

## **Legacy systems management**

*Rebuilding the plane in mid flight.*

**Charles Brewer**  
**Managing Director**  
**NaMax Limited**

## **Introduction**

1999's expectation was that within a couple of years, all interaction with computers would be via the Internet. This view has now virtually disappeared, but it has left behind a realisation that the systems built from the 1970s to the 1990s were designed with a set of profound misconceptions as to the future nature of computing. The rise of mobile computing, 24 by 7 operations (or at least, interaction for longer than the working day), the rise of web services (or at least services not provided by the mainframe) all challenge, and ultimately defeat, the batch architecture which underpins most large scale in-house systems.

Replacing these legacy systems has proved unaffordably expensive, and has usually resulted in total failure as the complexity of years of changes to business models and regulatory requirements has rendered the logic of legacy systems impenetrable.

But there is an escape. In this paper I shall outline how organisations can retain those elements which work well, and come to terms with the new demands of business to have extended operating hours, mobile computing, web access and all of the rest; and to finish with something more flexible, cost effective and resilient to future change than any other architecture.

## **The history of a problem**

The UK finance sector has always been in the forefront of information technology. This is, of course, both a strength and a source of weakness. On the one hand, new systems can bring competitive advantage either through better evaluation of business and superior information for making decisions or elimination of manual processing, on the other, we have systems built when there is little or no understanding of deep design issues and where systems were often built to a date, not a standard.

And of course, the developers at the forefront didn't have the option of buying packaged solutions; they built the systems which became the basis for the packages. In the base scenario, when a software house took over the system, it rebuilt large sections, documented the design, migrated it to the latest version of the operating systems, and designed an API set which allowed the system to use any one of a number of databases, not just the one for which it was built (and definitely didn't use those undocumented procedure calls of which the original developers were so proud, but which didn't exist in the next version...).

My career began when many of these big mainframe systems were being developed or implemented. There were certain things we all 'knew'. These included:

1. All the systems being put in today will be replaced in 5 years
2. Memory is expensive, so it is important to compress code as far as possible even if this makes it hard to understand
3. CPUs are slow, so using bits of assembler here and there will save a few CPU cycles and delay the cost of the next upgrade
4. At some point, there will be a method of programming so simple that even users will be able to create their own applications
5. Only properly security-checked employees will ever have access to terminals connected to the systems
6. Systems only need to be available for the working day, and only in one time zone
7. Weekends can be used for maintenance.

Some of these were disproved earlier than others. There was never money available for replacement of the old systems, if they worked reasonably well, there was only money available for new things. Moore's Law got going and meant that three year old systems looked like (and performed like) Model T Fords next to the latest Ferrari. Instead of chatty (verbose?) COBOL being the way of the future, we got C++.

But it is the last three which have both stood the test of time best, and which also threaten the biggest upset to established ways of doing things.

## **The castle and the market**

Older systems were build like fortresses. Massive defences surround the area to be protected, but once you are inside, security is pretty relaxed. The analogy would be that even to gain access to an old mainframe system you

would probably need to be an employee of the owning company, but that once you were inside the system, local security would only be whatever the local developers had put in place. Since there are no outside links, there is no possibility of being hacked from outside.

But remote working, sales or enquiries over the Internet, distributed services, links to payment systems and so on have all changed the security and operational landscape totally. Now the environment is much more like that of a market. Anyone may enter – even if they do so with malicious intent – indeed, the more who enter the more successful the system has been – but all are to be watched constantly, and the tasks they may undertake are strictly limited.

Castles typically had a ‘working day’ and at curfew, the doors were closed, drawbridge lifted after which time no-one could enter. Those already inside might carry out certain tasks, but essentially all contact with the world was cut off. In our analogy, this was equivalent to ‘running the batch’.

But now the systems may be open for business 24 hours a day, 7 days a week. Indeed, Sunday afternoon is potentially one of the busier times for people to buy insurance or sort out the household bills on the internet or over the phone.

Now you might find some way of beefing up security and limiting the transactions available to certain classes of user, but you are going to have to do some pretty radical surgery to allow data entry and motor insurance rating on a mainframe when all of the files are locked for batch update. How are you going to make sure that the renewal calculations for 10 000 policies are not held up while some 15-year old commandeers the rating engine trying to get a quote for a non-existent Porsche while pretending to have 24 points on his (also non-existent) licence?

### **A new beginning**

It is estimated that over 70% of major systems developments fail, and there do not appear to be any examples of large scale replacements of core systems in UK banks and insurers in recent years. Even amalgamation of systems in the aftermath of mergers appears to be a rare and immense undertaking.

Furthermore, most of the systems on offer stick remorselessly to the old paradigms, even if in some cases graphical front ends have been glued on, as often as not this is the classic ‘lipstick on a pig’ solution. In the PC world, we saw this kind of activity after Windows 95 appeared, when market leaders such as Lotus 1-2-3 and WordPerfect tried desperately to product Windows versions without re-engineering, resulting in underspecified, unreliable, underperforming systems – and handing desktop dominance to Microsoft.

So it would appear that replacement of legacy systems is often not an option – especially in today’s markets where financial institutions no longer have seemingly infinite funds to pour down the drain of IT innovation. So what is to be done?

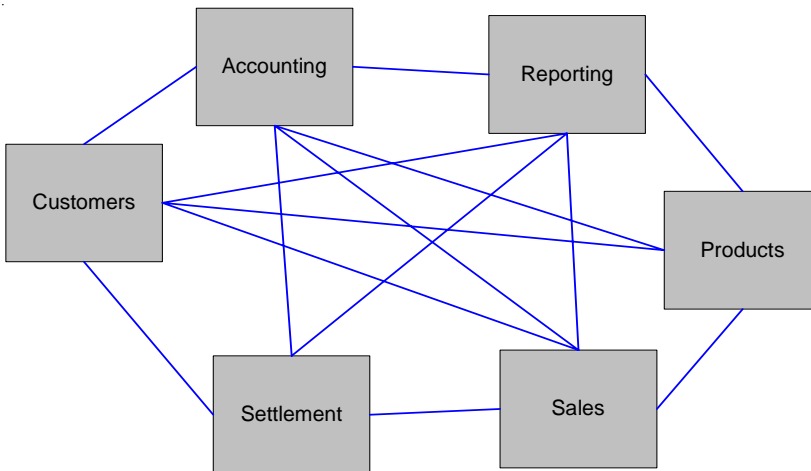
There are two answers to this question. One is to do with mindset, the other with technology.

The mindset issue is both easier and more difficult. It involves recognising both that the existing systems are entirely inadequate, and that they are better than any alternative, and cannot be replaced. This paradox has paralysed large scale development for years, and has led to the disappointments of CRM, the Internet and many outsourcing initiatives.

There is no way that a 1970s mainframe or AS/400 green screen application is ever going to support use by home users. There is no way that a new SQLServer based, J2EE compliant, thin client system is going to handle the complexities of back-dated mid term adjustments to motor fleet insurance policy just the way the old one did. What we must do is recognise what the old systems did well, and keep them doing it, and recognise what they cannot do, and build new systems to carry out those tasks. Then we must make these systems work in harmony.

The technical answer has been around for a few years, but is only now starting to take on the maturity which means that it can start to be considered mainstream. In technical terms, it’s middleware. Use of middleware can provide methods both of replacing legacy systems and retaining them for use as service providers. Either way, you get a low risk, low cost method of aligning what IT shops provide, with the requirements of the business.

In a typical software environment, where there are multiple systems all communicating. Even within a mainframe environment, it will be possible to identify many different components. In a typical environment, a schematic of systems and interfaces will look something like this:



In theory, all systems might interface to one another, giving a total number of connections of  $\frac{n!}{(n-r)!*r!}$ , where n is the number of components, and r is the number of ends, that is, 2. However, as we know, although sometimes there are no connections, sometimes there are more than two connections between individual components. We may thus have an unlimited number of connections. Even with this simple case, the number of connections rises rapidly with the number of components.

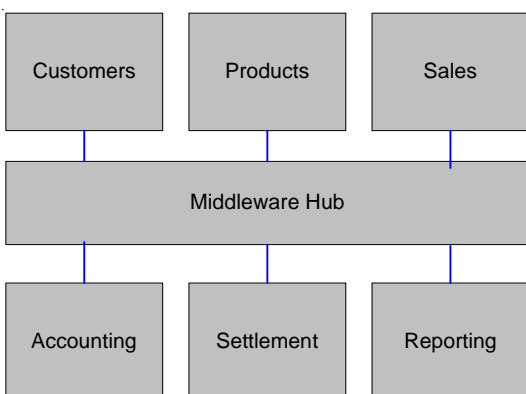
The following table show how the number of potential connections grows with the number of systems

Systems	1	2	4	8	16	32	64	128
Interfaces between systems	0	1	6	28	120	496	2016	8128

Each connection must be maintained, and if the organisation wishes to replace one of the components, all of the connections to that system must be replaced. We often see cases where the cost of a new system is dwarfed by the cost of its implementation. For example, a new reporting suite might cost £250,000, but each interface costs £50,000.

Once there are 5 connections, the cost of interfaces equals the cost of the system. If there were 20 systems (and frequently we find more than 200 in large organisations with a history of M&A activity), the interfaces alone cost £1 000 000. This is usually enough to kill any project before it even begins.

Using a middleware hub, however, the picture changes. In the perfect situation, the systems architecture looks like this:



Here we can see that the number of connections grows only with the number of systems. Indeed, applications which communicate via the hub do not even need to know what the system on ‘the other side’ is. Provided that the business data requirements do not change, the hub will be responsible for any data transformation for the specific requirements of the receiving system.

In principle, exchanging one general ledger system for another would, provided the underlying accounting principle were unchanged, require no changes to the debit and credit information generated elsewhere in the system.

Grown of connections, and the cost of their maintenance would be of a different order from the ‘spider’s web’ instance above:

Systems	1	2	4	8	16	32	64	128		
Interfaces to middleware		1	2	4	8	16	32	64	128	

Taking our previous example, and reworking it for the reporting system in a middleware environment, we see that the cost of implementing the system is unchanged at £250 000, but that the total cost of interfaces is that of a single interface: £50 000. Even if the new system requires different data from the existing systems, it will be a matter of changing data structures, not trying to make incompatible hardware, operating systems and business paradigms work together.

Furthermore, the timescales associated with linking in new functions is dramatically decreased.

The following two charts show typical development cycles (these are taken from plans and examples my colleagues and I have worked on in the past few years):

The first shows a fairly typical replacement programme for an insurance company system. The first product (that is the first benefit) of the system is delivered around 26 months after the start of the project. The organisation, however, continues to pay for the existing system for a further 15 -18 months after that, and there will still be an issue of residual business resulting from claims, regulatory and audit information and other historical data.

ID	Task Name	Start	Finish	Duration	2003				2004				2005				2006		
					Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3
1	Package evaluation	01/01/2003	01/04/2003	65d	█														
2	Implementation - back office	02/04/2003	01/07/2004	327d	█	█	█	█											
3	Implementation - front office	01/03/2004	01/10/2004	155d					█	█	█								
4	Data migration	02/07/2004	31/12/2004	131d					█	█	█								
5	Testing and training	01/10/2004	01/04/2005	131d								█	█	█					
6	First product live	01/04/2005	01/04/2005	0d															◆
7	Decommission current system	01/04/2005	29/09/2006	391d															█

The second case, which leaves the existing system in place, delivers at least the same back office functionality as the previous version (indeed, we all know that software, like port, improves with age – the older the code, the more bugs that have surfaced and been fixed). It also delivers the first front office product – in a period of 3 – 6 months, depending on the complexity of the initial choice.

Normally, we recommend taking on one of the more straightforward functions to begin with, since this allows all of the ‘plumbing’ to be put in place and fully tested before the more complex business functions are tackled. In this case the estimate is usually at the lower end of the timescale.

ID	Task Name	Start	Finish	Duration	2003			
					Q1	Q2	Q3	Q4
1	Middleware implementation	01/01/2003	28/02/2003	43d	■			
2	Front end development	01/01/2003	28/02/2003	43d	■			
3	Back end modifications and API	01/01/2003	14/03/2003	53d	■			
4	Testing and training	17/03/2003	15/04/2003	22d	■			
5	First product live	16/04/2003	16/04/2003	0d	◆			

### Why now?

What has changed? Why can we do this now, but we could not have done it a few years ago.

First, a recapitulation, the need was less obvious, green screen data entry was still just about acceptable, and the cost and risk of replacement were unacceptable so there was no overriding need. Now however, the legacy systems are clearly failing so action is required.

In IT terms, there are a series of technologies and approaches which greatly facilitate this move. To list just a few:

XML1 – simplifying the transfer of data between different systems

J2EE2 and .Net3 – architectures which deliver and expect components communicating with one another

Biztalk4, Websphere5, Tibco6 – different variations on the hub architecture, but all delivering a layer through which other systems communicate

Web services7 – creation of self-contained environments where complete business functions are carried out.

Only now that we have these environments at the same time can we begin seriously to consider replacement as an acceptable risk strategy. Taken together, these technologies address the issues of reliability – the technology is mature, longevity – now that IBM, Microsoft and Sun all support the approach in one way or another, a source of developers and a commitment to the continuation of the approach is sure; it is rather like when IBM ‘gave legitimacy’ to the PC market which before had been the province of hobbyists and hackers.

The existence of what amount to packaged versions of the development environment also mean that the intrinsic dangers of development – the complexity of the environment, the lack of pre-packaged components, has largely disappeared, and will continue to diminish.

### Conclusion

The euphoria of the early days of the internet has gone, but the expectations which were raised then have not.

Businesses have realised that the days when customers would find their products, understand them and buy them have also gone, they know they need to look out, not in. But just as well-constructed old buildings can prove some of the most attractive places to work or live in once they have been converted, so our older systems prove to be the best for handling those standard business operations which we have always done, and which we shall continue to do.

By using some of the most recent technological tools, we can begin to liberate the value of years of investment in our legacy systems, while adding the ability to achieve shorter times to market for products, obtain better understanding of our customers, and gaining the ability to modify and recast existing products.

The cult of the business leader is also taking a hammering, and businesses are realising that no-one has a reliable crystal ball, so early delivery and a flexible approach to the future are required.

The component based approach outlined in this paper goes a long way to meeting these issues in a way that allows IT directors to be the motors of enablement and no longer the storm anchors of their companies.

Charles Brewer  
NaMax Limited  
www.namax.com

**(Footnotes)**

1 <http://www.w3.org/TR/1998/REC-xml-19980210>

2 <http://java.sun.com/j2ee/index.html>

3 <http://www.microsoft.com/net/>

4 <http://www.microsoft.com/biztalk/default.asp>

5 <http://www-3.ibm.com/software/info1/websphere/index.jsp>

6 [http://www.tibco.com/solutions/products/active\\_enterprise/default.jsp?m=c24](http://www.tibco.com/solutions/products/active_enterprise/default.jsp?m=c24)

7 <http://www.w3.org/TR/wsdl>

(c) NaMax Limited 2002

