Submitted to COMPSAC 2002

Conference Theme: Prolonging Software Life: Development and Redevelopment

Dependency Analysis and Visualization as Tools to Prolong System Life

Dave McComb, Simon Robe and Simon Hoare, of **Semantic Arts**

Stew Crawford-Hines of the **Institute for Zero Defect Software**

ABSTRACT

This paper describes our experience using dependency analysis and visualization as a tool to identify intervention points for migrating applications to environments where they can live out their natural lives, less dependent of the vagaries of platform obsolescence. The first part of the paper describes our methodology and deliverables. The second part presents case studies on applying this approach to commercially installed applications.

Application Life

The typical business application has a useful life of 5 —15 years. We give special status to the venerated long-lived applications that have lived to a grand old age of 15-30 years. According to a recent survey by the Standish group over 30% of new application development projects are never implemented, and die stillborn if you will.

So why is it that a thing that is in essence pure intellectual property and expressed in an electronic medium that has no wear parts and will copy perfectly any number of times, would not last as long as a typical car, let alone a human?

In this paper we will briefly explore the forces that act on application software that make it old before its time. We will look at some techniques for estimating the remaining useful life of an application. These visualization techniques then suggest some specific interventions that can extend the life of the application.

For our purposes the death of an application occurs when it is no longer used; disability is when the application is used, but in a very degraded way, or in a way that does not support the business function in the manner that they could be supported. Premature death or disability occurs when the application has to be replaced without their being a major change in the business requirements that would have forced its demise.

Causes of Premature Death or Disability in Applications

From our observations, the most common causes of premature death and disability in business applications are:

- Obsolescence of the hardware or software platform upon which the application was built
- Obsolescence of the programming language or tools used to build the application rendering additional maintenance impossible or at least uneconomical
- Inability to keep up with requested functional enhancements
- Inability to keep up with technological advancements (faster processors, multimedia, handhelds etc)

The first two are caused and/or exacerbated by design styles that make the application overly dependent on the technology upon which it was built. It is not currently possible to build an application that is not dependent on the technology it was written in. In other words, all applications are written in some programming language and are written to run on some family of operating systems and/or hardware. However, just because it is not possible to write an application that is completely independent of the environment it runs in, that does not mean that there isn t a world of difference in the degree of dependence.

The typical fat client application of the 90 s consisted of millions of lines of procedural code (eg C, C++, Visual Basic), and contained hundreds of thousands of calls to the windowing and operating system API. The structure of the code was contorted around platform specifics such as dealing with the event loop, or specific styles of thread control. It was possible, and occasionally occurred, that the application was built around abstractions of the key capabilities that the platform was being used for (rendering, or saving data to disk) and these abstractions were coded once to the underlying platform and reused, either through object oriented design or very disciplined modular design. These applications were far less dependent on the underlying platform.

The second two causes are different variants of bad design. Poor separation of concerns typically meant that seemingly small changes in functional requirements which might have been handled in stride, create a string of side effects and long and complex regression testing to make sure the changes haven t set off a series of side effects. The final cause is often a corollary to the first two: if the program was too dependent on one set of technology, it is not unusual for it to be difficult to incorporate another technology.

Typical Configurations

The case studies below will describe in a bit more detail some of the representative situations we find when we look at existing installations. Typically mid sized enterprises ($100 million to $2 billion) have these characteristics:

- Some set of core transaction systems, which may have been custom written or purchased, but in either case their size and complexity have stalled out either replacement or major rehab. Typically there was an effort through most of the 90 s to standardize on these core systems, which was typically only partially

successful. The core system will be either mainframe or server based. The interface is either character based (still) or fat client.

- A series of smaller systems, in a variety of newer technologies (eg Dephi, PowerBuilder, Access) that extract some data from the core system and supplant it with additional data supplied by the users.
- A series of new initiatives in web-based technologies (Java, JavaScript, JSP, ASP) Very often these new initiatives are interfacing indirectly to the core transactional systems or satellite systems made from extracts of the core system.

This is creating a situation not unlike the current building (especially commercial and multi family residential). A hundred years ago buildings were typically over engineered. Heavy building materials such as brick, stone and timbers were used for all construction. Fifty years ago the move to engineering for useful life began, and builders started learning how to build with cheaper and lighter materials. In the last few dozen years this has advanced substantially. Big box retailers and fast food restaurants will typically tear down an existing structure to put up another with a short half life that more closely meets their needs. As a result, the newer a building is the shorter its half life.

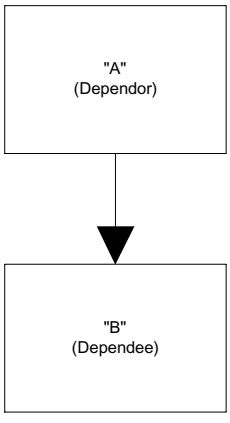
And so it is with business applications. Platform vendors have determined that it is in their best interest to accelerate the turnover of their platforms. As a result, in general, the newer a system the shorter its half life. Internet time strikes again. On top of that, most of the newer systems are deeply dependent on the older systems they are extending or partially replacing.

Software Rehabilitation

Ok, it s not quite that bleak. There are some ways out of this mess. But they require the same kinds of skills that it takes to renovate older buildings: you have to understand the static and dynamic underpinnings of the structural integrity of the structure at hand, and you have to have a deliberate, but incremental way to upgrade the structure to something that will have a longer life, without starting over completely. To make it worse, you typically have to do this while the occupants are still in the building.

Dependency Analysis v. Composition

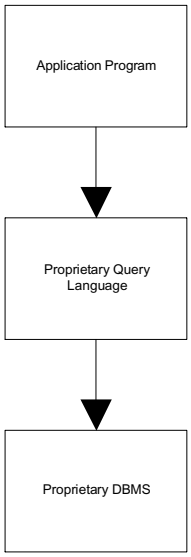
The main weapon we use is dependency analysis. Our diagramming convention starts fairly simply: we put the dependent object on top, with an arrow pointing to the object upon which it depends. In this case, A depends on B. If A were a program written using a special software library B, we d say the program depends on the library. The library doesn t depend on the program. The dependency is directed.



The acid test for dependency is: if B changes, must A change? Must it change to take advantage of the capabilities of the new B, or in the stronger case, must it change whether it wants the new B s features or not? We ll get into the special cases of it depends on the kind of change to B later, but for now take this to be the litmus test.

One of the most obvious kinds of dependency is dependency by composition. If program A contains code from library B, then the dependency comes from this composition.

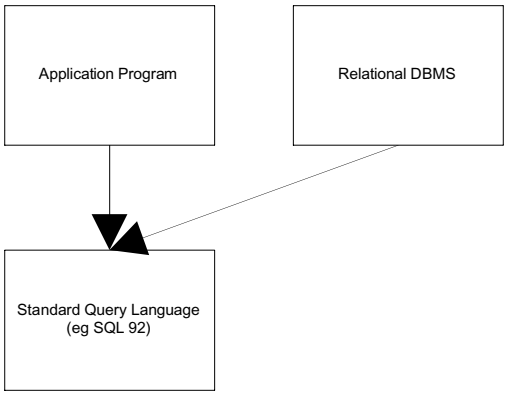
However, the graph gets more interesting, as we get beyond just the compositional dependency. Let s take the case of a proprietary database with a proprietary query language.



Very often the proprietary database has a language that you write your applications in. This makes the application very dependent on the query language. The query language in turn is dependent on the database, as no other database will work with this language. A change to the database (the most radical change being swapping it out, but less radical includes upgrading to a new version) may require changes to the query language.

At this point, we should note, that our assessment of dependency is a pragmatic one. People can split hairs about whether and what kind of changes could be made without affecting the dependee. As we ll get to a bit later, out motivation revolves around practical concerns including especially: what is the cost of overcoming this dependency?

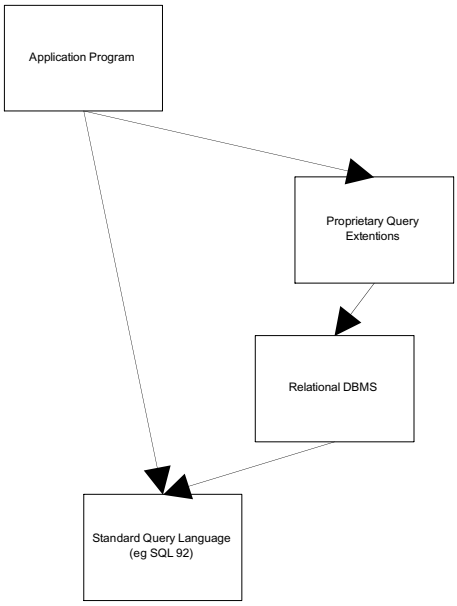
However, when we deal with standards the topology of the dependency changes. Let s continue with the example of a database, but this time use an industry standard query language.



From a practical standpoint the DBMS is now dependent on the query language standard. Luckily standards like this change slowly, but when they do, both the DBMS and the application are affected. Strictly speaking, changing out the DBMS, even if the application has been 100% standards compliant (which is almost never observed in nature) is still not a zero cost operation, but it is so much lower in cost than rewriting an app that we can consider it so. (If we wanted to be more precise here we would show the data being dependent on the DBMS, meaning that a change would require a conversion, and the application is dependent on the query language binding, meaning the source code wouldn t change but the program would need ot be recompiled and/or relinked. If we insert ODBC in as the query language we get the dependency only on the driver)

But life is never that simple. Typically, people buy a DBMS because it is relational, and believe that that is sufficient to keep their future switching costs low. They then proceed to turn their developers loose on the programming tasks. The programmers discover that the vendor of the relational database have supplied a number of proprietary extensions to the query language which make development easier for the developer. Oracle for instance has a bill of material processor built in to their query language using their own Connect By clause. The vendors generally will not distinguish the proprietary extension from the standard SQL in any of their reference material, and the developers,

having little compassion for future switching costs, and generally an aggressive budget to hit, are not particularly concerned. The only exception I know of was Digital Equipments Relational Data Base that was available on the VAX/VMS platform. As an aide to the developers they produced their manuals in two colors, black for everything that was implemented consistently with the SQL standard of the time, and red for their proprietary extensions. Despite the fact that this was a database that was much closer to the standard than most that were available at the time, the manuals looked like the New Testament (there was as much red ink as black). It s not clear whether their intent was noble (to help those who were sincerely trying to keep their switching costs in check) or bravado (to show how much better their product was than the standard).



The open source product MySQL has an interesting approach to their extensions: they code them in line as comments, so that other DBMS will ignore them, but MySQL interprets the stuff in the comments. The dependency problem is not completely solved by this approach as it is possible for the application programmer to code to the extensions, and have a dependency on these.

In any case, the dependency graph is now more complicated. The application program depends on the standard and any extensions used. Often the dual dependencies show up in the same SQL statement.

The degree of dependency is going to depend on how many programs use the extensions, how often they use them and to what degree they have bent the structure of the program around the extensions.

Platform Life Cycles

As we will describe in the case studies, we have been able to construct meaningful dependency graphs in fairly complex environments. However this is only part of the issue.

The next part is: how stable is the thing you are depending on? The more stable the thing you are dependent on, the less pressing the dependency issue is. The other consideration, even with a stable dependee is what are the annual costs to remain dependent (ie are their licensing fees, do you need a certain number of experts on staff to keep the platform or tool running etc)

We typically color code the dependees, based on our assessment of their likely half life. There are several things that contribute to a reduced half life for a platform, language or middleware, and they are mostly, but not entirely predictable:

- Platforms based on recurring revenue streams tend to be more stable, as the vendor prefers to minimize the disruption and possibility of the customer changing platforms. Vendors will often create a series of minor upgrades, which

can be installed at low cost, due to their backward compatibility. Many mainframe products have had non disruptive upgrades for decades.

- Platforms based on one time revenue tend to be less stable, vendors will continue to enhance the platform, attempting to entice buyers to repurchase. As an added incentive vendors will often discontinue support for older versions after a certain time period. Microcomputer and internet platforms typically have about a three year support window, after which there is limited support for a year or two and then none.
- Some vendors go out of business, or get acquired. Some products never take off and the supporting vendor decides it is not worth their while to continue to support their installed base. These provide some of the more catastrophic requirements for change.

Switching Costs

The final annotation that we use on our graphs is to graphically show the cost to switch out the dependee, if it becomes necessary. While we show the cost in thickness and/or color of the dependency arrows, the cost is almost always incurred in the dependor . The cost will be greater the bigger and more complex the dependor is. So, for instance if you have to recode a 3,000,000 line Mumps program, if the last Mumps vendor goes out of business, you are going to incur a higher switching cost than rewriting a 3,000 line Fortran program, for the same reason.

Integration and Coupling

Our final comment before we get to the case studies concerns integration and coupling, seen in this light. For the last decade, vendors (especially ERP and DBMS vendors, but also consultants) have sold customers on the reduced cost of interfacing if systems are more integrated. Generally integration is good, if by that we mean, when a business event occurs in system 1, it is automatically reflected in system 2 soon enough that there are no adverse business effects. This time lag is obviously different from system to system. One way to implement integration is to have a single unified database. So system 1 updates a table that is immediately available to system 2, because it is reading the same table. Unfortunately, the dark side of integration is the dependencies it introduces. In this scenario both systems are dependent on their shared data model (if either change it it affects both) and system 2 is dependent on system 1, any changes to the way it updates will affect it. The two systems in this case are tightly coupled. As we ll explore in the case studies there are many ways to integrate systems without tightly coupling them.

CASE STUDIES

World Minerals

World Minerals Pipeline system was implemented in the late 80s in a traditional COBOL environment running on a relational database. The system successfully automated many of World Minerals core business functions, and was used successfully for over 10 years.

Platform Life Cycles

Pipeline was implemented under VMS running on VAX/Alpha hardware. The database was RDB. After 10 years all of these technologies were showing their age, and were reaching the retirement stage in their life cycles. More critically, an IT risk analysis had identified a piece of middleware upon which Pipeline was built and which was no longer supported. This represented a compelling reason to replace Pipeline or to convert it to contemporary technology.

Dependency Analysis

A dependency analysis of the Pipeline implementation revealed that most of the application s logic was expressed in the form of database triggers, and that those triggers were written in standard SQL, taking little advantage of RDB s extensions. This meant that it was feasible to convert Pipeline relatively simply.

Switching Costs

World Minerals had to decide between converting Pipeline and replacing it with an off the shelf system. The dependency analysis indicated that a conversion might be less expensive than statistics for similar sized projects might suggest; at the same time a purchased ERP system would require extensive customization to accommodate some of World Minerals more unusual requirements. The decision was made to proceed with the conversion, which was ultimately completed at a lower cost than the estimated alternative.

Integration and Coupling

Rewriting the Pipeline system got the company out from under the threat of having their primary application fail on them, but it did not significantly change the probability that some latent software dependency would put them in the same position in the next two to five years. Their system architecture as a whole was aging and they needed a migration plan.

World Minerals application architecture was typical of many enterprises today. The Pipeline application represented a complex ERP type system that ran their core business operations. Over time this had been tightly integrated to a financial package (Platinum) and a whole raft of extract, process and report applications that had grown up to fill holes in the functionality of Pipeline.

This typical situation had caused some typical problems:

- Pipeline was too large and complex to change with any confidence. (The middle wares migration did not actually change any functionality).
- Most other satellite applications were tied so tightly into Pipeline that they were almost equally difficult to replace.

- The network of interfaces that connected the applications was fragile and introduced many data inconsistencies. Much energy, and additional coding effort, was expended reconciling one application to another.

The company had neither the inclination nor the experience for a high-risk rewrite of their entire application portfolio.

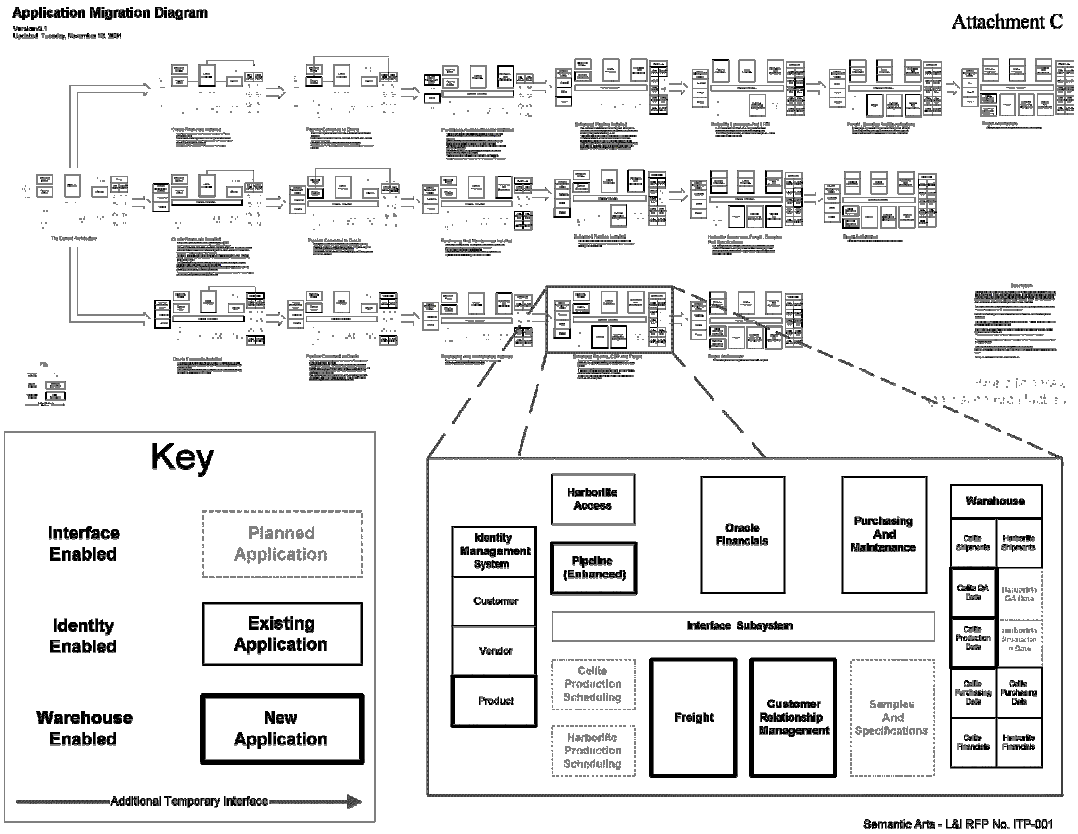
Most of the problems were related to their integration architecture. The solution needed to allow the applications to be loosely coupled yet tightly integrated. A message bus based architecture was chosen to achieve this.

In a message bus architecture applications talk to one another via messages (typically XML based messages) communicating via a message server. Each application is equipped with adaptors , code whose job it is to processes either incoming or outgoing messages. If implemented correctly any given application is dependant only on the design of the message rather than the application. This level of indirection has some significant advantages:

- You can change, or if need be replace, an individual application without impacting any other.
- It opens up the possibility of partitioning large applications piece by piece into a series of smaller applications, communicating across the bus.
- Common messages provide the mechanism for managing controlled data redundancy between applications.

At a high level the migration plan for World Minerals was to replace each of Pipelines subordinate applications one at a time, in each case interfacing them to Pipeline via the message bus. Once this had been achieved and Pipeline had been effectively wrapped , then it would be gradually partitioned into smaller more flexible components each of which could be replaced either by a newly developed solution, a package, or even a web service.

In the short term this gave the company a way of designing themselves out of their problems over time. In the long term it will greatly reduce their dependency on any single application or technology, as these would tend to be isolated from one another behind their adaptors. In the future as an individual application technologies become redundant the damage will be much more effectively contained.

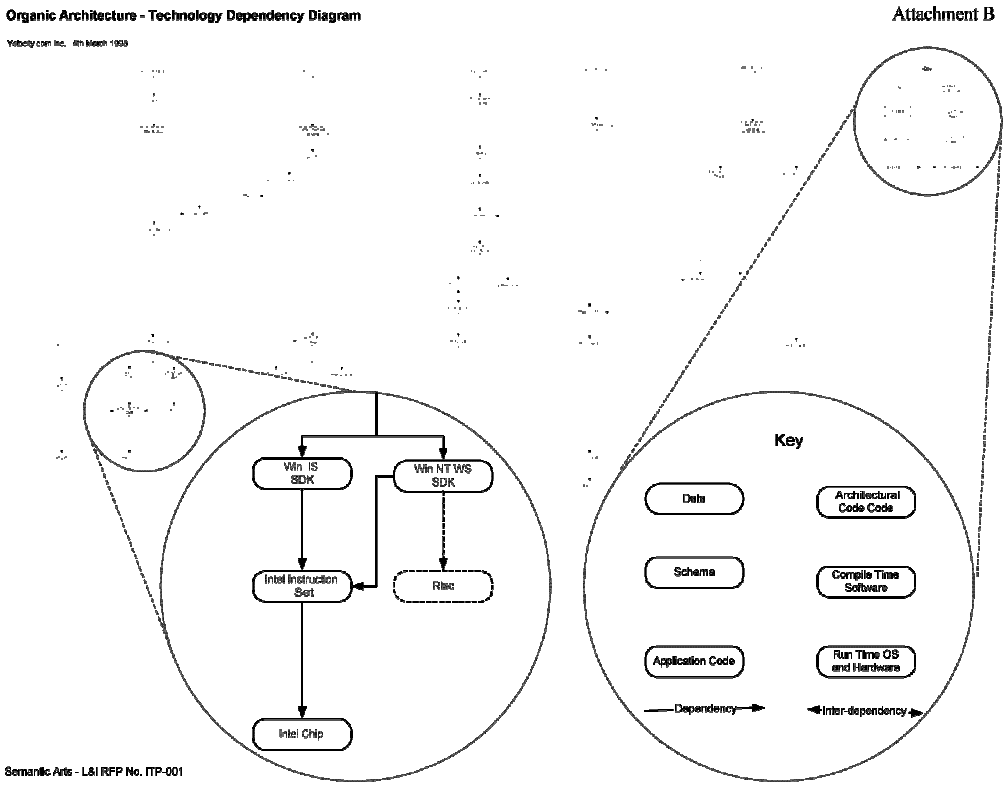


Velocity — Riding Lightly

The Organic Clinic project undertaken by Velocity.com from 1994-2000 is an interesting case study in minimizing dependencies. This project was implemented in C++ in a Windows/Unix environment. All operating system calls were encapsulated in objects performing the same functions. These wrapper objects tended to be very thin ; there was very little difference between their interface and that of the service they contained, although care was taken to avoid wrapping those parts of the interface that were especially idiosyncratic. Similarly third party add-on products such as imaging libraries and a database were also isolated. This approach has two benefits. Firstly it makes it possible to switch out third party services over time as necessary, which makes the code more portable and thus potentially longer lived. Secondly it isolates the consequences of change in the used services to the wrapping code. This benefit is especially useful for a long duration development effort, that are at risk of never reaching completion due to the rapid churn of the underlying components.

The approach also has a couple of negative aspects. Firstly there is an increase in the amount of code to be written. Wrappers do not appear for free, and even if they are shallow they still have to be designed. Secondly there is the least common denominator problem that comes about by creating abstractions that intentionally hide those features of the system being used that are potentially the most capable. In the case of Velocity this trade off created a database interface which performed substantially less efficiently than the underlying mechanism would have done if used directly.

Early in the project the validity of this approach was confirmed when the Unix part of the system was ported with minimal effort to Windows NT. This experience suggests that the wrapping approach will extend the life of a system by making components upon which the system is dependent somewhat cheaper to exchange. However, it is na ve to suggest that this is a complete solution. Wrapping is a relatively shallow approach. There is behavior that appears from the interaction of interfaces upon which the system will depend, even if the interfaces themselves are wrapped. (The Java platform revealed this problem early on, when different platforms would generate the same events but in different sequences.) A logical extension of the Velocity policy was to avoid using operating system facilities altogether if at all possible (see diagram). If you don t use COM, after all, the advent of COM+ is not much of a problem. The lessons to be drawn from the Velocity experience are quite clear. Wrapping can help isolate third party services thus reducing the impact of churn and reducing potential risk from services that become unsupported. Wrapping is not a complete solution, however, and it comes at a development cost and a cost in features.



Labor and Industries — Migration v. Big Bang

Washington State Department Of Labor And Industries is mandated with providing workers compensation insurance to all of the employees in the state of Washington. Like many big insurers, over time they had developed a large complex mainframe based application to collect employer payments and administer and pay claims. Their principle system LINIIS was developed on a highly centralized Adabase database using Natural and CICS for the online portion and Cobol and JCL for the batch processing.

In many ways L&I was a poster child for the Information Resource Management architectural approach popular in the eighties. The application though fifteen years old is still a close functional fit to the department s mission and still performed the core business functions effectively. Over time however as more and more functionality had been built around the central databases the application had become increasingly complex. In particular they were facing a number of pressures:

- The legislatures E-government initiatives were introducing a number of data access, security and technology management challenges.
- The mainframe technologies though not immediate danger of obsolescence were getting long in the tooth and the lead time for replacing them was lengthy.
- The application was increasingly hard to change.

Dependency Analysis

The first step was to perform a detailed dependency analysis of the Industrial Insurance applications to see what technologies they were dependent on. In concert with a comprehensive market analysis of those technologies each dependency was traced back to an application and a picture began to emerge of where the department was most at risk.

Platform Life Cycles

One interesting result of this was that though they were highly dependent on the aging mainframe technologies this was not where they were having their obsolescence problems. The dependency tree here was relatively simple and the technologies were very stable. By comparison some of the satellite applications, such as imaging, were experiencing an extraordinary turnover in their technology base as they tried to keep up with a complex and evolving dependency tree. Clearly just replacing the legacy systems was not the going to solve all your technology dependency problems. L&I now had a pretty good idea of what technologies they were going to replace, which contain (keep for now but do no further development in) or embrace.

Switching Costs

Rather than attempt to replace the legacy applications immediately L&I chose to wrap them using a message bus approach. Access to LINIIS was through a few canonical messages only. This enabled them to enhance their applications without further investing in the legacy technologies. When the time came to finally swap out their legacy apps they could do so with far less disruption to other applications and interfaces. It also left them the option of partitioning the legacy application further ,within the wrapper, and replacing it a bit at a time.

Integration And Coupling

For government organizations in particular this gradualist approach to application architecture has the advantage of not requiring a huge one time capital investment, which is hard to get funded. What purely architectural expenditure there is, setting up the

message bus itself for instance, is small enough that it can be bundled in with another more tangible project that is more easily approved.

Using the message bus architecture also isolated the new development projects for one another, which allowed more freedom to experiment in new technologies. For an organization that is trying to move away form a mainframe culture this is a distinct advantage.

Life & Death at the Leading Edge

Though smaller in scale, the story of a technology-leading application at Visible Productions has similar lessons to teach. The rapid pace of technological change was a primary factor in the premature disability of their novel system. Their software system for creating highly accurate & realistic biomedical models began life in a research environment. Its first life was as an integrated UNIX/C system on SGI machines, the premier choice for 3D visualizations in the early 1990 s. This environment, however, became expensive to use as business grew and more workstations were needed, and it wedded the organization to expensive SGI workstations at a time when the power of individual PCs was rapidly advancing.

To use this newer & more cost-effective technology, in the mid-to-late 90 s the system was mainly reworked. The image analysis / data acquisition piece was taken to the Mac platform to be rewritten and enhanced. The model manipulation piece was ported to the now powerful Windows environment. Fortunately, since SGI was a major player, the GL interface they proposed evolved into the OpenGL standard which had taken hold in the PC world, thus easing the port.

Thus the system stood, in its second lifetime, barely 5 years old, in the late 90 s. Now, however, the system is distributed over Mac, NT, and SGI platforms, which makes it cumbersome. The work on the Macs used a visual programming environment which is now no longer supported and only one developer knew that development environment, thus all enhancements waited on him. To provide a unified platform base and feature improvements, the 3rd life of the system was planned, a redevelopment of the system in Java. The essential benefit from Java is that the language allows the system to be compiled into a virtual system, the Java bytecode which is now implemented on a wide array of platforms. This bytecode, essentially, is a large-scale *wrapper*: it wraps the actual hardware in a virtual machine.

The system, in its second lifetime, provided an interesting perspective on standard interfaces. The system was tied together by exchanging data through files, in two formats. One format used was the OBJ format, an inefficient, ASCII format defined and used by the Wavefront 3D rendering software. Another format used was a private, custom designed, efficient binary format for the large files. When the original developer left, the knowledge of this private format left with him, making changes and interfaces to this system part totally cost prohibitive — any change would require reverse-engineering that interface to understand its workings. The bulkier OBJ format, luckily though, became a de-facto standard format, since it was defined and used by a major vendor. Its common use has continued to this day, which has allowed the 3D rendering

part of the system to smoothly evolve as new rendering software is swapped in: first Wavefront, then Electric Image, and now StudioMax.

The primary lessons here are twofold. First is the benefit of non-optimal, though standard, interfaces, which allow for smoother transitions across applications and platforms. Second is the notion of a virtual machine as a wrapper, which will hopefully be carried into the future significantly with the Java implementations. It is this virtual machine concept which IBM capitalized on so well with its 360/370 line of computers, enabling some business applications a quite-healthy lifespan of decades.