Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Software design through artificial expertise

P. Garratt, D. Karami
Department of Electronic and Computer Science, University of Southampton, Hampshire S09 5NH, UK

ABSTRACT

Software engineering activities involve tools, methods, programming languages, and human intelligence. In the past, software engineers were concerned with isolated criteria such as functionality, and scheduling in different periods of time. In the 1980's, when Information Technology was widely available to individuals due to the breakthroughs in computer hardware technology, they were mainly concerned with *the cost attribute* of software development and software quality was partially ignored. The term software quality has mistakenly been taken for *excellence* which is far from the quality of today's software systems. We believe the ultimate quality goal is user satisfaction. And in the 1990's software quality will be brought to the centre of the development process. Software quality is a multidimensional concept which covers:

- the entity of interest: the final deliverable items.
- the view point on the entity: the final customer's, the developing organisation's, and the project manager's viewpoints.
- the entity attribute contributing to the quality: the designer's viewpoint of the reliability of the deliverable items, and the manager's viewpoint of the elapsed time.

Meeting the quality objectives in the final product requires quality tools and methodologies contributing to the development process. This paper discusses the use of expert systems in the software development process. Their use contributes to higher quality in the entities mentioned above. Expert systems achieve part of their effectiveness by reducing the complexity of software development.

INTRODUCTION

Software Engineers must be primarily concerned with the optimisation of quality in their professional output. In a typical software development project it is never economic to maximise the quality of all deliverable items. Quality must instead be optimised in relation to other objectives. The reasons for this are well understood [25]. A software development project has a schedule and a budget and the attainment of high quality generally implies an extension of the time spent developing the software and an increase in the cost. The end result is always a trade-off between these three mutually conflicting objectives. Also, quality is a multidimensional concept. The professional output of a software engineer comprises many separate and diverse items, not only executable code but also documentation, manuals for the users and future maintainers, schedules, test plans, test harnesses and so on. Assessing the quality of all these diverse deliverable items is a many faceted task. It requires different quality metrics for

each item and recognition of the different factors affecting the quality of each item.

The viewpoint from which quality is assessed is never unique. The developing organisation has its view, as does the end user, the management of the end user, project manager, project worker and every professional involved in a software project. These differing perspectives complicate the assessment of quality. Major manufacturers such as ICL confront this problem with large scale company initiatives [16]. Some of the complication is shown in Table 1.

DELIVERABLE ITEM	PERSPECTIVE						
	U	UM	D	DM	SS	SA	DPF
FUNCTIONAL SPECIFICATION	3	3	3	3	2	2	0
PROJECT PLAN	2	2	3	3	2	2	2
QUALITY MANAGEMENT	1	1	3	3	0	2	0
ACCEPTANCE TEST SPEC.	1	3	2	2	1	1	0
SYSTEM MODELS	1	1	3	2	0	0	0
CODED MODULES	0	1	3	2	0	0	0
TESTED MODULES	0	1	3	2	0	0	0
INTEGRATED SUB SYSTEMS	0	1	3	2	0	0	0
INTEGRATED SYSTEMS	1	1	3	2	0	0	0
TOTAL SYSTEM	2	2	3	3	2	1	0
USER DOCUMENTATION	3	2	0	1	3	1	0
USER TRAINING PLAN	3	3	1	2	1	1	1
SYSTEM CONVERSION PLAN	2	3	2	2	1	0	2
RUNNING SYSTEM	3	3	1	2	3	2	1
INSTALLED SYSTEM	3	3	0	2	2	2	0

U: USER
UM: USER MANAGEMENT
D: DEVELOPER
DM: DEVELOPER MANAGEMENT
SS: SUPPLIER SALES
SA: SUPPLIER ACCOUNTANTS
DPE: DEVELOPER PERSONNEL FUNCTION
0: QUALITY IRRELEVANT
1: QUALITY LESS RELEVANT
2: QUALITY RELEVANT
3: QUALITY MORE RELEVANT

Table 1: Quality optimisation in software engineering

The trend in recent years has been to attach more and more importance to the quality view as perceived by the end user since information technology has become

used more and more by non-professionals with the growth in personal computing and the spread of user friendly Human/Computer Interfaces [17]. However, the quality views of other people involved in the system development and the end product should never be discounted. In particular, safety critical software has a special quality requirement regardless of the views of the end users [21].

Many software metrics have been developed whose objectives are in part to quantify the quality attributes and entities concerned in a software project [22]. Some metrics are appropriate to some deliverable items, none is appropriate to all. Some metrics are relevant to end users, some relevant to the developers, again no metric is relevant from all quality perspectives. A significant generalisation regarding metrics could be that the easier the metric is to measure the less use it is in practice. On a typical software project the software engineers will choose and use those metrics which are practical to gather and which give a good return in usefulness on the effort invested on gathering. For instance many count the number of bugs found during testing. When the rate falls to an acceptable level this metric has served its main purpose.

The factors affecting the professional activities of a software engineer and thereby influencing the quality of the deliverable items that he or she produces have not been thoroughly explored. The effects of, for instance, formal methods, programming support environments, software workbenches, intelligent design aids and other features in the development environment are under continual scrutiny in the industrial and academic world. However, most of the results of these investigations are empirical since the conducting of controlled experiments in software engineering is difficult or impossible [19]. The implication is that the true effects on metrics relevant to the deliverable items of features in the software development environment has rarely, if ever, been scientifically proven. Despite this, many mathematical models have been developed which attempt to predict cost, time scales and quality for new projects based on past similar projects, the nature of the software to be developed and the nature of the development environment [15]. These models are widely and successfully used in industry. But their results are interpreted by the human managers of the projects then adapted and modified before they are applied to the management of the project [23]. In general it seems to be true that the intervention of a human expert in applying and interpreting quality predictors is essential in practice. We believe that no single existing mechanism for quality prediction and management is broad enough in scope to cover the multidimensional nature of the concept of quality in the deliverable items of a software engineering project nor broad enough in scope to encompass the diverse viewpoints of quality of all the people involved in developing, maintaining, using and managing an information technology system. The key to successful optimisation of quality may be in heuristic decisions on software design, re-use and risk. We feel that there is a role for an expert system in assisting decision makers in software development and we have embarked on the construction of a prototype for assisting in quality optimisation.

Figure 1 illustrates the relationship between our quality optimiser and other expert systems which have been developed in software engineering. A quality optimiser can be regarded as a meta-assistant to help the manager choose the development environment, the management strategy and to assess the likely effect

of these factors on the qualities of the deliverable items that will emerge during the course of the software project.

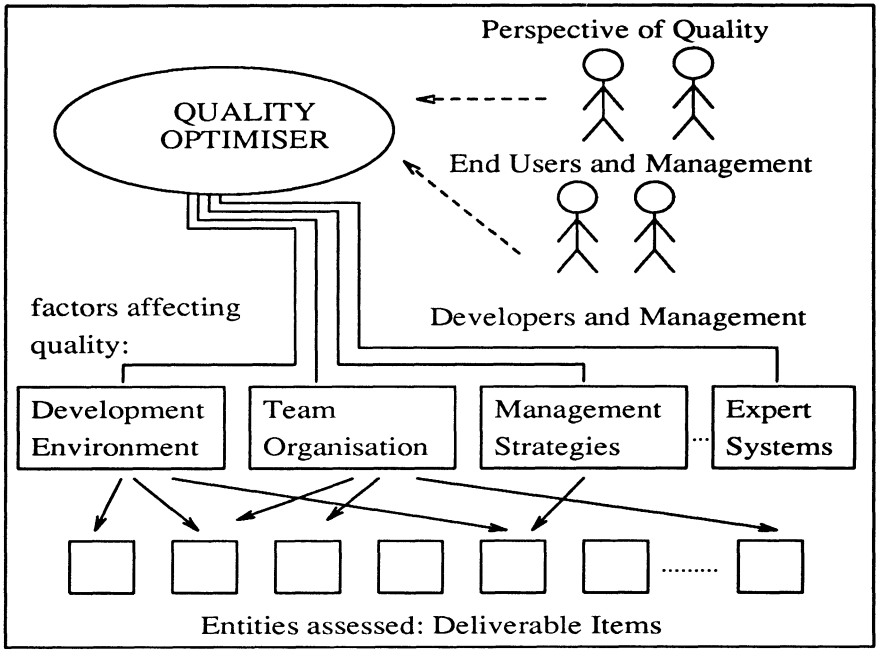

Figure 1: Quality optimisation in software engineering

In this paper we are particularly concerned with the application of knowledge based or expert systems in software engineering. We are investigating this avenue because we believe this research alternative is highly promising in addressing software issues [24].

THE SOFTWARE DEVELOPMENT ISSUES

Since the advent of transistors, computer technology has enormously influenced the quality of our life. As a result of technological improvements during the last twenty years, the affordability of computer hardware has dramatically increased. Consequently, computer-based solutions and computerization have found their way into many new industries and different aspects of our lives.

These increased demands for computer technology have been accompanied by corresponding demands for increased productivity, quality and reliability of software systems. Unfortunately the improved performance and increased cost-effectiveness of computer hardware components have not been parallelled by the same improvements in the software arena. We now find the industry in the situation where software production is the major cost component of a computerised solution to a problem. The software engineering of the right system itself is a major challenge. This state of software production has repeatedly been referred to as *the software crisis.*

Although today, software development is recognised as a legitimate engineering discipline whose methods have successfully been accepted in other areas of engineering, managers and developers alike recognise the need for a more disciplined approach to software development [1].

For hardware technology a well developed approach has been with us since the sixties. This approach includes scheduling, cost estimation and reliability considerations. A similar approach to software technology lacks 20 or 30 years of maturity [2]. This comes from the major difference between software and hardware, that is the difference between manufacturing or production and development.

The essential properties of software entities have contributed greatly in the current state of software crisis [3]. Software entities are complex in nature. To overcome complexity, mathematical modelling is often employed. This paradigm works well in situations where the complexities ignored by the model are not the essential properties. An example is the formulation of optimisation problems in production engineering as differential equations. The paradigm does not work where the complexities are the essence and can not be abstracted safely.

Although software entities are complex phenomena we must model, measure and manage software development if we want to keep the users satisfied. We must increase our understanding of the software product and the process through which it is designed, developed, and maintained.

Major breakthroughs in the past such as unified programming environments removed some of the accidental properties of software and eased the pressure of demands for higher productivity. Still, issues within the software development arena remain to be dealt with: the software quality issue and life cycle model and the associated problems.

The software life cycle

It is important to examine important software issues to study new alternatives for software development and supporting environments. Such investigation enables us to identify the potential improvements that directly address the problems and issues of software development.

The complexity of software projects has led the software industry to devise paradigms that view software development as a series of somewhat independent development phases. Such software life cycle models associate particular goals to each phase, and provide validation procedures as mechanisms to detect any errors before proceeding on to the later one. As additional checks, some models provide mechanisms to test against the products of the later phase to verify the products of the earlier phase, thus introducing feedback into the process. Even with these improvements, the life cycle model has several drawbacks which have been strongly criticised suggesting that the life cycle model is simply inappropriate [4].

Problems in the early phases The early phases of software development are often considered to be the most important phases since the cost of undetected errors is magnified with each successive phase. Unfortunately, system requirements can rarely be adequately stated in advance and are bound to change during the development phases. Dealing with such a moving target is completely beyond

the capability of the traditional life cycle activities.

Another problem is the lack of end-user involvement in these early phases of development. Consequently, the specifications may not be correct with respect to the needs of the user. This will increase the cost of product maintenance since many product faults are not detected until after the product is handed over to the user. The cost of modification in the maintenance phase could well be up to 65 times more than in the development phase [1]. Moreover, it is difficult for the end-user to appreciate the intermediate products of the early phases since there is nothing concrete and executable, to evaluate.

The lengthy life cycle In the traditional life cycle, no phases can begin until the preceding one has been completed. The sequential nature of the process causes the project to extend in time. Correspondingly, the final product takes a long time to appear. Even if the system requirements could be stated fully at the beginning of the project, they could not expected to remain unchanged throughout such long development period. This causes damage to the user's confidence. In some cases the final product may no longer satisfy the current user's needs.

The life cycle is lengthy because for a long time we believed that specifications are the only means to describe the system functionality without saying how that functionality is to be implemented. We believed that the implementation phase copes with the *how to* aspect of the development. In many cases the division between requirements and specifications are vague. The requirements normally identify the needs that have to be addressed by the system. A large portion of requirement definitions is the incomplete system specifications. The specifications could be viewed as the implementation of some higher level specifications, because there is a refinement relation between specification and implementation [5].

To manage the increasing complexity of software systems, several development models have been introduced during the past 35 years. The *code and fix* model used in the late fifties and early sixties, provided too much flexibility. The resulting systems were very difficult to test and maintain.

The waterfall model The *waterfall model* [6] provided more structure. The waterfall model is successful because it impose some discipline over the overall task of software development and is documentation-driven. The major disadvantage of this model is the lack of end user involvement in the specification phase.

The rapid prototyping model As its names suggests *rapid prototyping* deals with the lack of user involvement in the early phases of the life cycle by getting him involved as much as possible through interactions with the prototypes. The main idea is to catch specification errors and correct them before too much effort has been expended in implementing the erroneous specifications. Prototypes are discarded after they are evaluated. The overhead here is the cost of the throw away prototypes. There are alternatives which compensate for this drawback namely operational specifications, and automatic programming [5]. The problem with the operational specifications is that the program has to be specified in a somewhat restrictive formal language. The problem here becomes more clear as many real types of programmes can not be specified so easily, therefore the *rapidness* concept is lost in preparing the specifications.

The evolutionary model The *evolutionary model* [4] provided a multi-version product. The product operational growth is incremental to cope with the evolutionary nature of the requirements. The existence of this model acknowledges that fact that it is not possible to completely get the requirements of a complex system defined up front.

The spiral model The *spiral model* [7] was first introduced to explicitly encompass multiple cycles of planning and prototyping for risk management and put dynamism on top of the development process. Therefore, there was a paradigm shift from a document-oriented to risk-oriented development process. The X model, a deviant of the spiral model, was later introduce incorporating the concept of software reuse [8]. There have also been a few new paradigms that places emphasis on prototyping and re-use as a means of minimising risks to achieve higher productivity [11], [6], [12].

The mythology issues

Unlike human myths the software myths contribute the confusion and misunderstanding amongst the software engineer camps. They bear some elements of truth and appeared to be reasonable statements mainly gathered from past experience. They have produced confusion and misleading attitudes that have caused serious problems for managers and technical people alike. However, old habits die hard, therefore the software myths are still believed as we move towards the fifth decade of software. The following sections show the most common myths as well as the corresponding statements of reality [1].

The management's myths Software development managers are under the pressure to maintain the project within the budget, keep the time schedule from slipping, and improve quality. Therefore, relying on myths to avoid disappointment during the course of projects.

Why should we change our approach to software engineering? we are doing the same kind of programming as we did ten years ago. Although the application domain may be the same as what it was ten years ago (for many organisations it has changed substantially), the demand for greater productivity and quality and the importance of software for many strategic business objectives has increased dramatically.

Our staff has the best of state-of-art technology they can get for good software practise. It take more than the best mainframe or the best PC available to produce quality software. Software tools such as programs assisting programmers to create programs are much more important than the hardware needed for achieving productivity, quality, and reliability needed to meet the objectives of some strategic businesses.

If we are running short of time we always can add new programmers to catch up with the schedule. In the words of Brooks [3]: ' ... adding more people to a late running project can only makes it later.' This statement is reasonable because as new programmers are added the time wasted due to communication and learning amongst the staff reduces the time to be spent on the more productive development.

The customer's myths Customers do believe in myths as well as the managers and software practitioners. These misleading beliefs eventually leads to customer

false expectation of the software and ultimately to his dissatisfaction with the developer.

A general statement of the objectives is sufficient to begin writing programs. We can always fill in the details later. Poor program definition is the major cause of the failed software projects. Formal definition of the information domain, the exact functionality, the design constraints, interface, and the validation criteria are the essentials to start the project with. These characteristics are only determined when the true communication between the customer and the developer team has taken place.

Project requirement changes continually, but changes can be accommodated easily because software is flexible. It is true that software requirement change, but the impact of change varies with the time it is introduced. The modification introduced during the implementation leads to a sharp rise in the cost and can cause upheaval that requires additional resources and even dramatic changes in the design.

The practitioner's myths Myths beloved by software practitioners have been fostered by forty years of programming culture. Then programming was considered an art. Is it the case today?

The only deliverable for a successful project is the working program. A working program is only one part of a software configuration that includes all elements such as plans, requirements specification, design of the data structures, design of test specifications, and the working program. Documentation forms an important basis for a successful development and more importantly it provides guides for software maintenance task. Table 1 shows a minimum set of deliverable items.

Once we write the program and get it to do what it is meant to do, our job is completed. With computer software, the sooner you begin to write the code the longer it takes to get it done. The data obtained from industry indicates that 50 to 70 percent of all effort expended on the program will be expended after it has been handed over to the customer for the first time [1].

The quality issue

A poorly designed system although might be put together quickly so that it process an acceptance test correctly, has poor quality because it may cause extra repair cost later on. Thus improving the quality of the deliverable software is a major goal of research in software engineering.

Once in the eighties low-cost software applications became practical and widely implemented by the vendors, the importance of productivity in software development increased substantially. This informed a mass market of rather unsophisticated customers with its potential of large sales which increased the demand on quality. The cost, functionality, his satisfaction and the rate of software failure are important product quality from a user stand point.

From another stand point quality software has to be developed through a *quality-oriented development process.* Such development process views the final product passing through a series of stages. Each stage introduce an intermediate product with its own quality attribute which have to satisfied before passing on the intermediate product to the next stage. For example, the designer is the user of the requirements specification. The designers develop system architecture and

unit specifications. They ultimately produce the design document. The quality attributes of the design document are readability and completeness in meeting system requirements.

KNOWLEDGE-BASED SOFTWARE ENGINEERING

In the previous section, we discussed the main software development issues. The fundamental software challenge is now how to develop software faster, better and cheaper.

Most software problems come from the facts that the specification is not well understood by the programmers, the desired behaviour is not well understood by the requirement analyst or the user, and the programmers of one subsystem do not understand the effects their code may have on overall system performance. Besides, when the code is written a lot of knowledge is simply lost. That is the code is the concentrated version of decisions made from requirements down to design. Usually the knowledge regarding decision making and the reasoning behind them are not made explicit. Even if it is made available its volume makes it impossible to use in traditional software engineering practice. This makes software development an enormously knowledge-intensive task.

AI deals with how to extract useful information from a large bulk. From the handling of information standpoint, AI in general and knowledge-based systems in particular can improve software engineering not only by explicitly representing such a knowledge, but also by showing intelligent behaviour using that knowledge. A paradigm exploiting this knowledge-intensive nature of software development will be very appropriate to improve both the quality of software design and the flexibility and acceptability of final product.

One way to gain higher productivity and better quality is through changes in the educating process of software engineers in the universities [9] and through training human designers.

Alternatively, in an attempt to make the machine bear some of the burden of development task, one has to capture the expertise of human designers and make it available to the novice designers in the form of **artificial expertise** through knowledge-based or expert systems. Such systems emphasize the identification, encoding and automatic use of knowledge relevant to a task, and are built using techniques borrowed from AI and ES fields which especially address knowledge-intensive activities [10].

KBS and CASE software engineering CASE tools provides a step towards the automation of software development. The ultimate goal in CASE is integrated tools that support all phases of software process allowing information from one phase to be used in another. This information sharing typically involves the use of data bases to store and access the information from the various tools. The capability of CASE tools can be extended in two ways by the application of AI techniques:

The evolutionary approach At the simplest level the evolutionary approach involves replacing the data bases currently used in CASE tools with knowledge bases. Knowledge bases are the augmentation of data bases allow new facts to be derived from the combination of other facts. They also provide mechanisms such

as support for objects and feature inheritance. Therefore, knowledge bases can support reasoning rather than simply storing the facts. This improved reasoning ability therefore can be used to improve the performance and range of current CASE tools [13].

The revolutionary approach The application of AI techniques can be used to fundamentally change the notation of case tools. There are three points to consider in order to produce fully automated systems.

First, we can formalise the knowledge used in the development process. This formalisation means have a specification language, and an explicit set of operators. Refinements, can therefore be applied to the specification or the code at different stages of the development process.

Second, we have to develop and use formal domain models because software cannot be understood nor be developed if its relationship to the application domain is ignored. Having the domain knowledge separate from the programming knowledge will facilitate the transfer of the tools from one domain to another.

Third, we can change the development process so that the verification and maintenance are done on the specification rather than the source code. Our goal is to make the process of going from specification to code as easy as possible. When it is easy enough it will be feasible to make any changes on the specification and use the design system to reimplement from here.

Now, suppose our systems are not fully automated but remember all decisions made during development. If the decisions made are fully detailed and accessible understanding software during maintenance will be very much easier. During software maintenance one can replay the decision sequence and only change decisions of the first design, where the modification of software makes it necessary.

It is not enough to know what the decisions were; it is necessary to understand: the rationale behind the decision making, the goals being met, and the conditions which led us to this choice.

Software development and expert systems: Promises and applications The population of software engineers globally amounts to 10 million [14]. This number of software engineers can not simply cope with the increased demand for software systems. Since the software engineers are in short supply and more expensive, the promise of cheap advise through expert system technology has become more evident. To deal with the shortage of software engineers, even the great human designers, rather than rely on their expertise as we have done in the past, we develop expert systems.

There are other excellent reasons to use artificial expertise to enhance human reasoning: one advantage of artificial expertise is its performance. Human expert must constantly rehearse and use his expertise otherwise he will lose it. Whereas, once the artificial expertise is acquired, it is around forever. Another advantage of artificial expertise is the ease with which it can be transferred or reproduced. Transferring knowledge from one human to another is the lengthy, expensive and laborious process which is called education/knowledge engineering. Human expertise are more unpredictable in that human expert may make different decisions in identical situations because of emotional factors for example because of time pressure or stress he might forget using an important rule in a crisis situation.

Finally, artificial expertise is costly to develop, but once developed is relatively inexpensive.

The applications Today CASE technology faces new challenging areas. The investigation of such areas is likely to produce some application areas which suit expert system technology.

First, because software entities are extremely complex [Brooks87], therefore they are highly error prone. Roughly 25 to 50 percent of the development cost is spent on error removal activities [14]. Any aid promising the cost-effective error removal will be appreciated amongst the developers.

Second, huge amount of documentations are prepared which often forms up to 50 percent of the total cost of development. For example, 400 words of English form the documentation for a single line of code written in Ada for a typical U.S. military software development [14]. This challenging area could be addressed either by the CASE industry or through expert system technology.

Third, the concept of re-usability has been a weak link of software technology since it was formed 5 decades ago. The challenge for an expert system tool could be to support re-usability at multiple levels such as project plans, specifications, code, and documentation.

The optimiser expert system

Regarding the evaluation of software quality, many quality characteristics have been introduced. Each software entity owns many quality attributes for which a number of quality metrics exist. The existence of such numerous quality metrics has made the process of evaluating software quality exhaustive enough to undermine the accuracy of quality evaluation. The measurement of these quality metrics are mostly based on empirical techniques and heuristic knowledge employed by the experts in quality control. Such performance is labour-intensive, time-consuming, and exhaustive. The oversight and omission, performed by human, associated with such processes cause the result of quality evaluation to be less accurate.

In the department of Computer Science, we are developing an expert quality optimiser system to address the inaccuracy problem of software quality control. The diagnostic nature of such problem well fits in with the characteristics of an expert system.

THE QUALITY OPTIMISER AS AN EXPERT SYSTEM

In the previous two sections we have examined current software development issues and summarised the value of expert systems in relation to software engineering. We contend that a useful design approach to best meet the functions of a quality optimiser must involve the use of artificial intelligence. Partridge [24] suggests there are three classes of interaction between artificial intelligence and software engineering. Our quality optimiser falls into the class of an AI based support environment. It serves the purpose among others, of reducing the complexity of software development for the managers of the software project and for the software engineers.

The quality optimiser needs general software development knowledge and specific domain knowledge, it also needs to make heuristic decisions to provide

the software engineer with timely warnings and filter out unnecessary distractions. Expert systems perform a range of tasks as identified by Waterman's criteria [23]. Where an expert system performs a trivial task at which almost everyone is good this is considered an inappropriate use. At the other extreme it is considered impractical to design an expert system to perform a task which hardly anybody can do. Expert systems are best suited to the type of problem between these two extremes. The second criterion is that the task solution must be explainable in words rather than requiring explanatory pictures. The third criterion is the so-called telephone test. If the problem can be solved within an hour through a telephone conversation to an expert then the problem suits the expert system strategy. However if the problem takes a human more than a few days then it is far too complicated to be addressed by an expert system. Also, problems inviting one solution of many possible solutions are good candidates.

Types of decisions to be made by the quality optimiser and the types of advice it provides the software engineer suit the expert systems approach for reasons outlined above. For instance the quality optimiser advises which desk-top publishing package to employ on a project to minimise the learning curve and maximise both productivity and the quality of the documenters' output. It advises on the optimum frequency of progress meetings to minimise staff disruption and maximise communication among developers on the project.

We have divided the functional specification of the quality optimiser into several parts. The optimiser incorporates or subsumes the existing intelligent products in software engineering. The quality optimiser is concerned with support for software design, indicating design approaches which would lead to highest quality. A second major functional area is concerned with a selection of factors in the development environment. These factors include election of prototypes, choice of test tools, team organisation, management strategies and other factors which are known to effect cost, productivity and quality [15]. The speciality of staff on a project as well as their numbers affect the quality of deliverable items. The quality optimiser concerns itself with recommending numbers of systems analysts, analyst programmers, application programmers, systems programmers, technical authors, planning specialists, training specialists, implementation specialists, administrative support staff and others.

In the development environment the quality optimiser advises on the use of project management tool, quality management tool, configuration management tool, requirement specification workbench, design workbench, programming support environment, verification and validation tools, methodologies, graphic support, electronic mail, fourth generation languages, word processing, text processing, desk top publishing, implementation tools and others. The types of management strategy that are known to affect quality include the use of chief programmer teams, frequency and intensity of overtime working by the staff, the frequency and composition of progress meetings, whether the project management has an open door policy and so on. Among the deliverable items that are assessed for quality by the developers and users are the following: functional specification document, the project plan, quality management plan, acceptance test specification, the system models, system design specification, detail design documents, coded modules, tested modules, integrated subsystems, integrated

systems, the deliverable complete system, testing schedules, the user documentation for maintainers and for end users, the user training schedule, the system conversion schedule, the cut-over plan and others, see Table 1.

Reuse anticipation and Exploitation The quality optimiser has several further functional components. One of these is concerned with reuse and reverse engineering [8]. The growing importance of reuse in software engineering has special relevance to the demands made on safety critical systems [18]. All software engineers need to consider the concepts of reuse in relation to any of the deliverable items. The quality optimiser incorporates two reuse advisers. Reuse advice is of value to the software engineer when new software is being developed from scratch, in anticipation of future reuse possibilities. The software engineer makes decisions about design and system structure which are affected by the future reuse strategy. If the software and associated deliverable items are intended only for this particular client and this particular system environment then the developer will make design decisions accordingly. If however, the developer foresees a future possibility of reuse, then the design decisions are different, the structure is more generic. The developers can choose to build general purpose software components and generalised deliverable items such as plans, schedules, test harnesses. The reuse adviser is therefore useful before any reuse is being employed to advise on the desirability, costs and benefits of allowing for the future possibility of reuse.

In the future years, software engineers will find themselves in more and more projects where instead of developing software from scratch they will have access to a library of reusable components. Not only software but other deliverable items can also be stored in libraries for future reuse. When confronted by a new functional requirement, the software engineer needs to make a decision on which components to reuse and how much tailoring and adaptation they may need. He needs to develop a reuse exploitation strategy based on the costs and benefits of developing the item from scratch or reusing existing material. The reuse adviser is a valuable assistant in this type of decision making. This reuse advice takes place after the libraries of reusable components have been built up.

These two aspects of reuse advice, anticipation and exploitation, will become more important during the 1990s as libraries of reusable components grow and are shared among the community of software engineers. As reuse grows then the quality perspectives of Table 1 will change. For instance the quality of coded modules and integrated subsystems will become very relevant as potential future marketable items.

Another component of the quality optimiser is a model of the type exemplified by Qualcomo [25]. The adviser predicts quality using deterministic and probabilistic algorithms in the same way that COCOMO [15] predicts project costs. The COCOMO and Qualcomo models were developed from empirical data and did not use expert system techniques. They have proved valuable in software engineering and the results can be readily integrated with the heuristic features of the quality optimiser.

The quality optimiser advises on electing to build a prototype, or prototypes, of all or part of the system under development. Prototypes can help in clarifying user requirements and in exploring design approaches. Their drawback is their

time and cost of production. Reconciling these costs and benefits is difficult and affected by parameters some of which are unique to each project. Assistance from a quality oriented expert system could be invaluable in this area.

The functionality of the quality optimiser demands an integrated approach which incorporates the relevance to quality of the factors mentioned above. The perspectives of the different people assessing quality, that is to say the users, the user management, the developers and the developer management, are all different. The quality optimiser must encompass these differences in its handling of quality decisions and since the perspectives are interdependent the optimiser integrates their treatment. Table 1 illustrates the matrix of perspective weightings applicable to the deliverable items of a typical software project. The managers, accountants and marketeers of the developer organisation have the most obtuse views of the quality of deliverable items. They are nonetheless important so the quality optimiser takes them into account. The accountants, sales people, personnel staff have interests concerned with profits, marketability of the product and staff satisfaction which impose unusual parameters on the quality. For example sales personnel like to see sales-oriented documentation, literature that is persuasive as well as informative. Maintenance engineers and operational staff prefer pessimistic reports of the worst that might go wrong with the system so they can prepare for all eventualities. These perspectives will vary from project to project and from organisation to organisation and demand the heuristic treatment affordable from an expert system.

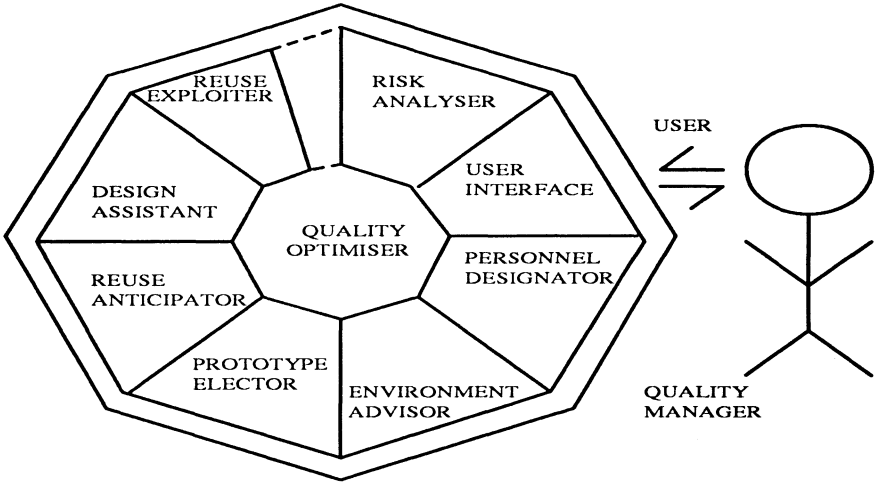

Figure 2: Function Structure of the Quality Optimiser

The way in which the different areas of functionality required of the quality optimiser should be integrated depends on the design approach. We have established that it combines expert system technology with a knowledge base and an inference engine. It also embodies straightforward mathematical modelling with probabilistic functions to predict cost and quality. The knowledge base is

drawn from the experience of software engineers, quality managers and software project managers. The models are derived from empirical observations on past software projects. The quality optimiser is a large undertaking and will evolve as experience is gained in the areas of reuse and risk management. Figure 2 shows the functional structure of the quality optimiser. Experiments with component prototypes are providing useful results.

CONCLUSION

In the fifth decade of software engineering the symptoms of the software crisis are still prevalent. The skills and technologies for software development have improved but at the same time the improvement has been counteracted. The relative costs of software compared to hardware have escalated and the demands of safety critical systems have increased. The combined effect has put more emphasis on quality in all aspects of the professional work of the software engineer.

Perspectives of quality derive from the developers and from the users of software systems and are multidimensional. Many deliverable items, along with the final working code, are assessed for quality. We contend that a holistic approach to quality management is justified. The strategy of quality management should integrate its treatment of different perspectives and of different deliverable items.

The interaction of artificial intelligence, especially expert systems, with software engineering has proved synergetic in many areas and we believe that many problems of quality management can be attacked using expert system methods. These methods can be combined with conventional mathematical modelling to build a holistic quality optimiser which assists the quality manager in some of his, or her, most difficult decisions.

We have devised a quality optimiser whose functional components include :- a design assistant, a reuse anticipator, a reuse exploiter, a personnel designator , a development environment advisor, a risk analyser, a prototype elector, a quality predictor and a user interface. Our experiments with prototype components of the quality optimiser are providing valuable results and we expect the prototypes to evolve and coalesce as we gather more expertise into the knowledge base and integrate existing management aids.

REFERENCES

1. Pressman, R.S. *Software Engineering: A Practitioner's Approach* Mcgraw Hill, Singapore, 1987.
2. Basili, V. and Musa, J. ' The Future Engineering of Software: A Management Prospective' *IEEE Computer*, Vol. 24, pp. 90-96, 1991.
3. Brooks jr, F.P. ' No Silver Bullet' *IEEE Computer*, Vol. 20, pp. 10-19, 1987.
4. Gladden, G.R. ' Stop The Life Cycle, I Want To Get Off' *ACM SIGSOFT*, Vol.7, pp. 35-39, 1982.
5. Balzer, R. and Musa, J. ' A 15-year prospective On Automatic Programming' *IEEE Transaction On Software Engineering*, Vol. 11, pp. 1257-1267, 1985.
6. Schach,S.R. ' Software Engineering ', Chapter 3, *Software Life Cycle Models*, Vol. 1, pp. 43-67, Aksen Associates, Boston, 1990.

7. Boehm, B.W. ' A Spiral Model of Software Development and Enhancement' *ACM SIGSOFT Software Engineering Notes*, Vol. 11, pp. 22-42, 1986.
8. Hodgson, R. ' The X Model: A Process Model for Object-Oriented Software Development ', pp. 1-37, *Proceedings of 4th Int. Conf. on Software Engineering and its Applications*, Toulouse, France, 1991.
9. Shaw, M. ' Prospect for an Engineering Discipline of Software ' *IEEE Software*, Vol. 16, pp. 15-24, 1990.
10. Sharp, H.C. ' KDA: A Tool for Algorithmic Design Evaluation and Refinement Using the Blackboard Model of Control', pp. 407-416, *Proceedings of Int. Conf. on Software Engineering* , Singapore, Singapore, 1988.
11. Balzer, R., Cheatham, T.E. and Green, C. ' Software Technology in The 1990's: Using a New Paradigm' *IEEE Computer*, Vol. 16, pp. 39-45, 1983.
12. Rine, D. ' Software Perfective Maintenance: By Retrain-able Software ' *Accepted for publishing in 1992.*
13. Chen, M., Nunamaker, J. and Weber, E. ' Computer Software Engineering: Present Status and Future Directions ' *Data Base*, Vol. 20, pp. 7, 1989.
14. Jones, C. ' CASE's Missing Elements ' *IEEE Spectrum*, Vol. 29, pp. 38-41, 1992.
15. Garratt, P. ' MULTIPLAN, a 'What if' Planning Tool for Software Development ', *Proceedings of the IEE Conf. on Management Technology*, Nicosia, Cyprus, 1990.
16. Flatman, A. and Russell, B. *The Evolution of Corporate Network Infrastructures Through the 1990's* ICL, Bracknell, 1990.
17. Mandell, S.L. *Computers and Information Processing* West Publishing, St. Paul, 1992.
18. Bennet, P. ' Software for Computers in the Application of Industrial Safety-Related Systems ', *Proceedings of Safety-Critical Systems Club*, Cambridge, UK, 1992.
19. Shepperd, M.J. and Ince, D.C. ' An Empirical and Theoretical Analysis of an Information Flow-based System Design Metrics ', *Proceedings of 2nd European Software Engineering Conf.*, Southampton, UK, 1989.
20. Williams, D. 'Software Engineering Manager, Marconi Systems', *Verbal Communication*, 1990.
21. Kitchenham, B.A. and Walker, J.G. ' An information Model for Software Quality Management ', *TSQM Deliverables, A24, STC Ltd.*, Newcastle Upon Lyne, UK, 1988.
22. Fenton, N.E. *Software Metrics: A Rigorous Approach*, Chapman and Hall, London, 1991.
23. Waterman, D.A. *Introduction to Expert Systems*, Oddison-Wesley Publishing Company, Reading, Massachusetts, 1986.
24. Partridge, D. ' Artificial Intelligence: A Survey of Possibilities ' *Information and Software Technology*, Vol. 30, pp. 146-152, 1988.
25. Dunsmore, H.E., Conte, S.D. and Shen, V.Y. *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company, Wokingham, Reading, and Massachusetts, 1986.