Porting vs. Manual Migration

Porting, Migration, Conversion, Modernization, Translation, Transformation are all the same thing. The difference is in the technology, methodology and the people behind it all.

Many words can be used to define a Porting Project. We can call it Migration because we are *migrating* to a different system. Or we can call it Conversion because the application is being *converted* to run on a new platform. We could call it Translation because, after all, the source language is being *translated* to a different language. And we could also call it Modernization, since the final product will (hopefully) be *modern* software.

Ultimately it doesn't really matter what we call it. The project *starts* with software written using Gupta SQLWindows/Team Developer and *ends* with similar software written in one of the .NET languages. The source application requires the Gupta runtime and the Gupta development tool. The target application requires the .NET Framework and Visual Studio.

In a nutshell, the Source Gupta Application is **Ported** to a Target .NET Application.

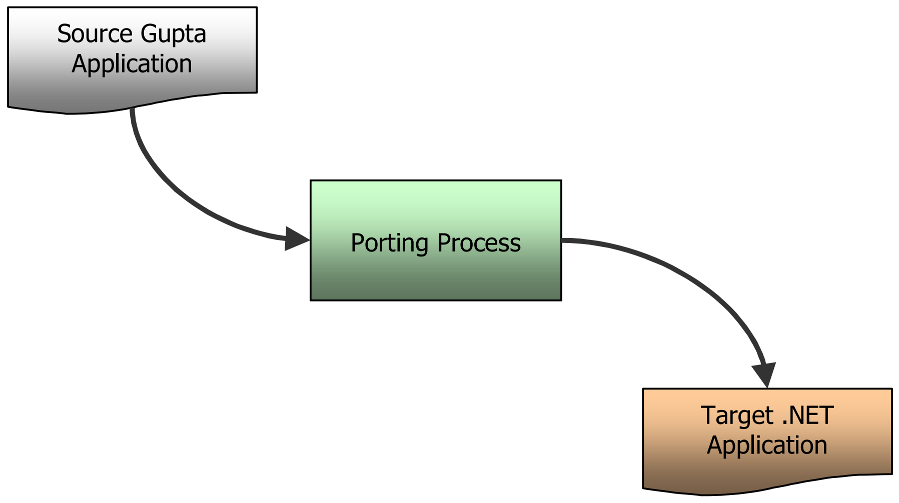

Figure 1 - Porting Project lifecycle

It doesn't get more complicated than this. However, the key is in the process box (if you remember flowchart diagrams), the one labeled "Porting Process". Whatever happens in that box is what makes the difference and what determines the shape of the Target .NET Application. (For a detailed description of what to expect in a Porting Process, read the *Porting Process* white paper.)

After the application has been ported to .NET, it must run and must run well. It must run at least as well as the source application was running before the migration. It must also be manageable, extensible, understandable to both .NET and Gupta developers, and it must be a good .NET and Visual Studio citizen.

The structure of the new code should be consistent, well organized, traceable from the original application and viceversa (to simplify the life of the developers and testers), should work well with Visual Studio's IntelliSense, Designers, Documentation tools, Code Analyzers, Refactoring tools, and any third party tools. Simply put: the generated code must be good .NET code.

Automated Porting or Manual Migration

Do you want to rewrite all or parts of your application using external developers that know very little about your system's architecture and your business processes? Or do you want to rely on an automated and tested tool that preserves all the original system's Business Logic and User Interface? This is the question.

We'd like to save your time: If you have decided that your current Gupta system is not worth saving and you want to redesign and rewrite the application from scratch, then you can stop reading this document.

If a sales person, probably without any real Gupta migration experience, is trying to convince you that rewriting parts of your code by using their consultants is better than automated porting, keep reading. We may be able to save you a lot of time, a lot of money, and a big headhache.

Manual Migration involves using 1) a tool that partially ports the simplest parts of the code; 2) consultants and developers that try to reverse engineer the source code and rewrite from scratch the parts that have been left out by the tool.

The tool usually generates large log files containing a long list of issues that must be addressed by the developers. The structure of the code that is generated by the tool is usually not very coherent and poorly organized. The focus of the migration is not the tool, it's the manual work needed to fill in the (sometimes large) gaps and make the application work. It's usually priced and managed as a *Consulting operation*.

Additionally, when the external developers start coding away they necessarily end up having to heavily modify the generated code to hammer it in place. The result is most likely "Frankenstein" code: A mix of different approaches, inconsistent coding styles, incompatible architectures (plural), and a nightmare of testing and potential production failures.

The sale pitch is that after the external developers are done rewriting your code you get "real .NET architecture", according to the architects, of course. But… If the manual work is so much better than the automatic work, why not go for a total rewrite? The answer is that it would be too expensive and unreliable: you'd be restarting from release 1.0.

So, the sales guy says, a mixed Manual Migration approach gives you the best of both worlds: you save money on the parts that can be generated automatically, and get a good architecture for the parts that are written manually. It's a cost/time/quality compromise. Sounds good, on paper, but it's **not true** in reality. By the time you find that out it may be too late to rollback. You will have to keep coding and coding and coding until the project works... or it's scrapped. And when you are done, the newly coded parts are probably at release 1.0 level anyway.

This is why we strongly advise to verify before buying into any buzzword or marketing pitch (including ours). Ask for a full translation of your code, ask to try a Proof of Concept –make sure that the POC contains a good selection of the various techniques used in the application– and ask to verify and test first-hand how the Proof of Concept behaves with real data and real users.

Automatic Porting uses 1) a sophisticated translation tool that ports *all* the source code using consistent and tested algorithms and structures; 2) developers that adjust and implement the few and well defined parts that have been clearly isolated and tagged by the tool. There is no reinventing the wheel and the outcome of the project is guaranteed to be a success. The result is a consistent, solid and coherent code based on a reliable architecture that is fully compatible with .NET architecture and your system's architecture.

We have the best *Automated Porting* solution in the world. That is why we are open to any test and evaluation, we do in fact encourage all porting candidates to verify and compare.

Porting an existing enterprise system to a completely different platform is a serious and delicate business. Thinking that you can remove the foundation from under a building and replace it with something else that will be designed as you go along, is not a good idea. Only a reliable, road-tested, and solid process and technology can be considered a viable migration solution.

We preserve your release level.

Checklist for migration solutions

In the table below we have collected a number of issues that we know to be crucial in any porting project. The issues are related to the translation of the source code, report templates, and to the compatibility library used by the new system (all migration solutions have a compatibility library).

This list is based on a vast experience in porting. We have ported a large number of applications, the biggest Gupta applications, and the most complex Gupta applications. In our experience, any of the listed issues is enough to break a project or a deployed application. Lack of full support for any of the issues listed below can mean countless hours of development work during the project, deployment failures, and ultimately a product that doesn't work as expected.

✓	**Late Bind Calls**
Description:	In SAL (Gupta's programming language) there is a peculiar syntax called "late bind call". It's used by prefixing a local method call with two dots. The result of such a call is to make the function being called fully virtual, therefore the control goes to the last override of the specified method. However, the same function can be called without the double dots, which results in calling the local implementation.
Our approach:	We fully translate all late bind calls into real virtual methods. Including late bind calls coming from multiple-inheritance classes. Our approach does not use any reflection or runtime trick, we transform the original SAL to the best possible Object Oriented equivalent. We do not require additional manual work.
Bad approach:	Calling late bind methods using reflection is probably the worst possible approach since it's not needed and results in breaking the reference to the method that is now a string in the new code, interprted at runtime.

✓	**Multiple Inheritance**
Description:	Gupta Team Developer supports Multiple Inheritance. Microsoft .NET supports only Multiple Interface Inheritance. Most Gupta applications use multiple inheritance to share functionality among classes without duplicating the code.
Our approach:	We fully translate all multiple inheritance structures to the best possible single inheritance equivalent by using the delegation technique and by leveraging the use of interfaces and operators overloading. The result of our transformation of SAL class structures into .NET class structures is that the inherited code is never duplicated. After the migration is complete, developers working on the new application can work on the shared base classes without having to modify the same code in all inherited classes.
Bad approach:	There are several approaches for simulating multiple inheritance in a single inheritance environment. The worst is to duplicate the code inherited from second-base classes into the derived classes. Other bad approaches consist in creating a parallel interface system and delegating calls to static methods.

✓	**Visual Inheritance**
Description:	Gupta Team Developer fully supports visual inheritance, where a visual class (i.e. DataField, CheckBox) inherits all the visual properties (i.e. Background Color, Alignment) of the base classes.
Our approach:	We fully support visual inheritance both at runtime and design time. That means that developers using the code that we generate, can alter a visual property in a base class and automatically all the derived classes inherit the new value. This is possible only because our support library (PPJ Framework) extends Visual Studio designers and considers properties inherited from base classes like properties inherited from a base form. By default, in Visual Studio, the property values inherited from classes are copied in the form and the inheritance is broken. That means that without our extension if you want to change the font of the base class for all labels, for example, you have to do it on each single label throughout the application.
Bad approach:	Copy visual property values in each form's InitializeComponent() method. Sometimes the property values are not copied during the initial translation, but as soon as the form is modified in the designer all the properties (even properties that should be hidden) are serialized into the form.

✓	**Visual Multiple Inheritance**
Description:	Gupta Team Developer fully supports also visual multiple inheritance, where a control class may inherit properties and Message Actions handlers from multiple base classes. Also, in Gupta, a form class may inherit child controls from multiple base form classes. Microsoft .NET doesn't support Multiple Inheritance and much less Visual Multiple Inheritance.
Our approach:	We correctly translate visual multiple inheritance into a special InitializeComponentEx() method and preserve all the property and event handlers inheritance both at runtime and design time.
Bad approach:	Ignore this (common) case and approach it by hand. It costs a lot of money and inevitably the solution is to duplicate code and property assignments. In same cases the solution added by hand ends up also being incompatible with Visual Studio.

✓	**Automatic Code Modularization**
Description:	Gupta Team Developer never supported real dynamic modules. Applications are usually compiled as big monolithic executables. Dynalibs are a partial solutions but have never worked well and are seldom used in production. One of the many benefits of changing technology and moving to Microsoft .NET is to be able to use assemblies and dynamic loading to modularize an application.
Our approach:	Our translation tool (Ice Porter™) is able to automatically reference multiple plug-in assemblies and remove the code already ported from the application being translated. By using the "differential" approach we can automatically split any application into dynamically loaded modules.
Bad approach:	Reorganize the code and references manually after porting. While additional reorganization of the separated module is not a bad thing by itself, approaching the entire modularization refactoring by hand is costly, inaccurate and will lead to many problems.

✓	Table Window Control
Description:	The most complex and thus the most custom control in Gupta Team Developer is the Table Window Control. Like most grid controls out there, it is heavily proprietary and tailored for the Gupta environment. Most SAL application use it in a many ways, including complex additional customizations.
Our approach:	We have licensed the most flexible and well-known grid control for the Microsoft environment (FlexGrid.NET from ComponentOne) and we have built a Table Window interface around it. All SAL functionalities are fully supported (including the different populate methods, column controls, split table, etc.), while the base grid control is still fully accessible to the code to allow for future enhancements. We also support the XSalTableWindow and M!Table extensions. We have an OEM license for the FlexGrid.NET that allows us to redistribute it with our PPJ Framework to any number of developers.
Bad approach:	Build a custom control that resembles the Table Window, or try to support only the most common features by porting to a DataGrid or a third party grid and leave most of the complex work to be done by hand. Also not supporting population methods such as FillNormal or FillBackground can be a major problem when the application has to deal with real data, which usually occurr in production, which is the last place where you want to see errors of this kind.

✓	Child Table Windows
Description:	Child table window controls are a very special object in Gupta Team Developer. They are a mix between an instance of a control on a form and a full fledged class. In fact, child table windows are the only control that allows you to add fields and methods and to override late-bind calls directly in the control definition on the form.
Our approach:	We generate child table controls as child classes fully encapsulated in the hosting form. This approach allows us to support all the features of child table controls, to keep the code clean, avoid the proliferation of oddly named classes, preserve and support all SQL statements, and to fully design the child control thanks to an enhanced designer that extends Visual Studio default control designer.
Bad approach:	Porting child table windows to a separate class is a decent alternative, however you end up with many additional classes that did not exist in the original application, you have a naming convention problem, maintenance problems and you still cannot fully design the child table control on the form. Trying to fit child table controls into a simple control instance is the worst approach because it doesn't work with most of the real code and requires a large amount of manual work and major rewriting of the code being ported and of all the SQL statements scattered around the application.

✓	Database Connectivity
Description:	Gupta Team Developer connects to a variety of databases through custom drivers. The most common database engines used by Gupta applications are Oracle, SqlServer, and Gupta SQLBase. Additionally, Gupta can connect to most ODBC and OLEDB sources. Connectivity was one of the major strength of SQLWindows/Team Developer.
Our approach:	Our connectivity layer is fully based on ADO.NET and we support **any** ADO.NET driver. We transparently support both connected (DataReader) and disconnected (DataSet) modes. Applications ported using our solution work equally well for few records or for several millions of records.
Bad approach:	Supporting only predefined ADO.NET drivers limits the connectivity of the new application. Not supporting the connected mode (DataReader) simply makes the application unusable for large result sets, increases the memory consumption on the client side, and slows down the application considerably.

✓	**Oracle Support**
Description:	Gupta Team Developer implements additional support for Oracle anonymous PLSQL blocks and stored procedure calls with return arguments and array return arguments in special functions.
Our approach:	We transparently support all Gupta's SqlORA* functions through Oracle's ADO.NET driver and Microsoft's Oracle ADO.NET driver.
Bad approach:	Ignore Oracle's non-standard functions thinking that they can be easily re-implemented manually. It's not easy and it will cause major problems in the project.

✓	**SalCompileAndEvaluate**
Description:	SalCompileAndEvaluate() allows Gupta applications to execute expressions at runtime. Unfortunately there is no equivalent in .NET and this feature is widely used in most SAL applications.
Our approach:	We fully support SalCompileAndEvaluate() using a simple and very fast script evaluation engine that can support any syntax. The built-in syntax is C#, but we can support SAL, VB.NET or any language that can be parsed. Our script evaluation engine used reflection and dynamic IL generation and executes much faster than the original SAL application.
Bad approach:	Pretend that SCAE is the same as reflection in .NET and claim that it's not a problem, until it is a problem and you have to pay dearly for all the extra work and delays caused by overlooking a very important feature.

✓	**Bind Expressions**
Description:	One of the strength of Gupta Team Developer was the support for bind expressions directly into SQL statements. It was a kind of "Embedded Sql" as seen in various C/C++ preprocessors and other languages. Bind expressions in SAL can be anything, including calculations, array element access, function calls, member access, etc.
Our approach:	We parse out all bind expressions and adapt the SQL statements to the selected ADO.NET driver, and we use our evaluation engine to resolve the bind expressions. The result is that the original SQL code doesn't need to be rewritten and the execution is extremely fast (faster than the original application and faster than reflection and many .NET object binding libraries) because of our advanced dynamic IL code generator.
Bad approach:	Attempt to reorganize and rewrite all SQL statements and code in the application to eliminate bind expressions. It costs a lot of time and money and the result is usually a mess because SQL execution and binding are usually at the foundation layer of applications. Some solutions try to use a callback system (via events) to simplify the manual code rewriting tasks which also slows down the code considerably and makes the logic of the application hard to follow.

✓	**Memory Leaks**
Description:	Gupta developers never have to think about releasing memory or releasing references. Team Developer takes care of memory management and doesn't keep any hard reference to controls.
Our approach:	We have eliminated any kind of root referencing problem that cause a lot of memory leaks in .NET. Microsoft .NET uses the garbage collector technique to free memory used by objects not referenced. However, it's enough to store a reference to a control or an object in a static variable (or array) to "leak" memory. For example, if the porting solution translates SAL's Window Handle to Control or to any custom type that keeps a reference to the control, you most likely will have a big memory leak to deal with. We use weak references and an idle loop to dereference closed forms to ensure that we do not cause any memory leak. We also have fixed several bugs in the .NET Framework that are known to cause memory leaks.
Bad approach:	Ignore memory management thinking that the Garbage Collector takes care of everything. Any .NET developer knows that it's not the case and you have to make sure that both the supporting library and migrated application do not hold on root references longer than needed.

✓	**.NET Missing Features and Controls**
Description:	Microsoft .NET Framework is a huge collection of libraries and classes. However, many features and controls that are available in Gupta Team Developer are not implemented in the .NET Framework.
Our approach:	We have extended all the basic WinForms controls by adding all the extended features that are needed by professional business applications and have implemented all the missing features that are needed by ported code.
Bad approach:	Implement missing features and controls in each project each time. The client will have to pay the cost for rewriting and retesting the same code, and future bug fixes can never be applied to "modernized" systems in a consistent and reliable manner.

✓	**.NET Native Types**
Description:	Gupta's SAL data types and .NET data types are similar but not quite the same. For example, in .NET you cannot have a null date, or a null number, cannot concatenate or use a null string, and so on. However, in .NET, there are extended types used for database access. There is a SqlDecimal, SqlString, OracleDecimal, OracleDate, etc.
Our approach:	We have developed extended types that replace Gupta types and are compatible with .NET's custom database types. Our type system is also fully compatible and based on .NET's native types: SalNumber wraps a Decimal value, SalDateTime wraps a DateTime value. Additionall all PPJ types support seamless and built-in casting to the corresponding native type. During the translation process, our tool decides to use native types directly when it's absolutely safe to do so (there is no risk of breaking code at runtime), and uses the extended types in a coherent and consistent fashion. We can also plug-in additional optimizers that are able to determine the usage of a variable and select to use the most optimized native type. In addition to all these options, we also support special comments in the original SAL applications that allow us to indicate the preferred type directly in the original application.
Bad approach:	Force the usage of native types without a proper analysis of the code and let the testers deal with the runtime problems. Have an incomplete type system that cannot be used in new development. Another bad approach is to write the extended type system by using classes instead of structs. Classes are garbage collected and any numeric value ever created in a function, loop, or any place in

	the app uses memory until the Garbage Collector frees it. In a normal application in production this can account for millions of dead objects waiting to be collected.

✓	**Visual Toolchest Support**
Description:	Gupta Team Developer includes a former third party library called Visual Toolchest or VIS. This library implements many functions and controls that are missing from the standard environment. Many controls that are custom implemented in VIS correspond to controls in Windows Common Controls library.
Our approach:	We have fully re-implemented the Visual Toolchest library in C# by extending native .NET controls and classes, which in turn are based on Windows Common Controls.
Bad approach:	Implement only the most common control and functions and rewrite the rest by hand when needed. You end up with a mix of re-invented wheels, partially tested existing code, and a nightmare of testing and runtime failures.

✓	**Third Party Libraries**
Description:	Gupta Team Developer has been extended over the years by a handful of third party libraries. The most used ones are: XSal2, M!Table, and BuildingBlocks.
Our approach:	We already support XSal2, M!Table and BuildingBlocks. XSal2 and M!Table have been redeveloped entirely in C#. BuildingBlocks has been ported in its basic form and customizations that are found in each new project are added limited to the project.
Bad approach:	Ignore third party libraries and sell additional man/hours for redeveloping workarounds on each project.

✓	**Object Oriented Transformation**
Description:	Most of SAL calls are not object oriented since all SAL types (visual and value types) do not have any built-in method. It would be a good idea to refactor the calls when porting to .NET.
Our approach:	We support transforming all type-related and visual-related functions to Object Oriented calls. We are also able to refactor calls that use a receive argument into straight assignments. For example: *Call SalStrLeft(s, 10, s)* is tranformed to *s = s.Left(10);*
Bad approach:	Not support object oriented transformations or support it partially without proper refactoring.

✓	**Refactoring**
Description:	Translating a Gupta Application into a .NET application without trying to refactor the code adds little value to the newly generated code. There are a number of refactoring transformations that can be applied safely to the code and that increase the value of the new source code.
Our approach:	We apply several standard refactoring to the code: Object Oriented Transformations, MessageActions Event Encapsulation, Virtual LateBind Renaming, Embedding Resources, WhenSqlError Try/Catch Enclosure, etc. Additionally we can apply any custom refactoring technique that is suited to the specific project, like: Fields Encapsulation, Hungarian Notation Removal, Camel Casing, Ambiguous Reference Resolution, SOA/WebService Wrappers, and much more.
Bad approach:	Settle for few and inflexible transformations or apply the transformations by hand.

✓	**Customized Transformations**
Description:	Automatic Porting doesn't mean that each and every application is treated blindly the same. Contrary to what some sales people may contend, professional automatic porting can and should allow for all sorts of customizations and highly customized transformations.
Our approach:	Our porting tool supports a flexible plug-in filters technology. Where custom written translation filters (written in .NET and debuggable directly in Visual Studio) receive over 30 events during several phases in the translation process. Custom plug-in filters can be fully customizable and can interact with the code being ported at all stages: directly with Team Developer's CDK, CodeDom pre and post generation, and finally directly with the code renderer. Using such flexible architecture, we are able to implement and automate just about anything, on top of the several standard options already available.
Bad approach:	Not supporting custom transformations.

✓	**Support Library**
Description:	All migration solutions have a support library. It is not possible to port an application to .NET without using a support library.
Our approach:	We invest a major part of our development and testing efforts into our PPJ Framework. We make sure that it's fully tested, complete and the most reliable piece of software that we can possibly distribute to our customers. Since we also realize that one of the reasons why our customers want to move away from Gupta is to not be *locked in* a proprietary system and small company like Gupta or Unify, or Ice Tea Group, we also distribute the full C# code for our PPJ Framework. Making our source code available to our customers also means that we write the code in a way that is elegant, consistent with C# coding standards and easy to maintain.
Bad approach:	Not distributing the source code of the supporting library is a bad approach. Not having a coherent and solid basic support library is another bad approach. Having a support library written in Visual Basic or mixing different languages is also a bad approach.

Table 1 – Migration Solution Checklist

There are many more issues to deal with, but we have decided to keep the list short. We know all this because we have encountered all these problems in an application or another and have fixed them all. Because of the way our porting process is structured, any experience we make on a project is transferred to both our porting tool and our support library so that the next project will not deal with the same problem. This is possible only because we invest in our automated tools and support library trying to minimize manual work as much as possible.

Others have a different opinion and think that porting an application must necessarily be expensive and it's a good opportunity to sell man-hours. Problems found in past projects have to be dealt with each time as if it was the first time. Some sort of knowledge base may simplify this approach, but the solution may almost never be the same otherwise it would be automated.

Conclusions

This paper is based on our experience in real and large porting projects (the largest known Gupta applications in the world). Others may have a different experiences and opinions. Therefore we invite you to test our approach and technology in comparison with any other approach, including a full manual rewrite.

We are confident that we deliver the best possible quality, the lower possible risk at the lowest overall cost.