell [50] and Schach [79]):

- **Fast results**: The client can view that which he is asking for and can make changes to his requirements if necessary.
- **Quicker evaluation of software**: The prototype being produced mimics a real program instead of consisting of only diagrams.
- **Documentation**: A framework which enables developers to write a specifications document according to the requirements of the client is developed.
- **Improved understanding**: The software development team gains insight into how the product should be functioning and continuous learning takes place during the prototype evaluation cycle.

A major weakness of using RAD prototyping for requirements analysis and elicitation is that, in some cases, the Rapid Prototype, which has been put together quickly, may eventually become the actual product through continuous refinement and developments (Schach [79]). The danger of this approach, according to Schach [79], is that the prototype is not specified or designed with flexibility or enhancements in mind. Another reason mentioned by Schach [79] is that the performance of the system might have been overlooked during the building of the Rapid Prototype. In other words, non-functional requirements are not considered and the continuous refining of such a prototype to eventually reach the final product may become a very costly exercise.

Despite the above criticisms against the possibility of a prototype becoming the final system, one should note that an earlier edition of Hughes and Cotterell [50], mentions three types of prototypes, namely **throw away**, **evolutionary** and **incremental** prototypes.

A **throw away** is, as the name suggests, the building of the prototype and discarding it after it has been used to test ideas. An **evolutionary** prototype is enhanced and modified until it becomes the product itself (the kind criticised by Schach [79]). An

incremental prototype captures the idea of an actual system being developed and implemented in small stages. Some may consider this technique, strictly spoken not prototyping. Hughes and Cotterell [49] therefore removed the discussion of the incremental prototype from later editions of their book.

As with the other prototyping techniques above, RAD may also fail to elicit the necessary tacit knowledge and hidden factors in the domains of the client and the users, simply because clients and users may neglect to probe their own knowledge domains sufficiently.

3.3.3.4 Prototyping with JAD (Joint Application Development)

In all of the prototyping techniques above it is possible to embark on one or more JAD sessions, although it is not necessary. The advantage of having a JAD session is that all stakeholders are involved right from the start and during any decision making. The effect of this approach is that an environment in which understanding of requirements for the product may be achieved much more quickly is created. The reason for this is that all members of the client, users and the software development teams have a chance to participate. This supports the sharing of knowledge during JAD sessions. (Hughes and Cotterell [50], Schach [79], Wood [93]).

The type of prototype used in a JAD session may, to a large extent, be immaterial - it depends on what the group is comfortable using. The use of RAD tools during a JAD session is generally a very good method, as it aids the client, users and developers in seeing how the product should look and feel. Productivity may also be increased by the use of RAD tools.

A strength of using a RAD tool is that users may view results quickly (in line with Norman [67]) and can give inputs while requirements are being elicited for the system.

Two competing pressures mentioned in Hughes and Cotterell [50] are to get the job done as quickly and cost effectively as possible and to ensure that the final product has a robust structure and could meet evolving needs. The core of JAD sessions is reflected in these two conflicting goals. JAD workshops obtain results quickly during the sessions and developers aim to design the product to be as robust and flexible as possible to cater for future enhancements. Robustness and flexibility are, however, not guaranteed. If the product has not been designed in a flexible manner, it could create problems when the product progresses into the maintenance phase, since to change the product may often be very difficult without affecting the entire system negatively, as clarified by Boehm [16] in an explanatory diagram [p40].

Figure 3.4 shows a group interacting during a JAD session. Such interaction forms the basis of a JAD session, since the opportunity is given for ideas and suggestions to be presented and discussed by everyone in the group. The danger of a JAD session, however, is that there could be one or more individuals in the group who dominate the discussion, imposing their ideas on the rest of the group, while others in the group remain uninvolved.

Figure 3.4 Example of JAD session (Picture from Google public domain)

From the above discussion it is evident that the main aim of prototyping is to aid requirements elicitation from the client for the new system that needs to be developed. This is normally the first phase of the software development life cycle, namely analysing and determining the requirements for a new system (Charette [25], Wilson, Rauch and Paige [92]).

3.4 UML (Unified Modelling Language)

One of the purposes of UML is to aid participants in software development to build models and scenarios through Use Cases and Use Case models (Jalloul [53]). These enable the software development team to describe and visualise the proposed system.

Part of the functions of UML is also to assist in establishing the architectural requirements for the system and to document what the software development team decides as and when they make changes while building the system.

Giese and Heldal [41, p197] also refer to the ability of UML to give an overview of the architecture and state of software: *"It permits several views of software systems, and it gives a good overview of the software's architecture."*

Use Cases are used in UML to describe what the system must be able to do. Scott [81, p2] states that: *"…UML was designed to help the participants in software development efforts build models that enable the team to visualize the system, specify the structure and* ***behaviour*** [Bolded by writer of dissertation] *of that system, construct the system, and document the decisions made along the way"*.

The ability of UML to give several views of a software system as stated by Giese et al. [41], reminds of Norman's [67] view in which he contends that the user becomes creative in envisaging the possibility of certain ideas and methods when these are presented to him in visible form. This is because each diagram viewed by the user allows him to understand the system better and also to explore further, using new ideas and aids in the process of requirements elicitation.

The development team could build any number of Use Cases in UML. *"A use case is a sequence of actions performed by one or more actors (people or non-human entities outside of the system) and by the system itself, that produces one or more results of value to one or more actors."* Scott [81, p3]

One of the difficult tasks which need to be performed by a development team is to determine the requirements of the client. Scott [81, p19] highlights this as follows:

- *"For all practical purposes, customers and developers speak different languages."* Figure 2.6 illustrated this point. The result could be that a communication gap could emerge during the course of a software development project. The more domains of IT and business (IT and business defined in section 2.6) integrate, the higher the probability that communication would be better and that a software development project would be a success.

- *"It's often hard to pin customers down about what they want: they don't know, or they think they know, but can't* ***articulate*** [Bolded by writer of dissertation] *it, they change their minds, they contradict themselves."* This is confirmed by Hudlicka [48, p4] who states: "*In other words, neither experts nor the intended system users are always able to state their knowledge and system requirements succinctly in response to direct questions.*"

- *"Requirements documents tend to be riddled with ambiguity, redundancy, and internal contradiction."* Opiyo, Horváth and Vergeest [72, p206] state: *"To overwhelming degree, the* ***requirement*** [Bolded by writer of dissertation] *specifications are generally unclear, incomplete, inconsistent, lacking in detail and* ***riddled with ambiguities*** [Bolded by writer of dissertation]*."* The inherent ambiguity of especially Natural Language specifications is further illustrated by Meyer [61].

UML addresses many of the issues highlighted by Scott [81] and focuses exactly on some of the problems that developers have experienced in software development projects.

Use Cases are the driving force behind the development of the system in UML, Scott [81, p3] states: *"…. (the) customer's requirements, as expressed within use cases, should be the most important driving force in software development," This is true* because the software development team creates the various scenarios to be provided for by using Use Cases which enable members of the team to elicit the requirements from the user much more clearly. Scott [81, p4] states that: "*use cases offer a*

considerably greater ability for everyone to understand the real requirements of the system than typical requirements documents."

Use Cases also support software developers during the implementation phase of the system, Developers develop code to cater for the various scenarios and testers (Refer to section 2.7) ought to test the system against those scenarios. This is where Use Cases are of great value. One is actually led to understand how important such Use Cases are when realising how often a development team refers to them during the development life cycle. Scott [81, p4] confirms this: *"…by keeping the use cases close by at all times, the development team is always in touch with the customer's requirements."*

Use Cases also form part of the overall documentation of the system, If the system is being validated and verified (in essence, for the purposes of this dissertation, tested by testers) and it does not function according to the specifications as stipulated, the problem can be validated or verified against the Use Case scenarios to ensure that it is indeed a problem. If, after investigation, it proves to be a problem, it could be referred back to the software development team for further investigation. If a correction needs to be made, the developer could correct the product and send it back to the testers for evaluation again.

Use Cases may be written in, for example, Natural Language (Bahrami [7]) or may employ a diagrammatic notation as follows:

- An **actor** is indicated by a stick man.
- One or more **arrows** link the actor to each step he needs to perform in the system.
- **Elliptical circles** indicate steps or processes the actor must perform in the new system.

An example of a simple UML Use Case diagram for a banking scenario appears in figure 3.5a.

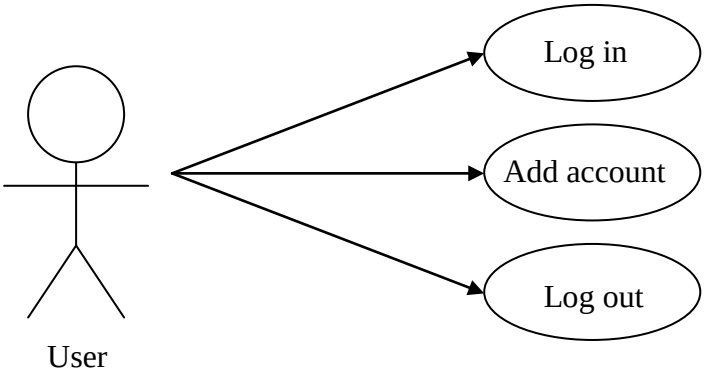


Figure 3.5a Example of Use Case diagram for adding account.

Figure 3.5a portrays a scenario in which a user may log into the system, add an account and log out of the system. This represents only a single scenario needed to be included into the functionality of the system. Developers could create many more use cases and each of them would represent a scenario to be catered for in the system.

Another scenario could be that in which the user needs to be able to allow only certain people to remove an account from the system. This scenario is shown in figure 3.5b.

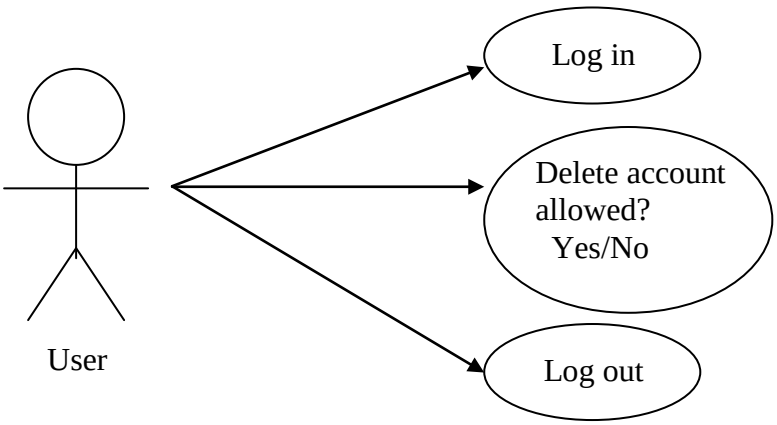


Figure 3.5b Example of Use Case diagram for deleting account.

The software development team should continue to create Use Cases until they have comprehensively captured all requirements from the client.

The following advantages of using UML may be synthesised from Scott [81]:

- **User centredness**: Use Cases are created from the perspective of the users. This translates into a higher comfort level for users, as they can see themselves reflected in the Use Case text.
- **Natural expressiveness**: Use Cases are often expressed in Natural Language. Well-written Use Cases are also intuitively obvious to the reader. Note, however, that this may bring about the well-known problems associated with the use of Natural Language, e.g. ambiguity (Meyer [61], Scott [81]).
- **Accessible requirements**: Use Cases offer a considerably greater possibility for everyone to understand the real requirements of the system as compared to typical requirements documents in Natural Language. The latter tend to contain a great deal of ambiguous, redundant and contradictory text.
- **Traceability**: Use Cases offer the ability to achieve a high degree of traceability of requirements into the models as a result of ongoing development.
- **Grouping**: Use Cases offer a simple way to arrange requirements into groups which allows for allocation of work to sub- teams.
- **Domain knowledge**: By using Use Cases, the development team may certainly gain great insight into how the business functions and how the program should be functioning within the business of the client.

The visualisation aspect of Use Cases could enable both clients and users to notice that some hidden factors (tacit knowledge) have been overlooked. Should this fact, however, not be pointed out to the software development team, those requirements would not be included in either the requirements for the system, or in the Use Cases used for documentation and testing purposes.

Next an example, given before, is revisited to illustrate the use of Use Cases further. It is also done to show that the mining of the necessary tacit knowledge and hidden domain knowledge from a client and users may still be a problem, despite the power inherent in the use of the Use Cases of UML.

Example 3.1

A client approaches a software development company with a request to develop a software program able to diagnose blood infections. After consultations with the client, the development team develops various Use Cases for the system.

Two Use Cases outlining the basic use of the system are illustrated in figures 3.5c and 3.5d below:

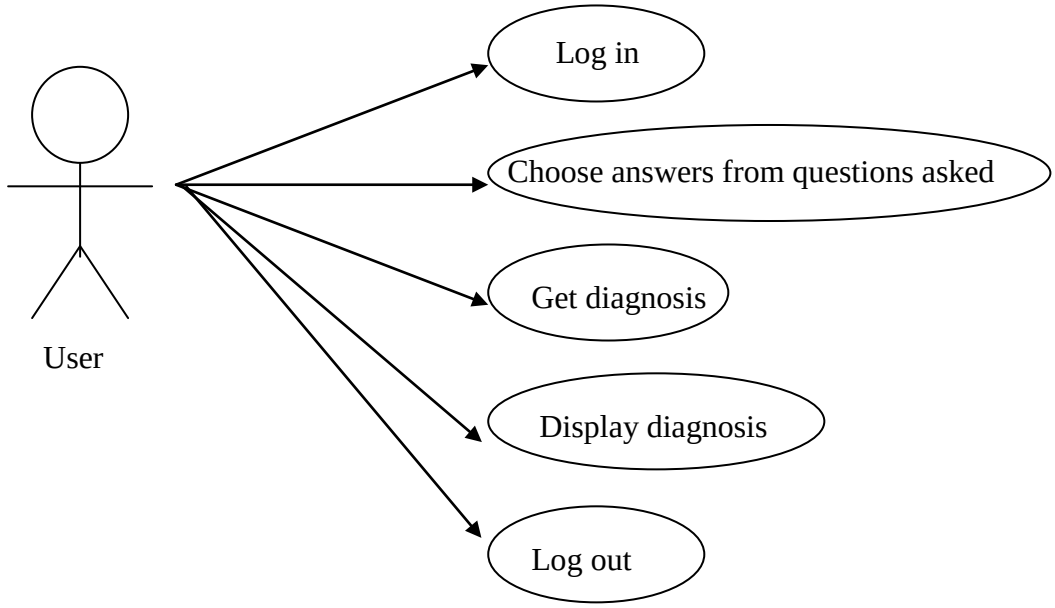


Figure 3.5c Use Case diagram for obtaining and displaying diagnosis.

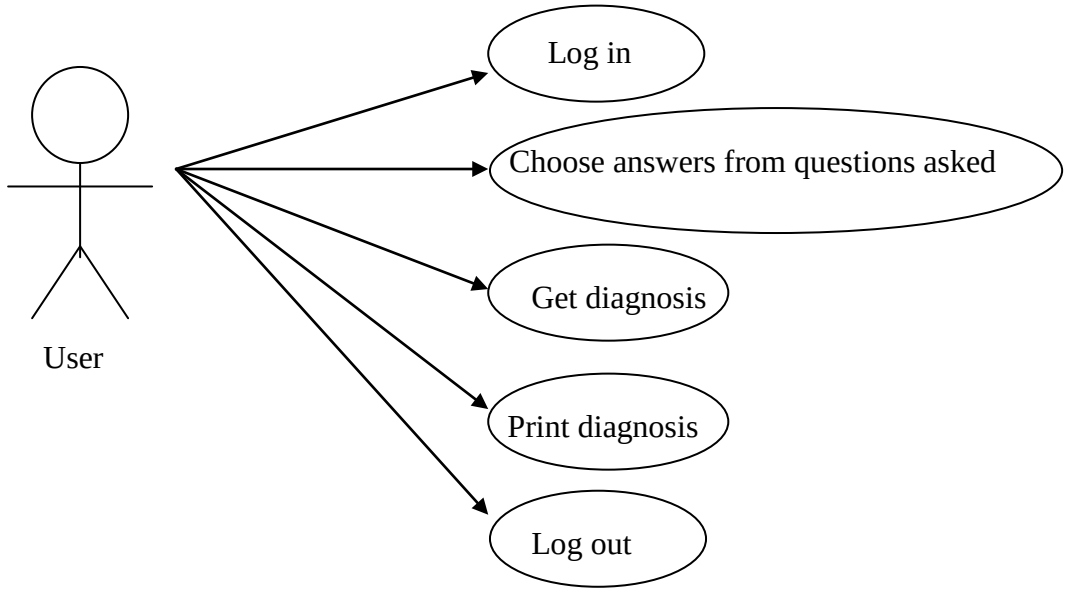


Figure 3.5d Use Case diagram for obtaining and printing diagnosis.

Figures 3.5 c and d respectively provide a high level view of the functionality of the system as per the requirements of the client and users. The difference is that figure 3.5c has an option for **displaying** the diagnosis of the system, whereas figure 3.5d has an option for **printing** the diagnosis of the system. In both cases the results would be the same.

Next the developers focus on the type of questions asked by the system and develop Use Cases around the specific shape of the organism. The Use Case in figure 3.5e is at a sub- level of the **Choose answers from questions asked** type, because it is in this part of the system where the questions need to be asked and where the answers to the questions would affect the logical flow of the system until it can establish a diagnosis.

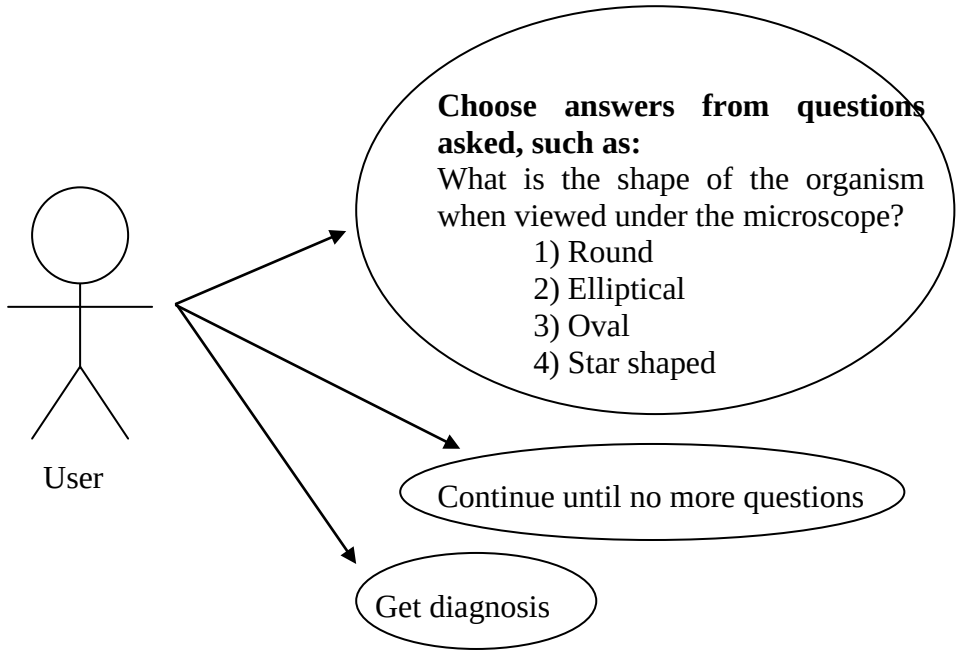


Figure 3.5e Use Case diagram for selecting answers until diagnosis can be established.

UML is certainly powerful and many software organisations have used it for software development projects, realising many of the benefits mentioned by Scott [81] and at the same time improving productivity amongst development teams. However, UML is no silver bullet (Brooks [22]), because developers can only create Use Cases from the information elicited by them from the client. If any misconceptions were created during the Use Case development stage, the system would not capture the

requirements of the client correctly. Developers might still only vaguely understand the bigger picture of the new system to be developed, thus causing them to make assumptions, which could lead to a faulty system. This point is stated by Dix [33, p46]: *"Other errors result from incorrect understanding, or model, of a situation or system. People build their own theories to understand the causal behaviour of systems."* Assumptions could still lead to clients uttering the following words: *"I know that this is what I asked for, but it isn't really what I wanted."* (Schach [79, p33]). Incorrect assumptions could lead to faulty systems.

As before, the discussion above emphasises the problem of the correct elicitation of tacit knowledge and factors residing in the hidden domains of clients and users.

Next the use of a formal method, in our case the Specification Language Z, is illustrated to specify the same system as in example 3.1 above.

3.5 Formal Method Z

Requirements for a system can also be specified in a Formal Language such as Z (Lightfoot [58], Potter et al. [74]). Developers work with a description of the system in a formal, i.e. mathematical manner when using this method. The use of formal methods removes the ambiguity normally associated with Natural Language specifications (Lightfoot [58]). A Formal Language uses mathematical symbols and the semantics of the symbols can be understood in many different spoken languages.

Z is based on a strongly typed fragment of the Zermelo-Fraenkel set theory (Enderton [35]). Some of the symbols available in Z, together with their explanations, are given below (Lightfoot [58]):

∅ – Indicates **emptiness**, e.g. an empty set.

Ξ – Indicates **no state changes to the system.** This is used in operations such as viewing.

Δ – Indicates **a state change to the system.** This is used when the before and after value of the same variable in the state may be different.

? – Indicates **a value that can be entered**, i.e. an input variable. For example, a person's name can be entered into a variable that will hold the name.

! – Indicates **an output of a value**, e.g. a variable that returns with a result after some processing has been done.

< – Indicates **less than.**

> – Indicates **greater than**.

= – Indicates **equals.**

´ – Indicates **not equal.**

e – Indicates **an element of a set.**

~ – Indicates **not an element of a set**.

Æ – Indicates **logical and**, e.g. p > q **and** s f r.

v – Indicates **logical or**, e.g. p > q **or** s f r.

Many other mathematical symbols are available in standard Z and the reader is referred to Spivey [84] for a more complete list.

Example 3.2

A client approaches a software development company with a request to develop a software program that will be able to diagnose blood infections. After consultations with the client, a formal specification of the problem is constructed.

It is customary in Z to start a specification document with the basic types that are to be used in the specification (Potter et. al. [74], Barden et al. [9]). A **basic type** serves a rather similar purpose to a **type** in a programming language, e.g. **Integer** or **String**, etc. The development team decides on the following basic types:

[NAME] - The set of all names

[PASSWORD] - The set of all passwords

[ANSWER] - The set of all answers

[DIAGNOSIS] - The set of all diagnoses

Next the team may decide to define all the input variables that are needed in the specification document. These are:

Name? – Variable that will hold the user's name.
Password? – Variable that will hold the user's password.
Answer? – Variable holding user's answer.
Diagnosis! – Variable that will output the information after the system has reached a diagnosis.

Free type definitions in Z are normally used to indicate the results of operations. This is in essence to establish some form of communication with the user. The development team defines one such free type:

ENTRYORNOT ::= accepted | rejected

The development team may also define some axiomatic definitions. These are definitions that hold throughout a specification document and are normally used to specify values of constants or boundary conditions, etc. The axiomatic definition below ensures the return of one or more diagnostic results, according to the questions answered in a particular case. N indicates that there will be at least one diagnostic result. This means the system would be able to reach at least one conclusion. Zero is not an option as the selection rules of the system are fixed. The client does not require any artificial intelligence capabilities. The system design therefore follows a predefined hierarchy of questions to reach one or more diagnostic probability.

diagnosisResults: N - The diagnosis results returned will be one (1) or more diagnosis.

Next the definition of the abstract state of the system to be developed is given by schema DiagnosisSystem below.

```
¿_______DiagnosisSystem_______________
ˀ
ˀ       Name : ℙNAME
ˀ       L_Name : ℙNAME
ˀ       Password : ℙPASSWORD
ˀ       Answer : ℙANSWER
ˀ       Diagnosis : ℙDIAGNOSIS
□_____________________
ˀ       L_Name ⊂ Name
ˀ       #Diagnosis ≥ diagnosisResults
|_____________________________________
```

DiagnosisSystem schema defines Name, L_Name, Password, Answer and Diagnosis to be power sets of NAME, PASSWORD, ANSWER, DIAGNOSIS respectively. The constraining predicate #Diagnosis ≥ diagnosisResults, simply states that the system could reach one or more probable diagnosis (diagnostic result). The set of all users currently logged in, is indicated by L_Name. Naturally, this set is a subset of all names known to the system.

The various operations are specified next. The first one is to initialise the system:

```
¿_____Init_DiagnosisSystem_________________
ˀ
ˀ       DiagnosisSystem'
ˀ
□_________________
ˀ       Name' = 0
ˀ       Password' = 0
ˀ       Answer' = 0
ˀ       Diagnosis' = 0
|_____________________________________
```

All component sets are initialised to be empty at this stage, meaning the system has started and nothing has been entered yet. Normally at this point, the software development team will pose and discharge a proof obligation, showing that the initial state of the system above may be realised, i.e. it actually exists.

The proof obligation can be written as:

∃ DiagnosisSystem' • Init_DiagnosisSystem

The proof obligation above stipulates that it must be possible to arrive at a system state (DiagnosisSystem') which satisfies the requirements of Init_DiagnosisSystem (Potter et. al. [74]). Note at this point that any proof of the formula above can necessarily only address aspects explicitly mentioned in the formula itself, i.e. the content of the schemas mentioned in the formula. Hence, any tacit knowledge or hidden factors in the domains of the client or the users may not be elicited.

The following operation allows a user (n?) access to the system via the use of a password (p?).

EnterSystem_OK

Δ DiagnosisSystem
n? : NAME
p? : PASSWORD
reply! : ENTRYORNOT

n? ∈ Name
p? ∈ Password
L_Name' = L_Name ∪ {n?}
reply! = accepted

Δ DiagnosisSystem indicates that there will be a change in the state of the system. The user needs to input his username and password. The set of users currently logged in is changed accordingly. The value of reply! indicates that the operation was successful.

If either the username or password entered is incorrect, the user is denied entry into the system. This scenario is captured by schema EnterSystem_Not_OK below.

EnterSystem_Not_OK

Ξ DiagnosisSystem

```
³   n? : NAME
³   p? : PASSWORD
³   reply! : ENTRYORNOT
Ã___________________
³   (n? ~ Name v p? ~ Password)
³   reply! = rejected
³___________________________
```

x DiagnosisSystem above indicates that there is no change to the state. The value of reply! indicates to the user that entry was rejected.

As is customary in Z, the next step is to define a robust version of the above operation, i.e. one that caters for both situations. The Z schema calculus is used to define a robust operation (Potter et al. [74]).

Robust_EnterSystem ° EnterSystem_OK v EnterSystem_Not_OK

```
¿____Robust EnterSystem_____________
³
³   D DiagnosisSystem
³   n? : NAME
³   p? : PASSWORD
³   reply! : ENTRYORNOT
Ã_____________
³   (n? e Name
³   p? e Password
³   L_Name' = L_Name U {n?}
³   reply! = accepted)
³          v
³   (n? ~ Name v p? ~ Password
³   reply! = rejected)
³_____________________________
```

The remainder of this dissertation will show the correct version for every Z operation and the incorrect one separately (if applicable) and omit the definition of the robust version without loss of generality.

An operation to add an answer to the set of user answers so far is:

```
┌─────AddAnswer──────────────────────
│
│   Δ DiagnosisSystem
│   a? : ANSWER
├──────────────────────────
│
│   Answer' = Answer ∪ { a? }
│
└──────────────────────────────────
```

When the user enters an answer, it is stored. As before, Δ DiagnosisSystem indicates that the state of the system changes. No error operation is defined for adding an answer from a user.

The next two operations respectively display a diagnosis on the screen and print a diagnosis on a printer.

```
┌────ViewDiagnosis──────────────────
│
│   ScreenDevice
│   Ξ DiagnosisSystem
│   d! : ℙDIAGNOSIS
├──────────────────────────
│
│   d! = {d : Diagnosis | d was selected by the system}
│   console' = stream(d!)
└──────────────────────────────────
```

Schema ViewDiagnosis displays a diagnosis from all answers given. The output variable d! contains the results from the internal operations and deduction rules that have been followed by the system after a user has entered all answers to diagnostic questions. The process of extracting the set of possible diagnosis is left unspecified, as is evident in the definition of d! above.

```
┌─────PrintDiagnosis─────────────────
│
│   PrinterDevice
│   Ξ DiagnosisSystem
│   d! : ℙDIAGNOSIS
├──────────────────────────
│
```

```
ʔ    d! = {d : Diagnosis | d was selected by the system}
ʔ    printer' = pstream(d!)
└──────────────────────────────
```

The system now prints a diagnosis from all answers given.

The above formal notation may be overwhelming to many, even to members of software development teams, but especially to clients and users. However, formal methods are precise and remove many ambiguities which are found in Natural Language descriptions, because of their mathematical notation (Lightfoot [58]). The mathematical notation removes ambiguity which could have arisen during the requirements elicitation process. The problem with this approach is that many clients might not understand the formal specifications and rules, because of the mathematical rigour. The client will often sign the specifications document under pressure, hoping the system will be developed correctly according to his requirements. (refer section 4.1)

Developers may be well supported by formal methods once they understand how to use the methodology to create precise software specification rules and requirements for a system. Mathematical notation could also be an aid to cross many boundaries and language problems which could exist within a development team. Many developers from different cultures and even in different locations may all work together and understand exactly what is needed and required for the new system by using formal methods. However, while the semantics of a formal notation might be clear to the software development team, it could be quite unintelligible to the client, who would still revert to Natural Language to express his concerns and requirements.

Figure 2.5 indicates that Z is one of the techniques which a software development team could use in its task to determine user requirements of a system. Other techniques in the figure include RAD, JAD, UML, Natural Language and Prototyping. Any of these techniques or any combination thereof could be used during requirements analysis. The software development team would decide which technique or combination of techniques would probably provide the best results during the

requirements analysis phase. Each technique could be applied in isolation or could be combined with others, as illustrated in the following section.

3.6 Combining techniques

This section describes the way in which the various requirements elicitation and specification techniques described in figure 2.5 and also discussed above could be combined. Details are provided in example 3.3.

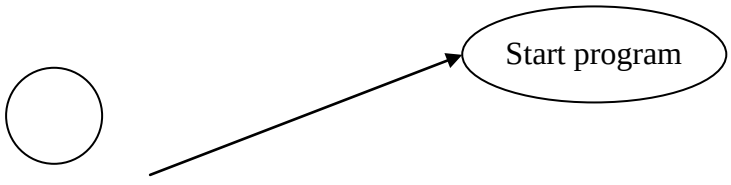
Example 3.3

The application to be considered is the same as before, namely, a system to diagnose blood infections. The combined use of Natural Language, JAD workshops (including RAD) and Formal Method Z is illustrated.

Natural Language: The client requires that the system should have security built in so that no unauthorized person could use the system. The system should also ask questions and, depending on the answers selected by the user, make informed decisions and in doing so, eliminate certain diseases until the system could reach a conclusion and determine a diagnosis. The user should then be able to view and print the diagnosis, which could then be used to determine what medication should be prescribed for the patient.

JAD workshop: The software development team decides to conduct JAD workshops to determine what the system should look like and what scenarios should be provided for in the system in order to build a system that would not reach incorrect conclusions. During the workshops a great deal of interaction between blood domain experts and the software development team takes place. Various scenarios are created and the software development team uses UML to create the scenarios via Use Case models.

The Use Case in figure 3.6 illustrates how a user would gain access to the system.



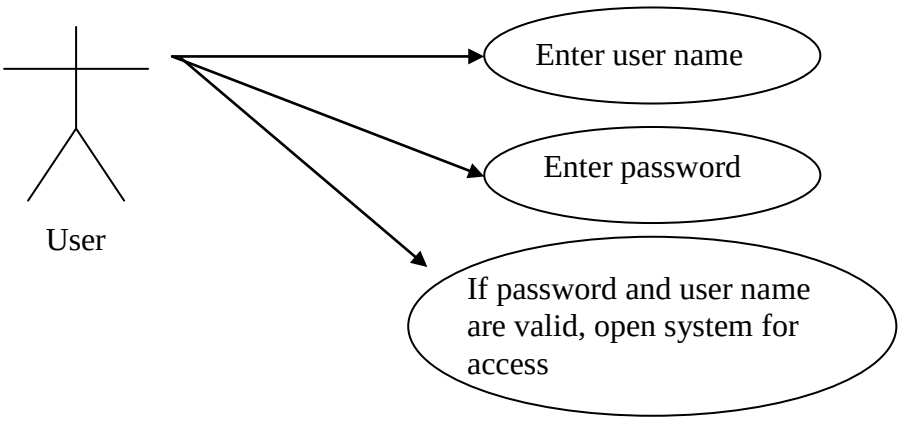


Figure 3.6a Use Case diagram for gaining access to system.

The next Use Case focuses on the system and the type of questions the system needs to ask according to the blood infection experts. In figure 3.6b one of these questions is shown.

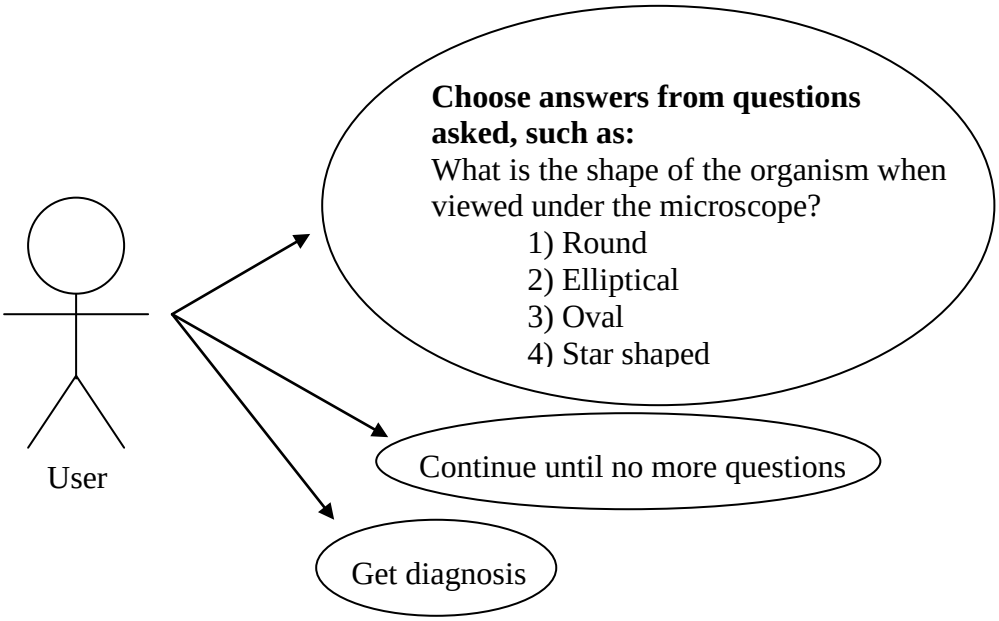


Figure 3.6b Use Case diagram for selecting answers until diagnosis can be established.

After the Use Cases have been done, the software development team decides, to build a prototype of the system and interact further with users of the system to ensure they have captured the functionality of the system correctly. The team decides to use a RAD tool for the creation of the prototype and to build a small version of what the system would visually look like to the user.

Examples of screens developed in the workshop by using a RAD tool are shown in figures 3.7 – 3.10 below.

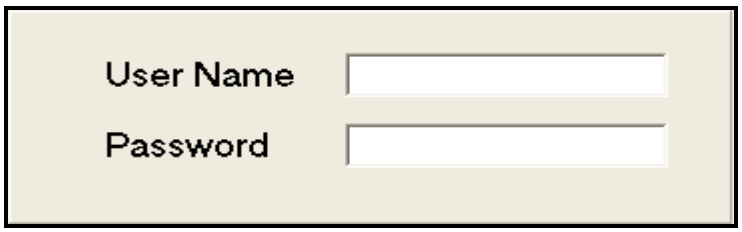


Figure 3.7 Example of user access screen of system.

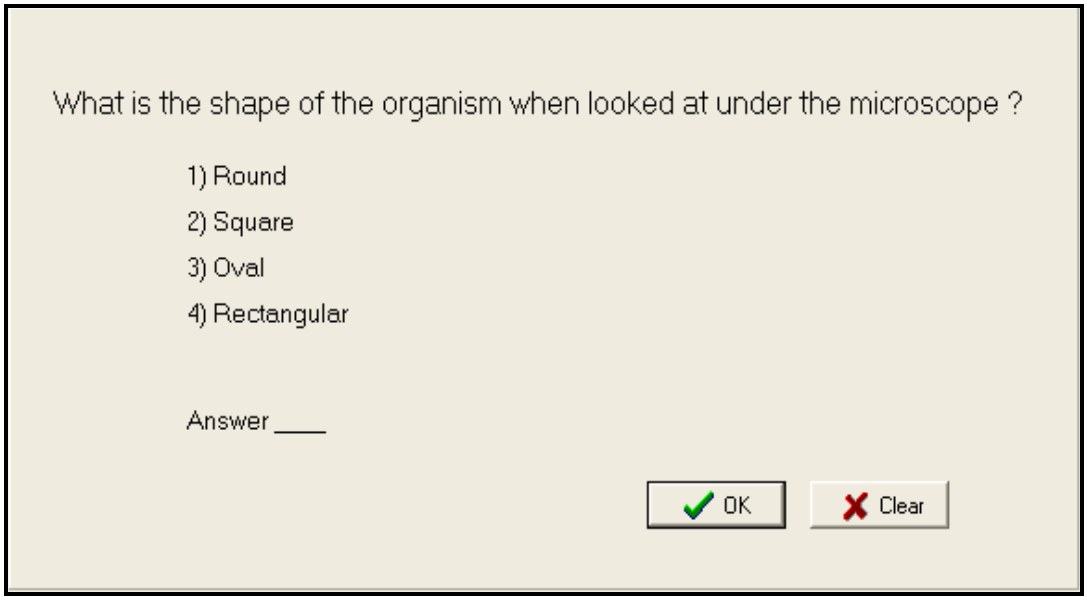


Figure 3.8 Example of question screen of system.

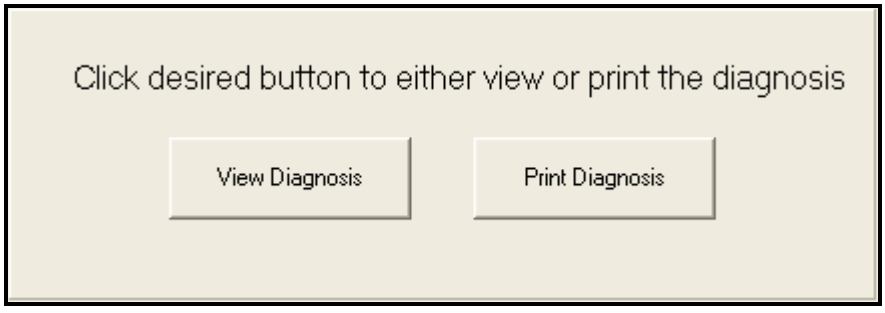


Figure 3.9 Example of user selection screen.

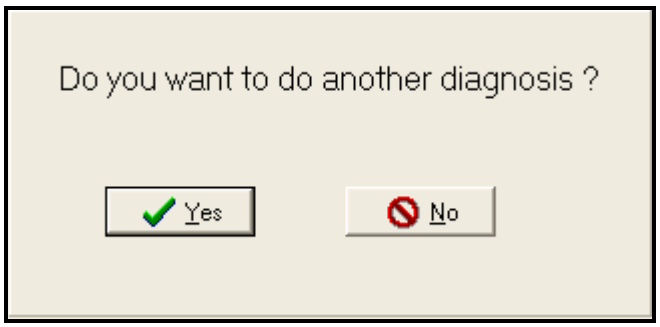


Figure 3.10 Example of user selection screen to do another diagnosis.

After the users have expressed their satisfaction with the functionality of the prototype, the team continues with the specification of the system by using Formal Language Z. The power of Z lies in the fact that it removes the ambiguity that could be caused by using Natural Language. This is one of the reasons for, the decision of the team to continue with Z in drawing up the specification. One of the lucrative features of using Formal Method Z is that the development team may reason about the properties of the specification within the limits of schemata and schema calculus (Woodcock [94]).

Z schemas created by the software development team are presented below. The reader is referred to section 3.5 for the introduction to Z and its symbols.

¿___DiagnosisSystem_____
?
? Name: ℙNAME
? L_Name: ℙNAME
? Password: ℙPASSWORD
? Answer: ℙANSWER
? Diagnosis: ℙDIAGNOSIS
□_____
? L_Name ≠ Name

ˀ #Diagnosis ¸ diagnosisResults
└──────────────────────────

DiagnosisSystem schema defines Name, L_Name, Password, Answer and Diagnosis to be power sets of NAME, PASSWORD, ANSWER, DIAGNOSIS respectively. The constraining predicate #Diagnosis ¸ diagnosisResults, simply states that the system could reach one or more probable diagnosis (diagnostic result). The set of all users currently logged in is indicated by L_Name. Naturally, this set is a subset of all names known to the system.

Operations:

¿──Init_DiagnosisSystem ──────────
ˀ
ˀ DiagnosisSystem'
ˀ
□──────────
ˀ Name' = 0
ˀ Password' = 0
ˀ Answer' = 0
ˀ Diagnosis' = 0
└──────────────────────────

All component sets are initialised to be empty at this stage as the system has started and nothing has been entered yet. Normally at this point, the software development team will pose and discharge a proof obligation, proving that the above initial state of the system may be realised, i.e. it actually exists.
The proof obligation can be written as:

∃ DiagnosisSystem' • Init_DiagnosisSystem

The above proof obligation stipulates that it must be possible to arrive at a system state (DiagnosisSystem') which satisfies the requirements of Init_DiagnosisSystem (Potter et. al. [74]). Note that at this point any proof of the formula above can necessarily only address aspects explicitly mentioned in the formula itself, i.e. the content of the schemas mentioned in the formula. Hence, any **tacit knowledge** or **hidden factors** in the domains of the client or users may not be elicited.

The following operation allows a user (n?) access to the system via the use of a password (p?).

SystemEntryAccepted

D DiagnosisSystem
n? : NAME
p? : PASSWORD
reply! : ENTRYORNOT

---

n? ∈ Name
p? ∈ Password
L_Name' = L_Name ∪ {n?}
reply! = accepted

D DiagnosisSystem indicates that there will be a change in the state of the system.

The user needs to input his user name and password. The set of users currently logged in is changed accordingly. The value of reply! indicates that the operation was successful.

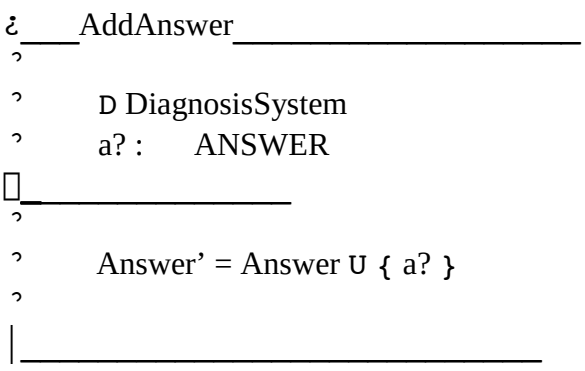
If either the user name or password is incorrect or does not exist, the user is denied entry into the system. This scenario is captured by schema SystemEntryRejected below.

SystemEntryRejected

Ξ DiagnosisSystem
n? : NAME
p? : PASSWORD
reply! : ENTRYORNOT

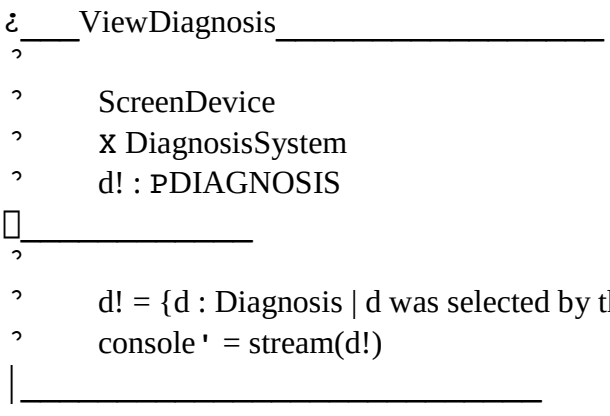
(n? ↦ Name ∨ p? ↦ Password)
reply! = rejected

X DiagnosisSystem above indicates that there is no change to the state. The value of reply! indicates to the user that entry was rejected.



When the user enters an answer, it is stored. As before, D DiagnosisSystem indicates that the state of the system changes. No error operation is defined for adding an answer from a user. A possible, yet trivial, proof obligation is to show that the set of answers is changed with a? as indicated above.

Each of the next two operations respectively displays a diagnosis on the screen or prints a diagnosis on a printer.



Schema ViewDiagnosis displays a diagnosis from all answers given. The output variable d! contains the results from the internal operations and deduction rules followed by the system after a user has entered all answers to diagnostic questions. The process of extracting the set of possible diagnosis is left unspecified as evident in the definition of d! above.

PrintDiagnosis

PrinterDevice
X DiagnosisSystem
d! : ℙDIAGNOSIS

d! = {d : Diagnosis | d was selected by the system}
printer' = pstream(d!)

The system now prints a diagnosis from all answers given.

Example 3.3 illustrates the way in which the various requirements elicitation techniques presented before may be integrated into one system. Each method could, however, be used in isolation, depending on the complexity of the project and that with which the software development team is comfortable.

An important issue regarding the desired outcome of the research for this dissertation, namely, the elicitation of tacit knowledge and factors in the hidden domain of clients and users, should be mentioned here. The above proof obligation showing that the list of current users is correctly updated when the last user logs on, merely addresses components and aspects in the schemas already present or which may be deducted from existing information.. The proof does not reveal any information or hidden knowledge not elicited from the client. It simply makes deductions from existing knowledge.

Another proof obligation illustrated with reference to schema AddAnswer above may be used to prove the same point. In the schema the shape of the organism (round, elliptical, oval or star shaped) is abstracted from the simple use of the input variable a? Nevertheless, even if such detail about the shape is given in the schema, no

mention is made of the **colour** of the organism, i.e. whether it is black or red. Hence, again any proof obligation in this regard would have been unable to elicit such important information which would have been extremely important in establishing a correct diagnosis (Refer to section 2.7, example 2.1).

It follows, therefore, that the elicitation of tacit knowledge and factors in hidden domains calls for additional techniques to be developed. This is, amongst other things, the topic of chapter 6.

3.7 Summary

In chapter 2 the problem of misconceptions in the area of concept understanding and transferral was highlighted. In particular the communication gap which could evolve between business and IT was addressed.

The main focus of this chapter was the existing requirements elicitation techniques, which included Natural Language, Prototyping, RAD, JAD, UML and Formal Method Z.

Prototyping as discussed in this chapter also included both RAD and JAD, because both techniques may be used in a prototyping environment. An example of the way in which a system could be developed to diagnose blood infections was used in both UML and Z. The use of a JAD session and an example of a RAD screen were included.

A combination of a number of these techniques was discussed in Example 3.3.

This chapter indicated that tacit knowledge and factors in the hidden domains of clients and users would probably remain hidden by employing traditional requirements elicitation techniques, namely, Natural Language, Prototyping, RAD, JAD and UML in industry. It was furthermore argued that such knowledge and factors would remain hidden even when using a formal method, despite the utility of the proving properties of a formal specification.

Chapter 4 will focus on the software development life cycle (SDLC) and software development models which have evolved over the years.

Chapter 4
Software Development Life Cycle Models

Chapter 2 introduced requirements elicitation techniques and the SDLC, as well as software development models. The previous chapter described requirements elicitation techniques in more detail and included examples of how those techniques function. Strengths and weaknesses of each were also outlined.

The main focus of this chapter is the SDLC and how each of the various phases of a software development project functions. Various models of the SDLC are described and illustrations are included to help explain the different models, as well as the way in which each phase in the SDLC flows into the next.

The focus of chapter 5 will be a case study explaining the functioning of some of the models and techniques introduced in chapter 3 and chapter 4.

4.1 Software Development Life Cycle (SDLC)

Traditionally the SDLC of a system is divided into various phases (Schach [79], Charette [25]):

- The Requirements and Analysis phase.
- The Specification phase.
- The Design phase.
- The Implementation phase.
- The Testing phase. (Validation and Verification).
- The Maintenance phase.

The first phase of the traditional SDLC is requirements and analysis and it is during this phase that prototyping plays a major role. In chapter 3 various prototyping techniques were discussed. It is during the requirements phase of a software development project that prototypes add the most value. The requirements phase determines the requirements to be met by the product.

Specification follows the requirements phase and it is during the specifications phase that requirements are translated into a precise description of the way in which the software product should operate functionally (Charette [25]).

Traditionally, design is the third phase of the SDLC and it is during this phase that the software development team determines the internal operation of the proposed system from the specification. If, at any time, the client decides that any of the previously

determined requirements should change, this decision has to be reflected in the documentation for the system. Developers decide on the best way to build a system which will conform to the established requirements. The system should also be relatively easy to maintain and enhance. If the requirements have been determined incorrectly, it could result in a faulty design, leading to faulty implementation of the requirements with a huge cost effect in the end as clarified by Boehm [16] in an explanatory diagram [p40].

Documentation is of the greatest importance to ensure that developers do not lose control of the project, especially when they become too creative and do not keep to the previously determined requirements. Good documentation kept during each phase will help the team stay focussed on the problem and the solution which needs to be developed. Documentation is one of the strengths of the Waterfall model (Schach [79], Charette [25]), described in section 4.2.

The design phase is an equally crucial phase, especially regarding enhancement and flexibility. Should the system be designed in a way which would make any enhancements very cumbersome, it could be quite costly to change it in any way. This could result in a system which is very costly to maintain. This is one of the drawbacks of the Waterfall model (Schach [79], Charette [25]), since any change made to the system after implementation, is normally a very costly exercise. The huge cost effect of making changes late during the life cycle is well known (Boehm [16]).

One of the major problems experienced by software developers is that the client's requirements often change during the life cycle of a software development project. This could lead to an ongoing cycle of reworking, which is a costly exercise. The Standish Group [85, p6] reports as follows:*"Changes, changes, changes; they're the real killers."* This implies that continual changes could cause problems leading to over expenditures of originally estimated costs as well as late delivery. A client should sign off after each stage of the SDLC to avoid this from happening.

A document which has been signed off helps both the client and developer to commit to an agreed set of standards and requirements before the system is developed further, provided of course that the client does not sign off under pressure.

The actual coding of the system begins after the system has been designed. This falls under the implementation stage of the SDLC. During implementation developers take the design and code the relevant modules accordingly, eventually reaching the stage where the complete system has been built.

Naturally the time it takes for a system to be developed cannot always be determined precisely, as various factors which have not been provided for during the initial design of the whole system, may present themselves. As a countermeasure, various models of cost estimation such as the COCOMO model (Boehm [16]), have been developed to facilitate the estimation process in determining the cost of developing the required program. A model of this nature is not exact, but takes certain risk factors into consideration when calculating estimated costs. Models such as these, however, provide a guideline for project managers who need to determine the cost benefit of developing the program.

Should a problem which is very difficult and time consuming to overcome be encountered, it might happen that the client could stop the project, because of timeline and budgetary constraints. Alvarez [4, p2] points out that: *"…many system failures can be attributed to a lack of clear and specific information requirements. Furthermore, errors during requirements analysis that are not found until later stages of the implementation process can cost significantly more to fix"*.

Stapleton, Smith and Murphy [86, p163] maintain: "*These ambiguities and uncertainties are part of human information processing, and it is not possible to reduce or simplify them away without losing core meaning, and misunderstanding key aspects of the organizations behaviour. This has created deep problems within requirements engineering leading to many system failures."*

The validation and verification (V&V) task, part of the testing phase, follows the implementation phase. In section 2.7, chapter 2, it is indicated that V&V in this dissertation in essence refers to **testing**. During testing there are also many factors to be considered namely:

- Whether the software works correctly. (Verification -Whitebox- consideration.)
- Whether the program fulfills the requirements. (Validation -Blackbox- issue.)
- Whether the program is user friendly. (Usability.)

White and black box testing and their role in the larger V&V domain have been explained in section 2.7.

In practice, software development however, often comes with program errors (bugs). These are problems picked up in the program over time after it has been introduced into the live working environment of the client. These errors are corrected in what is called the maintenance phase of the SDLC.

Program errors could present themselves where the program does not work according to the specification (validation problem) or when the client realises the program is not functioning as expected (verification problem). If this happens, the software development team will usually investigate the problem. If the program does not function according to the specification, it is a program error and is corrected. A new software version is built and released for testing purposes. If the program, however, works correctly and the developer realises that provision was not made for the expected function of the program in the specification, the system is enhanced and a new version is built and released for testing purposes. In both situations a problem is detected and is first investigated before any changes are made to the system. Hughes and Cotterell [50, p6] state the following: *"Once the system has been implemented there is a continuing need for the correction of any errors that may have crept into the system and for extensions and improvements to the system."*

Changes to a software product after the product has been developed, are made during the maintenance phase and this could at times be quite difficult, especially if a problem in the software is caused by another part of the program elsewhere in the system Alvarez [4, p2] points out: *"Furthermore, errors during requirements analysis that are not found until later stages of the implementation process can cost significantly more to fix."* If any of the business requirements change at all, the

developer will have to investigate what impact it would have on the software product and if necessary, change the product to conform to the new requirements of the client.

The value of comprehensive documentation should not be underestimated, Hughes and Cotterell [50, p289] state: *"Documentation is an issue that is difficult and important… ."* When confronted with program errors, a developer should investigate the problem and make sure that it is indeed a problem caused by the system. In some cases the program behaves exactly as was intended (i.e. it is not a V&V problem) and this is why documentation is so important. Hughes and Cotterell [50, p289] claim: *"However, inappropriate documentation can actually be an obstacle to effective working."* If provision has been provided for a specific condition and the user experiences it as a problem, the developer could refer to the documentation to determine whether a specific requirement to deal with such a scenario by the software was indicated. Should the developer realise that it is indeed a program error, he should effect the correction required. Alternatively it may be that a possible hidden domain factor has been revealed.

Usually, in practice, the software will be released for testing purposes after quite a number of corrections have been carried out. The program should only be implemented in the live working environment of the client after it has been tested thoroughly.

For the sake of completeness it should be mentioned that programs are sometimes proved to be correct mathematically. (Formal program **verification**.) (Backhouse [6]). Proving program code as correct remains a contentious issue, especially among software practitioners in industry. Brooks [22, p16] stipulates that while *"verification might reduce the program testing load, it cannot eliminate it."* It should be noted that verification increases the workload substantially. Therefore, according to Brooks, only *"a few substantial programs have been verified"*. Brooks also mentions that *"program verification does not mean error proof programs"*, because *"…Mathematical proofs can also have errors in them"*. A criticism against testing is articulated by Dijkstra in Backhouse [6] as follows: "*Program testing can be used to show the presence of bugs, but never to show their absence*." Naturally a

mathematical proof containing an error as provided for by Brooks above cannot be considered to be a proof.

The various phases of the SDLC have been touched on briefly above and the manner in which these phases form part of various models will be discussed in the remainder of this chapter. The models introduced below have been used over time and each model presents a unique approach to software development and contains the various phases already explained in the SDLC. These models are presented in order to provide an overview of the software development process and its strengths and weaknesses. They are also presented to determine to what extent, or not, they may be able to elicit tacit knowledge and factors in the hidden domains of users and clients.

4.2 SDLC models

Various models for software development have been developed over time. Some of these are (Schach, [79], Hughes and Cotterell [50]):

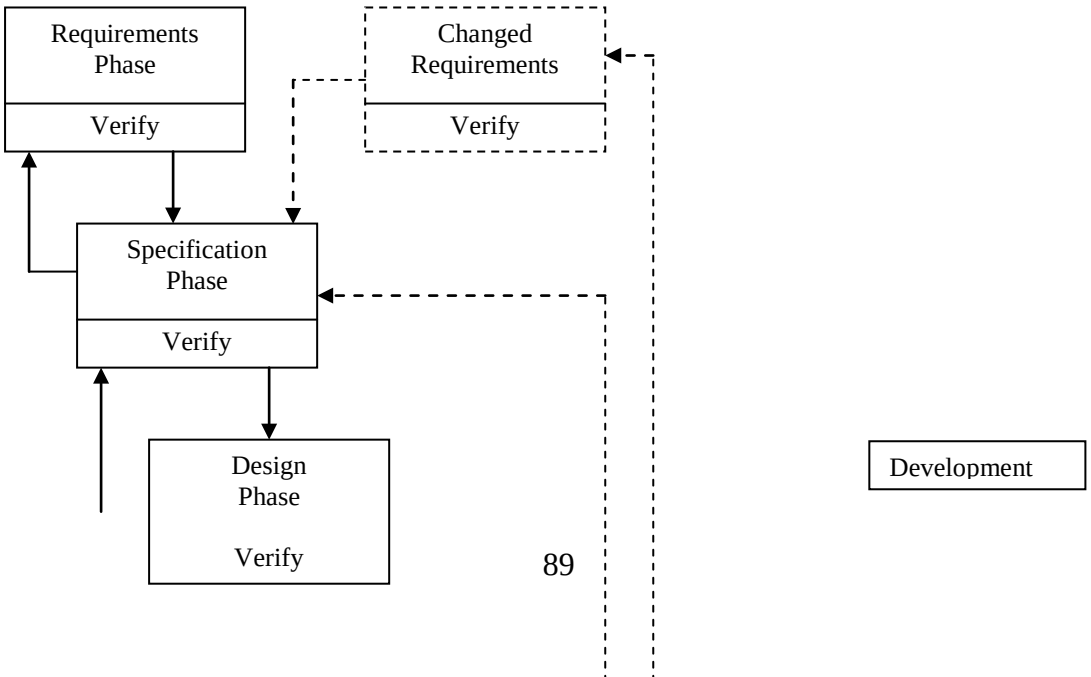
- The Waterfall model.
- The Rapid Prototyping model.
- The Build and Fix model.
- The Incremental model.
- The Spiral model.
- The V-Process model.

Each model, as well as its strengths and weaknesses, will be described in the sub-sections that follow.

4.2.1 Waterfall model

The Waterfall model (Royce [77]) is a classic model and has been used widely. Figure 4.1 shows the generic structure of this model, while depicting how each phase of SDLC (discussed in section 4.1) flows into the next. The dashed arrows pointing back

to a previous phase indicate that sometimes a change is required in the previous phase before one can advance to a subsequent phase. Each phase is validated and verified before the software development team continues to the next phase, as illustrated in figure 4.1. The **verification** (indicated in figure 4.1) between one phase and the next is related to, but clearly different from, the activities involved in the science of **formal** program **verification** covered in a number of texts, e.g. Backhouse [6]. A major strength of the Waterfall model is that each phase is thoroughly documented before continuing to the next phase.



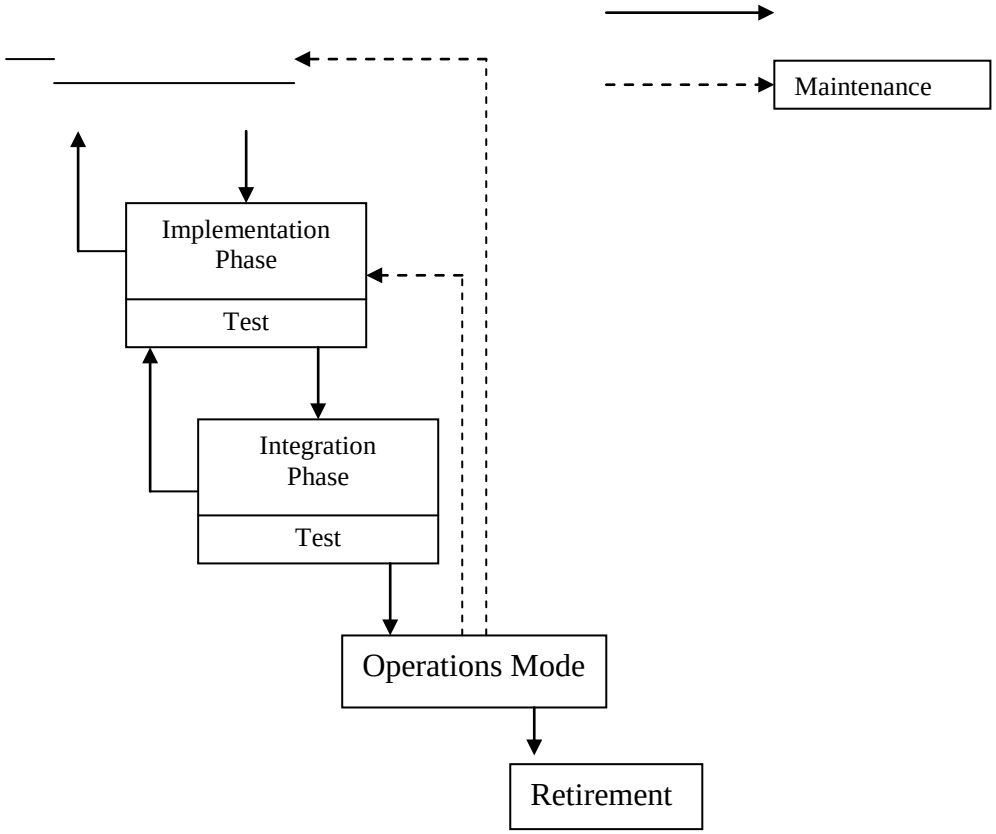


Figure 4.1 Waterfall model. (Schach [79])

The flowing of one phase to the next is the kind of continuity for which a project manager strives. Traditionally this model has worked well, because it allows for projections and timelines to be determined more accurately than would have been the case had a more iterative approach been followed (Hughes and Cotterell [50]).

The dashed arrows indicate how maintenance is achieved. Any changes made to the current system should be reflected in all relevant documentation. If this is not done, the software development team could end up in a situation where decisions are taken, but not documented. This could result in a system different from the one specified in the documentation of the system. Undocumented changes could cause huge problems later, when the system reaches a complex stage and the development team cannot recall the reason for implementing a certain solution. Furthermore, if a change is made to the system to correct an error exposed during use of the system in the live environment and this change is not documented, another team member may undo the change. This could result in a situation in which old system problems reappear after they have been corrected.

Another reason for keeping documentation current, is for validation and verification purposes, since this assists testers of the system in determining whether the system is performing as described in the documentation. If, according to the testers, the system is not functioning correctly, it is referred back to the software development team to investigate the possibility of errors. The system ought to go into operation only after it has been developed and tested (V&V). This means that some systems may take years to be developed and tested before they are actually used. The Waterfall model may therefore cause very large projects to take a long time to complete.

Strengths of the Waterfall model include (Schach [79]):

- It is a very document–driven model, enabling developers to keep a record of any enhancements or changes done to the system.
- A thorough validation and verification exercise is conducted at each stage. This helps developers while proceeding through the project as each phase is first checked to see if all requirements and problems have been addressed and resolved.

Disadvantages of the Waterfall model which often give rise to project failure have been documented over many years. These are discussed in the paragraphs below.

Building systems following the Waterfall model may involve a fair amount of risk. Should the end result not be what the client expected, it could not only be that the system might not be used, but the project might not go into its maintenance phase and could most likely be cancelled. The reason could be that the client may feel that enough resources were used without realising a return on his investment. It is unlikely that the client would start to develop a new system with the developers from the beginning again, having gone through what is sometimes called a learning curve. Unfortunately this has happened to many projects. Schach [79, p68] states: *"The waterfall model has been used with great success on a wide variety of products. However, there have also been failures."*

The above dilemma is highlighted in Schach [79] when he explains that the Waterfall model is a very documentation–driven model and that clients do not always understand very complicated technical documentation. The client, however, often signs off on those specifications as well as the technical documentation accompanying the specification, without properly understanding the contents of either (Schach [79]). The problem is, therefore, not seen at the beginning of the development of the product, but often only after the client has taken ownership of the product and then realises it is not what he needs. This, according to Schach [79, p69] is when the client utters the words: *"I know this is what I asked for, but it isn't really what I wanted."*

The Standish Group [85] identifies many factors for project failure. The most common ones of which some often surface in the use of the Waterfall model are:

- No client or user involvement.
- Little support from executive management.
- No clear statement of requirements.

When comparing the views of Schach [79] and The Standish Group [85] on project failures, it seems that the problem of the client not always getting what he needs is in many cases related to either no user involvement or no clear requirements specification or both. In essence it boils down to the tacit knowledge and factors in the hidden domains of clients and users not sufficiently well elicited.

Requirements could change during the execution of the project. Making sure that the requirements are correct is extremely important, because it is what affects the rest of the project and the complete system. This point is highlighted by Brooks [22, p17]: *"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."* Boehm's [16] explanatory diagram [p40], confirms this.

It is often hard to elicit the correct requirements, in particular any tacit knowledge and hidden domain factors from the client. In many cases clients do not understand technical questions and when an analysis is done to determine the requirements of the client, the client might not know or think of all the situations the system should be able to deal with. Prototypes may go a long way in determining requirements and could help to remove some of the confusion from the perspective of the client. However, it is plausible that not all tacit knowledge and hidden domain factors will surface through the use of a prototype.

Charette [25] also confirms that while the Waterfall model makes extensive use of documentation during the software phases, not all changes are referred back and filtered through all the necessary phases. This has the outcome that problems are left to be corrected in the maintenance phase and it could become a huge challenge depending on the impact the changes have on the system as a whole. He also mentions that the nature of software development is that requirements change during the development of a system and that factors which may influence change include, amongst others, project schedules and costs impacting on the project.

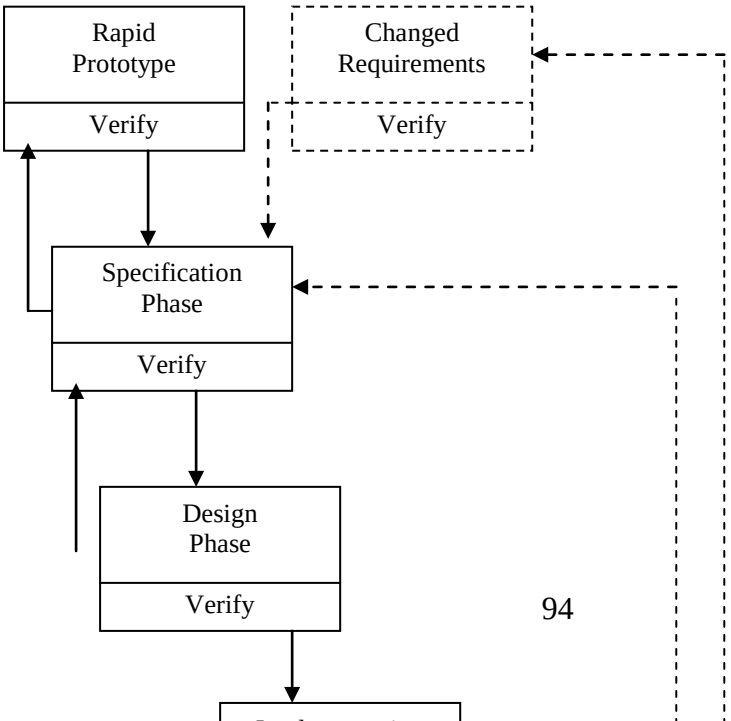
The Waterfall model, despite the many criticisms given above, presented a major advance in Software Engineering during its early years. Nevertheless, the main disadvantage of this model as far as the focus of this dissertation is concerned, is that it may, because of low client and user involvement during later phases, fail to sufficiently elicit tacit knowledge and factors from hidden domains of stakeholders.

4.2.2 Rapid Prototyping model

The Rapid Prototyping model depicted in Figure 4.2 is very similar to the Waterfall model, but instead of starting with the requirements of the client, it starts with a Rapid Prototype. This is a *"working model that is functionally equivalent to a subset of the product."* (Schach [79, p70]). The reason for building a Rapid Prototype is that users of the system can interact and experiment with the system and in doing so realise how it will function in a live working environment. Chapter 2 referred to the RAD requirements elicitation technique, which in essence is a tool-driven technique used to

develop a Rapid Prototype. This is done to assist in the process of requirements elicitation.

Brooks[22, p17] states the following: *"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines and to other software systems."* Rapid Prototyping aims to address some of the problems highlighted by Brooks [22], in that it helps the software development team and client to determine what the interfaces to people and other systems should be. In using the prototype the software development team may gain a better understanding of what it is the client requires and what the system should functionally be able to perform. However, there is no guarantee that the necessary tacit knowledge and hidden factors will be sufficiently elicited.



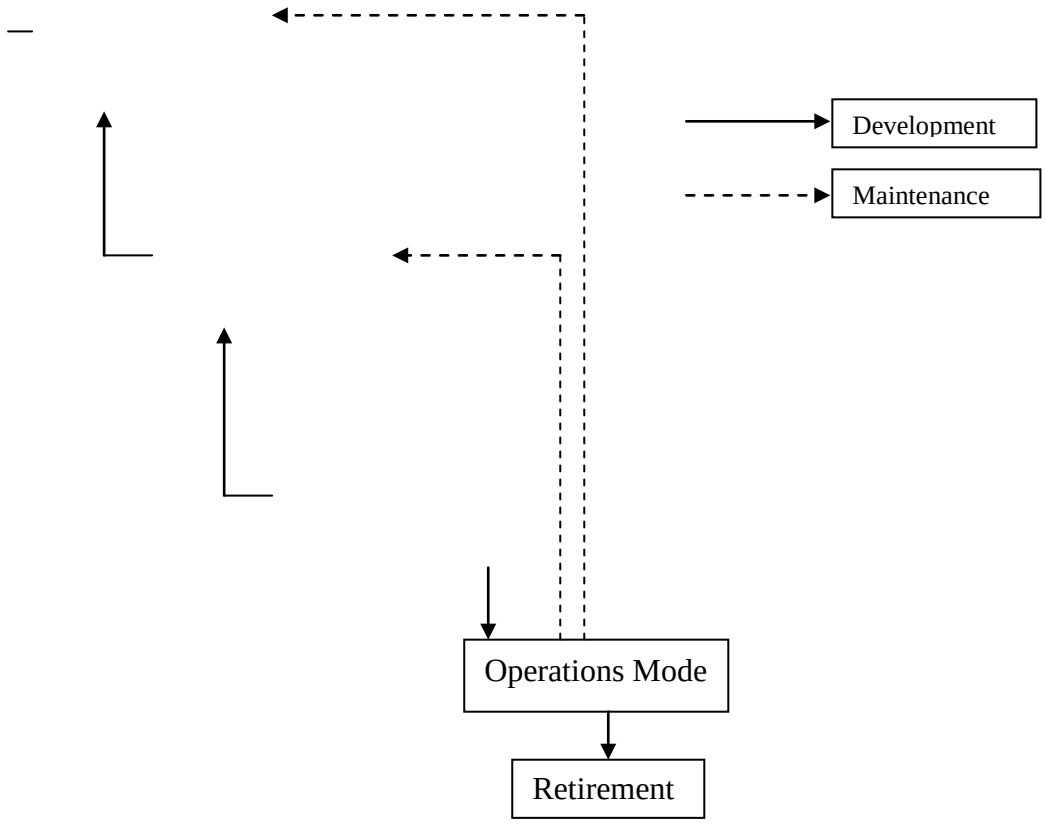


Figure 4.2 Rapid Prototyping model. (Schach [79])

A Rapid Prototype is built quickly with only the most important characteristics of the proposed system included (Brooks [22]). An accounting program to deal with accounts payable and accounts receivable, for example, might show how the program will update totals and accounts and even print a report or two, but would typically not include any database update capabilities. This is because the user is usually only interested in how the program interfaces are visually displayed and the sequences of the various processes. The main aim of a Rapid Prototype is to enable the developers to acquire the requirements for the system quickly and to ascertain whether they have captured these correctly while the users interact with the prototype.

Rapid Prototyping is a very powerful technique and because of its visual appeal can bring thought and practice together rapidly (Norman [67]). It may help to determine client and user requirements for a system more accurately. The opportunity that exists for misconceptions can be ironed out at this stage of the project, allowing the product to conform to the requirements of the client. It also enables the client to know what to expect as an end result. While a drawback of the Waterfall model is that the client

only sees the product at the end of the development life cycle (Charette [25]), Rapid Prototyping allows the client to gain a better view of the proposed system earlier.

The rest of the Rapid Prototyping model is similar to that of the Waterfall model and is conducted in the same manner, as discussed previously. Combining the two models may result in a very good methodology, as one can cover more scenarios and have improved brain storming sessions between clients, users and developers.

Developers can, however, only build what a client asks for. Most of the time, however, clients or users don't know how to ask for what they need and this is why the requirements elicitation sessions are extremely important.

The following example is explanatory in this regard: Two accounting practices may differ from each other in that, although both need a product that can deal with accounts payable and accounts receivable, one company may need an extra process using the records of accounts payable to look for marketing opportunities, while the other company may not require this. If the client does not explicitly mention the need for a report of all accounts payable at the end of every month for marketing purposes, developers will not know this and therefore not build a report for such a business case. This omission will only become evident when the client wants to issue such a report and discovers that such functionality is not available.

The above example is a very simple one and in such a case it would be quite easy to add another report and create the required results fairly quickly. There could, however, be cases which are not so simple and which could contribute to making a system obsolete quite soon. Therefore, a Rapid Prototype may equally well fail to elicit valuable tacit knowledge and hidden domain knowledge.

A weakness of the Rapid Prototype model in the opinion of the writer of this dissertation is that the prototype may become the real product. Schach [79] points out that the value of a prototype is to determine requirements, whereafter it ought to be discarded. In some cases, however, developers continue to build the real system by expanding the prototype. The problem of keeping the Rapid Prototype is that when

the prototype was being developed, non-functional requirements like flexibility and speed could have been overlooked.

A Rapid Prototype has, arguably, the best chance of all the methodologies discussed in this chapter to elicit tacit knowledge and hidden domain factors from stakeholders. The reason is that there is, much more user interaction involved. However, its very strength may also be a further weakness: Too much client and user involvement may lead to resentment. Stakeholders may feel that they are actually doing the work which the software development team is paid to do and lose interest or simply withdraw from the prototyping discussions. A way whereby this behaviour may be altered will be suggested in the SRE (Solitary Requirements Elicitation) methodology developed in chapter 6 of this dissertation.

4.2.3 Build and Fix model

Figure 4.3 illustrates the functioning of the Build and Fix model. A program or system is developed from an idea or a concept and then reworked and updated until the client and the users are satisfied. It differs from the Rapid Prototype model in that all functionality – not just core functionality – is built into the product. This model may cause problems during maintenance as very little is documented during the development of the system Schach[79, p64] states: *"The product is constructured without specifications or any attempt at design. Instead, the developers simply build a product that is reworked as many times as neccassary to satisfy the client"*. He also [p65] points out that: *"…maintenance of a product can be extremely difficult without specification or design documents…"*. Furthermore, decisions are changed on a regular basis leading to enhancements and updates being added or removed continually while the product is in the operations phase.

Schach [79] points out that most software developers relate to this model since it is the development model followed by most software products. Very fixed documentation levels are built into both the Waterfall model and Rapid Prototyping models and each phase is validated and verified before continuing to the next. This is not the case in the Build and Fix model.

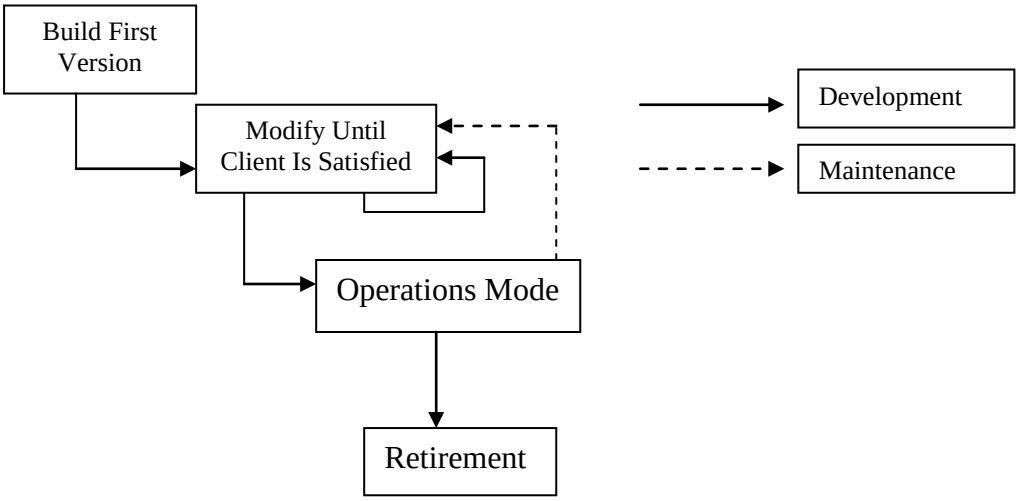


Figure 4.3 Build and Fix model. (Schach [79])

The Build and Fix model may not be an ideal one to use on large projects. The reason is that a project manager can hardly determine the actual cost of such a project if there is no specification document on which to base certain decisions. It could, for example, be extremely difficult to determine the number of resources needed for the project or the project deadline or even intermediate delivery dates. The end result may be a project running over time and over budget, costing the client much more than initially anticipated.

Some further weaknesses of the Build and Fix model are (Schach [79]):

- A product that is reworked until it functions according to the requirements of the client could become a rather expensive exercise.
- No documentation is kept. Maintaining the product could, therefore, be very difficult.
- No clear guidelines as to what needs to be developed are available, because the requirements given by the client could be vague.

The use of the Build and Fix model may lead to a successful system being developed, but, according to Schach [79], only if it is a rather small programming project, e.g. one in the region of 100 – 200 lines of code. Any project of a reasonable size needs a

proper project plan and project life cycle without which chances are that the project has the potential to end in failure.

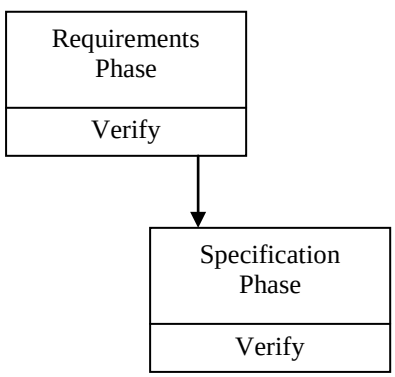
Since the Build and Fix model does not prescribe the establishment of clear guidelines as to what needs to be developed (refer to the last bulleted item above), it is difficult to conceive that much tacit knowledge and hidden domain factors could be extracted from any of the stakeholders involved during the use of this model.

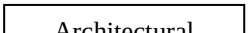
4.2.4 Incremental model

In the Incremental model a software program is built one module at a time. The complete software product is divided into smaller modules and each module is developed, tested and integrated into the complete system over time until the full system has been developed (Charette [25]). Figure 4.4 illustrates the functioning of the model. It also starts with a requirements phase followed by a specification phase as with the Waterfall and Rapid Prototyping methods.

The overall system design describing the way in which each module will function and be integrated into the core (main functioning module) of the system, is created in the architectural phase. After this phase, each module is designed, coded, and tested. If it functions as required, it is integrated into the overall system. The system is then tested as a complete program with added functionality to determine if the module is operating correctly and to ensure that the system is not causing unwanted side effects.

Each module can then be maintained and updated as required in the operations mode. This could be necessary if the product is to be enhanced or if a module is causing a problem in itself or causing the overall integrated system to malfunction.





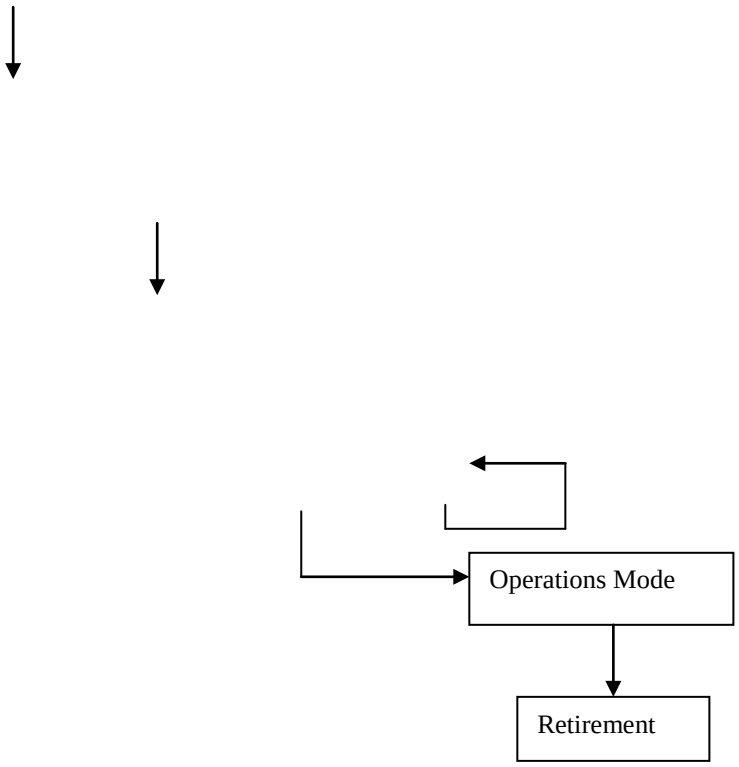


Figure 4.4 Incremental model. (Schach [79])

The Incremental model is also a powerful technique, because the method enables developers to build a module and test (V&V) it thoroughly before it is integrated into the larger system. This helps to build flexibility into the system, and facilitates the identification and subsequent correction of errors. To test a small piece of software is much easier than to test a complete program, as in the case of the Waterfall model.

The Incremental module allows a developer more flexibility and helps create a product that is able to adapt to changes more quickly, as he only needs to update a single module at a time and then integrate it into the whole system. If the updated module affects the system negatively, the developer replaces the new module with a previous version and then investigates the effect of the new module on the whole system. This approach improves stability as the module which creates a problem in the system can be identified, removed, corrected and re-integrated into the system more easily.

One of the strengths of the Incremental model is its modular approach which allows for the client to notice when the developers are on the wrong path. Corrections can be done sooner and at lower cost than with the previous models. Modules are simply changed or even cancelled and re-written before being integrated into the larger system.

In many companies the test system (the system with the latest modules integrated) is executed parallel with the currently used system and the results are validated and verified before changing over to the new system.

Other strengths of the Incremental model include (Schach [79]):

- The client's organisation receives a system over time, and this enables it to adapt to the new system gradually. There is, therefore, no traumatic experience in introducing a complete new system all at once.
- The client does not have to pay a huge amount of money for the new system all at once. Cash flow is therefore less of a problem.
- The client may put the system into practice and, although the system is given on a modular basis, work may still be performed. The system grows over time and the more functionality added by the developers, the more the client will be able to do.

Strengths according to Hughes and Cotterell [50], include:

- Feedback on early increments improves later stages.
- Changes to requirements are reduced, because product delivery time is reduced.
- Benefits reach users earlier.
- Early delivery of components improves cash flow.
- Smaller sub-projects are easier to manage.
- Developers' job satisfaction is increased, because they see their labour bearing fruits.

A weakness of the Incremental model is the possibility of its becoming a Build and Fix model, because requirements may change constantly. This could result in a scenario where modules are continually reworked. This may lead to the Incremental model having the same weakness discussed above with regards to the elicitation of tacit knowledge and hidden domain factors in the use of the Build and Fix model.

Another weakness is that if the project is not carefully monitored, the development team could be faced with another risk, namely that modules developed independently from one another might not be able to function together. This could cause the complete project to fail, as it might not be possible to incorporate these modules into one complete system. If the interface between two or more modules is ill defined, then it implies that an insufficient amount of tacit knowledge was exchanged between the programmers of the two modules.

Hughes and Cotterell [50] highlight additional disadvantages of the Incremental model:

- Software breakage, which might result in later increments requiring modifications to earlier increments.
- Developers might be more productive working on one large system than on sequences of smaller modules.

A special application of the Incremental model involves the use of Extreme Programming (XP), in which developers aim to develop code only for current requirements and clients and users may change requirements as often as needed.
Belonging to the same cluster of methodologies are the so called Agile methods.
These will be dealt with in sub-section 4.2.4.1

4.2.4.1 Extreme programming (XP) and Agile methods

Hughes and Cotterell [50, p75] refer to the tendency of XP advocates to: "*…argue that applications should be written in* ***increments*** [Bolded by writer of this dissertation] *of working software that should take a matter of weeks to complete. … The customer for the software can at any stage suggest improvements to the functionality."*

In XP small modules are built, tested and then integrated into the system. XP has the advantage that test cases are determined before any design starts, as this is on what the developers and client have agreed. This helps a new module introduced into the main system to function correctly according to test cases determined beforehand. If a new module nonetheless causes problems in the system, it can be referred back to the software development team for investigation purposes. Hughes and Cotterell [50] argue that XP is in conflict with Booch's [19] view that software development should have a preliminary design and a robust structure.

XP provides for pairs of programmers working together in the relationship of a chief programmer and pilot programmer. (Hughes and Cotterell [50]). This affords the more junior of the two programmers the opportunity to learn and experience coding while being supervised by a more experienced programmer. The advantage of having two programmers working together is that coding problems are being viewed from more than one perspective and the two developers' domains integrate more and more while they are in the process of developing the project. A disadvantage could be that one of the two programmers might not assist the other, or a situation of conflicting personalities which could cause late deliveries could arise. Naturally, personality traits play a very important role in the success of a software development project as echoed by Ally, Darroch and Toleman [1], according to Dick and Zarnett [32]. This problematic area of soft skills management is also addressed by Sukhoo et al. [87].

Problems of the XP methodology mentioned by Ally et al. [1], include:

- Many programmers resist pair programming.
- Software developers are often reluctant to share ideas.
- Developers often experience problems dealing with trust and egos.

A detailed discussion of possible conflicts and communication between team members is beyond the scope of this dissertation, but there are factors that should be taken into consideration when setting up a team to work on a project (Sukhoo et al. [87]). Internal problems could have a negative impact on a project as a whole.

Nielsen and McMunn [66] also highlight communication as a problem among team members. The communication challenge (refer to the section 2.6) described by them was significantly improved when the development team started applying Agile methods. In these methods – in which XP plays a large part, developers and users work together, (McMunn [66]). Other steps taken were to allow much more interaction between development team members and business (clients and users). This had a very positive impact on the projects developed by them. Both business and IT (section 2.6) learned more from this experience. Positive outcomes reached were faster turn around times and improved problem identification in newly released software builds.

The Agile development methodology has some similarities to the (Developer Domain Interaction) DDI methodology developed in chapter 6. In both cases business and the software development team work together, since user involvement is critical to the success of a project following either methodology (McMunn [66]). There are, however, important differences and these will be discussed in chapter 6.

Since XP and Agile are both, in essence incremental similar problems with regard to the elicitation of tacit knowledge and hidden domain factors may be experienced as referred to above.

4.2.5 Spiral model

The Spiral model has a similar flow as the Waterfall model, but with a **risk analysis** process preceding each of the stages. The Spiral model considers a greater level of detail at each stage of the life cycle of the project. This is indicated in figure 4.5 by the use of a loop or a cycle. Each stage is reviewed to establish a higher degree of

confidence in the success of the project. Commitment only takes place after the reviewing process has been completed successfully. Developers may only then continue to the next phase of the project.

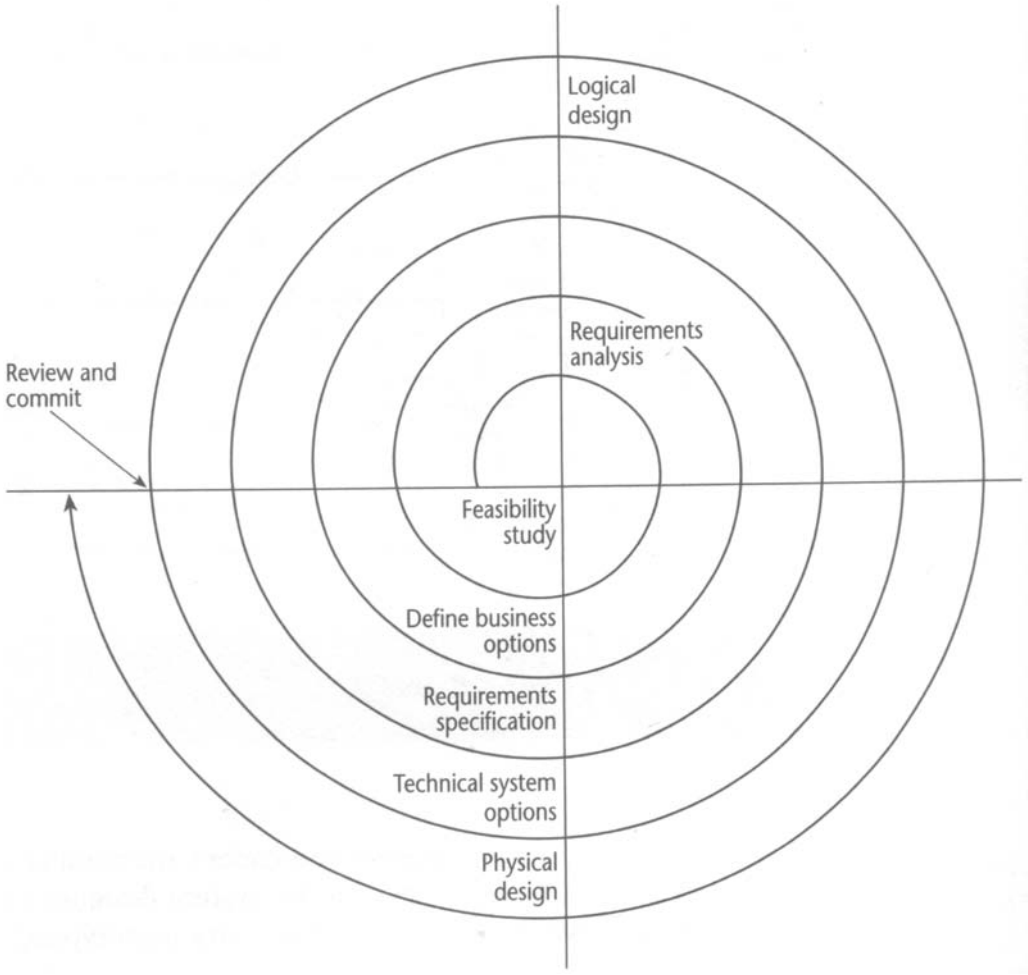
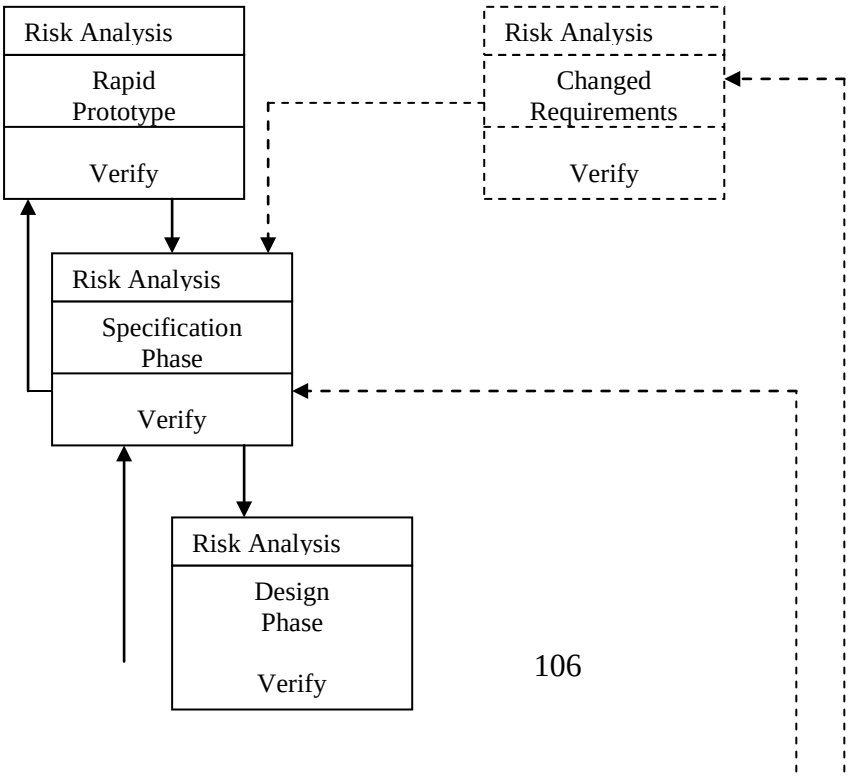


Figure 4.5 Spiral model. (Hughes and Cotterell [50])

An alternative view of the Spiral model is presented in figure 4.6. Boehm [17] refers to the way in which the Spiral model could enhance software development. Each stage in figure 4.6 is started with a risk analysis and the project will continue to the next phase only when all significant risks have been resolved. Figure 4.6 illustrates the use of the same phases as those of the Rapid Prototyping model, but with a risk analysis action included. The project could be terminated at any stage, should it not be possible to resolve a significant risk. Analysing risks at each stage of the project helps the software development team to focus on the project and its goal of building the correct solution as per the requirements of the client.

This initial risk analysis is evident in figure 4.6. If requirements change significantly and the risk analysis of feasibility provides evidence that completion will not be cost beneficial, the project could be terminated.



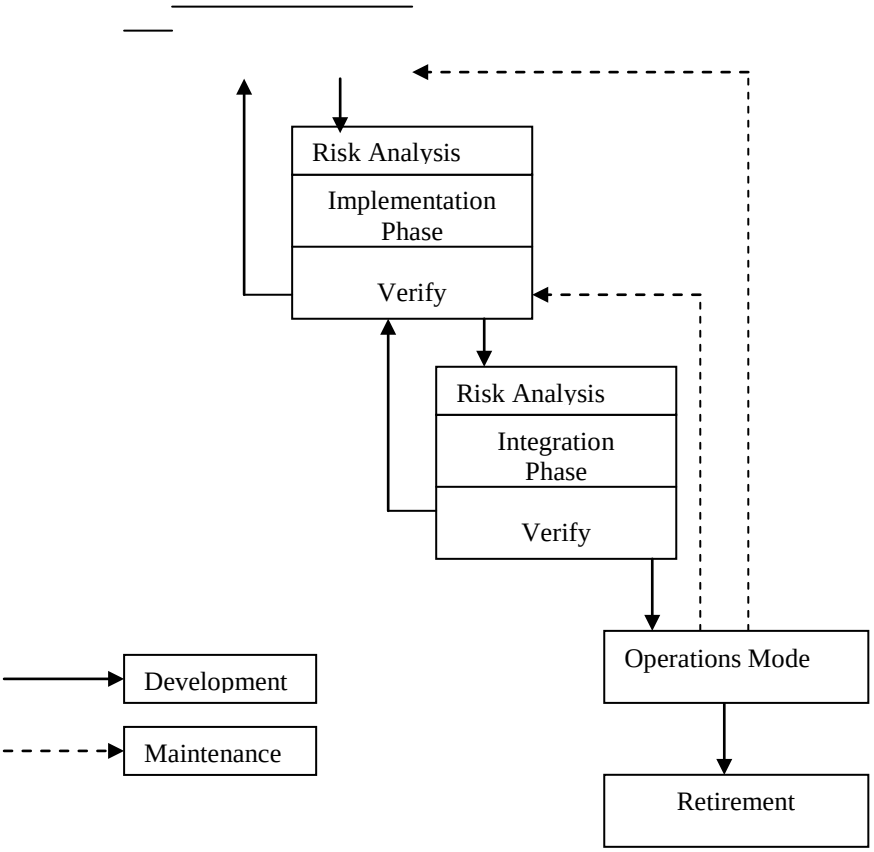


Figure 4.6 Alternative view of Spiral model. (Schach [79])

After termination of the previous project, team members may decide to start a new project as the risk analysis may have indicated that the project could still be feasible, but not in its current form. It could be possible that benefits may only be realised, should a new project be initiated. In such a case, the client could decide to continue with a new project or to terminate the project as a whole.

One of the strengths of the Spiral model is that it minimises risk in a project, as each stage is evaluated before the development team continues with the next stage.

Weaknesses of the Spiral model include (Schach [79]):

- The model is more suitable to be used for larger projects. The reason for this is that, if the risk analysis is done in each phase and the cost is higher than that of

the entire development project, it may not be feasible to analyse the risk for each stage.

- The development team may not be sufficiently skilled in identifying risks correctly. This may give the false impression that a development project is on schedule, while it is not on schedule or even feasible.

The writer of this dissertation agrees with the earlier mentioned strength of this model. It is universally accepted that a development team working on a project should have to assess risks continuously as software development is done in a very dynamic and volatile environment. This means that, while a development team is still busy working on a project, a sudden new development in technology may render the project obsolete or could enhance its success tremendously. The team should therefore be aware of such possibilities at all times.

A further weakness of the Spiral model is that it is basically a Waterfall model with a risk analysis added at the beginning of each phase. One would expect, therefore, that as in the case with the Waterfall model, much needed tacit knowledge and factors in the hidden domains of the client and users may not be elicited sufficiently.

4.2.6 V-Process model

Another model, which may be viewed as an enhancement of the Waterfall model, is the V-Process model (O'Leary T J and O'Leary L I [70], Hughes and Cotterell [50]). The main difference between the Waterfall and the V-Process model is that the former concentrates on the various **phases**, while the latter identifies the **products** at the end of each phase (NCC Education Limited [65]). Figure 4.7 is a graphic representation of the V- Process model depicting the functioning of its cycle.

Each step in the V- Process model has a matching validation process. This entails the validation of a phase after its completion. Should any problems be encountered, a corrections step is implemented. This allows the V – Process model to expand on the testing activity of the Waterfall model. Hughes and Cotterell [50, p64] states: "*The V-process model can be seen as expanding the activity testing in the Waterfall model.*

Each step has a matching validation process which can, where defects are found, cause a loop back to the corresponding development stage and a reworking of the following steps."

Only defects that are detected should be fed back (returned) and corrected. The project would otherwise run the risk of falling into an evolutionary cycle, which means that not only defects would be corrected, but changes would also be made to the product on a continuous basis.

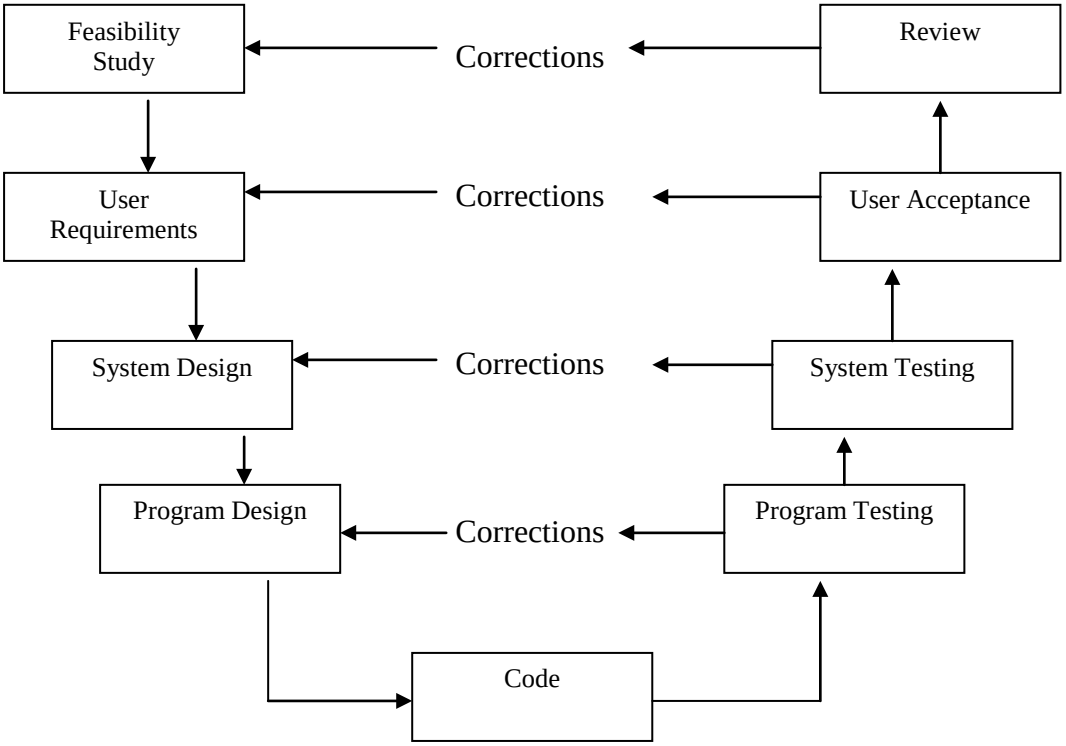


Figure 4.7 V-Process model. (Hughes and Cotterell [50])

From figure 4.7 it is clear that this model also aims at reducing risk in a software development project. Validation and verification is done at program as well as system level. If both stages provide positive results, the product is verified with the user for acceptance. If problems are encountered, they are first corrected before the development team continues with further development of the product.

The cyclical character of the V- Process model is similar to the Spiral model, which is also cyclic and has a risk analysis phase built in after each step. If a significant risk cannot be resolved in the Spiral model, the project could be heading for failure. This is similar to the V- Process model. If a defect in the model cannot be corrected, the

project could also be heading for failure. This is also true of the Incremental model which has a verifications process after each phase before the development team moves on to the next phase.

Despite the fact that the V-process model embraces the idea of a risk analysis after each phase as illustrated in figure 4.7, it may, as with the other models discussed above, fail to elicit the necessary tacit knowledge and hidden domain factors from clients and users.

4.3 Summary of characteristics of SDLC models

Table 4.1 is a summary of the different SDLC models, phases of the SDLC and which of these phases are implemented by which model.

Legend:

Req – Requirements; **Spec** – Specification; **Des** – Design; **Imp** – Implementation
Int – Integration; **Op** – Operations; **Ret** – Retirement; **Doc** – Documentation driven
Pro – Prototype/Program; **Mod** – Modular; **Risk** – Risk Assessment;
Fea – Feasibility; **Tacit** – Elicitation of tacit knowledge and hidden domain factors

SDLC Models	Waterfall	Rapid Prototype	Build and Fix	Incremental	Spiral	V-Process
Req	√			√		√
Spec	√	√		√	√	
Des	√	√		√	√	√
Imp	√	√		√	√	√
Int	√	√		√	√	
Op	√	√	√	√	√	
Ret	√	√	√	√		

Doc	√	√				
Pro		√	√		√	
Mod				√		
Risk					√	
Fea						√
Tacit	**Low**	**Med**	**Low**	**Med**	**Low**	**Low**

Table 4.1 Different SDLC models, phases and phase characteristics.

Table 4.1 illustrates the differences in each model. The Build and Fix model has the lowest number of phases and characteristics and should rather not be used in large software development projects. Unfortunately, as described previously under section 4.2.3, many projects operate according to this model. The Incremental model compares well with the Waterfall model, except that the Incremental model employs a modular approach in a software project, whereas the Waterfall model allows for a bigger implementation phase in which all of the development is first completed before integration takes place.

The last row in table 4.1 is the most significant one in the context of this dissertation. It indicates that few of these SDLC models have the ability to elicit much needed tacit knowledge and domain factors in the hidden domains of clients and users effectively.

4.4 Combining elicitation techniques and SDLC models

This section revisits the example used so far to illustrate how some of the requirements elicitation techniques in chapter 3 may be embedded in a SDLC model described in this chapter. The example used is the one to diagnose blood infections and is given as Example 4.1 below.

Example 4.1

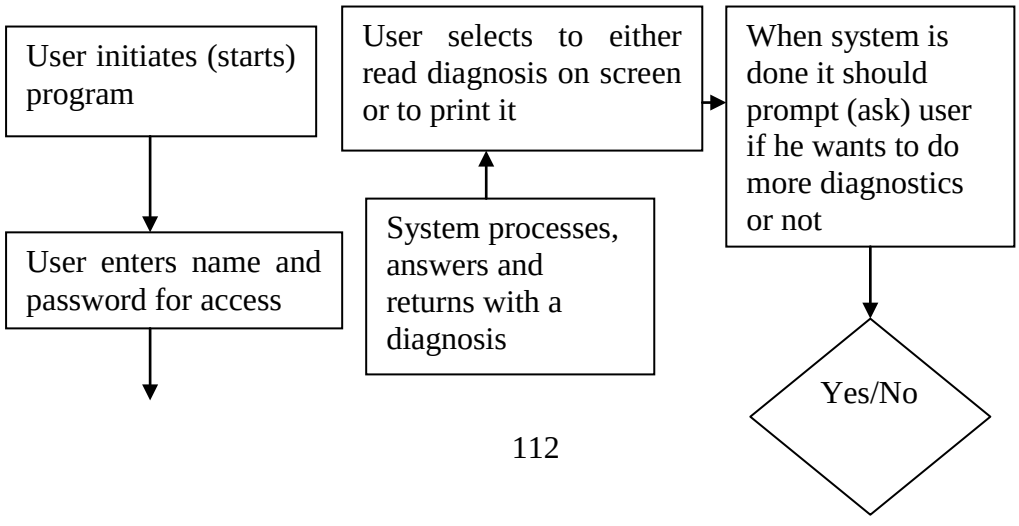
A client approaches a software development company with a request to develop a software program which will be able to diagnose blood infections based on certain criteria given to the system. The client uses Natural Language to describe the requirements of the system.

An example of a system which can diagnose blood infections and was developed and tested, is called MYCIN. (Refer to section 1.2)

The system must determine the blood infection by asking questions from the user. Each question must have certain options for answers associated with it. When the user selects an answer, the system must keep that answer and continue with more questions and answer options until the process has been completed. When the process has been completed, the user must be able to view or print the diagnosis of the blood infection.

The software development company decides to conduct a JAD workshop to help it understand the preferences of the client with respect to the interface and functioning of the product. During the JAD workshop, the developers decide to use a RAD tool to help design the user interface quickly. This enables both the client and the developer to view what the interface of the product will be. Enhancing visual clarity for both parties could help to remove some misconceptions which might arise when working from a description of the system in Natural Language.

From the JAD workshop the following flow diagram (figure 4.8) for the diagnosis program emerges:



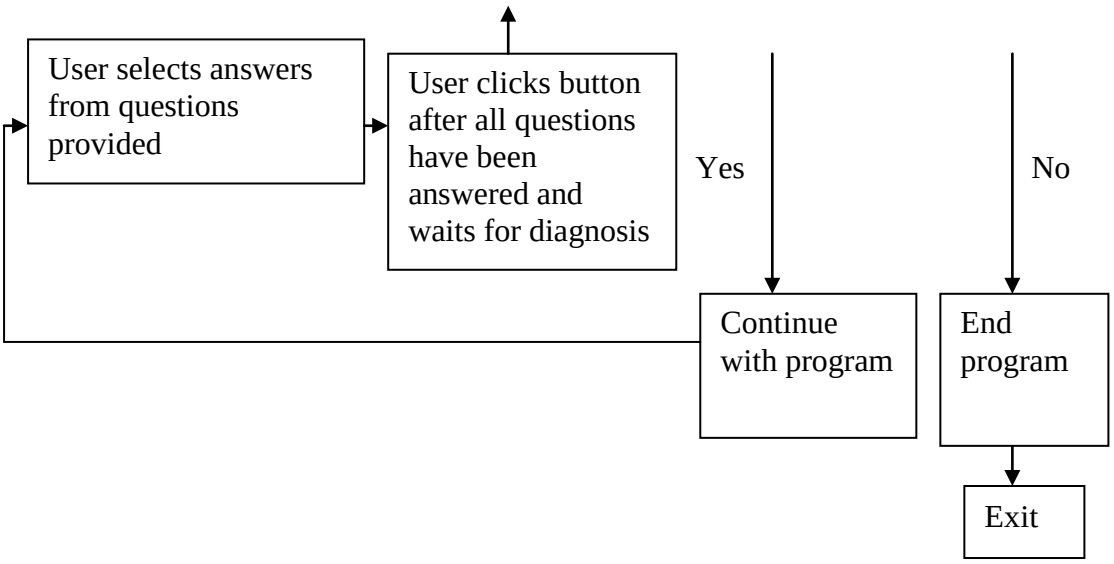


Figure 4.8 Flow of program which diagnoses blood infections.

Examples of screens developed in the workshop by using a RAD tool are shown in figures 4.9 – 4.12 below.

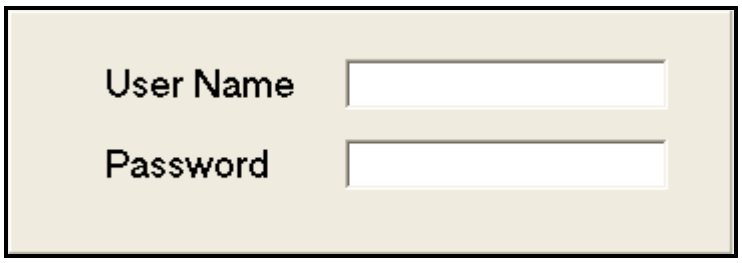


Figure 4.9 Example of user access screen of system.

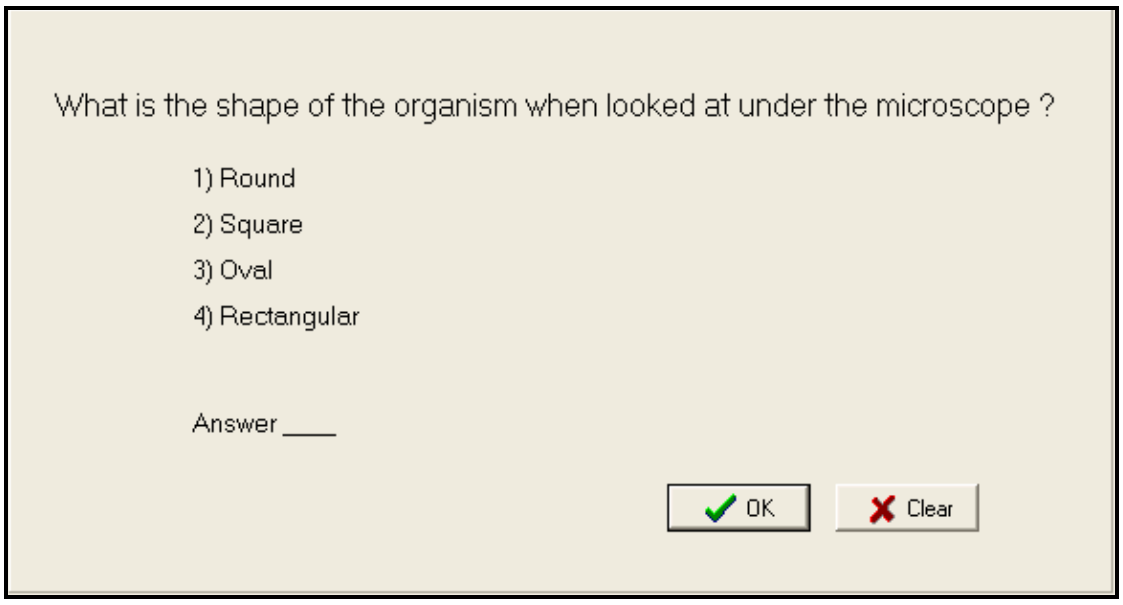


Figure 4.10 Example of question screen of system.

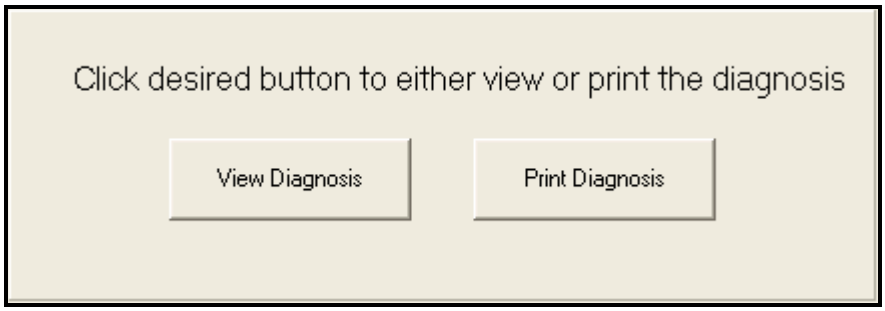


Figure 4.11 Example of user selection screen.

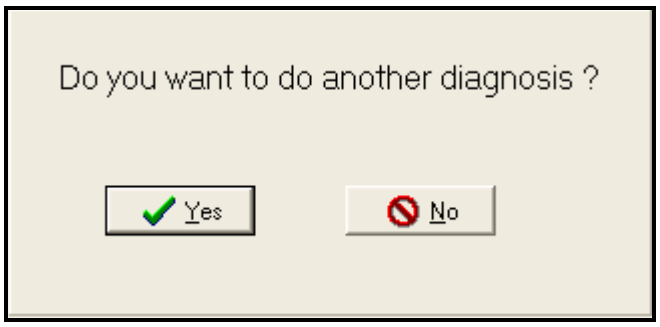


Figure 4.12 Example of user selection screen to do another diagnosis.

The development team decides that they have determined all the requirements of the client and proceeds in drawing up a specification document, which outlines precisely

how the system should function. The document is presented in a very technical format which is not fully understood by the client. He, nevertheless, signs the document. The development team proceeds and begins to design the system. The team decides on using the **Incremental model**, which enables them to make changes to a module, without affecting the entire system.

The development team starts the implementation of the product and works according to the specification document. Modules are developed and the testers test according to the specification. After the software development team has built some modules, these are implemented in the business environment of the client, where users start interacting with the system.

Over time all requests and errors are forwarded to the developers to investigate and correct if necessary. The team continues with this process until the complete system has been developed. The client is satisfied and accepts delivery of the complete system.

The system works and gives the same diagnosis as would be given by an expert on blood infections. The client can now implement the new system in places where there are no blood infection specialists. This will enable a doctor to decide how to treat a certain disease without having to wait for laboratory results which may take too much time to obtain, especially in serious cases.

All seems to be working fine and all rules have been implemented as provided by the client. **However, an error presents itself as a result of tacit knowledge in a hidden domain.** The important question of the **colour** and **shape** of the organism has been overlooked. The result is that the system does not follow a specific sequence of questions with options and therefore produces incorrect diagnostic reports. These incorrect reports could lead to death, as the wrong diagnosis could be used and the wrong medication which may not have the desired effect of destroying the virus causing the infection, may be prescribed.

Problems as described above may eventually be highlighted by the client and users, resulting in the software development team having to investigate the problem. If the

system is at fault, a correction will have to be made. If the system, however, functions correctly and developers realise that the problem has arisen because of incomplete requirements elicitation, a system enhancement needs to be made.

4.5 Summary

The focus of this chapter was on the software development life cycle (SDLC) and the various phases included in the SDLC, namely: Requirements and Analysis, Specification, Design, Implementation, Testing and Maintenance. Each of these phases can be found in the various models used in the SDLC and described in this chapter.

The models described included: The Waterfall model, the Rapid Prototyping model, the Build and Fix model, the Incremental model, the Spiral model and the V - Process model. It was discovered throughout that none of these models may elicit the necessary tacit knowledge and hidden domain factors from clients and users adequately.

An example was presented to show how requirements elicitation techniques (RAD and JAD) can be used in conjunction with a SDLC model (Incremental model) to create a software product. In the example it was shown how much needed tacit knowledge did not surface during requirements analysis but only later during the operational phase of the system.

In chapter 5 a case study for the development of a logistical distribution system will be extracted from literature and expanded to create an illustrative scenario to describe the use of different requirements elicitation techniques. In chapter 6 two methodologies, explaining the way in which tacit knowledge and hidden domain factors may be elicited will be described.

Chapter 5
Case Study for a Logistical Distribution System

Chapter 4 described the Software Development Life Cycle as well as various software development models.

Chapter 5 is based on a case study for a distribution system to be developed for a client as per his requirements. For the purposes of this dissertation, a case study carried out by Jalloul [53] has been taken as basis for illustration of the various requirements elicitation techniques discussed before. The techniques applied are: Natural Language, JAD, RAD, UML (the only technique described in this dissertation used by Jalloul [53]) and Formal Method Z. After the requirements elicitation phase has been completed, the SDLC continues. The various phases of the SDLC are then applied to the case study for illustration purposes by the writer of this dissertation.

Towards the end of this chapter consideration is given to the possibility of tacit knowledge and hidden domain knowledge not being adequately captured by the design. It is therefore indicated that these issues will be considered in chapter 6 using the same case study as vehicle of illustration. A summary concludes the chapter.

Any of the software development models described in chapter 4, namely the Waterfall, Rapid Prototyping, Incremental, Build and Fix, Spiral or the V – Process models, may be used to develop the system. Naturally, the decision on which one, or combination of techniques (refer to section 4.2) to use rests with the software development team and will be made based on what they consider to be the best approach for the specific environment.

5.1 User requirements

The case study developed below has been extracted from Jalloul [53] to sketch a scenario for a software development project. The case study is taken at face value and

then enhanced to describe how a software development project is initiated. The focus is on requirements elicitation. The problems in the case study will only be highlighted in chapter 6 once the hidden domain factors have been discovered to illustrate the tacit knowledge and hidden-domain problem discussed so far in this dissertation.

The following paragraph in Natural Language is a brief description of the client's requirements for an automated distribution system as quoted from the description of the case study in Jalloul [53, p209]:

"*DVC is a software solution for a given company. This company is a reseller of goods. It obtains its products from a vendor and resells them to a customer. The basic need of the company is a database to store specific information concerning the various vendors, customers and warehouses as well as a product database. In order to make the lives of the employees easier, the software features an easy-to-use interface for the database, which includes interactive forms and graphics. Using the information in the database, the user can access the information in several easy ways, such as browsing and searching.*"

5.2 Joint Application Design (JAD)

The above description was the requirements given for the system to be developed. The case study in Jalloul [53] is now used to illustrate the first of the phases referred to in the SDLC, namely the requirements phase.

In the illustrative scenario, the software development team decides a better understanding of what the system should be able to do functionally, is required. They decide to start with a JAD session during which the team could ask questions and interact with the client. This would help them in the requirements phase.

In the JAD session the software development team asks the following questions:

- What information should be kept for a customer of the client?
- What information about the vendor should be kept?

- What information about the product should be kept?
- What information about the warehouse should be kept?
- What information about an employee of the client should be kept?
- What information is important for an order to be processed?

These questions will be dealt with in the sub-sections to follow.

5.2.1 What information should be kept for the customer?

Answers given are, with reference to Jalloul [53]:

Customer Contact Detail Information:

- **Company name.**
- **Contact ID.**

  This a unique number used for customer identification purposes.
- **Address.**
- **Phone number.**
- **Fax number.**
- **E-mail.**
- **Website.**
- **Contact notes**.

5.2.2 What information about the vendor should be kept?

Answers given are, with reference to Jalloul [53]:

Vendor Contact Detail Information:

- **Company name.**
- **Contact ID.**

  Unique number used for vendor identification purposes**.**
- **Address.**
- **Phone number.**
- **Fax number.**
- **E-mail.**
- **Website.**
- **Contact notes.**

5.2.3 What information about the product should be kept?

Answers given are, with reference to Jalloul [53]:

Product Information:

- **Product code.**
- **Product name.**
- **Manufacturer.**
- **Shelf life.**
- **Storage required.**
- **Price per unit.**
- **Handling cost.**
- **Cost per unit.**

5.2.4 What information about the warehouse should be kept?

Answers given are, with reference to Jalloul [53]:

Warehouse Information:

- **Warehouse name.**
- **Contact ID.**

  Unique number used for warehouse identification purposes.
- **Address.**
- **Phone number.**
- **Fax number.**
- **E-mail.**
- **Website.**
- **Contact notes.**

5.2.5 What information about an employee should be kept?

Answers given are, with reference to Jalloul [53]:

Employee Information.

- **First name**.
- **Last name.**
- **Salutation (Title).**
- **Address.**
- **Phone number.**
- **Fax number.**
- **E-mail.**
- **Position/Job title.**

5.2.6 What information is important for an order to be processed?

Answers given are, with reference to Jalloul [53]:

Order Information for Purchase and Sales Orders:

- **AFS invoice number.**
  Audited Financial Statement invoice #.
- **AFS product code.**
  Audited Financial Statement product code.
- **Purchase order date.**
- **Sales order date.**
- **Purchase order number**.
- **Sales order number.**
- **Required by.**
- **ETA.**
  Estimated time of arrival.
- **ETD.**
  Estimated time of delivery.
- **Units on order.**
- **Price.**
  Price at which merchandise is sold to a customer.
- **Cost.**
  Cost at which merchandise is bought from a vendor.
- **Handling.**
- **Status.**
- **Ship date.**
- **Shipped to.**
- **Shipped via.**
- **Shipping cost.**
- **Payment terms.**
- **Vendor ID.**
- **Customer ID.**

It becomes clear from answers given during the JAD session that provision should be made to deal with purchase orders and sales orders separately. The team bears this in mind.

After the questions and answers above have been documented in the illustrative scenario, the software development team continues to determine client requirements by using Use Case models, which form part of UML (Unified Modelling Language).

The Use Case models drawn during the JAD session are described in the sub-section below.

5.3 Use Case diagrams as used as part of UML in JAD session

The following Use Case models are drawn during the illustrative JAD session. The first Use Case model (figure 5.1) is a top level model of the system while the other Use Case models (figures 5.1a to 5.1g) are sub-models of the top level Use Case model used.

5.3.1 Top level Use Case model

In the JAD session the development team produces the Use Case model in figure 5.1 to indicate how an employee of the client, for whom the system is built, would interact with the system as a whole.

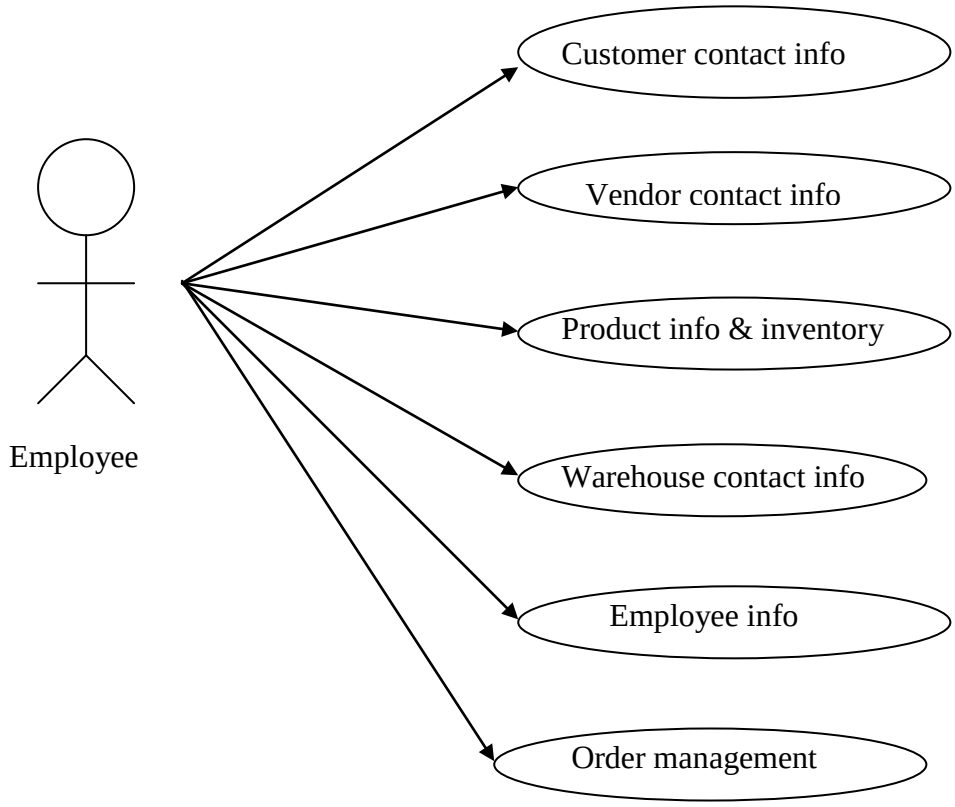


Figure 5.1 Use Case model for employee interaction with whole.

5.3.1.1 Customer contact information Use Case model

To process customer contact information requires: Company name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes. The diagram below illustrates how an employee would interact with the system while processing customer information.

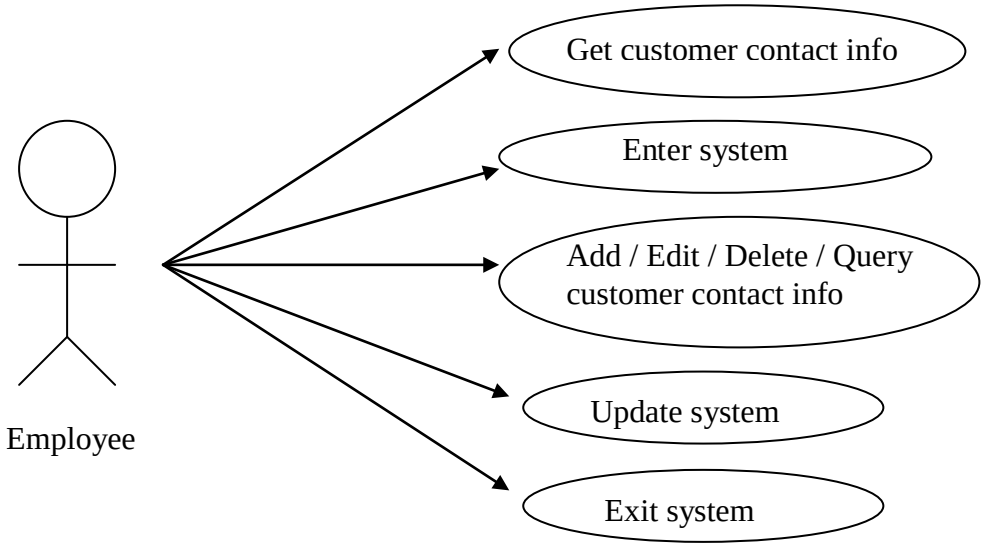


Figure 5.1a Use Case model of processing customer information by employee.

5.3.1.2 Vendor contact information Use Case model

To process vendor contact information requires: Company name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes. The diagram below illustrates how an employee would interact with the system while processing vendor information.

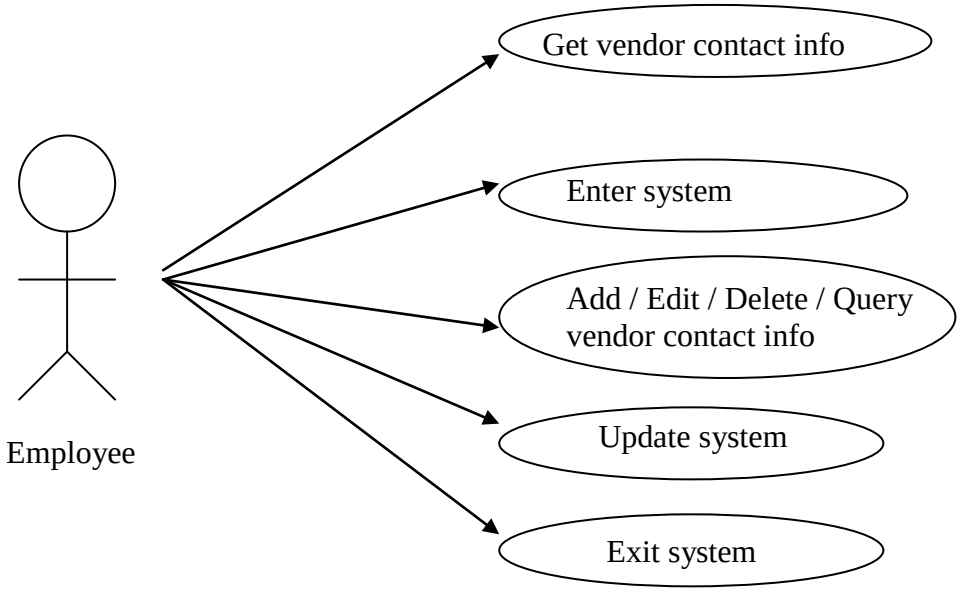


Figure 5.1b Use Case model of processing vendor information by employee.

5.3.1.3 Product information Use Case model

To process product information requires: Product code, Product name, Manufacturer, Shelf life, Storage required, Price per unit, Handling cost and Cost per unit. The diagram below illustrates how an employee would interact with the system while processing product information.

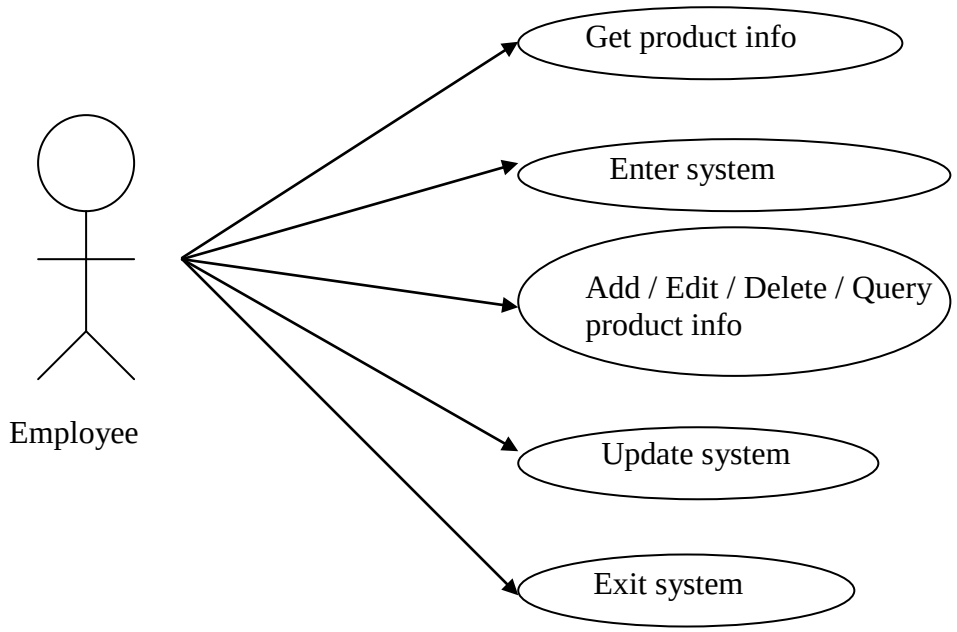


Figure 5.1c Use Case model of processing product information by employee.

5.3.1.4 Warehouse contact information Use Case model

To process warehouse contact information requires: Warehouse name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes. The diagram below illustrates how an employee would interact with the system while processing warehouse information.

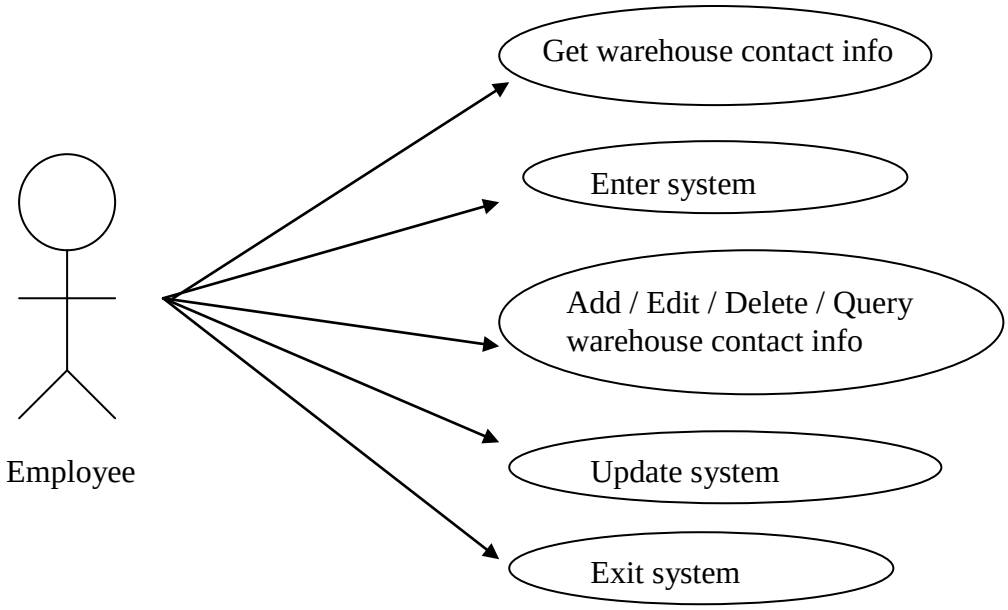


Figure 5.1d Use Case model of processing warehouse information by employee.

5.3.1.5 Employee information Use Case model

To process employee information requires: First name, Last name, Salutation (Title), Address, Phone number, Fax number, E-mail, Position (Job Title). The diagram below illustrates how an employee would interact with the system while processing employee information.

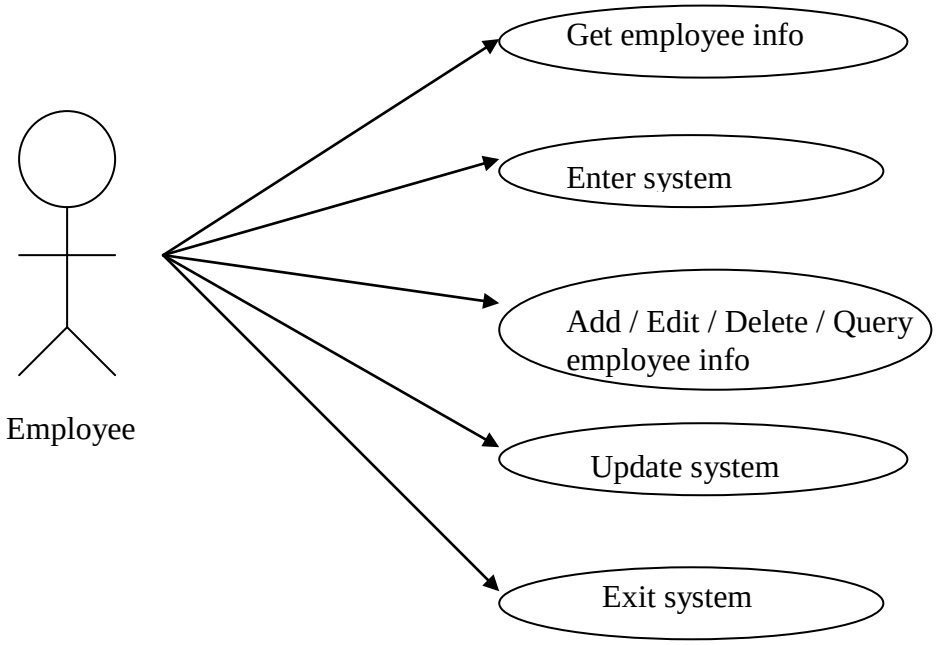


Figure 5.1e Use Case model of processing employee information.

5.3.1.6 Purchase order information Use Case model

As pointed out before, it becomes clear from answers given during the JAD session that provision should be made to deal with purchase orders and sales orders separately. The team has borne this in mind and therefore two separate Use Case models are drawn.

To process purchase order information requires: AFS invoice number, AFS product code, Purchase order number, Purchase order date, Required by, ETA, ETD, Units on order, Cost, Handling, Status, Ship date, Shipped to, Shipped via, Shipping cost, Payment terms, Vendor ID. The diagram below illustrates how an employee would interact with the system while processing purchase order information.

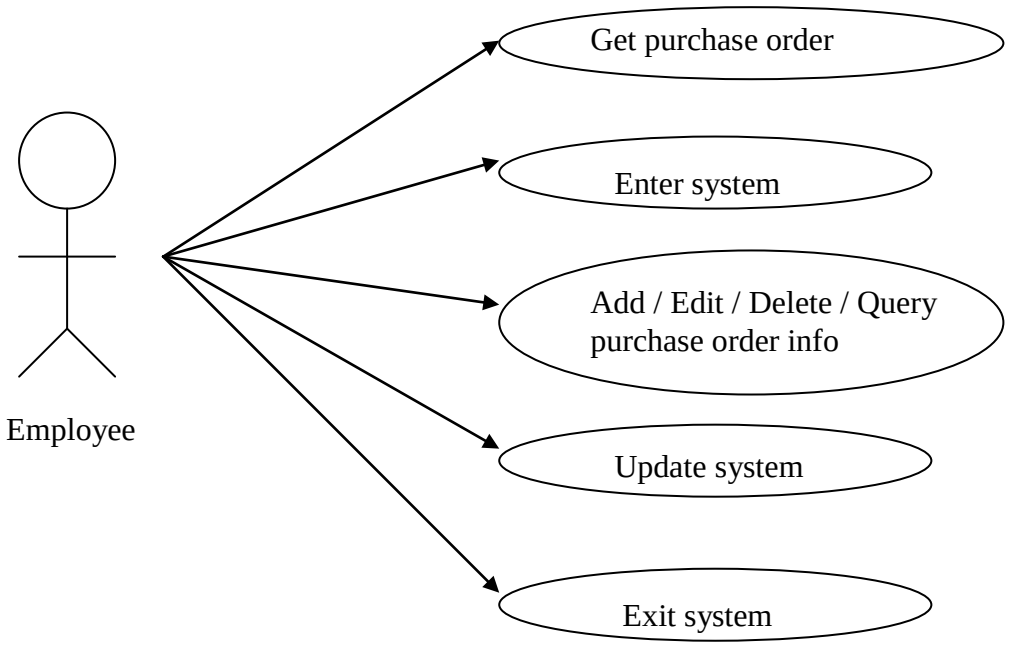


Figure 5.1f Use Case model of processing purchase order information by employee.

5.3.1.7 Sales order information Use Case model

To process sales order information requires: AFS invoice number, AFS product code, Sales order number, Sales order date, ETA, ETD, Price, Handling, Status, Ship date, Shipped to, Shipped via, Shipping cost, Payment terms, Customer ID. The diagram

below illustrates how an employee would interact with the system while processing sales order information.

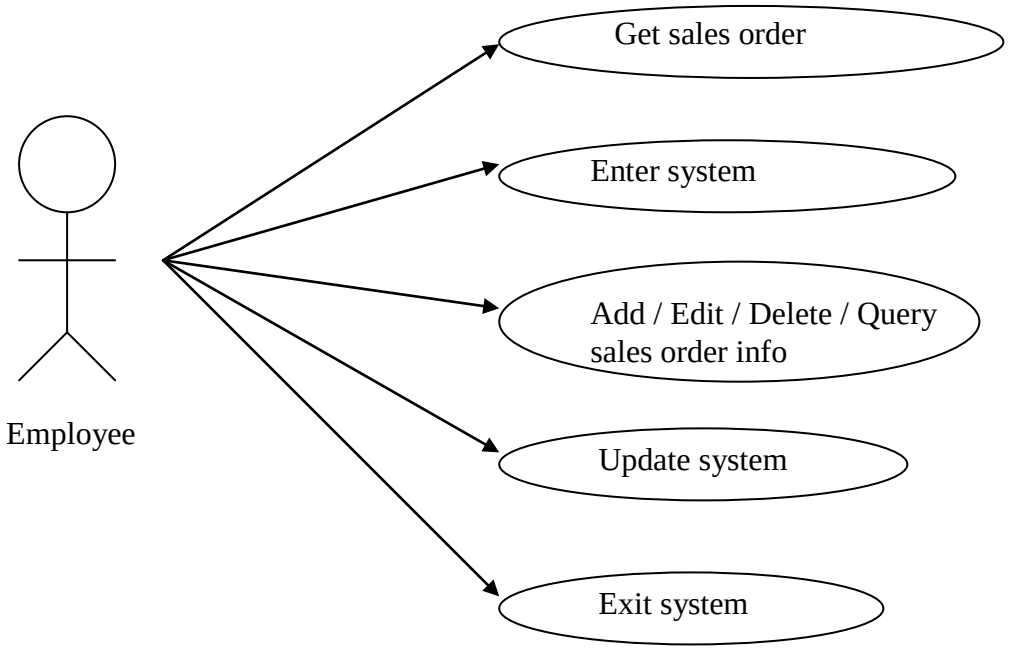


Figure 5.1g Use Case model of processing sales order information by employee.

After the Use Cases have been drawn and inspected, a class diagram (Object model) of the system is created in UML by the software development team. Note that an object model normally forms part of the design phase of a system, but it is shown here to clarify the system for the client.

This model extracted from Jalloul [53] is portrayed in the sub-section below:

5.4 UML object model

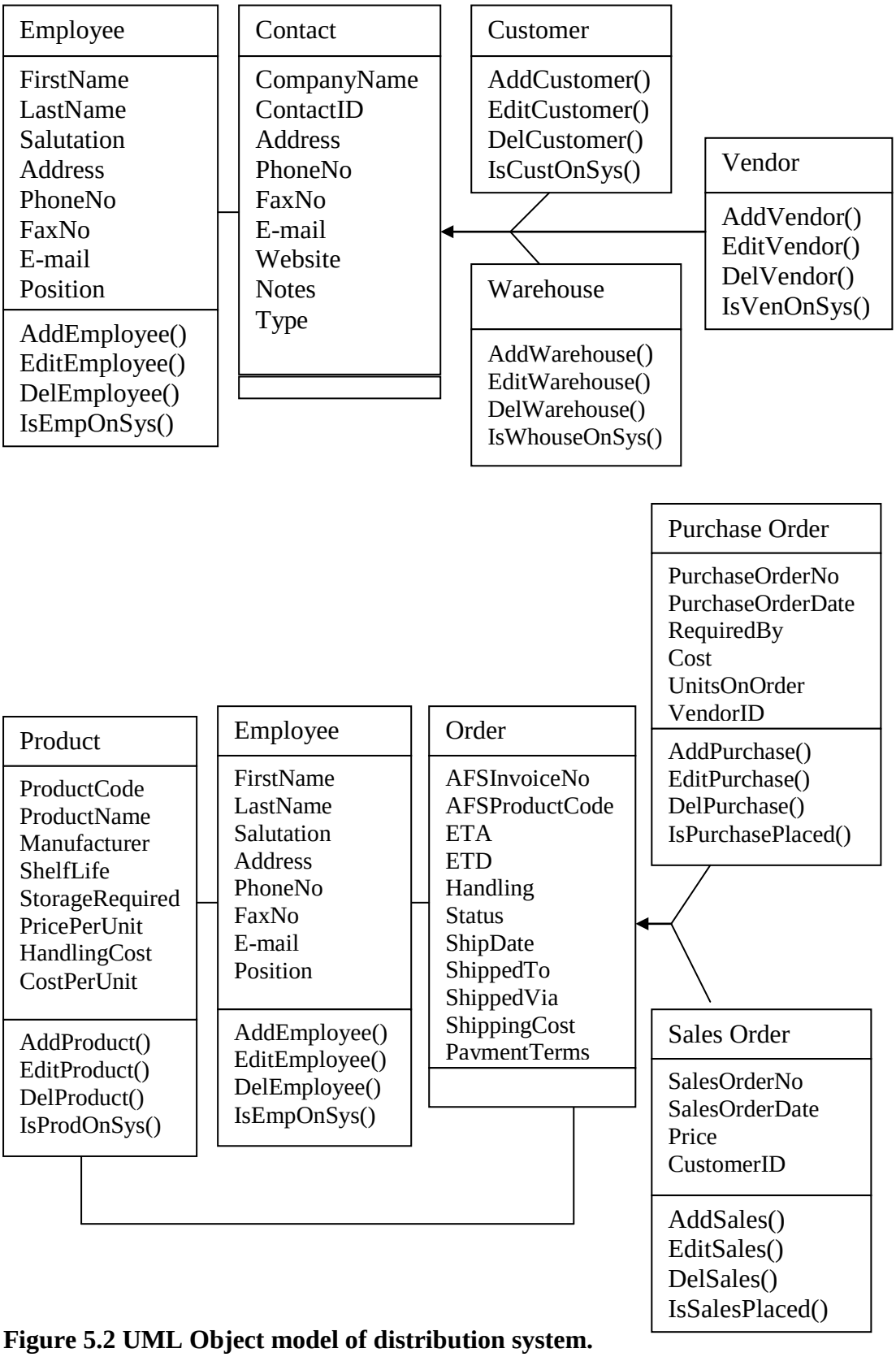


Figure 5.2 UML Object model of distribution system.

It is possible that the relations in figure 5.2 may not be in 3NF (Third Normal Form), which is the form often accepted as being sufficient to work with in practice. However, since the topic of normalisation (Coronel and Crockett [29]) is largely beyond the scope of this dissertation, we shall not pursue this matter further.

Two employee classes are shown in figure 5.2 to indicate the link between employee and contact details for customers, vendors and warehouses, as well as between employee and order and product. Only an employee may change contact details, orders and products. Purchase orders and sales orders are sub-classes of super class Order, because each is a specialised type of Order. The same applies to Customer, Vendor and Warehouse, which are sub-classes of the super class Contact.

During the illustrative JAD session the software development team decides that it would be beneficial to develop a prototype of the proposed system. They resolve to use a RAD (Rapid Application Development) tool for this purpose. This is described next.

5.5 Rapid Application Development (RAD)

The following screens are developed:

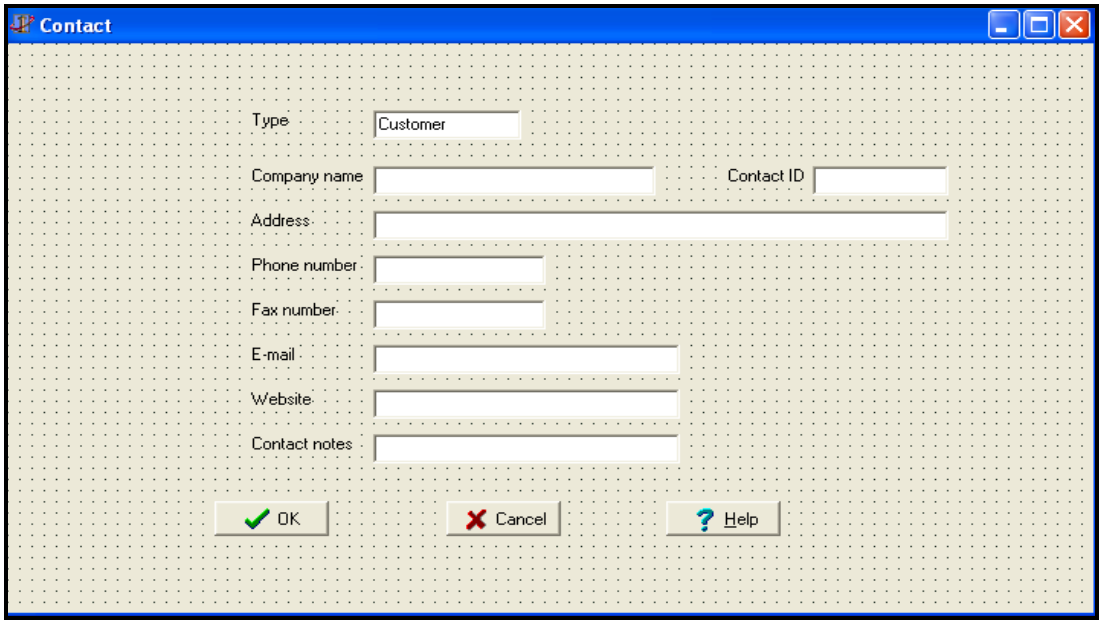


Figure 5.3a Screen displaying customer information.

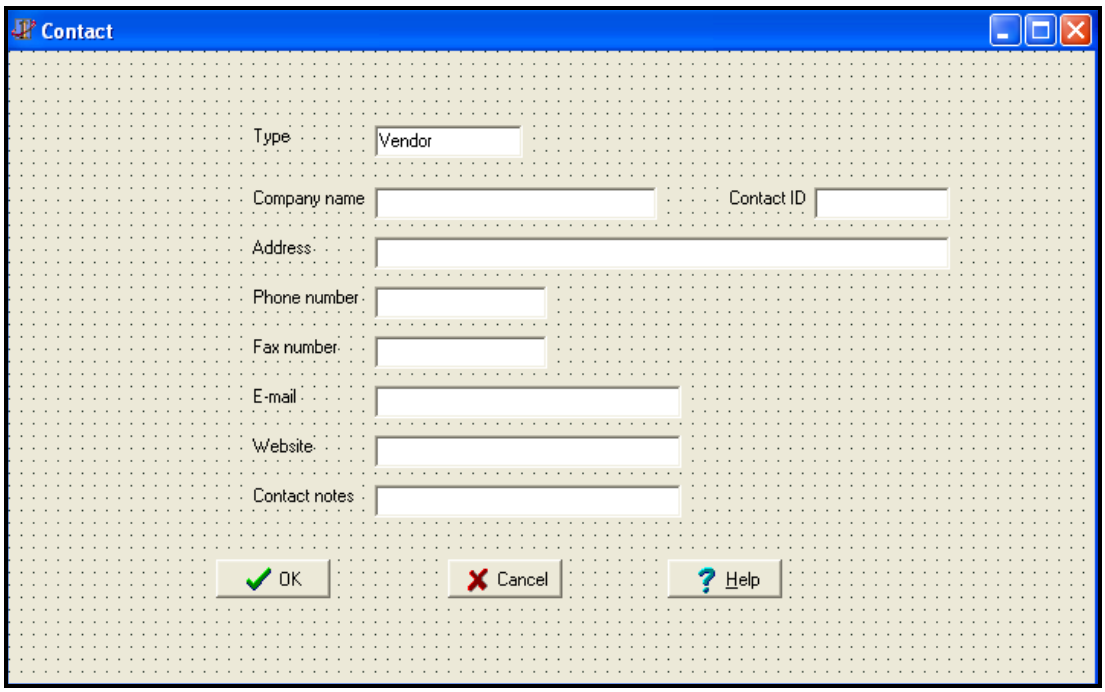


Figure 5.3b Screen displaying vendor information.

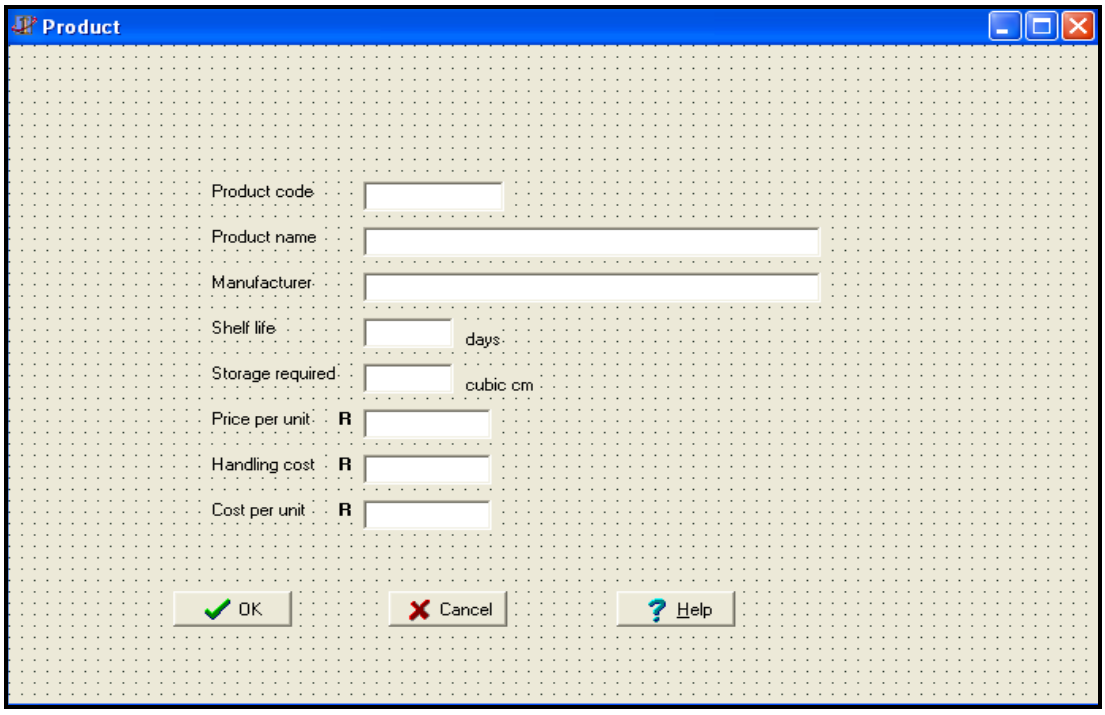


Figure 5.3c Screen displaying product information.

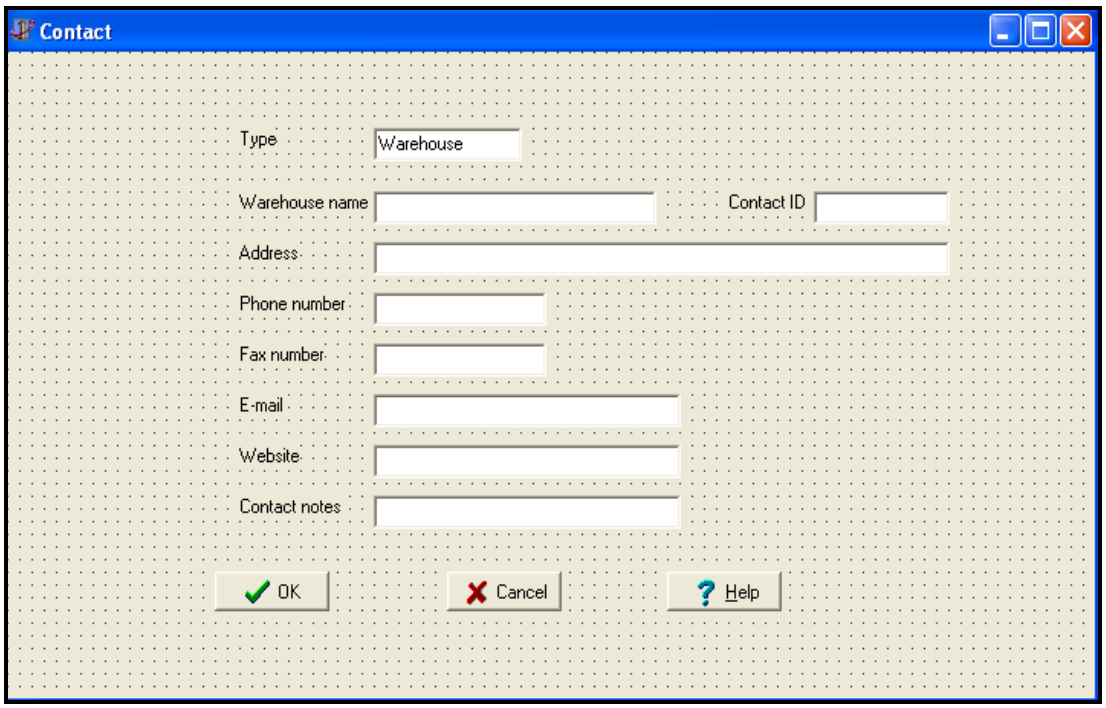


Figure 5.3d Screen displaying warehouse information.

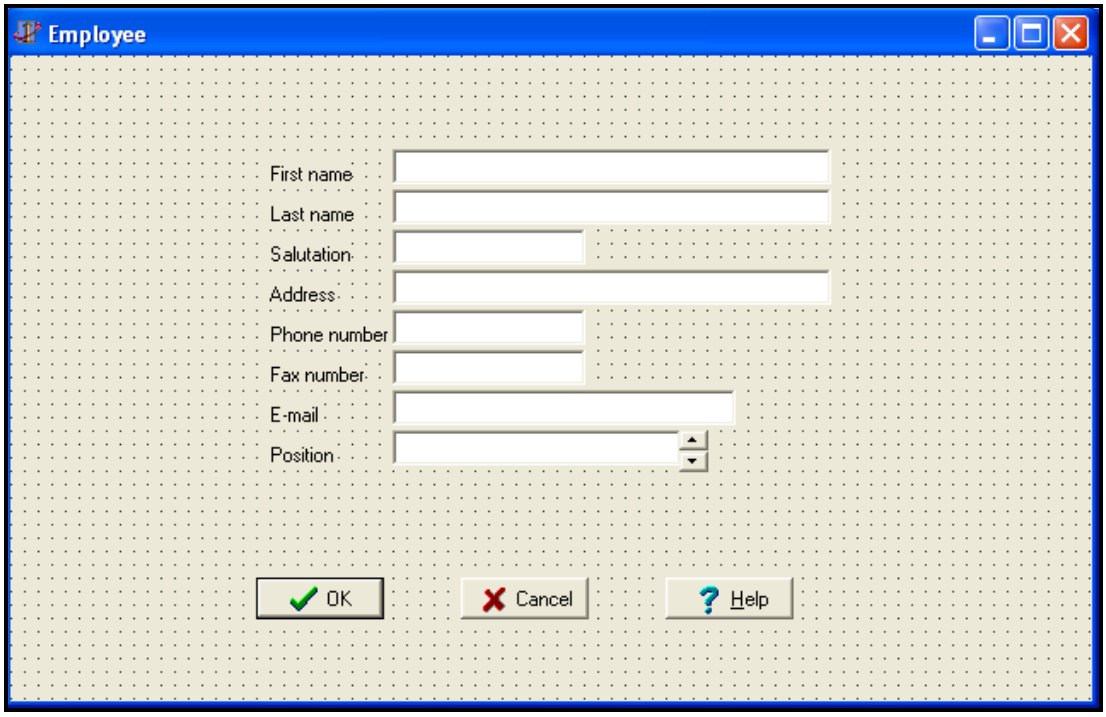


Figure 5.3e Screen displaying employee information.

As there are many similarities in the contact details of Vendor, Customer and Warehouse, the team decides to build one screen, which could show and hide attributes (fields) depending on the contact type selected.

As there are many similarities in purchase and sales orders, the team decides to build one screen to be able to cater for both order types.

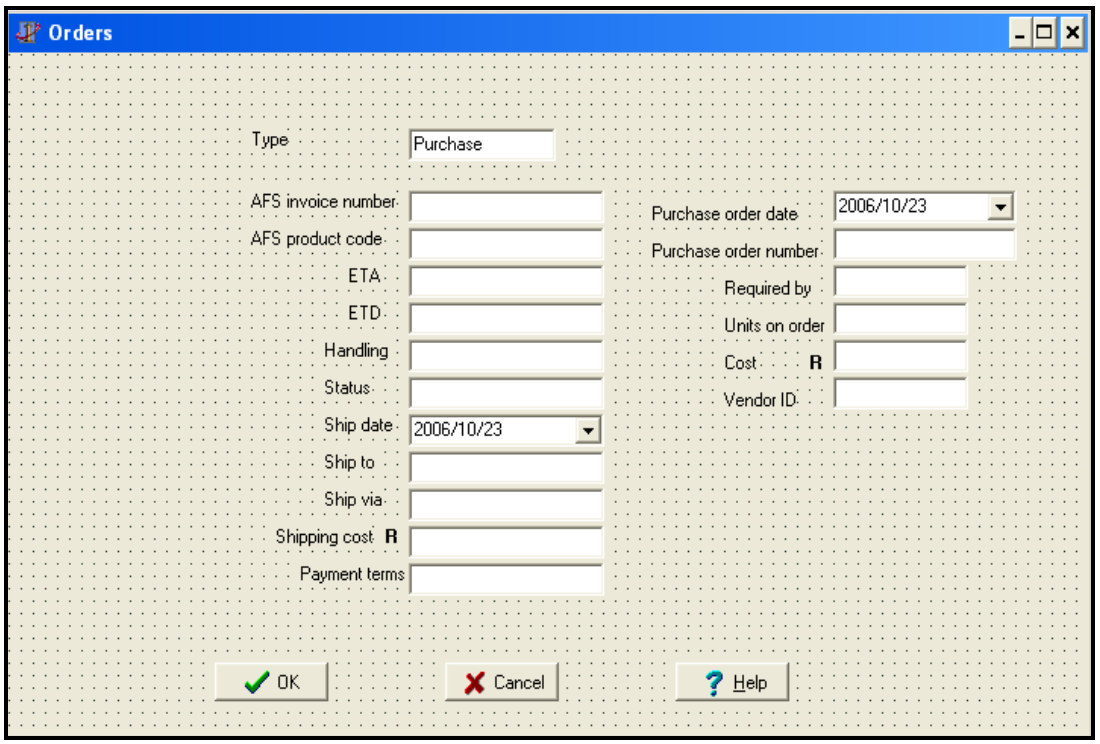


Figure 5.3f Screen displaying purchase order information.

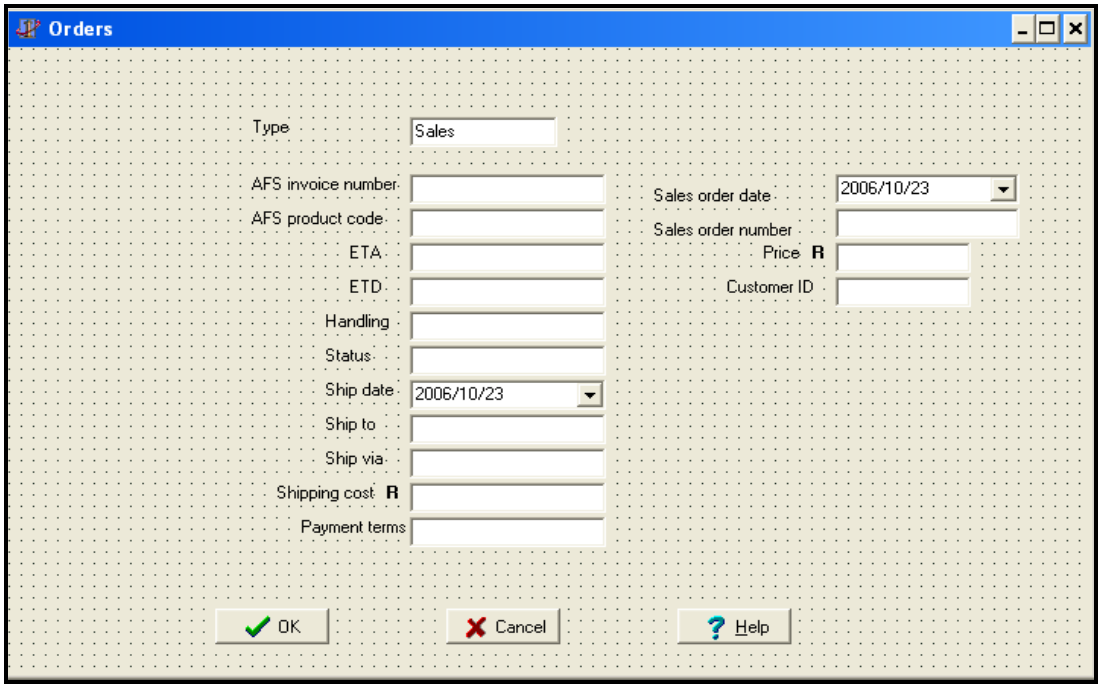


Figure 5.2g Screen displaying sales order information.

Once the user interface screens have been developed in this illustrative case study scenario, the software development team decides on another JAD session with the client, to ensure that the requirements determined in the first JAD session are clearly understood. It will also afford the opportunity to deal with any amendments, should there be any. JAD sessions will continue to be held until it is believed that all requirements are clearly understood. The number of these sessions may vary from project to project as each project has a different level of complexity. Hughes and Cotterell [50, p63] point out: *"In these workshops, developers and users work together intensively for say, three to five days and identify and agree fully documented business requirements."* Note that despite this rather confident claim by Hughes and Cotterell, valuable tacit knowledge and factors in hidden domains may still not be fully explored during a JAD session, as was argued in section 4.3.

At this point, the client agrees that the screens are a correct reflection of his requirements. He also indicates that they are very informative and user friendly, displaying all relevant information to be viewed. The client studies the basic functionality of the opening and closing of the screens and also remarks that they are easy to comprehend. Subsequently the client agrees that the software development team may continue. He believes that the software development team has understood the requirements fully.

At this stage the system does not include any functionality, e.g. it does not save any files nor does it populate or update any database records, because it is only a prototype. Functionality of such a nature would only be included in the final product.

The client then signs the requirements phase of the project off and the software development team continues with the specification phase which is described below.

5.6 Formal Specification using Z

The specification phase follows the requirements phase. (Refer to section 4.1). It is in the specification phase that requirements are translated into a precise description of what is expected from the software product functionally (Charette [25]).

The software development team decides to use Formal Method Z to develop the specification for the distribution system. The specification they develop is given below:

5.6.1 Logistical distribution system

A number of basic types are defined. The following is a deviation of the traditional way in which Z defines a type. Normally a type like CUSTOMER below would not be broken down any further, but to fit in with the UML object diagram and also the screens defined above, these types are broken down by showing their detailed attributes. In Z these could typically be shown through the use of a Cartesian product. For details about this technique the reader is referred to Van der Poll and Kotze [89].

For example the type CUSTOMER could be defined as:

CUSTOMER ≙ Company_Name × Contact_ID × Address × Phone_number × Fax_number × E-mail × Website × Contact_notes

However, in order to keep the specification simple, the above Cartesian product method is not followed in this specification. Instead CUSTOMER is assumed to be a **basic** type and its attributes are simply listed in brackets as shown below.

[CUSTOMER] - The set of all customers (Company name Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes)

The same approach is followed for the other basic types. These are given below:

[VENDOR] - The set of all vendors (Company name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes)

[PRODUCT] - The set of all products (Product code, Product name, Manufacturer, Shelf life, Storage required, Price per unit, Handling cost, Cost per unit)

[WAREHOUSE] - The set of all warehouses (Warehouse name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes)

[EMPLOYEE] - The set of all employees (First name, Last name, Salutation (Title), Address, Phone number, Fax number, E-mail and Position (Job title))

[PURCHASEORDER] - The set of all purchase orders (AFS invoice number, AFS product code, Purchase order number, Purchase order date, Required by, ETA, ETD, Units on order, Cost, Handling, Status, Ship date, Shipped to, Shipped via, Shipping cost, Payment terms, Vendor ID)

[SALESORDER] - The set of all sales orders (AFS invoice number, AFS

product code, Sales order number, Sales order date, ETA, ETD, Price, Handling, Status, Ship date, Shipped to, Shipped via, Shipping cost, Payment terms, Customer ID)

Next the software development team defines the inputs to the system:

Customer? – Variable holding the customer's details.
Vendor? – Variable holding the vendor's details.
Product? – Variable holding the details of the product.
Warehouse? – Variable holding the details of the warehouse.
Employee? – Variable holding the employee's details.
PurchaseOrder? – Variable holding the details of the purchase order.
SalesOrder? – Variable holding the details of the sales order.

The data type to be used for communicating with the user of the system is:

REPLY ::= Yes | No | employee_not_found | employee_added | employee_deleted | employee_updated | product_not_found | product_added | product_deleted | product_updated | order_not_found | order_added | order_deleted | order_updated | vendor_not_found | vendor_added | vendor_deleted | vendore_updated | customer_not_found | customer_added | customer_deleted | customer_updated | warehouse_not_found | warehouse_added | warehouse_deleted | warehouse_updated | view_record | cannot_add_record_exists

The state space follows below.

¿___DistributionSystem_____
¸

Customer: ℙCUSTOMER
Vendor: ℙVENDOR
Product: ℙPRODUCT
Warehouse: ℙWAREHOUSE
Employee: ℙEMPLOYEE
PurchaseOrder: ℙPURCHASEORDER
SalesOrder: ℙSALESORDER

The system operations are defined below:

Init_DistributionSystem

DistributionSystem'

Customer' = 0
Vendor' = 0
Product' = 0
Warehouse' = 0
Employee' = 0
PurchaseOrder' = 0
SalesOrder' = 0

All component sets are initialised to be empty at this stage, meaning the system has started and nothing has been entered yet.

As before a possible proof obligation is to show that the initial state of the system may be realised, i.e. it actually exists.

$\exists$ DistributionSystem' • Init_DistributionSystem

The above proof stipulates that it must be possible to arrive at an after system state (DistributionSystem') that satisfies the requirements of Init_DistributionSystem.

The following operation adds a customer (c?) to the system.

AddCustomer

Δ DistributionSystem
c? : CUSTOMER

```
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       c? ∉ Customer
ʔ       Customer' = Customer ∪ {c?}
ʔ       reply! = customer_added
ʔ
│_____________________________
```

Δ DistributionSystem indicates that there will be a change in the state of the system. The customer should not form part of the set of customers already known to the system. The value or reply! indicates that the operation was successful.

If the customer is known to the system the scenario is captured by CustomerRecordExist.

```
¿___CustomerRecordExist_____________
ʔ
ʔ       Ξ DistributionSystem
ʔ       c? : CUSTOMER
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       c?  ∈ Customer
ʔ       reply! = cannot_add_record_exists
│_____________________________
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the customer is already known to the system.

As before, a robust (total) operation for adding a customer to the system may be defined. This is normally done via the schema calculus of Z (Refer to section 3.5) as follows:

RobustAddCustomer ≙ AddCustomer ∨ CustomerRecordExist

At this point an appropriate proof obligation may be to show that operation RobustAddCustomer above indeed adds a new customer to the system in the case that such customer was not in the system before. Naturally any domain knowledge not

already captured by RobustAddCustomer will not surface as a result of discharging this proof obligation. Proofs normally just show possible consequences of operations (Woodcock & Davies [94]).

The following operation deletes a customer from the system.

DelCustomer

Δ DistributionSystem
c? : CUSTOMER
reply! : REPLY

---

c? $\in$ Customer
Customer' = Customer \ {c?}
reply! = customer_deleted

Δ DistributionSystem indicates that there will be a change in the state of the system. The customer should form part of the set of customers already known to the system in which case the customer is removed from this set. The value of reply! indicates that the operation was successful.

If the customer is not known to the system the scenario is captured by CustomerNotFound.

CustomerNotFound

Ξ DistributionSystem
c? : CUSTOMER
reply! : REPLY

---

c? $\notin$ Customer

```
³       reply! = customer_not_found
└──────────────────────────────────
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the customer is not known to the system.

The following operation allows a user to edit an existing customer.

```
┌──EditCustomer ──────────────────
³
³       Δ DistributionSystem
³       c?, new_c? : CUSTOMER
³       reply! : REPLY
├─────────────────
³
³       c? ∈ Customer
³       ∃ c : CUSTOMER • c = c? ∧  c' = new_c?  // Assign new details
³                                               // to existing record.
³       reply! = customer_updated
└──────────────────────────────────
```

A customer (c?) whose details is to be edited is given as input to the system. The new details (new_c?) to be entered are also given as input. The existing customer is retrieved (∃ c : CUSTOMER • c = c?) and his details are replaced as requested (c' = new_c?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a customer is already known to the system or not.

```
┌──IsCustOnSys ──────────────────
³
³       Ξ DistributionSystem
³       c? : CUSTOMER
³       reply! : REPLY
├─────────────────
³
³       (c? ∈ Customer ∧
³       reply! = Yes)
```

```
¿        ∨
¿        (c? ∉ Customer ∧
¿         reply! = No)
└──────────────────────────────
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the customer is known to the system or not.
The following operation adds an employee (e?) to the system.

```
¿___AddEmployee ____________________
¿
¿        Δ DistributionSystem
¿        e? : EMPLOYEE
¿        reply! : REPLY
¿_______________
¿
¿        e? ∉ Employee
¿        Employee' = Employee ∪ {e?}
¿        reply! = employee_added
└──────────────────────────────
```

The content of AddEmployee is structurally similar to that of schema AddCustomer above. The value of reply! indicates that a new employee record has been successfully created.

If the employee is already known to the system the scenario is captured by EmployeeRecordExist.

```
¿___EmployeeRecordExist_____________
¿
¿        Ξ DistributionSystem
¿        e? : EMPLOYEE
¿        reply! : REPLY
¿_______________
¿
¿        e? ∈ Employee
¿        reply! = cannot_add_record_exist
└──────────────────────────────
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the employee is already known to the system. As before a robust operation RobustAddEmployee can be defined, similar to schema RobustAddCustomer above.

The following operation deletes an employee (e?) from the system.

```
┌─── DelEmployee ──────────────────
│
│    Δ DistributionSystem
│    e? : EMPLOYEE
│    reply! : REPLY
├──────────────────
│
│    e? ∈ Employee
│    Employee' = Employee \ {e?}
│    reply! = employee_deleted
└──────────────────────────────────
```

As before, Δ DistributionSystem indicates that there will be a change in the state of the system. The employee should form part of the set of employees already known to the system and is removed from the set of known employees. The value of reply! indicates that the operation was successful.

Schema EmployeeNotFound caters for the scenario where an employee is not known to the system.

```
┌─── EmployeeNotFound ─────────────
│
│    Ξ DistributionSystem
│    e? : EMPLOYEE
│    reply! : REPLY
├──────────────────
│
│    e? ∉ Employee
│     reply! = employee_not_found
└──────────────────────────────────
```

x DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the employee is not known to the system. A robust operation for deleting an employee may be defined in the usual way.

The following operation allows a user to edit an existing employee (e?).

¿___EditEmployee_____
ʔ
ʔ Δ DistributionSystem
ʔ e?, new_e? : EMPLOYEE
ʔ reply! : REPLY
_____
ʔ
ʔ e? ∈ Employee
ʔ ∃ e : EMPLOYEE • e = e? ∧ e' = new_e? *//Assign new details*
ʔ *//to existing record.*
ʔ reply! = employee_updated
_____

The EditCustomer and EditEmployee operations are similar in nature. The difference is that in EditCustomer a customer's details are updated and in EditEmployee an employee's details are updated.

An employee (e?) whose details is to be edited is given as input to the system. The new details (new_e?) to be entered are also given as input. The existing employee is retrieved (∃ e : EMPLOYEE • e = e?) and his details are replaced as requested (e' = new_e?). The value of reply! indicates a successful operation.

The following operation queries the system as to whether employee (e?) is already on the system or not.

¿___IsEmpOnSys_____
ʔ
ʔ Ξ DistributionSystem
ʔ e? : EMPLOYEE
ʔ reply! : REPLY
_____
ʔ
ʔ (e? ∈ Employee ∧

ˀ reply! = Yes)
ˀ ∨
ˀ (e? ∉ Employee ∧
ˀ reply! = No)
└──────────────────────────────

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user whether the employee is known to the system or not.

The remaining schemas may be studied in appendix B. The corresponding operations for the remaining classes, namely, Vendor, Product, Warehouse, Purchase orders and Sales orders are similar as for Customer and Employee above.

After the specification phase the design phase commences and leads to the implementation phase. This is based on the definition of the classes. The net effect is that the program will function as prescribed by Z. If any hidden domain factors or tacit knowledge of some sort has not been discovered during the specification phase, these would be omitted during the coding of the product.

The last proof obligation mentioned above was in connection with schema RobustAddCustomer. In this schema the detail information of the customer is abstracted from the use of the simple, unstructured input variable c?, instead of the more detailed Cartesian product approach mentioned at the beginning of this section. Nevertheless, even if such detail about the customer were given in the schema, no mention is still made of any missing attributes of such a customer.

A similar argument can be made out for the schema RobustAddEmployee which adds an employee to the system or rejects the request if the employee is already on the system. In particular missing **Start** and **End dates** of employment or the **reliability** of a vendor or the **waste** of a warehouse could hardly be elicited through a proof attempt if these attributes were absent to start with.

The next phase after the product has been developed is the testing phase as part of the V&V exercise. Testing is based on the requirements of the client as well as the specification documentation. If any errors are discovered during testing, they can be

verified against these two sets of documentation. Should the implementation stage be reached in the illustrative scenario, it may appear that all requirements as given by the client have been met. However, **three hidden domain errors** have been overlooked during the requirements elicitation phase and subsequent design phase. Some of these have been hinted at above and will be described in chapter 6.

5.7 Summary

For the purposes of this dissertation, a case study was extracted from Jalloul [53]. An illustrative scenario based on a synthesis of the case study used by Jalloul [53] was further more created in order to describe the various requirements elicitation techniques discussed before.

The case study started with a description of what the client's requirements were. This description was given in Natural Language. In the illustrative scenario the

development team then decided to use one or more JAD sessions to reach a better understanding of the requirements of the client.

RAD and UML were tools employed in the illustrative JAD sessions in order to reach a better understanding of the full extent of the system that needed to be developed. Formal Method Z was used for the formal specification of the system. It was noted that the use of formal proofs may also fail to elicit important tacit knowledge and factors in hidden domains.

This case study will be revisited in chapter 6 to discuss, amongst other things, how certain domain knowledge did not surface during the requirements elicitation phase of the development. Three hidden domain factors, which were not elicited, as well as the effect they could have on the system should they come to light, will be described.

Chapter 6 develops two new methodologies to assist the software development team in eliciting tacit knowledge and hidden domain factors from the client and users. After description of the methodologies they will be applied to the illustrative case study of chapter 5 to reveal their value in eliciting possible domain knowledge. The newly elicited domain knowledge will then be incorporated into the system. Changes will be highlighted on the RAD screens, the UML object model and Z schemas relevant to the system. The rest of the system not affected will not be revisited.

Chapter 6

Two New Software Requirements Elicitation Methodologies

In chapter 5 a case study extracted from Jalloul [53] was presented and extended to illustrate the functioning of different requirement elicitation techniques. Chapter 6 proposes two new requirements elicitation techniques with the potential to extract

tacit knowledge and hidden domain factors. The case study is then revisited to illustrate the functioning of the two proposed methodologies and the benefits they could offer in revealing the hidden domain factors not addressed in the illustrative scenario depicted in the case study using existing elicitation methodologies. The impact these two could have on the system is then illustrated and discussed.

6.1 Current requirements elicitation techniques used in industry

Results from a questionnaire (Appendix A) sent out several businesses in the retail sector using software programs in the daily operations of their businesses, indicate that the communication gap identified, by BMC Software [15], is indeed a reality. In their responses to the questionnaire, information technology managers expressed the opinion that, even in cases where development has been outsourced, a continuous need for improvement of their systems is prevalent. In some of the cases responding to the questionnaire, users were involved from the start of the development of a system, while in others, they were used only to ensure the thorough testing of the functionality of the system.

The conclusion reached after studying the responses to the questionnaire, is that, in most cases, even though any of the current techniques described in chapter 3 were used, companies still experience the problem of corrective maintenance to the required system. Continuous improvement is necessary because the environment changes and the system needs to adapt, otherwise it will become obsolete. It is also evident that fewer changes to the system would be necessary, if certain forms of tacit knowledge and hidden domain factors were incorporated from the beginning of the development of the systems.

In the sub-section below the first proposed requirement elicitation methodology is described.

6.2 SRE (Solitary Requirements Elicitation) methodology

Several different techniques and methods – some more successful than others – to aid the software development process, have been both researched and applied. Brooks [22] maintains that there is no silver bullet for software development, meaning that there is no one technique to solve all problems, thus vastly improving software development processes. Cox [30] described that a silver bullet could potentially be realised by following the Object Orientation software development technique, but later Cox [31] indicated that this was not the case.

The SRE methodology is the first of the two new methodologies proposed in this dissertation which could aid the software development process. Its strength lies in the fact that it could elicit knowledge taken for granted. Grice (1975) in Blandford and Rugg [14, p78] explains the concept of knowledge taken for granted as follows: *"… taken for granted knowledge involves information which is so familiar to the respondents that they do not bother to mention it explicitly, since they (often incorrectly) assume that everyone else also knows it."*

This method has the potential to gain access to hidden domain factors which could seriously affect the specification, design and implementation of the software project. If a hidden domain factor is a critical factor and not merely one for which the software development team can easily provide by adding on an additional module, its inclusion has the potential to increase the success of the software project.

The SRE method which forms part of a larger process defined as Algorithm 6.1 below, may be viewed as an extension of, for example, a JAD workshop. In essence the SRE comprises **Step 1** in the Algorithm which itself may be implemented in three steps:

Step 1.

Each member in the team breaks away from the group for a period of solitude (a retreat). During such a retreat, each team member creates a list of all assumptions about the system of which he can think and writes these down. Emphasis should be placed on tacit or hidden assumptions. This step is at the heart of the SRE methodology proposed in this dissertation.

Step 2.

After the lists have been drawn up individually, the team comes together again and shares them. They work through each list until all lists have been exhausted. The contents of the lists are combined to form an improved understanding of implicit assumptions about the system.

Step 3.

This affords each member of the team the opportunity to realise that assumptions made could be entirely wrong or could even have been overlooked if it was not brought to the fore through a moment of silent reflection on the requirements of the system.

The above three steps above may be formalised as discussed next. Assuming a JAD-like session consisting of ***m*** developers and ***n*** clients and users, the above three steps constituting an SRE exercise are formalised by Algorithm 6.1:

Algorithm 6.1: Gather comprehensive tacit domain knowledge from all stakeholders.
Begin

(* Gather assumptions and tacit knowledge for each individual developer, client and user. *)

for *i* := 1 **to** ***m*** **step** 1 **do**

D_i := Unspoken assumptions about the system of developer *i*;

for *j* := 1 **to** ***n*** **step** 1 **do**

U_j := Tacit knowledge of client *j* or user *j*;

(* For the sake of simplicity we do not distinguish above between a client and users. *)

(* Amalgamate domain knowledge of developers and users into two separate sets D and U. $\cup$ denotes an arbitrary union – refer to section 2.3 *)

$$D := \bigcup_{i:=1}^{m} D_i ; \quad U := \bigcup_{j:=1}^{n} U_j$$

(* Combine outcome of SRE requirements or assumptions into set C. *)

C := D ∪ U

End.

The last step in the SRE above is to combine (through set-theoretic union) all relevant requirements, assumptions, etc. into a single set to be considered by the **team**, consisting of the client, users and members of the software development team. Another way to achieve the aims of the SRE is to try and create as much common knowledge as possible between the client and users on the one hand and the software development team on the other. This may be achieved by increasing the size of the **intersection** between sets D and U above. Therefore, the last step in Algorithm 6.1 could instead be written as C := D ∩ U and the aim would then be to maximise the size of set C, i.e. **maximise**(#C) where # represents the cardinality of a set. Translating these symbols back into practice, implies an attempt to maximise the cardinality of the intersection of the combined sets for software developers (D) and clients and users (U).

The writer of this dissertation hopes that the SRE exercise above would become part of a unified model of requirements elicitation as called for by Hickey and Davis in their paper "*Requirements Elicitation and Elicitation Techniques Selection: A Model for Two Knowledge Intensive Software Development Processes"*, delivered at the 36th

Hawaii International Conference on System Sciences, 2002: "*Although many papers have been written that define elicitation, or prescribe a specific technique to perform during elicitation, nobody has yet defined a unified model of the elicitation process that emphasizes the role of knowledge*."

The procedure described in Algorithm 6.1 for two developers and one client and one user is depicted in Venn-diagram notation in figure 6.1a. The hidden, tacit domains belonging to the client and user are indicated by areas **A1** and **A2** respectively.

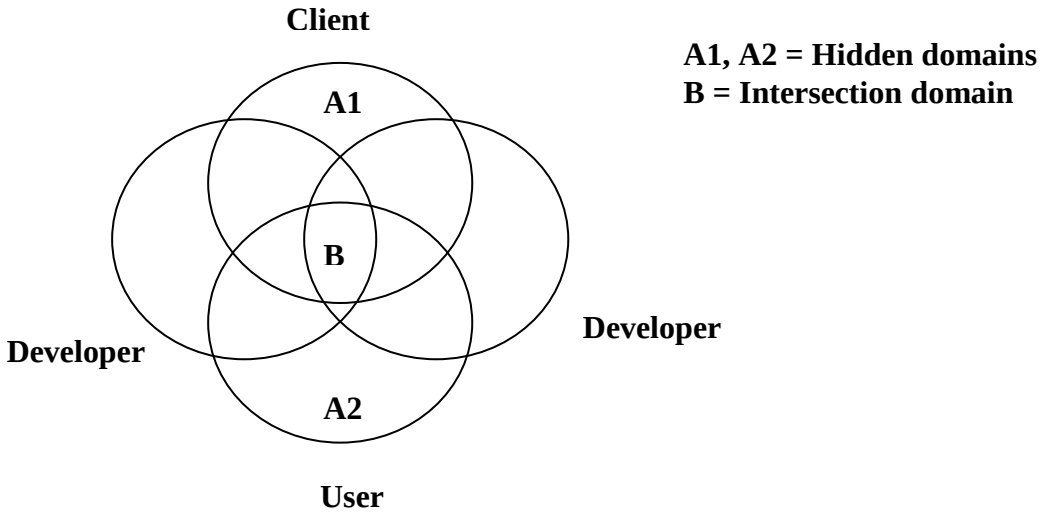


Figure 6.1a Knowledge domains of developers, client and user before SRE.

When each member of the team compiles his list, some of the hidden factors might surface when the team comes together again and works through each list until all lists have been exhausted. The SRE in Algorithm 6.1 specifies that the lists for developers on the one hand and the clients and users on the other hand are to be amalgamated separately. The idea behind this process is to try and keep similar things and issues together in two different sets before combining them into a single, larger set, or as discussed above, to attempt to maximise the intersection of the two sets.

The two resulting sets (D and U in Algorithm 6.1) are shown in figure 6.1b. As indicated in figure 6.1b, these two sets may already overlap to some extent (area **B**). This is because some of the assumptions made could have surfaced, allowing developers to understand the domain of the client and users better, thereby gaining more insight into their environment. There could, however, still be unspoken domain

knowledge hidden from the development team. Such knowledge is indicated by area **A** in figure 6.1b

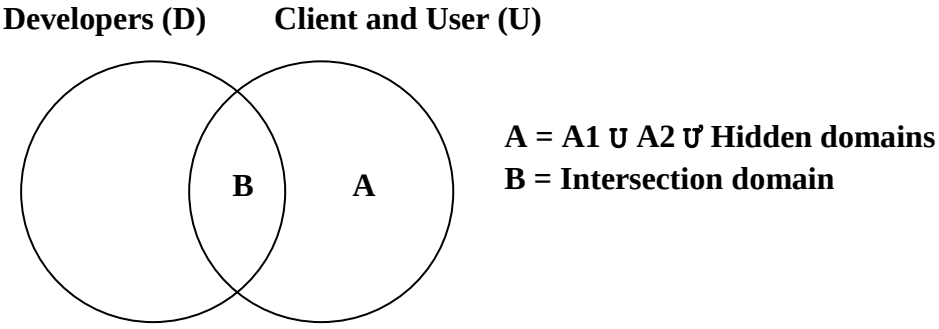


Figure 6.1b Domains integrating during SRE exercise. (Algorithm 6.1)

Naturally, domains may increasingly overlap during the SRE (Algorithm 6.1) as shown in figure 6.1c.

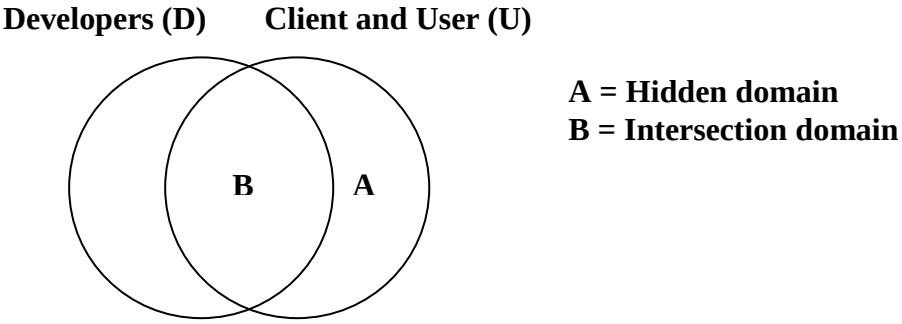
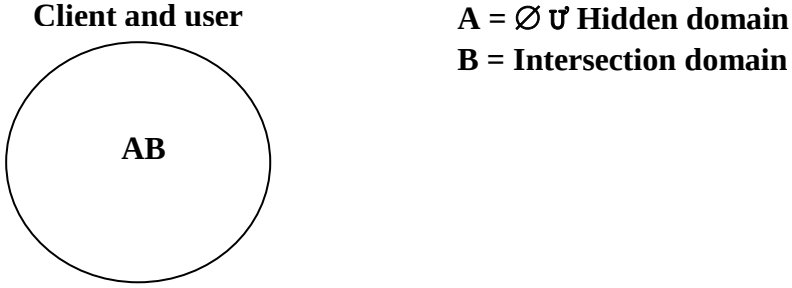


Figure 6.1c Increased domain intersection during SRE exercise. (Algorithm 6.1)

If the SRE methodology is applied comprehensively and the lists are done very well, the hidden domain of the client and users could shrink (area **A** in figure 6.1c), or in an ideal scenario, become empty (figure 6.1d). This would mean that all tacit knowledge or hidden factors have been elicited and that developers have identified the requirements of the client fully, thus understanding all of the requirements and not only those that have been elicited before SRE was followed.



Developers

Figure 6.1d Total intersection of domains of developers, client and users through successful SRE exercise.

Figure 6.1a to figure 6.1d illustrate the potential of the SRE methodology to elicit tacit (hidden) domain knowledge, sometimes overlooked by using existing methodologies.

Example 6.1

When a client approaches a bank for a loan, the bank investigates the risk and determines an interest rate to be charged over the term of the loan. The bank uses the prime lending rate, which is higher than the repurchase rate (repo rate) charged by the reserve bank in SA. A system is needed to calculate an interest rate based on information gained by applying certain criteria to the client.

After the completion of a JAD session, each team member is given the task to write a list of assumptions to be used in a follow-up JAD session. One member lists the assumption that everyone knows what the prime rate is. This is, however, not the case. Some members think it is a fixed rate (**assumption #1**) and that banks use the rate mainly for risk determination purposes. These members of the team assume that, should the risk be found to be acceptable, the bank would give another final interest rate.

Another assumption is that the prime rate is determined by the bank and may be changed at any time (**assumption #2**). The members of the team making this assumption do not know that external factors, such as the way in which the reserve bank influences the prime rate of the bank and whether it remains the same or not, need to be considered. (**Tacit knowledge in hidden domain)**.

At this stage, the assumptions on the list of one of the employees of the bank who serve on the software development team, are discussed and explained. The team now understands that the prime rate is the rate at which a bank may issue loans, but that it

may, depending on criteria applied to a client, decide to quote an interest rate of **prime + x or prime – x**. Should the risk be high, the bank would opt for the first. In the case of a very reliable client posing a low risk, however, the bank would quote a rate below the prime rate. Another clarification which comes to the fore is that the prime rate may fluctuate. This implies that the software product must have the capability to change the prime rate when required.

Figure 6.1e illustrates the different domains described above. The hidden domain **A** is the tacit knowledge which is not known to the developers, resulting in the reasons for their assumptions **C** and **D**. The aim with the SRE is to mine the knowledge in **A** via assumption lists.

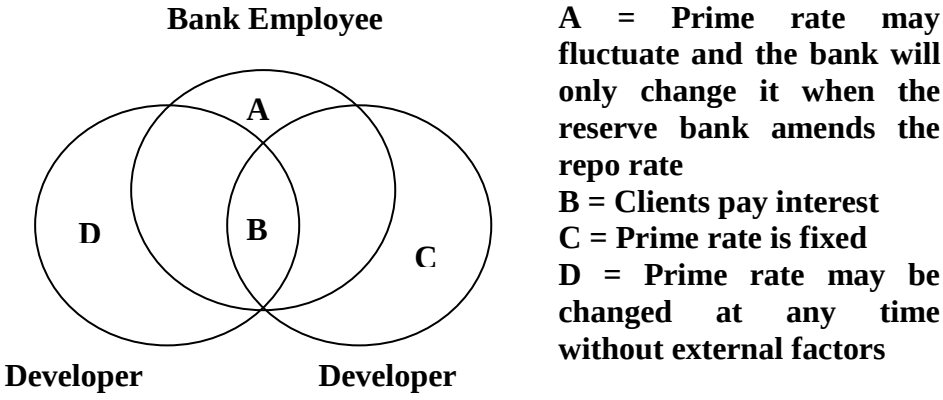


Figure 6.1e Intersection of domains of developers and bank employee before SRE exercise.

Figure 6.1f illustrates that all hidden domain factors have been elicited and that there is complete understanding of the business domain and what is required from the software product.

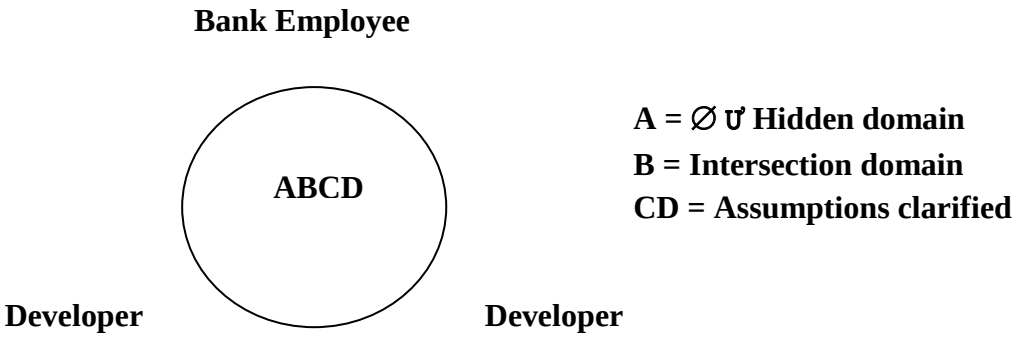


Figure 6.1f Ideal scenario of domains of developer and bank employee after comprehensive SRE exercise. (A=B=C=D)

The SRE methodology when employed in a JAD session, has the potential to elicit tacit knowledge and hidden domain factors, which would otherwise have remained hidden had individual members not been given the opportunity to break away and consider assumptions on their own.

The next section focuses on the second proposed methodology, namely the DDI (Developer Domain Interaction). This methodology could take more time to implement, but the advantages may be realised when the software product reaches its maintenance cycle.

6.3 DDI (Developer Domain Interaction) methodology

The philosophy underlying to the DDI methodology is that it is necessary for one or more members of the software development team to gain insight into the business of the client, as well as into its daily operations, if they wish to improve their chances of eliciting tacit knowledge and factors in the hidden domains of the client and users. It stands to reason that it is important for developers to be exposed to this environment. This will allow them to compare what the specification describes to how the specified product is expected to function in the live working environment of the client. This approach has the benefit of enabling understanding of the product and also aiding developers when they need to design user interfaces. It is anticipated that developers will be able to understand users' daily operations much more effectively and to identify product automation problems. The domain of the user is, in fact, revealed by applying the DDI methodology.

The following steps need to be taken by the **client** when following the DDI methodology in conjunction with a developer who is to be placed in his environment:

Step 1.

Determine whether the developer is a suitable candidate for the work at hand by:

- Introducing the developer to all members of staff with whom he will be working and interviewing each staff member who will work with the developer and getting feedback.
- Continuing with the next step described below if the feedback is satisfactory.

Step 2.

Assign a specialist to the developer by:

- Ensuring that a domain specialist is assigned to the developer for each section upon which the new system will have an impact, e.g. a Human Resources (HR) specialist should the program have any impact on HR functions.
- Continuing with the next step below when all domains which will be impacted upon by the new system have been identified and specialists have been assigned to the developer.

Step 3.

Identify all main issues and tasks to be dealt with together with domain specialists as follows:

- Identify main components of the domain.
- Determine tasks to cover the components identified above.
- Determine procedures for:
    - Domain specialist to train developer.
    - Tasks to be performed by developer.
    - Testing or evaluating the developer to determine whether domain has been understood and tacit and hidden knowledge have been transferred to his domain.

- Identifying corrective actions to be taken until domain knowledge has been transferred to the developer. This could call for the replacement of the developer if tasks are executed incorrectly and he continues to fail in the execution of the said tasks.

The following steps need to be taken by the **developer** when implementing the DDI method, i.e. before he enters the environment of the client as well as during his training period in the client's environment:

Step 1.

Establish whether he is comfortable working with the client and in the specified domain. Continue with step 2 below if the above has been established positively.

Step 2.

Observe, ask questions (enquire) and do tasks normally performed by users.

Step 3.

Continue with step 2 above until the knowledge in the hidden domain of the client has been transferred to his own domain.

From the above it is clear that developers need to **ask questions, study documents, observe** what users are doing and **understand** the internal operations of the business. They also need to **ask how exceptions are handled** and **what type of exceptions to the norm** occurred before. If possible, it would help the developer to **actually do some of the work** and **perform some of the tasks of the users** who will be using the new system. This would enable developers to acquire tacit and hidden domain knowledge. Stapleton et al. [86, p164] refers to Fleck's (1997) statement: *"Tacit knowledge is inherent in people's practice and know-how. It can only be articulated and transmitted through adept execution and for example through a learning cycle that involves demonstrating and emulating the 'teacher'."*

The DDI methodology of placing a developer in the environment of a client may be an exercise in ethnography: Wilcox [91] describes ethnography as a: "*…descriptive endeavour in which the researcher attempts accurately to describe and interpret the nature of social discourse among a group of people*". He also states: "*One must be in a position both to observe behaviour in its natural setting and to elicit from people observed the structures of meaning which inform and texture behaviour.*"

The above descriptions indicate very well what developers embarking on a DDI exercise should do. In fact, they should do what an ethnographer does. Developers also need to **observe** and **elicit hidden factors and tacit knowledge** which could offer the explanations for certain tasks being performed. Through this learning experience, they would gain insight into the internal workings of the client's organisation. This – employing the DDI methodology – has become necessary for developers, as current software development methodologies have, as seen in what goes before, not always been successful in the elicitation of requirements. The reader is again referred to figures 6.1a to 6.1d. As with the SRE, developers may gain more insight into the hidden domain of the client through the DDI methodology.

Naturally, the DDI methodology would vary depending on the project and the complexity involved and could result in a number of steps to be repeated continuously until the hidden domain knowledge is understood by developers.

Developers need to ask questions and need to observe users while users are performing their usual daily tasks (Friedrich and van der Poll [40]). Developers also need to perform certain tasks themselves if, after several iterations, tasks are still not understood.

Allowing a developer to perform certain tasks himself and thereafter testing or evaluating the developer leads to a more detailed description of a part of the DDI methodology as captured in Algorithm 6.2 below.

Algorithm 6.2: Modus operandi of developers with reference to evaluation of tasks.
Begin

{

If developers understand the task and understand why it needs to be performed **then**

{

Developers continue with the next task, operations, questions and observations.

}

else

{

loop (the task needs to be repeated until understood.)

{

Developers need to repeat tasks and operations until they are understood, at the same time continuing with questions and observations until understood.

Test: **If** tasks and operations have been understood by developers **then**

Exit out of loop.

}

}

}

End.

The above Algorithm should continue until all requirements for the system have been understood.

The DDI methodology described above has a close link with the SRE Algorithm 6.1 above. During the DDI exercise, the different domains of the users and the developer(s) need to overlap increasingly in the sense that developers need to ensure that **they** can actually perform the tasks which **users** are expected to do. This would help towards a developer understanding the environment in which the system needs to operate.

In general more than one member of the software development team may simultaneously be placed in the environment of the client. In this case, assuming ***m*** developers and ***n*** users, the above discussion constituting a DDI exercise may be formalised by Algorithm 6.3:

Algorithm 6.3: Integrate developers into the domain of the client and the users.

Begin

(* Gather two sets of tacit knowledge – developers and users *)

for *i* := 1 **to** *m* **step** 1 **do**

D_i := Establish tasks for developer *i* that may elicit tacit knowledge in hidden domains; Document the underlying **tacit knowledge**

for *j* := 1 **to** *n* **step** 1 **do**

U_j := Establish **tacit knowledge** of user *j*;

(* Amalgamate domain knowledge of developers and users *)

$$D := \bigcup_{i:=1}^{m} D_i ; \quad U := \bigcup_{j:=1}^{n} U_j$$

(* Gather common knowledge from DDI exercise into set C. *)

$C := D \cap U$

End.

As before, the intention is to maximise the cardinality of the intersection of the sets **Developer** (set **D**) and **Client and Users** (set **U**). Figures 6.1a to 6.1d illustrate that the domain of the developer increases with the body of knowledge of the user. The size of the domain containing tacit knowledge and hidden factors, therefore, decreases. Note that the users also gain greater appreciation of the hidden domain of the developers.

6.3.1 Differences between DDI and Agile methodologies

This section aims to clarify some differences between the DDI and Agile introduced in section 4.2.4.1 before.

The difference between the DDI and the Agile methodologies is that, though developers and users work together in the Agile method (Smialek [82]), they do not necessarily do so in the live working environment. In the DDI method, on the other hand, developers observe, question and carry out tasks in the live working environment while trying to elicit tacit domain knowledge.

The potential problem in employing the Agile method is that, while experts may give full answers to questions asked during interactive sessions with developers, they may not even think of areas not covered in the questions. They may also not even consider highlighting critical areas. Failure to elicit the hidden critical requirements for a product could result in project failure or a product that needs to be reworked until it can function correctly in the live working environment. This could result in escalating costs because of higher maintenance costs. Alvarez [4, p2] states that: *"…many system failures can be attributed to a lack of clear and specific information requirements. Furthermore, errors during requirements analysis that are not found until later stages of the implementation process can cost significantly more to fix."*

Agile focuses on the specific requirements that need to be met, but may fail in doing so because it is not able to mine all tacit knowledge and hidden domain factors.

DDI also focuses on the specific requirements to be met, but is in a very strong position to mine all tacit knowledge and hidden domain factors. The reason for this is that the DDI exercise takes place in the live working environment where the system is required.

Example 6.2

The reader is referred to example 6.1. The application of the DDI will be demonstrated by referring to the same scenario as in example 6.1 in which a software

product with the capability of changing the prime rate when required needs to be developed.

Developers are introduced to the appropriate banking environment and **observe** bank employees carrying out their tasks. **Questions** dealing with the processing of loan applications and criteria needed to process the application, are asked. Bank employees explain that applications are received from branches, as well as from clients who come in personally to apply. Information needed for the application include the client's period of residence at his current address and his monthly income. His current income to debt ratio, as well as his ability to afford loan repayments are also taken into consideration.

Up to this stage, the developers have not understood the full loan application process, as they do not understand what the prime rate is and how it influences the risk decision process. In exercising the DDI method, they are required to continue with observations, questions and even the actual processing of tasks. This will lead to their understanding the issues regarding the prime rate referred to above and eventually a full understanding of what is involved in the processing of a loan application. The loan process below is instantiated from the general description in Algorithm 6.2 above.

Begin
{
If developers observe the loan processing process and understand the complete process **then**
{
Developers observe the next process.
}
else
{
loop
{
Developers continue to execute the same tasks, ask questions and observe, until the full loan application process is understood.

Test: **If** tasks and operations have been understood by
developers **then**
Exit out of loop.

}

}

}

End.

Looping occurs, as long the developers do not understand the complete loan application process, because they are not sure what the prime rate is and how it is used. Once this is understood, the DDI allows developers to continue to the next process. This continues until the full process is understood.

Developers are now in a position to develop the required software product with greater confidence and may also be able to deal with exceptions which may occur in the actual business environment on a daily basis better.

The reader is referred to figures 6.1e and 6.1f, illustrating a similar process in which increasing interaction of knowledge domains take place until a 100% integration point is reached in an ideal situation.

6.4 Comparison of characteristics of SRE and DDI methodologies

Table 6.1 illustrates the characteristics of the SRE and DDI methodologies.

Methodologies	SRE	DDI
Assumption lists	√	

Domain interaction		√
Developer participation	√	√
Client/User participation	√	√
Task iteration		√
Observe & Questions		√
Tacit- and hidden domain knowledge elicitation	High	Very High

Table 6.1 Characteristics of SRE and DDI methodologies.

Table 6.1 shows that the SRE methodology concentrates more on the creation of assumption lists during the requirements elicitation phase, while the DDI methodology involves the direct interaction of developers in the actual working environment of the client and the users.

Task iteration is necessary in the DDI method, as it allows developers to learn from experience by performing the tasks continuously until understood. The more developers interact with the live working environment, the better their understanding of the client/user domain.

It is possible that the DDI method may incur higher requirements elicitation costs in the overall project because one or more developers are placed in the environment of the client and users. This could, however, be counter balanced by the benefits which could be realised as the project progresses, as costs during the implementation and maintenance phases could be reduced as an end result.

6.5 Case study revisited

The case study described in chapter 5 is revisited to illustrate the functioning of the SRE and DDI methodologies. Each methodology will be described as if it had been applied to the case study and the result will then be incorporated into the case study. Only the sections from chapter 5 that would have been affected by the use of these two requirements elicitation methodologies will be revisited.

The software development team in the illustrative scenario created in the previous chapter starts with an SRE exercise as part of a JAD session. During the solitary retreat as part of the SRE, a Human Resources specialist, who is a member of the team, realises), and subsequently indicates on his list, that the information kept for an employee (section 5.2.5) must have a **Start date** and **End date**. This is necessary for the calculation of employment benefits at the resignation or retirement of an employee. This was not highlighted in the JAD sessions.

Another oversight is identified when it is discovered that a user has listed **Waste** (section 5.2.4) under the information to be kept for a warehouse.

The software development team then decides to apply the DDI methodology and to interact with users in their working environment. After a week of being exposed to the live working environment, another oversight is discovered. The team realises that information on the **Reliability** of a vendor (section 5.2.2) should also be kept as this is important to users when placing an urgent order. This too, was not highlighted during JAD sessions.

The following oversights are identified after application of the SRE and DDI methodologies:

- Employees (section 5.2.5) should always have a **Start date** and **End date** (e.g. resignation, retirement or death). This information could have a significant impact on the calculation of the benefits of an employee. These dates are important when, e.g. calculating the amount that needs to be paid out to either the employee or his beneficiaries at resignation or death.
- Prior to the application of these two methodologies, it was very difficult (if not impossible) to determine how **Waste** is handled by the system. The information kept on the warehouse (section 5.2.4) pertains only to contact information and how it is handled. Information on waste is important. Should a warehouse produce a huge amount of waste, it is imperative for the business to decide to reduce the amount of waste. This will directly affect the profit margin of the company. Another product which could be distributed much

faster could be shelved in the space of the waste product. This would result in a higher turnover and less waste.

- The system does not display information on how **Reliable** a vendor is. (Refer to section 5.2.2). This could have serious consequences when products are ordered. Should there be more than one vendor for the same product, it would be safer to place urgent orders with a more reliable vendor. Placing orders with a less reliable vendor might even have legal implications, should stock not be delivered on time. This should clearly be avoided.

The above mentioned problems are only some of the domain specific factors not previously discovered by the software development team and the list is certainly not comprehensive as there could be many more.

Should either of the two or both of these methodologies have been used successfully, the impact on the system would have been the following:

- User interfaces of the screens would have looked different and would have functioned differently.
- The functionality of the system would have been affected, as the hidden factors previously missed would have been incorporated.
- The UML design of the system would have been affected, as the class diagram would have incorporated the extra functionality and attributes not included before. This will be described in sections 6.5.2 and 6.5.3 below.
- System testing would have involved extra functionality. Provision would have been made for more test cases. The extra test cases would have been necessary to test the added attributes to ensure the functioning of the system according to the specification.
- It would have had an impact on the implementation of the system. The coding would have had to reflect the extra domain requirements not elicited by using the existing requirement elicitation methods.
- The documentation of the system would have been affected, since it would have had to indicate the extra requirements.

- All of the hidden factors would have had an impact on any SDLC model – be it the Waterfall, Build and Fix or Incremental model, etc. – used.

6.5.1 Amended user interface screens

This section provides examples of how the user interface screens would have to be changed after application of the SRE and DDI methods.

The example below depicts the changes which would have to be made to the screen displaying employee information (Figure 5.3e) after the inclusion of the **Start date** and **End date.**

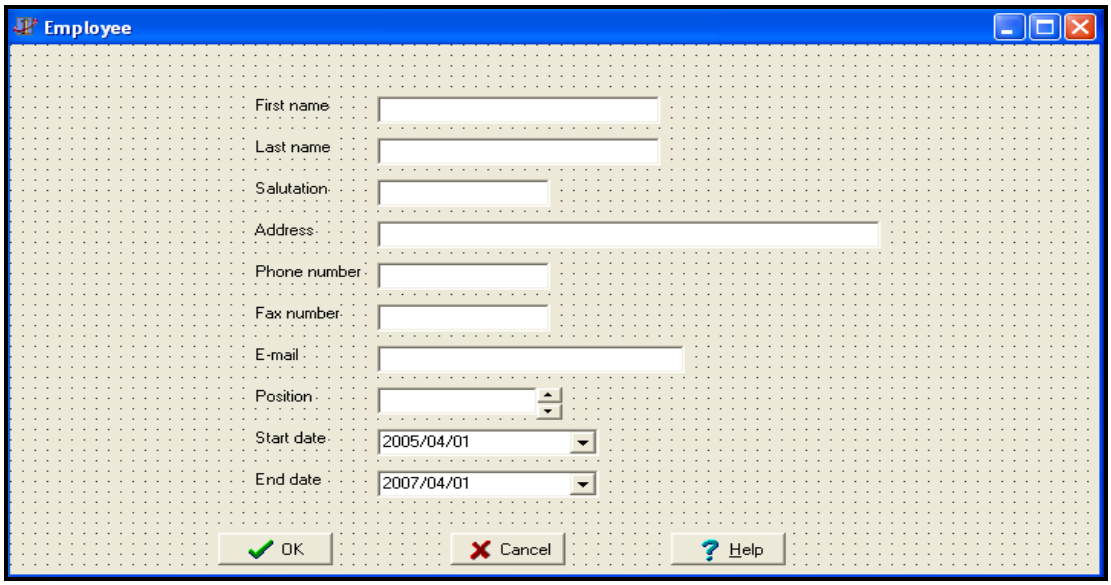


Figure 6.2a: Screen displaying employee information with start and end dates.

The start date would have to be added to indicate when an employee started with the company. The end date would have to be added to indicate when a particular employee stopped working for the company for whatever reason. These dates could then be used to calculate all benefits owed to the employee.

The example below depicts the changes which would have to be made to the contact screen for warehouse information (Figure 5.3d) with the **Waste** attribute added.

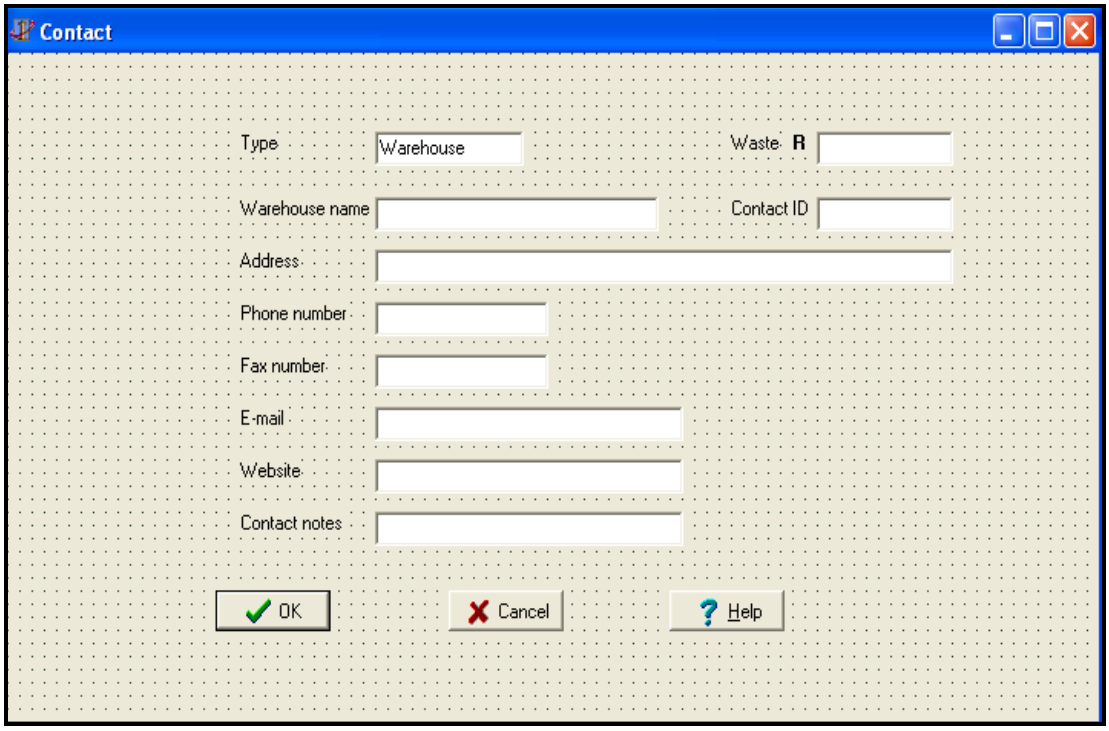


Figure 6.2b: Screen displaying contact warehouse information with waste field added.

The waste field would have to be added to indicate how much waste is produced at that specific warehouse. This would assist the client in determining the reasons for of large amounts of waste. It could be caused by their keeping stock which is not selling quickly enough.

The example below depicts the changes which would have to be made to the contact screen for vendor information (Figure 5.3b) with the **Reliability** indicator added.

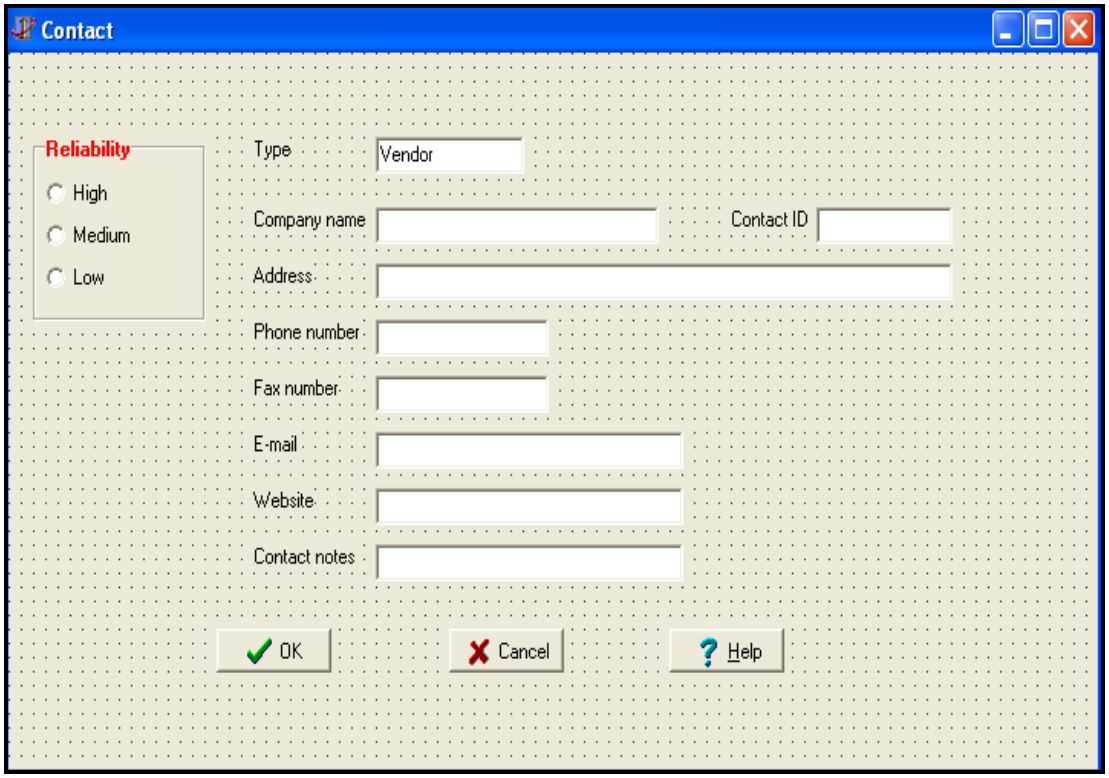


Figure 6.2c: Screen displaying vendor information with reliability indicator added.

Provision would have to be made to indicate how reliable a vendor is. This would help the client when urgent orders need to be placed. Should an order be extremely urgent, the client would select a vendor with high reliability, because there could be legal implications should the order not be delivered on time.

The degree of reliability of each vendor is determined by the experiences that the client has had in this regard in his relationship with the vendors over time.

The client may possibly rate a vendor as follows:

- **High** indicates that orders are always processed and deliveries are 80% on time.
- **Medium** indicates that orders are always processed and deliveries are 60% on time.
- **Low** indicates that orders are always processed but deliveries are only 40% on time.

6.5.2 Amended Use Case diagrams as used as part of UML

The following changes to the UML Use Case models discussed in chapter 5 would have to be incorporated. Use Case models for the Vendor, Warehouse and Employee could remain the same as in chapter 5. The attributes indicated below would have to be added, however.

In the **Vendor Contact Information** Use Case model (section 5.3.1.2), **Reliability** would have to be to be added.

Processing **Vendor Contact Information** would require the following: Company name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes and **Reliability**.

In the **Warehouse Contact Information** Use Case model (section 5.3.1.4), **Waste** would have to be added.

Processing **Warehouse Contact Information** would require the following: Warehouse name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes and **Waste**.

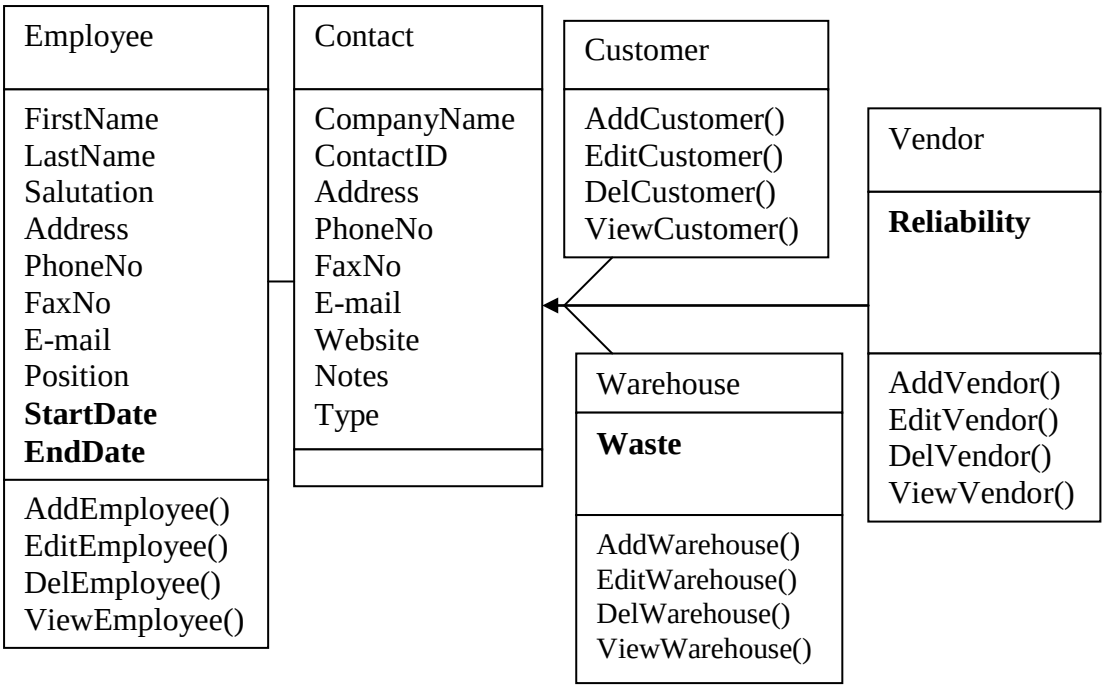
In the **Employee Information** Use Case model, (section 5.3.1.5) a **Start date** and **End date** would have to be added.

Processing an **Employee's information** would require the following: First name, Last name, Salutation (Title), Address, Phone number, Fax number, E-mail and Position (Job title), **Start date** and **End date.**

6.5.3 Amended UML object model

The additions to the Use Case models discussed in the previous section would necessitate the UML Object model described in chapter 5 to be amended as indicated below:

UML Object Model:



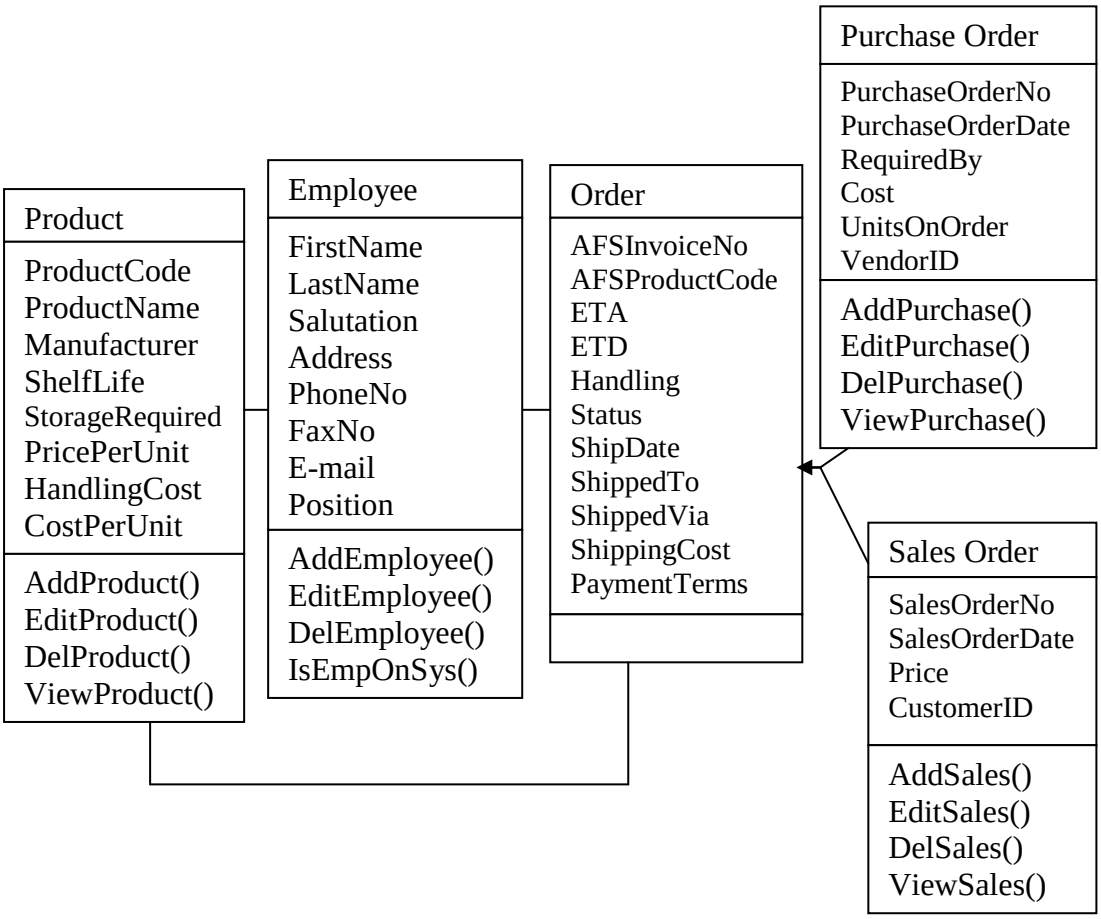


Figure 6.3 Amended UML Object model of distribution system.

6.5.4 Amended Formal Method Z

Formal Method Z, as discussed in chapter 5 (section 5.6), would also not have revealed the oversights and hidden domain factors discovered only after the application of the SRE and DDI methodologies. The reason for this is that operations are developed on predetermined sets such as the EMPLOYEE set made up of fixed attributes.

The discovery of the hidden domain factors would call for the changes to Z indicated below.

In the Z specification given before, only the type definitions and not the operations would have to be amended. The changes are reflected below.

[VENDOR] - The set of all vendors (Company name, Contact id, Address, Phone number, Fax number, E-mail, Website, Contact notes, YTD and Lifetime, **Reliability**)

[WAREHOUSE] - The set of all warehouses (Warehouse name, Contact ID, Address, Phone number, Fax number, E-mail, Website, Contact notes, Inventory, **Waste**)

[EMPLOYEE] - The set of all employees (First name, Last name, Salutation (Title), Address, Phone number, Fax number, E-mail, Position (Job title), **Start date**, **End date**)

6.6 Summary

Chapter 6 introduced two new methodologies namely the SRE (Solitary Requirements Elicitation) and the DDI (Developer Domain Interaction). It also contained algorithms for each method indicating that the larger the intersection domain between the client, users and the software development team, the better. This was shown to be true with the application of the SRE and DDI methods, as more hidden domain factors could be highlighted and taken into consideration while developing the system.

Chapter 5 dealt with a case study extracted from Jalloul [53] and an illustrative scenario created to describe requirements elicitation techniques. The two new elicitation techniques discussed in chapter 6 revealed hidden domain factors not taken into consideration in the scenario depicted in chapter 5. The way in which these factors could have affected the system was therefore also not discussed in chapter 5.

The effect of the hidden domain factors could have had on the system was discussed in chapter 6. The amendments to sections of the distribution system were described and illustrations given.

It should be pointed out that in some domains there could be less critical domain factors than in other domains. The important issue is that if there are critical domain factors they should be highlighted. The problem if a critical domain factor is not taken into consideration is that a complete software development project could fail. If the critical domain factors are not included the system may solve a problem, but it may not be the right problem or may solve only part of the problem.

The hidden domain factors discovered in chapter 6 are only minor compared to factors which could affect the complete core of a system and which, when overlooked, could cause a system to function incorrectly. These could lead to a decision to either amend the system or rewrite it.

Chapter 7 will revisit the research question and hypothesis, contain a summary, revisit the SRE and DDI methodologies include a conclusion as well as a reference to future work that could be undertaken.

Chapter 7

Summary, Conclusions and Future Work

In this, the final chapter of the dissertation, the main focus is to provide a brief summary of what has been done in the previous chapters and to consider the extent to which the research question posed in chapter 1 has been answered through the hypothesis posed thereafter. Some conclusions that may be drawn from the results presented in this work are presented and some indications for future work in this area are considered.

The chapter is structured as follows:

Section 7.2 revisits the research question as well as the hypothesis, posed in section 1.5 and section 1.6 respectively in chapter 1. Section 7.3 provides a brief summary of the dissertation in view of the main findings of earlier chapters. The SRE and the DDI methodologies, which represent the main contributions of this dissertation are considered together with their advantages and disadvantages in section 7.4. Some conclusions that may be drawn from the dissertation are presented in section 7.5, while considerations for future work in this area are given in section 7.6.

7.1 Research question and hypothesis

The research question (section 1.5, chapter 1) posed in this dissertation was to determine which **mechanisms** may be employed to facilitate the **mining of much needed tacit knowledge** to aid developers in capturing the requirements of a client. The question was approached through a study of existing requirements elicitation techniques as well as a number of Software Development Life Cycle (SDLC) models. These elicitation techniques and SDLCs were discussed in chapters 3 and 4 respectively and applied to an illustrative case study in chapter 5. The aim was to reveal that these techniques do not always elicit the necessary tacit knowledge and factors from the hidden domains of clients and users. This was shown to be the case.

In the light of the above findings it was possible to lay the basis for the hypothesis (section 1.6) proposed in this dissertation. This hypothesis proposed two new requirements elicitation techniques in the form of an SRE and a DDI methodology which may help in the elicitation of valuable tacit knowledge and hidden domain factors.

The SRE and DDI methodologies were both developed in chapter 6 and were then applied to the illustrative case study of chapter 5. It was then shown that employing the new methods provides the potential to extract tacit knowledge and hidden domain factors from clients and users.

In the following section we provide a summary of the parts of the dissertation not covered in the above discussion of the SRE and the DDI methods.

7.2 Summary

The software development team normally starts a software development project with a requirements elicitation process. This can be seen in each of the SDLC models described in chapter 4 of this dissertation.

The models described are:

- The Waterfall model.
- The Rapid Prototyping model.
- The Build and Fix model.
- The Incremental model.
- The Spiral model.
- The V-Process model.

It was indicated that a project should follow the following phases in the order given in section 2.7 and repeated below:

- The Requirements and Analysis phase.

- The Specification phase.
- The Design phase.
- The Implementation phase.
- The Testing phase as being part of V&V.
- The Maintenance phase.

Should a project skip a phase such as design, for example, and move to the next phase (implementation) instead, the risk exists that certain factors are not taken into consideration. This could lead to a software project which does not achieve the desired outcome. The first phase (requirements and analysis) is especially important. It is during this phase that the software development team elicits the requirements of the client and determines what the product should functionally be able to do in order to fulfil a certain mandate. Factors not taken into consideration during this vital important first phase could result in a faulty system or a system requiring much maintenance to bring it in line with requirements.

Venn diagrams were introduced (section 2.3) and examples drawn from them to illustrate how domains are created. This visual tool was used to describe different domains and the level of integration between clients, users and developers needed to increase the success of a software development project.

The communication gap between a client and the software development team was highlighted (section 2.6). Attention was also given to the functioning of software development teams, their intricacies and the communication channels between their members. Roles of team members focussed on include:

- The Project Manager.
- The Architect.
- The Business Analyst.
- The Systems Analyst.
- The Programmer/Developer.

A case study synthesised from Jalloul [53] and further expanded to provide for an illustrative scenario was introduced in chapter 5. It described the way in which an SDLC model (Incremental) and some of the requirement elicitation methods (JAD, RAD, UML and the Formal Method Z) could have been used and how they would have functioned.

It was shown in section 5.6.1 that methods such as RAD, JAD, UML, Natural Language, Prototyping and Formal Method Z may be used in the SDLC models mentioned above, but that these methods do not always highlight important tacit knowledge or hidden domain factors.

Chapter 6 followed with an introduction of two new requirements elicitation methodologies, namely the SRE and DDI mentioned above. Each method was developed and then applied to the illustrative scenario in chapter 5 to describe their functioning as well as the impact on the overall solution that was developed for the client.

7.3 Revisiting SRE and DDI methodologies

The SRE and DDI methodologies were introduced in chapter 6. The SRE method allows assumption lists to be developed in solitude by team members and these are then investigated in an attempt to determine whether any hidden factors have possibly been overlooked. DDI allows for one or more developers to interact directly with the business domain of the client from which these developers would then extract tacit knowledge and other hidden domain factors.

Both these methodologies, when used comprehensively, have the potential to elicit factors – the omission of which in especially highly critical domains – which could cause a project to fail. It should be noted, however, that both methodologies, especially DDI in which developers are introduced to the environment of the client, may become very time consuming exercises. The benefits, nevertheless, still appear to be worth the time spent, especially if one considers the enormous penalty paid in correcting errors during the maintenance phase (Boehm [16]).

A comparison of the two methods indicated that the SRE has the potential to be executed faster than the DDI, since assumption lists are quick to create and discuss in a follow up JAD session. Employing the DDI, on the other hand, may take much more time because developers first need to understand the business environment of the client. This implies that they have to go through a learning phase. This may not be the case with the SRE methodology in which each member only creates an assumption list.

Advantages of the SRE methodology are:

- Team members experience a sudden change of environment. They move from a busy and often noisy JAD session to an environment of solitude. This sudden change in the physical surroundings may stimulate their thought processes on issues not considered during the session.
- Assumptions are written down, thus helping team members to focus on the problem.
- Writing assumptions down helps members to think more broadly.
- Lists allow members to participate more. Some members may not have thought that their assumptions are relevant and thus not have raised them during an active JAD session.
- Lists enhance discussion and are a good documentation method. The team can always refer back to their lists if something is not clear and may even check the reasons for taking certain decisions by referring to the lists.

Disadvantages of the SRE methodology are:

- Assumptions not written down could still be overlooked.
- Active participation from certain clients and users could be lacking.
- Incorrect assumptions could still lead to wrong decisions being made regarding the functionality of the product.
- Lists may contain conflicting information.

Advantages of the DDI methodology are:

- Developers gain insight into the daily operations of the client's business.
- Developers understand how the product needs to function in the live working environment and how users could react to the system.
- Developers' communication is enhanced. They start asking questions from a non-technical point of view. In most cases users would be able to answer these.
- Buying into the process is increased. Both users and clients feel that they are part of the whole software development process and, therefore, take ownership of the system.

Disadvantages of the DDI methodology are:

- Developers may be negative about being exposed to the environment of the client, as they feel more comfortable in their own office setting.
- Clients may not want developers to enter their working environment as it may contain highly sensitive information. They may view allowing developers onto their premises as a risk to the company.
- Some developers may not participate actively in the environment by investigating scenarios and asking relevant questions.
- Users may feel threatened and not present all the information relevant to their domain.

7.4 Conclusion

Current requirements elicitation methodologies do not always tap sufficiently into the hidden domain of clients and users. The SRE and DDI methods have the potential to gain access to these hidden domains and extract factors residing within these domains. If critical domain factors are extracted, the potential to create the software that the client really wants is increased.

It should be noted that it has not been the objective of this dissertation to suggest that the two new methodologies introduced should replace existing requirements elicitation methods or techniques. The aim has rather been to show that the introduction of these methodologies may enhance existing elicitation techniques.

The author of this dissertation trusts that this study may lead to the SRE and DDI methodologies being adopted in industry and applied to software development projects. The SRE and DDI methodologies may also not be a silver bullet – or the final answer – but these methods could help decrease the number of software project failures.

7.5 Future work

Work needs to be done in refining the SRE methodology. One of the important aspects would be to deal with the different formats of information inside the sets drawn up using the SRE method. Conflicting information in different sets should also be studied and ways to resolve these problems could be investigated.

Future work in the area researched by this dissertation could also include a refinement of, as well as a study of the effects of the DDI in a live scenario. The results could then be compared to those of existing requirements elicitation methods. The impact the theory would have on the development of a safety-critical system such as that of a flight simulator system, medical diagnostics system or even a nuclear power plant monitoring system, could also be studied.

A study could also be conducted to address the problems identified in the DDI methodology. This may include looking into the psychological aspects of teams and their personalities. Developers who find interacting with clients and individuals easy could possibly have a higher rate of success employing the DDI method than those who are just technically inclined.

Case studies could also be done in industry to study the effects of using the SRE and DDI methodologies either separately or by combining the strengths of each method.

Appendices

Appendix A

Questions asked to various retailers:

1. Does your current system cater for the business requirements of the company?
2. Did a software development company develop the system or was it developed in house?
3. How were the requirements for the system determined for example through interviews, JAD, prototypes, questionnaires etc.?
4. Was there any interaction between users and developers or analysts?
5. After the requirements were determined, was a specifications document formulated or did the development continue without any formal specifications document?
6. How was the new application introduced to the users?
7. Is there many or have there been many changes to the program since its implementation date?
8. How would you aid an IT expert/Developer to understand your business environment and how would you test if this person does understand your business model?

Responses from various retailers:

Edgars

1. Remember we use a number of systems. Systems that we have are ok, but we can do better. So we have improvement programs and we plan totally new systems as well.
2. We outsource all our system development work, so all systems that needs changes, development are done by outside companies

3. All the above – always involve users.
4. Many interactions by all parties.
5. Yes every aspect of specification and changes on specifications are documented
6. By involving users from start (inception) we have buy-in and support.
7. Yes contact update and changes – don't know what you define as many, but do a significant amount of enhancements and changes
8. Will not test whether he understand business = he will demonstrate his knowledge – will give person access to documentation and allow him access to people that are relevant – will be well planned – get him involved with projects where he can learn.

Woolworths

1. We have various systems – all of which cater for various needs
2. Predominantly, software is bought from a Software developer and then customised for our needs. Some applications are developed in house but this is mainly Web based. Our use of in house software has declined over time
3. Generally as part of a project. A Business case is done first and upon approval the requirements are scoped and eventually end up in a System Requirements Specification. Once the software is installed there is normally a Post Implementation review to ensure that the Business case was met
4. Yes, at all levels
5. A requirement was formulated
6. Training & documentation
7. Yes, mainly enhancements and new functionality
8. No Response

Truworths

1. Yes
2. Most of systems are "in-house" developed applications but we do work with external vendors depending on the system architecture requirements

3. Meetings, PDW's (Project Definition Workshops) and questionnaires
4. We meet with the business to discuss their requirements after which IT run with the project to ensure that Technical requirements are met.
5. After the PDW, a specifications document will be prepared and then we will proceed to a Proof of Concept once all parties have agreed.
6. Proof of Concept demo and Training Workshops
7. Customization of applications happen on a regular basis to meet business needs.
8. No Response

Pick ´n Pay

Pick ´n Pay has embarked on a SAP ERP implementation. There are several supporting peripheral systems e.g. Point of Sale (POS) systems, back door receiving systems, which are a combination of in house development and purchased applications.

Pick ´n Pay utilises a system delivery framework that ensures the following:

- Business Analysts consult with users to determine their business requirements. This is done in a variety of ways but mostly through one-on-one user interviews and workshops.
- The Business Analyst will then compile a business requirements specification that will be reviewed and agreed with the user. Once user agreement is obtained, this specification document is presented to the development team for technical specification and development to commence.
- Once development is concluded, a formal system test cycle will take place.

New applications are introduced to the identified super users through user acceptance testing and to the balance of the user community via user training.

Since the first implementation cycle of SAP ERP within Pick ´n Pay, certain areas have been highlighted for enhancements. These enhancements form part of the

standard SDLC maintenance model and will be approached in the same manner described above.

Hyperama

1. No. It is a continuous changing environment that needs to be updated and improved as Processes and technologies change.
2. Inhouse 70% Purchased 30%. This will change to where the largest % will be purchased.
3. A combination of these but our JAD workshops are not as formal as they should be.
4. Yes without this there is no proper solution.
5. This is a formal process according to standards and with proper documentation.
6. Normally by letting them participate in the testing phases as part of the approval process to validate that the system meets their requirements
7. We have thousands of programs but basically 62 big systems. The older systems have a higher % of change. We have a calculation that shows us when the system is getting to old and needs replacement.
8. No Response

Appendix B

Schemas below need to be read in conjunction with chapter 5.

The following operation adds a warehouse (w?) to the system.

AddWarehouse

Δ DistributionSystem
w? : WAREHOUSE
reply! : REPLY

---

w? ∉ Warehouse
Warehouse' = Warehouse ∪ {w?}
reply! = warehouse_added

Δ DistributionSystem indicates that there will be a change in the state of the system. The warehouse should not form part of the set of warehouses already known to the system. The value or reply! indicates that the operation was successful.

If the warehouse is known to the system the scenario is captured by WarehouseRecordExist.

WarehouseRecordExist

Ξ DistributionSystem
w? : WAREHOUSE
reply! : REPLY

---

w? ∈ Warehouse
reply! = cannot_add_record_exist

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the warehouse is already known to the system.

The following operation deletes a warehouse from the system.

DelWarehouse

Δ DistributionSystem
w? : WAREHOUSE
reply! : REPLY

w? ∈ Warehouse
Warehouse' = Warehouse \ {w?}
reply! = warehouse_deleted

Δ DistributionSystem indicates that there will be a change in the state of the system. The warehouse should form part of the set of warehouses already known to the system in which case the warehouse is removed from this set. The value of reply! indicates that the operation was successful.

If the warehouse is not known to the system the scenario is captured by WarehouseNotFound.

WarehouseNotFound

Ξ DistributionSystem
w? : WAREHOUSE
reply! : REPLY

w? ∉ Warehouse
reply! = warehouse_not_found

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the warehouse is not known to the system.

The following operation allows a user to edit an existing warehouse.

EditWarehouse

```
¿       Δ DistributionSystem
¿       w?, new_w?  : WAREHOUSE
¿       reply! : REPLY
□_____________________
¿
¿       w? ∈ Warehouse
¿       ∃ w: WAREHOUSE • w = w? ∧  w ' = new_ w?    // Assign new details
¿                                                   // to existing record.
¿       reply! = warehouse_updated
|_____________________________________
```

A warehouse (w?) whose details is to be edited is given as input to the system. The new details (new_w?) to be entered are also given as input. The existing warehouse is retrieved (∃ w : WAREHOUSE • w = w?) and his details are replaced as requested (w ' = new_w?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a warehouse is already known to the system or not.

```
¿___IsWhouseOnSys_____________________
¿
¿       Ξ DistributionSystem
¿       w? : WAREHOUSE
¿       reply! : REPLY
□_____________
¿
¿       (w? ∈ Warehouse ∧
¿       reply! = Yes)
¿       ∨
¿       (w? ∉ Warehouse ∧
¿        reply! = No)
|_____________________________________
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the warehouse is known to the system or not.

The following operation adds a vendor (v?) to the system.

```
¿___AddVendor_____________________
¿
¿       Δ DistributionSystem
¿       v? : VENDOR
```

reply! : REPLY

---

v? ∉ Vendor
Vendor' = Vendor ∪ {v?}
reply! = vendor_added

Δ DistributionSystem indicates that there will be a change in the state of the system. The vendor should not form part of the set of vendors already known to the system. The value or reply! indicates that the operation was successful.

If the vendor is known to the system the scenario is captured by VendorRecordExist.

¿___VendorRecordExist_____

Ξ DistributionSystem
v? : VENDOR
reply! : REPLY

---

v? ∈ Vendor
reply! = cannot_add_record_exist

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the vendor is already known to the system.

The following operation deletes a vendor from the system.

¿___DelVendor_____

Δ DistributionSystem
v? : VENDOR
reply! : REPLY

```
ʔ
ʔ       v? ∈ Vendor
ʔ       Vendor' = Vendor  \ {v?}
ʔ       reply! = vendor_deleted
└──────────────────────────────
```

Δ DistributionSystem indicates that there will be a change in the state of the system. The vendor should form part of the set of vendors already known to the system in which case the vendor is removed from this set. The value of reply! indicates that the operation was successful.

If the vendor is not known to the system the scenario is captured by VendorNotFound.

```
¿___VendorNotFound _____________________
ʔ
ʔ       X DistributionSystem
ʔ       v? : VENDOR
ʔ       reply! : REPLY
□_______________
ʔ
ʔ       v? ∉ Vendor
ʔ        reply! = vendor_not_found
└──────────────────────────────
```

X DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the vendor is not known to the system.

The following operation allows a user to edit an existing vendor.

```
¿___EditVendor _________________________
ʔ
ʔ       Δ DistributionSystem
ʔ       v?, new_v?  : VENDOR
ʔ       reply! : REPLY
□_____________________
ʔ
```

```
ʔ       v? ∈ Vendor
ʔ       ∃ v : VENDOR • v = v? ∧   v ' = new_ v?          // Assign new details
ʔ                                                        // to existing record.
ʔ       reply! = vendor_updated
└──────────────────────────────────
```

A vendor (v?) whose details is to be edited is given as input to the system. The new details (new_v?) to be entered are also given as input. The existing vendor is retrieved (∃ v : VENDOR • v = v?) and his details are replaced as requested (v' = new_v?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a vendor is already known to the system or not.

```
¿___IsVenOnSys _____________________
ʔ
ʔ       Ξ DistributionSystem
ʔ       v? : VENDOR
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       (v? ∈ Vendor ∧
ʔ       reply! = Yes)
ʔ       ∨
ʔ       (v? ∉ Vendor ∧
ʔ        reply! = No)
└──────────────────────────────────
```

x DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the vendor is known to the system or not.

The following operation adds a product (p?) to the system.

```
¿___AddProduct_____________________
ʔ
ʔ       Δ DistributionSystem
ʔ       p? : PRODUCT
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       p? ∉ Product
```

Product' = Product ∪ {p?}
reply! = product_added

Δ DistributionSystem indicates that there will be a change in the state of the system. The product should not form part of the set of products already known to the system. The value or reply! indicates that the operation was successful.

If the product is known to the system the scenario is captured by ProductRecordExist.

ProductRecordExist

Ξ DistributionSystem
p? : PRODUCT
reply! : REPLY

p? ∈ Product
reply! = cannot_add_record_exist

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the product is already known to the system.

The following operation deletes a product from the system.

DelProduct

Δ DistributionSystem
p? : PRODUCT
reply! : REPLY

p? ∈ Product
Product' = Product \ {p?}
reply! = product_deleted

Δ DistributionSystem indicates that there will be a change in the state of the system. The product should form part of the set of products already known to the system in which case the product is removed from this set. The value of reply! indicates that the operation was successful.

If the product is not known to the system the scenario is captured by ProductNotFound.

```
¿___ProductNotFound _____________________
?
?       X DistributionSystem
?       p? : PRODUCT
?       reply! : REPLY
?_______________
?
?       p? ∉ Product
?       reply! = product_not_found
|_____________________________________
```

X DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the product is not known to the system.

The following operation allows a user to edit an existing product.

```
¿___EditProduct _________________________
?
?       Δ DistributionSystem
?       p?, new_p?  : PRODUCT
?       reply! : REPLY
?_____________________
?
?       p? ∈ Product
?       ∃ p : PRODUCT • p = p? ∧  p' = new_p?    // Assign new details
?                                                // to existing record.
?       reply! = product_updated
|_____________________________________
```

A product (p?) whose details is to be edited is given as input to the system. The new details (new_p?) to be entered are also given as input. The existing product is retrieved (∃ p : PRODUCT • p = p?) and his details are replaced as requested (p' = new_p?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a product is already known to the system or not.

¿___IsProdOnSys _____

ʔ

ʔ Ξ DistributionSystem
ʔ p? : PRODUCT
ʔ reply! : REPLY

ʔ

ʔ (p? ∈ Product ∧
ʔ reply! = Yes)
ʔ ∨
ʔ (p? ∉ Product ∧
ʔ reply! = No)

_____

x DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the product is known to the system or not.

The following operation adds a purchase order (po?) to the system.

¿___AddPurchaseOrder _____

ʔ

ʔ Δ DistributionSystem
ʔ po? : PURCHASEORDER
ʔ reply! : REPLY

ʔ

ʔ po? ∉ PurchaseOrder
ʔ PurchaseOrder' = PurchaseOrder ∪ {po?}
ʔ reply! = order_added

_____

Δ DistributionSystem indicates that there will be a change in the state of the system. The purchase order should not form part of the set of purchase orders already known to the system. The value or reply! indicates that the operation was successful.

If the purchase order is known to the system the scenario is captured by PurchaseOrderRecordExist.

PurchaseOrderRecordExist

Ξ DistributionSystem
po? : PURCHASEORDER
reply! : REPLY

---

po? ∈ PurchaseOrder
reply! = cannot_add_record_exist

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the purchase order is already known to the system.

The following operation deletes a purchase order from the system.

DelPurchaseOrder

Δ DistributionSystem
po? : PURCHASEORDER
reply! : REPLY

---

po? ∈ PurchaseOrder
PurchaseOrder' = PurchaseOrder \ {po?}
reply! = order_deleted

Δ DistributionSystem indicates that there will be a change in the state of the system. The purchase order should form part of the set of purchase orders already known to

the system in which case the purchase order is removed from this set. The value of reply! indicates that the operation was successful.

If the purchase order is not known to the system the scenario is captured by PurchaseOrderNotFound.

PurchaseOrderNotFound

Ξ DistributionSystem
po? : PURCHASEORDER
reply! : REPLY

---

po? ∉ PurchaseOrder
reply! = order_not_found

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the purchase order is not known to the system.

The following operation allows a user to edit an existing purchase order.

EditPurchaseOrder

Δ DistributionSystem
po?, new_po? : PURCHASEORDER
reply! : REPLY

---

po? ∈ PurchaseOrder
∃ po : PURCHASEORDER • po = po? ∧ po ' = new_ po? // *Assign new*
// *details to*
// *existing record.*
reply! = order_updated

A purchase order (po?) whose details is to be edited is given as input to the system. The new details (new_po?) to be entered are also given as input. The existing

purchase order is retrieved (∃ po : PURCHASEORDER • po = po?) and his details are replaced as requested (po' = new_po?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a purchase order is already known to the system or not.

```
¿___IsPurchaseOrderPlaced _____________
ʔ
ʔ       Ξ DistributionSystem
ʔ       po? : PURCHASEORDER
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       (po? ∈ PurchaseOrder ∧
ʔ       reply! = Yes)
ʔ       ∨
ʔ       (po? ∉ PurchaseOrder ∧
ʔ        reply! = No)
│_____________________________
```

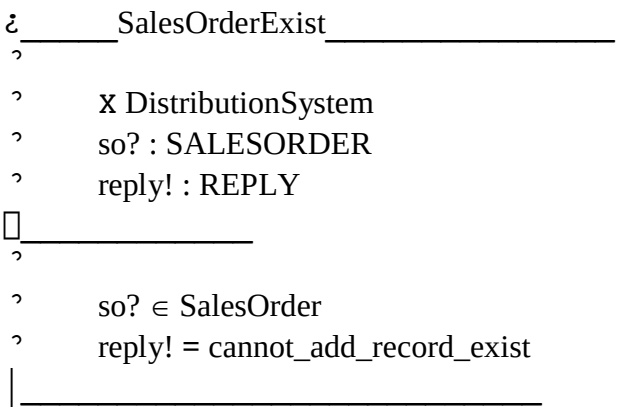
Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the purchase order is known to the system or not.

The following operation adds a sales order (so?) to the system.

```
¿_____ AddSalesOrder_____________
ʔ
ʔ       Δ DistributionSystem
ʔ       so? : SALESORDER
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       so? ∉ SalesOrder
ʔ       SalesOrder' = SalesOrder ∪ {so?}
ʔ       reply! = order_added
│_____________________________
```

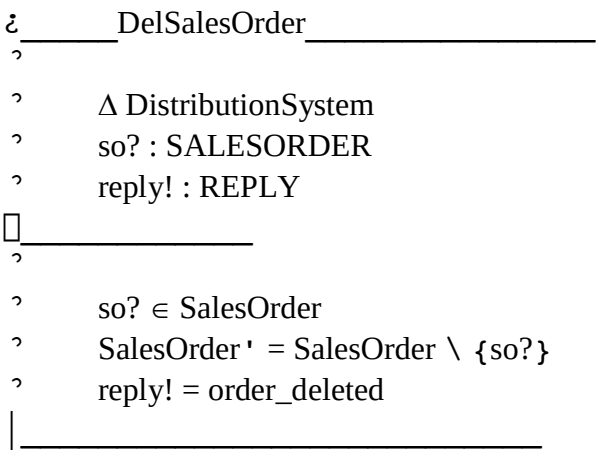
Δ DistributionSystem indicates that there will be a change in the state of the system. The sales order should not form part of the set of sales orders already known to the system. The value or reply! indicates that the operation was successful.

If the sales order is known to the system the scenario is captured by SalesOrderExist.



Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user of the system that the sales order is already known to the system.

The following operation deletes a sales order from the system.



Δ DistributionSystem indicates that there will be a change in the state of the system. The sales order should form part of the set of sales orders already known to the system in which case the sales order is removed from this set. The value of reply! indicates that the operation was successful.

If the sales order is not known to the system the scenario is captured by SalesOrderNotFound.

```
¿_____SalesOrderNotFound_____________
ʔ
ʔ       Ξ DistributionSystem
ʔ       so? : SALESORDER
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       so? ∉ SalesOrder
ʔ       reply! = order_not_found
│_____________________________________
```

Ξ DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user that the sales order is not known to the system.

The following operation allows a user to edit an existing sales order.

```
¿_____EditSalesOrder_________________
ʔ
ʔ       Δ DistributionSystem
ʔ       so?, new_so?  : SALESORDER
ʔ       reply! : REPLY
□_____________
ʔ
ʔ       so? ∈ SalesOrder
ʔ       ∃ so : SALESORDER • so = so? ∧   so ' = new_ so?        // Assign new
details
ʔ                                                               // to existing record.
ʔ
ʔ       reply! = order_updated
│_____________________________________
```

A sales order (so?) whose details is to be edited is given as input to the system. The new details (new_so?) to be entered are also given as input. The existing sales order is retrieved (∃ so : SALESORDER • so = so?) and his details are replaced as requested (so ' = new_so?). The value of reply! indicates a successful operation.

The following robust operation enquires whether a sales order is already known to the system or not.

```
¿_____IsSalesOrderPlaced _________________
?
?       Ξ DistributionSystem
?       so? : SALESORDER
?       reply! : REPLY
□_____________
?
?       (so? ∈ SalesOrder ∧
?       reply! = Yes)
?       ∨
?       (so? ∉ SalesOrder ∧
?        reply! = No)
|_____________________________
```

x DistributionSystem above indicates that there is no change to the state. The value of reply! indicates to the user if the sales order is known to the system or not.

Appendix C

Towards a Methodology to Elicit Tacit Domain Knowledge from Users

WERNHER R. FRIEDRICH AND JOHN A. VAN DER POLL

University of South Africa, Pretoria, South Africa

wernher@doc.gov.za
VDPOLJA@unisa.ac.za

Abstract

This paper seeks to address a problem ubiquitous in many software development environments today, namely, building software from requirements that are incomplete and not fully understood, thereby creating products that are either faulty or ultimately not being used at all. This gap that exists between software engineers and clients is highlighted in this paper and suggestions on how to overcome the identified gap are presented. The proposed methodology is to introduce developers into the client's environment, which can be more time consuming and more resource intensive than traditional knowledge elicitation methods, but has the potential to satisfy more of a user's needs in the long run. It also does not seek to replace any of the existing elicitation methods; rather it is complementary to knowledge elicitation techniques

currently used by software engineers as well as to enhance current understanding of such processes.

Keywords: Software Engineering, Requirements/Specification, Elicitation methods, Rapid prototyping, Human factors, Software Psychology, Domain knowledge, Domain expert, Tacit knowledge.

Introduction

> 'Tacit knowledge according to (Blandford & Rugg, 2002) is 'knowledge which is not accessible to introspection via any elicitation technique.'

The reason for not being able to gain easy access to this deep level of knowledge is because it is what we humans call 'experience', something which we gain through time and exposure to different environments and situations. It is precisely the experience factor that creates experts in certain fields around specific subjects or subject matter. For example, a mathematician is an expert on how to solve mathematical problems by applying mathematical formulas and reasoning to a given problem. A medical doctor is an expert in the field of medicine and human diseases, so when a patient displays signs of a particular illness a doctor would sometimes perform tests to reach a plausible diagnosis, in order to treat the patient correctly.
Both of the examples mentioned have a certain amount of *tacit* knowledge included in them and it is this tacit knowledge that is hard to convey successfully from an expert to a non-expert. Hudlicka (1996) states 'In other words, neither experts nor the intended system users are always able to state their knowledge and system requirements succinctly in response to direct questions.' If a software engineer does not grasp the domain and the intricacies of the said domain correctly, he or she could end up developing software that is not useful to the client at all, i.e. resulting in the user uttering the well-known phrase 'this isn't what we asked for' (Hudlicka, 1996).
We begin the paper by briefly revisiting the work of Fred Brooks (1987), arguing that the same problems touched on above are still in force two decades later. A brief introduction to three requirements elicitation techniques, namely, certain parts of the Unified Modelling Language (UML), Joint Application Development (JAD) and Rapid Application Development (RAD) are presented and possible disadvantages are given. Through the use of a small case study we illustrate, using UML and JAD how tacit knowledge remains hidden for a software developer who is not necessarily an expert of the underlying domain. Thereafter we propose a simple set-theoretic notation to reason about the problem of different knowledge domains of developers and users. An enhancement to JAD and a lightweight formalisation of an algorithm to enhance the elicitation of tacit knowledge is presented, followed by further mechanisms to elicit such tacit knowledge. We conclude with an analysis and some pointers for future work in this area.

The Problem Domain

In his classic text *No Silver Bullet, Essence and Accidents of Software Engineering*, Brooks (1987) wrote the following: 'There is no single development, in either technology or in management technique, that by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity.' Brooks identified a number of problems that were being experienced by software development companies and organisations worldwide. These problems include late

software, cost overruns, insufficient requirements and documentation and resource constraints.
Although the work by Brooks is dated and has been disagreed with (Cox, 1995; Cox, 1990) the situation had not improved eight years later as reported on in the Standish Group's Chaos report (1995). They researched successes and failures of projects in the United States (US) and came up with the following statistics: Overall, only 16.2% of all projects investigated were completed on-time and on-budget with all requirements as originally specified, implemented. Projects eventually completed but over-budget, over the time estimate and with fewer functionality as originally specified make up 52.7%, while 31.1% of all the projects were cancelled at some point during the development cycle. Executive managers who took part in this survey give three major reasons for project failure: *lack* of user involvement (12.8% of respondents), *incomplete* requirements and specifications (12.3%) and *changing* requirements and specifications (11.8%). All these reasons concern the user and his or her incomplete or changing requirements. It is anticipated that a lack of clear articulation of tacit knowledge from the user's part is much to blame for project failure. A study recently done by a private South-African company (BMC Software, 2004) confirms an equally bleak picture and blames much of project failure on poor communication between developers and users.

Role Players in Industry

In a typical industrial software development environment one often finds the following job titles responsible for developing software and business cases for either internal or external clients:

Systems Architect – responsible for constructing an overall solution, inspecting hardware and software requirements;

Systems Analyst – analyses both any existing and proposed system to ensure that the proposed system will be able to fulfil the requirements as set out by the client;

Business Analyst – responsible to investigate how the system and the client's business need to integrate. The analyst needs to understand the environment and the business of the client before a proposed solution that would enhance the client's operational activities can be constructed;

Software Developer / Programmer – person(s) responsible for the actual coding and development of the system. In this paper this group of people are referred to simply as 'developers';

Software Tester – person(s) responsible to test the system functions as per the set out requirements and documentation;

Project Manager – responsible for the project as a whole and has to ensure that the resources are used in the best possible manner; also draws up a project plan with guidelines and deadlines.

In some software development organizations there is a one-to-one mapping between an individual and the above roles since each role has some form of specialisation

attached to it, but in many other environments a particular individual can fulfil more than one such role. For example, the software developer would in some cases also be the systems analyst and even the architect, all depending on budgetary constraints and the availability of qualified staff.

Requirements Elicitation Techniques

A number of requirements elicitation techniques have been developed to extract requirements from a user. Some of these are rapid prototyping (Friedrich, 2005; Gordon & Bieman, 1995), Joint Application Development (JAD) (Wood & Silver, 1995; Hughes & Cotterell, 2006), Storyboarding (Snyder, 2001), Rapid Application Development (RAD) (Hughes & Cotterell, 2006; Pressman, 2005) and some parts of UML (Ambler, 2004; Jalloul, 2004). In this paper we will briefly focus on the use of JAD, RAD and UML and propose incremental enhancements to the elicitation mechanisms of some of these techniques.

JAD Workshops

JAD is a method where a software development team and clients of the proposed system all come together in a workshop environment (preferably a single room called a 'clean room') to brainstorm and discuss different solutions to the new system (Wood & Silver, 1995). JAD workshops are not only used to create ideas for new systems beforehand, but also to aid the software development team to raise issues and concerns at later stages while the project is already progressing, attempting to adhere to the client's requirements and expectations as was set out initially. The very power of a JAD session lies in the fact that it is highly interactive and that the users are involved right from the start as well as throughout the duration of the entire project. This aims to address the concern raised by executives in the Chaos Report, namely, that users are not actively involved during project development. Involving users in an interactive mode throughout a project may very well increase the probability of it being a success. Figure 1 shows a picture of a typical clean room during a JAD workshop.

Figure 1: Example of a JAD workshop

There are some disadvantages in using JAD sessions:

Owing to the highly interactive mode of JAD and complicated group dynamics one may have that one or more individuals force their own viewpoints onto other members, resulting in optimal decisions not being reached.

If there are too many JAD sessions while the project is progressing then users may develop a feeling that the developers are shifting their work and responsibility onto the users. This may lead to resentment and user withdrawal, having a reverse effect of what JAD is trying to achieve.

These disadvantages may lead to users not being very alert to opportunities during a JAD workshop where tacit domain knowledge needs to be introduced to the developers.

UML: Use Case Models

Arguably the Use Case Model (UCM) is the most useful part of UML (Jalloul, 2004; Booch, Jacobson & Rumbaugh, 1999) for eliciting requirements from a client. A UCM is constructed by identifying actors (i.e. role players in the client's world), thereafter identifying a set of use cases, via scenario construction (Bahrami, 1999) for each actor and finally relating the actors and use cases together with elements from a communicate relation (Jalloul, 2004). A UCM is, therefore, drawn to aid software development teams in understanding the dynamics in which the system will be used and what the client's expectation of the system is. The UCMs also give the team some up-front and visual clarity as to what they need to build. Scenarios are usually easy to create and should a scenario not fit into the larger project scheme it can simply be removed.

The use of UCMs agrees with an important design principle by Norman (1998):

> 'In each state of the system, the user must readily see and be able to do the allowable actions. The visibility acts as a suggestion, reminding the user of possibilities and inviting the exploration of new ideas and methods.'

Below is an example of a UCM involving a single actor and three use cases:

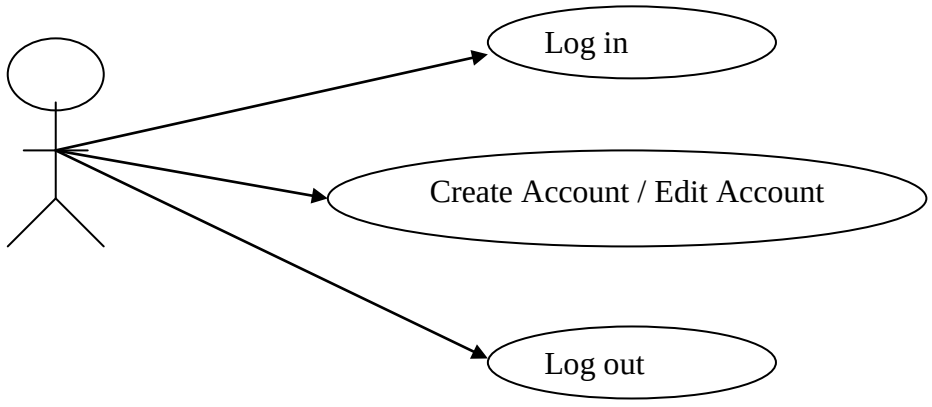


Figure 2: A Simple UCM

The use of a UCM is less interactive than a JAD workshop discussed above and may, therefore, also fail to elicit the necessary tacit knowledge from users. Note, however, that such failure in essence stems from a reverse of the criticism given of JAD workshops.

Prototyping Using RAD

Some software development teams use RAD, a technique that enables fast user interface screen design but without any underlying functionality. In essence RAD tools help create screens while the developers and client discuss various fields and buttons that are needed. Like UCMs, RAD adds visual clarity to scenarios and dialogue structures. An example of a screen developed by a RAD tool is given in Figure 3.

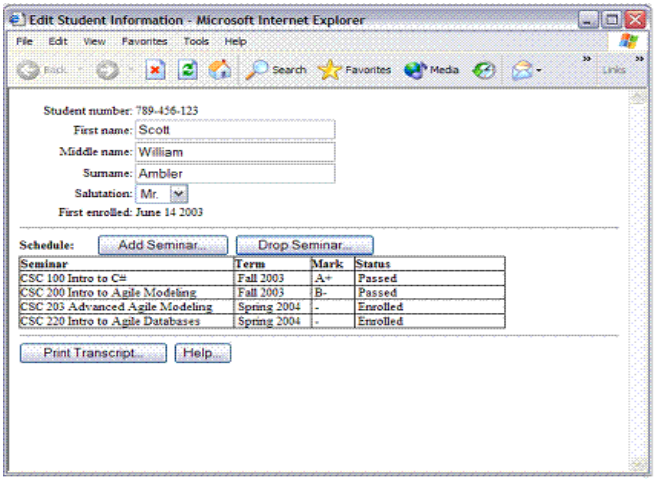


Figure 3: Example of a screen developed with a RAD tool

Upon seeing the actual screen layout and content the client starts to gain a deeper understanding of the high-level mechanics of the proposed system and can, therefore, make constructive suggestions and changes to the requirements. There is, however, a possible danger in taking a user through a RAD exercise: Since the screens have no underlying functionality yet, developers tend to spend much time making them visually attractive using well-established human-computer interaction (HCI) layouts of fields and colours for headings, etc. This tends to distract the user from the real task at hand, namely, to evaluate the *functional* usability of each screen. This may result in distracting the client from coming up with important tacit knowledge much needed by the developer.

In the next section we develop a small case study and show how the absence of tacit knowledge results in a system less useful than would otherwise have been the case.

A Case Study

Suppose a client contracts a software development company to develop a program for a new vehicle insurance product. The program has to perform calculations depending on an option a user selects for his or her insurance needs. If the user selects a value added product then the premium on his/her vehicle would be calculated differently than would have been the case if the product had not been opted for.

Suppose the development team takes the following decisions:

Every step in the software development life cycle (SDLC) is to be fully documented so that every decision taken could be justified throughout. Such documentation would aid developers as well as software testers afterwards.

UML in conjunction with various JAD workshops are to be used. It is expected that the JAD sessions would facilitate their understanding of specific requirements for the system as well as the design of a flexible solution.

During a JAD workshop the development team raised direct questions, trying to clarify specific requirements.

Beginning of JAD session:

Developer #1: What are the options available to a user?

Client: Users may select either full comprehensive cover or limited cover.

Developer #1: Could you explain these two options?

Client: Yes. Full comprehensive cover is calculated by taking the full amount requested for cover, followed by multiplying that amount by 0.03 and then dividing it by 12 to get a monthly premium. E.g. (R250 000 * 0.03) / 12 = R625 per month.
Limited cover is calculated by taking the full amount requested for cover and multiplying that amount by 0.01 and then dividing it by 12 to get a monthly premium. E.g. (R250 000 * 0.02) / 12 = R416.67 per month.

Developer #2: Must there be security build into the system?

Client: Yes, only authorised call centre agents must be allowed to login in and give valid quotes to a client.

Developer #3: Should every quote you give be kept for the sake of future queries?

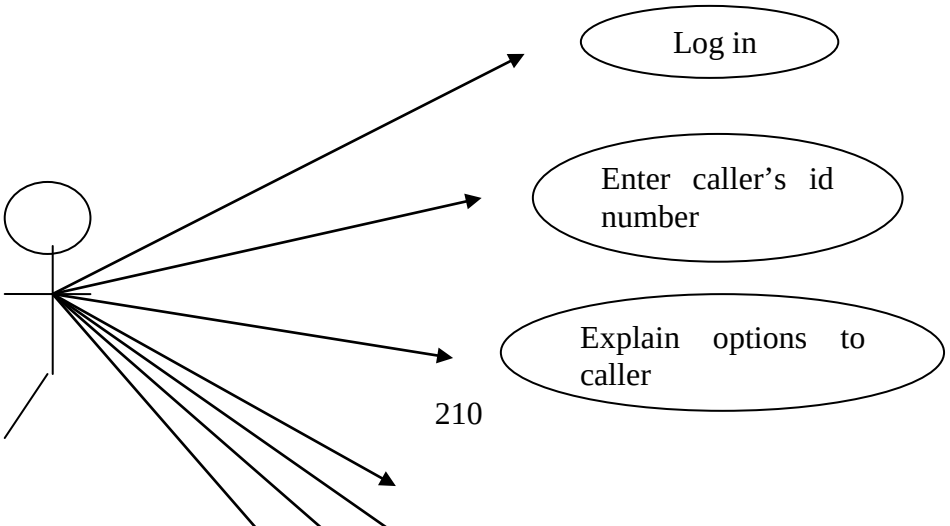
Client: Yes, we need to be able to retrieve the quote even after 3 months, should the client decide to take up the policy.

Developer #3: How do you currently keep track of quotes?

Client: We use a reference number; we take the caller's id number as the reference number.

End of JAD session

After the JAD session the developers created the UCM in Figure 4:



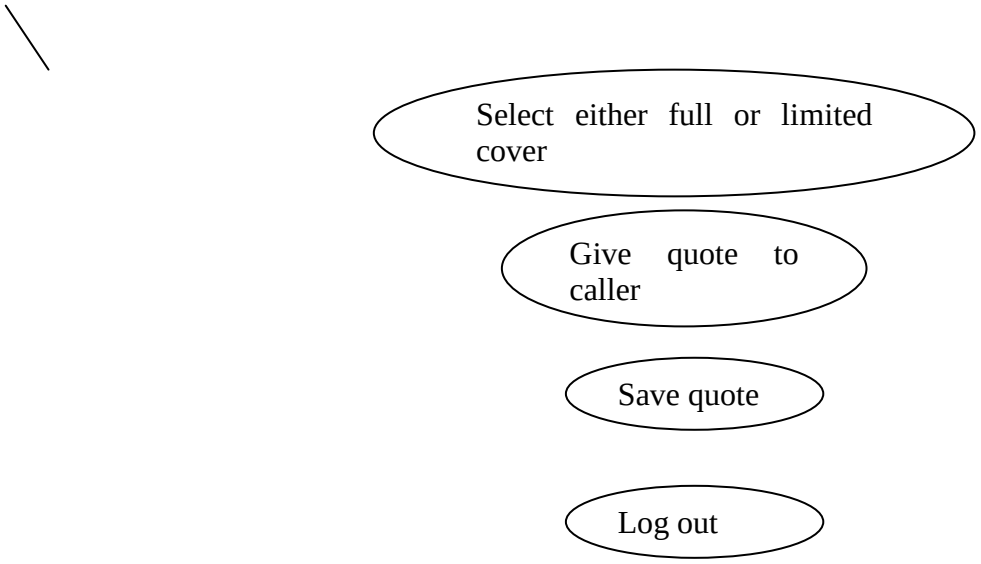


Figure 4: UCM of vehicle insurance

Thereafter the development team drew up a functional specification, viz:

A log-in screen for the call centre agent to have access to the system is to be displayed.

Agent's log-in id is to be checked against the system to ensure only authorised access.
If agent fails 5 times to login in, lock the agent out completely.
If agent is locked out, agent needs to call the administrator to unlock his/her user id.
System will not use passwords, only login security codes which agents are not allowed to share.
Login security screen will only display stars (*), when an agent enters a security code.
A field for entering the caller's identity number is to be displayed.
Identity number verification and control needs to be built in, to ensure data integrity should the quote become a valid policy.
An option, preferably radio buttons, to select either full or limited cover must be available.
The system must have an option to save quotes.
If an agent made a mistake, the agent must have the option to correct the mistake before saving the quote.
The call centre agent must be able to log out in order to prevent unauthorised access to the system.

During the next phase of the project developers then designed the system, incorporating all the requirements of the system, elicited during the JAD sessions and UML use cases. Following the design phase is the implementation phase in which the physical programming was done. The last phase was testing and the new system was thoroughly tested against the system's specification and documentation. The system was then ready to be installed in the client's business environment. Training was given to all call centre agents on the use of the new system and the system's functionality was clearly explained to all stakeholders. The developers also developed documentation and gave a copy to each agent.

Following one week after delivery of the system, the client comes back to the developers and claims that the system is not functioning correctly, because *it does not cater for an out of the ordinary scenario needed by the insurance company*. The scenario involves the insurance company giving more competitive quotes to their customers to not lose them as clients. Somewhat surprised the developers refer back to the documentation of the system and establish that it is not a system functionality problem, since the system was never required to do the out of the ordinary scenario. The client responds by saying that it is normal and *common* practice in their business environment (i.e. vehicle insurance) to have such an option. In fact, it is considered to be such an obvious facility that our client did not even mention it.

Neither the UCMs nor the JAD sessions revealed this tacit, common-practice, domain knowledge. This is just a small example of tacit knowledge not being uncovered, because the
client never thought of mentioning the (obvious) scenario,
developers did not ask the right questions,
developers did not know about the common practice and did not incorporate it into the specification of the system,
client did not point out the problem in the system and made certain assumptions about what the developers ought to know (maybe because of inexperience with such systems, e.g. being a first-time client).

The client (i.e. the insurance company), therefore, wanted the system to be able to also cater for a query coming from a client of the insurance company who already has been doing business with the company for at least 3 years. In such a case the formula should be augmented to calculate a much lower quote be it for full or limited cover as follows:

If the client of the insurance company is on *full cover* and requests a quote the final amount should only be 50% of the quoted amount. In this case full comprehensive cover is calculated by taking the full amount requested for cover and multiplying that amount by 0.03 and then dividing it by 12 to get a monthly premium. E.g. (R250 000 * 0.03) / 12 = R625 per month. If the client has been insured by the company for at least 3 years, then the final premium is R625*50% = R312.50.

If the client of the insurance company is on *limited cover* and requests a quote the final amount should only be 60% of the quoted amount. In this case limited cover is calculated by taking the full amount requested for cover and multiplying that amount by 0.01 and then dividing it by 12 to get a monthly premium. E.g. (R250 000 * 0.02) / 12 = R416.67 per month. If the client has been insured by the company for at least 3 years, then the final premium is R416.67*60% = R250.00.

Note that in both cases an additional, simple calculation had to be performed on the premium calculated, rather similar to adding VAT (value-added tax) to the basic price of an item.

Our case study is a rather simple illustration of how clients and software developers after having used techniques such as UML and JAD, still run the risk of ending up with a system not fulfilling the clients' expectations, i.e. not correctly simulating the business environment for which it was designed. Integrated and standard business practices do not always come to light, despite using established elicitation techniques.

Furthermore, the various role players mentioned under the heading 'Role Players in Industry' above, tend to make assumptions and create their own view of the environment as the project progresses. In fact, Dix, Finlay, Abowd and Beale (1998) highlight this problem as follows:

> 'Other errors result from incorrect understanding, or model, of a situation or system. People build their own theories to understand the causal behaviour of systems.'

In the next section we introduce mechanisms aimed at addressing some of the problems mentioned above.

Towards The Elicitation of Common Domain Knowledge

Although JAD workshops and some parts of UML do aid in eliciting some domain knowledge, they often do not elicit tacit or hidden domain knowledge effectively. When discussions and brainstorming take place in JAD workshops, these different domains of all involved come together and is not only enlarged by information from other members' knowledge and experiences, but also contribute to the overall understanding of the environment as a whole in which the new or enhanced system will be functioning.

Suppose we have 2 people in the development team (D1and D2) and one client. Initially, then, the situation is often as depicted in Figure 5.

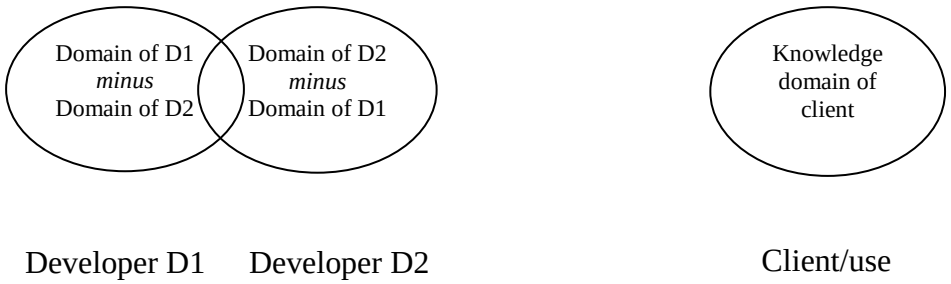
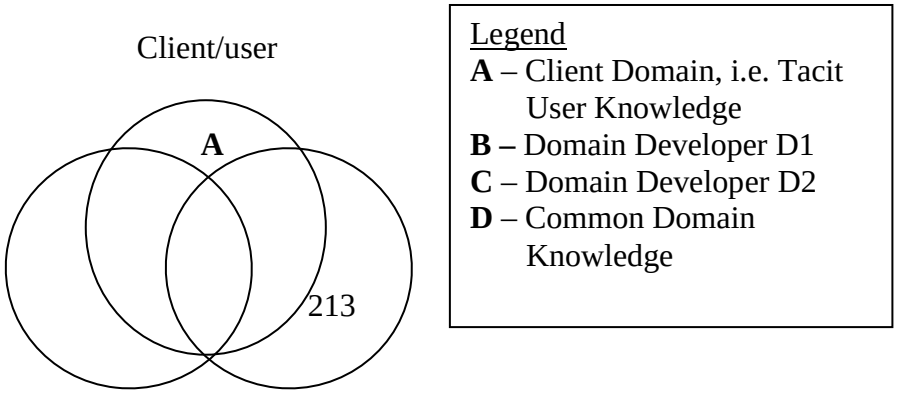


Figure 5: Separate knowledge domains of developers vs user

The domains of developers would normally overlap to some extent and it is possible that both these may also overlap with the knowledge domain of the client. In general, therefore, one would indicate a three-way intersection, but to illustrate the point, namely, that the domains of developers and those of clients very often have little in common, we prefer to draw these with an initial empty intersection

During, for example, a JAD workshop each domain is augmented with knowledge from other domains, i.e. they increasingly overlap with one another and cross-pollination takes place, facilitating a common understanding of the environment for the new system. Figure 6 illustrates this effect.



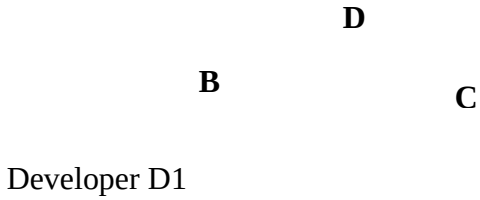


Figure 6: Non-empty intersection of knowledge domains of team members

Figure 6 displays a non-empty intersection of domains of all team members and it is through this intersection that a common understanding is formed among the various team members as a JAD workshop progresses. This common domain can grow in size through brainstorming and the use of a RAD tool to elicit the client's requirements via visualisation. Although JAD workshops can increase such common understanding among all of the team members, the intersection may still not highlight all requirements a client could have. Subsequently, if the client fails to elicit all of his or her tacit requirements (part **A** of the diagram) the end product will not reflect those needs and the following well-known phrase is likely to be verbalized soon after system implementation (Schach, 2002).

> 'I know that this is what I asked for, but it isn't really what I wanted.'

If part **A** in Figure 6 is too large, according to some threshold and contains critical information, despite numerous JAD sessions, the project runs a serious risk of ultimate failure. Part **A** is an important and missing ingredient in the construction of Fred Brooks' (1987) elusive silver bullet for software construction as he writes:

> 'We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems.'

Next we introduce a methodology aimed at allowing system developers to tap more into the tacit knowledge held by users.

Augmenting the Information Content of JAD

During a JAD workshop participants attempt to gain knowledge held by other members of the group. The problem, however, is that members are normally unaware of what they actually know and, therefore, take for granted. In an attempt at further eliciting such tacit knowledge we suggest the following enhancement to a JAD workshop.

Each member of the panel goes on a *retreat*, reflecting on the system as follows:

If the member is a developer, he or she writes down all (as yet) unspoken assumptions he/she makes about the system under development. Developers normally make certain basic assumptions about the systems they develop, e.g. a choice of search algorithm that they never discuss with users who might not understand (in the opinion of the developer) such detail anyway. However, a linear search may influence response times adversely as opposed to a hashing algorithm.

If the member is a user, he or she writes down all expert knowledge of the system which may be taken for granted, e.g. any quote given to a potential client is automatically accompanied by a better quote should the client not be happy with the initial quote. If the client does not complain initially, the better quote is not shown to the client.

During a subsequent JAD workshop, the panel use the lists compiled above to augment the requirements of the system.

Since each panel member draws up these lists in solitude we call this a *solitary requirements elicitation* (SRE) exercise. We hope that the SRE exercise above could become part of a unified model of requirements elicitation as called for by Hickey & Davis (2002):

> 'Although many papers have been written that define elicitation, or prescribe a specific technique to perform during elicitation, nobody has yet defined a unified model of the elicitation process that emphasizes the role of knowledge.'

Assuming *m* developers and *n* users, the above three steps making up a SRE exercise are formalized by Algorithm 1:

Algorithm 1: Gather common domain knowledge from all stakeholders.

Begin

(* Gather assumptions and tacit knowledge for each individual developer and user respectively. *)

for i := 1 **to** m **step** 1 **do** D_i := Unspoken assumptions of developer i;
for j := 1 **to** n **step** 1 **do** U_j := Tacit knowledge of user j;

(* Amalgamate domain knowledge of developers and users into 2 separate sets D and U. *)

$$D := \bigcup_{i:=1}^{m} D_i ; \quad U := \bigcup_{j:=1}^{n} U_j$$

(* Gather common knowledge from SRE exercise into set C. *)

$C := D \cap U$

End.

Note that the purpose of a SRE is to maximise the size of set C above, i.e. *max*(#C) where # represents the cardinality of a set. Next we address another common problem in requirements elicitation, namely, information is given by users but interpreted differently by developers.

The Communication Gap Identified

A further problem leading to incomplete or incorrect requirements emerged from the research recently done by a South-African company, namely, BMC Software (2004). In their study they found that many projects fail because a communications gap exists

between business (users) and the IT industry (developers), resulting in a misalignment of their combined goals. Their MD Brian Whittaker stated the following (ITWEB, 2004):

> 'A communication gap appears to be at the heart of the problem, with SA [South Africa] being among the worst offenders.' 'Leadership is an issue because CEOs are not getting the message across to IT.'

The BMC study was done over a wide spectrum of companies: Manufacturing, Finance, Utilities, Retail/Distribution, Public Sector and Professional Services. Each of these sectors followed a similar pattern, namely, IT projects failing due to the communication gap that exists between business and IT. The report reveals that '25% of companies report losses of between 50,000 Euros and 10 million Euros as a direct result of IT failures.' (BMC Software, 2004).
Figure 7 illustrates the communication gap between what users ask for and what IT specialists understand by the requests.

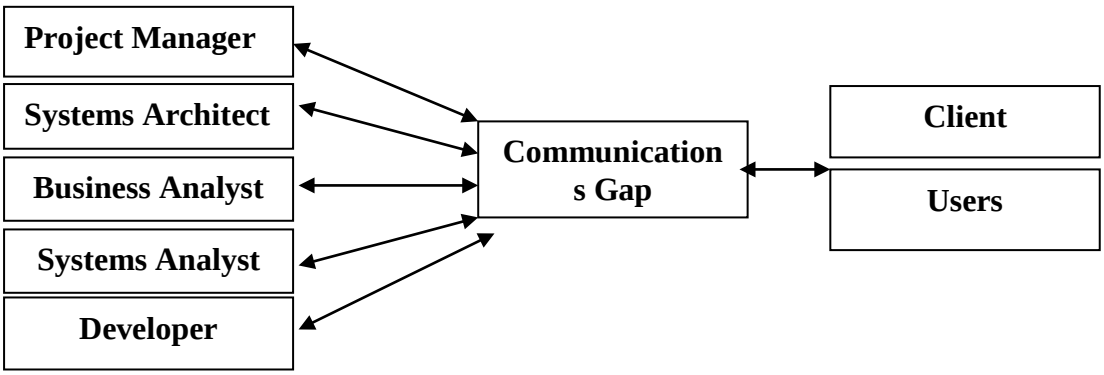


Figure 7: Communication gap between IT specialists and users

Business analysts may have some technical background, but cannot always aid in both the technical as well as the business requirements when analysing a client's business. Developers and programmers are more technically orientated and usually develop software products according to specifications. The system architect's role is that of providing a holistic view of the overall solution, looking at how the complete solution needs to be integrated to make both front end and back end processing possible for the client's business. The project manager in turn will only provide a project plan and make sure enough resources are available to complete the task at hand (Hughes & Cotterell, 2006), A potential trap many project managers fall into is that, should there not be enough resources at any stage during the project they add more personnel to a project, hoping that the project will still be completed on time. However, when adding additional resources to a project it does not always enhance the speed of delivery.

Often it hampers the activities of already productive members, since these new resources need to be brought up to speed with the current status of the project before they can start to be productive. This learning curve that new members of the development team need to go through will take some time, influencing existing members negatively.

Access to the Client's Environment

To overcome the communication gap of the previous section we propose that developers are not kept away from the client's business environment, but interact with that environment as much as possible before any development commences. Many software development organizations still follow a hierarchical approach whereby each person in the team performs a specialist function, embracing a traditional software development life cycle, e.g. the Waterfall methodology (Pressman, 2005). Our small case study above illustrated how communication gaps may occur which may worsen when not all developers are part of every JAD session, but are merely given a written specification afterwards from which to develop the system. Flaws in the handed-down specification will not be addressed by any developer further down the chain, since they take the document at face value, believing it has been sufficiently validated by the business analyst, systems architect and project manager as well as signed off by the client.

If developers are exposed to the client's business environment, they may gain access to tacit knowledge in the domain of the user. Developers will see what users do in their normal operations; they will witness day-to-day tasks that users do not attach any importance to but simply perform in a monotonous and automated way. Developers may then go through a learning curve as to how business is conducted in the domain of the client and understand why certain requirements have been laid down. More importantly, developers may spot possible omissions in the said requirements. Time consuming as this exercise may be, it has major benefits, as a more accurate requirements document may result.

Placing developers in the clients business has the effect of increasing the body of common domain knowledge of developers and the client, taking a further step in solving the problem articulated by Hudlicka (1996):

> 'In other words, neither experts nor the intended system users are always able to state their knowledge and system requirements succinctly in response to direct questions.'

Figures 8a – 8c illustrate this cross-pollination over time. The intension is to maximise the cardinality of the intersection of the sets Developer and Client where Developer is represented by D and Client by U in Algorithm 1 respectively, i.e. we maximise #C, where $C = D \cap U$.

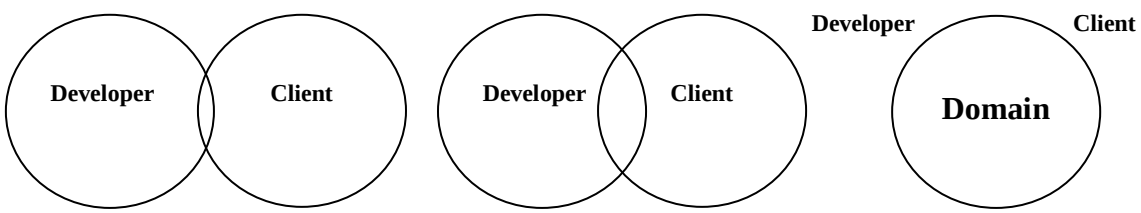


Figure 8a **Figure 8b** **Figure 8c**

Naturally, the ideal situation is when the client and developer is the same person, namely, a developer develops a system for himself/herself. In this case the developer is familiar with most of the scenarios the system needs to be able to handle and can develop the system accordingly. This situation is equally captured by Figure 8c.

Modus Operandi of Developers

How does one then aid the developer to gain a better understanding of the clients' business environment so as to move from Figure 8a towards Figure 8b and if possible towards Figure 8c, even though the developer and client might not be the same person? It is anticipated that a three-tiered approach could be followed in placing a developer in the business environment of the client. This approach embodies the components of *observation*, *enquiry* and interleaved *testing* as follows:

Observation

An important activity for a developer is to observe what happens in the environment of the user. To this end a useful requirements elicitation technique is that of Contextual Enquiry (CI) (Cohene & Easterbrook, 2005). This technique explicitly focuses on the understanding of user needs, desires and work models. Customer needs in their workplace are elicited through interviews (Beyer & Holtzblatt, 1998). Through CI developers are able to 'make observations within the customer's context, discuss the observations as they happen, and determine the implications for the design' (Cohene & Easterbrook, 2005). Such activities would bear fruit also during the implementation phase of the project at which stage developers will have better insight into the product they are developing, what the product's functionality should be and the environment the product has to cater for.

Enquiry

Naturally, developers can also ask questions while being in the environment of the user, especially if they are unsure of why a certain function is being performed. This will aid the developer in the linking of processes on how the organization performs tasks and why actions are being performed in a certain manner. Users can also be interrogated while they are performing these actions.

Testing

Developers can also be tested at certain predefined points during their training in the environment of the user; developers can also be put into practice mode, so as to perform some of the tasks users do, all aimed at familiarizing the developer with the user's environment. In this way the domain of the developer ought to grow closer to that of the users, giving the effect of moving from Figure 8a to Figure 8c. It is anticipated that a developer will gain access to much needed tacit knowledge, so often held captive by a user, without the latter even being aware of it.

Conclusions and Future Work

In this paper we revisited the well-known problem of software failure, i.e. software that is delivered late, over budget and which often does not satisfy the client's need in the end. As a result many software projects are never finished and cancelled by the client before completion. We looked briefly at a number of elicitation techniques,

namely, the use of JAD sessions, RAD elicitations and the use of UCMs in UML. All these techniques are aimed mainly at eliciting requirements from users. Possible problems with some of these techniques were pointed out and as a result we proposed an extension to JAD workshops, namely, a solitary requirements exercise (SRE) at strategic points during requirements gathering. In addition we suggest that a developer be placed in the working environment of the user to gain a better understanding of the system to be developed. An algorithm to model these proposed changes was given and the ultimate goal is to maximise the body of common knowledge between the developers and clients.

Placing a software developer in the environment of a user may not be without problems, since many developers do not feel comfortable working outside of their own domain. Furthermore, with this method the time and resources needed to complete a project may be higher initially, but dividends could pay off towards the end, since the cost of correcting mistakes later on in the project will be reduced. The enormous cost involved in rectifying conceptual errors later on in the project (Potter et al., 1996; Fagan, 1976) may well justify spending quality time during the elicitation phase.

Future work could proceed in a number of areas: In combining (i.e. taking the union) the information in any two sets built up during JAD and a SRE exercise, one may find that 2 pieces of information may either already be present in both sets and agree in semantic content, or in one of the sets but not the other one, or may actually contradict each other. Identifying these cases may be a non-trivial exercise and we anticipate that the work done by Sven Hansson (1999) in belief revision may prove useful in our approach.

Interviews with stakeholders in industry are also on the cards. The purpose of such interviews would be to determine from these users what role they see developers play in their company, should such developers temporarily be seconded to a company. It is anticipated that a useful program with daily, weekly and even monthly routines for these developers could be developed over time.

References

Ambler, S.W. (2004). *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd edition. Cambridge University Press.

Bahrami, A. (1999). *Object-Oriented Systems Development using the Unified-Modelling Language*. McGraw-Hill.

Beyer, H and Holtzblatt, K. (1998). *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Publishers. San Francisco, CA.

Blandford, A. and Rugg, G. (2002). A Case study on integrating contextual information with analytical usability evaluation. *Int J. Human-Computer Studies 57*, 75 - 99.

BMC Software. (2004). *The Communication Gap: The Barrier to Aligning Business and IT.* Study by Winmark.

Booch, G., Jacobson, I. and Rumbaugh, J. (1999). *The Unified Modelling Language User Guide*. Object Technology Series, Addison-Wesley.

Brooks F. P. (1987). No Silver Bullet, *Essence and Accidents of Software Engineering. IEEE Computer*. Vol 20, no 1. April, pp. 10 – 19.

Cohene, T. and Easterbrook, S. (2005). Contextual Risk Analysis for Interview Design, In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pp. 95 – 104.

Cox, B. (1990). There is a Silver Bullet. *BYTE* 15 (10), pp. 209 – 218.

Cox, B. (1995). No Silver Bullet Revisited. *American Programmer Journal*. November, pp. 1 – 8.

Dix, A., Finlay, J., Abowd, G., Beale, R. (1998). *Human Computer Interaction.* Prentice Hall.

Fagan, M. E. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, Vol 15, no 3, pp. 182 – 211.

Friedrich, W. R. (2005). *Prototyping Techniques and Software Development Models*. Unpublished manuscript. UNISA.

Gordon V. S. and Bieman J. M. (1995). Rapid Prototyping Lessons Learned. *IEEE Software*, Vol 12, January, pp. 85 – 95.

Hansson, S. O. (1999). *A Textbook of Belief Dynamics: Theory Change and Database Updating*. Springer.

Hickey, M. and Davis, A. (2002). Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge Intensive Software Development Processes. In *Proceedings of the 36th Hawaii International Conference on System Sciences(HICSS'03)*

Hudlicka E. (1996). Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE 96)*, pp. 4 - 11.

Hughes, B. and Cotterell, M. (2006). *Software Project Management*. 4th edition. McGraw-Hill.

ITWEB. (2004). *Root of IT Failure Exposed.* http://www.itweb.co.za/sections/business/2004/0411111148.asp?O=S&cirestriction=brian%20whittaker Last accessed on 11 November 2004.

Jalloul, G. (2004). *UML by Example*. Cambridge University Press.

Norman, D.A. (1998). *The Design of Everyday Things*. The MIT Press.

Potter B., Sinclair, J. and Till, D. (1996). *An Introduction to Formal Specification and Z*. 2nd edition. Prentice Hall.

Pressman, R. S. (2005). *Software Engineering – A Practitioner's Approach*, 6th edition. McGraw-Hill Companies.

Schach S. R. (2002). *Object-Oriented and Classical Software Engineering*. 5th edition. McGraw-Hill.

Snyder, C. (2001). *Paper Prototyping*. http://www-106.ibm.com/developerworks/library/us-paper Last accessed on 12 January 2003.

STANDISH GROUP. (1995). *Chaos Report*. http://www.projectsmart.co.uk/docs/chaos_report.pdf Last accessed on 19 May 2006.

Wood J. and Silver, D. (1995). *Joint Application Development*. 2nd edition. John Wiley & Sons, Inc.

Index

S

T

U

V

W

Z

References

[1] Ally, M., Darroch, F., Toleman, M., *A Framework for understanding the Factors Influencing Pair Programming Success*, H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp 82 – 91, 2005.

[2] Ambler, S.W., *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd edition, Cambridge University Press, 2004.

[3] American Cancer Society, http://www.cancer.org/docroot/CRI/content/CRI_2_4_3X_How_is_myelodysplastic_syndrome_diagnosed_65.asp

[4] Alvarez, R., *Discourse Analysis of Requirements and Knowledge Elicitation Interviews*, In Proceedings of the 35th Hawaii International Conference on System Sciences, 2002.

[5] Baber, R.L., *The Spine of Software*, John Wiley & Sons Ltd, 1987.

[6] Backhouse, R.C., *Program Construction and Verification*, Prentice Hall, 1986.

[7] Bahrami, A., *Object-Oriented Systems Development using the Unified-Modelling Language*, McGraw-Hill, 1999.

[8] Baumard, P., *Tacit Knowledge in Organizations*, Sage, London, 1999.

[9] Barden, R., Stepney, S., Cooper, D., *Z in Practice*, Prentice Hall, 1994.

[10] Beizer, B., *Black – Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons Inc, 1995.

[11] Beyer, H. and Holtzblatt, K., *Contextual Design: Defining Customer-Centered Systems*, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[12] Bickel, D., *3D real time simulation and VR-tools in the manufacturing industry*, VR for Industrial Applications, Springer, pp 123 - 138, 1998.

[13] Black Box and White Box Testing compared, http://www.scism.sbu.ac.uk/law/Section5/chap3/s5c3p23.html, (Accessed Dec 2005).

[14] Blandford, A. and Rugg, G., *A Case study on integrating contextual information with analytical usability evaluation*, Int J. Human-Computer Studies *57*, pp 75 – 99, 2002.

[15] BMC Software, *The Communication Gap: The Barrier to Aligning Business and IT*, study by Winmark, 2004.

[16] Boehm, B.W., *Software Engineering Economics*, Prentice Hall PTR, 1981.

[17] Boehm, B.W., *A Spiral Model of Software Development and Enhancement*, IEEE Computer 21, pp 61 – 72, May 1988.

[18] Boehm, B.W., *Software Risk Management, Principles and Practices*, IEEE Software 8, pp 32 – 41, January 1991.

[19] Booch, G., *Object Solutions: Managing the object oriented project*, Workingham, Reading, MA, Addison Wesley, 1996.

[20] Booch, G., Jacobson, I. and Rumbaugh, J., *The Unified Modelling Language User Guide*, Object Technology Series, Addison-Wesley, 1999.

[21] Bowen, J., *Formal Specification and Documentation Using Z: A Case Study Approach*, Wiley, John & Sons, 1995.

[22] Brooks, F.P., *No Silver Bullet, Essence and Accidents of Software Engineering*, Computer Vol. 20, no. 1, pp 10 - 19, April 1987.

[23] Bowen, J., *Formal Specification and Documentation using Z: A Case Study Approach*, 2003.

[24] Cambridge Learner's Dictionary, Cambridge University Press, 2004.

[25] Charette, R. N., *Software Engineering Environments*, McGraw-Hill, 1986.

[26] Choi, S.H. and Samavedam, S., *Visualisation of Rapid Prototyping*, Rapid prototyping Journal, Volume 7 - number 2, pp 99 – 114, 2001.

[27] Cohiene, T. and Easterbrook, S., *Contextual Risk Analysis for Interview Design*, In Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05), pp 95 – 104, 2005.

[28] Cooper, A., *The Perils of Prototyping*, http://www.cooper.com/articles.htm, (Accessed June 2004).

[29] Coronel, R. P. and Crockett, K., *Database systems design, implementation and management*, International edition, London, Cengage Learning.

[30] Cox, B., *There is a Silver Bullet*, BYTE 15 (10), pp 209 – 218, 1990.

[31] Cox, B., *No Silver Bullet Revisited*, American Programmer Journal. November, pp 1 – 8, 1995.

[32] Dick, A.J. and Zarnett, B., *Paired Programming and Personality Traits*, In Proceedings of XP2002, Sardinia, Italy, May 26–29, pp82 – 85, 2002.

[33] Dix, A., Finlay, J., Abowd, G., Beale, R., et al., *Human Computer Interaction*, Prentice Hall, 1998.

[34] Dubinsky, Y., Hazzan, O., Keren, A., *Introducing Extreme Programming into a Software Project at the Israeli Air Force*, H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp 19 - 27, 2005.

[35] Enderton, H. B., *Elements of Set Theory*, Academic Press, Inc., 1977.

[36] Examples of Prototypes, http://www.daimi.au.dk/ispace/prototypes.html, (Accessed Dec 2005).

[37] Fagan, M. E., *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal, Vol 15, no 3, pp 182 – 211, 1976.

[38] Freeman C. and Louçã, F., *As Time Goes By : from the industrial revolution to the information revolution*, Oxford University Press, 2002.

[39] Friedrich, W. R., *Prototyping Techniques and Software Development Models*, Unpublished manuscript, UNISA, 2005.

[40] Friedrich, W. R. and van der Poll J. A., *Towards a Methodology to Elicit Tacit Domain Knowledge from Users*, Interdisciplinary Journal of Information, Knowledge and Management, Vol 2, pp 179 - 193, 2007.

[41] Giese, M. and Heldal, R., *From Informal to Formal Specifications in UML*, T. Baar et al. (Eds.): UML 2004, LNCS 3273, pp 197 - 211, Springer-Verlag Berlin Heidelberg, 2004.

[42] Gordon, V. S. and Bieman, J. M., *Rapid Prototyping Lessons Learned*, IEEE Software, Vol 12, January, pp 85 – 95, 1995.

[43] Gronbak, K., Gundersen, K. K., Mogensen, P., Orbak, P., *Interactive Room Support for Complex and Distributed Design Projects*, In Human Computer Interaction INTERACT 01, IOS Press, IFIP, pp 407 – 415, 2001.

[44] Hall, A., *Realising the Benefits of Formal Methods*, K.-K. Lau and R. Banach (Eds.): ICFEM 2005, LNCS 3785, pp 1 – 4, 2005.

[45] Hansson, S. O., *A Textbook of Belief Dynamics: Theory Change and Database Updating*, Springer, 1999.

[46] Hickey, M. and Davis, A., *Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge Intensive Software Development Processes*, In Proceedings of the 36th Hawaii International Conference on System Sciences(HICSS'03), 2002.

[47] Hoffman, R.R., Shadbolt, N. R., Burton, A. M., and Klein G., *Eliciting Knowledgde from Experts: A Methodological Analysis*, Organizational Behaviour and Human Decision Processes, Vol 62, no 2, May, pp 129 – 158, 1995.

[48] Hudlicka, E., *Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods*, In Proceedings of the 2nd

International Conference on Requirements Engineering (ICRE 96), pp, 4 – 11, 1996.

[49] Hughes, B. and Cotterell, M., *Software Project Management*, 4th edition, McGraw-Hill, 2006.

[50] Hughes, B. and Cotterell, M., *Software Project Management*, McGraw-Hill, 2002.

[51] ITWEB, *Root of IT Failure Exposed*, http://www.itweb.co.za/sections/business/2004/0411111148.asp?O=S&cirestriction=brian%20whittaker, Last accessed on 11 November 2004.

[52] Jackson, P., *Introduction to Expert Systems*, Addison Wesley, 1999.

[53] Jalloul, G., *UML by Example*, Cambridge University Press, 2004.

[54] Klopper, R., Gruner, S., Kourie, D. G., *Assessment of a Framework to Compare Software Development Methodologies*, SAICSIT 2007.

[55] Ko, D., *Information Requirements Analysis and Multiple Knowledge Elicitation Techniques with the Pricing Scenario System*, In Proceedings of the 32nd Hawaii International Conference on System Sciences(HICSS'03), 1999.

[56] Kujala, S., Kauppinen, M., Lehtola, L., Kojo, T., *The Role of User Involvement in Requirements Quality and Project Succes*, Proceedings of the Thirteenth IEEE International Conference on Requirements Engineering (RE 2005).

[57] Labuschagne, W., *A User Friendly Introduction to Discrete Mathematics for Computer Science*, UNISA, 1993.

[58] Lightfoot, D., *Formal Specification Using Z*, Palgrave, 2001.

[59] Littman, D.C., *Modeling Human Expertise in Knowledge Engineering: Some Preliminary Observations*, International Journal of Man Machines Studies, Vol 26, No1, pp 81 – 92.

[60] Longman Dictionary of Contemporary English, Pearson Longman, 2006.

[61] Meyer, B., *On Formalism in Specificatios*, IEEE Software, 2(1) p6 - p26, January 1985.

[62] Mitroff, I. I., and Mohrman, S., *The whole wystem is wroke and in desperate need of fixing. Notes on the Second Industrial Revolution*, International Journal Technology Management Vol1, no 1/2, pp 64 – 75, 1986.

[63] Mouton, J., *How to succeed in your Master's & Doctoral Studies, Van Schaik Publischers*, 2001.

[64] Mullin, T.M., *Experts estimation of uncertain quantities and its implications for knowledge acquisition*, IEEE Transactions on Systems, Man, and Cybernetics, 19, pp 616 – 625.

[65] NCC Education Limited, *Systems Development*, International Diploma in Computer Studies, B & Jo Enterprises PTE, Singapore, 2001.

[66] Nielsien, J. and McMunn, D., *The Agile Journey*, H. Baumeister et al. (Eds.) XP 2005, LNCS 3556, pp 28 – 37, 2005.

[67] Norman, D.A., *The Design of Everyday Things*, The MIT Press, 1998.

[68] Northover, M., Northover, A., Gruner, S., Kourie, D. G., Boake, A., *Agile Software Development: A Contemporary Philosophical Perspective*, SAICSIT 2007.

[69] Onuh, S. O., *Rapid Prototyping Integrated Systems*, Rapid Prototyping Journal, Vol 7, no 4, pp 220 – 223, 2001.

[70] O' Leary, T. J. and O'Leary, L. I., *Computing Essentials*, Mcgraw Hill, 2000.

[71] Olsen, H., *The Bottom Line of Prototyping and Usability Testing*, http://guuui.com/posting.asp, (Accessed Dec 2004).

[72] Opiyo, E. Z., Horváth, I., Vergeest, J. S. M., *Computers in Industry*, Volume 49, Issue 2, pp 195 - 215, October 2002.

[73] Polanyi, M., *Knowing and being*, Mind NS 70, pp 458 - 470, 1961.

[74] Potter, B., Sinclair, J. and Till, D., *An Introduction to Formal Specification and Z*, 2nd edition, Prentice Hall, 1996.

[75] Pressman, R. S., *Software Engineering – A Practitioner's Approach*, 6th edition, McGraw-Hill Companies, 2005.

[76] Prototype, http://searchsmb.techtarget.com/sDefinition/0,,sid44_gci1000947,00.html, (Accessed Dec 2005).

[77] Royce, W.W., *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, pp 328-38, May 1989.

[78] Schach, R.S., *An Introduction to Object Oriented Systems Analysis and Design with UML and the Unified Process*, McGraw-Hill, 2004.

[79] Schach, R.S., *Classical and Object-Oriented Software Engineering with UML and C++*, McGraw-Hill, 1999.

[80] Schach, S. R., *Object-Oriented and Classical Software Engineering, 5th edition*, McGraw-Hill, 2002.

[81] Scott, K., *The Unified Process Explained (UML),* Addison Wesley, 2002.

[82] Smialek, M., *From User Stories to Code in One Day?*, H. Baumeister et al. (Eds.) XP 2005, LNCS 3556, pp 38 – 47, 2005.

[83] Snyder, C., *Paper Prototyping*, http://www-106.ibm.com/developerworks/library/us-paper, (Accessed Jan 2003).

[84] Spivey, J. M., *The Z Notation,* A Reference Manual, Prentice Hall, London, 2nd edition, 1992.

[85] STANDISH GROUP, *Chaos Report*, http://www.projectsmart.co.uk/docs/chaos_report.pdf, Last accessed on 19 May 2006.

[86] Stapleton, L., Smith, D. and Murphy, F., *Systems Engineering Methodologies, tacit knowledge and communities of practice*, pp 159 - 180, Springer Verlag, 2004.

[87] Sukhoo, A., Barnard, A., Eloff, M. M., Van der Poll, J. A. and Motah, M., *Accomodating Soft Skills in Software Project Management*, Issues in Informing Science and Information Technology, Vol 2, pp 691 - 703, 2005.

[88] The Concise Oxford Dictionary (5th edition), Oxford University Press, 1964.

[89] Van der Poll, J. A. and Kotze, P., Weick, K., *Enhancing the Established Strategy for Constructing a Z Specification*, South African Computer Journal (SACJ) Number 35, pp 118 – 131, Desember 2005.

[90] Weick, K., *Sense-making in organizations*, CA, 1995.

[91] Wilcox, K., *Ethnography as a Methodology and Its Application to the Study of Schooling a review*, Holt, Rinehart & Winston, pp 456 – 479, 1982.

[92] Wilson, D., Rauch, T., Paige, J., *Prototyping and the software development life cycle*, http://www.firelily.com/opinions/cycle.html, (Accessed Nov 2005).

[93] Wood, J. and Silver, D., *Joint Application Development*, 2nd edition, John Wiley & Sons, Inc, 1995.

[94] Woodcock, J. and Davies, J., *Using Z: Specification, Refinement, and Proof*, Prentice Hall, London, 1996.