

Software – a risky business

Louise Pryor shows us how to improve our programming methods.

ALMOST EVERY WEEK, it seems, there is an article in the newspaper about some software project that has gone hideously wrong: massively over budget and behind timetable. Software products are apparently rarely released on time, and some first releases are so bug-ridden as to be nearly unusable. Software development is obviously a risky business. As we enter the post-N2, post-Turnbull world, and actuaries become more conscious of the risks in their own business, you may find yourself heaving a sigh of relief that at least you are not in the software business.

Well, I have news for you. The bad news is that many actuaries are indeed in the software business. The good news is that many of the risks are easy to mitigate through the use of simple techniques of software engineering.

Actuaries are software developers

If you are building spreadsheets, or building your own models in an off-the-shelf modelling package, you are developing software and are exposed to risk. The two major things that can go wrong are that the software you develop is full of bugs, or that it can take much longer to develop than you had anticipated. These two problems are interconnected: it may be that it is the time taken to remove the bugs that makes the development run behind schedule.

Considered as software development projects, most spreadsheets are of course 'small beer'; we are not talking of hundreds of person-years of development time, nor of projects lasting many months. But the risks are not proportionate, eg for many actuaries, spreadsheets that go wrong or produce the wrong results can affect their whole business. (In this article I'll concentrate on spreadsheets for the sake of simplicity: the issues are the same for financial modelling packages.)

Some examples

Bugs are not necessarily a big problem *per se* – they become so only when they lurk undiscovered. The really obvious bugs, such as reversed positive and negative cashflows, are usually evident early on, when it is easy to fix them. Some bugs are more insidious, and if they are not spotted can result in significant errors in the overall results. When they are eventually spotted, they can result in significant time being spent in trying to fix them and in rerunning sets of calculations.

For example, many of us will have come across those rogue cells that refuse to recalculate when you expect them to. If such a cell is buried deep inside the intermediate calculations, rather than being one of the headline results, you may simply not notice it, or

when you do notice it, you have to rerun several sets of results.

In some cases it is not difficult to find the bug, but fixing it is another matter. You know the story: it was only a small change, and should have affected only a small part of the overall results, but somehow something has gone wrong in a totally different part of the spreadsheet... so you undo the change, but can't seem to get back to the original state, try as you might. This experience is extremely frustrating and worrying – what other unexpected dependencies are lurking beneath the surface, waiting to trip you up?

Sometimes there are assumptions built into the spreadsheet logic; a parameter is assumed to be non-zero, or to lie within a certain range. A future user who is unaware of the assumption is also likely to be unaware that the results are wrong. In the best case the future user is the same person as the original developer, using the spreadsheet only a few days after it was first built. There is then some hope that their memory will be jogged, and no irretrievable mistake will be made. However, this situation will not apply to other users.

If you are in the habit of reusing code on a regular basis, either by reusing whole spreadsheets or even just macros, you are exposed to any bugs that there might be in the original code. Even if the bug has been found and fixed, are you sure that you know about it, and are using an up-to-date version? This is by no means an argument against code reuse, which is usually considered to be a good thing, but you should be aware of the risks.

Luckily, many of these problems are avoidable in most circumstances.

Avoiding the risks

It is a truism in the software development world that the earlier a bug is found, the less trouble it causes and the less time it takes to fix. The best case of all is of course not to introduce any bugs in the first place, but that is a counsel of perfection. However, it is possible to reduce the number of bugs and to minimise their adverse effects. In order to do so, you should use a suitable software development process. A good process is one that facilitates the production of good-quality software, and that is not too onerous in practice. It must provide readily perceivable advantages to the software developers. In my experience, initial scepticism is usually short lived, provided that developers immediately find their lives less frustrating and more productive. The process must be adapted so that it directly addresses the organisation's needs, and fits in with existing processes such as peer review.

Key components of any process are control, testing, and documentation.

■ **Control** means being disciplined about the changes you make. Use a source code control system such as CVS or Microsoft SourceSafe, so that you can always revert to previous versions. Make only one change at a time, so that if anything goes wrong you have a fairly clear idea of where to look for the problem. Have a central repository of macros or DLLs, so that you make sure you are always using the latest version. Keep track of what macros or DLLs you are using, and where they have come from. Have coding standards, so that your spreadsheets separate inputs, outputs, and internal workings, protect cells that should not be changed, and so on. The aim is to make it as difficult as possible for people to mess things up.

■ **Testing** is vital for bug control. The idea is that by using sets of inputs for which you know what the outputs should be, you can tell immediately when anything has gone wrong. The easier it is, the more testing will be performed, so set up an automated system: every time a spreadsheet is changed, you should rerun the tests. Choose the tests carefully, to ensure that boundary conditions are covered as well as expected conditions. Simply building in checks for consistency is not enough: unfortunately, calculations can be consistent but wrong. Test individual calculations in detail as well as the overall results; this will help in debugging, as well as detecting some of the more obscure problems. Often, the very process of structuring the calculations in such a way that they can be easily tested helps improve the formulae used and uncovers bugs.

■ Good **documentation** makes the difference between a spreadsheet that has a long and productive life and one that can never be used again. Documentation that is included in the spreadsheet itself is easy to update, and is more likely to stay synchronised with the spreadsheet contents. You should document:

- ◆ any assumptions made;
- ◆ the formulae used;
- ◆ what inputs are required;
- ◆ the order that macros should be run in; and
- ◆ any other information that will make the spreadsheet easier to use or update.

Again, you should have standard documentation methods, so that other users will know where to look to find the information they need.

These three components should be present whenever any software is developed, even if you are the only person involved and you don't think that you or anyone else will ever need to use the software again. You



can't tell at the outset what the future life of software is going to be, and time spent at the beginning can be repaid many times over further down the line. Often, 'down the line' turns out to be days, rather than months in the future; it's surprising how quickly you forget the details that were so important when you first built the spreadsheet. Moreover, techniques such as documentation and systematic testing can actually save time up front, as they help you organise your thoughts and spot future problems before they arise.

If you are working as part of a larger team, there are more options open to you. There are various techniques for reviewing code, for example, or programming in pairs rather than individually, that are well documented as improving both software quality and long-term productivity. Again, the review may take time at the beginning, but if it prevents future bugs it saves time overall.

In general, setting up automated tests, or using a source code control system – or indeed any activity that appears not to make an immediate contribution to progress – is often resented as a time-wasting activity when the results are needed urgently. However, if you use techniques such as these regularly, you soon learn that they are not optional extras, but that in the long term they repay their investment many times over. And in software projects, the long term can be as short as half an hour, if disaster strikes. □



Louise Pryor is an independent consultant; she has encountered many software risks at first hand in her career as an actuary and in developing commercial software products