

Re-Engineering BLUE Financial System Using Round-Trip Engineering and JAVA Language Conversion Assistant

Salem Al-Agtash*, Tamer Al-Dwairy†, Adnan EL-Nasan†, Bruce P. Mull*, Mamdouh Barakat‡, Anas Shqair†

*School of Informatics and Computing, German–Jordanian University, P. O. Box 35247, Amman 11180, Jordan

†Hijawi College of Engineering, Yarmouk University, Irbid, Jordan

‡MBRM — MB Risk Management, Warnford Court, Throgmorton Street, London EC2N 2AT, UK

Abstract—Conversion of legacy software applications into a new technology platform is common in many of today’s ICT (Information and Communication Technologies) companies. The objective is to improve performance, as a result of modeling important aspects and features through the development of conversion technologies. Much focus has been made on devising efficient methodologies in software architecture research. This paper presents a combination of round-trip engineering (RTE) and use of JAVA Language Conversion Assistant (JLCA) to migrate legacy software applications developed in multiple programming languages into a uniform object-oriented platform. This re-engineering process is applied to MB-Risk Management’s BLUE™ Financial System software. An automated process is derived to migrate code from VJ++ (Visual JAVA™), C, and C++ (as used in BLUE™) to a consistent C# platform. The results of conversion show an overall efficiency of 93% of full code conversion for this automated process.

I. INTRODUCTION

Conversion of legacy software applications into a new technology platform is common in many of today’s ICT (Information and Communication Technologies) companies. Often, the objective is to improve performance, as a result of modeling important aspects and features through the development of conversion technologies. Sometimes, the objective is to convert software from an obsolete, unsupported environment into one which is current and fully supported to allow the software to continue to be maintained.

Much of the focus has been on devising efficient methodologies in software architecture research, including: reengineering class hierarchies using concept analysis, capsule-oriented reverse engineering, generating different views of a model using a slicing-based approach, and round-trip engineering.

Re-engineering class hierarchies [12] increases quality factors such as low coupling and high cohesion, but requires large numbers of complex analytical computations. Often the resources necessary to fully conduct this analysis are unavailable to the company, so other approaches must be used.

Capsule-oriented reverse engineering [1] involves the use of object-oriented mechanisms to determine an application’s

main features and includes data abstraction, information hiding, and inheritance. This facilitates system maintainability and enables software reuse, thus contributing to the improvement of the software’s evolution. The approach, however, has limitations, including: difficulty in reverse engineering of sophisticated inheritance hierarchies (together with polymorphism) and strict encapsulation of the generation of private class attributes.

The slicing-based approach [15] on the other hand, allows rapid reuse of the existing architectural design resources. Its problem is that it mainly focuses on the architectural side and not on implementation. Thus, this method is not really suitable for converting software between platforms.

Unified Modeling Language (UML) [10] has become a *de facto* design tool in the industry. Using UML, it is possible to translate up to 80% of the code by providing only about 20% of the modeling diagrams.

Round-trip engineering (RTE) [11] is considered to be the most efficient re-engineering approach. Its advantages include: automation of the design process of an application, automatic derivation of an application’s implementation, automatic updating of design documents, and the ability to update the implementation without loss of manually-created code.

II. AN OVERVIEW OF ROUND-TRIP ENGINEERING

The round-trip engineering process [3] was created as a result of merging two main processes, reverse and forward engineering. The complete process is illustrated in Figure 1 (next page). It is formally defined as:

Definition 1: Let D be the design of a product P . Let R be a reverse engineering procedure such that $R(P) = D$. And let g be a product generation procedure such that $g(D) = P$, then if $g[R(P)] \equiv P$, and thus $R[g(D)] \equiv D$, then (g, R) defines a complete round-trip engineering process.

Notice that the reverse and forward engineering processes are interrelated—the outcome from either process is used as input to the other. Each process is distinguished in a simple way. Forward engineering, for instance, involves generating

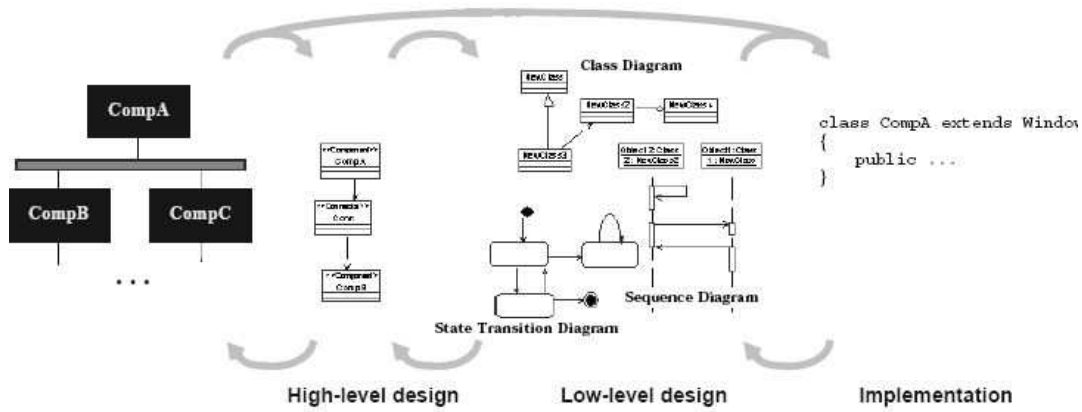


Fig. 1. The Round-Trip Engineering (RTE) process.

one or more software products that are closer in form and level of detail to the final system when compared to its inputs. Reverse engineering, on the other hand, involves generating one or more software designs having the same abstract details as the original system, but are possibly presented in a different form than the original input artifacts; the goal being to recover any information lost in the forward process.

Reverse engineering is first used on the entire application to extract its components and the associated relations among its classes in the form of class diagrams which help translate the design into a specific language. The class diagrams are usually not enough to generate a full conversion of the system, but they convert the most important parts: relations between classes, attributes, and headers of the methods. Therefore, they are used, together with language-conversion software, to forward engineer the application into the target language.

Much research has focused on RTE. In [4], different views of class diagrams are presented using the RTE process with FUJABA (From Uml to Java And Back Again) software as a CASE (Computer-Aided Software Engineering) tool. In [6], some of the important issues relating to automating the RTE process are clarified. In [5], different CASE tools are compared by their outcomes with respect to various issues. In [8], RTE was considered as a method for enhancing UML by assessing its suitability for modeling the architectural concepts. The result was a UML Analyzer, which provides automated support for both forward and reverse engineering using UML class diagrams.

III. REVERSE ENGINEERING OF BLUE™

In the initial phase, the system consists of multiple components. Some UML notations describe this state as a class tree hierarchy. Figure 2 shows the tree view of the BLUE™ project. The tree hierarchy is then iterated into a class diagram — one of the 9 standard UML diagrams.

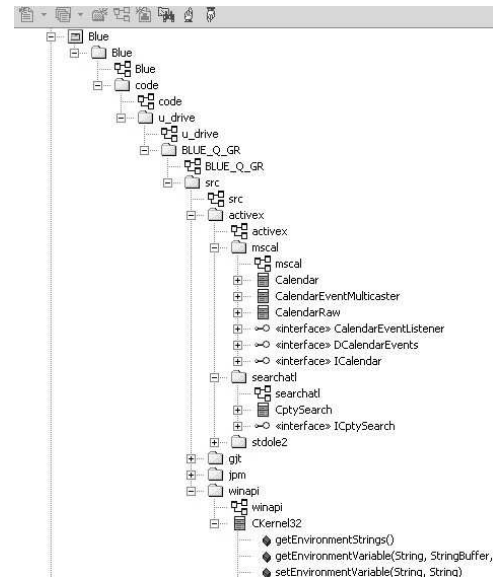


Fig. 2. Tree view of the BLUE project

The reverse engineering of the BLUE application was done using Enterprise Architect™ UML tools. Reverse engineering works in the following manner: The execution path of the original code is given to the CASE tools. Some tools also require the type of the language to which the code is to be converted. When the process starts, each file is imported into the CASE tool's data and converted into its corresponding class diagram.

Some UML tools extract more diagrams allowing the generation of sequence diagrams and transition and collaboration diagrams [9]. Extracting more diagrams allows more code to be generated inside the bodies of methods with a class; this leads, naturally, to a fuller conversion of the code between source and target languages. However, extraction of more than

TABLE I
FORWARD ENGINEERING OF THE ALIB_MONITOR1 CLASS FROM C++ CODE TO C#

<p>The old C++ Blue (alib_monitor1) code</p> <pre>#include "stdafx.h" #include "resource.h" #include <initguid.h> #include "alib_monitor1.h" #include "alib_monitor1_i.c" #include "AlibMonitor1.h" const DWORD dwTimeOut = 5000; // time for EXE to be idle before shutting down const DWORD dwPause = 1000; // time to wait for threads to finish up static DWORD WINAPI MonitorProc(void* pv) { CExeModule* p = (CExeModule*)pv; p->MonitorShutdown(); return 0; }</pre>	<p>The Converted C# Blue (alib_monitor1) code</p> <pre>// Static Model using ExternalClasses; public class Utility { } public class alib_monitor1 { } public const ExternalClasses.DWORD dwTimeOut = new 5000; public const ExternalClasses.DWORD dwPause = new 1000; public static DWORD MonitorProc(void* pv) { } The body does not get converted }</pre>
--	---

one class diagram through reverse-engineering is a complex process. In fact, many of today's CASE tools support only the extraction of one class diagram from the code [10]. FUJABA [14] is a CASE tool that *does* support the extraction of more than one class diagram, but it only supports RTE on JAVA applications.

After importing all the files, a tool is used to determine relations between the classes. The purpose is to determine the relations of each class with all other classes. This is why all class diagrams were generated at the beginning. In the final stage, a class diagram is created for each folder or workspace according to the options stated at the beginning of the process.

The main outcomes of reverse engineering were class diagrams showing the different interactions of the classes in the software. These interactions include relations between different classes like associations, aggregation, and generalization. The class diagram presents a full static design of the system, leading to full understanding of its architecture.

IV. FORWARD ENGINEERING OF BLUE™

Reverse engineering allows us to extract the full analysis model of the system. It provides an object-oriented analysis of the system by autogenerating class diagrams from the original source code. The model is a class diagram with all relations and associations. Even with this, a full conversion into C# is not possible, but it does generate an integrated architectural design for the system. Thus, code generation is not fully complete. In fact, this generation procedure cannot generate the body of the methods within the code. It should also be noted that the extracted class diagram is abstract and does not model the code in detail. It only models relationships between the code, its attributes, and methods. Therefore, language conversion software is used to perform the conversion of the code imbedded within the body of methods in each corresponding class.

Forward engineering is applied after extracting the class diagram. This gives almost unlimited capabilities to generate code into any language that the tool supports. Some tools,

such as Rational Rose™, support more than 20 programming languages. Table I gives a snapshot of a forward engineering of the alib_monitor1 class from C++ code to C# code.

JAVA Language Conversion Assistant™ (JLCA) is an additional tool used to achieve this goal. JLCA™ compares favorably to many of today's UML CASE tools and has additional features, such as flexibility, framework integration, and body conversion. JLCA™ was originally designed by Microsoft™ for the conversion of JAVA and VJ++™ applications into C#—enabling the merging of the applications into the .Net™ framework in the easiest possible way. JLCA™ is very powerful in code conversion and it successfully converted, in full, the 1505 JAVA files of BLUE™.

Due to differences in architecture between the JAVA and C# environments, some conversions still must be made manually. Upgrade issues are concerned with unconverted fields. Upgrades-to-do are concerned with logical differences, or changes in certain values as a result of the conversion process; for example, the file was successfully converted, but the naming, or return value was changed. This could easily affect the logic of the code, sometimes causing compilation errors. Upgrade notes give the programmer an indication of what changes were made in the conversion process. Moreover, a report is generated showing each one of 1505 files' conversion statuses. It explains if a file was fully converted or if there were any issues in its conversion. Figure 3 (next page) shows a sample of the report for two files, Assert.java and Box.java, which were converted to Assert.cs and Box.cs. We see that the Assert file was fully converted without any problem, while the Box file was converted with 53 issues.

The generated code is similar to the original code in a number of ways. The comments for example are not changed, so tracing of new code will be easy to programmers of the original code.

In addition, an upgrade report of the whole system is generated. This shows all issues that couldn't be resolved or might have been in the original code, see Figure 4 (next page).

Conversion Report for src.vjpp						
Time of Conversion: 4/18/2005 1:40 PM						
List of Project Files						
New Filename	Original Filename	Status	Errors	Warnings	Total Is	
(Global Issues)			0	7	7	
Assert.cs	Assert.java	Converted	0	0	0	
Box.cs	Box.java	Converted with issues	53	0	53	

Fig. 3. A part of the generated report showing the statuses of two of the converted files

Upgrade report for src.vjpp

Time of upgrade: 4/6/2005 2:12:45 PM
 Upgraded project file: src_MBT_upgrade
 Project type: Console Application
 Startup object: jsm.kba.C2bae

Project was upgraded with 3 issues

List of upgrade issues

Category	Issues	Occurrences
* Unresolved COM references encountered in upgrade		
# Issue	Impact	Occurrences
1 *The type library for class or interface not found	Build error	3
Class or interface	GUID	File affected
ICopySearch	DCAFF7C6-C293-11D2-9CA2-00C04F79F2B6	3
IAMMonitor1	F8E606A-AE74-11D3-8094-00A0248A44E7	
IDataMonitor2	A7464955-B23A-11D3-8094-00A0248A44E7	
* Issues in upgrading prevent file to new format		
# Issue	Impact	Occurrences
1 Source control settings not upgraded to the new project	Minor	3
2 Analysis of source files was not completed due to errors	Minor	1

Fig. 4. Upgrade report showing different issues in the conversion process

Another benefit is that the generated code is located in the same folder names as the original ones—different paths are used to avoid overwriting the original code.

V. COMPARING CASE TOOLS

Lately the number of conversion tools supporting RTE has increased dramatically. These include: Rational Rose™ and Rational XDE™, known for its powerful architecture and visual modeling; Microsoft™ Visio™, a comprehensive tool for building and designing applications; FUJABA (From UML to JAVA And Back Again), a public domain prototype tool that especially supports RTE for UML class and behavioral diagrams to and from JAVA; Multi-language UML modeling solution™; and Enterprise Architect™, a modeling tool that supports 8 of the 9 UML diagrams and also supports RTE of many languages including VB.Net™ and C#.

Even though many researchers have compared UML tools, most of the comparisons were done on the basis of number of classes, number and type of associations, and the way of handling these interfaces [5]. In contrast to the work previously done in this field, the comparisons here will be done on: integration with Visual Studio .Net™, support for round-trip engineering; the capability to generate UML diagrams through reverse engineering, and cost.

A. Integration with Visual Studio .Net™

Most developers spend their time switching between a modeling tool and VS.Net™, so tools try to provide tight coupling with VS.Net™ in a variety of ways. Some of them provide a very high level of integration, such as sharing structures and interfaces, but it is more common for the tools to provide standalone functionality and loose coupling.

Rational Rose™ is not integrated with VS.Net™ and does not support RTE for C# or VB.Net™. However, IBM also produces Rational XDE™ which has very close integration with VS.Net™. It is hosted entirely within the user interface provided by VS.Net™. In addition, it shares the same menu structures, interface, and usage metaphors as the VS.Net™.

Microsoft Visio™ is included within the VS.Net™ package, but is only loosely coupled with it. Menu items can be added to VS.Net™ to activate modeling functionality.

Another tool is Together Software™, originally designed for Borland™ applications. It provides very good integration with VS.Net™ through Together Edition for VS.Net™ that integrates Together Control Center™ with VS.Net™ [13].

In contrast with the previous tools, Enterprise Architect™ isn't very well integrated with VS.Net™.

B. Round-Trip Engineering support

Modeling and coding are separate activities in the software development process. The application is first modeled and then the code is generated. The process of implementing the model is simplified using forward engineering, which automatically generates code directly from the structure specified by the model. A major step has come with the introduction of reverse engineering, which analyzes an existing code base and translates it into UML model elements: classes, attributes, methods, interfaces, and others. Reverse engineering is a popular feature of modeling tools because it enables the modeling of legacy code without requiring a preconstruction of the initial model. A modeling tool that can seamlessly blend forward and reverse engineering is said to support round-trip engineering. This feature enables independent changes to the code and model to be synchronized. A developer can work in whichever domain is more productive and see the results of the changes in the other domain. The degree of support for round-trip engineering for applications developed in different languages varies by tool vendor.

Rational Rose™ supports C++, VC++™, VB6™, JAVA, J2EE/EJB, CORBA™, Ada83, Ada95, Database (including Oracle™, DB2, SQL92, SQL Server, and Sybase™), and Web Applications. For .Net™ applications, IBM has Rational XDE™, which is especially designed to fully support RTE for applications designed in the .Net™ environment.

Microsoft Visio™ also supports a number of languages: C++, VB6™, JAVA, C#, VB.Net™, Database (Oracle™, DB2,

SQL92, SQL Server, and SybaseTM), and DelphiTM. VisioTM comes as a part of the VS.NetTM environment and especially supports .NetTM applications and VS6TM like C++, VC++TM, VB6TM, VB.NetTM, VC.NetTM, and C#.

Enterprise ArchitectTM supports the same languages as Microsoft VisioTM, but is not well integrated with the .NetTM environment. It does, however, have excellent support for round-trip engineering.

TogetherTM is another tool that supports C++, JAVA, C#, VB6TM, VB.NetTM, and CORBA IDLTM. TogetherTM can also support design-based projects independent of any language.

FUJABA is a research tool especially designed for JAVA applications but with much more advanced capabilities than the other tools [14]. This tool is in the public domain and supports round-trip engineering for JAVA applications.

C. Generation of UML Diagrams

UML has nine standard diagrams, which are divided into two types: Static diagrams that capture information about static structure of the software; and dynamic diagrams that give information about dynamic structure of the software. Most CASE tools support creation of both types of diagrams, but only a limited number of tools support generating these diagrams through the RTE process.

Rational RoseTM and Rational XDE support the generation of UML diagrams through the reverse engineering process. Both construct a tree view that contains classes, interfaces, and associations found at the highest level. Methods and variables are nested under owner classes. They also construct the class diagram representation for the extracted information and generate a default layout.

FUJABA works in a little different way. It has the capability of generating code from class diagrams, activity diagrams, state charts, and collaboration diagrams; but, it is specifically designed for JAVA applications. FUJABA has the unique capability of generating code for dynamic diagrams and can reverse engineer the code and generate both static and dynamic views.

Enterprise ArchitectTM supports generation of class diagrams from source code. It can generate class diagram packages, either per directory or per namespace, as well as per file. Also it gives one the option to synchronize classes or overwriting them.

Microsoft VisioTM generates a tree view similar to Rational RoseTM and has the ability to construct class diagrams on demand.

D. Support for Design Patterns

Patterns are used to describe software design structures that can be used over and over again in different systems. They provide general solutions that can be applied to particular situations. Design considerations are used to decide whether a particular pattern is useful and how it could be best implemented. The first important task in reverse engineering of software systems is to understand the system's source code. Design patterns are used in the forward engineering portion

of the RTE process as good solutions to recurring problems and form a common vocabulary among developers. Thus, recognizing the implementation of design patterns in existing software systems helps the reverse engineer to understand the system. There are many design patterns, so tools vary in their support.

Rational RoseTM, for instance, provides 20 of the GOF (Gang of Four) design patterns for JAVA [2]. It also supports pattern combinations to develop new patterns.

FUJABA uses a pattern specification language to recognize design patterns. They are in the process of developing an editor that will allow any FUJABA user to add practical experience to their design patterns.

Enterprise ArchitectTM supports design patterns, but the user needs to create the patterns.

TogetherTM also supports design patterns and integrates them with their JUnit (the open-source testing framework they use).

Microsoft VisioTM doesn't handle design patterns at all.

E. Choosing the best CASE tool for the process

Up to now we have compared the various CASE tools with regard to integration with the .NetTM environment, support for round-trip engineering, and the generation of UML diagrams. An even more important criterion for conversion of BLUETM is support for JAVA and C#, as these are the source and target languages for the conversion process. Another important criterion is suitability for the merging process, since the code using RTE must be merged with that generated using JLCA.

Rational RoseTM, Rational XDETM, FUJABA, Microsoft VisioTM, and Enterprise ArchitectTM all support JAVA as a source language, so the first criterion is not really an issue. The first three, however, do not support forward engineering into C# [5]. Microsoft VisioTM and Enterprise ArchitectTM do support this, so after the initial step our choices are limited to only two CASE tools.

For the merging process, Microsoft Visio generates its own library folder, changing some of the folder names, class and method names, and references. Enterprise ArchitectTM, on the other hand, keeps all of the original names and references. The directory tree is an exact copy of the original. When dealing with over 1500 names, this is important—it makes the job of tracing update issues easier. For this reason we chose Enterprise ArchitectTM for the RTE process.

The RTE process was accomplished as follows: BLUETM, with its more than 1500 JAVA files, was reversed engineered using the Enterprise ArchitectTM UML CASE tool. The result was a class diagram showing the various interactions of the system. Using forward engineering, the class diagram is converted into C# code. However, the classes are all abstract and the methods within the classes have empty bodies. Therefore, the process must be enhanced and completed using specialized tools.

F. Enhancing the RTE process using specialized tools

As we have seen, round-trip engineering has many advantages but still suffers distinct disadvantages in the implemen-

tation phase. These problems need to be solved, since most companies care about analysis and design, implementation, or both. The only way to achieve a full system conversion is by summing the advantages of the reverse and forward engineering processes and discarding their disadvantages. In this way we achieve a fully converted system from design and analysis to implementation and documentation.

Most of the work previously done in round-trip engineering has focused on enhancing the reverse engineering side. These enhancements were aimed at extracting more modules than just static class diagrams from source code. They enabled development of CASE tools like FUJABA to allow the extraction of dynamic diagrams such as sequence and collaboration diagrams. This allows generation of some of the code within method bodies.

There has not, however, been much research done in developing outcomes from the forward engineering process to achieve full system conversion. We will use the outcomes of the specialized tool discussed earlier to finish the RTE process by integrating the JLCA converted method bodies into the empty body of the forward engineered code. The result will be a fully converted system. The design will then be updated by reverse engineering the converted system having this new code. In this way the design will reflect the fully converted system.

G. Conversion of BLUETM using a specialized tool

JAVA Language Conversion Assistant is a specialized tool used to convert code from JAVA to C#. It works strictly from an implementation point-of-view and does not give any consideration to the design phase of a system.

The tool runs entirely from the .NetTM environment. Most of the code is generated during this process, including the method bodies for the classes. JLCA was designed for JAVA applications and supports VJ++TM, used in BLUETM, with some restrictions. The conversion process here is not complete yet because of differences in architecture between C# and VJ++TM. Another source of errors is due to missing and/or corrupted files in the original source code. Hence, there will still be unconverted lines within the code. However, the JLCA tool still greatly enhances the RTE output through the autogeneration of large chunks of code.

H. Updating existing design

The code generated by JLCA was merged with code generated from the RTE process. The result was about a 93% conversion of the original application into C#. This 93% is effective enough to give a new design for the application using a C# platform. Code updates to the design are added using a reverse engineering process. Figure 5 shows the full conversion process.

VI. EFFICIENCY OF THE RTE-JLCA PROCESS

This conversion process was not only performed on the BLUETM application, but also on nine other test applications. The conversion results are given in Table II

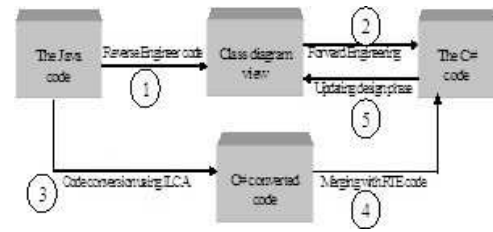


Fig. 5. The full conversion process of the code using RTE

TABLE II
CONVERSION RATES FOR TEN APPLICATIONS

Application Name	Lines of Code	No. of Errors	% of Acc.	% Conv. Files
Chat Server	11500	693	94.0	79.1
Job Scheduler	20000	1760	91.2	69.3
Debugging Tool	28150	2311	91.8	78.1
Network Client FTP	33350	2516	92.5	65.5
Accounting	35750	1926	94.6	76.2
Investment	40880	2107	94.8	74.5
Charting Tool	45000	1877	95.8	76.3
Mail Client	46250	2689	94.2	72.1
Multiclient TCP Server	49150	1838	96.3	76.8
BLUE TM Financial System	100000	7000	93.0	76.5
Overall	410030	24717	94.0	74.4

TABLE III
COMPARISON OF JLCA ALONE TO FULL PROCESS

Application Name	JLCA alone	Full Process
Chat Server	91.3	94.0
Job Scheduler	81.5	91.2
Debugging Tool	76.5	91.8
Network Client FTP	86.2	92.5
Accounting	89.7	94.6
Investment	86.3	94.8
Charting Tool	88.0	95.8
Mail Client	83.9	94.2
Multiclient TCP Server	87.8	96.3
BLUE TM Financial System	84.3	93.0
Overall	87.6	94.0

With few exceptions, the accuracy seems to be proportional to the number of lines of code. The accuracy ranged from 91.2% for the schedule to 96.3% for the TCP server. Had BLUETM not had missing and corrupted files, we believe its accuracy rate would have been much better.

The percentage of fully converted files ranged from 65.5% to 79.1%. There is no clear relationship to the size of the application. It may, however, be correlated with the number of files, but the number of files in each application has not been recorded.

Table III shows accuracy of the full process compared with accuracy achieved using JLCA alone. This improvement

TABLE IV
DISTRIBUTION OF ERRORS FOR FIVE APPLICATIONS

Application Name	0	1	2	3	4	5	6+
Chat Server	79.1	1.4	2.8	2.4	4.3	2.0	8.0
Job Scheduler	69.3	8.7	1.4	2.7	1.4	2.5	14.0
Debugging Tool	78.1	5.0	9.9	3.3	0.6	0.5	2.6
Network Client FTP	65.5	15.0	8.7	0.6	0.8	0.5	8.9
TCP Server	76.8	7.7	2.9	1.4	2.9	0.8	7.5

ranged from 2.7% for the chat server to 15.3% for the debugging tool. Overall, the average improvement was 6.4%. The percentage improvement does not seem related to the size of the application, but may depend upon internal features of the application instead.

After examining and resolving the migration issues that could not be resolved by the JLCA tool, we looked at the distribution of errors in five randomly picked application from these ten. Table IV shows this distribution. Some files had as more than 300 errors! For example, `MainFrame.java` was reported to have 405 errors when converted to `MainFrame.cs`. In looking at the contents of the files, most of the files having few errors (1 or 2) were involved with calculations. In contrast, those with many errors were usually the files containing the code for the graphical user interface.

VII. CONCLUSIONS

RTE has made it possible to produce analysis and design architectural models of BLUETM. The results of conversion show a significant improvement over use of JLCA alone and had an overall efficiency of 93% of full code conversion.

It is not surprising that the best conversion rates were achieved by features common to both the source and target languages, such as calculations, while the worst were achieved by language-specific code, such as the graphical user interface. Thus, the contents of each file should be used to help determine whether a specific file should be converted, and the errors traced and corrected; or whether it would be best to write a file from scratch in the new language.

The presented conversion methodology proves efficiency in of the migration and conversion of large object-oriented applications. The sample case was used to test the conversion of JAVA applications into C#. The approach also provides analysis and design artifacts by providing the design of class diagrams. The class diagrams can be updated to reflect any changes on the system, since it is linked with the code generated through the RTE process.

This approach compares well to other conversion methodologies. The results give an accuracy of about 93% of full code conversion to C#. There are still a few issues to be resolved before 100% accuracy of conversion can be achieved. These include embedding advanced features, functionalities, and interface capabilities to resolve language limitations and conversion constraints. It is desirable to create dynamic conversion procedures in the design phase to generate full code

reflecting all changes in the design without affecting the balance of the entire code. Code generation is widely accepted in the industry due to its clear structure.

The class design generated during the RTE process is regularly updated during the merging process to reflect any design enhancement. Much effort is being done to achieve 100% code conversion. This will require eliminating all types of errors from the original code. The objective is to not only to reflect new changes on the architectural design of the system, but also to minimize compilation errors; allow design enhancements; and provide flexibility, reusability, and ease of maintenance features to the current system.

ACKNOWLEDGEMENT

The authors are grateful to MBRM — MB Risk Management — for both financial and technical support of this research. Special thanks are due to the professional team at MBRM — headed by Gibran Hamed — for valuable guidance and recommendations during the development of this work.

REFERENCES

- [1] H. C. Gall, R. R. Klosch, and R. T. Mittermeir. *Using domain knowledge to improve reverse engineering*. **International Journal of Software Engineering and Knowledge Engineering**, 6(3):477-505. World Scientific Publishing Company, 1996.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, **Design Patterns**, Addison-Wesley, 1997.
- [3] Anders Henriksson and Henrik Larsson, *A Definition of Round-trip Engineering*, Linköping University Sweden, January 2003.
- [4] Thomas Klein, Ulrich A. Nickel, Jörg Niere, and Albert Zundorf. *From UML to JAVA and Back Again*, University of Paderborn, Paderborn, Germany, September 1999.
- [5] Ralf Kollmann, Petri Selonen, Eleni Stroulia, Tarja Systa, and Albert Zundorf, *A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering*, **9th Working Conference on Reverse Engineering**. IEEE, Los Alamitos, 2002.
- [6] Jochen Kuster and Shane Sendall. *Taming Model Round-Trip Engineering*, in **Proceedings of Workshop "Best Practices for Model-Driven Software Development"**, Vancouver, Canada, Oct. 2004.
- [7] MB Risk Management documentation, **BLUE Development Manual**, 2000.
- [8] N. Medvidovic, A. Egyed, and D. S. Rosenblum, *Round-Trip Software Engineering Using UML: From Architecture to Design and Back*, in **Proceedings of the Second International Workshop on Object-Oriented Reengineering**, September 6, 1999.
- [9] Ulrich A. Nickel, Jörg Niere, Jörg P. Wadsack, and Albert Zundorf. *Roundtrip Engineering with FUJABA* in **Proc. of 2nd Workshop on Software-Reengineering (WSR)**, Bad Honnef, Germany, August 2000.
- [10] OMG, editor. **OMG Unified Modeling Language Specification**, Version 1.3. Object Management Group, Inc., Framingham, Mass. <http://www.omg.org>, June 1999.
- [11] Wilhelm Schäfer and Albert Zundorf. *Round-trip engineering with design patterns, UML, Java and C++*. International Conference on Software Engineering, **Proceedings of the 21st international conference on Software engineering**. Pages: 683-684, 1999.
- [12] G. Snelting and F. Tip. *Reengineering class hierarchies using concept analysis* in **Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering**, pages 99-110, Orlando, FL, November 1998.
- [13] Together Soft Corporation, **Together**. Retrieved from the World Wide Web: <http://www.togethersoft.com>, 2001.
- [14] University of Paderborn, Fujaba. Retrieved from the World Wide Web: <http://www.fujaba.de>, 2002.
- [15] M. Weiser, *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*, PhD thesis, University of Michigan, Ann Arbor, 1979.