Migrating from COBOL to Java

A Report from the Field

Harry M. Sneed
ANECON GMBH, Testing Department
Vienna, Austria
Harry.Sneed@T-Online.de

Abstract—**This paper is an industrial report on a project for migrating an airport management system from a Bull mainframe using COBOL as a programming language and IDS as a database system to a distributed UNIX platform using Java and Oracle. The focus here is on the automated language transformation, performed in three phases – reengineering, conversion and refinement. The tools used are COBRedo for reengineering the COBOL code, COB2Java for converting COBOL to Java and JavRedoc for documenting the converted Java code. The paper describes the migration process and the tools used in it and the reviews the current state of the project. Keywords: Migration, Reengineering, Legacy systems, COBOL, Java, Object-oriented transformation.**

Keywords-legacy systems, COBOL, Java, migration, code transformation (key words)

I. PROJECT BACKGROUND

The airport in question has been managing its airport operations – flight scheduling, baggage distribution, air cargo logistics, etc. – for the past 30 years with the aid of a Honeywell-Bull mainframe computer using IDS as a database system and COBOL as a programming language. Over the years a significant amount of legacy code has accumulated. There are 435 main programs, 187 subprograms and 59 database schemas with one million lines of code and 15,486 function points. This code is still evolving and is being maintained by five COBOL programmers, all of whom are now close to retirement.

In the past 10 years a number of java applications have grown up adjacent to the existing COBOL ones. The Java development group now numbers some 25 developers working on a Unix development platform using ORACLE as a relational database. There are now some 240 Java components with 245,000 lines of code and 13,207 function-points. The data from the IDS networked database is copied over to the ORACLE relational database every night, so that the legacy data is already available in the new database system to be used by the new Java applications.

The problem is with the legacy COBOL systems. After the present five programmers go into retirement there will be no one left to maintain them. Besides the BULL mainframe is no longer supported. Within the next two years the machine must be phased out. These circumstances coupled with the fact that a Java development group is already in place makes it appealing to migrate the COBOL systems over into Java. The strategy is obvious, but the implementation is not so obvious. Two attempts have already been made to redevelop the COBOL applications in Java. Both of them failed – one for functional and the other for non-functional reasons. As it turns out, it is much more difficult to replace an existing application than it is to develop one from scratch. Users expect to have all of the old features of a system plus some new ones.

This was the background of a decision to convert the COBOL systems to Java using automated tools. The aim is to preserve the legacy functionality within an object-oriented architecture with a hierarchy of components, classes and methods. The problem is how to translate procedural, GOTO-based, code using redefined data structures with multi dimensional arrays and special COBOL-data types into a class based structure with methods and attributes.

II. PREVIOUS WORK ON THIS PROBLEM

There is no lack of previous work on this problem. In fact, the reengineering community has been dealing with it since object technology emerged in the late 1980's. The first papers on migrating procedural software into object-oriented software appeared in the early 1990's. This author and others were concerned with converting the existing procedural applications into object-oriented ones to take advantage of the new technology [1], [2], [3]. This turned out to be very difficult, so an alternate solution was tried to at least generate an object-oriented documentation from the procedural systems so they could be reimplemented based on that specification [4], [5], [6], [7]. This approach was relatively successful, but users were reluctant to invest so much in reimplementing existing systems. There were also many risks involved in such reimplementation projects. Since most of the work had to be done manually, there was a high probability that functionality would be lost or distorted [8].

The jump from COBOL to C++ or Java appeared for many to be a jump too high, so Object-Cobol came into fashion in the mid 1990s [9]. The updating of COBOL-85 to COBOL-96 appeared to be a logical evolution of that language and it was supported by both IBM and MicroFocus. There was also an international standards committee – CODASYL – involved [10]. This author had already developed a set of tools for converting COBOL-85 to OO-COBOL. It was presented at the

WCRE workshop in 1996 [11]. However, for some reasons object-oriented COBOL never caught on and this approach withered away. By the year 2002 the market was focused on the languages Java and C# as being the programming languages of the future.

Another alternative appeared at this time in the form of wrapping. Legacy code can be encapsulated and embedded in new applications without having to change the language or the structure of the modules. One need only alter their interfaces. Then those modules or procedures can be bound into new systems in Java or C#. This path has been followed by many enterprises looking for a quick and cheap transition to a client / server architecture. A similar approach has been taken for moving legacy code to the web [12]. COBOL modules are now being wrapped as web-services. For this purpose the author developed a set of tools referred to as COBWRAP and applied them in two wrapping projects [13].

The drawback of wrapping is that the legacy COBOL programmers are still needed to maintain the wrapped components and in many parts of the world they are dying out. The older generation of COBOL programmers is phasing out and there is no new generation to replace them. This has lead to a renewed demand to migrate the legacy systems in COBOL and PLI even though the languages are still supported. The first tools to make such a transformation were developed by Triangle technologies in cooperation with the St. Petersburg State University [14]. The Relativity tool has used in several projects but the transformation was never fully automated [15]. The project described here is one of the many that are now going on to complete that automation [16].

III. THE STATE OF THE LEGACY COBOL CODE

The difficulty of transforming source code from one language to another is determined by the quality of the original code. If the original code is well structured and uniform, it is easier to transform than code which is poorly structured and written in an irregular mode. How the code is written depends partly on the editor the programmers use and partly by their thinking habits. If programmers are accustomed to thinking in Assembler they will write all programs as if they were Assembler programs, using built-in constants instead of variables and using GOTO branches to steer the control flow. The control flow of these programs is based on conditional branching. The program tests a condition and branches somewhere. There it tests another condition and branches somewhere else. If on top of this the programmers are using a line editor, the code is highly compressed. There can be two or more statements on every line. This, in turn, encourages the programmers to use as short as possible variable names. The average length of the variable names in this software is 5 characters. These practices have lead to a code which is neither readable nor readily machine processable.

The storage area of the Bull/COBOL programs in this study is divided into five subareas or sections. There is one section – the Record Section – where the IDS database records are read in and out. Another section – the Linkage Section – is for receiving the input messages from the user interface. The Communication Section is for marshalling and dispatching output messages to the user interface. In between these input and output sections is a Work Section where the data is moved for processing. This section also contains the literals and constants used by the program. Finally, these programs also have a Cache Section used to hold copies of the database records for internal processing. This was made for performance purposes. The structure of the legacy Bull/COBOL programs is depicted in Figure 1. The five storage areas containing several hundred data fields grouped into records are accessed by several thousand procedural statements grouped by paragraph which are jumping from one paragraph to another.

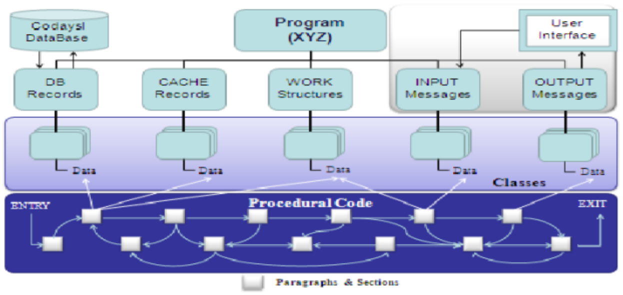


Figure 1:The Architecture of the Bull/COBOL Programs

IV. THE COBOL TO JAVA MIGRATION PROCESS

The transition process from COBOL-74 to Java is of course much too complex to be resolved in a single step. It is a multi-step process consisting of at least eight steps.

1) The first step is to reengineer the COBOL-74 programs into COBOL-85 with several improvements to the code.

2) The second step is a preprocessing of the code to resolve the copies, analyze the data flow and to create the data description tables.

3) The third step is to generate the classes from the Data Division, one for every 01 level data structure.

4) The fourth step is to generate the methods from the Procedure Division, one for every paragraph.

5) The fifth step is to merge the methods into the classes based on the frequency of data reference.

6) The sixth step is to refine the converted Java code detecting and correcting any errors that may have been made in the transformation.

7) The seventh step is to assign the classes to separate source files.

8) The eighth and final step is to create an interface to all of the subprograms called by the COBOL main program.

The major steps of the transformation process are depicted in Figure 2.

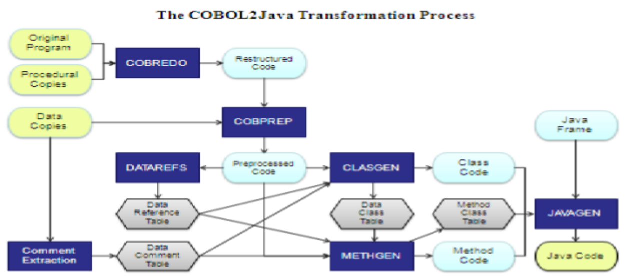


Figure 2: The COBOL2Java transformation process

V. REENGINEERING AS A PREREQUISITE TO TRANSFORMATION

As a prerequisite to the transformation in Java, the first step is to reengineer the COBOL code into a more readable and processable form. This is done by the tool COBRedo in a five step process [17]. In the first step the procedural copies are inserted into the procedure division and the replacement option – copy replacing – satisfied. In the second step the code is reformatted. This entails the cutting up and uniform indentation of lines. Lines with two or more statements are cut up into a line per statement. In the third step the code is converted to standard COBOL-85 with END-IF and END-PERFORM terminators. Obsolete statements such as EXAMINE and NEXT SENTENCE are replaced. In the fourth step the hard-coded data-numeric constants and text literals – are extracted from the procedural code and moved out to tables. This makes the code much more flexible. In the fifth step the COBOL specific data type such as packed, binary and floating are converted to numeric character format, so that all data can be represented as standard ASCII characters. Another problem was that of oversized programs which had to be downsized [18].

The results of the reengineering phase are uniformly formatted semi structured and disinfected COBOL sources which only use character type data. This is a prerequisite to the next transformation, which is to convert the code into Java.

VI. PREPROCESSING OF THE COBOL CODE

After the COBOL sources have been reengineered but before the Java transformation the COBOL code is preprocessed. The preprocessing involves three separate passes of the COBOL source, one of which alters the source and two of which only parse it to produce information required for the transformation.

In the first pass the data copies are inserted in the data division replacing the names and types according to the replacement option. In addition, all elementary data elements with the level 01 or 77 are collected together and placed in a common data structure – the working structure. Since the partition of the COBOL code into classes is based on the COBOL data structures, all data must be assigned to a structure. This generated artificial structure is placed at the end of the working – storage.

In the second pass the data usage is analyzed to create a data reference table. This table includes an entry for every data reference giving the name of the paragraph where the data is used, the data name and the type of reference – whether as argument, result, predicate or parameter. This table is required to record which data is actually used, but also to document the parameters to each JAVA method, since every paragraph becomes a method.

In the third pass, the COBOL data division is parsed to create a data table with the names, types, positions, lengths, values and precisions of all data elements to declare in the program. Here the data elements are all assigned to a data structure, which is equivalent to an object. This table is used for generating the classes. It is also intended to be used as a basis for generating and validating test data. The regression test is aimed at testing the migrated systems against the original data and for that a specification of the original data is required. This data table is stored as a CSV file which may be loaded into an excel table for viewing.

The results of the COBOL preprocessing are then:

- a common data structure for all elementary data items outside of structures,
- a data reference table with all data usages by paragraph and section,
- a data table with a description of all data declared in the COBOL program.

VII. TRANSFORMING COBOL TO JAVA

After having reengineered the COBOL code with the tool COBREDO and preprocessed it with the COBOL data analyzer it is now ready to be converted over into JAVA. The main transformation process involves three steps:

- class generation
- method generation and
- package generation.

A. Class Generation

The template is used to format your paper The first step is to process the data table and the data reference file created by the preprocessor in order to produce a class framework. This framework contains a header, an objects definition, an initialization procedure and a list of those data attributes actually used. The class objects correspond to a COBOL level 1 structure. It is a static array of characters in the length of the original COBOL record. By using a character array to store a singleton object using physical displacements to address the data the sticky problem of redefinitions could be circumvented [19]. For every data element referred to, a set and a get method are generated in the class. The get method extracts the substring starting with the position of the field up to the length of the field and converts it to a standard JAVA data type to be returned to the caller. The set method converts the JAVA data type to a character array and overwrites the existing substring. All data is accessed via the set and get methods. In this respect the criteria of encapsulation is fulfilled. (see Figure 3)

For 88 level data container types are generated with an array of constants. If they are referred to, they check the variable passed against the constants contained and return a boolean true if the parameter matches a constant and a boolean false if it does not.

An initialization method is inserted to supplement the object constructor. It assigns an initial value to each attribute of the class. This is particularly important for the constant values of the COBOL program.

Finally, for all classes which encapsulate a database record, a file, a report, a user interface or a system interface standard access methods are built into the class. The other methods can only access these external objects by invoking their access methods. Typical access methods for a database object are the CRUD methods. Files are accessed via read, write, and rewrite and delete methods. User interfaces are displayed or accepted. System messages are sent or received. Reports are formatted and printed. The same access patterns are used for external objects of the same type. Only the names of the data and the keys are altered.

When generating the class framework, the tool also makes a table of data to class assignments. This is used later by the method generator to recognize which classes are referenced by each method. The class most frequently referenced is the class to which the method will be assigned.

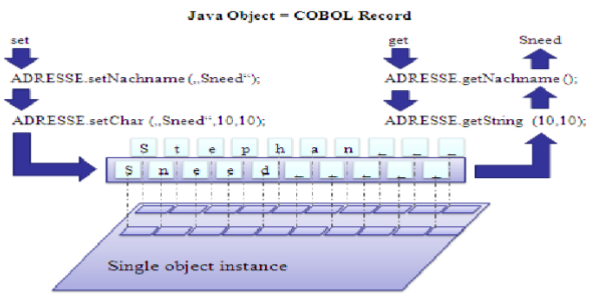
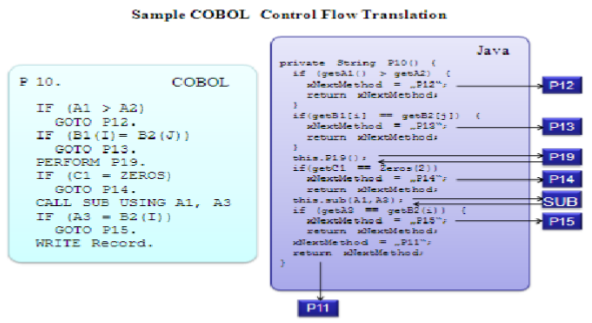

Java Object = COBOL Record
set
get
Sneed
ADRESSE.setNachname („Sneed"");
ADRESSE.getNachname ();
ADRESSE.setChar („Sneed"",10,10);
ADRESSE.getString (10,10);
S t e p h a n
S n e e d
Single object instance

Figure 3: Accessing the Data Attributes

B. *Method Generation*

The second step in transforming COBOL to JAVA is the method generation. Here the procedure division of the COBOL program is parsed to convert the COBOL statements 1:1 over into equivalent JAVA statements. Without having restructured the COBOL code this would not be possible. For the most frequent COBOL statements such as IF, MOVE, COMPUTE, ADD, SUBTRACT, STRING, INSPECT, etc. there are predefined patterns for translating them. The problems come up in handling multi –indexed variables and statements such as UNSTRING and TALLY for which there is no direct equivalent in JAVA. At the moment these statements are marked to be converted manually.

There are two classes of statements which have to be handled in a different way. These are the control statements and the IO- statements. There are three statements for passing control in COBOL – GOTO, PERFORM and CALL. The GOTO causes a branch to a label within the same program with no return. The PERFORM causes a branch to a block of code – section or paragraph – within the program with an automatic return at the end of that block. The CALL is a link to an external module with an automatic return. There is no equivalent to a GOTO statement in JAVA as that would violate all the principles of encapsulation and locality of reference. Therefore, this transfer of control has to be emulated. A label variable – next method – is set as the return value and control returned to the controller. The controller has a recursive function to invoke the next method indicated by the label variable returned from the last method executed. (see Figure 4)



```
P 10.
  IF (A1 > A2)
    GOTO P12.
  IF (B1(I)= B2(J))
    GOTO P13.
  PERFORM P19.
  IF (C1 = ZEROS)
    GOTO P14.
  CALL SUB USING A1, A3
  IF (A3 = B2(I))
    GOTO P15.
  WRITE Record.
```

```
private String P10() {
  if (getA1() > getA2) {
    xNextMethod = „P12";
    return xNextMethod;
  }
  if(getB1[i] == getB2[j]) {
    xNextMethod = „P13";
    return xNextMethod;
  }
  this.P19();
  if(getC1 == Zeros(2))
    xNextMethod = „P14";
    return xNextMethod;
  this.sub(A1,A3);
  if (getA3 == getB2(i)) {
    xNextMethod = „P15";
    return xNextMethod;
  xNextMethod = „P11";
  return xNextMethod;
}
```

Figure 4: Converting the COBOL Control Flow

The PERFORM statement is similar to a JAVA method invocation, only it has neither parameters nor a return value. The return value assigned here is the label variable next method. Thus, if there is a GOTO out of a performed method it will be reverted to the controller to execute that method when control is returned.

The CALL statement has parameters but no return value. The parameters passed are addresses of data fields in the storage of the calling program. For conversion to JAVA the transformer must know whether the parameters are inputs or outputs to the calling programs. The inputs are assigned to the return object. The inputs remain as parameters. If they are both input and output, they are assigned to both.

The external modules called by a COBOL program are simulated by stubs in a service class, where only their interface is available. This interface causes the invocation of a method in a foreign component. The linking of the foreign components is implemented via a remote procedure call using an API. In this way the foreign components may be in a separate address space, i.e. on another computer.

The conversion of input/output operations is a challenge to all migration projects, especially if the legacy software was using same ancient database and teleprocessing monitor as is the case in this project. Here the database was a networked CODASYL database – IDS – which was once the state of the art before relational databases took over. The access operations are coded in the DML language, and include such commands as FIND, GETNEXT, GET, MODIFY and DELETE. Since the database records are assigned to AREAs and connected by pointers, there are also commands like if AREA empty and if POINTER = NULL. Most of these DML operations can be mapped to equivalent SQL commands. In this case the corresponding access method of the database class is invoked. For those DML operations for which there is no SQL equivalent, stubs are generated which have to be filled out manually.

The result of the method generation is a source text in which only the converted methods are listed in the order in which they appeared in the COBOL program. Before each method is a header comment containing all the data attributes used by that method, their usage type and the class to which they belong to. After each method is a footing comment containing the names of all possible successor methods and how they are reached, by GOTO, PERFORM, CALL or by FALL through. In addition, the original comments from the COBOL code are embedded in the methods to which they pertain.

At this moment the Java code still reflects the structure of the COBOL procedure division, but this is only an intermediate state. In the next step the Java methods will be taken out of the method file and assigned to the generated classes on the basis of their data usage. The classes which originally contained only data attributes are supplemented by those methods which process their data most.

C. Component Generation

The third step in the transformation process is to merge the class file with the method file to produce a composite source including all classes and all methods. The challenge here is in assigning methods to classes and qualifying all the method references. From the previous data flow analysis it is known which data is used by that method in what way. A method is assigned to that class to which most of its outputs belong. If it has no outputs but only inputs as is the case with control procedures it is assigned to that class to which most of its inputs belong. If it has neither inputs nor outputs, but is only a control mode, it is assigned to the controller class. In the end every method is assigned to some class.

There is also a special subcontrol class inserted here. It contains a method for every COBOL section and every PERFORM THRU. In both cases a sequence of paragraphs is executed. Thus, a subcontrol method contains a sequence of method invocations, one for every paragraph in the COBOL section or for every paragraph in the PERFORM THRU sequence. After every method is invoked the next method label variable is checked to see if it has not been altered by the last method executed. If so control is returned to the controller for redirection.

It should be pointed out that this last step marks a significant structure break from the original code. In the original program all of the procedural code blocks or paragraphs were together in one procedure division. As pointed out at the beginning many transformation approaches leave them there in what is referred to as a control class. This is not the case here. Here the procedural code blocks are distributed among the classes based on their data usage. As such the structure of the final Java package has no similarity to that of the original COBOL package. It is truly object-oriented. This may or may not be a good thing. It is disputable whether it is beneficial to impose an object structure on code designed to be in a procedural one.

VIII. POSTPROCESSING OF THE JAVA SOURCE

After the JAVA components have been generated there is still some work to be done. This work is accomplished by the postprocessor. For one, the source code of a component has to be split up into a source file for each – class and the framework classes added. These are the control and the service classes. There is also a skeleton class for the database accesses. This class remains to be finished by the developers depending on the exact database version to be used. Finally a test class is added which allows test data in the form of ASCII character strings to be submitted either as user interface maps or as database records. In this way the old environment – the TP – monitor and the database system – can be simulated. The same test class also writes out the map contents returned to the user and the database records inserted or modified. With the help of the data table created from the COBOL data structures these outputs can be interpreted and validated.

The final result of the COBOL to JAVA transformation is a library for each subsystem with a package for each former COBOL program plus a framework. In the packages there is a source file for each JAVA class taken from the COBOL code as well as a source file for the five storage classes – Database, Cache, Input, Output and Work. The packages are compiled and the byte code libraries prepared for execution. (see Figure 5)

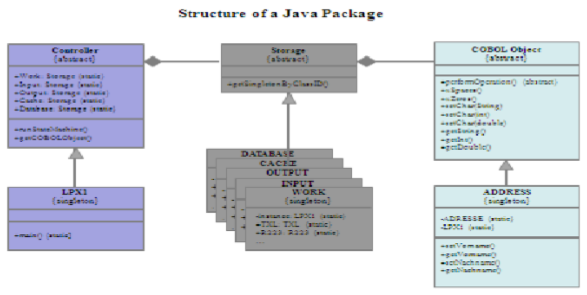


Figure 5: The architecture of the Java Components

IX. SUMMARY OF THE MIGRATION PROJECT

Since the submission of this paper the first migration project has been completed. Altogether 95 COBOL online programs and 41 subprograms with 140 copy members were converted, giving 104 Java packages. The actual project lasted 3 calendar months and required 28 person months of effort. Before this, another 10 person months went into the development of the COB2Java tool including two months for the Java framework made by the author's son. At the end of the project all 104 Java packages could be compiled and four were functionally tested. The size of the resulting Java code amounted to three times as much as the original COBOL code. This can be explained by the following facts.

In generating the Java code, each COBOL procedural statement is mapped to one or more Java statements. Thus, alone from the procedural code there is an increase of circa 50% in the number of statements. For instance the GOTO is translated to an Assign and a Return. On the data side each data declaration is converted to a set and a get method on the character array of the static object. Each such method contains three statements – an Entry, an Assign and a Return – so there are three Java statements for each COBOL declaration. This expansion is partly compensated by fact that only the data items used are accessed, but including the initial assignment there are still 6 to 8 Java statements for every referenced COBOL data field. On top of that the Java framework code of some 600 statements is copied into each package, to make it independently testable. A small COBOL program with 1561 statements resulted in a Java package with 6034 statements. A large COBOL program with 13,887 statements resulted in a Java package with 33,280 statements.

Altogether the converted COBOL code included 257,825 statements and 304,791 uncommented lines of code with 12,773 function-points. The generated Java code included 814,016 statements, 830,508 uncommented lines of code with 12,794 function-points. This amounts to a code size increase of 272 %.

There would be two ways to reduce the Java code size. One is by optimizing the code transformation, the other by eliminating the redundant COBOL code before the transformation to Java. This was partially done with the one largest COBOL program reducing its number of statements by more than 40%. However, this reengineering work required an extra 3 person weeks. Optimizing the code transformation, might save at most 20% of the size increase. Removing redundant COBOL code would save up to 50% of the size increase. Over 50% of the COBOL code was redundant since the programmers simply copied sections of procedural code from one program to another. The transformation to Java only magnified this redundancy. This fact reinforces the insight that legacy code should first be reengineered before migrating it. If the legacy code is of a poor quality, as was the case here, there should be two projects – a reengineering project, followed by a migration project. The problem lies in selling this extra cost to management. Here it was not possible.

At the beginning of the migration project there was still a need for manual corrections to the automatically generated code. This was due to omissions and errors in the code transformer. By the end of the project almost all of those errors and omissions had been corrected and those that were not, were corrected automatically. Thus, in the final transformation there was no need for manual intervention. The entire code body could be transformed fully automatically with a day. This contradicts the assumption that some manual adjustment will always be required. Any correction that a human translator can make can also be programmed into the tool.

Of the 95 main programs converted all were compiled but only four were tested. This was because testing was not part of the contract. The customer only wanted compiled and documented components which could be separately tested. This is what he got. The test would have required a substantial effort requiring between 3 and 20 tester days of effort depending on the component size and complexity. Even if it could be fully automated, the test would still require some 1 to 4 tester days per component. So that adds another 12 person months to the migration costs. Testing is definitely a problem which has to be resolved in order to make migrations feasible.

X. Generality of the Approach

The 8 step approach to transforming COBOL to Java code was designed for this particular project but with reuse in future projects in mind. The algorithm for generating classes from the COBOL record structures is generally applicable. The same is true for the transformation of the procedural statements. There is a separate plug-in for every statement type. The constraint to generality lies in the framework. COBOL systems run in different environments, e.g. CICS, IMS, IDMS, AS400, Bull/IDS, etc. The environment determines how the COBOL programs are structured and how they communicate with their environment. Consequently, there has to be a different Java framework for each environment. The CICS framework will be quite different than the one made here, which was customized to the Bull/IDS environment.

This implies that before any new project, a new framework has to be implemented to accommodate the target environment. In this case, the implementation of the framework required two person months. However, the person or persons making the framework need to have both good knowledge of the legacy environment as well as good knowledge of Java.

The first line of reengineering and transformation modules are implemented in MicroFocus Object-Cobol and embedded in a Borland Delphi framework. The second line of tools for completing the transformation and preparing the Java classes for compilation are written in Object-Pascal. The code reengineering and transformation take place on a PC-Workstation under MS-Windows. The compilation and documentation is made with Eclipse.

XI. CONCLUSION AND PLANS FOR FUTURE

In this industrial report a set of tools have been described which are currently being used to transform a million lines of COBOL code into JAVA. Over 200 thousand lines have already been converted to an object – oriented solution. Whether the converted JAVA code can be readily maintained remains to be seen. It is still disputed whether or not object-orientation really improves maintainability. The complexity may be only shifted from the module level to the system level [21]. In any case the quality of the code has been somewhat improved as witnessed by the code quality metrics. The overall code quality index of the legacy COBOL code was

0.392. That of the generated Java code is 0.633. In addition, the complexity of the code has been somewhat decreased. The overall complexity index of the COBOL code was 0.546, whereas that of the Java code is 0.511. It is now possible to convert the rest of the COBOL code within a year. The decision has been made not to use the entire Java code as it is. Instead, only portions of it will be built into the new applications now under development.

The plan for the future of this particular project is to convert the remaining packages. There are still some

- 300 main programs
- 90 sub programs and
- 50 copy members

to be migrated. For these it is planned to refine the reengineering tools so as to remove some of the redundancy and to improve the structure of the original code. Many of the GOTO branches can be eliminated. Many of the code clones can be recognized and factored out into subroutines. There are unlimited possibilities for improving the generated code, but the main problem, that of renaming the variables and procedural labels will remain unresolved, since this can only be solved by the user himself.

Since testing remains the major cost driver in migration projects, it is also planned to generate a test frame from the code which will feed the transformed JAVA components with artificial test data aimed at exercising the main paths through the component. This should significantly reduce the effort required to test the final system since those errors which can be located at the unit level will be uncovered there. For validation, it is planned to compare the actual test results with the output of the old COBOL programs. For testing a separate data description table is created as a CSV file. This table has already been used to test the first four programs.

Another test support measure will be to insert probes into every Java method. These probes will record how often the method is executed and by what test case. In this way it is possible to trace the test cases thru the code. The author already has a tool TestDoc to record test coverage and to trace test paths thru Java components in general. For this tool the instrumentation of the code is done separately. Here it can be built into the code generator.

Ensuring functional equivalence of the newly generated JAVA components with the original COBOL programs remains as the greatest barrier to making an economical and quick migration. Overcoming this barrier remains as a challenge to the software reengineering community [20].

As far as the tools are concerned, the author has started to develop a similar solution for the PL/I language. There are still many legacy PL/I applications around especially in Vienna where the language originated. Once it is finished the author will be able to offer both a transformation and a wrapping alternative to IBM mainframe users with either COBOL or PL/I.

References

[1] Jacobson, I.: “Re-engineering of old systems to an object-oriented architecture”, Proc. Of OOPSLA-91, ACM Press, New Orleans, 1991, p. 340

[2] Sneed, H.: “Migration of procedurally oriented COBOL programs in an Object-oriented Architecture” Proc. of 8th ICSM-1992, IEEE Computer Society Press, Orlando, Nov. 1992, p. 105

[3] Tomic, M.: “A Possible Approach to OO-Reengineering of COBOL Programs”, ACM Software Eng. Notes, Vol. 19, No. 2, April 1994, p. 29

[4] Gall, H., Klösch, R.: “Finding Objects in Procedural Programs – An alternative Approach”, Proc. of 2nd WCRE, IEEE Computer Society Press, Toronto, July 1995, p. 208

[5] Sneed, H., Nyary, E.: “Extracting Object-oriented Specifications from Procedurally oriented Programs”, Proc. of 2nd WCRE, IEEE Computer Society Press, Toronto, July 1995, p. 217

[6] Pidaparthi, S., Luker, P., Zedan, H.: “Conceptual Foundations for the Design Transformation of Procedural Software to object-oriented Architecture” in Proc. of 6th IWPC, IEEE Computer Society Press, Ischia, June 1998, p. 162

[7] Kontogiannis, K., Patil, P.: “Evidence driven Object identification in Procedural Systems” Proc. of STEP-99, IEEE Computer Society Press, Pittsburgh, Sept. 1999, p. 12

[8] Sneed, H.: “Risks involved in Reengineering Projects”, Proc. of 3rd WCRE, IEEE Computer Society Press, Atlanta, 1999, p. 204

[9] Penteado, R., Masiero, P., do Prado, A.F., Braga, R.: “Reengineering of Legacy Systems based on Transformation using the object-oriented Paradigm” Proc. of 5th WCRE, IEEE Computer Society Press, Honolulu, Oct. 1998, p. 144

[10] Coyle, F.: “OO-COBOL and Legacy Migration” COBOL-Report, Vol. 1, No. 2, Dallas, Sept. 1996, p. 6

[11] Sneed, H.: “Object-oriented COBOL Recycling”, Proc. of 3rd WCRE, IEEE Computer Society Press, Monterey, Nov. 1996, p. 169

[12] [12] Sneed, H.: “Transforming procedural Program Structures to object-oriented Class Structures”, Proc. Of ICSM-2002, IEEE Computer Society Press, Montreal, 2002, p. 286

[13] Sneed, H.: “Wrapping Legacy COBOL Programs behind an XML Interface”, Proc. Of WCRE-2001, IEEE Computer Society Press, Stuttgart, Oct. 2001, p. 189

[14] Terekhof, A.: “Automating Language Conversion – A Case Study”, Proc. of ICSM-2001, IEEE Computer Society Press, Toronto, Oct. 2001, p. 654

[15] Mossienko, M.: “Automated COBOL to Java Recycling”, Proc. of 7th CSMR, IEEE Computer Society Press, Benevento, March, 2003, p. 40

[16] Martin, J., Mueller, H.: “C to Java Migration Experience”, Proc. of 6th CSMR, IEEE Computer Society Press, Budapest, March, 2002, p. 143

[17] Sneed, H.: “Architecture and Functions of a commercial Software Reengineering Workbench”, Proc. of 2nd CSMR, IEEE Computer Society Press, Florence, March, 1998, p. 2

[18] Sneed, H./Nyary, E.: Downsizing Large Application Programs”, Journal of Software Maintenance, Vol. 6, No. 5, Sept. 1994, p. 235

[19] Ceccato, M., Dean, T., Tonella, P., Marchignoli, D. “Migrating legacy data structures based on variable overlay to Java”, Journal of Software Maintenance and Evolution, Vol. 22, No. 3, April 2010, p. 211

[20] Seacord, R., Plakosh, D., Lewis, G.: Modernizing Legacy Systems, Addison-Wesley, Boston, 2003, p. 87

[21] Eierman, M. / Dishaw, M.: “The process of software maintenance–a comparison of object-oriented and third generation development languages”, Journal of Software Maintenance and Evolution, Vol. 19, No. 1, Jan. 2007, p. 33