33

How business relationship modelling supports quality assurance of business objects

R.A. Veryard
Texas Instruments
Wellington House, 61-73 Staines Road West, Sunbury-on-Thames, Middlesex UK TW16 7AH Phone +44 1784 212560
Fax +44 1784 212600 Email 5913301@mcimail.com

Abstract

Many claims have been made for the benefits of software reuse, in terms of enhanced quality as well as productivity. Widely reused objects are supposed to possess several desirable characteristics, such as reliability and flexibility as well as efficiency. But reusable objects may be used for unpredicted purposes in unpredicted contexts. Furthermore, in an open distributed world, technical, geographical, organizational and other boundaries, as well as significant time lags, may separate the software developer from the publisher, and the software librarian from the user. From these premises we argue that responsibility for the quality of reusable software artefacts cannot be taken by the developer (or development organization) alone, but must be shared between developer and other agents. Recent work in enterprise modelling for open distributed processing has led to new techniques for modelling responsibilities across organizational boundaries, and these techniques are introduced here as a way of determining and clarifying effective structures for the quality assurance of reusable business objects.

Keywords

Business Object, Software Reuse, Quality Assurance, Open Distributed Processing, Responsibility Modelling, Enterprise Modelling, Requirements Engineering

Acknowledgements

Some of the work described in this paper was carried out within the Enterprise Computing Project, grant-aided by the UK Government Department of Trade and Industry, involving the John Dobson, Ian Macdonald, Rob van der Linden, David Iggulden and the author. Thanks are also due to John Dodd, Aidan Ward, John Reilly, Michael Mills and Richard Gilyead.

S. Bologna et al. (eds.), *Achieving Quality in Software*

1. INTRODUCTION

This paper introduces the techniques of business relationship modelling to reconcile an apparent mismatch between the expectations and requirements of software quality assurance on the one hand, and the expectations and requirements of reusable business objects on the other hand.

Levels of confidence in software quality vary widely, from misplaced complacency at one end of the spectrum to excessive caution at the other end of the spectrum. The purpose of quality assurance is to establish reasonable and realistic levels of confidence. Confidence is associated with predictability, stability and trust. But technology trends, especially object reuse and open distributed processing, appear to reduce the factors that lead to confidence in software quality. This is the challenge addressed by this paper.

If software users want to take advantage of unknown objects from anonymous sources, how confident can they reasonably be that these objects are fit-for-purpose? What assumptions can the users make about such objects? Frakes and Fox (1995) argue that quality concerns do not currently inhibit reuse, but they indicate that this situation may change. Meanwhile, what quality checks can and should a developer perform, before submitting an object for publication and dissemination, which may result in the object's being used in unanticipated contexts and for unanticipated purposes. What kind of systems and relationships are necessary, to attain the maximum reasonable level of confidence in software quality, without unduly restricting technological progress?

Table 1 Stakeholder concerns motivating quality assurance

User concerns	*Developer concerns*
• what evidence is there that this object is likely to work properly in my application?	• what evidence is there that this object is likely to work properly in real user applications?
• has the object been tested in a way that is relevant to my intended use?	• has the object been tested in a sufficient variety of situations?
• how much serious usage has this object had, in areas similar to my intended use?	• is the object designed for efficient performance in a reasonable range of contexts?
• what are the performance / capacity implications of using this object?	

This paper argues that full responsibility for quality can be taken neither by the developer nor by the user, nor by any intermediate party (such as a broker or software publisher). Thus effective quality assurance needs to focus on the roles and responsibilities and relationships between the various stakeholders in the object delivery chain.

The paper is in two parts. The first part states the problem: it describes quality assurance as a way of acquiring reliable knowlege about the quality of objects, and indicates how reuse cuts across our traditional ways of knowing. The second part indicates the solution: the application of responsibility modelling techniques to determine structures for the sharing of responsibilities across the object delivery chain.

The paper is derived from, and is intended to demonstrate the connections between, Texas Instruments' recent work in three separate areas:

- Software quality management
- Business relationship modelling for open distributed processing (also known as enterprise modelling)
- Business object modelling and component-based development.

2. SOFTWARE QUALITY ASSURANCE VERSUS REUSE

In this part of the paper, we describe what quality assurance is, and what makes it possible. We then describe what software reuse is, and how it potentially conflicts with the enablers of quality assurance.

2.1. Quality assurance - a process of discovery

Quality assurance is defined as 'all the planned and systematic activities implemented within the quality system, and demonstrated as needed, to provide adequate confidence that an entity will fulfil requirements for quality' (ISO 8402, 1994). (In this context, the word 'entity' may refer to an activity or a process, a product, an organization, a system or a person, or any combination thereof.) In short, therefore, quality assurance provides **knowledge** about the quality of entities.

In common with most other attempts to gain knowledge, quality assurance cannot avoid affecting the entities about which it attempts to gain knowledge. Any attempt to discover the degree of quality of an entity may bring about an improvement in the quality of that entity. Indeed, such improvement is often seen as the primary justification of quality assurance.

Conversely, any prediction of the quality of an entity should take into account the extent of quality assurance that may be applicable. In particular, if a software developer has access to quality assurance mechanisms that make the emerging quality of a software artefact visible during the development process (other conditions being favourable), this should reduce the occurrence of defects in the developed artefact.

Quality assurance can focus on three areas:

- Product certification, based on product standards
- Process assessment or audit, based on a repeatable software process
- Organizational capability and commitment, based on market regulation and/or long-term business relationships.

Although some software product standards exist, and further standards are being developed, these only address a limited subset of the desired quality characteristics.

2.2. Quality - a context-dependent property

As stated above, quality assurance provides knowledge about the **quality** of entities.

Quality is commonly (and properly) defined in 'fit-for-purpose' terms. The official ISO definition (ISO 8402, 1994) defines quality as "The totality of characteristics of an entity that

bear on its ability to satisfy stated or implied needs." (This is an incomplete definition: it allows for some uncertainty about **when** the needs are stated, and **what** they may be implied by.) To be valid and useful, quality assurance must find some way of addressing the stated and implied needs. Any attempt to construct a 'pure' quality assurance, independent of user context or purpose, would necessarily fall short.

Similar arguments apply to software quality metrics. Although generic software quality measurements have been attempted, often copied from the equivalent hardware quality measurements (e.g. Mean-Time-Between-Failures, Mean-Time-To-Fail), software quality of service is often better expressed in application-dependent terms. For example, mission time, which is defined as the time during which a component is to stay operational with a precisely defined probability of failure (Siewiorek & Swarz, 1992)

Within the traditional software development paradigm (waterfall development, central design authority, fixed requirements, or what has been ironically called the 'fixed point theorem' (Paul 1993)), the evident ambiguities of the ISO 8402 definition of quality are usually glossed over. Quality assurance is carried out against a fixed set of requirements of a fixed user or user group.

ISO 9126 offers a standard framework for evaluating the quality of software products (ISO 9126, 1991). Of the six standard characteristics, four are defined in terms of stated requirements (reliability, efficiency, maintainability and portability), while the other two are defined in terms of stated or implied requirements (functionality and usability). Reusability does not appear explicitly in the ISO 9126 model, but it can probably be regarded as a special type of usability, namely usability-by-developers.

A statement of requirements is a description which an object must satisfy for its actual use, for a given purpose, in a given context. We call these the actual requirements. When developing an object for reuse, however, the developer usually does not have access to the complete set of concete requirements. Instead, the developer attempts to build reusable objects by working against an generalized statement of requirements that is hoped to cover a reasonable range of actual requirements. Carrying out QA against an generalized statement of requirements, however, begs the question: to what extent will the developer's generalized notion of the users' requirements match the users' actual requirements?

Thus the problem with restricting our notions of quality (and therefore our notions of quality assurance) to the formally stated requirements is twofold:

i it fails to reflect the practical realities of software development, namely that requirements emerge and evolve, and are never fully stated upfront, despite the insistence of theorist

ii it fails to reflect the chronology of object reuse (see Figure 1), whereby most of the uses of an object are invented after the object has been developed, thus the concrete requirements don't yet exist at the time of development.

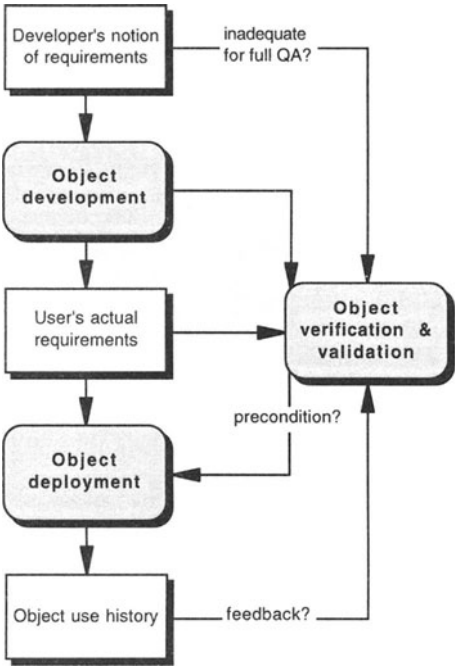

Figure 1 Chronology of object reuse.

2.3. Changing trends in information systems development

Until recently, most developers of Information and Communication Technology (ICT) systems have worked within a certain 'paradigm' - a set of assumptions about the nature of the systems they have been developing, and about the nature of the development work itself. These assumptions have included things like:

- central problem ownership, central design authority, central funding authority
- complete understanding of a given problem/situation
- permanent/final solution to a given problem
- stable or slow-moving market environments, technical architectures and policy frameworks
- life-cycle model of systems development (waterfall, spiral or whatever)

Perhaps these assumptions were never really true. But in most situations they could at least be regarded as reasonable approximations to the truth. However, this old paradigm is becoming increasingly untenable.

As requirements engineering and information systems engineering become increasingly dynamic and decentralized, quality assurance itself needs to become equally dynamic, equally decentralized. The acquisition of knowledge about quality becomes more problematic than

ever: when (if ever) and from what standpoint (if any) can quality be assured? To examine these issues in more detail, let us look at two major themes in the breakdown of the traditional software development paradigm:

i object orientation and reuse

ii open distributed processing.

2.4. Object orientation and reuse

Considerable work has been done in recent years on techniques of reuse. This has been largely focused on object-oriented development methods, and on so-called 'business objects'.

The Object Management Group special interest group on Business Object Management (BOMSIG) has developed the following working definition of 'business object':

> "A representation of some thing active in the business domain, including at least its behavior, attributes, constraints (business rules), associations, a business name, and a business definition. A business object may represent, for example, a person, place or concept. The representation may be in a natural language, a modeling language, or a programming language." (OMG 1994)

A business object might be something internal to the enterprise, such as an Employee or a Department, or something external to the enterprise such as a Supplier or Tax Demand. Business objects may be conceptual, in which case we may think of them as business or design **patterns** (Alexander et al 1977), but they may also be implemented in executable software. In any case, the dividing line between conceptual objects and executable software is tenuous (Negishi 1985, Locksley 1991).

The potential benefits of software reuse are certainly considerable, not merely for enhanced productivity but also for enhanced quality. Much has been said (in passing) about the impact of object-oriented techniques on quality:

- "Reusable components ... provide products of higher quality, since components have already undergone rigorous testing before they are installed."
- "High reliability is attained because components are designed and tested as separate units, and in many different systems."
- "Object-oriented programming languages make it possible to develop reusable, modular code segments, improving the quality of the development process and of the resulting software product."

However, although the claimed productivity benefits of reuse have been extensively analysed (Veryard 91), little empirical evidence is available to support the claimed quality benefits.

The attainment of these benefits is not just a technical matter. Benefits may be lost by poor project management, or by lack of engineering discipline. This is where quality management comes in. It provides a systematic framework underpinning good management and engineering practices, thereby providing reasonable assurance to all stakeholders (including end-users and senior management) that the potential benefits of a given technology will indeed be forthcoming.

For example, it is all very well using a reusable component from a catalog or library, on the assumption that someone else has rigorously tested it. But what if they haven't? And if they have, who is to say that their testing is relevant to your intended use of the component?

An effective process of reuse relies on adequate and accurate descriptions of the objects. Indeed, where reuse crosses commercial boundaries, procurement decisions may be based solely on the description: a purchaser may only get the rest of the object after accepting certain payment obligations.

But there may be semantic subtleties or ambiguities in these descriptions. For an illustration of this, consider a business object called FIXED ASSET that implements a particular interpretation of a particular accounting convention to evaluate fixed assets. An incorrect choice of accounting convention may significantly affect the balance sheet. Therefore, a quality evaluation of a business object must address its description (or catalog entry) as well as the quality of any executable code. Quality may be improved by clarifying or extending the description. But the perfect description may never be achievable.

Various mechanisms are used in the software industry to allow the purchaser to try out a business object before committing to purchase. These mechanisms often involve giving the purchaser access to a restricted version of the object. Although these mechanisms may enable the user to verify that the object fits the user's requirements, they also introduce additional quality complications: what guarantee is there that the full version of the object and the restricted version behave identically.

The supply chain may be complex. Departmental software specialists will be using desktop tools for rapid assembly of software applications from existing components from many sources. There will be few if any software developers who develop everything from scratch; instead, even apparently original objects may contain smaller components from other sources.

2.5. Open distributed processing

Open distributed processing (ODP) is an umbrella term for a number of concepts and architectures for distributed computer systems. Architectures include the ODP Reference Model (ISO 10746, 1995), CORBA (OMG 1992), DCE (OSF 1991) and ANSA (APM 1991). For an analysis of ODP market trends, see (van der Linden, 1995). The key features of ODP are shown in Table 2:

Table 2 Key features of Open Distributed Processing

Federation	The lack of central authority over software design or configuration
Interoperability	The ability to link and reconfigure systems and services
Heterogeneity	The ability to link across different platforms and protocols
Transparency	The ability to hide complications from users
Trading / broking	The presence of intermediary agents, to promote and distribute software artefacts and services.

The relevance of ODP to this paper is that ODP further undermines the assumptions on which traditional software quality assurance is based. End-to-end knowledge and authority cannot be guaranteed; information flows must be negotiated across organizational boundaries.

2.6. Summary of changes to software quality assurance

- Patterns of (re)usage of a software artefact are not predictable
- Technical characteristics of software artefacts are hidden from the user, in the name of 'transparency'
- Individual developers may have no monopoly over supply
- Individual brokers and traders may have no exclusive control over the distribution of any given artefact
- Individual users may have no exclusive control over usage

These changes affect at least three important aspects of software quality assurance: design reviews, testing and configuration control.

The developer of a software artefact typically has incomplete knowledge of how it will be used, when, where, by whom, in what contexts, for what (business) purposes. Nobody has the complete picture: not the software tester, nor any individual end-user, nor any intermediary (planner, publisher, broker/trader, librarian, ...).

Although often there is no fixed specification of what the software artefact is required to do, some limited testing is possible, for example:

- whether the software artefact conforms to its own description
- whether the software artefact conforms to standard constraints (such as absence of side-effects)

Where testing is more difficult, and its results less conclusive, is in the compatibility of multiple objects and artefacts. This is complicated by the fact that we don't always know what compatibility is required. This is related to the problem of **feature interaction**.

3. STRUCTURING QUALITY ASSURANCE

3.1. Sharing responsibilities for quality assurance

Under the conditions described above, there is no individual standpoint from which software quality assurance can be effectively performed. The combination of partial knowledge and partial authority make it impossible for any single agent to take full responsibility for quality. Responsibility then attaches to the group, not to the individual. A 'system' of reuse is owned collectively by all the participants in the supply chain; quality assurance must be a collective responsibility.

In general, there are two approaches to organizing collective responsibilities, as shown in Table 3.

Table 3 Collective responsibility - two approaches

Homogeneous closed group stable network based on informal commitments (trust)	In some situations, responsibility can be shared by creating a stable group or team. The members of the group **identify with** the group. This has two aspects: • Each member depends on the group - personal rewards are significantly affected by the success of the group as a whole • Each member feels that the group depends on him/her - the personal contribution to the group is understood and appreciated by the rest of the group
Heterogeneous open group dynamic network based on formal commitments (contract)	In situations where stable teams cannot develop (or be built), formal structures of individual duties need to be established, to enable responsibilities to be shared. This tends to be necessary under the following conditions (Elster 1978, pp 134 ff): • High turnover of group membership, interfering with the emergence of stable relations of personal trust • High cultural / organizational diversity, increasing the barriers that have to be overcome.

Because of its tilt towards heterogeneity, open distributed processing forces us to consider the latter approach.

3.2. Business relationship modelling

Business relationship modelling (also known as responsibility modelling) is required to explicitly share the responsibility for quality across a network of interacting agents, each possessing partial knowledge and partial authority.

We define responsibility as a three-part relationship, involving two agents and a state. A state is typically a property of an entity; remember that an entity may be an activity or a process, a product, an organization, a system or a person, or any combination thereof.

Example: *The software developer is responsible to the software publisher for the thoroughness of testing.*

In this example, the software developer and the software publisher are the two agents, and thoroughness of testing is the state, which can be decomposed into a property (thoroughness) and an entity (the testing activity).

One of the reasons we are interested in the thoroughness of testing is that it is a way of gaining confidence in the robustness of the tested object(s). There are thus dependencies between states (robustness is dependent upon thoroughness), which can be represented as dependency hierarchies. The robustness of the tested object(s) is thus a state at the next level in the hierarchy.

Responsibilities can be delegated, but not escaped. Thus if the software developer delegates the responsibility for the thoroughness of testing to a third party, the software developer

remains answerable to the software publisher. Meanwhile the software publisher is (let us suppose) responsible to the software user for the robustness of the tested object, which (as we have seen) is dependent upon the thoroughness of testing.

Current thinking in both ODP and quality management agrees on the fact that delegation should be (formally) 'transparent', in other words the original source(s) should be invisible to the user or customer. If I purchase an object or service from a software publisher, I don't want to have to chase up the supply chain to get satisfaction. (In practice, of course, the procurement organization does want to know about the original source, doesn't want complete transparency of supply, but this may be a consequence of incomplete trust in the vendor.)

Responsibility modelling is a technique for mapping out these delegation structures for specific responsibilities across multiple agents. Delegation structures may be informal or formal, hierarchical or contractual. Delegation structures should be reflected in contracts and agreements between agents.

Delegation structures should also be reflected in information systems. According to good delegation practice, if the software developer delegates the responsibility for the thoroughness of testing to a third party, the software developer must have some mechanism for monitoring and assessing the third party. (This is of course demanded by ISO 9000.) This means that there must be an information system (not necessarily computerized) that enables the software developer to continue to provide the necessary assurance to the software librarian, and means that the delegation can be (in principle) transparent to the users.

Previous techniques for responsibility modelling, such as RAEW analysis (Crane 1986, Texas Instruments 1990), have regarded responsibility as a two-place relation between an agent (or role) and an activity. This has proved extremely useful for identifying inconsistencies between responsibility structures and authority structures, or between responsibility structures and the distribution of knowledge. What the present approach offers in addition is the ability to model delegation structures, especially where these cross organizational boundaries.

There are many approaches to the formal division of responsibilities for quality. We shall examine three:

i Allocation of characteristics
ii Counter argument
iii Collective feedback

3.3. Allocation of characteristics

An agent takes on the responsibility towards a second agent for the possession by an artefact of a given subset of the desired characteristics.

A statement such as 'this artefact has such-and-such properties' can never be guaranteed 100%. However, the responsible agent provides a mechanism, accessible to the second agent, for corrective action.

For example, separating responsibility for the efficiency of an object from the responsibility for its functionality. Such separation of concerns will typically be based on the ISO 9126 standard.

This may be of some limited use, but fails to address the holistic aspect of quality explicitly addressed in the ISO 8402 definition.

3.4. Counter argument

A common approach is to establish a divided responsibility:

- the developers (or their representatives) attempt to prove the quality of a developed object
- a separate team of testers and/or inspectors attempt to prove the lack of quality of a developed object

Quality assurance emerges from the dialogue between two (or more) agents with clearly defined responsibilities and obligations. Responsibility modelling allows us to represent and analyse the respective duties of the two groups, as well as the necessary interactions between them, thus ensuring that the responsibilities and obligations are both clear and optimal.

3.5. Collective feedback

A more sophisticated approach is to create a collective feedback and feedforward system. This passes information from the users of an object backwards to the object source, and forwards to other users and potential users, thus establishing a self-correcting process.

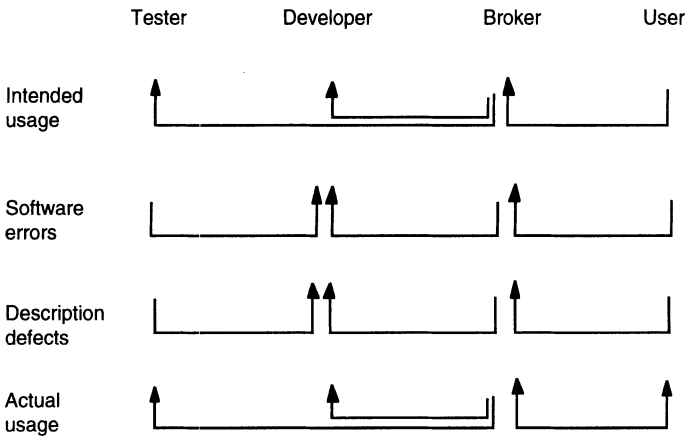

Figure 2 Feedback chains.

Intended and actual use should be fed back to testers as well as developers. Errors and defects should be fed back to the developers by the users, via intermediaries with responsibility for software distribution, such as software publishers or software brokers. They may also be fed back to the testers (since they provide some feedback on the thoroughness and relevance of the testing). Other users may want to be notified, not only of any outstanding software errors and description defects, but also of the pattern of error rates and defect rates, from which they may want to make judgements about expected future error and defect rates.

3.6. Organizational issues

One of the uses of responsibility modelling is to assess whether the organizational boundaries are in the right place. For example, does it make more sense for the development organization to employ the testers (supply-side testing) , or the procurement organization (demand-side testing). These questions need to be negotiated between suppliers and purchasers, based on a clear understanding of their implications for effective and efficient quality assurance.

The feedback chains described in the previous section usually need to work across organizational boundaries. This is likely to raise additional organizational issues. Business relationship modelling may also indicate the need for additional agents, such as independent (third party) assessors and advisors, as well as market regulators.

Detailed description of the use of responsibility modelling to support such structuring judgements can be found in the reports of the Enterprise Computing Project (Veryard 1995)

The developer or vendor may offer specific warranties, or may be subject to various liabilities, either through formal contract or through common practice. These legal aspects of the relationships need to be considered carefully.

4. CONCLUSIONS AND FURTHER WORK

Quality assurance can focus on the quality of the product, the quality of the process, or the quality of the business and organizational relationships. In an open distributed world of reusable business objects, the relationships between parts of the same organization - or between different organizations - are crucial to effective quality assurance. This paper has indicated the use of responsibility modelling as a way of supporting the organization and management of quality assurance.

The necessity of this approach has been argued on the basis of the impossibility of predicting all the uses and use-contexts of a reusable object, and the inappropriateness of unduly restricting such uses and use-contexts. This largely invalidates traditional quality assurance mechanisms.

Further development in this area may benefit from the production of support tools, as well as standards for responsibility distribution.

5. REFERENCES

C. Alexander, S Ishikawa and M. Silverstein (1977) *A Pattern Language*. Oxford University Press, New York.

APM (1991) *ANSA: A Systems Designer's Introduction to the Architecture.* APM Ltd., Cambridge UK: April 1991.

Roger Crane (1986) *The Four Organizations of Lord Brown and R.A.E.W.*. Doctoral Thesis, Kennedy-Western University.

John Dobson and Ros Strens (1994) Responsibility modelling as a technique for requirements definition. *Intelligent Systems Engineering* **3** (1) pp 20-26.

John Dodd (1995) *Component-Based Development: Principles.* Texas Instruments Methods Guide: Issue 1.0, March 1995.

Jon Elster (1978) *Logic and Society: Contradictions and Possible Worlds* John Wiley & Sons, Chichester UK.

W.B. Frakes and C.J. Fox (1995) Sixteen Questions about Software Reuse. *Communications of the ACM* **38** (6), pp 75-87.

ISO 8402 (1994) *Quality Management and Quality Assurance Vocabulary* International Standards Organization, Geneva.

ISO 9000 (1994) *Quality Management and Quality Assurance Standards* International Standards Organization, Geneva.

ISO 9126 (1991) *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use* International Standards Organization, Geneva.

ISO 10746 (1995) *Basic Reference Model for Open Distributed Processing* International Standards Organization, Geneva.

Rob van der Linden, Richard Veryard, Ian Macdonald and John Dobson (1995) *Market Report* Enterprise Computing Project.

Gareth Locksley (1991) Mapping strategies for software business, in (Veryard 1991) pp 16-30.

H. Negishi (1985) Tentative Classification of Global Software *Behav. Inf. Technol* **4** (2), pp 163-170.

OMG (1992) *Object Management Architecture Guide* Revision 2, Second Edition, Object Management Group, September 1992.

OMG (1994) Minutes of BOMSIG meeting, Object Management Group, April 7, 1994.

OSF (1991) *DCE User Guide and Reference* Open Software Foundation, Cambridge MA.

Ray Paul (1993) Dead Paradigms for Living Systems. Paper presented at the First European Conference on Information Systems, Henley, 29-30 March 1993.

D.P. Siewiorek and R.S. Swarz (1992) *Reliable Computer Systems* Digital Press.

Texas Instruments (1990) *A Guide to Information Engineering using the IEF™* Texas Instruments Inc., Plano TX, Second Edition.

Texas Instruments (1991) *Strategy Announcement: Arriba! Project.* Version 1.0, TI Software Business, Advanced Technology Marketing, January 5th, 1995.

Richard Veryard (1991) (ed) *The Economics of Information Systems and Software.* Butterworth-Heinemann, Oxford.

Richard Veryard (1994) *Information Coordination: The Management of Information Models, Systems and Organizations.* Prentice Hall, Hemel Hempstead UK.

Richard Veryard and Ian Macdonald (1994) EMM/ODP: A methodology for federated and distributed systems, in *Methods and associated Tools for the Information Systems Life Cycle* (ed. A.A. Verrijn-Stuart and T.W. Olle), IFIP Transactions, Elsevier/North-Holland, Amsterdam.

Richard Veryard, Ian Macdonald, Rob van der Linden and John Dobson (1995) *Enterprise Modelling Methodology.* Enterprise Computing Project.

6. BIOGRAPHY

Richard Veryard is a Principal Consultant in the Software Business of Texas Instruments, working within the Group Quality Department. He was one of the developers of IE\Q, which is a TI proprietary methodology for software quality management. He has been working with advanced software tools and methods for over fifteen years, and is the author of several books on information systems. He is a member of IFIP Working Group 8.6.