Interdisciplinary Journal of Information, Knowledge, and Management Volume 2, 2007

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

Editor: Kathy Lynch

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, 1990, 1995), 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) (Hughes & Cotterell, 2006; Wood & Silver, 1995), 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 or more individuals may 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 (Booch, Jacobson & Rumbaugh, 1999; Jalloul, 2004) 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.'

Figure 2 is an example of a UCM involving a single actor and three use cases:

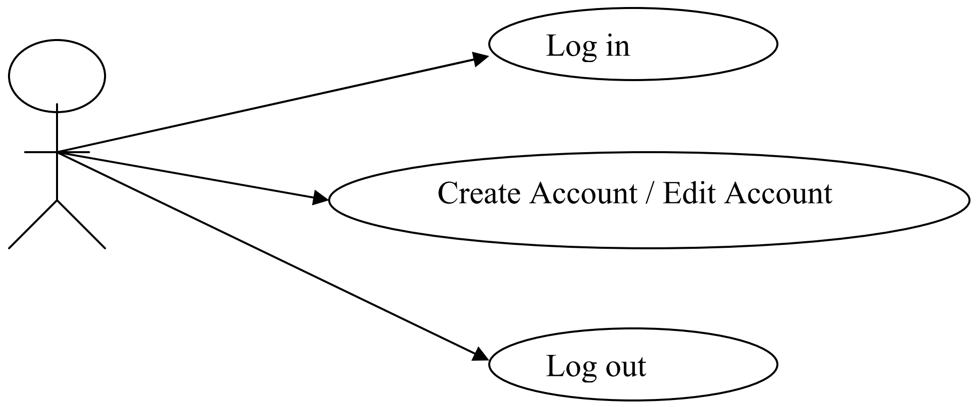
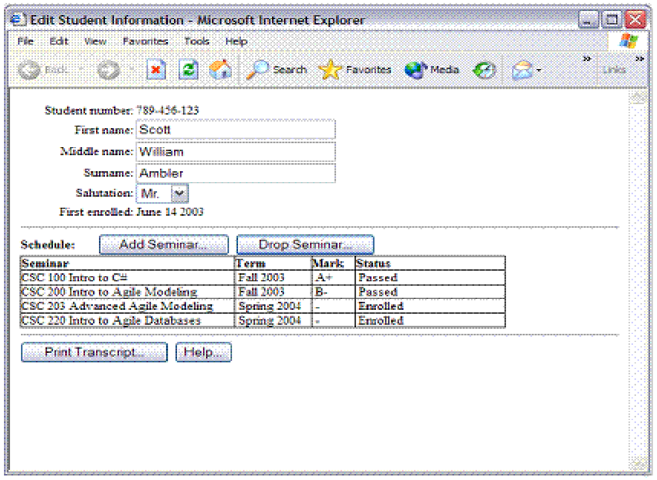


Figure 2: A Simple UCM

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.


Edit Student Information - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Links
Add Seminar...
Drop Seminar...
Print Transcript...
Help...

Student number: 789-456-123
First name: Scott
Middle name: William
Surname: Ambler
Salutation: Mr.
First enrolled: June 14 2003

Schedule:

Seminar	Term	Mark	Status
CSC 100 Intro to C#	Fall 2003	A+	Passed
CSC 200 Intro to Agile Modeling	Fall 2003	B-	Passed
CSC 203 Advanced Agile Modeling	Spring 2004	-	Enrolled
CSC 220 Intro to Agile Databases	Spring 2004	-	Enrolled

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 with 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:

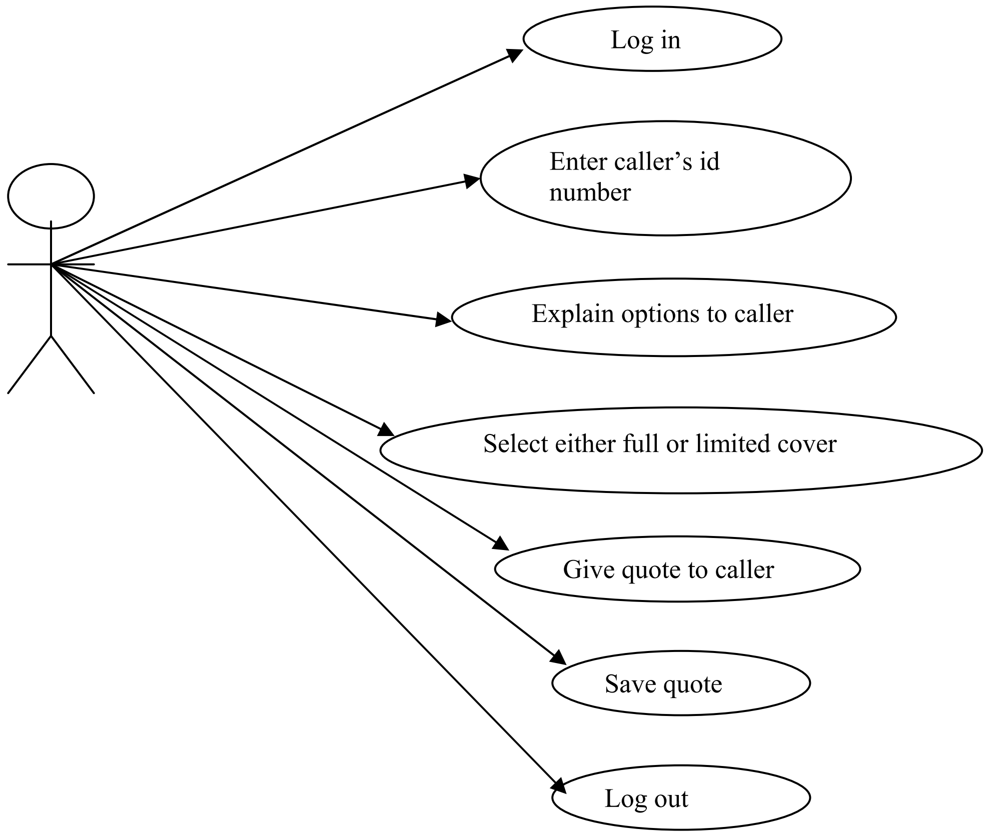


Figure 4: UCM of vehicle insurance

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 need 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 makes 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 designed the system, incorporating all the requirements of the system elicited during the JAD sessions and UML use cases. Following the design phase was 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 are 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 two people in the development team (D1and D2) and one client. Initially, then, the situation is often as depicted in Figure 5.

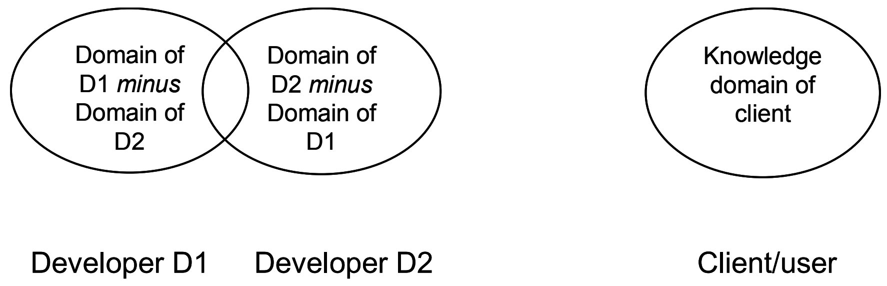

Domain of D1 *minus* Domain of D2
Domain of D2 *minus* Domain of D1
Knowledge domain of client
Developer D1
Developer D2
Client/user

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.

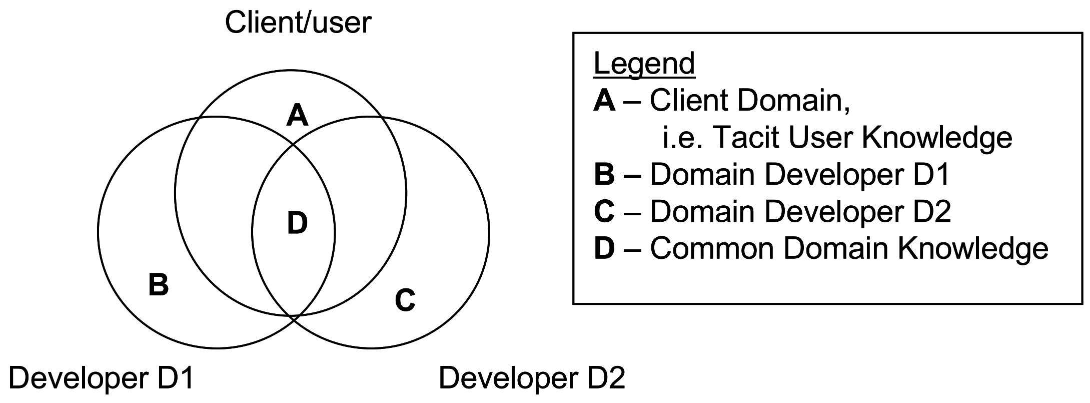

Client/user
A
D
B
C
Developer D1
Developer D2
Legend
A – Client Domain, i.e. Tacit User Knowledge
B – Domain Developer D1
C – Domain Developer D2
D – Common Domain Knowledge

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.*)
    $D := \bigcup_{i:=1}^{m} D_i$;   $U := \bigcup_{j:=1}^{n} U_j$
    (* 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.

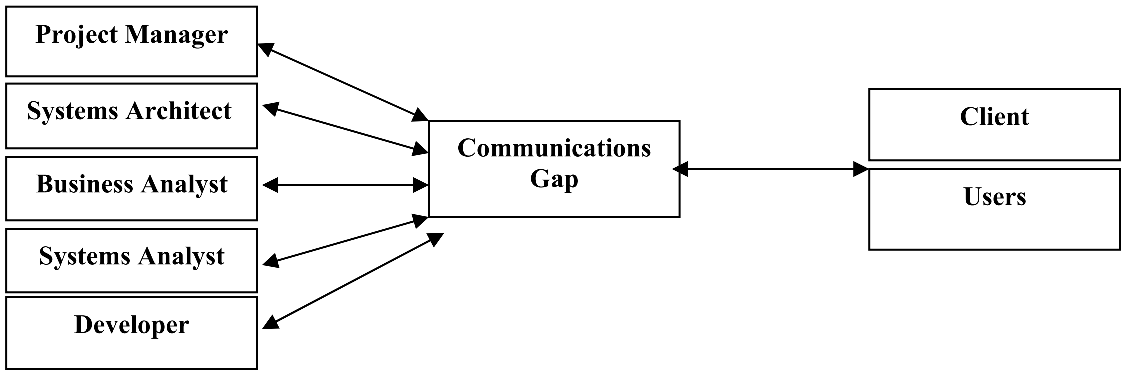


Figure 7: Communication gap between IT specialists and users

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, adding additional resources to a project 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 $C = D \cap U$.

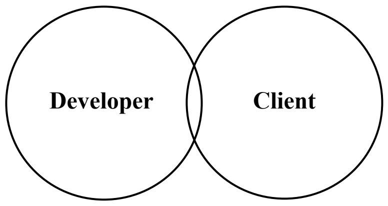


Figure 8a

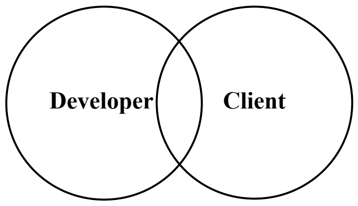


Figure 8b

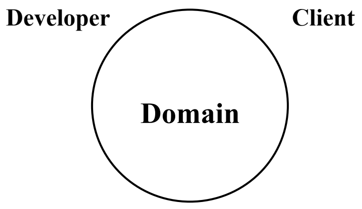


Figure 8c

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 (Fagan, 1976; Potter, Sinclair, & Till, 1996) 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 ed.). Cambridge University Press.

Bahrami, A. (1999). *Object-oriented systems development using the Unified-Modelling Language*. McGraw-Hill.

Beyer, H., & Holtzblatt, K. (1998). *Contextual design: Defining customer-centered systems*. San Francisco, CA: Morgan Kaufmann Publishers.

Blandford, A., & Rugg, G. (2002). A case study on integrating contextual information with analytical usability evaluation. *International Journal of 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., & 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, 20*(1), 10-19.

Cohene, T., & Easterbrook, S. (2005). Contextual risk analysis for interview design, *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), 209-218.

Cox, B. (1995, November). No silver bullet revisited. *American Programmer Journal,* 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*, *15*(3), 182-211.

Friedrich, W. R. (2005). *Prototyping techniques and software development models*. Unpublished manuscript. UNISA.

Gordon, V. S., & Bieman, J. M. (1995). Rapid prototyping lessons learned. *IEEE Software*, *12*(January), 85-95.

Hansson, S. O. (1999). *A textbook of belief dynamics: Theory change and database updating*. Springer.

Hickey, M., & Davis, A. (2002). Requirements elicitation and elicitation technique selection: A model for two knowledge intensive software development processes. *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. *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE 96)*, pp. 4 - 11.

Hughes, B., & Cotterell, M. (2006). *Software project management* (4th ed.). McGraw-Hill.

ITWEB. (2004). *Root of IT failure exposed.* Retrieved November 11, 2004, from http://www.itweb.co.za/sections/business/2004/0411111148.asp?O=S&cirestriction=brian%20whittaker

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., & Till, D. (1996). *An introduction to formal specification and Z* (2nd ed.). Prentice Hall.

Pressman, R. S. (2005). *Software engineering – A practitioner's approach* (6th ed.). McGraw-Hill.

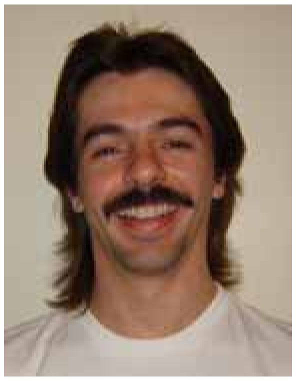
Schach S. R. (2002). *Object-oriented and classical software engineering* (5th ed.). McGraw-Hill.

Snyder, C. (2001). *Paper prototyping*. Retrieved January 12, 2003, from http://www-106.ibm.com/developerworks/library/us-paper

Standish Group. (1995). *Chaos report*. Retrieved May 19, 2006, from http://www.projectsmart.co.uk/docs/chaos_report.pdf

Wood J., & Silver, D. (1995). *Joint application development* (2nd ed.). John Wiley & Sons.

Biographies



Wernher R. Friedrich holds a Bcomm Information Systems degree, Honors Bsc Computer Science degree and is completing his Msc Computer Science degree. In his current role he is a director responsible for software development and works on national government level together with the minister of communications. His main focus area is on how software needs to simulate environments and how users form an integral part of how requirements are correctly captured to be able to simulate such environments. Looking onwards he passionately wants to do his Phd in computer science.

John A. van der Poll holds a PhD in Computer Science and is a full professor in the School of Computing at the University of South Africa (UNISA). He teaches an undergraduate course in operating systems as well as a postgraduate course in formal program verification. His research interests are in formal specification techniques and automated reasoning.