Salutary lessons in software engineering

Author: Les Hatton, Oakwood Computing, U.K. and the Computing Laboratory, University of Kent, UK.

The situation so far ...

This article is not written specifically with the aviation MRO industry in mind but the problems I will describe here are common to all industries which are involved with the procurement and use of software-controlled systems.

Readers will be aware that all is not necessarily well with such systems. In this article, I will try and describe why and what the prospective user can do about it.

Anybody who has used a PC will be all too familiar with the dreaded BSOD (Blue Screen of Death). This is the monitor picture presented to the users of some systems when the PC simply freezes for some unimaginable reason, often accompanied by a breathtakingly inscrutable error message.

It is easy to dismiss this sort of thing as simply one of the facts of life when using PCs but the problem is much more widespread than that. Software today is present in everything from electric razors, (which contain something euphemistically known as 'head-adjusting' software, which I will leave the reader to interpret), to commercial aircraft which are controlled by the equivalent of millions of lines of source code. Source code is simply the language in which the system behaviour is defined and rather resembles a cooking recipe. Table 1 gives you some idea of the quantities involved.

Device	Size of software (in source lines)
Cellular telephone	30,000
Cellular telephone	30,000
Hand-held bar scanner	~ 50,000
Aileron control, Boeing 777	130,000
Line-scan television	250,000
Air-traffic control component	250,000
Automated bank teller network	600,000
Telephone call-routing	2,100,000
B-2 Stealth bomber	3,500,000
Combat computers aboard SeaWolf submarine	3,600,000

Table 1: The size of the software systems in various devices.

To put Figure 1 in context, a typical line of code has around 20 characters on average, so the combat computers aboard the SeaWolf submarine are

controlled by a text approximately 6,500 times bigger than this article. A single transposition of one pair of characters could cause catastrophic failure.

Not only this but the amount of software in consumer electronic devices is growing extraordinarily quickly. Today, the amount of code in some devices is doubling about every 18 months. Figure 2 gives some idea of the scale of this in one particular class of commercial aircraft.

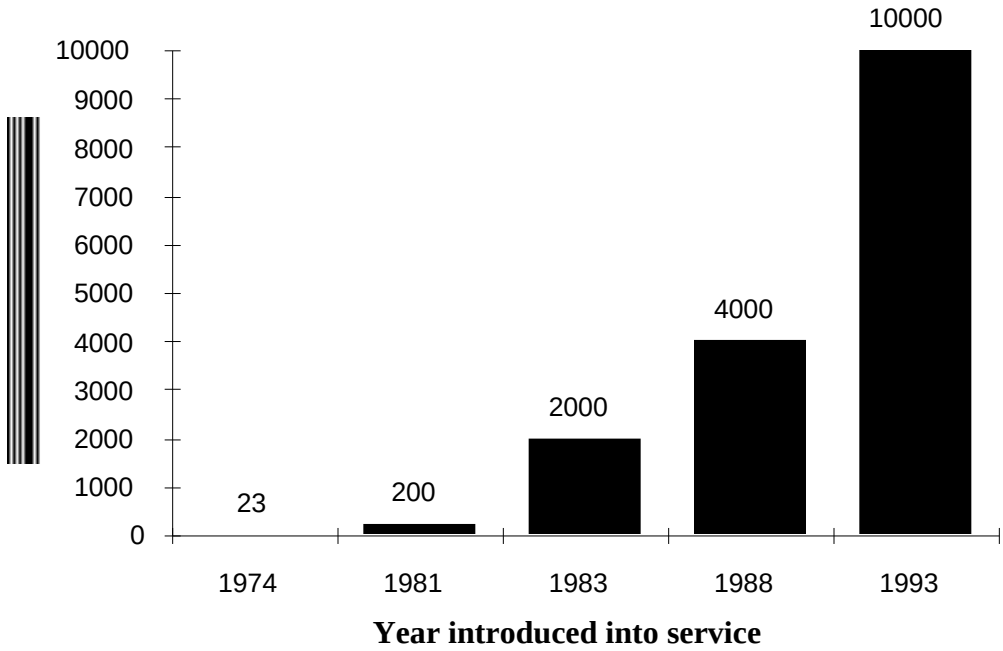

Figure 2 Graph illustrating the rate of increase in software onboard the Airbus A3XX series.

In spite of our now critical dependence on software, the kind of unfortunate behaviour we observe in PCs also manifests itself in critical systems. A critical system is one on which we depend in some important way. The problem is quite simple:- we do not have the technology to produce an error-free system of any significance. We simply don't know how to do it. Not only that, the imperfect systems we deliver are usually delivered late, delivered incomplete and to put the icing on the cake, do not usually do what the user originally requested, assuming that he or she knew, which in itself is far from common. Finally, many systems are so hopeless, they never appear at all.

Let me quote a few examples of each of these categories. In January 1990, engineers at AT&T, the US telephone giant made a single mistake in upgrading a communications system containing some 3 million lines of source code. The effects of this mistake were to paralyse the entire US long distance telephone network for several hours. Total paralysis took only a few minutes to achieve once the initial mistake, (software engineers call them faults), had actually caused the system to fail. In September 1997, the US Navy's high-tech wonder ship, the USS Yorktown sat dead in the water off the Virginia coast until it could be rebooted. Yes, the wonder ship depends on software to such an extent that critical functions are not available if the

systems are down. This could lead to an addition to the international ship to ship flag signalling code of “Please wait while we keelhaul the software engineers AFTER they have got the damn ship going again ...”. It is tempting to think that another 20 years or so of this kind of development and we will no longer be able to go to war which is no bad thing of course. In July 1999, the Washington Post reported that General Motors are finally going to have to recall 3.5 MILLION vehicles because of software defects in their anti-lock braking systems which actually extended braking distances by some 50 feet, which is not exactly the desired effect. Each of these failures were small defects which happened to cause complete failure of the system in terms of its intended action. Engineers and scientists call this chaotic behaviour. It is one of the fundamental properties of software systems which are not in general shared by other kinds of engineering systems such as those occurring in civil and mechanical engineering.

All of these failures had something else in common - they were extraordinarily expensive. The AT&T outage was estimated in some quarters as having cost around 1 billion dollars, whilst the cost of recalling 3.5 million vehicles must be a source of some little concern to GM’s accountants. Closer to home, ICL were successfully sued by St. Alban’s district council for around £1million because they delivered a poll tax collection system which miscalculated the number of people eligible to pay. The result was a directly quantifiable failure and the Court of Appeal in a very enlightened decision in my view, ruled that ICL were liable for this loss because it could not have been in the contemplation of the parties of the contract that delivering software with such a basic fault (the engineering term is a ‘show-stopper’) would be acceptable. This neatly side-stepped the current great controversy in the legal world as to whether software is goods or a service and therefore into which statutory regime it falls.

These are the good systems by the way. They were developed, released and generally do what they are supposed to do. In contrast, the original London Ambulance service system, the London Stock Exchange TAURUS system, a number of infamous NHS systems and many others too numerous to count were enormously expensive failures - they didn’t produce anything of lasting benefit at all. They simply blundered forward absorbing more and more money until finally reality hit home very hard indeed and the projects were mercifully canned before they could waste any more. This problem is universal. In the early 1990’s the US Department of Defence published an astonishing set of figures reproduced as Figure 3 showing that in its military avionics procurement in the period 1985-1990, (avionics are software based systems which control aircraft in this sense), 90% of the systems were either never delivered or never worked ! This isn’t because nearly everybody was incompetent by the way, its because with the best will in the world, we don’t know yet how to produce them successfully in a systematic way.

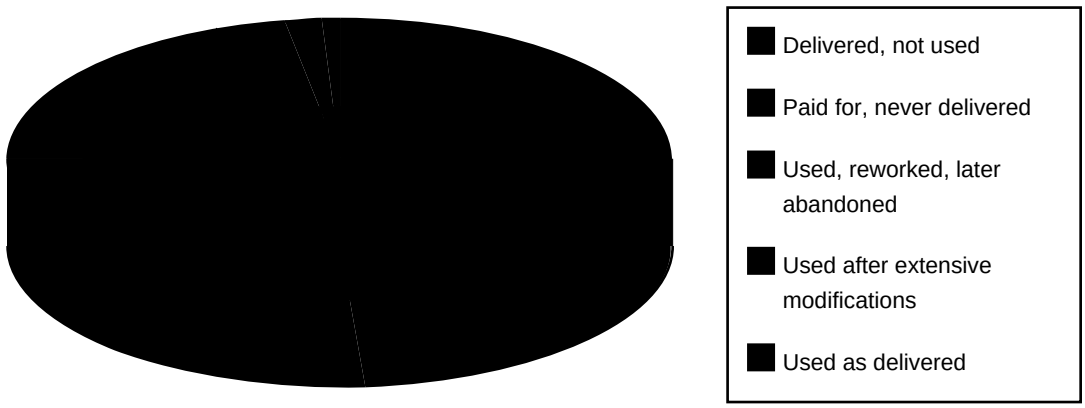

Figure 3: Data from the Audit Office of the US Department of Defence giving the breakdown status of military avionics procurement programs 1985-1990.

Against all these drawbacks we must balance the fact that in spite of their vagaries, software controlled systems are a massive step forward in many areas and we simply could not manage the complexities of modern society without them. Take for example communication systems. These are wholly controlled by vast amounts of software and although they fail catastrophically every now and then, they are now completely indispensable.

The whole thing then boils down to something known as risk assessment and risk management. Given that software based systems have a large benefit and given the not inconsiderable risk of either never appearing at all, or if they do appear, appearing in an imperfect form, how do we effect a good business trade-off ?

What you can do ...

Readers I'm sure will fall into many categories. In contrast, the three main categories into which we divide software are, a) COTS systems, b) Hybrid systems and c) Bespoke systems.

COTS systems

Every PC user is familiar with COTS systems. This is an acronym for Commercial Off The Shelf. They are characterised by the important attribute that they are one of the 10% or so which actually made it to the starting line. In other words, you can evaluate their suitability before purchase, or at least before major commitment because somebody else undertook the development risk. Against this benefit, there are two risks. First of all, the software may not have been developed to standards appropriate to the environment in which you intend to use it. In other words it might crash too

often or produce the wrong results. Legally, this is a very tricky area. If you assess the software and decide that it is suitable and it later fails badly, you probably have no legal comeback. The best advice is to take your time during the assessment making sure that the software is sufficiently robust.

Second, it might not do precisely what you require. You have two choices here. You can either modify your business practices if practicable to suit the software and a number of small to medium sized companies have done this successfully. This might sound a little outlandish but there is a very strong precedent for this in accountancy practices. Today, no company would set itself up with alternative accounting methods to the universally accepted ones. Instead, companies modify their own practices to comply. Doing the same thing with other business practices might be highly beneficial.

Alternatively, if you can't operate within the restrictions posed by the software, you might choose to request the developers to produce a special version modified to your requirements. This we call a hybrid system.

Hybrid systems.

A hybrid system will normally lead to a contract between the developer and the end-user to take a standard piece of COTS software, for example, a database management system and modify it in some way to satisfy the user's requirements. Providing the modifications are not too large, this is a good way of balancing the risk of the system not appearing at all against the risk of the system not satisfying the user's requirements. Many successful systems have been produced this way. However, as the modifications increase, the risk of not appearing at all increases.

It is difficult to over-estimate the importance of the contract in these kind of developments. In law, contracts are generally designed not to be used. In other words, the contract only becomes relevant when there is a dispute. Most of the time this does not happen and the parties to the contract happily conclude their mutual business.

In software development however, problems are the status quo regrettably. In other words, in contrast to normal contracts, it is very likely that a software contract will be used. Because of this, it is very important to craft a contract which deals sensibly with the possibility of failure, simply because some kind of failure is very likely. The contract should carefully lead the developer and end-user to the most satisfactory outcome for the two parties, perhaps of a partially completed system for a certain price. The contract should provide if possible for incremental development and release whereby a system is supplied in pieces. This is an excellent way of judging whether there are any major problems in store. Most of the great failures to deliver anything at all have been so-called 'big-bang' deliveries where the whole of a mighty system is scheduled to appear at some date. The date is continually put-off until finally somebody mercifully pulls the plug.

Writing contracts such as these requires very close consultation between both

parties, their lawyers, the users and the development engineers. The cost of this is simply weighed against the cost of the development failing without suitable contractual guidance. At least some of it would be part of a desirable process called requirements capture anyway.

Bespoke systems

In contrast to the previous two categories, a bespoke system is built from scratch to the user's requirements. This has by far the biggest risk of failure to appear. In general, such developments should be avoided like the plague, but if they are necessary, they should be made as simple and unambitious as possible. There are a number of development techniques which I haven't the space to go into to ameliorate the risk but this kind of system remains generally problematic. The contract is of crucial importance here and detailed requirements for the system should be part of it.

AUTHOR BIOGRAPHY:
Les Hatton has been suffering from computer systems for 30 years. He is a software consultant and Professor of Software Reliability at the University of Kent. In October 1998, he was named in the 'world's leading 15 scholars of systems and software engineering' by the US*Journal of Systems and Software*. He also has an LL.M. in IT law from the University of Strathclyde.