

Using risk as the basis of test planning

Consultant and trainer **Felix Redmill** says there is such a thing as tolerable level of failure

If we have learned one thing in software development, it is that we must get the specification right. More project and system problems arise from the gaps, inaccuracies, and ambiguities in specifications than from any other cause. Yet, specification problems recur in project after project. The fact is that producing and maintaining good specifications is not trivial. Acquiring the necessary information is difficult, analysing it is a particularly problematic task that is often omitted, verifying the results is tedious, and expressing the requirements in clear, grammatically-correct language is impossible for many.

The test of a lesson being learned is evidence of change in the thinking process as well as in the way of doing, and the recurring lesson should by now have influenced our approach to everything. Yet, not only do deficient specifications continue to plague projects, but casual observation suggests that software engineers have not transferred the lesson into their own lives. They seem to take no more care in specifying the requirements for meetings with friends, journeys through unfamiliar parts of the country, and picnics for their children, than do other members of society who have not had the benefit of their salutary experiences.

Perhaps the preparation of specifications, and planning in general, do not come naturally to us. It seems that our heuristic approach is to identify a goal and to take intuitively defined steps towards it, rather than to plan the route in detail. In life we are usually oblivious to the inefficiency of this strategy because, in the main, it seems to work, and when it doesn't, the consequences are mostly small and not far-reaching. For example, having not clearly specified their meeting place, two people wait for each other in different parts of a train station, but they laugh it off later; a traveller doesn't plan his journey and becomes lost, but arrives only an hour late and is relieved at the outcome; the ice and sandwiches are forgotten but the picnic goes ahead anyway, and everyone enjoys the cake and says that the drinks were really cool enough.

In simple situations, intuition gets us by, inefficiently but mostly without catastrophe.

But in development projects and other non-trivial endeavours, intuition is not enough. In well managed projects, the fact that specification and planning do not come naturally is compensated for by the provision of training, structured methods of working, quality assurance, and other means. Because planning needs to be based on a specification, difficulty in planning is often an indication of specification deficiencies. Yet, the opportunity that this presents for specification clarification or improvement is frequently ignored, with useless or misleading plans being produced rather than specifications being revised.

The importance of specification is as great in testing as at other stages of software development projects. Without clarity of what is required, test planners cannot carry out their task with confidence; they cannot plan the most effective testing. Yet in many (perhaps in most) instances, vague specifications are accepted without challenge. The culture of testers seems to be to accept the inadequacies of others rather than to challenge them, even though the quality of their own work is compromised.

This point was made by the writer of a letter published in a recent issue of *Professional Tester* (issue 16, October 2003). The writer claimed to be confused, having read 'some amazingly ridiculous things', and provided two examples of them. The first was 'the requirement is for 80% statement coverage', and the letter writer asked which 20% of the statements do not need to be tested, and whether this could be thrown away and a 20% discount obtained on the cost of the application. The second example was, 'a package could make do with a medium level software quality', in response to which the letter writer asked how much is saved by accepting medium rather than high quality, and what additional saving would be made by settling for low quality. Both the original expert's statement and the letter writer's question assume a correlation between quality and cost, but it is worth noting that inferior programmers and slack management are almost certain to produce very costly low-quality software!

The letter writer called on experts to desist from making such statements and to provide a

sound basis for test planning. The trouble is, though, that this is hard to come by, and the silence of testers suggests that it isn't thought to be needed. There isn't a universally agreed basis for test planning, and it doesn't seem to be missed. Perhaps a part of the cause lies in the fact that, although students are sometimes (but not always) taught testing, they are almost never taught how to plan it.

Returning to the quoted statements, are they indeed amazingly ridiculous, or are they founded on some element of good sense but just inadequately expressed? If the latter, why do the experts who write them use forms of expression that first baffle and then amuse readers, instead of creating statements that ring true? They raise doubts about authorship and editing in the field of testing, and about expertise and professionalism too. They also raise questions about test planners and testers who, although admitting to not knowing how to interpret the statements, seldom question them. But is there some rationale behind such statements? If the authors understood it, surely they would express it and not leave the statements apparently groundless.

Without justifying their conclusions, experts define requirements for 80% statement coverage rather than, say, 40% or 90%, and for "medium-quality" rather than "high-" or "low-quality" software. Perhaps they sense something that they don't understand. If testing is time-consuming and costly, and cannot be carried out to the same extent on all software, then preference might usefully be given to the software that matters most. Risk is implicated. Indeed, it is often mentioned. But how it forms the basis of test planning is not made clear. Yet, if the risks arising from system or software failures were understood – identified, analysed and understood, and not merely guessed at – it would be possible for stakeholders to use them as a basis for deciding what likelihood of failure they were prepared to tolerate from a system, or from particular subsystems. How such decisions may be translated into test plans is the subject of the remainder of this article.

Risk is a function of two variables: the probability that a defined undesirable event

will occur, and the potential consequences if it did. In the context of software that has not yet been tested, it is possible to determine failure's consequences but not easy to estimate its probability. Thus, although we speak of 'risk', in the present context, the single factor, consequence, is proposed as the basis of test planning. The consequences are likely to be different for the various stakeholders. They may involve financial loss, safety or security breaches, loss of goodwill, or mere inconvenience, and each of these may be more or less serious, depending on the circumstances. Their identification, including its difficulties, has been addressed in previous articles in this column and will not be explained here. In some cases, the consequences of failure may be assessed for the system as a whole; in others, it may be possible to determine them for subsystems - but this needs to be done with caution, for the effects of interactions between software items can easily be overlooked. But whatever the reference point, if the consequences are categorised according to severity in the context of the system's use, the categories may be used to inform test planning.

Suppose we define four consequence categories, or classes, Class 1 being trivial, Class 4 being catastrophic, and Classes 2 and 3 forming categories between the two. The software in question would then be accorded the class of its highest-class consequence. So we know one of the components (consequence) of risk but not the other (probability). We also know that we can reduce risk by reducing either the consequence or the probability of a given type of failure. Assuming the potential consequences in each case to be fixed, because they are linked to the system's objectives, risk reduction must be achieved by reducing the probability of failure. It is true that failure may be caused by operator error, the probability of which should be reduced by design, documentation, training, supervision, and other means, but these issues are beyond the scope of this article. What we are concerned with here is reducing the probability of failure due to software faults - by reducing the number of faults and by finding and removing those that would result in the greatest consequences. (Of course, it would be preferable to produce better software in the first place, but that is another matter. The subject now is testing rather than development.)

The logic that we are following leads to the proposition that testing would be most effective if Class 4 software were tested more thoroughly than Class 3 software, Class 3 more thoroughly than Class 2, and so on. In other words, testing should be risk-based. But the process depends on the classes being predicated on properly identified and analysed risks and not supposition. Nor should it stop there. It requires the methodical development of test programmes, one for each software class. Naturally, test cases must be designed specifically for individual items of software, but the general plan, based on the context of

system use, and also on the intention to put most effort into trapping the faults with the greatest potential consequences, should be designed such that there is progression from the basic test level to the most rigorous, ie:

Test programme 3 = Test programme 4 - something;

Test programme 2 = Test programme 3 - something;

Test programme 1 = Test programme 2 - something.

Test programme 4 needs to be designed for providing confidence that the probability of failure of Class 4 software is reduced to a tolerable level. For example, in the context of safety-critical systems, it may include substantial static analysis as well as formal proof of correctness with respect to a mathematically based specification. Other test programmes are then reductions on programme 4. Test programme 1 may in some cases call for no more than a level of black-box testing - and it could include a requirement for '80% statement coverage'. But here the basis for making it is clearly stated, and the requirement is seen to be not ridiculous but a purposeful attempt to achieve cost-effective testing appropriate to the circumstances.

But what is a tolerable level of failure, and how can we know when we have achieved it? By judgement (which is always subjective and always needs to be justified) we can define a failure rate that we would consider tolerable in the circumstances. A continuous-operation example is, 'no more than one failure per ten thousand hours of operation', and an on-demand example is, 'no more than one failure per thousand uses of the application'. But we cannot prove achievement in advance. We can never be certain. We must seek to increase confidence, and we do this by increasing the rigour of testing in proportion to the judged acceptable failure rate (based on the severity of the consequences of failure).

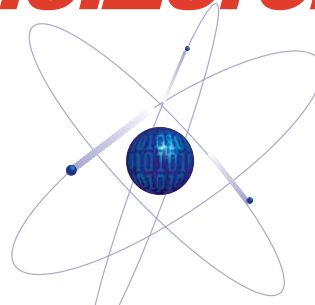
This approach relies on the intuitively plausible assumption that a more rigorous test process achieves better results. However, finding bugs depends not only on the test plan but also on the way in which it is implemented, and on whether the faults found are corrected and the software re-tested. Importantly, it depends on the designers, programmers, and their managers. If sows' ears

are produced in the first place, good testing will not make silk purses of them.

If the version of risk-based testing described here seems a good idea, then let us not ignore the assumptions made in executing it. Determining the various consequences of failure can be difficult, and we can easily overlook or underestimate some of them. Further, linking consequences to individual subsystems through a cause-and-effect chain, based on top-down decomposition, may implicitly assume the independence of subsystems from each other, but, unless design has included partitioning, this is unlikely to be valid. Then, the confidence that we derive from our testing is strongly linked to our starting assumptions. Although it is not difficult to devise test plans and cases, we cannot be sure that they will uncover the faults that need to be found, and our final confidence that they have done so is related to our initial assumption that they will. Notwithstanding these and other assumptions that we might make, the approach is methodical; it is preferable to basing testing only on intuition.

Returning to the statements that our letter writer referred to as ridiculous, it is now apparent that their use implies an intuitive recognition that testing is most cost-effective when based on risk. But intuitive recognition does not amount to understanding. The statements are not couched in terms of risk and they do not escape ridicule. We need an understanding of the subject of risk so that risk-based testing can be well done. PT

GUI testing with no scripts. *None.Zero.Zip.*



MITS.GUI & MITS.Comm

Technology totally unique on earth, in the universe, and . . . well actually we're making assumptions beyond our solar system, but you get the idea.

OMSPHERE, for the inside track in an upside down world.

OMSPHERE

San Francisco CA 94013 USA www.omsphere.com
phone (+US) 415 439 5272 fax (+US) 360 397 7221