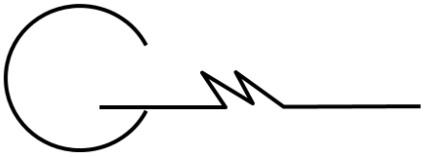
Comparing Migration Methodologies



Great Migrations LLC
December, 2007

Introduction

Purpose

The purpose of this document is to introduce the concept of a software migration project and compare two migration methodologies.

- **Manual Rewrite Migration,** which leverages hand coding to rebuild the legacy codebase from scratch. Optionally, this approach may also leverage a low-yield translation tool to convert the legacy application to a new codebase that still requires significant manual work to take it to the desired state.

- **Tool-Assisted Migration,** which balances customization of translation tools with manual work to convert the legacy application to an upgraded codebase. The Great Migrations Methodology will be discussed specifically.

Audience

This document will be helpful to individuals who are planning a software migration and want to understand how or if the Great Migrations Methodology can help them in their efforts.

Why Migrate?

Change Happens

The details of every software system are defined in terms of two foundation technologies:

- The programming languages used to describe the system, and
- The various runtime libraries used to access external services

The programming language dictates how developers can describe data structures, interfaces, and algorithms. The libraries provide an extensive array of advanced services to the program such as data access, communications, and graphical user interface. The language and libraries are called the platform. The platform sets the rules and makes the system possible.

Over the course of a system's lifespan the platform can and will change: languages and libraries inevitably evolve and are replaced by next generation technologies. Usually, the changes are gradual with an appropriate measure of backward compatibility, so we can adapt through standard maintenance activities. Sometimes, however, the changes are more radical and disruptive, so a more focused effort, called a migration project, is called for.

The Business Case for a Migration

The business case for a migration is predicated on two things: 1) the migration will allow you to take advantage of new capabilities on the new platform, and 2) the migration will give you access to the community and vendor support for the new platform. We will assume the new platform measurably out-performs the legacy platform in terms of capability and support, but your mileage will vary according to your specific business needs and the value you place on community support, including the supply of skilled developers. Most often, the reality of platform change is that you just have to migrate. The only question is how to do it – in a timely and cost effective fashion.

What is a Migration Project?

An example of a migration project would be to begin with an application written in Visual Basic (using various COM libraries) and replace it with a similar application written in C# (using the .NET framework). The distinguishing assumptions of a typical migration project are listed below:

1. You have a reason and ideally a clear business case to migrate your legacy system to the new platform.
2. You have to make architectural improvements as you migrate to the new platform.
3. You are actively maintaining your legacy codebase.
4. You want to migrate without breaking the legacy functionality.
5. You want the new system to be more maintainable than the legacy system.
6. You want to migrate efficiently, with minimal disruption to the users of the system and other ongoing work.

Key Point	The Wikipedia defines Refactoring [1] as the process of rewriting a computer program or other material to improve its structure or readability, while explicitly preserving its meaning or behavior. A migration is more than refactoring: it is modifying a codebase to take advantage of a new platform and ensure that it can be efficiently maintained with the tools of that platform.

Migration Methodologies

Manual Rewrite Migration

A manual rewrite migration is re-building a legacy system on a new platform by writing new source code for it from scratch, sometimes in a different language and usually by hand. Such a re-write is usually based on an **external view** of the legacy system – how it looks and what it does, not how it is implemented at the code level – or it is based on a brand new vision of what the legacy system should be on the target platform. Occasionally the rewrite team will attempt to leverage the source code as a specification for their work. This code is typically very large and very complex having emerged over a period of years. There will also be many subtle, and not so subtle, differences between the legacy and new platforms and these make interpretation very difficult. An uninitiated developer can quickly be distracted by the many subtle differences and become unproductive due to the overhead of constantly switching back and forth between old and new tools and platforms.

In contrast, a tool-assisted rewrite is based on an **internal view** of the legacy system – i.e., how it is implemented at the source code level -- and results in a complete reproduction of the legacy system on a new platform. This is done by tool translation of the entire code of the legacy system to a new language and then by editing the translation by hand, if necessary, to take advantage of any advanced features offered by the new platform.

A translation tool is not typically employed on a manual rewrite, but one may be used to help developers figure out how to rewrite the code or give them a starting point. Beware that most off-the-shelf translators are low-yield and they leave a tremendous amount of difficult work to be done by hand.

Side Bar: The Cost of a Rewrite Migration

There has been a lot of discussion about how to estimate the size of a migration project, but there are still no easy answers. Some simplistic models look something like this [2]

Migration Effort = X * Y	Where X is a percentage, say 60% Y is the "original development effort"

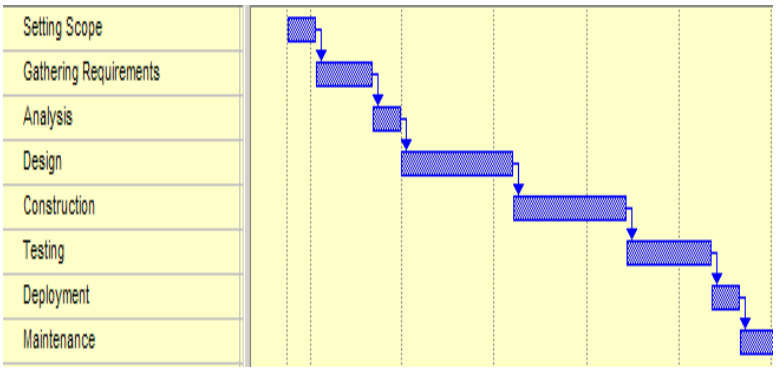
There are two problems with this model: both factors are unknown and difficult to estimate. First, the value of X carries a lot of weight, but it is usually an arbitrary guess. Second, it is almost impossible to come up with an accurate value for Y, "the original development effort". The "original development effort" to launch the application was probably for a much smaller and simpler scope that the current application delivers. The migration will have to deliver the current functionality, not the functionality of the first release, so that cannot be "Y". The cumulative effort of evolving a system over time may be in the tens or hundreds of person-years of effort and includes a lot of overhead. This is not a good "Y" either. To further confound this, your organization has probably matured over the lifespan of the system, but they may not be ready for the massive change entailed in a migration. This must be factored into the equation.

The true effort for a migration is much more sensitive to variables like organizational maturity, methodology, scope, and duration, than to an arbitrary X or the mysterious "original development cost". The only factor that is vaguely quantifiable is current system size – but no one agrees on what that means either.

The manual rewrite can be executed via a waterfall or a more iterative methodology.

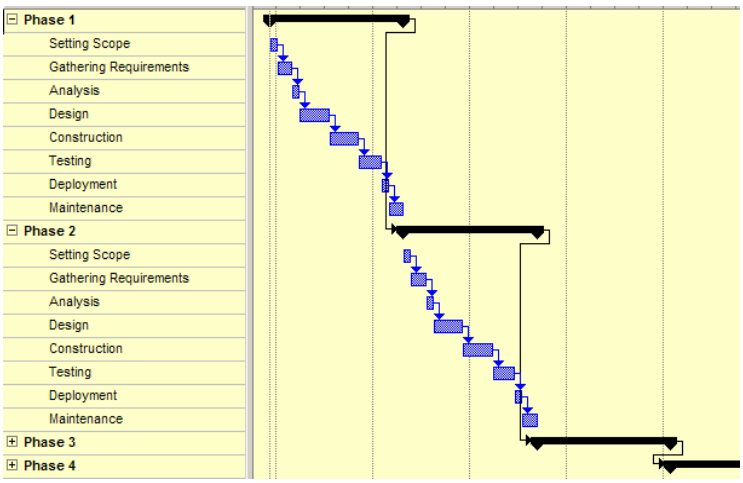
Waterfall Migration

The waterfall methodology is usually not palatable because it implies a long time between the start of the project and the eventual delivery of the new system. During this time, business requirements will change -- a known problem with a waterfall methodology. The changing requirements are even more troubling for a rewrite migration because the changes may have to be made in both the legacy and the new systems. If you are using a changing legacy code as specification for the rewrite, it will soon be out of date.



Incremental Migration

Instead of a waterfall, you will probably want a more iterative approach, or more precisely, an incremental approach that delivers the new system through a series of migration phases. In an incremental rewrite methodology, the legacy system continues to run as the new system is built, tested, and deployed. You will want to try to build the new system in a way that seamlessly integrates with the old system to avoid running redundant applications in production. The side-by-side environment will also create overhead for users trying to cope with changing rules for which application to use as functionality moves from the old to the new system. Even if you can hide the complexity from the users, the runtime environment, as well as deployment and operations activities needed to support both the old and new platforms and applications may be painfully complex.

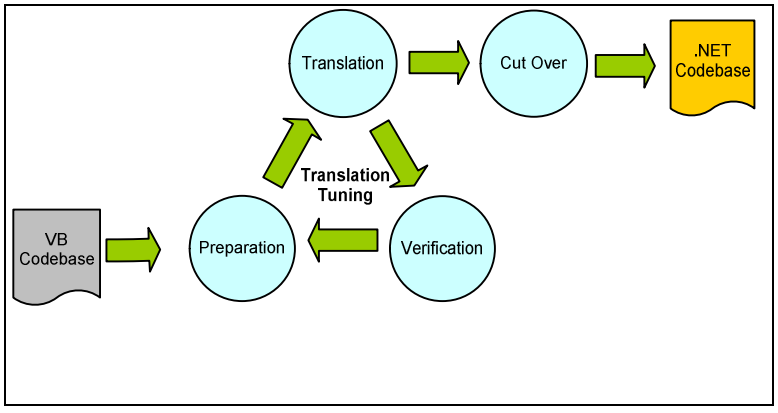


Tool-Assisted Migration

Key Point	In this document, the phrase "tool-assisted migration" means a high-yield tool-assisted migration based on the Great Migrations Methodology (GMM).

The defining feature of the GMM is its highly iterative approach and its use of proprietary migration tools to perform a highly-customized, automated, system-wide refactoring.

Every GMM project includes a translation tuning phase in which the migration team identifies and solves strictly migration problems. The deliverable is a codebase produced automatically by a custom-tuned translation tool and a “Fit and Finish” plan for balancing the automated and manual work to ensure the new codebase meets particular production standards on the target platform.



Side Bar: The Cost of a Tool-Assisted Migration

The major driver of cost for a custom tool-assisted migration is translation tuning. The cost of this work can be computed based on the following formula:

```
Migration Effort
= SUMi (Ti + Fi)
```

```
Where
Ti  Is the effort to tune the translator for a given type of
    migration issue in the legacy codebase.
Fi  Is any manual coding needed to finish translations
    produced by the translator (if any).
```

The effort to tune the translator for a given migration issue (T) is proportional to the complexity of the differences between the code the translator produces and the desired translation. The second factor, F, is any manual coding needed to finish translations produced by the translator. Note that investing more in T (tuning) will reduce F (manual finish work). Note also that greater economies of scale are achieved when the tuned translator is applied to larger codebases.

Phase by Phase Methodology Comparison

The following sections compare the Great Migrations Methodology to other approaches in the context of the standard software development lifecycle phases starting with setting scope and running through maintenance.

Setting Scope

Job one for your migration project will be to identify a set of one of more legacy systems that will be migrated. You will also have to scope the project in a way that meets the business case and minimizes disruption to other work.

Know what you are getting into and Why

New technology is usually hyped up as the magic solution to our business problems, and this can create excitement and momentum for a migration project. You should expect business and technical people in your organization as well as your technical service providers to try to use your project to cross-sell technology, make application enhancements, address the maintenance backlog, and in general grow the migration into an ever-more-ambitious project. Some of this is taking advantage of economies of scale, but if it spirals out of control, the migration will end up trying to completely re-engineer the system and the processes that it serves. An out-of-control migration brings in technology for technology's sake, demands unnecessary technical risk, and promises to deliver an ever expanding array of new business features.

Pace Yourself: understand the technical features of the new platform – including updated in-house architecture and coding standards and SDLC changes. Research the most compelling features of the new platform then decide if you will include them, defer them, or avoid them altogether. Remember also that new technology is like a puppy, it is cute and fun when you first bring it home, but it grows up to be a lot of an on-going responsibility.

Must Use
- Required: you must use these features to create a valid application that builds and runs.
- Known to be mature and stable. (If a Must Use features is not mature and stable, you should defer your migration until it is.)
- These are clearly in scope.

Could Use
- These are optional features of the new platform that *might* improve the quality of your app in an important way. However, since they are probably not core features of the new platform, their maturity and stability, as well as ease of use and applicability in your application must be researched.
- Weigh the time-to-market case for the feature against the incremental costs of doing it with the migration project. Consider adoption after the initial migration when the feature is more mature and you are comfortable with the new platform,

Should Not Use
- Not useful to you, not mature, or known to be problematic.
- These are definitely out of scope.

Manage Disruption

Migrations can be disruptive if your development teams have to straddle the fence: supporting both the old and new world, but not firmly in either. This disruption means added complexity and operating costs, so a migration project should allow a quick transition if possible. If your legacy codebase is too large to allow a quick transition and will be done over many release cycles, you will have to make migration work part of your standard SDLC and portfolio management process. For very large migrations the codebase will be divided into migration sets that are converted with each release cycle.

One of the key benefits of a tool-assisted migration is speed of transition. The main effort is tuning the translator and this will be completed for each system before you decide to "cut over" to the new platform. Note also that legacy development streams can continue to evolve in parallel with translation tuning; there is no need for a code freeze until you are ready to translate the legacy code and begin maintaining it on the new platform.

Without adequate tool support, it will probably take you a very long time to migrate the codebase, so you will end up supporting both the legacy and new development streams, doing duplicate work, and merging changes. If you do not have resources to support parallel streams, you will have to freeze or chill legacy development and shift those resources to the migration stream.

Key Point	The primary goal of a migration is to move an existing system efficiently to a new platform so the system team can resume work on meeting new business requirements.

Summary

Remember that even a bare bones migration is complex and risky. It requires significant planning, preparation, and testing. Exploring all the possible technical and functional nice-to-haves demands more coordination and due diligence and the increased scope will increase cost and risk and ultimately reduce the ROI of the effort.

- Functional enhancements should be evaluated, scoped, scheduled and delivered through your standard project methodology – not combined with migration work. The ideal functional scope of a migration project is functional equivalence; going beyond this may make sense, but it is beyond the scope of migration work.

  If functional changes are timed properly, the GMM allows them to be made in parallel with and independently of the migration work. See the discussion on testing for details.

- The ideal technical scope for a migration project is limited to the set of changes needed to provide a functionally correct code that meets your architectural and coding standards for the new platform. Bear in mind that a high-yield translation tool is an enabler, not a constraint. It can help you improve existing source code by making API replacements, and restructuring the code to meet new architectural standards. A translator can help you stretch your migration budget and allow for more, not less, technical improvements.

Gathering Requirements

Migration projects, like all other IT projects, are defined in terms of two main types of requirements: Functional (what you want the system to do) and Technical (how you want the system to do it).

Functional Requirements

One of the first questions for any IT project is: what should the new system do? For a migration, the simplistic answer is: "the new system should do the same things as the legacy system." Now consider that most large-scale business systems are born through a big first release or launch and then they grow, and grow, and grow over a period of years: release after release, fix after fix, enhancement after enhancement. By the time a major business system is five years old, it can easily contain over 50 person-years of development effort. So the simplistic "do the same thing as the legacy system" leads to a huge question: how to figure out, in complete and accurate detail, everything the legacy system does. Answering this question is at the heart of the migration project.

Technical Requirements

Technical requirements include coding and architecture guidelines and library and component use standards that make the new code more maintainable, reusable, scalable, robust, etc. There may also be changes to configuration management policies, unit testing, and deployment and operating procedures. Scalability and robustness requirements in particular should also be formalized and new technologies should be stressed to ensure that their beauty is not just skin deep. Clearly the migration is more than just a language problem and the migration team must carefully control both the total cost of conversion (TCC) as well as the long term total cost of ownership (TCO); careful selection of technical requirements is the key controlling both of these costs.

Migration Requirements Gathering

In Table 1, the effectiveness of various sources are rated (graded from “F” to “A”), depending on the relative value each source delivers to the process of gathering Functional and Technical specifications for the migration.

Table 1: Effectiveness of Migration Requirements Sources

Source	Functional		Technical	
Users	D	• Insufficient detail, possible inaccuracies • Mixed with enhancement Requests • Expensive, difficult to use requirements	F	• Typically lack the of depth technical understanding needed to define detailed technical requirements for the migration.
Developers	D	• Insufficient detail • Mixed with enhancement Requests • Expensive	A	• Good understanding of old/new system limitations and target standards. • Highly motivated to provide standards.
Existing Documents	C	• Insufficient detail, possible inaccuracies • Difficult to use requirements • Probably do not even exist	B	• Useful for understanding the target platform's technical capabilities, constraints, procedures, etc. • May be out of date.
Read the Old Code	A	• The original source code is a detailed and tested specification of the application functionality and low-level design. • Exception processing and many otherwise "hidden functions" are revealed in the code.	A	• The original source code is a detailed and tested specification of the application implementation. • A huge number of low-level technical requirements are specified in the source code.

One of the most important advantages of the GMM is that it can use your legacy source code as a functional and low-level technical specification for the new system. The GMM also makes extensive use of developer knowledge and best-practice documentation for the high-level architecture standards that cannot be derived from the legacy source code.

Analysis

The purpose of the analysis phase for a migration is to more rigorously understand the requirements and the problems that you will face during the migration. Some examples of analysis tasks include:

- Elaborating the details of the application functionality that will be migrated.
- Identifying specific language incompatibilities between source and target platform that will have to be solved and addressed throughout the system wherever they occur.
- Identifying legacy APIs that must be replaced with new APIs throughout the system wherever they occur.
- Identifying source code that has technical problems that must be addressed in order to provide correct results.
- Identifying features of the new platform that have sufficient ROI to warrant an investment in re-architecting some or all of the application to use them.
- Identifying software development, configuration management, testing and deployment processes that will have to be changed to work better with the new platform.
- Identifying the system boundaries that can be used to group legacy components into "migration sets" and processed in a series of migration phases or over a series of release cycles.

In general, the analysis phase of a tool-assisted migration project will be the same as one that will be done by hand. The main difference is that an in-depth elaboration of application functionality is not done because the legacy code will be used as a detailed functional/technical spec. Also many of the technical problems will be routed to the translation tuning team so they can be handled by the translator instead of being passed off to developers for manual work.

Design

The purpose of the design phase for a migration is to solve the problems identified in the analysis phase.

- Designing detailed solutions for replacing language program structures.
- Designing detailed solutions for replacing legacy APIs and class libraries.
- Designing detailed solutions for improving portions of code that have to be redesigned and re-written.
- Designing software development, configuration management, testing and deployment processes for the new platform.

The design deliverables for both a tool-assisted migration and a manual migration are the same: documents that describe how migration problems will be solved. However, the format and usage of these documents in the two approaches are very different.

For a manual migration, designs are in higher-level documents. They are verified by developers as they are used, recurring problems typically go unreported as most developers choose to implement solutions and move on rather than try to get the documents changed. Deviations from the published standards may be addressed after the fact, if code reviews are done and followed up on; otherwise, different developers will solve the same problems over and over again.

In the GMM, the design docs become the translation configuration (See Sidebar: Translation Configuration). The configuration is written in a formal XML-based notation that controls the translation toolset. The configuration is verified by an iterative process called translation tuning. Translation tuning uses manual and automated code reviews, builds, and a variety of runtime tests to identify issues in the translation and address them in the translation configuration.

Sidebar: Translation Configuration

The Promula translation technology used in the GMM is controlled by configuration files (XML) that specify translation details. Some of the things controlled via configuration are:

- Rules for mapping of internal and external COM/Win32 APIs to .NET replacements;
- Rules for mapping ASP/VB6 coding patterns to .NET coding
- Processing order for a multi-component translation
- Instructions or codeblocks to address problems or anomalies in the source codes
- Rules for file names, folder names, and the breaking of source files into separate parts
- Rules for target project settings (csproj, vbproj, resx, assemblyInfo)
- Rules for code formatting – blank lines, comments, indenting, boilerplate code

Construction

The purpose of the construction phase for a migration is to implement the new application on the target platform.

Migration projects often require a tremendous amount of construction work because they have to deliver a mature business system that evolved over many years to provide many functions. Even if a translation tool is used, there may be a fair amount of original coding needed to create architecture components needed for the foundation of the new application. An additional type of construction typically done for tool-assisted migrations is the construction of small test codes, built on the legacy platform, and used to provide isolated tests of specific translation problems.

The majority of coding however is automated – the conversion of application business logic is handled by the tool. Promula Technology allows you to translate an entire migration set – often hundreds of thousands of lines of code in a matter of minutes.

Testing

The purpose of the testing phase for a migration is to verify the new application on the target platform. The testing activities are normally the same for manual and tool-assisted migration work.

In the GMM, the new application will be up and running earlier in the project. This is important because runtime testing is a critical source of tuning information. Also you will be leveraging the tested legacy code as a design, and testing begins and runs throughout translation tuning, so few defects should be found after cutover.

Warning	With a manual migration you will have much more original work: new requirements were gathered, new designs were created, and plenty of new code was written by hand. As a result, you are likely to face a higher than usual number of defects, and you should plan for a longer (more costly) test phase.

Sidebar: Side-by-Side Testing

You will use the latest, stabilized production source for CutOver, so it is likely this version of the legacy application will still be installed in a test environment. This will make it easier to set up side-by-side testing between the new system and the legacy system.

Side by side testing can be a life saver for verifying migrations of large systems that do not have robust regression testing processes.

Side-by-side testing includes visual comparison of the application UI as well as automated comparison of digital outputs such as reports and web pages.

Warning	Functional differences between your legacy application and your migrated application make side-by-side testing more difficult and this can drive up already significant testing costs for the migration.

Cutover

Cutover is when the migration team takes on the tasks of finishing up the system and getting it certified for deployment. With a manual translation this begins after design and runs through construction and testing. A manual cutover usually takes a very long time because there is so much new code to refine, build and test. Because of its long duration, the legacy system will have to be maintained in parallel with the cutover effort — probably including merging changes.

The GMM also has a cutover, but it occurs only when the team is satisfied with the quality of the translations and confident in their ability to quickly take the system to production and support it. To do a Cutover, the latest, verified production source code is translated one final time. From that point forward, the application will only be maintained, tested, and deployed according to the rules of the new platform. The tuned translation should be very near production-ready which allows it to be deployed soon after translation. Some post-cutover construction tasks may be needed but they are identified and planned during translation tuning, so there are no surprises. In fact, much of the work can be done and verified in advance. Translation tuning means the work is done up front so you can enjoy a quick and easy cutover.

Key Point	**No Need to Freeze:** With few exceptions, the legacy codebase continues to change until right before cutover. A copy of the latest, verified production codebase is usually used.

Deployment

The purpose of the deployment phase for a migration is to put the new application into a runtime environment for testing or use. This phase is essentially identical for manual and tool-assisted approaches. The one difference is that in a tool assisted approach, the new codebase will be ready for testing sooner and thus build and deployment processes can be designed and refined sooner.

Maintenance

The purpose of the maintenance phase for a migration is to get back to the business of enhancing and evolving the system – and of course demonstrating to your users that you are much more productive because of the new platform.

In general maintenance is maintenance whether you did a manual or tool-assisted migration. But there are a couple a subtle differences:

- In a tool-assisted migration the new code is derived from the legacy code and so it will be familiar to the legacy development staff. This flattens the learning curve so the staff can become productive with the new system sooner.

- In manual migration you will be running more new work: new requirements were gathered, new designs were created, and plenty of new code was written and some details probably slipped through the cracks. As a result, you are likely to find a few more defects as the system settles under the load of production use.

Summary

A technology platform change can be extremely disruptive. No matter what approach you take, your organization will be expected to master new paradigms, change processes, learn new tools, and implement new technologies. As you plan and execute your migration, remember that although migration is extremely technical work, in the final analysis, preserving functionality is paramount.

In conclusion, the GMM minimizes cost and risk while preserving quality and control:

- First, it fully leverages your legacy source code investment. There is a tremendous amount of information in the legacy code and using it properly can reduce the migration effort and risk dramatically.

- Second, our translation technology is an enabler not a constraint. It allows the migration team to refine and channel their collective knowledge into a way that reproduces and multiplies what they would do manually. The tuned translator systematically produces very good code very quickly, and by a repeatable, measurable, improvable process. Our translation technology, will allow you to translate your codebase – often in a matter of minutes – not just once, but many times. You can make specific improvements until you are satisfied with the translations.

- Third, you control when to do a cutover. The latest, verified production source code is translated one final time and the resulting code will meet your coding, architecture, and configuration management targets so you can expect a smooth test cycle and an extremely efficient transition.

Note: this article was published in the January 2007 issue of .NET Developer's Journal.

http://dotnet.sys-con.com/read/346924.htm.

Appendix: Translation Myth Busters

"We just have to rewrite it."

Re-writing *some* of the application makes sense and our translator is designed to take advantage of this. We can configure the translator to integrate translated code and hand-written code automatically. In addition, the real work of a re-write is in designing the new architecture standards and coding patterns – not typing them in – and our translator can be configured to automate radical transformations and then apply them correctly across an entire codebase. This flexibility delivers the best of both worlds: the quality of a custom rewrite delivered with the efficiency and accuracy of an automated tool.

"The code produced by the translator will be 'junk'."

Look at the translation, find the 'junk' and tell us how you would write it better. We will then reconfigure the translator to your specifications and produce an improved translation. We can repeat this process until all "junk" is removed. If there is something we cannot do 100%, we can probably get you very close so you can finish it more efficiently than trying to re-write it all by hand.

"Only a person can do 'it'."

What exactly is 'it' that the person will do? Can re-factoring tools help with some of those tasks? If you can specify what a person can do, then the tool can be customized to do it and do it repeatedly – if not all of it, at least most of it so that only "finishing touches" are left to be done by a person. Balancing the work that a person should do with the tasks that a translator should do results in increased efficiency while preserving quality.

"The legacy code was designed for a different platform. It is worthless"

The legacy code meets the business requirements of your customers; and it describes what your program does in detail. Since we must preserve the functionality of the application, the majority of the code will be semantically similar, even on in the new platform. The source code will do the same functions; it will just describe those functions in a different way.

"The legacy code lacks the detail to describe the new system."

What additional information is needed to describe the new system? You will have to figure that out even if you plan to do it by hand. By putting this additional information into a form that the translator can use, we can let the translator do most of the work for us. For example, if one describes how to replace legacy COM APIs with .NET classes, the translator will make those replacements for us.

"It cannot be done."

"What is 'It'? What are the specific things that the translator cannot do? Why do you think the translator cannot do them?" Bear in mind that your system is processed by a "translator" every time you do a build: it is translated from source code into machine code by a type of translator called a compiler. Our translator works very much like a compiler, the difference being it emits source code instead of machine code, and it is extremely configurable so that the format and content of the source code can be customized to your requirements.

The problem of computer language translation is a difficult one, and this fact is underlined by the many translation tools that are inflexible or do a poor job. The Great Migrations tools are different:

- The tools have matured over almost thirty years of research and refinement in the science of computer language translation.
- The tools use a semantic approach that handles complex restructuring much better than a syntax-driven approach.
- The tools are flexible and can be tuned to meet different source and target code requirements.
- The tools are designed to work with large codebases comprised of many interdependent components.
- The tools are part of a methodology that refines and balances automated translation with manual work.

"It is too much work to figure out how to restructure the old code to new code."

Yes it will be a lot of work, but if you want to use the old code as specification you must figure how to restructure it anyway. On the other hand, if you are not using the old code, you are basically building a brand new application and, for any significant application, this is known to be a very expensive and time-consuming process, and you will end up doing much more work than a tool-assisted conversion. Consider how many person years of work are in the legacy application: 50? 100?

Appendix: References

[1] The Wikipedia Refactoring: http://en.wikipedia.org/wiki/Refactoring

[2] A Cost Model for .NET Code Conversion. Gartner Group, COM-16-1385, R. Valdes, M. Driver