A LARGE EMBEDDED SYSTEM PROJECT CASE STUDY

First published in "Software Engineering for Large Software Systems"
Edited by B A Kitchenham, Elsevier Applied Science, 1990
Made available in this format by kind permission of Kluwer Academic Publishers

Bob Malcolm
ideo ltd
Kimpton Bottom
Hertfordshire

Abstract

This paper presents a discussion of why large software projects go wrong. It attempts to cut through the swathes of myth and misrepresentation, and to dig deeper than the press and other superficial pundits. The tentative analysis is based on a case study of a project which is representative of the typical 'project disaster'.

INTRODUCTION

Embedded real-time systems typically comprise chains of different types of information processing in and out of a central computer-based information presentation and decision support system.

The input chains start with analogue signal processing of the raw data from 'front-end' sensors, followed by digital signal processing, and then the digital data processing of a computer-based system. This may itself have associated digital electronics, such as operator work-stations, often purpose-built.

The output side is much the same in reverse, starting with the computer-based system controlling digital, possibly microprocessor-based, sub-systems which in turn control analogue actuators and transmitters.

This is a greatly simplified picture of just the primary information channels of a system. In addition there will be many subsidiary control loops comprising special-purpose digital circuitry and perhaps several microprocessors within each box of electronics.

The case study project was for the bespoke development of just such a system.

Apparent and real causes of problems

After the disaster come the recriminations - attempts to apportion blame to the design, the design method, the designers, or the management of the project:

"Whose fault was it? It *must* have been a bad design. They must have used poor (ie Not Invented By The Critic) design techniques. They should have had a quality management system. The whole thing was mismanaged wasn't it? It *must* have been the software, mustn't it?"

And, of course, there is much argument over the requirements. "Did they or did they not keep growing?; did they or did they not keep changing? Was it really the customer's fault?"

"How *could* they do it?"

But there isn't a 'they'. *They* are individuals, usually well-qualified, well-intentioned, well-motivated - at least as competent as their critics.

The body of this paper examines the requirements, the design, the quality assurance, and the management of the project, looking at both the commonly blamed problems and the real problems. This is followed by an analysis of the underlying factors which created the environment in which the technical difficulties arose, and an attempt to draw some still-tentative conclusions.

REQUIREMENTS

Well, did the specification grow like Topsy?

Yes . . . . and No. Towards the end there was a fairly typically bitter contractual battle which led as usual to all parties leaning on the written word, whatever had been the spirit and intent of the contract. The customer went back to the original few pages of outline requirement, which was the basis for the proposal and thence the contract, and said that it had not changed. Indeed it had not: but the requirements had moved on somewhat. *Apart from* official contractual changes, the actuality of the requirements - the fleshing-out and interpretation - was embodied in a myriad documents, designs, undocumented decisions, and assumptions.

Nevertheless, and perhaps as was only to be expected in such a difficult situation, the customer, publicly at least, flatly denied the relevance of anything outside the original flimsy statement.

In fact, very early in the project it was recognised by both customer and supplier that there were likely to be problems with the requirements. They decided to establish a group to resolve uncertainties. It was to be small, so that it could make decisions *very quickly*, with only three parties represented - end-user, supplier, and customer's technical advisers.

The road to a project disaster is also paved with good intentions. Ten years later this small, fleet, group had become a *committee* with over a score of customer representatives from various departments. It met every six months. So the multi-humped camel was born, though really it never got beyond gestation since, by this time, what was actually happening on the project bore little relationship to the formal committee statement of the long-term requirements.

What about the software requirement specification?

During tests and trials with the real hardware in a real environment, there were reports back from both the customer and the trials team that the software was not coping well with the demand. The software team tried to ameliorate the problem with a series of changes, but this 'software problem' did persist. Only when one of the managers who had been involved in the early design work became involved did reality reveal itself. The demand with which the software was 'having difficulty' was *more than two orders of magnitude greater than the design target*. A software problem indeed!

But the supplier should not be complacent in a situation like this. The software might well satisfy the software requirement specification, and in the case study dramatically beat it. The hardware might even have met its specification, though it did not in this case. But if the real world load is greater than expected, then the customer will not be happy if the supplier tries to hide behind the specification. In the case-study project, the whole system would have been quite useless if the software had simply met its specification. The supplier might have a legal case, but if it comes to an argument, then the battle is already half lost and a disaster imminent.

This is yet another case of the important distinction between meeting the specification and *real* quality - providing genuine customer satisfaction.

DESIGN

Design philosophy

We must take care to distinguish between the design philosophy and the actual design. The software philosophy was simplistically expressed at the beginning as "1. Don't rush (ie don't introduce into the design unnecessary real-time constraints); 2. Do only one thing at a time". In essence this meant message-passing, interrupt-free, design - some years ahead of its time.

The philosophy worked well. The first build of the system integrated successfully in *one week*, rather than the expected three months, much to the surprise of the software manager who had not been in favour of this newfangled philosophy.

But the ultimate vindication must surely be that although it was creaking badly, the system did actually cope with more than one hundred times the original expected data rate, with a consequent combinatorial explosion in some aspects of the data-handling.

However, there were indeed some horrors in the actual design of the "How *could* they?" sort. And there were some design decisions which have since been criticised but which were well-founded at the time. Which is which is usually clear: but not always. Some of each type will be recounted in this section.

Hardware-software interface
One of the ways to 'avoid rushing' was to buffer data. There were therefore large 'software' buffers in the main computer, to temporarily accommodate input data which did not need an instant response. In addition there were buffers in the hardware lines, prior to transferring data into the main computer. But the function of these hardware buffers was slightly different. When they were not full, they acted simply as buffers. But, unusually, they were kept as small as possible, consistent with not losing data under normal circumstances. This provided a simple and cost-effective counter-measure to spurious unmanageable surges of noise.

The idea was that the little hardware buffers would be deliberately overwritten in the event of such surges - and there are a lot of these in reality. In this way most of the noise would be lost, and the main computer would not be overloaded. (Some real data would be lost as well but not, relative to the noise, a significant amount, and it was better to let the computer spend its time processing clean data from less noisy areas than clog it up trying to sort wheat from chaff with a low probability of success.)

Years later there were some problems with the main computer servicing of its buffers. It transpired that, somewhere along the line, the software manager, finding that he was running out of store, had persuaded the hardware team that the hardware buffers should be expanded, so that the software buffers could be shrunk. Now, not only was the hardware not limiting noisy input data, but the main computer had less capacity to absorb the load. Is it any wonder there were problems?

A box too few
Fairly early in the project it came to light that a major function had been completely left out of the initial design. Its natural place lay between the digital signal processing and the digital data processing. Between is right: it transpired that both the design managers responsible either side had assumed that the other had dealt with it. They sat in adjacent offices, did not get along together, did not talk, and certainly failed to communicate.

Despite recognition of this oversight, the effect persisted. Years later, in trying to sort out some interfacing problems, it became clear that the missing functionality had been 'bolted on', along with some other forgotten bits and pieces.

It had, for instance, been a stated starting assumption of the software team that there would not be spurious multiple copies of the same input data, and also that the data would already be sorted on arrival. Both of these assumptions were 'remembered' - ie rediscovered - when the serious effect of them not being satisfied became evident. Again, these functions had been belatedly 'bolted on' in another peripheral processing unit.

But by this time such inelegances were everywhere as people and teams optimised their own sub-system at the expense of others and the whole. Sometimes they were grabbing responsibility for functions from others - either to build empires or because the new bit was technical fun. Sometimes they were dropping responsibility for functions to reduce the load on themselves and on the processing capability in their bit.

A box too many
And sometimes these things happened for no obvious reason at all: thus it was with the interfacing unit. This box was originally devised as a solution to the interface limitations of the first main computer. Again, quite late in the project there were some technical difficulties. The design team eventually called in help so as to better understand the function of the unit. Again, only when one of the old hands became involved did he realise that this box was still in the system. The original computer had been replaced some years previously by one with far superior interfacing capability, completely obviating the need for the interface unit.

"We'll just do it in the software"
There were a lot of other interface problems, with both the main computer and with the subsidiary microprocessors which had sprung up throughout the system. Often the hardware engineers relied on 'doing in the software' jobs which would be simple in hardware but horrendously difficult and expensive in software.

There were wires soldered the wrong way round on a parallel data input channel, so that the least significant bit was where the most significant should have been and vice versa. ("Is bit 1 or bit 16 the most significant bit? - or should that be 0 or 15?") Rather than resolder the wires, the software was expected to turn every data word coming across that interface back to front, bit by bit.

Elsewhere some microprocessor software received data from a specially built keyboard. Now, electromechanical keys have a tendency to 'bounce' for several tens of milliseconds after being pressed. The effect, if not handled properly, is to generate a stream of spurious data. It had been common practice for years to build in simple 'contact debounce' circuitry at the interface. Later this became encapsulated in standard chips. But in this case the hardware engineer did not know about them, or forgot, or couldn't be bothered to use them. So the software was expected to do 'contact debounce'. It is *possible* in software, but it is very painful.

Elsewhere again, in another microprocessor, the software was 'thrashing'. It was having to inspect an input channel so frequently that it had no time to do any actual processing of the data. The software team were struggling, being criticised by the hardware team for poor performance, until the fight got so bad that yet another old hand was brought in to referee. It turned out that the hardware team had failed to put in a one word register as a buffer which would have latched the data - stored it temporarily until the processor was ready.

But one of the biggest interface boobs of all concerned the distribution of peripheral functions from the main computer to local subsidiary processors. Even before the availability of microprocessors such distribution had been part of the design philosophy. The approach was then novel and well-publicised, being presented at international conferences.

The original, and well documented, design concept, was that each packet of data should have an identity related to its source. This identity would be passed to the subsidiary system. When the subsidiary system required more information from a particular source, it would request it from the main computer, referencing this identity, making it a fairly simple matter for the main computer to find it. Nowadays we would talk of 'object oriented design', but again this project was ahead of its time. Unfortunately, as the project progressed and personnel changed, a hardware engineer, in order to save a small amount of store, decided not to bother to save this identity. So the only information passed back to the main computer was the value of some of the data already sent. The main computer then had to work out from which of several hundred possible sources of data this packet had come.

The original concept was intended to simplify the interface, and reduce the associated processing so that it was virtually negligible. The consequence of the actual design was to bring the main computer to a near standstill. (Later this was rectified, to some extent.)

Yet another example of corporate forgetfulness, despite clear documentation, concerned the Built-In Test Equipment (BITE). Simplifying, this takes two forms. Static BITE is a facility to monitor such things as voltages in analogue circuitry, to ensure that components are operating properly. Dynamic BITE often requires some input-to-output test to check that the functional operation delivers expected results. The need for both was recognised and an outline approach documented. It was pointed out that the dynamic BITE needed further consideration. It got it - but years later, after the kit had been built without it.

Good programming practice

Moving onto the software design *per se*, the problems were different, but just as perverse.

Only when a new programmer had to amend an old module of someone else's code did its rather curious structure come to light. The input to the module was a single piece of data, together with a number. The data was to be placed into a table of data. The number was a pointer to the right place in the table.

This job *could* have been done simply with just a single program statement (an array assignment). Instead of which it went on impenetrably with a long chain of if's and then's: "If the pointer value is one then put the data into the first place in the table else if it is two then put it into the second place else if ..." and so on for half a page. (Luckily the table did not have a million entries!)

Of course, the programmer of the original had thought that he was simply following the company's very thorough and copiously documented structured programming rules. It is rather like someone, having been told that the shortest distance between two points on the globe is a great circle, sets off from London to Brighton - heading north. It *is* a great circle, but...

More seriously in effect, it was realised quite late on that at least ten per cent of machine time was spent shuffling unused - that is not worked on - data about the main store, ending up unchanged exactly where it started. This was discovered only when a performance analysis was done to see where processing time might be saved. But it had not been done unknowingly. The particular designers thought that it was 'good' programming practice to make local copies of data which might be needed, before execution of a subprogram. Indeed that is what some of the text-books taught. But nobody at the time saw, directly, the effects of this 'good' design. And who would have thought that they would go one stage further and copy it all back again after it had been 'used' (ie, usually, *not* used).

Estimates and resources

On the general subject of estimation and performance, this project was much like any other. The estimates for the requirements which had been the centre of attention during feasibility studies were not *too* bad. But lurking beneath the waterline were the other nine-tenths of boring, forgotten, and late-coming bits. For instance, the estimate for one suite of 'subsidiary' software facilities was sub-contracted out to a firm of software specialists. The store estimate was 200 words. Ten years on it was 600K and rising daily - now two-thirds of the total main computer store requirement.

Partly because of the performance problems, the choices of computer and programming language both came in for criticism. Apart from upgrades, there was one mid-development change of main computer type. At that time there had been some support in the supplier organisation for shopping abroad. But there was then a clear and well-documented instruction from the customer to "Buy British". Interestingly, this seemed to get forgotten during the later recriminations.

By the end of the project the programming language chosen was considered by many to be passé. But at the beginning it was the only available high level language for this type of application. It was also the customer's standard. It was ahead of its time - so far ahead that an appropriate compiler did not exist at the beginning. And throughout the project there was a running battle with the compiler suppliers to support it adequately. (A familiar story?)

The possibilities of changing horses in midstream - whether of language or the main computer again - were frequently explored, with thorough studies of the options available. Each time it was felt that the technical advantages would be outweighed by the management disadvantages in simply being able to handle such a massive change. So the less than perfect choices were made not by default, but consciously, carefully looking at the trade-offs.

QUALITY ASSURANCE

Quality Management System

At about the same time as the project was getting under way, the company was one of the first anywhere to introduce an explicit software quality management system with a dedicated software quality department.

At least ten per cent of software costs were allocated to the quality management function. Its operation was regularly vetted by the customer, and in general got a clean bill of health.

Closely associated with the quality system was the training department which, in essence, taught the technical and procedural standards for the project. All new staff, however highly educated, went through an intensive four-week course in these.

Some features of the quality management system might now be considered old-fashioned but, essentially, at first at least, it worked. But the reason that it worked, and the extent to which it worked, and the way that it worked were due much less to *what* was done than to *why* it was done.

The benefits were to a great extent in *communication* - giving a large and changing team a common way of doing things, of pulling teams together in some direction.

It also gave everyone a good feeling to know that they were trail-blazing these new techniques - that they were held up as a shining example to others of the bright and better way to do things. So what, if anything, went wrong?

Structure and style: procedures and paper

First and foremost the creation of a large and separate quality department, certainly *in the form which it took here*, caused some of the problems. The approach is quite common in the industry and it is taken for good reasons. Even if given explicit responsibility for performing quality-related functions, development staff notoriously treat them with lower priority than their current technical or managerial challenge. Furthermore, it is often felt that the independence of a separate department enables it to act as a kind of corporate conscience.

But that separation led to different goals, different styles of operation, and a different type of staff. Unable to see into the technical content of the project to directly assess its actual quality, they concentrated on execution of the procedures, and on providing documented evidence of the execution of the procedures. Paper, paper, and more paper. Rows and rows of filing cabinets full of forms. After all, this was what was needed to have the quality management system 'registered'.

So programmers came to look on 'quality' as something to do with form-filling - nothing more than a boring clerical task.

They willingly relinquished control, chairmanship, and conduct of their design reviews to the willing expansionist quality department. How else, on such a project, could "Not Applicable" have been accepted as the answer to a standard checklist question of "Have the real-time constraints been satisfied?"

Elsewhere, they encouraged the configuration control clerk to fill in the configuration control forms for them. Of course the consequence was that the forms might have had all the right numbers in all the right boxes, but absolutely **no** content. This was fine for registration, but clearly, without real content, the forms could never have been the basis of *control*. And indeed they were not, since most of them were completed as a matter of form (!) some time after changes had been made.

Configuration management: pretence or reality?

Behind this superficial problem lies a much bigger problem associated with software configuration management generally. Part of the problem lay in the fact that the way it was done (and in many places, still is) was based on received wisdom. And because the development team had abdicated responsibility for these issues, no-one there looked closely enough to analyse the problem and see, and be brave enough to say, that received wisdom was wrong.

Software configuration management procedures were based on those for hardware. But few software quality managers realise that the hardware procedures are generally concerned with change *embodiment*. That is, they are concerned with the control of changes to production, *after a design change has been decided.* The control of consideration of *design* changes is something else entirely. So software designers were being asked to give details of their proposed change which they could not possibly do until they had actually designed it and convinced themselves that it was satisfactory. In software, though, that means 'until they had actually implemented the change and tested it'.

The forms could be completed *only* retrospectively, and therefore perfunctorily - which means probably badly and, often, plain wrong.

Some might say that even if the quality system did not properly *control* what was done, at least it would provide the information to know *what* was done. Oh yes? It only took *three months* to find the right sources so that the accepted system could be rebuilt from registered library components.

And hereby hangs another tale... To some extent the difficulties experienced in rebuilding the system from library components was because the procedures, such as they were, had anyway been ignored in the heat of last minute panic before customer tests.

In recognition of this the quality department had accepted a streamlined procedural system for use during tests and trials, but this was also being ignored. This time it was because of the cavalier attitude of the front-line test-support team who felt themselves to be above such things. After all, they knew what they were doing, and what they had done: nobody should interfere with *them.*

Meanwhile, just before each test, the software project manager was faced with both the customer and his own management asking for eleventh hour changes, while his own team were unearthing bugs and proposing fixes. *And* he was supposed to know what was going on.

Procedures with purpose
And that was the starting point for the solution - to recognise that the project manager had *really* to be in control. Faced with all the demands, he needed to have at his fingertips knowledge of what options were available - what he could and could not do, both technically with respect to the software design, and in terms of deploying effort to different tasks.

New procedures were devised. They were simple but effective. They were based entirely on the needs of the project manager to be able to answer questions about possible rescheduling of changes. Some paper was involved, but not a lot. The emphasis was now on real *information*, not on *documentation.* And the information demanded from the team was now driven by genuine need from the top, rather than by some abstract idea of what documentation 'should' be provided. So those who had to provide it were brutally made aware of that need. They soon learned to cooperate.

The quality manager took some convincing - or telling. Reluctantly he accepted that photocopies of the project manager's information - basically three sheets of paper, with many annotations, were all he was going to get. He did not readily accept that these were the best records he could possibly have, and that they were worth far more than all the filing cabinets full of (literally) meaningless forms.

From the need for information, there were also implications for the information flow. These led to other major changes to the structure of the project team, which are described further later under the topic of project management.

MANAGEMENT
Running on from the organisational issues associated with quality management, we will first investigate some general aspects of organisational structure and management, before returning to the specific question of project management.

Organisational structure
One of the criticisms levelled at the project in an external audit during its last few years was that the software team was separate from the rest of the project. It was, in effect, acting as a subcontractor. In consequence there had developed an 'us and them' attitude between the software division and the division holding the main contract.

This was to some extent true. Right from its initial establishment it had been a bone of contention within the company. However, after an acrimonious paper war just as the project was beginning, 'The Programming Group' was born.

In the later criticism of the separation, the benefits were forgotten. By having a single group responsible for all the software in the company's different projects, this 'programming group' was able to develop a critical mass which could sustain a proper infrastructure of a development bureau, a software support group, the quality department, the training school, and a research and development team.

It enabled a career structure for software staff, who would otherwise be a difficult fit in a company dominated by electronic engineers. A corollary was that it had, to some extent, pay scales of its own, in recognition of the general shortage in supply of suitable software staff. This did cause some problems of resentment elsewhere, but it was felt to be better to have staff and put up with the resentment than not to have staff.

And if all these failed to satisfy an individual software engineer, the organisation was resilient to staff changes.

Project management
Turning now to project management specifically, there were, in addition to the handling, avoidance, or otherwise of the problems described earlier, other difficulties. These were primarily to do with project scheduling.

Early in the project there arose what seemed like the sensible idea of software prototyping. Even today there is discussion and argument about the appropriateness of different approaches to prototyping. Then it was thought that there should be just a few, discrete rather than evolutionary, prototypes. Each new prototype would learn from the previous, in a process of refinement. The first would be simply to evaluate functionality. The second would accept changes arising from the first and test them with realistic-seeming hardware, but not the actual kit. Finally there would be a 'final draft' for use with near-production hardware, as the basis for the deliverable system.

This all sounded fine - until the project schedules were collapsed. So the second prototype development began long before the first was finished - even had it not been over-running. There was then no chance to use lessons learned from the first in the second. The two developments continued in parallel, with the first becoming more and more of an academic exercise, necessary for the supplier to meet its contractual obligations and to get payment; hated by everybody for consuming resources (ie people) and diverting attention from the main goal.

But even *that* lesson was not learned later on. Some years later, the same thing happened again, with knobs on. The next prototype was, of course, late: and the overall schedule was foreshortened because a trials version was required early. So the concept of a prototype was abandoned, and the prototype already on the stocks transmogrified into the first of a string of deliverable versions, now relabelled phase 1, phase 2, etc.

This all sounded highly plausible while work proceeded on the first phase, and indeed even during the early days of the second phase. It was only during the middle of the second phase that it was realised that the software delivered from the first phase, and now looked after by a separate team, had evolved considerably in response to feedback from trials.

Of course, the first phase team were supposed to keep the others informed of all the changes. But we have already heard how successful the procedures had been. Perhaps they would have got round to filling in the paperwork some time. But you know how it is under trials conditions...

However, there was now no way that the customer would accept a second phase deliverable which did not incorporate the changes incorporated into the much evolved first phase. Of course, to change the second phase to incorporate the first phase changes would have been a mammoth task, even if the trials version were not changing daily.

And after the second phase, waiting in the wings, were the third, fourth, fifth, ...

Eventually the management team bit the bullet. The second phase was, as a separate and parallel development, abandoned. The first phase was taken in hand and controlled properly. The second phase functions were designed as add-ons to be gradually incorporated along with changes arising from trials and with other changes in the customer requirements arising from the changing world of the customer.

There was now just a single stream of development - incremental, evolutionary, development.

At about the same time, major structural changes were made to the project team. The impetus for the changes came from the new software project manager who had inherited a deep, compartmentalised, hierarchy and who wanted much better visibility and understanding of what was actually happening on the project, and of what relationship it bore to the present requirements. This was connected with the new approach to configuration management, which demanded real information, rather than mere documentation (discussed above under ‘quality assurance’).

The new structure was very ‘flat’, with a loose collection of ‘task force teams’ grouped by the different functional areas of the system. Senior subordinate managers who would usually be assigned sub-projects within a hierarchy were instead given functional roles - responsible for knowing precisely, at any moment, the status of the present work packages, the availability and suitability of staff for new work packages, the available flexibility for rescheduling, the financial position, and the technical position - in depth, not just that it was said to be alright, or that there were problems, but the precise nature of any technical problems.

The senior technical staff responsible for knowing this last set of information would, on many other projects, have been promoted into administrative positions in the hierarchy. Now they were placed in the various functional areas as team leaders. (Though these were relatively low-level in hierarchical terms, it was clear to all that they were by no means junior.)

The project manager could now know the precise *actual* position at a moment’s notice; was able to discuss new and changed requirements with his own management or with the customer on the basis of good information; and knew and was able to discuss technical problems as they arose, with the support of top-flight technical advisers who had equally good information.

This approach is not recommended as a recipe for all projects and all project managers. It was just one way of achieving *real* control, rather than a paper pretence, and one way of achieving proper project communication (see later section on "what might be done").

In addition, the structural changes included a clear separation between the development 'task force teams' and the integration team. This removed the bottleneck which conventional integration techniques had imposed. This in turn enabled a dramatic increase in the effort which could be applied to the development - by an order of magnitude.

In parallel, and in addition to the major changes to configuration management procedures discussed earlier, there were detailed alterations to working practices - like removing all private temporary storage so that all work went through the project library.

Within six months of these changes the next software delivery took place on schedule - the first time this had happened since the minutes-before-the-deadline delivery of the proposal, ten years previously.

From then on there would still be problems. The incremental development was not always as flexible as was hoped, since the sequencing of the incorporation of some features was sometimes quite critical. Some things could not be done a bit at a time. Some were just hard. But there was no longer the threat of tens or hundreds of man-years of work simply being shelved.

Except, of course, that that is precisely what happened to the project as a whole.

ANALYSIS AND DISCUSSION

Did it go wrong?

Was it a disaster? If a disaster is something which adversely affects human beings then yes. Maybe the politicians, empire builders, and plain dimwits simply got their due comeuppance. But the majority who suffered sweeping redundancies, or other more subtle damage to their psyches and careers, were honest hard-working engineers who had achieved great things. Perhaps, though, we should reclassify it as a natural disaster, given the combination of perversity and inevitability.

Why did it go wrong?

Why was it a disaster? Well, it was not at source the software, nor any of the other commonly quoted culprits. Certainly there were things wrong with all of them, but that is to ignore the many things which were right. Now getting some things right - even, with cooler retrospection, a lot of things right - can hardly be an excuse for getting others wrong. But it does indicate that the problem lies deeper than a simple change of management team. After all, these were mostly well-qualified, well-intentioned, well-motivated people, intrinsically capable of making decisions at least as sensibly as their critics.

To say simply that if this had been done or that had not been done then the project would have been a success is flawed in two respects. Firstly, it is to ignore the beneficial aspects of many of the decisions, without suggesting how those benefits might otherwise be obtained (the grass is greener effect). Secondly, to say that decisions could have been better is to beg the question of *how* they might have been made better.

It is difficult to see what might be done better to foresee the undesirable side-effects of otherwise well-considered and sensible decisions. It is facile, though perhaps true, to talk of learning from experience and consulting widely in order to tap into the experiences of others.

Parochialism and politics

So what of bad decisions? The case study illustrates that, in general, the problem is not *directly* that people make the wrong decisions. They do this as a *consequence* of approaching the decision from the wrong direction. They look after their local interests, acting parochially. Sometimes this is *because* they forget or are never aware of important considerations outside their own area; sometimes it is perhaps *why*.

‘Parochialism’ may, though, sound more pejorative than intended. It has overtones of deliberate ill-will to others, but that is rarely the case - at least in the beginning.

There are *some* bad guys, who start out by seeing a project as one round in their fight to develop a career *rather than* a system. It is not that they want to do a disservice to the customer, just that doing a good job is not their number one priority. So why not build a career by building a good system?

Well, on projects which last more than ten years, in an industry in which career steps - and even salary assessments - may be no more than six months apart, being seen to perform in the short term can easily be more important than the real thing in the long run. ‘Performance’ is the right word - as in *acting*: they are the corporate politicians.

But the majority at least start out with goodwill to all men.

For an analogy, look at life on the farm - a coherent food-production organisation, when viewed abstractly and from a distance. Now step inside. The pigs see other pigs and a sea of mud. They may through cracks in the pig-sty door see a glimpse of the world beyond - perhaps the side of the sheepshed - and wonder what sort of pigs there are on the other side. The sheep see other sheep, some straw, and, at lambing-time when they most want to be alone, a lot of human heads. The farmyard geese see quite a lot as they strut up and down. Sometimes they see the horse’s head sticking out of the stable door. Given its size, they muse on the strength of its two webbed feet. The battery hens don’t see a lot. And we have all heard the one about being a mushroom.

So it is on a project. For all the best reasons we encourage small teams to have a strong sense of identity; we minimise the number of people reporting to each line manager; to avoid confusion we insist on formal communication channels - the pig-sty doors. So we create deep management hierarchies and the mushroom syndrome. How many times have senior managers been appalled that their junior team members did not even know who the project manager was when he visited (let alone the customer).

So both managers and members of teams within a large project end up pursuing local goals. What is this but modularity, and separation of concerns, after all? - a good thing, surely? It is not (at least not always - see later) that they would not *like* to identify with the overall project goals. Indeed they perhaps think that they do, but as in the farmyard, who knows? In its mild form the symptoms are no more severe than parochial decision-making, though as discussed above the technical consequences can be very damaging. But there is only a thin line between parochialism and politics. It takes only a little resentment between individuals or teams for the one to degenerate into the other, and for long-running feuds to form and fester.

And then, to compound the difficulties, things go wrong. Even if each manager had not been set an impossible task, delivering to time and budget is rarely easy and there is rarely any contingency for any difficulties which might arise. Once the budget is blown, careful, steady, conscientious management is not easy. Getting away with the bare minimum and justifying it becomes more important than the customer’s needs or even the boss’s.

So even the good guys can, under pressure, be more concerned with self-preservation than cool, honest, appraisal of the situation. Who can blame them? Only those who have never been there. And can anyone tell which are the bad guys and which the good guys in a bad mess?

And these last two problems - of both bad guys and good guys under pressure - are not restricted to teams within large projects, but to the senior project managers and company managers as well.

Politics and parochialism affect not just suppliers, either: customer organisations can be equally afflicted. We talk simplistically of ‘the customer’, but in large organisations there may be several departments responsible for different facets of the acquisition of new systems. (Think of the twenty-odd on the case-study requirements committee.) There can, for instance, be a continual tug-of-war between end-users and their centralised ‘buying department’.

On one project, worse even than the case study, these customer departments could not even agree on which one of them was legally responsible for the contract. The supplier could not find out who to sue for non-payment, after difficulties with delays and disagreements on the specification.

In these situations a project can be just a pawn in the political manoeuvring of different customer departments. Even if it does not start out like that, when things begin to go wrong the same forces as in the supply side will start to surface. There will be those trying to save face with their superiors and colleagues (ie competitors in their own rat-race), and those trying to make capital out of the problems. Those faces which were so friendly during all the fun of feasibility studies can turn very sour when their owners are in big internal trouble.

Perversity

After parochialism, whether of the benign farmyard variety, or of the malignant political kind, comes perversity. This is not, in this context, deliberate wilful waywardness of individuals. Rather, it is the failure of 'the system' - the people system: failure to communicate, despite every effort; corporate forgetfulness. "People squared equals perversity".

Such perversity characterises those examples given earlier in the paper where, with *apparently* good communication between teams working together, and nobody acting parochially, still things went wrong.

In some cases the requirements evolved, invalidating the original design assumptions. Hence the basic form of the design was no longer appropriate. Then that form of design became a shackle, constraining both performance and the further enhancement of the design.

In others, where the design assumptions were still valid, the design evolved for other reasons. In doing so it diverged from the original design concepts. No-one remembered them or recognised what was happening. So the bastardised remnants of the original design actually degraded performance, and hindered further design enhancement.

Some of the unexpected side-effects of otherwise good ideas seemed quite perverse, too.

It seemed like a good idea to base the software design on an assumption that modern signal processing hardware would produce clean data; like a good idea to use advanced techniques like message-passing software design; like a good idea to reuse existing hardware change control standards.

Indeed it sometimes seems that any attempt to improve things is to tempt fate of the perverse kind. We know and try to anticipate the obvious opposition from reactionaries and entrenched NIH. But there is also a basic problem with novelty itself. New approaches are simply not understood. In a way they are not even understandable if they are not part of the 'culture' of an organisation or the discipline. The new ways are not easy to adopt. Applying them can seem contrived. It may be easier to choose a 'worse' solution which is 'understandable' (ie personally familiar) and which conforms with the existing culture (ie corporately familiar).

But these are not the *really* perverse problems of novelty. *They* arise when the new techniques *are* applied, but without understanding. Thus the strange, but perfectly well-structured, program module.

SO, WHAT MIGHT BE DONE?

A problem in perpetuity?

The historical pattern is that we begin with an individual. This individual designs or in other ways works with a set of mental building bricks - concepts, or even physical components.

To do more than can be achieved by one individual, we form teams, and teams of teams - big projects.

From asking a single monkey to write "Hamlet", we now expect co-operating teams of monkeys to deliver the Complete Works of Bertrand Russell.

We can improve the process in a variety of ways to enable an individual or a team to do more, or the same but better. We can give the individual 'bigger' concepts or components. We can develop design techniques which enable the individual to make better use of existing concepts (such as structured programming). And we can try to better order the organisation - in the most general sense, including the project organisational structure, the development schedule, the procedures.

The biggest advances, since they allow equivalent sized and organised teams to do better, come from 'bigger concepts'. But, in general, individual team members, project managers, and even companies have little control over these. They evolve from the industry at large.

Individual organisations may have their in-house design techniques, and usually have their own approaches to company and project organisation. But, as we have seen, conventional procedures, however carefully conceived, are not necessarily enough for large projects. So it *is* worth looking for generically applicable ideas, not encapsulated in standard management techniques, about how to do things better.

It might be argued that the problems will go away as we give designers bigger and better concepts and components, since they make projects smaller and simpler. But that would require that we exhaust the capacity of application domains to find new and ever more complex requirements. Yet there is little evidence that we will cease to stretch our capabilities, and sometimes to over-reach them - to bite off more than we can chew.

There is even an argument that it is good and stimulating for society to take on at least some high-risk projects. But high-risk of some means almost certain failure of at least a few. Which customers will volunteer for disaster projects in the interests of cultural enhancement?

So what does make big projects go right?
Looking at *good* decisions and well-managed projects and working out who were the people behind them, the important factors seem to be luck, together with parochialism, power, personalities, and so on - all the same factors - which by, happenchance, happen to lead to success rather than failure. In other words - people, again.

But there do seem to be some common success factors. Not counting luck, two stand out. The first is a good project manager; the second that the team have done it - or something like it - before. These tend to be discounted by technologists, perhaps because they have so little control over them - like the adage about choosing one's parents more carefully.

At first sight there are no obvious connections between these factors and those which led to the mixed bag of successes and failures of the case study. But what is it that the 'good project manager' *does*? We talk of 'a strong sense of purpose', 'commitment', 'knowing what he wants' coupled with an ability to communicate those things and to motivate everyone else.

Moreover, this is a two-way process. The 'good manager' will also be a listener - able to pick up the 'vibes' from the project team; able to sort out the wheat from the chaff of project reports and meetings; able to appreciate what he is hearing so that his actions are based on understanding rather than rote and rule-books.

Is this not just one approach to overcoming the problems of parochialism and lack of understanding of, and identification with, the real customer requirements and with the design concepts? In other words, a 'good manager' may be able to achieve the effects which we seek. We must try to achieve the same effects without necessarily having any choice over the project manager.

Out of a project employing several thousand people, perhaps only a handful *really* appreciate the customer's requirements; only a handful really understand the design concepts; and if we are lucky these handfuls may overlap. Many of the problems of the case study were a consequence of a breakdown in (effective) communication of this appreciation and understanding. A 'good manager' is one way, and we may always be better with a good manager than the alternative. But if we recognise what it is that he does, then we may be better able to facilitate the achievement of the same effects in other ways.

In a way, that is one of the purposes of design reviews and requirements reviews - to ensure that proper communication has been achieved. When viewed in this light it becomes clear why it is so important to have customer participation at such reviews (at the appropriate level). It also becomes clear why they should not be treated as an administrative formality.

But there are more fundamental things we can do which can have a bigger effect. Thus the 'flat structure' described above for the software team, towards the end of the case study.

The purpose of that structure was originally to give the software project manager hands-on control - contrary to received wisdom on management style. Not only was that achieved, the benefits were much more.

The 'trusties' embedded into the development teams were placed there to give the project manager good information. But clearly this worked both ways. All members of the development teams were much more aware of the customer's needs, of the design strategy, and of how their own work contributed and fitted with the overall design and the work of other teams. Moreover, morale was thereby improved.

There may be other ways of achieving the same effect: one alternative is to have a 'flying squad' of technicians who are made guardians of the design, its integrity, and its coherence with the (perhaps evolving) design concepts. This has the additional advantage of separating the roles of technicians - perhaps not the best managers - from management. It may have the disadvantage of separation and isolation from the line management - it will need a determined project manager to make it work. In a way, the 'trusties' of the case study fulfilled this role, but without becoming remote from the line.

Turning now to the second oft-quoted success factor, should only those organisations familiar with the type of work undertake new projects?

Let us set aside political considerations such as the drive to open markets up to new suppliers, to provide opportunities for small and medium-sized enterprises, and generally to encourage competition. Let us further set aside concerns that innovation would be inhibited. Let us instead ask what is meant by 'corporate familiarity'.

Even if a supplier has done a similar large project before, it would probably also have been a lengthy project. And an organisation is a collection of people who, in this fast-moving industry, will mostly have moved on - as will the technology which they used, and indeed the operational requirements. So we are unlikely ever to find the *same* project team which has even done the same *sort of* thing before.

What we have instead is a collection of people who might know something about some aspects of this sort of job. The message is the same - we must organise to communicate this knowledge, rather than leave it locked up in a few isolated heads. (Though care must be exercised so that over-conservative individuals do not stifle innovation.

But: as the case study demonstrated, *documentation* - however thorough, and however carefully design decisions and the reasoning behind them are documented as well as the design itself, and even if there are no hidden assumptions - is not enough if no-one will read it. The organisational structure must be such that those who have the knowledge, or who know it exists, are able to make sure that it is applied effectively.

So far we have talked of ways in which we might get the best out of people, their knowledge, and the technology they use. An alternative approach is to attempt to build into the management and technical organisation mechanisms for monitoring and feedback. So we would have a 'closed-loop' development system, rather than rely on the 'open-loop behaviour' of the project team. But this is what quality control is supposed to do and, as with quality control, it might sound fine on paper; it *might* even work in 'peacetime'; but it will probably break down in the heat of action, when the urgent takes priority over the important.

We should not expect procedures based upon some ideal, but unrealistic, model of development to turn untidy reality into the tidy ideal. And it is often at the very time that control is most needed, that the control mechanisms break down. We should avoid *reliance* on monitoring mechanisms. As in so many control systems, the greater the dependence on feedback, the greater the risk of instability.

We have heard how the case study was awash with procedures and forms, driven by the needs of certification. But the forms did not say anything useful, and the procedures did not help anyone actually *control* anything. The institutionalisation of the procedures - leading to abrogation of responsibility to a separate quality department - diminished their value still further. Finally, they were ignored anyway in the heat of last minute panics. But this last should not really be seen as an additional problem: it was more a *consequence* of the contempt felt for the official system.

It was only when the project manager took full responsibility for both his own needs for control *and* the needs for quality assurance that the procedures and paperwork were redesigned to provide, use, and record the real information required for the real decision-making. *Then* there was real motivation to make the procedures work.

So there is a positive conclusion here. Procedures should *primarily* be designed not to check that the right things are done, but to facilitate the doing of the right things.

SUMMARY

We spend a lot of time agonising about which techniques and tools to use in future - and even whether to suffer the trauma of a mid-term change to solve today's project problems. But who cares what colour we paint the hangar when we should have been building a ship?

Most designers can do a good job with the techniques and tools they have to hand, as long as they understand their capabilities and limitations, and as long as they understand what they should really be doing. But there are no technical fixes for problems of understanding, nor for fundamental organisation and management problems which inhibit that understanding.

It is sometimes said that all we have to do with a large project is make it a collection of small projects. The argument in this paper is that that is not enough. It is not even enough to make it a collection of the *right* small projects. It must be more than the sum of its parts, and that added value has to come from communication and cooperation, and from structuring the project, the project team, and the procedures to achieve effective communication. Else the sub-projects will, first through parochialism, and then politics, and peppered with perversity, fragment and fail.

Final footnote on the 'software problem'

The first instance of a 'software problem' given in this paper was the difficulty that the software had in coping with more than two orders of magnitude more data than the design target. To conclude, here is another 'software problem'.

One day a serious software fault was reported. Apparently, the screens had simply gone blank during tests. From the subsequent 'fault investigation' the following story emerged.

During previous tests the customer had complained of an annoying rattle in the metalwork of the system. The mechanical engineers had cured the rattle by adding a metal brace to the offending panel.

Unfortunately this brace passed across the duct for a cooling fan. The fan was that for the main computer which, some time during the later tests, overheated and quite properly tripped out, switching off automatically to avoid damage. The screens went blank and in the history books - in this case the system fault log - was entered another statistic - yet another 'software problem'.

--ooOoo--

Bob Malcolm

Bob is managing director of the consultancy ideo ltd, which provides unbiased advice on research strategy and management for a range of private & public sector clients.

Before becoming independent in 1989, he worked for twenty years in systems development. From 1978, he was Chief Engineer of the Airborne Software Division in GEC Avionics, managing their software engineering research and development for future application systems. Later he took responsibility for the software in the Nimrod Airborne Early Warning system and the Fox-hunter radar system for the Tornado aircraft. During his time later with CAP Scientific, he established their Research Centre, before becoming Research Manager for CAP Group and Sema Group (UK). He has worked as systems engineer, research manager, quality manager, project manager, and business manager.

Bob was Project Director for the DTI sponsored study of Software in Safety Related Systems, chairman of the associated all-industries working party, and then programme co-ordinator for the £40M Safety Critical Systems Research Programme supported by the DTI and the EPSRC. Subsequently he co-ordinated the Systems Integration Initiative on behalf of EPSRC, DTI, and MoD.

Bob is a member of a number of professional organisations. He is a former chairman of the Research & Development Society, inaugural chairman of the IEE (now IET) Informatics Division, and has chaired a variety of other professional, industrial and academic groups.