Theory and practice of risk-based testing[1]

Felix Redmill
22 Onslow Gardens
London N10 3JU

ABSTRACT

This paper extends a theory of risk-based test planning that was outlined in a previous paper, introduces techniques to facilitate it, explains their practical application, and highlights opportunities for research to enquire into their feasibility and efficacy.

Keywords: risk-based testing; software testing; test planning; integrity levels; quality factors.

1 INTRODUCTION

In a previous paper (Redmill 2004a) the current author proposed a theory of risk-based testing, and here the theory and its practical application are explained. This paper describes the use of risk as the basis of test planning; the selection of specific testing techniques is beyond its scope.

Risk is a complex subject, but may be considered to be a function of two components: the probability of occurrence of a defined undesirable event and the severity of the event's potential consequences. Risk analysis is summarised in an Australian and New Zealand standard (AS/NZS 1999) and is carried out in four stages:

- Scope definition, in which the context and terms of reference of the analysis are defined;
- Hazard identification, in which methodical exploration is carried out to identify the things that could go wrong (the 'hazards');
- Hazard analysis, in which the identified hazards are analysed to estimate their potential consequences and probabilities of occurrence, and thus the risks that they pose;
- Risk assessment, in which the risks are assessed against tolerability criteria.

The purpose of risk analysis is to inform risk-management decisions. Thus, risk assessment should be followed by decisions and appropriate actions to reduce, or otherwise manage, the risks. Indeed, levels of tolerability may be directly linked to risk-mitigation activities.

Typically, risk analysis addresses both of the components of risk: probability and consequence. In system and software development, consequence values may be estimable, more or less accurately, at all stages, but probability estimates cannot be made until the software is produced. This is recognised in the paper, which proposes three forms of risk analysis on which to base software test planning: single-factor consequence-only analysis is presented in Section 2, single-factor probability-only analysis in Section 3, and two-factor analysis, in which consequence and probability are combined, in Section 4. In each case a methodical process for putting the theory into practice is explained. In all cases it is intended that analysis should meet the objectives of the four stages defined above.

Various standards take different perspectives on test planning. For example, IEEE 829 (IEEE 1998) addresses documentation and BS 7925-2 (BS 1998) considers component testing. But none gives advice on its basis or method. This paper addresses these points, offering opportunities to both academics and practitioners to research the feasibility and effectiveness of its proposals.

----
[1] Published in Software Testing, Verification and Reliability, Vol. 15, No. 1, March 2005

2 SINGLE-FACTOR ANALYSIS: CONSEQUENCE

When software has been developed, it may be possible to derive evidence on which to base estimates of its likelihood of failure, but test planning is properly carried out when no such evidence exists, and only consequence estimates are possible. At that stage a full risk analysis, based on both consequence and probability, is impossible, but a single-factor analysis, based on consequence only, may be conducted. The principle is to identify consequences, focus testing according to their severity, find faults, make corrections and test again, and thus base the reduction of the probability of failure on the risks. This section presents a methodical approach, identifies the issues that need to be addressed, and proposes research to establish the theory and lay the foundations of good practice.

2.1 Consequence identification

The consequences of software failure are different for a system's various stakeholders (Redmill 2004a), so the first step is to identify these. Often the first to come to mind are the system users, who may be inconvenienced by the loss of the system or some of its services. But there are other stakeholders, such as the company that owns the system and to whose objectives the system contributes – and, within the company, the system may provide different services to different departments. The company's customers may also depend directly or indirectly on the system, and developers, testers and maintainers may all suffer losses if it failed. A loss not often included in calculations is 'opportunity costs', which occur if, for example, the developers are precluded from taking on a second project by failing to complete the first on time.

A system provides a number of services, and the failure of each is likely to have different effects on different stakeholders. Indeed, some stakeholders may not use some services at all and would not be affected by their absence. Thus, a refined approach to determining consequences is to address each service with respect to each stakeholder. A tool to facilitate this is a matrix of one against the other (see Figure 1).

	Stake-holder 1	*Stake-holder 2*		*Stake-holder N*
Service 1				
Service 2				
Service N				

Figure 1: A matrix showing the use of services by stakeholders

A service may be defined as the provision of output that has been designed for a particular use. In a control system it may be a control signal, for example to ignite a furnace or open a valve. An advisory or internet service might take the form of a screen of information. A commercial service may be the provision of invoices or account statements. A system usually provides multiple services. For example, as well as providing control signals, a control system may have to accumulate maintenance and operational information and inform management of equipment usage for customer invoicing and historic records.

The cells of the matrix of Figure 1 can be used to store information on service usage. In the simplest case, this may be binary: '1' for use of a service by a stakeholder and '0' for non-use. When estimates of the consequences of service failures have been made, by interview with carefully selected stakeholders and by other means, an expanded version of the matrix may be used for their presentation, and also as the basis of calculations of total consequences. The principle is shown in Figure 2, where the total potential consequences of service failures are shown in the cells at the ends of the rows, the stakeholders' total potential losses are accumulated in the cells at the bottoms of the columns, and the overall total loss is summed in the cell at the bottom-right of the matrix. From this, the most significant services and stakeholders can be identified.

	Stake-holder 1	*Stake-holder 2*		*Stake-holder N*	*TOTALS*
Service 1					
Service 2					
Service N					
TOTALS					

Figure 2: A matrix for presenting all of the potential losses

2.2 Consequence analysis 1: different types of service failure

Considering services produces more refined results than merely addressing total system failures. Yet, even these results may be improved on, for there may be several ways in which each service could fail, and the potential consequences may be different for each. For example, a bank's client may be indifferent to the late arrival of a monthly statement (rendering the failure that caused it of small consequence) but may withdraw goodwill and custom from the bank if it is incorrect. Similarly, total failure of a control system may lead to the automatic, but safe, shutdown of the controlled plant, but an incorrectly timed signal that wrongly closes a valve and prevents the flow of a chemical may lead to an explosion. Thus, considering the various ways in which each service could fail (its failure modes) would produce a more refined process and more accurate results.

An approach to such an investigation is offered by the hazard and operability study (HAZOP) technique, which is widely employed in the analysis of safety risks. Indeed, McDermid and Pumfrey (1994) applied it to the analysis of software-based systems. Important qualities of this are that it is methodical, it is carefully managed, and it employs 'guidewords' to focus the study. In the present context, an effectively moderated team, which should include representatives of principal stakeholders, should study previously drawn-up lists of the system's services by applying a set of guidewords to them.

The principle of HAZOP, as adapted to the present purpose, is as follows. Each service provided by the system possesses attributes, or has requirements imposed on it, that must be met within the bounds imposed by 'design intent'. For example, a bank statement must contain information that must be both up-to-date and accurate. The statement should also be delivered to the account holder on time. Failure to meet any of these criteria constitutes a deviation from design intent, which, in the terms of the current exercise, is a service failure. Guidewords are designed to focus the study on the various types of failure that might occur, and a generic list, from Redmill et al (1999), is given in Table 1. Having been found adequate for safety risk analysis, this set is likely to cover risk-based testing applications, but this hypothesis should be tested by research.

It should be noted, however, that in most cases guidewords require interpretation. For example, 'more' is meaningful when applied to an amount of, say, data, but it may need to be interpreted as 'too long' in the context of a time interval, or 'too high' when applied to a rate of data transmission. Not to make such interpretations would be to assume that the guideword is not applicable and to conclude that a new guideword needs to be invented. Thus, creativity must be employed in planning and carrying out a HAZOP study.

By methodically applying each guideword to each service, in the context of each stakeholder, the effects of each possible (or plausible) type of failure can be identified and calculations of the various potential consequences derived. Entering the results into the matrix of Figure 2 allows the potential consequences of failure of each service, and of the entire system, to be summed, and the most critical services and most significant stakeholders to be identified.

Guidewords	*Definitions*
No	No part of the design intent is achieved
More	A quantitative increase
Less	A quantitative decrease
As well as	All of the design intent, but with something more
Part of	Some of the design intent is correctly achieved
Reverse	The logical opposite of the design intent
Other than	The design intent is not achieved, but something else is
Early	The design intent is achieved early, by clock time
Late	The design intent is achieved late, by clock time
Before	The design intent is achieved early in a sequence
After	The design intent is achieved late in a sequence

Table 1: The generic set of guidewords, with definitions

It is worth mentioning that stakeholders such as developers, testers and maintainers, may not be users of the system's services and are not likely to be included in such studies. A separate study should be conducted to identify the losses incurred by them as a result of their involvement in system or service failures. Care should be taken to ensure that duplication of losses does not occur.

2.3 Consequence analysis 2: tracing the causes of failure

Identifying individual services allows more accurate estimates of failure consequences to be made, but the analysis can be time-consuming. Analysts should determine to what extent it is necessary to distinguish between services rather than to treat all software in the same way. If the system is small, or is for some other reason to be treated as a whole, with no distinction as far as test planning is concerned between its software items, then arriving at a total consequence figure is sufficient. But if the intention is to employ risk-based testing within a system, so as to optimise the testing of its various software items with respect to each other, then their relationships to the system's services must be established.

The services to stakeholders, identified at the interface between a system and its 'outside world', are provided by functions designed into the system, and these are implemented in software. Thus, three types of entity within a system should be recognised: the services, the functions that provide the services, and the software items of which the functions are composed (see Figure 3). The purpose of the present analysis is to determine the influences of the various software items on the different types of service failure, so that their testing can be planned accordingly. For this, the software design should be known. In a relatively small system, a starting point may be to create a matrix of services versus functions and to use ones and zeros to indicate relationships (as in the matrix of Figure 1). For a larger system, the number of individual services may be too great, particularly if each variant of a screen format is considered to be a service. Then, services may need to be grouped together into classes of service.

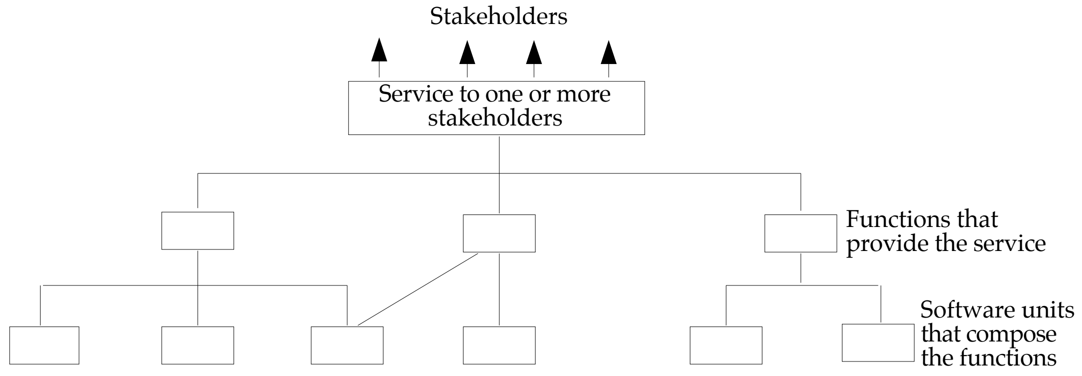
Stakeholders
Service to one or more stakeholders
Functions that provide the service
Software units that compose the functions

Figure 3: Three types and levels of entity

In traditional software design, where a top-down architectural model shows the functional dependencies and a detailed design shows the hierarchical decomposition, these together reveal the relationships between functions and software items. In object-oriented design, a functional model is required. Whatever the technology behind the system design, it needs to be clear which software items compose each function so that a chain can be defined, starting at the interface and working backwards to the software items. From this, potential causal links can be identified between the software items (or, at a higher level, the functions) and the various types of service failure identified in the HAZOP study.

This is a form of fault tree analysis (Vesely et al 1981), with the 'top event' being the service to the outside world. The analysis of each service requires a separate tree. In many cases, all of the software items in a function in a service tree will be considered to be of equal importance; in others, some will be of greater significance than others, for example when a highly consequential failure could only be caused by one particular item.

Care needs to be taken in defining the functional composition of a service. For example, in the simple case of Figure 4, Service S may be defined as comprising functions B, which selects data from a database and sorts them, and A, which further analyses the data and structures them into the information that forms the service. This definition, however, ignores functions D, which derives the data in the first place, perhaps by scanning an external source, and C, which receives the data and files them appropriately in the database. It also ignores the database itself, whose integrity could be crucial to that of the service. Further, there is the security function, E, whose monitoring might protect the service from hackers or other sources of corruption or theft. Because these functions contribute to the provision of Service S, and are potential causes of its failure, they should all be included in its analysis.

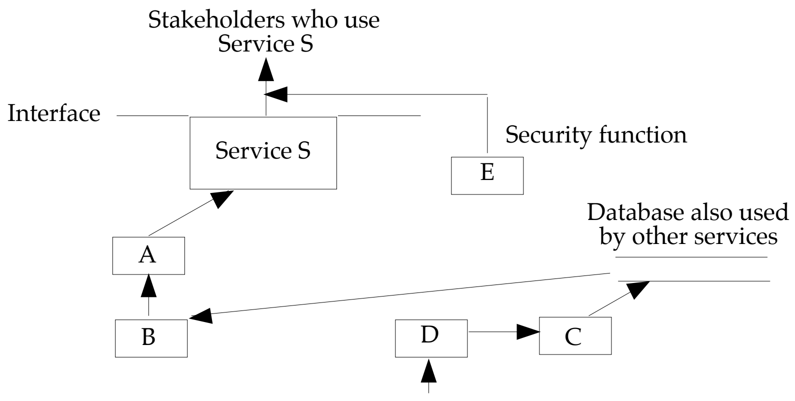
Stakeholders who use Service S
Interface
Service S
Security function
E
Database also used by other services
A
B
D
C

Figure 4: An example of functions on which a service depends

In the example given, functions C and D and the database are defined as contributing to more than one service, so they are potentially the common causes of multiple service failures. The consequence of their own failures must therefore be the sums of the consequences of the failures of the services to which they contribute. To identify the relationships between services and functions, a matrix may be drawn up as in Figure 5. If a

	Service 1	*Service 2*		*Service N*	*TOTALS*
Function A					
Function B					
Function N					
TOTALS					

Figure 5: A matrix for presenting all of the potential losses

zero is entered in a cell to indicate no relationship, and the consequence of the service failure is inserted where a relationship exists, the totals at the ends of rows reveal the summed consequences of the common functions.

2.4 Assessment on the basis of consequence

As seen above, analysis can provide three levels of refinement. The crudest is to treat the system as a whole. Depending on the circumstances, and with more or less attention to detail, a value may be derived for the totality of the consequences of failure of all of the system's services (as in the total of totals in Figure 2), and this may be used to inform risk-based testing of all the system's software. The next level of refinement is to distinguish between the services provided by the system and to use the estimated consequences of their modes of failure to determine their relative importance – from which may be deduced the necessary rigour of testing of their software. The most refined process is to identify the importance of individual functions to the provision of the services and from this to inform the testing of the software items that make up the functions. Whichever process is employed, the result is that software – either all the system's software, or the software in a function – is identified against a consequence value. The next step is to determine how the consequence value is translated into test plans for the software items.

The Motor Industry Software Reliability Association's guidelines (MISRA 1994) offer a simple example of an assessment process. They instruct the suppliers of software-based systems to carry out a simple risk analysis. The first step is to identify all hazards that could result from failures of their system. The second step is to determine the 'controllability category' to which each hazard could lead, five categories being defined, from 'nuisance only' to 'uncontrollable'. The highest category identified defines the system's controllability category, which is directly translated into an 'integrity level', which in turn is interpreted as a definition of the system's acceptable failure rate, stated qualitatively as in Table 2. In step 3 the integrity level is used as a basis for identifying development processes (including testing methods) that are considered appropriate to achieving the desired failure rate.

Controllability Category	*Integrity Level*	*Acceptable Failure Rate*
Uncontrollable	4	Extremely improbable
Difficult to control	3	Very remote
Debilitating	2	Remote
Distracting	1	Unlikely
Nuisance only	0	Reasonably possible

Table 2: Assessment according to the MISRA Guidelines

A given development process cannot be guaranteed to achieve a desired failure rate. However, by adjusting variables such as the testing methods used and the test coverage, a test programme can be designed to be more or less rigorous, and an assumption of testing is that the more rigorous the test plan the better it is at finding bugs. Then, because rigour is also correlated with testing time and cost, it becomes most effective to apply test rigour in proportion to the consequences of failure. That is, the greatest rigour is applied to software items whose failure would cause the most consequential service or system failures – which is the rationale of the current exercise. Thus, defining an acceptable failure rate does not imply confidence in demonstrable achievement, but it provides guidance on what the target should be, which, in turn, enables an appropriate test plan to be devised.

Thus, in the present context the need is to assign a category to each software item and then to use the category to define an appropriate test programme. The starting point must therefore be to define consequence categories. Five categories are used in the MISRA Guidelines, but the choice is arbitrary. However, more categories, suggesting greater refinement, need to be justified by confidence that the accuracy of the consequence estimates allows the implied distinction between categories. In the present example, four categories are defined (see Table 3). Category definition must be system-specific, so, for example, if the maximum loss were 100,000 units of currency, a scheme would be inappropriate if category 3 were equated to losses above that figure and category 4 to losses above a million units, for two categories would be wasted. Further, it is not always the case that there are single units of consequences; there may be a possibility of loss of (say) life as well as of money, and categorisation should address both. So far, the general word 'category' has been used, but, as in the MISRA Guidelines, the term 'integrity level' will be employed as an intermediate stage in translating consequence into risk-reduction activity.

Consequence category	*Integrity level*
> 1 million currency units, or loss of one or more lives	4
> 500K currency units but ≤ 1 million, or serious injury to one or more people	3
> 100K currency units but ≤ 500K	2
≤100K currency units	1

Table 3: An example of consequence categorisation

One of the tasks to be carried out at the 'scope definition' stage of the risk analysis is to define the number of integrity levels and their meanings – in other words, to draw up a table similar to that of Table 3. Then, the analysis and assessment stages should lead to each software item being assigned an integrity level. Test planning then equates each category to a programme of testing, as discussed in the next subsection.

The method employed in the MISRA Guidelines is a sector-specific interpretation of the integrity-level principle, which is a means of categorising risk or consequence values for the purpose of defining risk-reduction or probability-of-failure targets. The principle is expressed in various standards and its interpretations vary. It is defined in a generic standard for safety-critical systems (IEC 2000) but, as shown in Table 3, its use is not confined to safety but may be applied to all forms of risk. Indeed, another standard addresses its use in information technology (ISO/IEC 1998). Reid (2004) offers a brief review of some of its references. However, in this author's view, the simple MISRA Guidelines interpretation provides a suitable model for its use in risk-based testing.

2.5 Risk management decisions

What remains to be done is to plan the risk reduction – which is to say, to design an appropriate test plan. The principle is that testing should be appropriate to the consequence of failure, so the higher the derived integrity level, the higher should be the rigour and thoroughness of testing. A test programme must be planned for each integrity level. The test programmes should be related to each other, with the most rigorous (P4) defined first and the others based on it and being of decreasing rigour. Thus, for four integrity levels:
P3 = P4 minus something;
P2 = P3 minus something;
P1 = P2 minus something.

It is conceivable that, if the distribution of consequences of the various failure modes of a system is not wide, the same methods might be used in testing all categories of software, with the differences in rigour lying in test coverage. In the main, however, differences would also lie in method. For example, the MISRA (1994) Guidelines state testing requirements as in Table 4, with accreditation to ISO 9000 being a requirement at all levels.

What is appropriate in any given circumstances must be determined in test planning, and the requirements of the MISRA Guidelines may not be deemed appropriate to software that is not safety-critical. And, of course, test specifications and test cases need to be defined subsequently, in the light of the detailed designs of the software items. However, the effectiveness of risk-based planning will have been built into the programmes.

Integrity level	*Testing requirements*
0	No requirements stated.
1	Show fitness for purpose. Test all safety requirements. Repeatable test plan.
2	Black box testing.
3	White box module testing – defined coverage. Stress testing against deadlock. Syntactic static analysis.
4	100% white box module testing. 100% requirements testing. 100% integration testing. Semantic static analysis.

Table 4: The MISRA Guidelines' testing requirements

3 SINGLE-FACTOR ANALYSIS: PROBABILITY

Estimates of the probability of system and service failures are not possible prior to the production of software. Thus, in the first place risk-based test planning must be informed by consequence-only analysis – which produces probability targets that are applied to the design of test plans. But testing is not the prime factor in achieving quality, and developers should also use the targets, in design, choice of programmers, control of processes, management, code inspection, and other means (Redmill 2004b).

Once software has been written, probability estimates may be derived (though, not always reliably or with confidence). Is it then feasible to draw conclusions about the probability of failure of a system or its services from estimated probabilities of failure of its software items? To do so would require the probabilities of the various software units to be combined according to the ways in which the units form functions and the ways in which the functions compose the services - which would be difficult, perhaps impossible, and certainly time-consuming. Further, as confidence in the accuracy of the probability estimates, made prior to testing, is likely to be low, confidence in the result of their complex combinations would necessarily be even lower. Thus, the approach here is not to attempt to derive probabilities of system or service failure, but to address only the software units themselves. Analysing them using both risk variables, consequence and probability, is considered in the next section. Here only single-factor analysis, based on probability, is addressed.

In the previous section, consequence was the object of analysis, so focus was placed on the concept of failure because it is out of failure that consequences arise. However, failure is not the only event whose probability it would be useful to deduce. The probability of there being at least one bug in a software unit, or more than a given number, or one that would cause a certain type of failure, would also be of interest. But what is under consideration is newly developed software, as yet untested and with no history of use in its operational environment, without which the direct estimation of such probabilities is impossible. But if it were possible to estimate the quality of a software item, it might be possible to make inferences about the above probabilities. Yet, it might reasonably be assumed that a quality estimate would reflect on all of them, thus making it unnecessary to carry out further translations. Thus, this section addresses the derivation of 'quality factors' and uses them as surrogates for probabilities of failure or of bug infestation.

3.1 Relevant factors and their attributes

In considering quality, two primary factors are relevant, the product itself and the processes of its creation. In addition, there is a subsidiary entity, the product's documentation, which may offer evidence from which to draw inferences about the product and its creation. Thus, indicators of the quality of each of these, considered together, could provide the basis of a judgement on the quality of software. The indicators should come from attributes of the three proposed factors, in the manner suggested by Gilb (1988).

Starting with the quality of the product itself, software may be assessed by testing, performance in operation, reasoning, and analysis. In the present context the first three of these are precluded, for testing has not been carried out, there is as yet no experience of operation, and software developed from formal specifications is not within the scope of this paper. The software may be analysed, however, and three attributes suggest themselves for analysis: the software's structure, comments, and complexity, the last of which may be measured prior to testing using, for example, McCabe's complexity measure (McCabe 1976) or other metrics (Fenton and Pfleeger 1998).

Regarding documentation, four attributes useful as indicators of quality are proposed: structure, readability, accuracy, and completeness. Judgement on any of these necessarily incurs subjectivity, but making a subjective case for the quality of the software is preferable to not making a case at all.

Development processes are equated with software quality by standards such as DO-178B (1992) and the Motor Industry's Guidelines (MISRA 1994) on the assumption that the better the process the better the product. Thus, if test planners could discover which processes had been used in the specification, design and coding of each software item, it would be possible for them to make inferences about their quality. Indeed, if risk-based thinking were employed throughout development, developers would choose processes in accordance with the requirements of the standards, having determined the required integrity levels.

There is evidence, however – although empirical proof is not available – to suggest that the level of care on the part of designers and programmers is a determinant of software quality irrespective of the processes used. So, just as fundamental as the processes are the people involved in them: the development manager, designer, and programmer being most relevant.

	Software Item A	*Software Item B*
Software:		
Structure		
Comments		
Complexity		
Overall software assessment		
Documentation:		
Structure		
Readability		
Accuracy		
Completeness		
Overall documentation assessment		
Development:		
Specification process		
Design process		
Coding process		
Overall process assessment:		
Development personnel:		
Manager		
Designer		
Programmer		
Overall personnel assessment		
Overall assessment (quality factor)		

Figure 6: Categorisation of Product and Process Quality Attributes

The question then is how to use these attributes (or others selected by an analyst) to arrive at estimates of software quality. A categorisation system may be used, and it is proposed that four levels of categorisation would be sufficient. For most attributes the levels could be defined (say) as excellent, good, moderate, and bad. But complexity needs to be categorised between very high and very low, with very low being equivalent to excellent. Thus, for the attributes to be expressed consistently and combined in a single table (see Figure 6), a numeric scale from 1 to 4 is proposed, with 4 representing the best (software structure is excellent, or complexity is very low) and 1 representing the worst (software structure is bad, or complexity is very high).

3.2 The need for sub-attributes

Assessing the proposed product attributes need not be time-consuming, but it requires a methodical approach, thoroughness in exploration, and experience in knowing what to look for. For example, it is necessary not only to ensure that comments are included in the software but also that they are unambiguous, accurate and useful; not only that the documentation is readable but also that it is accurate and complete with respect to the software that it is supposed to describe.

Under time constraints, an analysis may be limited to a direct examination of the attributes proposed above. But, in general, greater refinement would be achieved, and both its objectivity and the chance of consistency between analysts would be increased, if sub-attributes were identified and assessed – except in the case of complexity, whose measurement is made directly by a pre-programmed tool. Then, just as each factor's assessment is a composite of the assessments of its attributes (see Figure 6), so the overall assessment of each attribute is a composite of the assessments of its sub-attributes.

Sub-attributes need to be decided on by practitioners, and a list may be drawn up for repeated use in an organisation. Research is needed to determine their relative value as representatives of their parent attributes. For assessment, an attribute matrix may be created (see Figure 7 as an example), and judgements on the category values (1 – 4, as before) made and entered for each software item. The overall score for each software item of a given attribute is then derived from the various sub-attribute categories and entered into the table in Figure 6.

Attribute and Sub-attributes	*Software Item A*	*Software Item B*
Readability		
Style		
Punctuation		
Sentence length		
OVERALL SCORE		

Figure 7: Quality judgements on the sub-attributes of documentation readability

Sub-attributes of development personnel may include judgements made on the basis of historic data on the quality of software that they have produced, and their experience of their current type of work. Such data is likely to change with every new job and as experience is gained, for example relevant training can have an immediate improving effect, so the database of developer's attributes, which would need to be maintained, should constantly be under review, and the data from it should be used with some caution. However, whereas designers and programmers may change relatively rapidly as they gain knowledge and experience, the development manager is less likely to do so, and his or her effect on other personnel can be extremely influential. A training course that should improve a programmer's ability may have a minimal effect in practice if the development manager inculcates a casual attitude and a quality-free approach. Thus, it may be appropriate to place an added weighting on the manager's influence on quality, and this is a topic that could make interesting research.

3.3 Notes on assessing developers

Whereas the quality checks of the software and its documentation are non-contentious, it is more unusual, and it could be considered provocative, to make, let alone to use, estimates of the development team's quality. Yet, if it is done openly and included in an improvement campaign, with the participants encouraged to admit their shortcomings and to seek to improve on them, it can be achieved. The information could be used to inform test planning and as the basis of feedback to the developers. Such feedback would indicate the need for training, supervision and counselling, and it could be used in the appropriate allocation of developers to tasks, both to achieve quality and for personal development.

At the same time, it may be claimed that assessment of the developers' products – the software and its documentation – is adequate for the purpose in hand and that it would not be worth taking on the problems implicit in maintaining and using information on the developers. This is the case in some organisations. But individuals have considerable influence on the quality of their output, whether it is a product or a service, and their omission from an analysis would almost certainly lead to an underestimate of the risk. It would be useful to research the extent of their influence and the value of using developers as indications of the quality of their software.

3.4 Assessing the information

When the various sub-attributes have been categorised, and the attributes of the software, its documentation and its development derived, an overall assessment (1-4) – the 'quality factor' – must be made of the software item in question. This stage of the process, like those before, cannot avoid subjectivity. It may depend not merely on the absolute categories of the attributes but also on weightings accorded to them. For example, if it were considered that the number of bugs in software increases with complexity, there might be a rule that category-1 complexity would lead to the quality factor being one, whatever the categories of the other attributes. Practitioners must determine a set of rules appropriate to their prevailing circumstances. There is also a need for research to deduce the sensitivity of the quality factor to adjustments in the various attributes.

3.5 Using the information

Each software item's quality factor is used to inform decisions on what risk-management action to take. One approach is for quality factors to be equated to test programmes, as described in Section 2.5. Another is for the worst quality factor (1) to be used as an indicator that the software item carries an intolerable risk and should be rewritten or redesigned, and, thus, returned to the developers without being tested. Or, if the context were of very high criticality, for example safety, it may be decided that software of the lowest two categories of complexity – or of overall quality – would be returned. Criteria need to be defined during test planning.

Returning software for redesign or rewriting would be controversial in many organisations. But if it is a part of the agreement between developers and testers, and its criteria are well defined and understood, it can work well. It places expectations on developers, to which they usually respond. Further, although developers are often loath to rewrite programmes, claiming that it adds time to a project, it can be cost-effective, for the second attempt derives lessons from the first, and savings can be made in debugging and later maintenance.

4 TWO-FACTOR ANALYSIS

Having described the two forms of single-factor analysis, it is possible to carry out a full two-factor risk analysis, and sometimes it may be worth doing so. Given that test plans are initially developed on the basis of consequence-only analysis, it should be used as the basis of tactical adjustments to the plans rather than as a primary form of risk-based testing.

Two approaches suggest themselves. The first is to base action plans on the combination of the previously derived integrity levels and quality factors. The second is to return to the consequences from which the integrity levels were deduced and to translate the quality factors into probabilities of failure and, thus, to carry out a risk analysis in a more recognisable form. Both types of analysis are discussed below. First, however, it must be ensured that the integrity levels and quality factors both apply to the same software items.

The services whose failures are of interest are composed of functions, and integrity levels apply to the software items (at any level of integration) that compose the functions. Quality factors are derived for primary software units only. As already observed, it is unlikely that quality factors could with confidence be extended forward to reveal reliable estimates of the probability of service failures. So integrated software items must be decomposed into their primary units, and the derived integrity levels applied to these. Two-factor analysis, as described here, is only recommended for individual software items at the unit-testing stage.

4.1 Combining integrity levels and quality factors

An integrity level defines the target acceptable failure rate of a software item, and it may be translated into appropriate development and testing processes. A quality factor may at first sight be taken for an indicator of the extent to which the software meets the target probability of failure. However, this would suggest that a high quality factor justifies a reduction in planned testing, or even no testing at all, which would be contrary to good practice. Software should always be expected to be of high quality, and testing is not a means of making it so but of verifying whether or not it is. The higher the integrity level, the more rigorous the test programme needs to be if verification is to provide the necessary level of confidence. A high quality factor should not negate testing.

An indication that the software is of a lower than acceptable quality is a warning of future problems, such as a high fault rate, high maintenance costs, a great deal of inconvenience, and difficulty of change, among others. Then the return of the software without testing it, for remedial work and perhaps for complete rewriting or redesign, would not only reduce future costs but also save on the added expense of a test programme made more difficult and time-consuming by the low quality of the software. The quality factor may be used as an indicator of basic adequacy, offering confidence to proceed (or not) with testing.

The use of two-factor analysis as a means of deriving the level of confidence, and of defining appropriate actions, is achieved by employing a 'confidence matrix' and 'confidence classes'. At the scope-definition stage of the analysis, the categories of integrity levels and quality factors (in the present case, four of each) should be defined. When two-factor analysis is carried out, the software items under consideration are positioned, according to their own integrity levels and quality factors, in the cells of a confidence matrix, as in Figure 8. This is modelled on a 'risk matrix', of probability versus consequence, in which each cell represents a risk value, but in the present case the cells designate not risk values but confidence in the software items, as designated by appropriate confidence classes.

A risk class is a category of risk tolerability, and it also defines the risk-management action to be taken. Its derivative, as proposed here, is a confidence class that performs a similar purpose. In planning the analysis, the number of confidence classes and their associated test-management activities must be decided. At that time, too, judgement is used to populate a copy of the confidence matrix with them, thus creating a generic confidence-class matrix, as in Figure 9. In the present context, the issue is not the choice of a test programme, but what confidence there is in the software to be tested. When confidence is very low (when the quality factor is 1) the most appropriate action may be to return the software to its developers for improvement, thus saving not only on testing but also on the costs of future

	4		Item 3		
Integrity levels	3	Item 2			
	2	Item 4			Item 1
	1				
		1	2	3	4
			Quality factors		

Figure 8: A 'confidence matrix', an indicator of confidence in software items

maintenance and change. When confidence is acceptably high, it is appropriate to test as previously planned, based on consequence-only analysis. And when confidence is very high and the integrity level is not the most demanding, it may be appropriate to permit a reduction in the planned test programme in exceptional circumstances (such as when time is running out). Three risk classes, A, B and C, and their associated test-management activities may thus be summarised:
A: return the software to the developers to be re-written, or for defined quality attributes to be improved;
B: apply the full test programme as planned;
C: the planned test programme may be reduced under exceptional circumstances.

	4	A	A	B	B
Integrity levels	3	A	B	B	C
	2	A	B	C	C
	1	A	C	C	C
		1	2	3	4
			Quality factors		

Figure 9: An example confidence-class matrix

4.2 Returning to consequence and probability

Risk analysis customarily involves probability and consequence. If it is desired to use these as the dimensions of two-factor analysis, integrity levels must be translated back into consequences, and probabilities of failure must be derived from quality factors. In making the conversions, the chosen numbers of consequence and probability categories must equal the numbers of integrity levels and quality factors respectively (four in each case in the examples in this paper). For integrity levels the translation is straightforward: high consequence is equated to high integrity level (see Table 5). But the probability of failure is related inversely to quality factor, i.e. software with a low quality factor may be assumed to have a high probability of failure – as reflected in the table.

Integrity level	*Consequence category*	*Quality factor*	*Probability of failure*
1	1	1	4
2	2	2	3
3	3	3	2
4	4	4	1

Table 5: Translations of integrity level into consequence category and quality factor into probability of failure

The categories must be defined at the scope-definition stage of the analysis, and a risk matrix produced. The principle is the same as that shown in Figure 8, except that the rows and columns of the matrix represent consequence and probability categories rather than integrity levels and quality factors. Similarly, the assessment process is the same as that described above. Risk classes (rather than confidence classes), prepared at the planning stage and used to populate a risk-class matrix similar to that of Figure 9, are used to define appropriate actions.

Of the two forms of two-factor analysis, the first, using integrity levels and quality factors, is recommended, for two main reasons. Firstly, the translation of quality factor to probability of failure is unreliable. Secondly, a matrix of consequence versus probability of failure can give the misleading impression that its cells represent genuine risk values, whereas the derivation of probability from quality factors does not allow this. The concept of confidence is more appropriate to the current circumstances than that of risk.

5 DISCUSSION

Risk-based testing, as presented in this paper, is founded on risk-analysis principles, and its practice employs techniques that have been proven in other fields. The analysis described in Section 2, based on the consequences of various types of failure, allows target probabilities of software failure to be set, and these may be used to inform the focus and rigour of testing. Consequence-only analysis provides a strategic means of test planning, as it can be carried out at software's design stage and can be applied to all stages of testing. The two forms of analysis described in Sections 3 and 4, which involve making quality estimates of the software after it has been produced, facilitate tactical adjustments to test plans and are appropriate to testers rather than, or as well as, test planners. When time is short, or software has been delivered late, and the planned test programme cannot be executed in full, probability- or quality-based analysis offers risk-based guidance on where it is most and least important to exercise the software. However, the rules of operation and the risk-management options should be defined during test planning.

Whereas this paper is concerned with the use of risk only for test planning, a risk-based approach would beneficially be taken throughout the development process. Indeed, this is proposed in an earlier paper (Redmill 2004b) that shows its use in the context of the V model (STARTS 1984).

The citing of risk-based testing in a test-practitioner syllabus as a subject for study and examination (ISEB 2001) places a requirement for a well-defined theory to underpin the subject, and for a body of knowledge that tutors can confidently teach. The content of this paper makes an initial proposal and offers an opportunity for research into its feasibility and efficacy. Researchers should not, however, be limited to trying what is proposed, but should modify and extend it so as to iterate towards a basis for testing that is more effective than current practice.

There should be a distinction between research into risk-based testing in absolute terms – that is to say, trials that attempt to assess its merits *per se* – and that to compare it with other forms of test planning. Research in absolute terms seeks answers to questions such as: is this or that aspect of the theory practicable? How easy, quick, or cost-effective is it? What problems occur when the theory is applied, and how can they be overcome? Does the type or size of a project impose constraints on the efficacy or ease of application? Are there development technologies for which it is inappropriate? Answers to such questions may be provided not only by academic investigators but also informed experimental practitioners, and there is a need for the latter to make trials and publish their results.

Research in relative terms may encounter problems in identifying proper bases of comparison. For one thing, risk-based test planning is likely to lead to test programmes that would not otherwise have been produced, so comparisons of such variables as testing cost and time, and failure rates and costs subsequent to testing, could be misleading. For another thing, it is possible to do a good job of identifying and analysing the risks but then to do a bad job of designing test plans, so comparing only test results could also be misleading. Researchers therefore need to take care when defining benchmarks in their experiments. Longer-term results, such as the costs and times of testing programmes, taken in conjunction with the numbers of failures and the costs of maintenance over the first year or years of a system's operational life, may provide more meaningful comparisons. When effectiveness rather than efficiency is the subject, a business perspective is appropriate.

Risk-based testing as described in this paper is a methodical process. It demands information. Indeed, the collection of information is a necessary and integral part of the process. One of its propositions is that intuition alone is inadequate and that a methodical approach is essential if risks are properly to be identified and analysed. If, to start with, nothing is known about a software system, an attempt must be made to find out something, at least about the consequences of its failure. Without this, risk-based testing cannot be applied, for there would be no basis for understanding the risks.

REFERENCES

AS/NZS (1999). *AS/NZS 4360, Australian/New Zealand Standard on Risk Management (4360:1999).* Standards Australia, 1999

BS (1998). *BS 7925-2-1998, Software Component Testing*. British Standards Institution

DO-178B (1992). *Software Considerations in Airborne Systems and Equipment Certification*. EUROCAE, Paris, 1992

Fenton N E and Pfleeger S L (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS ISBN (0534-95429-1), 1998

Gilb T (1988). *Principles of Software Engineering Management*. Addison-Wesley, Wokingham, UK, 1988

IEC (2000). *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems* (standard in seven parts). International Electrotechnical Commission, Geneva

IEEE (1998). *IEEE Std. 829 – 1998, Standard for Software Test Documentation*. Institute of Electrical and Electronic Engineers

ISEB (2001). Information Systems Examinations Board. *Practitioner Certificate in Software Testing - Guidelines and Syllabus. Version 1.1*. British Computer Society, Swindon, 2001

ISO/IEC (1998). *ISO/IEC 15026: 1998, Information Technology – System and software integrity levels*. International Standardisation Organisation, Geneva

McCabe T J (1976). A complexity measure. *IEEE Trans. on Software Engineering*, SE-2 (4), 308-320

McDermid J A and Pumfrey D J (1994). A Development of Hazard Analysis to Aid Software Design. *COMPASS '94: Proceedings of the Ninth Annual Conference on Computer Assessment, Gaithersburg, MD*. IEEE

MISRA (1994) . *Development Guidelines for Vehicle Based Software*. The Motor Industry Software Reliability Association, UK, 1994

Redmill F (2004a). Exploring risk-based testing and its implications. *Software Testing, Verification and Reliability* 14, 3-15, 2004

Redmill F (2004b). Risk-based test planning during system development. *Proc. of 6th National Software Engineering Conference, 6-8 Oct 2004, Gdansk, Poland*. Wydawnictwa Naukowo-Techniczne, 2004

Redmill F, Chudleigh M and Catmur J (1999). *System Safety: HAZOP and Software HAZOP*. John Wiley & Sons, 1999

Reid S C (2004). Software Testing Standards. *Jornada sobre Testeo de Software (JTS 2004), Valencia, 25 - 26th March 2004*

STARTS (1984). *The STARTS Guide*. NCC Publications, Manchester, 1984

Vesely W E, Goldberg F F, Roberts N H and Haasl D F (1981). *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Washington D.C., 1981