
**UNLOCKING TACIT KNOWLEDGE
BY CONCLUSION AND JUSTIFICATION
NEXT GENERATION REQUIREMENTS CAPTURE**

**BY PHILIP RICE, DR TOBY SUCHAROV AND DR LEON SUCHAROV OF ERUDINE
A WHITE PAPER FROM ERUDINE**

3	ABSTRACT	
3-4	INTRODUCTION	
4-5	THE ISSUES OF REQUIREMENTS CAPTURE AND MANAGEMENT TACIT KNOWLEDGE / MENTAL MODELS / THE HEART OF THE PROBLEM?	
5-6	WHAT ARE THE ALTERNATIVES?	
6-7	CAN WE NOW DEFINE THE PROBLEM? ARE WE MISSING THE REAL ISSUE?	
7-9	HOW DO WE LEARN AND GAIN KNOWLEDGE LEARNING BY DOING / THE LEARNING CYCLE / CONCLUSIONS DRAWN	
9-11	THERE IS A NEW WAY TO CAPTURE KNOWLEDGE THE IDEAL SITUATION / CONCLUSION AND JUSTIFICATION – THE HEART OF ERUDINE / BEHAVIOUR CREATION / SYSTEM VALIDATION / LEGACY ELIMINATION / REQUIREMENTS CAPTURE AND TESTING	
11	CONCLUSION	

ABSTRACT

The problems and issues around requirements capture and management are well documented within the IT industry. It is the defects in the requirements definition process that very often lead to failure in the successful implementation of major IT projects. We would all contend that truly successful development can only take place if we can elicit all the relevant ‘Tacit Knowledge’ that is contained within the experience of the domain expert and then transfer this effectively into working code.

This paper examines the key issues around the industry’s failure to create this effective transfer through traditional means. It also looks at some of the alternatives to the traditional methods of knowledge capture that have been put forward– are they truly effective and will they significantly reduce failure rate?

From this examination of capture techniques we pose the questions – Is the goal of specifying consistent and complete requirements futile? Should we not accept the reality that we will never have satisfactory systems and software to enable efficient and effective requirements capture? In this paper we look at what is possible but significantly we challenge what is said by many to be impossible.

Finally we come to the heart of the paper and consider some basic and fundamental realities and flaws in the way we seek solutions to the critical issues of requirements capture. The paper places before the industry, the Erudine Behaviour Engine, a unique and revolutionary system that harnesses tacit knowledge, in the marketing sense of the word, and enables a true and full capture of requirements and the turning of these requirements into rules and code that allow the developer to create dependable software.

INTRODUCTION

The IT Industry is well aware that there are major problems in the management and development of major IT projects. Problems that are inherent in the very nature of software development processes. These problems result in very high failure rates in large projects.

The facts provided by the US National institute for Science and Technology (NIST) and the Standish Group highlight without any doubt that failure costs industry and the economy billions of dollars each year. The NIST states that: The annual cost to the US economy in terms of overruns and the accumulated cost of business processes being unavailable, due to software failure, is around \$60 billion.

The Standish Group in their well-cited Chaos Report stated that: 31.1% of projects will be cancelled before they ever get completed. Further results indicate 52.7% of projects will cost over 189% of their original estimates. Only 16.2% for software projects that are completed on-time and on-budget, but this drops to 9% for large companies. Even completed projects by the largest American companies have only approximately 42% of the originally-proposed features and functions.

The KPMG Canada Survey found: Over 61 % of the projects that were analyzed were deemed to have failed by the respondents. More than three quarters blew their schedules by 30% or more; more than half exceeded their budgets by a substantial margin.

The OASIG Study stated: The IT project success rate quoted revolves around 20-30% based on its most optimistic interviews. Bottom line, at best, 7 out of 10 IT projects “fail” in some respect.

Other surveys have shown many IT managers believe that there are more failures now than there were 5 years ago. Are these statements highlighting industry incompetence, or just a true reflection of the present day reality of project development?

We are all aware of the documented facts from Standish, so the word failure is specifically relevant in the context given. Incompetence can only be a relevant term if we have at our disposal the necessary tools to create dependable software but we then go ahead and apply them in an ineffectual way. Quite clearly it is not incompetence but a more fundamental issue of not having the necessary technical tools.

If we are to accept the issues above then what are the root causes of failure within the IT industry? We believe that the cause of these issues are the twin problems in the delivery of consistently reliable software:

- Requirements capture and management.
- Software dependability.

The twin problems are of course fundamentally linked as they are based on the establishment of a truly effective connection between the IT developer and the domain expert in the capture and management of requirements and turning these into usable dependable code. Let us consider these issues further.

THE ISSUES OF REQUIREMENTS CAPTURE AND MANAGEMENT

TACIT KNOWLEDGE

Effective requirements capture and management have at their very core the need to successfully elicit tacit knowledge from the domain expert. Traditionally the IT developer will go through a series of refinements with the domain expert to deal with the inevitable exceptions that occur as work progresses, the code is developed, and further tacit knowledge is gained. The difficulties in achieving a successful requirements capture are evident in the very definition of tacit knowledge.

“Tacit knowledge can be defined as knowledge that is not made explicit because it is highly personal, not easily visible or expressible, and usually requires joint or shared activities to transmit it.”

Polanyi (1966) suggested that knowledge may be split along two dimensions and these enable us to consider knowledge as either explicit or tacit. Tacit knowledge describes knowledge which can not be transmitted in words such as the definition of the colour blue. This term has been adopted as a marketing term to mean knowledge too complex to be transmitted in its entirety, such as the description of how to drive a car, this is the context used in this paper.

There is no real difficulty with explicit knowledge as it can be satisfactorily captured and articulated by the holder and effectively documented. Of course explicit knowledge can be turned into code.

But how do we deal with the knowledge that is so complex that the expert can not fully articulate it? This is knowledge gained by insight, subjective judgements, intuition, and life experiences. Knowledge in its simplest and most basic form that we could say is acquired from such things as ‘learning by doing’.

The reasons that the capture and management of requirements are inherently difficult is that the more complex the project, the more tacit knowledge the users possess, and the harder it is to make it explicit. With traditional software projects the requirements capture is performed by trained business analysts who have a wealth of techniques available to them including structured and unstructured interviews, role play, check lists, prototypes. If tacit knowledge is such a huge issue then what does this mean to the IT developer?

MENTAL MODELS

During a project, the IT developer will build up a mental model of the software they are constructing, generally the project will progress quickly as long as the developer is able to visualise the entirety of the project model within their head.

However, as the complexity of the code increases, good IT skills become increasingly important. The link between the domain expert and the IT developer does not just involve the extraction of tacit knowledge but he or she needs to begin to deal with the exceptions. These are the exceptions that result from the inability to elicit all necessary tacit knowledge at the start of the project.

We have a further critical dimension when we reach the point where the developed code exceeds the developer's ability to store the coded model in his or her head. Naturally progress slows dramatically at this point.

If we also consider that developer-changes are also a fact of life – then the code becomes very difficult to alter as a new developer, introduced to the project, will not possess the mental model of the software. It is often necessary for the new developer to rewrite parts, or all, of the code in order to help them build up the mental model of the particular code.

The IT industry is acutely aware of this problem and considerable efforts are undertaken to generate documentation that will cope with system architecture. Even without the inevitable developer changes – there is a need to involve numerous developers within critical projects so that the loss of any one developer will not cripple the project. It is essential that we now require each developer to build their own mental models of the software and maintain similar models between developers.

THE HEART OF THE PROBLEM?

Are we now at the heart of the problem? Tacit knowledge is highly personal and the domain expert has difficulty in expressing their tacit knowledge as they do not consciously know all the rules that they apply to a situation. What is written into the software are the commonly occurring rules, logical processes and inevitabilities.

Hidden in the expert's tacit knowledge are the multitudes of subtle influences and exceptions that need to be included for the software to be effective. Both the capturing of tacit knowledge and the mental modelling necessary are significant issues that contribute to code clash and subsequent software and project failure. What then are the alternatives?

WHAT ARE THE ALTERNATIVES?

A simple Google search will highlight a huge number of companies that suggest there are real alternatives to the traditional methods of requirements capture and that these alternatives are a panacea, or indeed 'the' panacea to achieving success. They are very clear that this is the way to avoid project failure. Their companies abound with detail on what the industry is doing wrong and how this can be remedied. Deeper investigation shows that they do provide a clear insight into the many and varied ways used to get the appropriate end result.

The champions of these alternatives invariably suggest the full involvement of project stakeholders. Naturally we want and need this type of involvement but is this really the way to overcome the twin issues mentioned earlier?

Paul Grunbacher and Robert O Briggs in their paper to the IEEE's 2001 International Conference on System Sciences suggest that improvements to the requirements definition process could substantially reduce many of the risks we feel are inherent within major IT projects. They state that recent advances in requirements engineering research have gone part way to addressing many of the problems highlighted in the Standish Group study. They say that:

- there has been a move away from focusing on requirements modelling without understanding the organizational and social context.
- there is an increased emphasis on approaches, modelling the system environment together with the system, by focusing on stakeholder goals, scenarios or usage etc.
- the goal of specifying consistent and complete requirements is seen as futile, and there is an increased focus on negotiation techniques for analysing and resolving conflicting requirements.

Does Grunbacher et al, in fact highlight a dichotomy here? We are exhorted to spend a significant amount of time and resources to seek a solution to what is described as a failure to deliver successful projects but, the reality, they suggest, is that failure is an inevitable consequence of specifying consistent and complete requirements. This is because the transfer of domain-expert tacit knowledge to explicit knowledge will never have the necessary 'translation' conduit.

So can we say that the alternatives to traditional methods are a full involvement with Stakeholders as described by Grunbacher and Briggs and others? This broad and deep collaboration will indeed create an environment that will probably elicit more tacit knowledge than through traditional means and this must ultimately mean that there will be a reduction in the need for the handling of exceptions later.

However do we want to expend stakeholder time and resources at this stage of development and in this way? Are there not more productive uses of stakeholder time? Are we not just adding another layer of complexity? So are there any real alternatives?

CAN WE NOW DEFINE THE PROBLEM?

Let us consider the key issues at this point:

- **We can employ many sophisticated techniques for requirements capture but what is suggested is that there is no real effective way to capture tacit knowledge, for by its very nature it is incredibly difficult to elicit and interpret.**
- **People such as Grunbacher et al in fact state that it is futile to expect that we will ever fully capture the knowledge. Even with all the sophisticated means of knowledge capture postulated by the industry.**
- **A key issue is the ability of the developer to hold mental models and manage conflicts, clashes and the inherent constraints that continually conflict with the development of dependable software.**

ARE WE MISSING THE REAL ISSUE?

Perhaps we should accept that the work of Grunbacher and Briggs (2001) as a benchmark and that the only logical conclusion is that failure is inevitable for complete requirements capture. But are we missing the real issue here?

The focus is, in fact, on how we effectively capture requirements within the constraints and complexities of tacit knowledge. Should we not go back to basics and focus on the way people develop their tacit knowledge – what is their learning process? – how do people acquire knowledge? If we are able to examine the way people learn and acquire their tacit knowledge then surely we are at the heart of the problem and can begin to look at real alternatives

HOW DO WE LEARN AND GAIN KNOWLEDGE

If we go back to the most basic example – the way children learn – there are some key pointers to the questions raised previously. There are volumes of text and research on the subject from the internationally respected work by both Piaget (1896 -1980) through his Levels of Learning to Kolb's experiential learning cycle (Kolb and Fry 1975) and many others.

LEARNING BY DOING

Parents teach their children to learn by doing from their first steps to riding a bike. They act as the experienced mentor to check and correct where a child is going wrong. The child builds on these experiences and begins to experiment and pushes at the boundaries. Their own store of tacit knowledge begins to expand and becomes personal and unique to them.

The Chartered Institute of Personnel and Development and the Investors in People initiative have recognised that if we give a person a well written specification of a job it will still take time for that person to become competent and then expert at the job. In order to acquire the tacit knowledge needed to perform the job well they need to have shared experiences with an expert at the job, a supervisor and/or mentor. One of the best ways to learn is to do the job under guidance and have their mistakes corrected as they go along.

It would seem that whatever educational sophistications we employ what we are in fact focusing on is the way people learn by doing. It would seem appropriate at this stage to examine some of the research that is available on this subject as a means of validating the view that 'Learning by Doing' will provide a better model for transfer of knowledge.

Research in cognitive science has shown that the most effective way to teach new skills is to put learners in the kinds of situations in which they need to use those skills, and to provide expert practitioner-mentors who are on hand to help the learners when they need it.

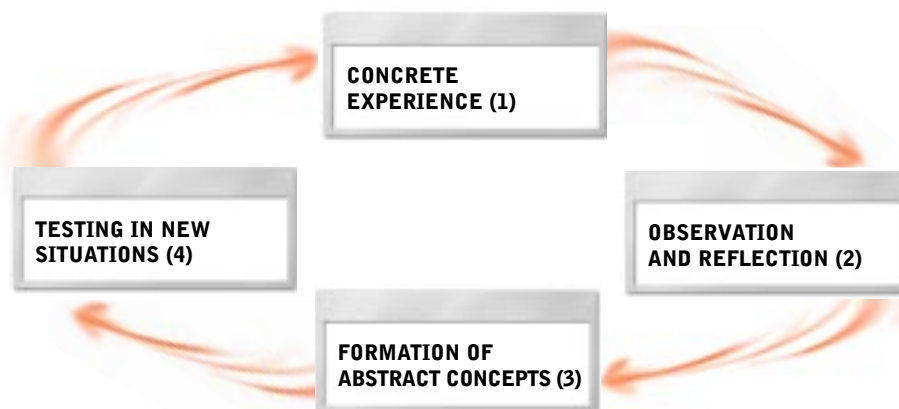
The process helps learners come to understand when, why, and how they should employ target skills on a job or process. They receive just-in-time key lessons, meaning that they learn the information when it will make the most sense to them.

Carnegie Mellon University (CMU) has undertaken significant research in the way people learn by doing. Significantly for the subject we are now considering, CMU is expanding its learn-by-doing master's degree programs in the computer sciences by adding a two-year track for college graduates with no prior computer science experience who wish to pursue a master's degree. This says quite loudly that in their view that the best way that tacit knowledge is acquired is through learning by doing.

The university's philosophy is based on the work of the world renowned education innovator Roger Schank – professor in computer sciences at CMU and John Evans Professor Emeritus in computer science at Northwestern University. Instead of attending classes and taking tests, learn-by-doing is a basic aspect of the programme. Students work on ongoing projects and are mentored by the faculty. CMU said its approach simulates the environment that students will enter after graduation, preparing them for the real world.

THE LEARNING CYCLE

Mezirow and Freire suggested that learning was in fact a cycle that begins with experience, continues with reflection and later leads to action. David A. Kolb (with Roger Fry) further substantiated this idea by putting forward his famous learning cycle model. This model contains four elements: concrete experience, observation and reflection, the formation of abstract concepts and testing in new situations. This was represented in an experiential learning circle.



Kolb and Fry (1975) argue that the learning cycle can begin at any one of the four stages – and that the experiential circle should really be approached as a continuous spiral. We can consider that the learning process begins quite often with a person carrying out a particular action and then seeing the effect of the action in the determined situation.

Kolb notes the importance of understanding these effects in the particular instance so that if the same action was taken in the same circumstances it would be possible to anticipate what would follow from the action. Following this pattern the third step in this pattern would be an understanding of the general principle under which the particular instance falls.

CONCLUSIONS DRAWN

If we have the support for learning by doing by such notable figures as Piaget and Kolb whose work has been a foundation of learning theory for many years. If we also accept that there is massive support from such people as Roger Schank – who has committed the resources of a key university to the concept of learning by doing– then surely it would be folly to ignore their work.

Let us be reminded why we took this route through learning. If it is difficult to deal effectively with the complexities of tacit knowledge and the assertion that we will never be fully able to accurately capture all requirements, then we must go back to the very basis of 'how do people learn' and in that understanding we truly deal with the way tacit knowledge is learned, or gained.

Let us also now look afresh at the IT development process and apply our learning by doing concept. One alternative approach to traditional requirements capture would have the expert mentor the developer and enhance the way he understands the task. However this is time consuming for the developer and still requires him to go away on his own and try to translate this experience for the computer..

The ideal, but radical process would be one in which the expert interfaces directly with the computer so that it learns by doing first hand from the expert.

It is this very concept that is the revolutionary idea that will solve the problem of requirements capture. We need to create the process of allowing the computer to acquire knowledge the way humans do. It is with this in mind and with the considerable weight of support by the many researchers, writers and activists that Erudine was born as a concept and developed into a major product that has been used within the complex theatre of major IT projects.

We also put forward as further support to the argument that the definition of tacit knowledge by the Colorado enTWine project strikes directly at the heart of this revolution and supports the proposition of Erudine.

“Unspoken knowledge that only surfaces in the context of doing something or when the knower is somehow reminded of it. For example, the knowledge we use to do everyday tasks is tacit. It is taken for granted and rarely discussed. Much of an expert’s knowledge is tacit and cannot be articulated in abstract contexts such as interviews. For this reason, so called knowledge acquisition schemes cannot hope to capture all of what an expert knows. Systems for experts must therefore be open-ended so experts can add their knowledge as it arises in the contexts of doing work”.

So we are now considering the scenario where the main ingredients have been eliminated and a revolution has been achieved.

THERE IS A NEW WAY TO CAPTURE KNOWLEDGE

THE IDEAL SITUATION

In an ideal world the expert – the user – will work directly with the computer and its software, teaching the computer to learn the way a human learns. The software learns by ‘doing’ the task with the expert, observing the experts actions in increasingly complex situations, creating rules added quickly and simply whenever the expert teaches the system how to handle a new situation.

This way of development will clearly revolutionise the way we work on major projects and most significantly will address the major issues voiced by The Standish Group and the NIST. The Erudine Behaviour Engine allows you to work this way now.

The Erudine Behaviour Engine provides a direct interface between the expert (the user) and the developing project code. A link where the domain expert can justify the conclusions made and from this allow the computer to interpret the justifications so that code clashes are avoided and mental modelling is not limited by human ability to house complex models in the mind. A link that uniquely allows the computer to learn by doing as it interfaces with the expert and learns to elicit ongoing tacit knowledge.

Erudine is based on the twin premise of Conclusion and Justification.

CONCLUSION AND JUSTIFICATION – THE HEART OF ERUDINE

We all recognise the widely accepted observation that while the expert cannot explain the rules that they use in advance, he or she can always defend their conclusions when presented with a situation. This observation lies at the very heart of the Erudine Behaviour Engine

The Erudine Behaviour Engine lets you start with no rules at all. As the user (probably not an IT developer) performs a task, i.e. makes a conclusion the software asks them to justify what they have done. This justification can be made in a variety of ways, using graphical user interfaces or plain text. The uniqueness of the Erudine Behaviour Engine, in the light of what we have been considering, is that it actually automates the task of turning conclusions and justifications into rules.

As it learns about the task, it starts to make suggestions that the user can accept or reject. The rejections require the user to justify their reasoning, thus the system learns what are the salient details that make the current situation different from its previous experience.

Just as in learning in humans, the system quickly learns to handle simple situations and only needs correcting as more complex or obscure scenarios are encountered. Thus the Erudine Behaviour Engine provides good results after a remarkably short time as it learns the rules that cover common situations first.

The user simply continues to work within their knowledge arena, on their usual tasks responding to the queries made by the software. As rules are introduced in this way then the critical problem of mental modelling that is necessary in the traditional way is automated.

BEHAVIOUR CREATION

The Erudine Behaviour Engine is responsible for creating new behaviour. It asks the expert directly why this situation is different to superficially similar situations. This enables the building of the test suite and ensures that the test suite is never broken. The test suite in fact consists of situations with expected answers. Every time a rule is added to the system, one or more tests are added to the test suite.

SYSTEM VALIDATION

The Erudine Behaviour Engine performs system validation to ensure that it will never enter an illegal state. When the user enters new behaviour into the system the Erudine Behaviour Engine performs a suite of checks to make sure that the added behaviour will not break old behaviour. These checks are auto generated each time any behaviour is added to the system, the user does not have to be concerned with how or when tests are run, the entire process is a core part of the Erudine Behaviour Engine's operation.

The user is also able to request a full audit trail of any package or transaction passing through the Erudine Behaviour Engine. Erudine will remember the state of that package at every point through the system and the user is able to run these audits at any time they wish during development or deployment.

System validation is designed to stop the user from placing the project into an illegal state during project development or code maintenance and gives the user total confidence in the continual and correct operation of the project.

LEGACY ELIMINATION

Legacy Elimination is Erudine's patented process for rebuilding Legacy systems. The Legacy Elimination works by analysing the inputs and output from a system at data gateways and rebuilding from scratch the behaviour encapsulated between these two data gateways. As the user builds up the behaviour the Erudine Behaviour Engine will become more proficient at mirroring the legacy system, Erudine will only query the user about cases where Erudine's behaviour differs from the legacy systems.

This powerful tool can be run in parallel with the legacy system, and when confidence in the system is achieved it can be deployed live replacing the Legacy system. The deployed Erudine Behaviour Engine system will be easier to maintain and a requirements document of its full behaviour will be available to the user, if the user wishes to capture the requirements throughout the Legacy Elimination process.

REQUIREMENTS CAPTURE AND TESTING

The Erudine Behaviour Engine allows users to manage their requirements documents with the application itself. Users are able to read their documents into the Erudine Behaviour Engine and mark off requirements as functionality is added to the project. Also the user is able to insert new requirements into the system as behaviour is added.

The Erudine Behaviour Engine encourages users to make sure that all added system behaviour is marked off against a requirement. This will mean that the requirements document will accurately match the delivered project, capturing both the tacit and explicit knowledge of the system.

CONCLUSION

This paper examines the main problems for the IT industry in the development of major IT projects. We detail the findings of Grunbacher who highlights the complexities of acquiring tacit knowledge and states that while there are many complex techniques in use within the IT industry for acquiring tacit knowledge they are still just a sophistication of the same flawed process.

We describe the Erudine Behaviour Engine, a new system for requirements capture that uses as a basis the same learning patterns under which humans learn. The Erudine Behaviour Engine allows the acquisition of tacit knowledge by using a learn-by-doing work flow. Thus it is both a new process, and the new technology that implements it.

Critically the Erudine Behaviour Engine does not use the traditional model whereby the expert has to describe the requirements to the developer (containing both tacit and explicit knowledge) and then the developer has to translate the expert's requirements into a software package. The Erudine Behaviour Engine uses an intuitive justification and conclusion language enabling the expert to enter the requirements directly into the software package. It allows requirements to be refined seamlessly at any point during development. This is one of the critical problems in Grunbacher's paper detailing successful requirements capture techniques within complex IT projects.

WWW.ERUDINE.COM

Tel: +44 (0) 8456 123 862

technical@erudine.com / www.erudine.com

Design: www.solutiongroup.co.uk