A Fast Track to Software Modernization

Introduction

Legacy software applications are very valuable assets. Most CIO's and CEO's know this, in spite of the fact that today's balance sheets rarely show their real value. Typically, over the years large investments in intellectual capital have been imbedded in the legacy systems, but more often than not, those investments have been booked as expenses (the salaries of all those involved in development, maintenance and enhancement of legacy applications).

There are vast amounts of legacy systems throughout the world; there are also many definitions of what a legacy system is. It is in fact of little value to lucubrate on definitions of legacy systems; suffice it to say that legacy systems are valuable (they work). In this document, the term "legacy" refers to any existing software application that works, disregarding of its programming language age.

Legacy applications, like other intangible assets, are harder to imitate by competitors, and thus differentiators and a source of competitive advantage[1].

This White Paper describes a Computer-Assisted Source Code Transformation methodology and discusses how it can accelerate financial pay back as well as many intangible benefits of Legacy Modernization. The methodology can also be used in conjunction with code analysis and understanding approaches, once the legacy has been migrated.

The main body of the paper is a step-by-step description of a methodology (based on computer-assisted source code transformation technology and principles) utilized to transform legacy applications to modern platforms in a safe, fast and cost effective manner. It briefly describes the high-level evaluation criteria that lead to the decision to modernize a large legacy system.

During the first 40 years of software engineering, deciding on a target platform and trying to have a single platform in larges organizations has been hard, conflictive and elusive. It is only now, after decades of building and rebuilding systems, that a single unified target platform is both feasible and a practical objective to aim for. It is argued that the large integration problems that most large organizations face today essentially disappear once a unified modern software platform is achieved.

The existence of software tools and methodologies to support the software transformation process enables an increase in flexibility, efficiency, control and cost effectiveness of IT evolution. These technologies improve return on investment on software assets, now and in the future.

Legacy Modernization needs

The decision to modernize a legacy system can be related to many circumstances, cost not being the least of it. Retaining the value of the legacy application is a common objective of most modernization efforts, for even if the value is not accurately known, it is certainly worth retaining. The cost of discarding a valuable software asset goes far beyond its value as the organizational disruption and loss of competitive edge can dwarf the additional cost of the new replacement asset.

Apart from cost, organizations are often moved to modernize legacy systems by several factors which can be summed up as lack of strategic flexibility. Some of the factors that can severely hinder the organization are:

- vendor dependence,
- lack of trained personnel in the labor market,
- long time to market,
- legal/regulatory compliance requirements, and
- poor integration capability.

While none of these factors is in and of itself crippling, they all lessen an organization's agility, increase operating costs and tend to undermine its competitive position.

The need to modernize legacy systems is not new; it has been a thorny issue from the beginning. Obviously, as time passes the factors listed above are exacerbated prompting management to act. The sharp increase in legacy modernization in recent years is, however, due to two separate and relatively new developments.

First, there is now a clear path as to what technologies to aim for. As recent as 5 years ago, this was not clear, but now it is agreed that n-tier architectures based on either .NET or J2EE software are the desired target platforms. Secondly, also in the last few years, there has been a substantial improvement of the quality and completeness of software transformation and understanding tools.

So, the question top management faces when addressing the need to modernize legacy systems is: how to retain the value of these software assets whilst freeing the organization from some or all the factors listed above?

Due to the very favorable cost, time and risk associated, source code transformation solution is, in many cases, the preferred approach. This approach allows the intellectual capital invested in the legacy system over the years to be fully recovered and combined with all the benefits associated to the new technology platforms. This is done in a very efficient and safe manner, especially when compared to highly labor-intensive alternatives.

We go as far as to say that transformation should happen very early in the process, even before the legacy system, and all its architectural quirks, are fully understood. Experience shows that understanding and re-architecting the legacy application is best performed under the new technology platforms, as not only the power of the new IDEs becomes available, but also a plethora of other new software technologies can aide in this process, making the control of the process much more effective. With the old programming languages it is difficult or unnatural to represent some of the modern software engineering concepts, such as components, tiers, etc. Furthermore, correctness of results can be directly corroborated by applying the Functional Equivalence principle to the original application.

Finally, those difficult synchronization problems associated with the need to freeze code evolution during the modernization process are greatly reduced as explained below.

If the approach presented in this paper has a flaw, it must be the possibility that, having achieved a very high return on investment after the source code transformation, and due to day-to-day pressures, some of the re-architectural work may not be conveniently concluded before the project is delivered.

When and How to modernize

A two step approach is presented. First, concepts from Kaplan & Norton are used to aid the decision to modernize a legacy application. Once the decision to modernize has been taken, the methodology follows with concepts from Declan Good's research[2] to decide how best to modernize.

Kaplan & Norton's Strategic Readiness concept evaluates how readily an application can support the crucial business processes required to achieve the organization's strategic objectives. It does this methodically; by first identifying which are the crucial processes and then evaluating how well the existing application(s) and its infrastructure support them.

The degree of readiness, according to Kaplan & Norton, is measured in a subjective way but relative to all applications being thus evaluated. Applications required to support the crucial business processes are evaluated on how well can they support the processes now, and when will they be available to give the required support. Some, albeit few, applications will be deemed "ready", these are probably already available in the form and with the required infrastructure, others will be deemed "almost ready" depending on how much time and effort is required to make them available, whilst other still will be deemed "not ready". Legacy applications that fall into these last two categories must clearly be modernized.

Legacy applications that do not support crucial business processes (for example back office systems) may still require modernization, but this typically is a financial decision rather than a strategic technical one.

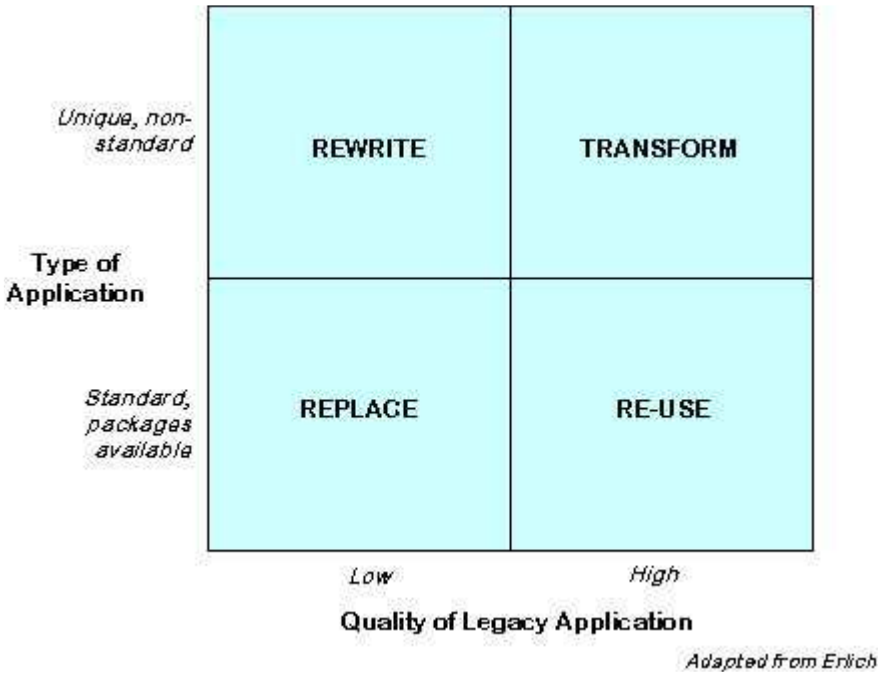
In general once the decision to modernize a legacy application has been taken, there are four broad alternatives (routes) that can be followed: re-write the application, replace it with a package, wrap it with modern technology to make it look better, or transform it using a combination of automated and/or manual tools.

Declan Good's describes of this decision process as an evaluation of the quality of the application and of the availability of replacement packages. The term "quality" in this context refers to how well the application does what it does, and whether the functionality it lacks is due to it's architecture or its infrastructure. In general the "quality" of an application is considered low when the application fails often, rework and workarounds are often required, and/or it is deemed that the architecture is a severe impediment to developing the new required functionality, even if the infrastructure (software and hardware platforms) were different.

The availability of a replacement package refers to the uniqueness of the application and/or the organization. In general if the application under evaluation supports, or will support, crucial business processes that are critical to achieve strategic objectives, then using a package implies that the process in question is not unique and therefore not a source of competitive advantage. At best a

package can, under these circumstances, deliver a short term advantage because packages can be easily imitated.

The following diagram[3] shows the combinations of factors that might lead to the four options:



Good's approach allows a first hint of which alternatives are best suited for a given application. The decision is hardly ever clear cut as suitableness of a package and/or the quality of the existing applications is a rather subjective measurement and especially because such decision cannot be taken without considering time, cost and risk.

All alternatives should have their time and cost measured in the same, most comprehensive, way. Risk, on the other hand, must be evaluated relative to the other alternatives. Rather than an absolute measurement of risk, a ranking indicating the risk of each alternative in relation to the others must be arrived at.

Risk, in this context, has to do with organizational disruption, past experience in meeting deadlines, available knowledge (related to new technology or packaged application), future dependence on a supplier, and other factors that can influence the final outcome of the project.

Once cost, time and risk are evaluated for all alternatives, the relative standing of each alternative should be graphed in order for top management to make the final decision. The following example presents a hypothetical comparison between Replace, Rewrite and Transform alternatives for given application. The Re-use alternative refers to wrapping the legacy application with new technology to make it look better, but this alternative is only skin deep whilst making the overall application more complex. Almost by definition, the re-use alternative is only a short term measure.

Cost	Time	Risk
Transform ($300K)	Transform (9 months)	Replace (low)
Replace ($500K)	Replace (12 months)	Transform (medium)
Rewrite ($1M)	Rewrite (18 months)	Rewrite (high)

Obviously, only when the same alternative offers minimum cost, time and risk, is the decision simple. In the example above, management must decide whether to accept some risk in order to minimize cost and time, or pay more and wait longer in order to minimize risk.

Methodologies for replacement by a package, entire rewrite, or re-use by wrapping exists. In here, the following presentation is specifically focused on the required methodology for transforming applications.

The methodology required for Application Transformation begins when the scope of the system to be modernized has been defined together with the target platform and architecture. Detailed planning of a legacy modernization project must include: project assessment, transformation strategy, test suite definition, final architecture requirements, testing strategy, training strategy and, implementation strategy.

Assessment and Preparation

Large systems usually require a modular transformation, implementation and re-architecturing strategies. It is not always straightforward to break up a legacy system into modules that make sense to transform, implement or re-architect on their own. Interdependencies within and between modules can be, and usually are, quite complex.

Computer-assisted source code transformation tools typically do a first run looking for (and tagging) excessively redundant code (very common when reuse was achieved through cut and paste),

graphing out interdependencies within and between all the code to be migrated, performing basic code re-factoring, and assessing the manual work that would be required once the transformation tool is used.

Excessively redundant code can be encapsulated and simplified during the next phase. The interdependencies graph is a valuable tool if modules need to be carved out for implementation purposes (See Figure 1). Figure 1 not only shows dependencies but also suggests logical application subsystems that could be considered module candidates.

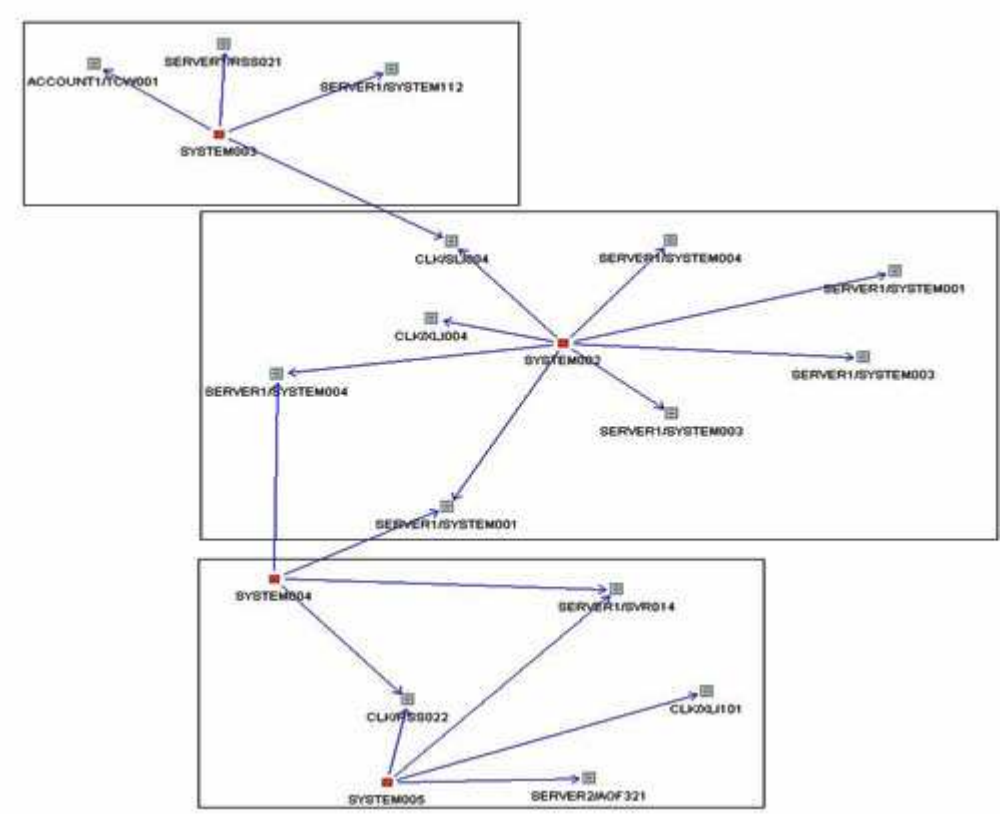


Figure 1. Example of a simple dependency graph. Rectangles indicate the partition detected in the graph.

Code redundancy can be found not only at the application logic level, but also at the data structure level, when repeated structures are used in registers and working areas, such as:

01 W0-DATE

05 W0-MONTH

05 FILLER

05 W0-DAY

05 FILLER
05 W0-YEAR

The first run over the source code can also show syntactic constructs which if changed before using the transformation tool would enhance the automation and increase the quality of the generated code (basic re-factoring). In these cases, those constructs are manually or automatically changed before migration.

Functional Equivalence

In order to be successful, software modernization projects need to keep a clear separation of concerns in subsequent process phases. Experience has demonstrated that the transformation of the application infrastructure, i.e. language, operating system, etc., needs to be isolated from other modernization duties, such as new functionalities, re-architecture or application deployment strategies.

The principle of Functional Equivalence is highly useful tenet of computer-assisted software transformation. Functional Equivalence guarantees that each action taken on the application logic doesn't change its functionality and therefore assures a correct recovery of the program logic. Utilizing automated tools without this principle will most certainly imply long and painful debugging processes.

Functional Equivalence must be attained before other actions that can change the application functionality, such as re-factoring, re-architecture, or further application evolution tasks, are taken on the code.

To assure Functional Equivalence, it is necessary to put together comprehensive unit and system test suites, for the whole system to be migrated, as well as for individual migration modules (if they are required). Test suites must be exercised by automatic tools that compare expected with actual results.

It is the obsessiveness with which Functional Equivalence is maintained that makes the risk of this project so much less than other alternatives (like rewriting or replacing the system). Any automated tools used for the transformation of the code must guarantee the preservation of Functional Equivalence; each low or high level transformation performed must always guarantee the

preservation of the semantic contents. In this way, it is possible to perform a bottom-up transformation of a large system with minimum risk.

The process is not, however, without risk. The future modernization phases, such as code understanding, maintainability, and evolution, demand a high quality of the generated code and therefore impose important constraints on how far the automated transformation phase should go. Furthermore, when transforming legacy languages and programming paradigms to modern ones, there are, invariably some constructs, which do not have an equivalent in the new platform. Powerful pattern matching technologies allow searching and mappings of large constructs (as opposed to term by term mapping/translation), this greatly increases the level of automation and the quality of the thus generated code, but even then it hardly ever reaches 100% functional equivalence through fully automated procedures.

Source code transformation

Powerful transformation tools can transform source code from existing environments (like Informix4GL, or Visual Basic 6) to modern platforms (like .NET) to about 90% effectiveness. The code that is not automatically transformed is documented and marked for revision by trained engineers. This is when software engineering discipline must be exercised, test suites executed, transformation rules changed and manual changes made in an iterative loop until 100% Functional Equivalence is reached.

The tools available nowadays are not only capable of matching and transforming large chunks of code to their equivalent in the destination platform, but are also able to find and eliminate repeated/redundant code and suggest its encapsulation/reuse for further consideration of the trained engineers. In this way, the source code transformation can be seen as not only transforming the source language, but also the programming paradigm (like from procedural to OO).

Automatic recording of manual changes and ways of extending and adapting the transformation tools with specific-purpose rules supplement modern transformation tools. The use of these companion techniques brings very significant reductions on the time required to freeze code evolution during migration.

A tool that is capable of transforming 90% of any source code for a give pair of languages requires 10 times more time and effort to perfect than a tool capable of transforming 90% of a specific source

code. For this reason, it is found that often the tools, and their rule base, need a certain amount of tuning to reach the required level of automation for a given application. There are very few tools in the market today that can claim a high level of automation for any application.

With a tool that is 90% efficient, experience shows that trained engineers can easily work through 28000 lines of the code per month (correcting the issues that were not automatically transformed). This means that, the automation of the transformation process increases the productivity of the engineers (compared to manual development) by at least 8 times, using productivity statistics for modern programming languages from the Software Research Institute, and Gardner Group. When re-architecturing and functionality enhancements are added to the equation, the engineers' productivity will be greatly reduced, to about 3 times more productive than manual development.

Migration

Once Functional Equivalence is reached, the system can be migrated to the new hardware and/or software platform, and the platform specific optimizations efforts be performed. Since the users require no retraining (due to Functional Equivalence) the system can usually go into production very quickly, and, although modernization is usually by no means complete, a large portion of the financial benefits (usually related to the hardware and software platform) can now be accrued.

The time required for a transformed system to go into production varies depending on the size of the system, the number and geographical dispersion of the users and on logistical issues, which can be associated to the roll out of the new platform. But in general, for a large system, this time is within a couple of years of the project commencing. For medium and small systems the time can be as little as a few months.

If, for a number of reasons, migration is to be done one module at a time, it is then necessary to build "scaffolding" to integrate the new module with the legacy system. This is readily achieved by manually or automatically generating the necessary middleware code (this is helped by the interdependencies graph produced in the very early stage of transformation, or a reviewed version of the new code). After each module is implemented, the whole middleware code thus generated is discarded and a new one generated. Obviously, if the number of modules is more than two or three, the generation of the middleware code could be done automatically.

Migration of a medium size legacy system (5 or 6 million lines of code) typically takes around 6 to 9 months. After which the organization is already enjoying the benefits of the new platform as well as the financial savings from discarding the old one.

Enhancement

The modernization process however, is usually not completed at this stage. The resulting software running in the new platform might still require reengineering, componentization, refactoring and possibly functionality replacement and/or enhancement.

Powerful source code understanding tools can now be used on the transformed code to gain insight on how best to re-architect the code. It is one of our main contentions that this is best done after, and not before, the code is transformed because, not only are the tools to work on the new platforms much more powerful, but also the new generation of engineers is much more conversed with them.

A corollary to platform migration and unification (all the organization's systems can end up in the same platform) is that the application integration problem, disappear. When all organization's systems run on the same modern platform, Service Oriented Architecture is readily implemented and the integration between modules and/or systems becomes simpler and transparent.

[1] Kaplan & Norton, "Measuring the Strategic Readiness of Intangible Assets", HBR, February, 2004
[2] Declan Good, "Legacy Transformation", CIT 2002
[3] Diagram adapted from a presentation by Len Erlikh of Relativity Technologies, Inc