



ERUDINE[®]
replaces legacy with flexibility



THE BURDEN OF LEGACY

WHITE PAPER BY DR TOBY SUCHAROV AND
PHILIP RICE

Contents

EXECUTIVE SUMMARY	3
WHAT IS A LEGACY SYSTEM?	3
SCALE OF THE LEGACY PROBLEM	3
IMPACT OF LEGACY SYSTEMS	3
HOW DO SYSTEMS BECOME LEGACY SYSTEMS?	4
1. DEVELOPERS LEAVE	4
2. CODE COMPLEXITY INCREASES	4
3. OBSOLETE HARDWARE AND SOFTWARE	4
4. INHERITED AND COMMISSIONED SYSTEMS	5
MENTAL MODELLING CAPACITY – THE HEART OF THE PROBLEM	5
What effect does it have?	7
HOW DOES THE IT INDUSTRY PREVENT CURRENT SYSTEMS BECOMING LEGACY?	8
WHAT DO WE DO IF WE HAVE A LEGACY SYSTEM?	10
1. MAKE A NEW SYSTEM	10
2. REPLACE THE OLD SYSTEM BIT BY BIT	10
3. LIFE EXTENSION	10
WHAT TOOLS EXIST TO HELP WITH LEGACY SYSTEMS?	11
CURRENT METHODS THAT CAN HELP WITH A LEGACY SYSTEM	11
1. Legacy migration	11
2. Legacy re-engineering	11
3. Integration servers	11
4. System audits	12
Summary of legacy tools	12
WHERE IS THE IT INDUSTRY HEADING?	13
SECOND GENERATION LEGACY SYSTEMS	13
FUTURE LEGACY SYSTEMS	13
WHAT IS NEEDED TO SOLVE THE LEGACY ISSUE?	14
ERUDINE BEHAVIOUR ENGINE	14
BIBLIOGRAPHY	15
WEB DOCUMENTS	15

EXECUTIVE SUMMARY

One of the most difficult challenges facing IT directors today is maintaining and upgrading legacy systems. Legacy is becoming a hot topic: research by the Gartner group show that the main concern amongst IT managers in mid-sized IT companies is integration with legacy systems.

This paper endorses the opinion that large and complex IT projects will turn into legacy systems and that such collapses are inevitable. This bleak view comes from the fact that the forces that turn software into legacy systems have not been fully addressed. New programming techniques and strategies offer only small, incremental improvements in the size and complexity of new projects before a collapse occurs.

The real issues of legacy are all believed to be associated with the 'mental models' built up by the system developers and maintainers. Once a system becomes too complex for the maintainers to understand fully, or when the people who understand it leave the project, the system becomes a legacy system.

In this paper the idea of software that can maintain its own mental model is introduced and offered as a solution to the inevitable trend for software to turn into legacy systems.

WHAT IS A LEGACY SYSTEM?

The IT world has many different definitions of 'legacy'. But all focus on one theme:

A legacy system can be defined as a bespoke application that processes large or complex data that have become cumbersome and hard to maintain. Legacy systems are, by definition, hard to modify or update.

It is worth noting that a legacy system is not necessarily an application that has been written 10 years ago in COBOL – a legacy system may be an application that has only just been delivered to the company by an External Service Provider (ESP).

SCALE OF THE LEGACY PROBLEM

It has been estimated that about eighty percent of IT systems are running on legacy platforms. International Data Corp. estimates that two-hundred billion lines of legacy code are in use today on more than ten-thousand large mainframe sites. It has also been estimated, by the Hurwitz Group, that only ten percent of companies have fully integrated their most mission-critical business processes. The cost of legacy systems as understood from industry polls suggest that as much as sixty-to-ninety percent of IT budget is used for legacy system operation and maintenance.

Legacy systems typically run mission critical business processes. In such cases, legacy systems generally offer excellent accuracy – these systems have often run for many years, allowing time for any critical errors to be fixed. Any replacement or modification to these legacy systems would likely introduce some initial bugs and on mission critical processes, such bugs may be disastrous.

IMPACT OF LEGACY SYSTEMS

The primary impact of legacy systems can be subtle rather than hard numeric values; however legacy systems can cost a company its competitive advantage.

The business owner loses control of the business processes. This is a very worrying predicament. The person in charge of business processes can no longer make changes without the permission of the person in charge of the IT processes!

The company loses its ability to make new offerings quickly. Nowadays there is an increased awareness within the business and marketing communities of the importance of flexibility and agility, allowing companies to take advantage of new market opportunities. The slower the speed of change in the company's software, the slower the ability of the company to make new offerings and react to the market.

HOW DO SYSTEMS BECOME LEGACY SYSTEMS?

There are numerous ways in which an application can turn into legacy:

1. DEVELOPERS LEAVE

Legacy systems are created when no one person is confidently able to alter or maintain the code. Just one key developer leaving a project is often enough to turn a system into a legacy system. In the IT industry the typical IT personnel turnover rate is about eleven months. A key member leaving is not just a possibility, it's likely. IT managers should consider the 'hit by a bus' factor of their projects. For any project this factor represents the number of critical developers who could be lost before the project becomes legacy. If this number is one, which is often the case, then the IT manager should consider placing more developers on the system to increase this factor.

2. CODE COMPLEXITY INCREASES

Legacy systems are created when the current developers no longer fully understand the code because it has been modified numerous times and the clarity of the code has been greatly reduced.

A powerful concept to use when understanding how complex systems change is 'entropy'. The entropy of the system is a measure of how much 'rubbish' or how much unclean code is in it. Any change in a software system is likely to increase its entropy. The entropy increases whenever requirements change, or when there are pressures to finish the job quickly, rather than provide an elegant solution, or when weak developers are in charge of the design.

A vicious circle arises when the system is sufficiently complex that it can no longer be effectively modelled. Once the consequences of change can no longer easily be predicted, and changes can no longer be made that work within the architecture, solutions often become hacks. The code rapidly becomes more complex, often to the point where it can no longer be effectively changed... that is to say it becomes a legacy system.

3. OBSOLETE HARDWARE AND SOFTWARE

Legacy systems are created when the hardware or software platforms on which the systems are written become obsolete. The COBOL language is a good example of a software language that is now obsolete. It becomes increasingly difficult to recruit programmers with the skills needed to maintain these legacy systems.

Historically, mainframes were often required for computationally intensive tasks; these became obsolete in time.

Now commodity processors can be connected by a commodity network, such as Beowulf Clusters and server farms, allowing for massive computational tasks to be handled using mainstream hardware.

With the rise in platform independent programming languages and code translation tools, software is becoming more portable. In the future it is likely that moving code out of obsolete software languages and off obsolete hardware platforms will become a non-issue.

4. INHERITED AND COMMISSIONED SYSTEMS

Legacy systems are created when systems are inherited from other companies or when systems are commissioned.

This creates exactly the same situation as that caused by a developer leaving: it creates a situation where there are no developers available within the company with the knowledge to update the system.

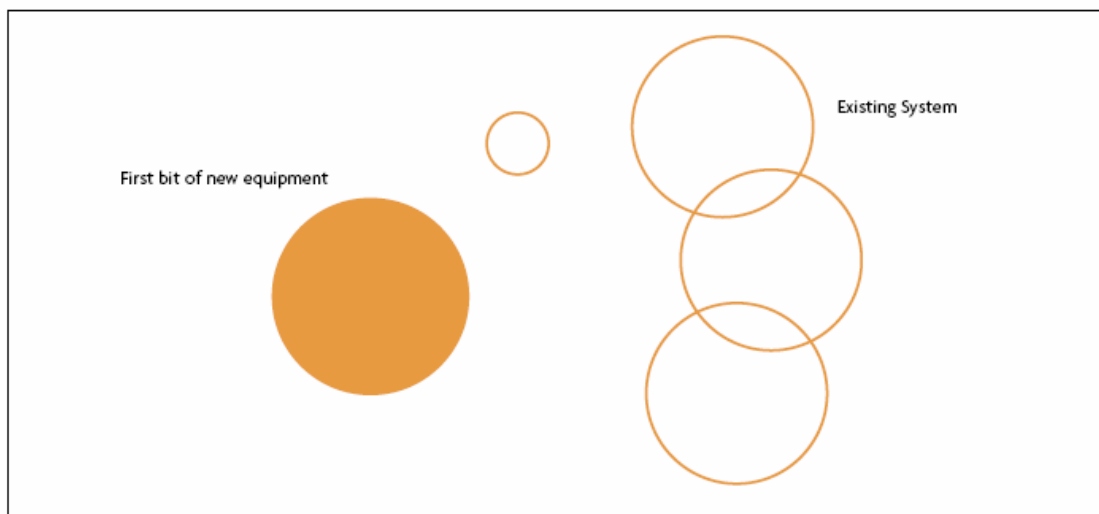
Often the best option for commissioned systems is to maintain the system on a service contract from the original vendor; while this approach can be expensive it removes the headache of having to maintain such a legacy system.

In addition, IT managers may find that they are unable to maintain a base of exceptionally skilled programmers with the ability to create and maintain their systems. Using external service providers may be a better option than spending time and money on building and maintaining such highly skilled teams of programmers.

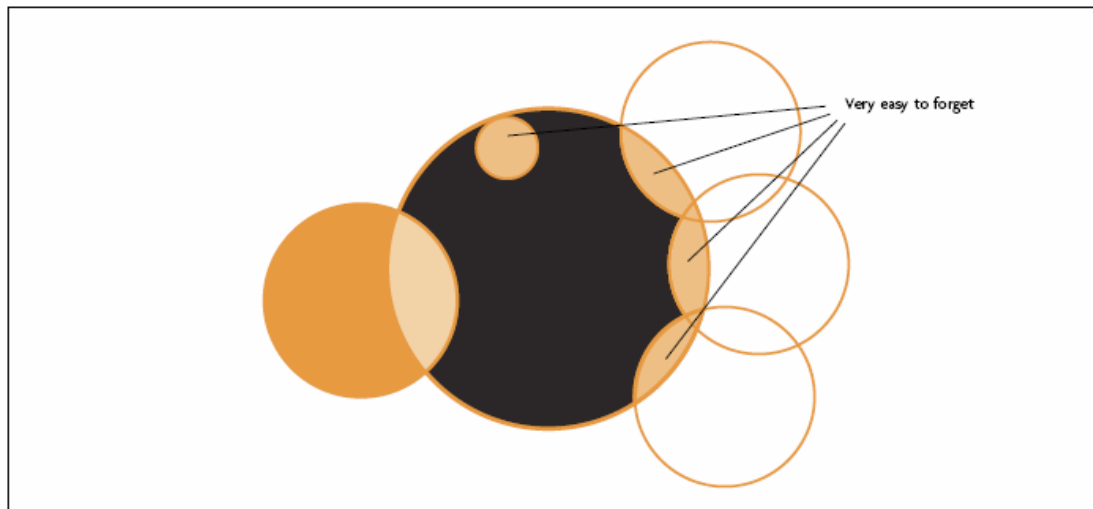
MENTAL MODELLING CAPACITY – THE HEART OF THE PROBLEM

The inability of developers to ‘mentally model’ the software, is at the heart of legacy creation. When a developer leaves; or the code becomes too complex, or a commissioned system arrives, there is no one who knows how the code works or has a sufficiently good mental model of it to make changes to it with confidence.

The diagram below illustrates some of the problems associated with mental models.



In this scenario the developer has to implement some new functionality and make it work correctly with the old system, taking into account conflicts and overlaps with existing behaviour. The orange circle shows how a new separate module of behaviour is added to the existing code. The work proceeds quickly because the changes do not impact the old system.



A second modification to the code (the black area) adds functionality that impacts both the new and old code base, and this is where the problems start. This is often described as 'Integration Hell'. The very pale orange area is easy for the developer to remember as it has been worked on recently in the orange area of code and the developer still remembers all the details of it. The shaded orange code is more difficult: as can be seen there are many code areas that overlap with the old system and these will need special handling.

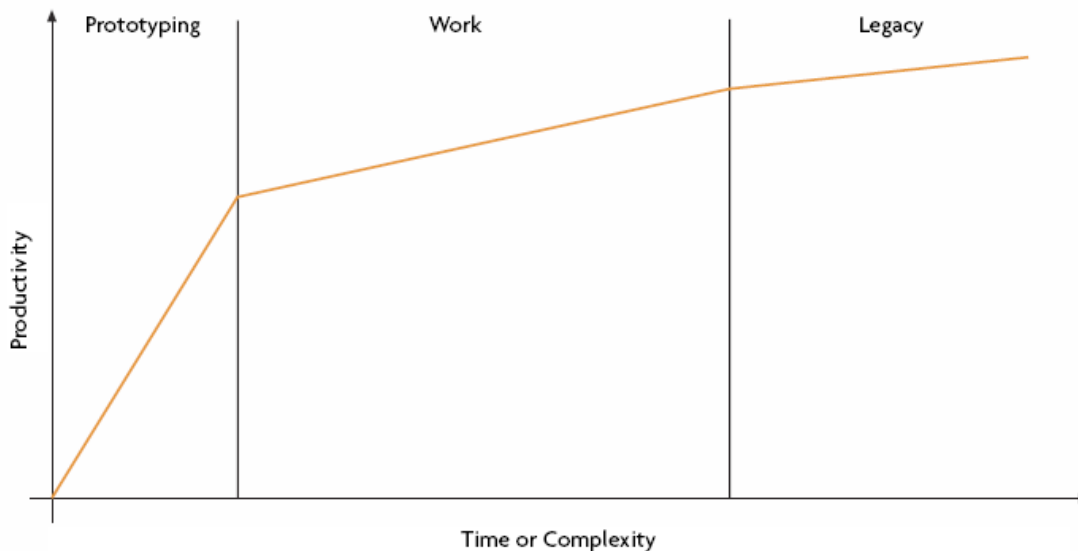
Importantly, if the developer is able to mentally model the shaded orange areas accurately, few defects will be introduced. However, it is entirely possible that the developer is unfamiliar with, or even entirely unaware of, the details of this old behaviour and that such conflicts will go unresolved. As the code becomes more complex than the developer's capacity to mentally model, more and more defects are introduced.

In practice this is grossly simplified representation of code, but the diagram serves as a simple illustration of the problems of adding functionality to existing systems.

What effect does it have?

The mental modelling ability of a developer is critically important in all aspects of software development from the initial design through to code maintenance.

A typical graph of a developer's productivity against time or complexity looks like this:



In this graph we see the initial development progressing at a tremendous rate. This occurs when the developer can mentally model the whole system. There is no time spent on understanding the system, the consequences of changes are readily apparent, and little time is spent fixing defects.

Unfortunately this modelling capacity is like the memory cache on a microprocessor and is often just too small for complex problems. Once it has been exceeded we drop into a work mode, in which productivity settles to a rate that is perhaps two orders of magnitude slower than while prototyping. The reason for this drop is that the developer needs to be continually reminded of how the code works and has to keep 'swapping' bits of the mental model into his 'cache' (to continue the microprocessor analogy). A study of the work in this mode shows that over ninety-percent of developers' time is spent in code maintenance. Of this time, more than half is spent in understanding the code, and most of the remaining time is spent fixing the bugs caused by a poor understanding of the code.

Eventually the system becomes too complex for a single individual to model effectively, and the system moves into the legacy area; the code enters a vicious circle of increasing entropy, decreasing understanding of the code; so making modifying it a very time consuming and expensive job.

HOW DOES THE IT INDUSTRY PREVENT CURRENT SYSTEMS BECOMING LEGACY?

There are several traditional solutions to the legacy problem. Whilst all of them help, none of them addresses the key problem: increasing the mental modelling capacity of the developer, and retaining that mental model when a developer leaves.

The following traditional approaches go some way to addressing some elements of legacy:

- Lots of documentation.
- Design patterns.
- Improved architectures.
- Powerful software development processes.
- Reverse engineering tools.
- Refactoring.
- Test-first development.

'Lots of documentation' sounds like a laudable goal. Unfortunately it is very expensive and difficult to provide. For documentation to assist in the mental modelling it has to describe the code exactly. To do this, the documentation effectively needs to be a programming language; this is the thrust of techniques such as model oriented programming. Unfortunately when the documentation becomes as complex as the code, it has to be maintained in exactly the same way as standard code, and the developers still need to model it mentally.

When a new developer starts working on a project, lots of documentation can help, but more often than not the documentation is found to be inaccurate and outdated. This inaccuracy serves to confuse mental models: a major problem when designing for maintenance.

Design patterns are currently very popular. From the perspective of mental modelling they are probably the best tool that has emerged in the last decade. Design patterns offer one the ability to simplify a description of something using a language of patterns; design patterns provide developers with a common language in which to describe their code.

They allow the developer to make very concise statements about parts of the code, and effectively simplify the mental model. As the model is now simpler, more of it can fit into a developer's mental capacity.

Software development processes include the Rational Unified Process (RUP) and agile software development.

Both of these processes attempt to ensure that the mental models of how the software works are kept alive. In the case of heavyweight processes like RUP, this is kept in documentation and traceability matrices. In the agile processes this is kept in people's heads, and an extensive suite of executable tests.

Improved architectures such as the client-server or n-tier hierarchies are dramatically better than monolithic structures when designing for maintenance. Separation of presentation and content is now universally accepted as 'a good thing', with many designers moving to separation of presentation, persistence and behaviour.

Reverse engineering tools are a very powerful aid to mental modelling. Given a block of code, they produce diagrams and reports about the code. They speed up the rate at which a mental model can be understood, and suffer much less from the problem of accuracy of documentation.

Refactoring is a relatively recent idea, although software developers have been using it for some time. Refactoring is the act of changing the design of the code, while keeping the behaviour the same. The primary goal of refactoring is to reduce the architectural entropy within the system. Effectively, refactoring recognises the importance of mental models, and simplifies the code to ensure that more of the system can fit inside the developer's precious mental modelling capacity.

Test-first development is a very powerful approach to developing systems. It accepts at the start that a developer's mental modelling capacity will be inadequate, and requires every aspect of the system to be thoroughly tested. When changes are made to the code, any unexpected consequences will be picked up and localised by an extensive series of tests, allowing the developer to get to work fixing the broken areas of code.

The effect of all the above approaches is to cause an incremental increase in the maximum system size before a legacy collapse occurs. What is clearly needed is some new way of increasing the mental modelling size.

WHAT DO WE DO IF WE HAVE A LEGACY SYSTEM?

There are three main approaches to dealing with a legacy system

1. MAKE A NEW SYSTEM

The first, most obvious approach is to replace the existing legacy system with a powerful, modern system. This technique was very popular about ten years ago, but it was realised that such an undertaking would carry the same high chances of failure as any other large IT project. Legacy modernisation projects have failed in the past, leading companies to throw out a new project and revert to their legacy system.

When these projects were successful, it was found that such systems turned into second generation legacy systems. Unless legacy modernisation projects address the primary causes of legacy, then these new projects are likely to experience the same forces as their predecessors and turn into legacy.

Complete rebuilds for legacy systems are also the hardest to get funded. It is generally the case that management is not keen to fund large projects whose only payoff is reduced software maintenance problems in the future.

From a management point of view, such large financial investments require that new functionality is added to the current system so that there is some perceived return on their investment. Although it is possible to add in functionality during the rebuild process, it further complicates an already difficult task. These issues may be psychological in nature, but the result is that management rarely funds re-engineering projects.

2. REPLACE THE OLD SYSTEM BIT BY BIT

A different approach, strongly endorsed by the DARWIN project, is to divide the legacy system into modules and replace each module with a modern system, one at a time. Numerous data gateways are constructed that control communication at critical locations throughout the legacy system, which allows developers to add new modules into the legacy system. New modules can be run alongside the old modules to verify that the behaviour of both modules are the same; the old module can then eventually be removed. If correctly done, this process should not affect the functionality of the system at any time.

Such modular re-engineering is very appealing as the system can be updated one part at a time, as money and time become available to each module.

Re-engineering can be an ongoing, or even continuous, process and any failures in this process will affect only the module currently being re-engineered.

When the legacy system is truly antiquated and built using outdated designs rather than using an object – or service-oriented architecture, this approach can be very difficult. Correct placement of the data gateways in such systems is often critical to the success of the re-engineering project. Such gateways are usually very complex as they have to perform additional functionality to allow them to communicate between the new clean architecture and the old weak architecture.

Construction of these data gateways often greatly complicates such re-engineering projects. Even worse, as more parts of the system are re-engineered, the complexity rises, and the data gateways themselves become legacy systems.

3. LIFE EXTENSION

This is the current preferred option, in which systems are coaxed into limping along for as long as possible. External systems are bolted onto them to add new functionality and to interface with the outside world. Most IT directors today regard 'interface to legacy systems' as their major area of concern. The hope is that, in time, a better and less risky approach to legacy elimination will emerge.

WHAT TOOLS EXIST TO HELP WITH LEGACY SYSTEMS?

A number of companies sell toolkits that are designed to help one live with, or replace, legacy systems.

Gartner's research on legacy modernisation tools shows that there is approximately a two-hundred million dollar market for such tools. This figure is not large, considering that legacy systems cost the IT industry approximately sixty to ninety percent of their IT revenue, and it implies that although there are tools available for legacy modernisation, they have not been taken up as enthusiastically as might be expected.

Companies in this sector sell solutions that range from legacy migration to legacy re-engineering to system auditing tools. There is no one company that sells a complete legacy modernisation package; instead a variety of individual companies sell useful and even complementary solutions to solve parts of the legacy problem.

Current offerings do not provide a unified solution to bespoke IT systems. For this reason the revenue of the legacy modernisation market does not reflect the massive costs that are being incurred by legacy.

CURRENT METHODS THAT CAN HELP WITH A LEGACY SYSTEM

1. Legacy migration

Legacy migration is a process by which code is automatically converted from an obsolete language, such as COBOL, to a new platform-neutral language, such as Java. This allows systems to expand easily into new environments and provide services that are strongly supported by modern languages. Migration does not improve the structure of the code: such systems usually require re-engineering before, after or during the legacy migration to transform them completely into modern systems.

2. Legacy re-engineering

There are many modernisation tools designed to aid developers in understanding and modifying legacy systems.

These typically offer the developer methods of viewing the structure of the code or produce diagrams of method calls between objects. Such tools support the developer by allowing greater access to the model of the code, enabling the code to be modified faster and with more confidence.

3. Integration servers

Integration servers provide mechanisms to interact with old legacy systems. Rather than modify the old system, they use the existing interfaces, often designed for 3270 terminal users. These integration servers either provide new improved user interfaces, or expose the behaviour as web services. The better servers allow new functionality to be created, wrapping multiple legacy operations with the scope of single transactions.

4. System audits

System audits are a process by which the code architecture and structure are regularly reviewed. They increase the lifespan of such systems by providing a formal process by which to review the structure of the code. System auditing and regular refactoring are very powerful tools for maintaining the architectural integrity of the code and any IT manager should consider these options as part of their maintenance process. However, when the system has already collapsed into legacy the system needs complete re-engineering and not mere maintenance.

Summary of legacy tools

Although the efforts being made in the industry are helping to alleviate the problems caused by legacy systems, none of them addresses the forces that initially cause systems to become legacy systems. They merely provide a sticking plaster.

In fact, extracting the 'business rules' held in a system still remains very difficult. According to Gartner, "Many enterprises have little desire to open a 'Pandora's Box [sic] of legacy systems". Current trends indicate that most prefer non-invasive extension approaches. It provides immediate, short term and low risk resolution to immediate e-business demands".

Gartner goes on to say that this approach is tactically useful but strategically dangerous, as it increases the complexity of an already complex system.

WHERE IS THE IT INDUSTRY HEADING?

This paper endorses the rather bleak opinion that most large and complex IT projects will turn into legacy systems and that such collapses are almost inevitable. The current approach of waiting is leading to a build up of legacy systems, with increasing numbers of legacy systems critically needing to be changed, but without any low risk approach to updating them. All the approaches described above are tactically useful, but do not address the underlying forces, hence they provide only sticking plasters, rather than tackling the underlying causes.

SECOND GENERATION LEGACY SYSTEMS

An unfortunate trend when dealing with existing legacy projects is the creation of second generation legacy systems. A legacy replacement project often quickly becomes legacy itself because the forces that create legacy systems have not been dealt with: critically the maintenance of the mental model, and the increase of complexity as an inevitable result of change.

FUTURE LEGACY SYSTEMS

This problem is so endemic that the term 'Future legacy system' is now being used. This term is mainly used for infrastructure projects. A classic example would be the Bowman project used by the UK Army, which acts as a network backbone. A wide range of projects will use this, and even while Bowman was under development, it was regarded as a project that would be a legacy system as soon as it was released: hence the term 'Future legacy system'.

WHAT IS NEEDED TO SOLVE THE LEGACY ISSUE?

We believe that in order to prevent systems becoming legacy systems, it is necessary to automate large aspects of the mental modelling of the software.

This automation must enable the system to work out automatically the consequences of a change before making it, and ensure that the developer actually fixes all these consequences.

ERUDINE BEHAVIOUR ENGINE

Further white papers are available from Erudine showing how the Erudine Behaviour Engine resolves these issues (www.erudine.com).

The Erudine Behaviour Engine models the code for the developer. Requirements are captured as executable tests, and each test must always be passed. When making changes to the behaviour of the project being implemented,

Erudine presents to the developer only those areas on which the new behaviour impacts, and forces the developer to resolve them. This is a useful simplification as it presents exactly as much detail as needed to resolve conflicts; the Erudine Behaviour Engine is responsible for identifying these areas and flagging them up.

When a developer leaves the project, the impact on the Erudine project is much reduced. A new developer will still need to learn the domain specific aspects of the job and understand the architectural overview; however the

Erudine Behaviour Engine's automation of the mental modelling ensures that any changes that the new developer makes do not break existing behaviour. The Erudine Behaviour Engine's automation of the model also acts as a very powerful teaching mechanism to explain how the system works.

The ability of the Erudine Behaviour Engine to model software vastly exceeds the capacity of a developer's mental model. This means that massively complex software can now be written. The feedback loop that usually causes the code to deteriorate rapidly as it becomes more complex is now broken.

BIBLIOGRAPHY

Crossing the Chasm, Geoffrey A. Moore, Capstone Publishing Ltd, ISBN: 1841120634, 1999.

WEB DOCUMENTS

DARWIN: On Incremental Migration of legacy systems. Michael Brodie and Michael Stonebraker.
<http://db.cs.berkeley.edu/papers/S2K-93-25.pdf>

CIO Update: AD Survey of Enterprise IT Managers Reveals Top Concerns. Joseph Feiman. Gartner Research. <http://www3.gartner.com>

CIO Update: legacy Modernization Magic Quadrant Helps in Providing Applications for Tomorrow. Dale Vecchio. Gartner Research. <http://www3.gartner.com>

Chickens and Turkeys Migrate, but not Necessarily in IT. Ben Wilson.
<http://www.anubex.com/migrationscentre!chickensandturkeys.asp>

Issues and Challenges Facing legacy systems. Federico Zoufal.
http://www.developer.com/mgmt/article.php/11085_1492531_2

Applications Intelligence Survey. Hal Knowledge Solutions. <http://www.halks.com>

legacy Integration. Something Old, Something New: Integrating legacy systems. Bob Sutor.
http://www.ebizq.net/topics/legacy_integration/features/5229.html

© Remote Operations Limited (trading as Erudine)

No part of this publication can be reproduced, stored in a retrieval system, transmitted or made available to the public in electronic form or by any other means (electronic, mechanical, photocopying, recording or otherwise) without the written permission of the publisher. Whilst every care has been taken to ensure the accuracy of the editorial content the publisher makes no representation and gives no warranty as to its accuracy and cannot accept any liability for any direct, indirect or consequential damage or loss howsoever caused arising out of or in connection with the content of this publication.

Erudine, Erudine Behaviour Engine, the Erudine logo and erudine.com are trademarks or registered trademarks of Remote Operations Limited (trading as Erudine) in the United Kingdom, other countries, or both. These and other Erudine trademarked terms are marked on their first occurrence in the information with the appropriate symbol (® or ™), indicating UK registered or common law trademarks owned by Erudine at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries.

The trademarks of other companies are marked on their first occurrence with the appropriate symbol (® or ™). Other company, product or service names may be trademarks or service marks of others.

In line with Erudine's environmental policy, this document has been optimised for double-sided printing.

info@erudine.com
+44 (0)8456 123 862
www.erudine.com

Central House
Beckwith Knowle
Otley Road
Harrogate
North Yorkshire
HG3 1UF

