Quality criteria for multimedia

Nick Rushby

PA Consulting Group, 123 Buckingham Palace Road, London SW1W 9SR

The meaning of the term quality as used by multimedia workers in the field has become devalued. Almost every package is promoted by its developers as being of the 'highest quality'. This paper draws on practical experience from a number of major projects to argue, from a quality-assurance position, that multimedia materials should meet pre-defined criteria relating to their objectives, content and incidence of errors.

What do we mean by quality?

Almost every piece of multimedia software is extolled by those who developed it and those who publish it as being of the 'highest quality'. In practice, most multimedia packages contain content and software errors, and do not realize their full potential in terms of communicating a message that endures. This does not mean to say that the users are disappointed with their multimedia. They may not realize that the communication could be more effective, they may not notice some of the errors, or they may have been led to believe that this is the natural order of things. The accepted meaning of the term *quality* as used by workers 'in the field' has become devalued. Discussions with typical purchasers and users over recent months indicate that they are still remarkably tolerant of low quality. In this respect, multimedia lags behind much of the IT industry, and it is unlikely that the tolerance will last much longer. As users expectations and demands increase, those multimedia developers who have quality processes in place and are able to assure the quality of their products will have a clear business advantage.

The European Commission has identified among its priorities for educational multimedia, the need for quality control – methods and procedures evaluating the technical quality of educational software and the infrastructure for checking and certifying quality procedures (European Commission, 1996).

This paper draws on the practical experience of managing and commissioning a large

number of complex multimedia projects for major clients. It argues that we should set our expectations, both as developers and as users, far higher than they are at present, and work towards creating multimedia software that does what the developers and publishers claim it will do. In this context, assuring the quality of multimedia means providing an assurance that the product meets specific criteria, typically in respect of its effectiveness and robustness. Unless we set out those criteria in advance, it is difficult to manage a project so as to achieve them, or even to recognize whether they have been achieved at all. It is easier to improve the quality of any product if we know what criteria we are achieving now.

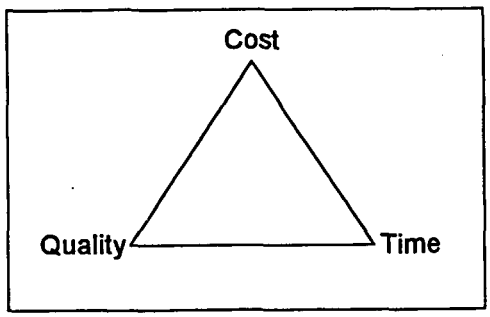

Figure 1:The project manager's dilemma

Not all multimedia needs to be of the highest quality. Excellence may be sacrificed deliberately to reduce costs or time, to meet constraints imposed on the project. Balancing time, cost and quality is the dilemma that faces every project manager.

However, within those constraints it behoves us to deliver the most quality that we can and, as users, to be less tolerant of multimedia that falls short of our expectations. In the body of this paper I look at the impact of rapid applications development, the quality of communication, and the quality of the product, discussing what criteria are appropriate and giving some thought to how they might be achieved.

The Impact of Rapid Applications Development

There are some very good reasons why a Rapid Applications Development (RAD) approach to multimedia, introducing a highly parallel, iterative model as shown in Figure 3, is used in preference to a more traditional waterfall process shown in Figure 2. A major benefit is the early and continual involvement of end users, which maximizes their commitment to the completed package. However, giving users a real say in the evolution of the package through its prototypes, and empowering them to suggest significant changes to structure, content and functionality, results in significant rework.

A significant reason why projects overrun is not because of poor productivity but the amount of work that has to be re-done late in the project timeframe. An important conclusion to draw is that projects will more easily hit their targeted end-dates if rework is identified early in the project. Traditional software development approaches do not promote the early discovery of rework – typically major amounts of rework are identified only during testing after all the software has been 'completed'.

Changes suggested by users are identified earlier through the use of iterative development techniques. Each iteration delivers a working component of the final system and ensures

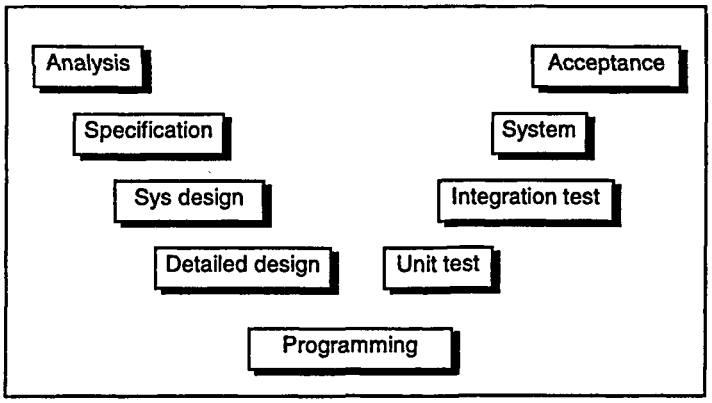

Figure 2: The traditional 'waterfall' development

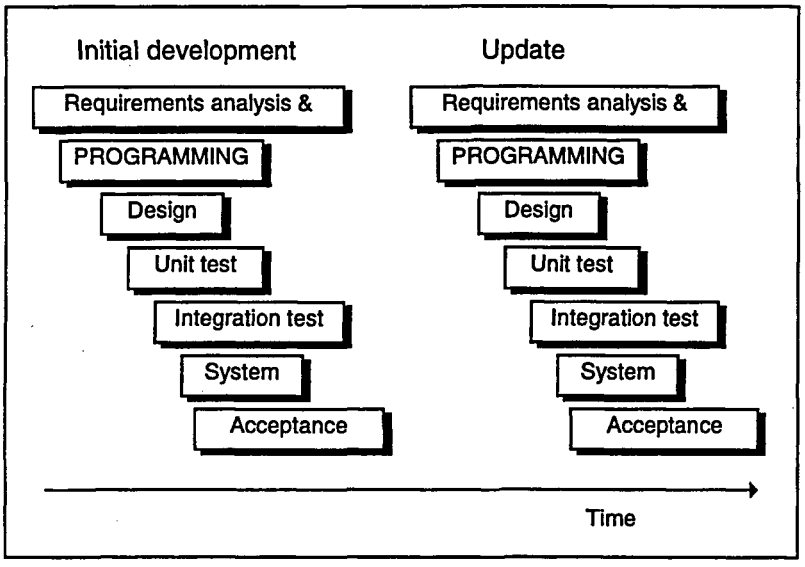

Figure 3: The Rapid Applications Development (RAD)

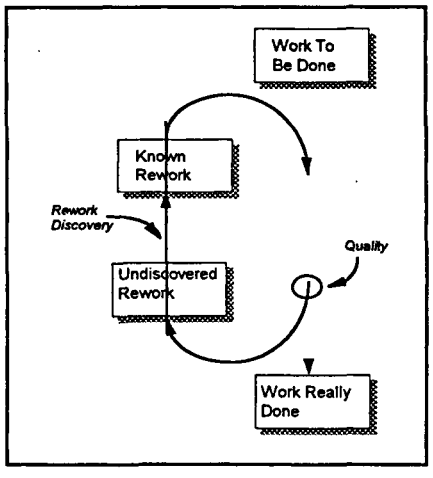

Figure 4: The rework cycle

that effective feedback of a functional and technical nature is obtained. Care must be taken to ensure that the changes to the successive prototypes are managed carefully, and that quality is not damaged by a torrent of amendments.

In traditional developments, much of the documentation produced is done so to facilitate communication between the team members or between the team and outside agencies. In RAD, documentation is primarily produced for the business and operational users of the system. Some key baseline documents are produced within the team but these are brief, factual and instructive rather than huge manuals or specifications. As we shall see later, the absence of a detailed specification is a potential problem for the testing process.

A key aspect is that quality assurance and testing become continual processes throughout the lifetime of a package. Testing can start as soon as the first prototype has taken shape, and the testing procedures then evolve in step with the package itself.

Quality of communication

The evaluation of multimedia packages – in particular of multimedia training packages – has been extensively discussed elsewhere. Comprehensive and authoritative accounts are given by Romiszowski (1984), Rowntree (1992) and Barker and King (1993). In practice, evaluation is honoured more in the breach than in its observance. Evaluation – literally, establishing the value – comes at the end of a project, after the excitement of developing the package and implementing it in the field, after the plaudits, and after the adrenaline has ceased to flow. The team is tired; the project is often over budget; there is the tiresome process of responding to the errors which are starting to be discovered by real users; everyone is looking forward to the next exciting project; and there is little enthusiasm for establishing the value of something that has been completed. Why spend more time and resources on proving what is already known? The package is clearly successful and is delivering real value.

In educational environments enquiry into outcomes, and into how and why they came about, is a legitimate occupation, justified by its contribution to our understanding of communication and cognition, which can be funded and which can yield papers published for the benefit and enlightenment of others. The quality of the evaluations is variable, but the best add value to the multimedia industry as well as valuing its products. It is less common to find examples of evaluation in the business world that go beyond superficial enquiries of users to find our whether they enjoyed the multimedia experience and subjective measures of whether it was helpful to them. Perversely, summative (*post-hoc*) evaluation is more helpful to those developing the package, since it helps them improve the quality and effectiveness of their future products, rather than assuring the quality of work in progress now. If the timescale of the project permits, it may be possible to include some formative evaluation. This aims to establish the value of the development processes and their outcomes at intermediate points throughout the project, the results being used to inform the later stages, so making it possible to improve quality as the work progresses.

A more relevant activity is validation – establishing whether the multimedia package has met its goals in bringing about the correct changes in knowledge, behaviour and attitude in the intended users. If it does not, then it is unfit for the purpose for which it was commissioned and produced. This is a key quality issue.

Desired changes should be set out in advance as part of the quality criteria for the package. They may look similar to the learning objectives advocated by Mager (1975): 'After working through the package the learner will be able to . . .' The problem with such objectives is that they assume a perfect user who is not able to do whatever it is before working through the package and, if the package meets its quality criteria, is able to do it afterwards. Unfortunately, real users are less than perfect, and there are some who will never be prompted by even the most effective package to acquire new knowledge or skills, to change their behaviour or their attitude. We need to express the criteria in statistical terms so as to aggregate the changes in a number of users. As a consequence of using the package:

- '90 per cent of trainees will be able to demonstrate their competence in . . .';
- 'the average time spent retrieving marketing information will fall by 20 per cent';
- 'customer complaints will fall by at least 45 per cent'.

In the long term, success against these criteria can be measured across the whole of the user population by determining the situation now (for example, how many trainees can demonstrate these competences? how long does it take to retrieve marketing information? how many customer complaints do we receive?), and then repeating the measurements when the package is established to determine the change. However, it would be helpful to have an earlier assessment of the changes that the package will bring about, to inform the quality assurance process and acceptance. To do this we need:

- a representative sample of users – a sufficient number to ensure that any changes we measure are not due to random effects;
- a means of establishing their knowledge, skills and behaviour before using the package;
- a means of establishing their knowledge, skills and behaviour after using the package;
- sufficient time to carry out the validation.

Note that a single sample of users will suffice for validation: there is no need for a control group since the aim is to find out whether the desired changes come about, not whether the process is better, faster, cheaper, etc. than an alternative way of doing things.

Care is needed to devise instruments that are themselves reliable and valid. For example, if the purpose of the package is to train users – to help them acquire competences which they do not already have – then some form of competence-based assessment is indicated, rather than written questions that may only test their knowledge. The competences assessed should be those addressed by the package. The Performance Improvement Process developed at PA Management Centre at Sundridge Park (Savage, 1994) asks users – in this case delegates on management development programmes – and their line managers to rate their perception of a user's competences on a 4- or 6-point scale before and after the programme, and again three to six months after the end of the programme. The responses are scored using an optical mark reader and the improvements in perceived competence for individuals and the group as a whole are computed automatically. It should be noted that this technique is based on the *perceptions* that the individual and his/her line manager have about specific competences rather than a more *objective assessment* carried out by a trained

assessor in the workplace; and also that the binary 'does/does not demonstrate competence' is replaced by a 6-point scale. Finally, this approach will not, of course, illuminate unexpected outcomes of learning through multimedia. Although no rigorous research has been carried out to validate the Performance Improvement Process, experience has shown good agreement between the individuals' perceptions and those of their line managers, indicating an acceptable level of reliability for the instrument. Self-assessment by questionnaire reduces the time and resources required to a practicable level. There seems no reason why, with carefully worded questions, this technique should not also be applied to the evaluation of other forms of multimedia communication, providing a systematic means of establishing the quality of the package against pre-determined criteria.

The validation exercise should also look at the usability of the package in question. The development team may have made certain assumptions about the users' familiarity with keyboards, or their levels of knowledge before using the package. If these assumptions are not justified, the users may be unable to operate the package correctly or may be bewildered by the content. It follows that the effectiveness of the package will be significantly compromised.

The RAD process lends itself to the incorporation of validation exercises at each iteration. If the validation process starts early in the project, misconceptions and mis-directions are detected quickly and the results can be used to improve the effectiveness of later work. A mini-evaluation may be used to address part of the whole package, or the exercise may take the form of a workshop where a few key users gather together to work through a prototype and discuss their experiences with members of the development team. The aim is to provide continual assurance that the package will achieve its objectives.

Quality of the software

What are the quality criteria for the software itself? First, it is reasonable to ask that the content be accurate and up to date. Some content is relatively static; other content may date rapidly so that currency can be maintained only by frequent new releases of the multimedia package. As we shall see, frequent updates bring their own inimitable problems. Secondly, it is a basic requirement – often sadly neglected – that there should be no typographical errors. Thirdly, we need to think about the number of software errors that an average user should have to endure while using the package. Let us consider two examples.

1. It may reasonable that the average user of a multimedia reference package should encounter no more than one error during a year of use. If this average user works with the package for one hour each day, this amounts to 220 hours in that year. If one error in a year is acceptable, then for safety, the mean time between software errors or failures (MTBF) should be no less than 500 hours. An MTBF of 220 hours would result in an average of one error per year with some users experiencing more. To reach the criterion of not more than one error per year we should therefore err on the side of caution.

2. Students or trainees working through a multimedia induction programme lasting four hours should have only a 2 per cent probability of encountering an error. This equates to a minimum of 200 hours MTBF.

In both these examples, we can define an error as an event which causes the multimedia package to stop running, or exhibit some behaviour which the user realises is not normal.

It is arguable that these are unreasonably stringent criteria. Those of us who use IT continually in our daily lives have grown accustomed to much higher error rates. I was saddened to hear one colleague suggest that he would find one error per hour acceptable – although on further interrogation he admitted that this was based on his expectations rather than on his aspirations. A selection of published software reviews indicated that reviewers are quite tolerant of errors. Given that they typically only use the review copy for a few hours and encounter one or more errors during that time, we may infer that a MTBF of 10 hours is not uncommon.

I would argue that these more stringent criteria are not unreasonable but are comparable with the errors rates of other job aids. A desktop computer with an MTBF of 500 hours would break down on average every three months, and would be deemed unreliable. The operating system or network probably fails more frequently. The combination fails more often still, because the error rates are additive. Error-prone software just makes the problem worse.

The key is to set appropriate quality criteria and engineer the multimedia package to meet them.

Accurate and up-to-date content

Responsibility for providing the correct content (text, graphics, sound, video, etc.) and ensuring that it is up to date must be clearly allocated. In a large project it can be difficult to keep track of all the content and it may be helpful to set up a database to hold information on each item of content and its status. Typically, the life of each item will follow the cycle shown in Figure 5.

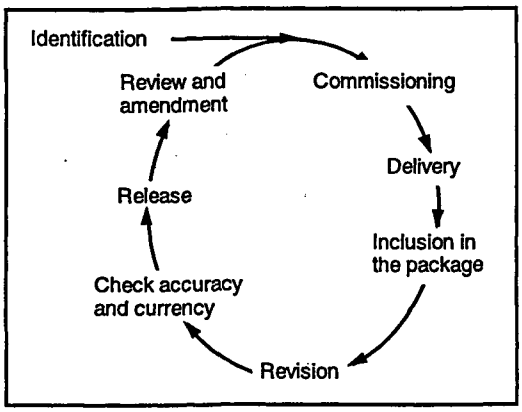

Figure 5: The content cycle

A database can simplify the process of ensuring that all the content is tracked and of producing status reports on specific items. Time invested in maintaining the database and plotting the progress of each item as it moves through the cycle will be more than repaid later in the project, with greater certainty on the status of each item and less rework to replace outdated content.

Because of the difficulty in plotting a simple route through a complex multimedia package that takes in every individual screen, it may be easier for the person (or people) responsible

for content to work from hard copies of the screens (generated from the authoring tool) rather than from the package itself. The hard copy is likely to be in black and white rather than in colour (for reasons of cost) and it may be necessary to carry out a final check with the actual package to ensure that the colours are correct.

Typographical accuracy and consistency

We require a rigorous process to ensure that there are no typographical errors or inconsistencies in the content. Receiving and processing most of the text electronically will tend to reduce the number of residual typographical errors (in contrast, continual retyping tends to increase the number of errors), but there should be a systematic proof-reading stage. This is best carried out by a professional proof-reader who has had no previous involvement with the project and is not compromised by what he or she expects to see. A good proof-reader will also check for consistency of style and layout, and can also be used to comment on grammar, readability, obscure terminology and the inappropriate use of acronyms or jargon. The proof-reader may find it easier to work from hard copy rather than follow a long and systematic route through the package itself.

Software accuracy

The software that drives a multimedia package is complex and intangible. Its complexity makes it prone to errors, and it is difficult to demonstrate the absence of errors in something that cannot easily be inspected.

The traditional testing process involves someone working through the package, usually at the Beta testing stage when the package is essentially complete, and recording any errors encountered. As the number of remaining errors decreases, it takes longer and longer to find each one. Since the psychological reward for testing is finding errors (and thus proving your superiority over the person who wrote the program), motivation decreases, and the testing process ends when the tester decides that the reward does not justify further efforts. This, of course, does not mean that there are no errors left to be discovered.

If the software is rigorously structured, it is possible to carry out all-path validation. This is a process whereby, with a small number of carefully devised tests, every path in the program is exercised and can be validated as correct. It can therefore be demonstrated there are zero errors. However, the hyperlinking that is at the heart of most interesting multimedia programs, and that and adds significantly to its value, compromises that structure. There is an impractically large number of paths to be validated: time and resources do not permit total validation and we must resort to exhaustive testing that still leaves a finite possibility of an undiscovered error. The challenge now is to devise and manage the testing process to achieve the criteria for MTBF within an affordable budget and realistic timescale.

The test schedule

It is reasonable to suppose that the pattern of usage for most multimedia packages follows the Pareto Principle: 90 per cent of the users will only use 10 per cent of the total package. Clearly that 10 per cent requires careful testing, but so does the remaining 90 per cent. Total coverage of every single path through the package may be impractical for the reasons discussed earlier; we thus have to devise a systematic schedule to ensure that:

- every piece of content is visited and checked;
- all the main paths are followed and exhaustively checked;
- there is reasonable justification to assume that any paths which are not to be exhaustively checked do really work correctly (for example, in the figure below, if the software is rigorously structured, and the functionality and content of section A has been exhaustively tested through path α, and the hyperlink path β which goes to A and then returns has also been checked, then we might reasonably assume that A will work correctly through path α without testing the whole of A again);

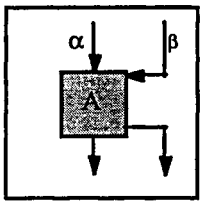

Figure 6: Component validation

- careful attention is paid to areas which are likely to be error-prone (for example, areas that are more than typically complex);
- particular attention is paid to areas where errors have been discovered in the past – we acknowledge that a small number of errors will escape detection and will manifest themselves only once the package is released and used in the field, and when they are found and corrected the package must be re-tested with additional tests that exercise the area that caused the error; those new tests must be retained in the evolving schedule.
- the sequence of steps in the test is documented and followed so that the test is always repeatable, which makes it possible to isolate the circumstances leading up to the error and collect forensic evidence that will identify the cause. Non-repeatable or random errors provide an intellectual challenge to the programmer but are not conducive to quality multimedia packages.

There are four steps in developing the software test schedule, as follows.

1. Identify the key scenarios – the main operations the user will carry out.

2. Develop and document a series of steps that will exercise the package through every part of that scenario.

3. Add perverse actions. Naïve users do not always do what you expect them to, through innocence and inexperience and sometimes through frustration. The schedule should include some totally unexpected and perverse steps (for example, resting a book on top of the mouse so that the mouse input buffer suffers terminal overload).

4. Keep the test schedule up to date. The schedule, which may run to many thousands of steps, is an evolving document. Each time the structure or functionality of the multimedia package changes, the schedule must be amended to include the new and changed scenarios. If an error is discovered and corrected, the schedule should be amended to ensure that this problem area is included for further tests.

The test schedule should be developed by someone who is not a member of the

programming team but who is familiar with the structure of the package and has a good oversight of its intended content and functionality. Separation from the programming team reduces the possibility that the tests will be compromised by assumptions as to what the package does.

The absence of an initial formal specification in an RAD environment creates problems for the test designer because there is no single definitive document describing the functionality and content against which to test the package. If the test designer is working only from a prototype without an overall understanding of the package itself, there is a risk that the tests will be defined by inaccurate code: there is a tendency to assume that 'this is what the package does and is therefore what the package should do'.

The schedule will be a multi-page document setting out a series of steps that instruct the tester what to do and what should happen, with columns to record the success of failure of each step and a description of any unexpected result. An example can be seen in Figure 7.

Package name Test no:		date:		tester:
step	action	expected result	* or x	actual result
1.				
2.				
3.				
etc...				

Figure 7: Section from a test schedule

The final schedule will probably be highly detailed to the point of being obsessive, but the devil of testing is in the detail. For example, an important (but often overlooked) area of testing is to verify that all the hot buttons surrounding icons and hot text are the correct shape and size and are in the correct position. Such attention to detail may not endear the test schedule (or its author) to the programming team!

There is a balance to be struck between the programming team and the QA (Quality Assessment) team. A certain amount of intellectual rivalry sharpens the wits so that the programming team takes greater care with its code and the QA team becomes more penetrating in its tests. But the ultimate aim is to produce a quality package – not to score points against the opposition, and the creative tension must be carefully managed. The continual iterative process inherent in RAD enables the first test schedules to be devised and run very early in the development process. Feedback on errors can start with the first prototype so that the programming team can embark on continual quality improvement. In a traditional waterfall model, the detection and reporting of errors is held back until the final stages of the project when, as we have seen, it is likely to be too late for any real improvement in quality.

The evolving test schedule can be used not only as an assurance of the quality of each prototype but also as a development tool for the programming team to support its own internal testing. And this provides an opportunity for the programming team to suggest improvements to the schedule.

Regression testing

As each prototype is completed it is tested against the evolving test schedule. Almost invariably, there will be discrepancies. These are of various kinds:

- the test schedule may be wrong – the QA team has made incorrect assumptions or straightforward mistakes (it happens);
- the package does not behave as expected and the resolution of the discrepancy is to change the declared functionality of the package (for example, the backtracking through a lengthy hypertext sequence returns to an intermediate point instead of the last screen);
- the package does not behave as expected and clearly needs to be changed; this will include all occurrences of abnormal terminations and situations where the user is bewildered by the outcome.

Because the test sequence is documented step by step, the programming team should be able to repeat the sequence leading up to the error and obtain the forensic evidence it needs to correct it.

Having corrected a batch of errors, the test schedule must then be run again in its entirety – and again, and again until all the discrepancies have been resolved. It is quite possible that, in correcting one identified error, another has been introduced (during the development of IBM's MVS, it was estimated that 20 per cent of the bugs in the software were caused by corrections that failed).

Error management

The traditional testing process yields a stream of errors which eventually decreases until the package is deemed to be 'working'. In contrast, systematic testing against a carefully constructed schedule results in a large number of discrepancies each time the tests are run, until the final tests when no errors are left. The approach is used precisely because its greater effectiveness finds more errors. But the greater rate of error-detection brings its own problems in managing the corrections, integrating the revisions, and ensuring that re-testing is managed efficiently and effectively. There need to be processes to allocate known errors to the appropriate members of the development team, and to co-ordinate the changes they make.

Computer-assisted multimedia testing

The test schedule for a complex piece of multimedia software may run to several hundreds of pages and many thousands of steps. At each step, the tester must check carefully that the package does what is intended, and that what is presented on the screen is correct. One pass through the full test can take several days of meticulous work.

Automated tools are available to assist in the testing of software (see, for example Graham *et al*, 1995). The computer-assisted software testing (CAST) process typically involves:

- identifying the user-scenarios;
- documenting the paths through these scenarios;

- 'teaching' the CAST package to work through the paths and automatically generate a script that will enable the CAST package to replicate the test automatically;
- editing the script to deal with situations not encountered in the teaching phase;
- running an automated test in whole or in part each time the software is changed, to ensure that it meets the quality criteria.

Most CAST packages have been developed for use in a development process that starts with an agreed software specification, and with software that runs in a true Windows environment where the attributes of each window are unambiguous and known. They can bring improved productivity to the testing of software written using, for example, Visual Basic or Visual C. Unfortunately, the RAD process tends to eliminate the need for a formal specification, replacing it with a series of iterative prototypes. There is no longer a definitive list of each window and its attributes that can be used by a CAST package. The starting point for the test schedule is an early (perhaps the first) prototype. This presents a major problem for current CAST packages because there is little or nothing for them to examine. The automated test becomes a trivial pass through the multimedia package, either failing everything or failing nothing.

Many tools also offer facilities for testing software destined for client-server environments, checking the impact of increasing numbers of users, the complexity of interactions, etc. In most cases these are not relevant for the kinds of multimedia were are considering here. PA's experience indicates that, at present, computer-assisted software testing has little to offer for packages that are written using tools such as Authorware and ToolBook rather than Visual Basic.

Conclusion

Multimedia presents a unique set of challenges for developers who aim for zero errors in their work and who are prepared to meet pre-defined criteria for the quality of the packages they produce. However, it is no longer acceptable to tolerate errors and lack of effectiveness. Market forces will favour those who can demonstrate quality in their products: those who cannot are likely to fade quietly into obscurity.

Note

An earlier version of the paper was presented at the PEG Conference in Sozopol, Bulgaria, in May 1997.

References

Barker, P. and King, T. (1993), 'Evaluating interactive multimedia courseware – a methodology', *Computers and Education*, 24 (4), 307–319.

European Commission (1996), *Report of the Task Force Educational Software and Multimedia (Working document of the Commission Services)*, SEC (96), 1426, Brussels.

Graham, D., Herzlich, P., and Morelli, C. (1995), *CAST Report*, London: Cambridge Market Intelligence.

Mager, R. F. (1975), *Preparing Instructional Objectives,* Belmont CA: Fearon Publishers.

Romiszowski, A. J. (1984), *Producing Instructional Systems*, London: Kogan Page, 215ff.

Rowntree, D. (1992), *Exploring open and distance learning*, London: Kogan Page.

Savage, C. (1994), *The Performance Improvement Process: PIP*, Bromley: Sundridge Park Management Centre.