Automated test case generation
Since its inception, IT has been about automating business processes. However, it has been slow to apply that concept to itself.

Author Philip Howard
Executive summary

Application software that does not do what it is supposed to do costs money.

Not only does it cost money in terms of re-work but it can also adversely impact morale if for internal use and can adversely affect your reputation if exposed to public gaze. In the latter case, it can directly hit your bottom line if, for example, you have to take down your website while the problem is fixed (cf. National Westminster Bank: according to the Daily Telegraph “in June 2012, a failed software update locked millions out of their accounts for up to a month”; and, more recently, the £3m fine incurred by EDF when, to quote the Times: “a new system for handling calls went into meltdown”).

The reason why web sites have to be taken down and why application software does not do what it is supposed to do is because there are defects in the software. These defects should have been detected and fixed before the application was allowed to go live. Why didn’t that happen? It didn’t happen because the software was not tested properly. Why wasn’t it tested properly? Typically, the answer to this question is twofold. In the first place, development is under pressure from the business to get the new application up and running as soon as possible—ignoring the old saying about more haste, less speed; and, secondly, because of budget constraints in the IT department, which means that it is easier to hire more (temporary) testers to do a rush job than to get approval to license the sorts of tools that would make the whole process of testing more efficient and accurate, and which would automate a significant part of the testing lifecycle.

In this paper we are not focusing on testing per se but rather on automating the testing process. It is there where we believe that the greatest savings and efficiencies can be achieved.
Introduction

Let us start at the beginning. Testing, whether of software, a system or some other form of product, is intended to ensure that, whatever it is, it does what it is supposed to do.

This includes whether it functions correctly, whether it functions fast enough and whether it scales (that is, it won’t crash when multiple people are using it) sufficiently. The last two of these are referred to as performance testing and stress testing respectively. These are outside the scope of this paper, which is focused on software testing to ensure that the software or system does exactly what it is supposed to do. In other words, that it is actually doing something useful and that it meets business needs.

The concept behind software testing is simple. In theory, you test every single path the software can run through, with every possible combination of data values. If the software does what it is supposed to in every single instance then the software is ‘fit for purpose’ and if it doesn’t it isn’t. Unfortunately the world is not like that and 100% testing is rarely achieved in practice, partly because of budget and time constraints and partly because, historically at least, relevant tools have not been up to the job.

The initial problem is twofold: firstly, identifying all the relevant paths through the software and, secondly, identifying representative datasets to check those paths against. Any one combination of these is referred to as a test case. Of course, you want to only test the data against the paths that data might go down. In other words you want to minimise the amount of testing while maximising the coverage (how many of the paths and data are tested). In essence you want to remove redundant and unnecessary testing.

There is also a follow-up issue. Requirements are not generally static. Users who define those requirements are apt to change their minds or refine their needs during the development process, especially in agile environments that have been designed to cater to that fact. However, when changes are made it is important (because it costs time and money) not to re-test things that you don’t need to re-test: you really only want to be test what has been affected by that change. Again, you don’t want redundant testing.

So, let’s look at what can potentially go wrong and prevent software from doing what it is supposed to do. The major risks associated with software development are:

1. The software does not meet the requirements of the business: because testing has not sufficiently reflected those requirements or because the requirements were not sufficiently well defined so as to be amenable to proper testing.

2. The software in theory meets business requirements but is so bug-ridden as to be unusable: typically because testing has been inadequate, or has tested the wrong things, or both.

3. Budget issues: testing, even when tools are deployed, is a resource-intensive process and many organisations will have budget constraints with respect to how much testing they can do. However, these budgets are effectively reduced by redundant testing (multiple test cases that test the same thing), over-testing (where you test the same thing repeatedly, when you didn’t need to: this is common in agile environments) and unnecessary testing where you are testing things that are irrelevant simply because that is one of the test cases you always use (“test cases never die”, which also imposes a management overhead).

We would like to emphasise this last point. In many organisations, testing represents as much as half of all
development costs. We regard this as far too high by a factor of at least 3 and as much 5 or 10. Testing, with the possible exception of critical environments such as nuclear power plants or aircraft or space stations, really shouldn’t take more than 5 to 15% of the total development budget. The reason for this overspend is partly because of over-testing the wrong things, partly because of under-testing the right things, and probably most of all because it is often easier to get budget to employ some additional testers on a temporary basis than it is to license proper tooling in the first place. Even today, some 70% of all testing is manual. We suggest that the high cost of testing is correlated with the low rate of investment in automation.

In any case, we can say that:

1. Requirements need to be sufficiently rigorous for meaningful test cases to be generated.
2. Those test cases should be minimal in number while providing maximum coverage.
3. Test cases should be linked to requirements in a ‘where used’ fashion so that changes to requirements again minimise the number of test cases that need to be re-tested.

It is our contention in this paper that the best way of achieving these goals is by automation of the link between requirements and testing. However, there is more than one way to skin this particular cat and we need to discuss how different approaches work.
Linking requirements to testing

In order to link requirements to the generation of test cases so that the latter process is automated means that the definition of the requirements must be sufficiently rigorous, both to capture what the business requires and to enable test case generation. We need to think of these as separate steps.

Defining requirements

There are two ways to define anything: you can use words or you can use some sort of model. The former is much better for abstract concepts but models are useful when you wish to be precise. We would suggest that a specification of requirements for computer software is an area requiring precision. We do not think anyone would argue with that. Therefore both words and pictures are in the frame.

The advantage of using words to define something is that they are easy to understand. However, misconceptions and misunderstandings can occur unless the words used have precise meanings. For example, what exactly do you mean by “profit”? Thus semantics are important, as is grammar (think “eats shoots and leaves” versus “eats, shoots and leaves”). Without going into detail it should be obvious that defining requirements in terms of vocabulary is not trivial: it means that you need to actually learn (or re-learn) terminology and vocabulary, which is complex and time consuming.

In the case of application code the most obvious model-based approach is to use a flow chart or, rather, multiple flow charts, where each flow chart represents a specific process or sub-process. After all, a program is effectively a series of ‘if this, then that’ operations, which can be easily captured in a flow chart. In fact, you will probably also need a hierarchy of flow charts that defines how the different flow charts fit together. There is still going to be an element of semantics involved because, for example, decision boxes need labels. However, these labels are in bite sized chunks which, we would suggest, are more digestible than a full text-based approach. In particular, when developers go through requirements with the relevant users, we believe in the old adage about a thousand words: we think flow charts will be easier for users to follow, especially if the requirements document is extensive. In addition, it is easier to check the veracity and correctness of requirements when using flow charts because you can apply mathematical (predicate analysis) and/or use case review processes to the flow charts. In the first instance, this simply wouldn’t apply to purely semantic approaches and, in the second instance, again, use cases should map directly to particular paths through the flow chart.

Requirements-based testing

We have deliberately not referred to requirements-based testing (RBT) in this document because there is some suspicion about RBT. In particular, there are multiple vendors in the RBT space, where there are almost as many views about how to provide RBT as there are suppliers. Moreover, some evangelists in this space could perhaps better be described as fundamentalists who believe that their approach is right and everybody else’s is wrong. Not only do we disagree but we also believe that such views have helped to put people off RBT.

The problem with requirements-based testing is not to do with the particular approach used but, rather, with the accuracy of the requirements themselves. Specifically, the problem is twofold: how do you ensure that the requirements are correct and that nothing is missing and how do you ensure that there aren’t requirements that are incorrect or that are unnecessary? The truth is that you can’t guarantee either of these things but that doesn’t mean, as some anti-RBT developers maintain, that you should throw the baby out with the bathwater.

The problem lies in the fact that when users specify requirements these are frequently not complete, for entirely understandable reasons: people aren’t perfect. What is required is an approach that makes it as easy as
possible to capture as much as possible, as accurately as possible and to ensure that as little as possible is missed. In particular, requirements need to be unambiguous. Ambiguity can easily lead to multiple interpretations, with the direct implication that what gets developed isn’t what was intended. In other words quality needs to be built into the project at the outset: low quality (ambiguous) requirements means low quality software. As we have already stated, Bloor Research believes that flow charts, for most people, are easier to follow and check for completeness, accuracy and ambiguity than an approach based solely on semantics or, worse, one that simply uses either conventional documents and/or spreadsheets.

Nevertheless, however good the initial requirements capture process is, and however precisely requirements are defined, the initial requirements specification will never be the same, at least for any reasonably complex application, as what is eventually delivered. Therefore the process used for capturing requirements must be flexible enough to easily allow for changes to be made. Consider how this would be done if using a text-based approach: essentially it would be the same as turning tracking on in a Word document, highlighting insertions and deletions. Conversely, when using flow chart models, new or changed elements can be identified by means of colour coding. In our opinion the latter approach is more easily understood because you can see changes at a glance rather than having to examine text.

Generating test cases
If you want to generate test cases from requirements—as opposed to developing them by hand (a costly and necessarily error-strewn process)—then requirements need to be formal, either using words (semantics) or models (flow charts). Provided that the results are sufficiently well defined then either of these approaches can be, and are, used to automate the production of relevant test cases. This is a major benefit: it means that test cases accurately reflect requirements and should mean that the number of test cases generated are precisely tailored to requirements: no more or less testing than you need.

However, there are two further points to consider. The first is that test cases are not solely related to process flow but also to the data that flows through them. It will be helpful if test case generators also have some sort of understanding of the data to be tested in conjunction with relevant processes. For example, if a particular field has a maximum value then you should only need to test one out of bounds value, not multiple such values.

Secondly, not all test cases have equal value. Some are testing processes and activities that occur on a regular basis, while others will test events that occur only very rarely. Ideally, you therefore want to be able to prioritise test cases, for which you will need to be able to estimate the frequency of occurrence of a particular action. Furthermore, you need to understand the impact of such an event—if a particular scenario is rare but will have such a significant impact on the business that it needs to have a high priority for testing purposes whereas a low impact event will have a lower priority. Therefore the requirements will need to include information about frequency and impact, so that relevant priorities can be assigned to different test cases. As we have mentioned already, in most environments 100% testing is not feasible but, if you are not going to test everything, then it is best to test what is most important and most frequent. One important thing to note is that it is going to be quite difficult to assign probabilities within a semantic approach whereas it is relatively easy when using flow charts as you can have a value assigned to each path or sub-path.
Changes to requirements

Requirements change. This is either because of a genuine change of heart or because the user realises that something has been omitted or something that has been included is not required or because some particular definition is wrong or needs amending. How do you handle this? Firstly, the software needs to recognise any changes, secondly it needs to generate new test cases to reflect the new reality and, thirdly, it also needs to identify what does not need to be re-tested. If you don’t have automated test case generation that can do these things for you, then it may not be easy to identify which tests that you have already run that do not need to be run again. So there is a tendency to run all the tests again, because it is simpler, but is also a waste of time. This is especially a problem in agile testing environments where testing is recurrent and ongoing and where over-testing can be a real problem.

There are several issues here with respect to automation. To begin with, you need a tool that will recognise and process just the changes that have been made to the requirements. What you don’t want is to regenerate all of your test cases for the entire set of requirements: just the ones that have changed. Secondly, unless you have some sort of test case management system it is of limited use to simply generate a ‘where used’ report against changes. This is because there is no way to tie this automatically to the test cases you already have, which means reverting to a manual process, which would defeat the object.

While on the subject of test case management our experience is that most companies do not use such technology. There may be some central folder in which test cases are stored but there are no mechanisms in place to ensure that test cases are reused, no procedures to identify when reuse would be appropriate, no ability to recognise duplicated or overlapping test cases, and no facilities to identify out-of-date test cases than can usefully be deleted. The result is that most development shops are bloated with unnecessary and redundant test cases that cost time and money to maintain.

“…there is a tendency to run all the tests again, because it is simpler, but is also a waste of time.”
Conclusion
Since its inception, IT has been about automating business processes.

However, it has been slow to apply that concept to itself. In part, at least, this is because IT departments are primarily tasked with serving the business; improving their own performance so that they can better serve the business is all too often regarded as secondary and budget is often not forthcoming for this purpose. However, this is a short sighted view: the cost to the business of manual software testing results in more defects than there should be, detected later than they should be, and costing more to rectify than they should. That is always supposing that you don’t go ahead and release the software in a defect-laden state, in which case it won’t just cost you more but way, way more than it should.

While automating test case generation isn’t sufficient on its own to ensure fully functional, accurate application development, it is necessary. Without tying test cases to requirements, testing will continue to be ad hoc, time consuming, inefficient and wasteful. At Bloor Research we are inclined to prefer model-based approaches to this issue compared to semantic methods but that is really a different discussion; the point at hand is that a formalised approach needs to be adopted.

FURTHER INFORMATION
Further information about this subject is available from http://www.BloorResearch.com/update/2233
About the author
PHILIP HOWARD
Research Director / Information Management

Philip started in the computer industry way back in 1973 and has variously worked as a systems analyst, programmer and salesperson, as well as in marketing and product management, for a variety of companies including GEC Marconi, GPT, Philips Data Systems, Raytheon and NCR.

After a quarter of a century of not being his own boss Philip set up his own company in 1992 and his first client was Bloor Research (then ButlerBloor), with Philip working for the company as an associate analyst. His relationship with Bloor Research has continued since that time and he is now Research Director focused on Data Management.

Data management refers to the management, movement, governance and storage of data and involves diverse technologies that include (but are not limited to) databases and data warehousing, data integration (including ETL, data migration and data federation), data quality, master data management, metadata management and log and event management. Philip also tracks spreadsheet management and complex event processing.

In addition to the numerous reports Philip has written on behalf of Bloor Research, Philip also contributes regularly to IT-Director.com and IT-Analysis.com and was previously editor of both “Application Development News” and “Operating System News” on behalf of Cambridge Market Intelligence (CMI). He has also contributed to various magazines and written a number of reports published by companies such as CMI and The Financial Times. Philip speaks regularly at conferences and other events throughout Europe and North America.

Away from work, Philip’s primary leisure activities are canal boats, skiing, playing Bridge (at which he is a Life Master), dining out and walking Benji the dog.
Bloor overview

Bloor Research is one of Europe’s leading IT research, analysis and consultancy organisations, and in 2014 celebrates its 25th anniversary. We explain how to bring greater Agility to corporate IT systems through the effective governance, management and leverage of Information. We have built a reputation for ‘telling the right story’ with independent, intelligent, well-articulated communications content and publications on all aspects of the ICT industry. We believe the objective of telling the right story is to:

- Describe the technology in context to its business value and the other systems and processes it interacts with.
- Understand how new and innovative technologies fit in with existing ICT investments.
- Look at the whole market and explain all the solutions available and how they can be more effectively evaluated.
- Filter ‘noise’ and make it easier to find the additional information or news that supports both investment and implementation.
- Ensure all our content is available through the most appropriate channel.

Founded in 1989, we have spent 25 years distributing research and analysis to IT user and vendor organisations throughout the world via online subscriptions, tailored research services, events and consultancy projects. We are committed to turning our knowledge into business value for you.
Copyright and disclaimer
This document is copyright © 2014 Bloor. No part of this publication may be reproduced by any method whatsoever without the prior consent of Bloor Research. Due to the nature of this material, numerous hardware and software products have been mentioned by name. In the majority, if not all, of the cases, these product names are claimed as trademarks by the companies that manufacture the products. It is not Bloor Research's intent to claim these names or trademarks as our own. Likewise, company logos, graphics or screen shots have been reproduced with the consent of the owner and are subject to that owner's copyright.

Whilst every care has been taken in the preparation of this document to ensure that the information is correct, the publishers cannot accept responsibility for any errors or omissions.